Package ‘tfprobability’

May 19, 2020

Title Interface to 'TensorFlow Probability'

Version 0.10.0.0

Description Interface to 'TensorFlow Probability', a 'Python' library built on 'TensorFlow' that makes it easy to combine probabilistic models and deep learning on modern hardware ('TPU', 'GPU'). 'TensorFlow Probability' includes a wide selection of probability distributions and bijectors, probabilistic layers, variational inference, Markov chain Monte Carlo, and optimizers such as Nelder-Mead, BFGS, and SGLD.

License Apache License (>= 2.0)

URL https://github.com/rstudio/tfprobability

BugReports https://github.com/rstudio/tfprobability/issues

SystemRequirements TensorFlow Probability
(https://www.tensorflow.org/probability)

Encoding UTF-8

LazyData true

RoxygenNote 7.1.0

Imports tensorflow (>= 2.1.0), reticulate, keras, magrittr

Suggests tfdatasets, testthat (>= 2.1.0), knitr

VignetteBuilder knitr

NeedsCompilation no

Author Sigrid Keydana [aut, cre],
Daniel Falbel [ctb],
Kevin Kuo [ctb] (<https://orcid.org/0000-0001-7803-7901>),
RStudio [cph]

Maintainer Sigrid Keydana <sigrid@rstudio.com>

Repository CRAN

Date/Publication 2020-05-19 20:00:02 UTC
R topics documented:

glm_families ..................................................  8
glm_fit .......................................................  8
glm_fit.tensorflow.tensor ..................................  9
glm_fit_one_step ...........................................  11
glm_fit_one_step.tensorflow.tensor .....................  11
initializer_blockwise .....................................  13
install_tfprobability ......................................  13
layer_autoregressive ......................................  14
layer_autoregressive_transform ..........................  16
layer_categorical_mixture_of_one_hot_categorical ..  17
layer_conv_1d_flipout ....................................  18
layer_conv_1d_reparameterization .......................  21
layer_conv_2d_flipout ....................................  23
layer_conv_2d_reparameterization .......................  26
layer_conv_3d_flipout ....................................  29
layer_conv_3d_reparameterization .......................  32
layer_dense_flipout ......................................  34
layer_dense_local_reparameterization ..................  37
layer_dense_reparameterization .........................  39
layer_dense_variational ..................................  41
layer_distribution_lambda ................................  43
layer_independent_bernoulli .............................  44
layer_independent_logistic ..............................  45
layer_independent_normal ................................  46
layer_independent_poisson ................................  47
layer_kl_divergence_add_loss ............................  48
layer_kl_divergence_regularizer .........................  49
layer_mixture_logistic ....................................  50
layer_mixture_normal .....................................  51
layer_mixture_same_family ...............................  52
layer_multivariate_normal_tril .........................  54
layer_one_hot_categorical ................................ 55
layer_variable ............................................  56
layer_variational_gaussian_process ....................  57
mcmc_dual_averaging_step_size_adaptation .............  58
mcmc_effective_sample_size .............................  61
mcmc_hamiltonian_monte_carlo ...........................  62
mcmc_metropolis_adjusted_langevin_algorithm ..........  64
mcmc_metropolis_hastings ................................ 65
mcmc_no_u_turn_sampler ..................................  67
mcmc_potential_scale_reduction .........................  70
mcmc_random_walk_metropolis ............................  71
mcmc_replica_exchange_mc ................................ 73
mcmc_sample_annealed_importance_chain ...............  74
mcmc_sample_chain .......................................  76
mcmc_sample_halton_sequence ............................  78
mcmc_simple_step_size_adaptation ........................................ 80
mcmc_slice_sampler ......................................................... 83
mcmc_transformed_transition_kernel .................................... 85
mcmc_uncalibrated_hamiltonian_monte_carlo ......................... 86
mcmc_uncalibrated_langevin .............................................. 87
mcmc_uncalibrated_random_walk ....................................... 88
params_size_categorical_mixture_of_one_hot_categorical ......... 89
params_size_independent_bernoulli ................................... 90
params_size_independent_logistic .................................... 91
params_size_independent_normal ...................................... 91
params_size_independent_poisson .................................... 92
params_size_mixture_logistic .......................................... 92
params_size_mixture_normal .......................................... 93
params_size_mixture_same_family ................................... 93
params_size_multivariate_normal_tri_l ............................... 94
params_size_one_hot_categorical .................................... 94
sts_additive_state_space_model ..................................... 95
sts_autoregressive ........................................................ 98
sts_autoregressive_state_space_model ................................ 99
sts_build_factored_surrogate_posterior .............................. 102
sts_build_factored_variational_loss .................................. 103
sts_constrained_seasonal_state_space_model ...................... 104
sts_decompose_by_component .......................................... 107
sts_decompose_forecast_by_component ................................ 108
sts_dynamic_linear_regression ....................................... 109
sts_dynamic_linear_regression_state_space_model ............... 110
sts_fit_with_hmc .......................................................... 112
sts_forecast ................................................................. 115
sts_linear_regression .................................................... 117
sts_local_level ............................................................. 118
sts_local_level_state_space_model .................................. 119
sts_local_linear_trend .................................................... 121
sts_local_linear_trend_state_space_model ......................... 122
sts_one_step_predictive .................................................. 124
sts_sample_uniform_initial_state ..................................... 125
sts_seasonal ................................................................. 126
sts_semi_local_linear_trend ............................................. 128
sts_semi_local_linear_trend_state_space_model ................... 130
sts_smooth_seasonal ........................................................ 132
sts_smooth_seasonal_state_space_model .............................. 134
sts_sparse_linear_regression .......................................... 136
sts_sum ......................................................................... 139
tfb_absolute_value ........................................................ 141
tfb_affine ................................................................. 142
tfb_affine_linear_operator ............................................. 143
tfb_affine_scalar .......................................................... 145
tfb_batch_normalization .................................................. 146

R topics documented:

3
R topics documented:

- `tfb_blockwise` .................................................. 149
- `tfb_chain` .................................................. 150
- `tfb_chol_legendre_outer_product` ................................ 151
- `tfb_chol_to_inv_chol_legendre` ................................. 152
- `tfb_correlation_chol_legendre` .................................. 153
- `tfb_cumsum` .................................................. 155
- `tfb_discrete_cosine_transform` .................................. 156
- `tfb_exp` ................................................... 157
- `tfb_expm1` .................................................. 158
- `tfb_ffjord` .................................................. 159
- `tfb_fill_scale_tri_l` ........................................... 162
- `tfb_fill_triangular` .................................. 163
- `tfb_forward` .................................................. 164
- `tfb_forward_log_det_jacobian` .................................. 165
- `tfb_gumbel` .................................................. 166
- `tfb_gumbel_cdf` ........................................... 167
- `tfb_identity` .................................................. 168
- `tfb_inline` .................................................. 169
- `tfb_inverse` .................................................. 170
- `tfb_inverse_log_det_jacobian` .................................. 171
- `tfb_invert` .................................................. 172
- `tfb_iterated_sigmoid_centered` .................................. 173
- `tfb_kumaraswamy` ........................................ 174
- `tfb_kumaraswamy_cdf` .................................. 175
- `tfb_lambert_w_tail` ......................................... 176
- `tfb_masked_autoregressive_default_template` ................ 177
- `tfb_masked_autoregressive_flow` ............................. 179
- `tfb_masked_dense` ........................................... 182
- `tfb_matrix_inverse_tri_l` .................................. 183
- `tfb_matvec_lu` .................................................. 184
- `tfb_normal_cdf` ........................................... 185
- `tfb_ordered` .................................................. 186
- `tfb_pad` ................................................... 187
- `tfb_permute` .................................................. 188
- `tfb_power_transform` ........................................... 189
- `tfb_rational_quadratic_spline` .................................. 190
- `tfb_real_nvp` .................................................. 192
- `tfb_real_nvp_default_template` .................................. 194
- `tfb_reciprocal` .................................................. 196
- `tfb_reshape` .................................................. 197
- `tfb_scale` ................................................... 198
- `tfb_scale_matvec_diag` ................................ 199
- `tfb_scale_matvec_linear_operator` .................................. 200
- `tfb_scale_matvec_lu` .................................. 201
- `tfb_scale_matvec_tri_l` .................................. 202
- `tfb_scale_tri_l` .................................................. 203
- `tfb_shift` ................................................... 205
- `tfb_sigmoid` .................................................. 206
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>tfb_sinh_arcsinh</td>
<td>207</td>
</tr>
<tr>
<td>tfb_softmax_centered</td>
<td>208</td>
</tr>
<tr>
<td>tfb_softplus</td>
<td>209</td>
</tr>
<tr>
<td>tfb.softsign</td>
<td>211</td>
</tr>
<tr>
<td>tfb_square</td>
<td>212</td>
</tr>
<tr>
<td>tfb_tanh</td>
<td>213</td>
</tr>
<tr>
<td>tfb_transform_diagonal</td>
<td>214</td>
</tr>
<tr>
<td>tfbTranspose</td>
<td>215</td>
</tr>
<tr>
<td>tfb_weibull</td>
<td>216</td>
</tr>
<tr>
<td>tfb_weibull_cdf</td>
<td>217</td>
</tr>
<tr>
<td>tfd_autoregressive</td>
<td>218</td>
</tr>
<tr>
<td>tfd_batch_reshape</td>
<td>221</td>
</tr>
<tr>
<td>tfd_bernoulli</td>
<td>222</td>
</tr>
<tr>
<td>tfd_beta</td>
<td>223</td>
</tr>
<tr>
<td>tfd_beta_binomial</td>
<td>225</td>
</tr>
<tr>
<td>tfd_binomial</td>
<td>227</td>
</tr>
<tr>
<td>tfd_blockwise</td>
<td>229</td>
</tr>
<tr>
<td>tfd_categorical</td>
<td>230</td>
</tr>
<tr>
<td>tfd_cauchy</td>
<td>232</td>
</tr>
<tr>
<td>tfd_cdf</td>
<td>233</td>
</tr>
<tr>
<td>tfd_chi</td>
<td>234</td>
</tr>
<tr>
<td>tfd_chi2</td>
<td>236</td>
</tr>
<tr>
<td>tfd_cholesky_lkj</td>
<td>237</td>
</tr>
<tr>
<td>tfd_covariance</td>
<td>239</td>
</tr>
<tr>
<td>tfd_cross_entropy</td>
<td>240</td>
</tr>
<tr>
<td>tfd_deterministic</td>
<td>241</td>
</tr>
<tr>
<td>tfd_dirichlet</td>
<td>242</td>
</tr>
<tr>
<td>tfd_dirichlet_multinomial</td>
<td>244</td>
</tr>
<tr>
<td>tfd_doublesided_maxwell</td>
<td>246</td>
</tr>
<tr>
<td>tfd_empirical</td>
<td>248</td>
</tr>
<tr>
<td>tfd_entropy</td>
<td>249</td>
</tr>
<tr>
<td>tfd_exponential</td>
<td>250</td>
</tr>
<tr>
<td>tfd_exp_relaxed_one_hot_categorical</td>
<td>252</td>
</tr>
<tr>
<td>tfd_finite_discrete</td>
<td>253</td>
</tr>
<tr>
<td>tfd_gamma</td>
<td>254</td>
</tr>
<tr>
<td>tfd_gamma_gamma</td>
<td>256</td>
</tr>
<tr>
<td>tfd_gaussian_process</td>
<td>258</td>
</tr>
<tr>
<td>tfd_gaussian_process_regression_model</td>
<td>261</td>
</tr>
<tr>
<td>tfd_generalized_pareto</td>
<td>264</td>
</tr>
<tr>
<td>tfd_geometric</td>
<td>266</td>
</tr>
<tr>
<td>tfd_gumbel</td>
<td>267</td>
</tr>
<tr>
<td>tfd_half_cauchy</td>
<td>269</td>
</tr>
<tr>
<td>tfd_half_normal</td>
<td>270</td>
</tr>
<tr>
<td>tfd_hidden_markov_model</td>
<td>272</td>
</tr>
<tr>
<td>tfd_horseshoe</td>
<td>274</td>
</tr>
<tr>
<td>tfd_independent</td>
<td>275</td>
</tr>
<tr>
<td>tfd_inverse_gamma</td>
<td>277</td>
</tr>
<tr>
<td>tfd_inverse_gaussian</td>
<td>279</td>
</tr>
</tbody>
</table>
R topics documented:

tfd_joint_distribution_named ........................................... 281
tfd_joint_distribution_sequential ..................................... 283
tfd_kl_divergence ............................................................ 284
tfd_kumaraswamy ............................................................. 285
tfd_laplace ................................................................. 287
tfd_linear_gaussian_state_space_model ................................ 288
tfd_lkj ...................................................................... 291
tfd_logistic ................................................................. 292
tfd_logit_normal ............................................................ 294
tfd_log_cdf ................................................................. 295
tfd_log_normal .............................................................. 296
tfd_log_prob ................................................................. 297
tfd_log_survival_function ................................................ 298
tfd_mean ................................................................. 299
tfd_mixture ............................................................... 300
tfd_mixture_same_family .................................................. 301
tfd_mode ................................................................. 303
tfd_multinomial ............................................................. 304
tfd_multivariate_normal_diag ............................................. 306
tfd_multivariate_normal_diag_plus_low_rank ......................... 308
tfd_multivariate_normal_full_covariance ................................ 310
tfd_multivariate_normal_linear_operator .............................. 312
tfd_multivariate_normal_tri_l ........................................... 314
tfd_multivariate_student_t_linear_operator ........................... 316
tfd_negative_binomial ....................................................... 318
tfd_normal ............................................................... 320
tfd_one_hot_categorical ................................................... 321
tfd_pareto ................................................................. 323
tfd_pert ................................................................. 325
tfd_pixel_cnn .............................................................. 326
tfd_plackett_luce .......................................................... 328
tfd_poisson ................................................................. 330
tfd_poisson_log_normal_quadrature_compound .......................... 332
tfd_prob ................................................................. 334
tfd_probit_bernoulli ......................................................... 335
tfd_quantile .............................................................. 336
tfd_quantized ............................................................. 337
tfd_relaxed_bernoulli ....................................................... 339
tfd_relaxed_one_hot_categorical ......................................... 341
tfd_sample ................................................................. 343
tfd_sample_distribution ..................................................... 343
tfd_sinh_arcsinh ............................................................ 345
tfd_stddev ............................................................... 347
tfd_student_t .............................................................. 348
tfd_student_t_process ....................................................... 350
tfd_survival_function ...................................................... 353
tfd_transformed_distribution .............................................. 353
tfd_triangular ............................................................. 355
R topics documented:

tfd_truncated_normal ............................................. 357

tfd_uniform .......................................................... 359

tfd_variance .......................................................... 360

tfd_variational_gaussian_process ................................. 361

tfd_vector_deterministic ........................................... 366

tfd_vector_diffeomixture ........................................... 367

tfd_vector_exponential_diag ...................................... 370

tfd_vector_exponential_linear_operator .......................... 372

tfd_vector_laplace_diag ............................................ 374

tfd_vector_laplace_linear_operator ............................... 376

tfd_vector_sinh_arcsinh_diag .................................... 378

tfd_von_mises .......................................................... 381

tfd_von_mises_fisher ................................................ 383

tfd_wishart ............................................................. 385

tfd_wishart_linear_operator ....................................... 387

tfd_wishart_tri_l ...................................................... 389

tfd_zipf ................................................................. 391

tfp ....................................................................... 392

tfp_version ........................................................... 393

vi_amari_alpha ....................................................... 393

vi_arithmetic_geometric ........................................... 394

vi_chisquare .......................................................... 395

vi_csiszar_vimco ..................................................... 396

vi_dual_csiszar_function .......................................... 398

vi_fit_surrogate_posterior ....................................... 399

vi_jeffreys ............................................................ 401

vi_jensen_shannon .................................................. 402

vi_kl_forward ........................................................ 403

vi_kl_reverse ........................................................ 404

vi_log1p_abs .......................................................... 405

vi_modified_gan ...................................................... 406

vi_monte_carlo_variational_loss .................................. 407

vi_pearson ............................................................. 409

vi_squared_hellinger ............................................... 410

vi_symmetrized_csiszar_function .................................. 411

vi_total_variation ................................................... 412

vi_triangular .......................................................... 413

vi_t_power .............................................................. 413

Index 415
A list of models that can be used as the model argument in `glm_fit()`:

**Details**

- Bernoulli: Bernoulli(probs=mean) where mean = sigmoid(matmul(X, weights))
- BernoulliNormalCDF: Bernoulli(probs=mean) where mean = Normal(0, 1).cdf(matmul(X, weights))
- GammaExp: Gamma(concentration=1, rate=1 / mean) where mean = exp(matmul(X, weights))
- GammaSoftplus: Gamma(concentration=1, rate=1 / mean) where mean = softplus(matmul(X, weights))
- LogNormal: LogNormal(loc=log(mean) - log(2) / 2, scale=sqrt(log(2))) where mean = exp(matmul(X, weights)).
- LogNormalSoftplus: LogNormal(loc=log(mean) - log(2) / 2, scale=sqrt(log(2))) where mean = softplus(matmul(X, weights))
- Normal: Normal(loc=mean, scale=1) where mean = matmul(X, weights).
- NormalReciprocal: Normal(loc=mean, scale=1) where mean = 1 / matmul(X, weights)
- Poisson: Poisson(rate=mean) where mean = exp(matmul(X, weights)).
- PoissonSoftplus: Poisson(rate=mean) where mean = softplus(matmul(X, weights)).

**Value**

list of models that can be used as the model argument in `glm_fit()`

**See Also**

Other `glm_fit`: `glm_fit.tensorflow.tensor()`, `glm_fit_one_step.tensorflow.tensor()`

---

## glm_fit

**Runs multiple Fisher scoring steps**

**Description**

Runs multiple Fisher scoring steps

**Usage**

`glm_fit(x, ...)`
glm_fit.tensorflow.tensor

Arguments

x  float-like, matrix-shaped Tensor where each row represents a sample's features.
...  other arguments passed to specific methods.

Value

A glm_fit object with parameter estimates, number of iterations, etc.

See Also

glm_fit.tensorflow.tensor()

Description

Runs multiple Fisher scoring steps

Usage

```r
## S3 method for class 'tensorflow.tensor'
glm_fit(
x, 
response, 
model, 
model_coefficients_start = NULL, 
predicted_linear_response_start = NULL, 
l2_regularizer = NULL, 
dispersion = NULL, 
offset = NULL, 
convergence_criteria_fn = NULL, 
learning_rate = NULL, 
fast_unsafe_numerics = TRUE, 
maximum_iterations = NULL, 
name = NULL, 
... 
)
```

Arguments

x  float-like, matrix-shaped Tensor where each row represents a sample’s features.
response  vector-shaped Tensor where each element represents a sample’s observed response (to the corresponding row of features). Must have same dtype as x.
model
a string naming the model (see glm_families) or a tfp$glm$ExponentialFamily-like instance which implicitly characterizes a negative log-likelihood loss by specifying the distribution’s mean, gradient_mean, and variance.

model_coefficients_start
Optional (batch of) vector-shaped Tensor representing the initial model coefficients, one for each column in x. Must have same dtype as model_matrix. Default value: Zeros.

predicted_linear_response_start
Optional Tensor with shape, dtype matching response; represents offset shifted initial linear predictions based on model_coefficients_start. Default value: offset if model_coefficients is NULL, and tf$linalg$matvec(x, model_coefficients_start) + offset otherwise.

l2_regularizer
Optional scalar Tensor representing L2 regularization penalty. Default: NULL i.e. no regularization.

dispersion
Optional (batch of) Tensor representing response dispersion.

offset
Optional Tensor representing constant shift applied to predicted_linear_response.

convergence_criteria_fn
callable taking: is_converged_previous, iter_, model_coefficients_previous, predicted_linear_response_previous, model_coefficients_next, predicted_linear_response_next, response, model, dispersion and returning a logical Tensor indicating that Fisher scoring has converged.

learning_rate
Optional (batch of) scalar Tensor used to dampen iterative progress. Typically only needed if optimization diverges, should be no larger than 1 and typically very close to 1. Default value: NULL (i.e., 1).

fast_unsafe_numerics
Optional Python bool indicating if faster, less numerically accurate methods can be employed for computing the weighted least-squares solution. Default value: TRUE (i.e., "fast but possibly diminished accuracy").

maximum_iterations
Optional maximum number of iterations of Fisher scoring to run; "and-ed" with result of convergence_criteria_fn. Default value: NULL (i.e., infinity).

name
used as name prefix to ops created by this function. Default value: "fit".

Value
A glm_fit object with parameter estimates, and number of required steps.

See Also
Other glm_fit: glm_families, glm_fit_one_step.tensorflow.tensor()
Description

Runs one Fisher scoring step

Usage

glm_fit_one_step(x, ...)

Arguments

x float-like, matrix-shaped Tensor where each row represents a sample's features.
...
other arguments passed to specific methods.

Value

A glm_fit object with parameter estimates, number of iterations, etc.

See Also

glm_fit_one_step.tensorflow.tensor()
Arguments

- **x**: float-like, matrix-shaped Tensor where each row represents a sample's features.
- **response**: vector-shaped Tensor where each element represents a sample's observed response (to the corresponding row of features). Must have same dtype as x.
- **model**: a string naming the model (see glm_families) or a tfp$glm$ExponentialFamily-like instance which implicitly characterizes a negative log-likelihood loss by specifying the distribution's mean, gradient_mean, and variance.
- **model_coefficients_start**: Optional (batch of) vector-shaped Tensor representing the initial model coefficients, one for each column in x. Must have same dtype as model_matrix. Default value: Zeros.
- **predicted_linear_response_start**: Optional Tensor with shape, dtype matching response; represents offset shifted initial linear predictions based on model_coefficients_start. Default value: offset if model_coefficients is NULL, and tf$linalg.matmul(x, model_coefficients_start) + offset$ otherwise.
- **l2_regularizer**: Optional scalar Tensor representing L2 regularization penalty. Default: NULL i.e. no regularization.
- **dispersion**: Optional (batch of) Tensor representing response dispersion.
- **offset**: Optional Tensor representing constant shift applied to predicted_linear_response.
- **learning_rate**: Optional (batch of) scalar Tensor used to dampen iterative progress. Typically only needed if optimization diverges, should be no larger than 1 and typically very close to 1. Default value: NULL (i.e., 1).
- **fast_unsafe_numerics**: Optional Python bool indicating if faster, less numerically accurate methods can be employed for computing the weighted least-squares solution. Default value: TRUE (i.e., "fast but possibly diminished accuracy").
- **name**: used as name prefix to ops created by this function. Default value: "fit".
- **...**: other arguments passed to specific methods.

Value

A glm_fit object with parameter estimates, and number of required steps.

See Also

Other glm_fit: glm_families, glm_fit.tensorflow.tensor()
initializer_blockwise  

Blockwise Initializer

Description

Initializer which concats other initializers

Usage

initializer_blockwise(initializers, sizes, validate_args = FALSE)

Arguments

- `initializers`: list of Keras initializers, eg: `keras::initializer_glorot_uniform()` or `initializer_constant()`.
- `sizes`: list of integers scalars representing the number of elements associated with each initializer in `initializers`.
- `validate_args`: bool indicating we should do (possibly expensive) graph-time assertions, if necessary.

@return Initializer which concats other initializers

install_tfprobability  

Installs TensorFlow Probability

Description

Installs TensorFlow Probability

Usage

install_tfprobability(
  method = c("auto", "virtualenv", "conda"),
  conda = "auto",
  version = "default",
  tensorflow = "default",
  extra_packages = NULL,
  ...
)


Arguments

- **method**: Installation method ("virtualenv" or "conda")
- **conda**: Path to conda executable (or "auto" to find conda using the PATH and other conventional install locations).
- **version**: Version of Keras to install. Specify "default" to install the latest release. Otherwise specify an alternate version (e.g. "2.2.2").
- **tensorflow**: TensorFlow version to install. Specify "default" to install the CPU version of the latest release. Specify "gpu" to install the GPU version of the latest release. You can also provide a full major.minor.patch specification (e.g. "1.1.0"), appending "-gpu" if you want the GPU version (e.g. "1.1.0-gpu"). Alternatively, you can provide the full URL to an installer binary (e.g. for a nightly binary).
- **extra_packages**: Additional PyPI packages to install along with Keras and TensorFlow.
- **...**: Other arguments passed to tensorflow::install_tensorflow().

Value

- **invisible**

---

**layer_autoregressive**  
*Masked Autoencoder for Distribution Estimation*

Description

layer_autoregressive takes as input a Tensor of shape [..., event_size] and returns a Tensor of shape [..., event_size, params]. The output satisfies the autoregressive property. That is, the layer is configured with some permutation ord of {0, ..., event_size-1} (i.e., an ordering of the input dimensions), and the output output[batch_idx,i,...] for input dimension i depends only on inputs x[batch_idx,j] where ord(j) < ord(i).

Usage

layer_autoregressive(
    object,
    params,
    event_shape = NULL,
    hidden_units = NULL,
    input_order = "left-to-right",
    hidden_degrees = "equal",
    activation = NULL,
    use_bias = TRUE,
    kernel_initializer = "glorot_uniform",
    validate_args = FALSE,
    ...
)
}
Arguments

- **object**: Model or layer object
- **params**: integer specifying the number of parameters to output per input.
- **event_shape**: list-like of positive integers (or a single int), specifying the shape of the input to this layer, which is also the event_shape of the distribution parameterized by this layer. Currently only rank-1 shapes are supported. That is, event_shape must be a single integer. If not specified, the event shape is inferred when this layer is first called or built.
- **hidden_units**: list-like of non-negative integers, specifying the number of units in each hidden layer.
- **input_order**: Order of degrees to the input units: 'random', 'left-to-right', 'right-to-left', or an array of an explicit order. For example, 'left-to-right' builds an autoregressive model: \( p(x) = p(x1) p(x2 \mid x1) \ldots p(xD \mid x<D) \). Default: 'left-to-right'.
- **hidden_degrees**: Method for assigning degrees to the hidden units: 'equal', 'random'. If 'equal', hidden units in each layer are allocated equally (up to a remainder term) to each degree. Default: 'equal'.
- **activation**: An activation function. See keras::layer_dense. Default: NULL.
- **use_bias**: Whether or not the dense layers constructed in this layer should have a bias term. See keras::layer_dense. Default: TRUE.
- **kernel_initializer**: Initializer for the kernel weights matrix. Default: 'glorot_uniform'.
- **validate_args**: logical, default FALSE. When TRUE, layer parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs.
- **...**: Additional keyword arguments passed to the keras::layer_dense constructed by this layer.

Details

The autoregressive property allows us to use output[batch_idx,i] to parameterize conditional distributions: \( p(x[batch_idx, i] \mid x[batch_idx, ] \text{ for ord}(j) < \text{ord}(i)) \) which give us a tractable distribution over input x[batch_idx]:
\[
p(x[batch_idx]) = \prod_i p(x[batch_idx, \text{ord}(i)] \mid x[batch_idx, \text{ord}(0:i)])
\]
For example, when params is 2, the output of the layer can parameterize the location and log-scale of an autoregressive Gaussian distribution.

Value

A Keras layer

See Also

Other layers: layer_conv_1d_flipout(), layer_conv_1d_reparameterization(), layer_conv_2d_flipout(), layer_conv_2d_reparameterization(), layer_conv_3d_flipout(), layer_conv_3d_reparameterization(), layer_dense_flipout(), layer_dense_local_reparameterization(), layer_dense_reparameterization(), layer_dense_variational(), layer_variable()
layer_autoregressive_transform

An autoregressive normalizing flow layer, given a layer_autoregressive.

Description

Following Papamakarios et al. (2017), given an autoregressive model \( p(x) \) with conditional distributions in the location-scale family, we can construct a normalizing flow for \( p(x) \).

Usage

layer_autoregressive_transform(object, made, ...)

Arguments

object Model or layer object
made A Made layer, which must output two parameters for each input.
... Additional parameters passed to Keras Layer.

Details

Specifically, suppose made is a [layer_autoregressive()] – a layer implementing a Masked Autoencoder for Distribution Estimation (MADE) – that computes location and log-scale parameters \( made(x)[i] \) for each input \( x[i] \). Then we can represent the autoregressive model \( p(x) \) as \( x = f(u) \) where \( u \) is drawn from from some base distribution and where \( f \) is an invertible and differentiable function (i.e., a Bijector) and \( f^{-1}(x) \) is defined by:

```r
library(tensorflow)
library(zeallot)
f_inverse <- function(x) {
  c(shift, log_scale) %<-% tf$unstack(made(x), 2, axis = -1L)
  (x - shift) * tf$math$exp(-log_scale)
}
```

Given a layer_autoregressive() made, a layer_autoregressive_transform() transforms an input tfd_* \( p(u) \) to an output tfd_* \( p(x) \) where \( x = f(u) \).

Value

a Keras layer

References

Papamakarios et al. (2017)

See Also

tfb_masked_autoregressive_flow() and layer_autoregressive()
**Description**

$k$ (i.e., `num_components`) represents the number of component OneHotCategorical distributions and $d$ (i.e., `event_size`) represents the number of categories within each OneHotCategorical distribution.

**Usage**

```r
layer_categorical_mixture_of_one_hot_categorical(
  object,  
  event_size,  
  num_components,  
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,  
  sample_dtype = NULL,  
  validate_args = FALSE,  
  ...
)
```

**Arguments**

- **object** Model or layer object
- **event_size** Scalar integer representing the size of single draw from this distribution.
- **num_components** Scalar integer representing the number of mixture components. Must be at least 1. (If `num_components=1`, it's more efficient to use the OneHotCategorical layer.)
- **convert_to_tensor_fn** A callable that takes a `tfd$Distribution` instance and returns a `tf$Tensor-like` object. Default value: `tfd$distributions$Distribution$sample`.
- **sample_dtype** Dtype of samples produced by this distribution. Default value: `NULL` (i.e., previous layer's dtype).
- **validate_args** Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **...** Additional arguments passed to `args` of `keras::create_layer`.

**Details**

Typical choices for `convert_to_tensor_fn` include:

- `tfd$distributions$Distribution$sample`
- `tfd$distributions$Distribution$mean`
- `tfd$distributions$Distribution$mode`
Value

A Keras layer

See Also

For an example how to use in a Keras model, see `layer_independent_normal()`. Other distribution layers: `layer_distribution_lambda()`, `layer_independent_bernoulli()`, `layer_independent_logistic()`, `layer_independent_normal()`, `layer_independent_poisson()`, `layer_kl_divergence_add_loss()`, `layer_kl_divergence_regularizer()`, `layer_mixture_logistic()`, `layer_mixture_normal()`, `layer_mixture_same_family()`, `layer_multivariate_normal_tri_l()`, `layer_one_hot_categorical()`

---

layer_conv_1d_flipout  
1D convolution layer (e.g. temporal convolution) with Flipout

Description

This layer creates a convolution kernel that is convolved (actually cross-correlated) with the layer input to produce a tensor of outputs. It may also include a bias addition and activation function on the outputs. It assumes the kernel and/or bias are drawn from distributions.

Usage

```r
layer_conv_1d_flipout(
  object,
  filters,
  kernel_size,
  strides = 1,
  padding = "valid",
  data_format = "channels_last",
  dilation_rate = 1,
  activation = NULL,
  activity_regularizer = NULL,
  trainable = TRUE,
  kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
  kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
  kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
  bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  bias_prior_fn = NULL,
  bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  ...
)
```
Arguments

- object: Model or layer object
- filters: Integer, the dimensionality of the output space (i.e. the number of filters in the convolution).
- kernel_size: An integer or list of a single integer, specifying the length of the 1D convolution window.
- strides: An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
- padding: One of "valid" or "same" (case-insensitive).
- data_format: A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, length, channels) while channels_first corresponds to inputs with shape (batch, channels, length).
- dilation_rate: An integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.
- activation: Activation function. Set it to None to maintain a linear activation.
- activity_regularizer: Regularizer function for the output.
- trainable: Whether the layer weights will be updated during training.
- kernel_posterior_fn: Function which creates `tfd$Distribution` instance representing the surrogate posterior of the kernel parameter. Default value: `default_mean_field_normal_fn()`.
- kernel_posterior_tensor_fn: Function which takes a `tfd$Distribution` instance and returns a representative value. Default value: `function(d) d %>% tfd_sample()`.
- kernel_prior_fn: Function which creates `tfd$Distribution` instance. See `default_mean_field_normal_fn` docstring for required parameter signature. Default value: `tfd_normal(loc = 0, scale = 1)`.
- kernel_divergence_fn: Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are `tfd$Distribution`-like instances and the sample is a Tensor.
- bias_posterior_fn: Function which creates a `tfd$Distribution` instance representing the surrogate posterior of the bias parameter. Default value: `default_mean_field_normal_fn(is_singular = TRUE)` (which creates an instance of `tfd_deterministic`).
- bias_posterior_tensor_fn: Function which takes a `tfd$Distribution` instance and returns a representative value. Default value: `function(d) d %>% tfd_sample()`.
bias_prior_fn  Function which creates tfd instance. See default_mean_field_normal_fn
docstring for required parameter signature. Default value: NULL (no prior, no
variational inference)

bias_divergence_fn  Function which takes the surrogate posterior distribution, prior distribution and
random variate sample(s) from the surrogate posterior and computes or approx-
imates the KL divergence. The distributions are tfd$Distribution-like in-
stances and the sample is a Tensor.

...  Additional keyword arguments passed to the keras::layer_dense constructed
by this layer.

Details
This layer implements the Bayesian variational inference analogue to a dense layer by assuming the
kernel and/or the bias are drawn from distributions.
By default, the layer implements a stochastic forward pass via sampling from the kernel and bias
posteriors,

outputs = f(inputs; kernel, bias), kernel, bias ~ posterior

where f denotes the layer’s calculation. It uses the Flipout estimator (Wen et al., 2018), which
performs a Monte Carlo approximation of the distribution integrating over the kernel and bias.
Flipout uses roughly twice as many floating point operations as the reparameterization estimator
but has the advantage of significantly lower variance.
The arguments permit separate specification of the surrogate posterior (q(W|x)), prior (p(W)), and
divergence for both the kernel and bias distributions.
Upon being built, this layer adds losses (accessible via the losses property) representing the di-
vergences of kernel and/or bias surrogate posteriors and their respective priors. When doing
minibatch stochastic optimization, make sure to scale this loss such that it is applied just once
per epoch (e.g. if kl is the sum of losses for each element of the batch, you should pass kl
/ num_examples_per_epoch to your optimizer). You can access the kernel and/or bias pos-
terior and prior distributions after the layer is built via the kernel_posterior, kernel_prior,
bias_posterior and bias_prior properties.

Value

a Keras layer

References

- Yeming Wen, Paul Vicol, Jimmy Ba, Dustin Tran, and Roger Grosse. Flipout: Efficient
  Pseudo-Independent Weight Perturbations on Mini-Batches. In International Conference on

See Also

Other layers: layer_autoregressive(), layer_conv_1d_reparameterization(), layer_conv_2d_flipout(),
layer_conv_2d_reparameterization(), layer_conv_3d_flipout(), layer_conv_3d_reparameterization().
**Description**

This layer creates a convolution kernel that is convolved (actually cross-correlated) with the layer input to produce a tensor of outputs. It may also include a bias addition and activation function on the outputs. It assumes the kernel and/or bias are drawn from distributions.

**Usage**

```r
layer_conv_1d_reparameterization(
  object,
  filters,
  kernel_size,
  strides = 1,
  padding = "valid",
  data_format = "channels_last",
  dilation_rate = 1,
  activation = NULL,
  activity_regularizer = NULL,
  trainable = TRUE,
  kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
  kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
  kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
  bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  bias_prior_fn = NULL,
  bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  ...
)
```

**Arguments**

- **object**
  - Model or layer object

- **filters**
  - Integer, the dimensionality of the output space (i.e. the number of filters in the convolution).

- **kernel_size**
  - An integer or list of a single integer, specifying the length of the 1D convolution window.

- **strides**
  - An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
padding: One of "valid" or "same" (case-insensitive).

data_format: A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, length, channels) while channels_first corresponds to inputs with shape (batch, channels, length).

dilation_rate: An integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.

activation: Activation function. Set it to None to maintain a linear activation.

activity_regularizer: Regularizer function for the output.

trainable: Whether the layer weights will be updated during training.

kernel_posterior_fn: Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().

kernel_posterior_tensor_fn: Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().

kernel_prior_fn: Function which creates tfd instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: tfd_normal(loc = 0, scale = 1).

kernel_divergence_fn: Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes an approximation of the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

bias_posterior_fn: Function which creates a tfd$Distribution instance representing the surrogate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular = TRUE) (which creates an instance of tfd_deterministic).

bias_posterior_tensor_fn: Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().

bias_prior_fn: Function which creates tfd instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: NULL (no prior, no variational inference).

bias_divergence_fn: Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes an approximation of the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

... Additional keyword arguments passed to the keras::layer_dense constructed by this layer.
Details

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

\[ \text{outputs} = f(\text{inputs}; \text{kernel}, \text{bias}), \text{kernel}, \text{bias} \sim \text{posterior} \]

where \( f \) denotes the layer’s calculation. It uses the reparameterization estimator (Kingma and Welling, 2014), which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias.

The arguments permit separate specification of the surrogate posterior \( \text{q}(W|x) \), prior \( \text{p}(W) \), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the losses property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if \( k_l \) is the sum of losses for each element of the batch, you should pass \( k_l / \text{num_examples_per_epoch} \) to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the kernel_posterior, kernel_prior, bias_posterior and bias_prior properties.

Value

a Keras layer

References


See Also

Other layers: layer_autoregressive(), layer_conv_1d_flipout(), layer_conv_2d_flipout(), layer_conv_2d_reparameterization(), layer_conv_3d_flipout(), layer_conv_3d_reparameterization(), layer_dense_flipout(), layer_dense_local_reparameterization(), layer_dense_reparameterization(), layer_dense_variational(), layer_variable()
Usage

layer_conv_2d_flipout(
    object,
    filters,
    kernel_size,
    strides = 1,
    padding = "valid",
    data_format = "channels_last",
    dilation_rate = 1,
    activation = NULL,
    activity_regularizer = NULL,
    trainable = TRUE,
    kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
    kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
    kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
    bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    bias_prior_fn = NULL,
    bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    ...
)

Arguments

object          Model or layer object
filters         Integer, the dimensionality of the output space (i.e. the number of filters in the
                convolution).
kernel_size     An integer or list of a single integer, specifying the length of the 1D convolution
                window.
strides         An integer or list of a single integer, specifying the stride length of the con-
                volution. Specifying any stride value != 1 is incompatible with specifying any
                dilation_rate value != 1.
padding         One of "valid" or "same" (case-insensitive).
data_format      A string, one of channels_last (default) or channels_first. The ordering
                of the dimensions in the inputs. channels_last corresponds to inputs with
                shape (batch, length, channels) while channels_first corresponds to inputs
                with shape (batch, channels, length).
dilation_rate   An integer or tuple/list of a single integer, specifying the dilation rate to use
                for dilated convolution. Currently, specifying any dilation_rate value != 1 is
                incompatible with specifying any strides value != 1.
activation      Activation function. Set it to None to maintain a linear activation.
activity_regularizer
                Regularizer function for the output.
trainable       Whether the layer weights will be updated during training.
layer_conv_2d_flipout

kernel_posterior_fn
Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().

kernel_posterior_tensor_fn
Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().

kernel_prior_fn
Function which creates tfd$Distribution instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: tfd_normal(loc = 0, scale = 1).

kernel_divergence_fn
Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

bias_posterior_fn
Function which creates a tfd$Distribution instance representing the surrogate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular = TRUE) (which creates an instance of tfd_deterministic).

bias_posterior_tensor_fn
Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().

bias_prior_fn
Function which creates tfd instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: NULL (no prior, no variational inference)

bias_divergence_fn
Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

... Additional keyword arguments passed to the keras::layer_dense constructed by this layer.

Details
This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

outputs = f(inputs; kernel, bias), kernel, bias ~ posterior

where f denotes the layer's calculation. It uses the Flipout estimator (Wen et al., 2018), which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias. Flipout uses roughly twice as many floating point operations as the reparameterization estimator but has the advantage of significantly lower variance.
The arguments permit separate specification of the surrogate posterior ($q(\omega|x)$), prior ($p(\omega)$), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the `losses` property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g., if $k_l$ is the sum of losses for each element of the batch, you should pass $k_l / \text{num\_examples\_per\_epoch}$ to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the `kernel\_posterior`, `kernel\_prior`, `bias\_posterior` and `bias\_prior` properties.

Value

a Keras layer

References


See Also

Other layers: `layer\_autoregressive()`, `layer\_conv\_1d\_flipout()`, `layer\_conv\_1d\_reparameterization()`, `layer\_conv\_2d\_reparameterization()`, `layer\_conv\_3d\_flipout()`, `layer\_conv\_3d\_reparameterization()`, `layer\_dense\_flipout()`, `layer\_dense\_local\_reparameterization()`, `layer\_dense\_reparameterization()`, `layer\_dense\_variational()`, `layer\_variable()`

---

`layer\_conv\_2d\_reparameterization`

2D convolution layer (e.g. spatial convolution over images)

Description

This layer creates a convolution kernel that is convolved (actually cross-correlated) with the layer input to produce a tensor of outputs. It may also include a bias addition and activation function on the outputs. It assumes the kernel and/or bias are drawn from distributions.

Usage

```python
layer\_conv\_2d\_reparameterization(
    object,
    filters,
    kernel\_size,
    strides = 1,
    padding = "valid",
    data\_format = "channels\_last",
    dilation\_rate = 1,
)```
activation = NULL,
activity_regularizer = NULL,
trainable = TRUE,
kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
bias_prior_fn = NULL,
bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
...)

Arguments

object Model or layer object
filters Integer, the dimensionality of the output space (i.e. the number of filters in the convolution).
kernel_size An integer or list of a single integer, specifying the length of the 1D convolution window.
strides An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
padding One of "valid" or "same" (case-insensitive).
data_format A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, length, channels) while channels_first corresponds to inputs with shape (batch, channels, length).
dilation_rate An integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.
activation Activation function. Set it to None to maintain a linear activation.
activity_regularizer Regularizer function for the output.
trainable Whether the layer weights will be updated during training.
kernel_posterior_fn Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().
kernel_posterior_tensor_fn Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().
kernel_prior_fn Function which creates tfd$Distribution instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: tfd_normal(loc = 0, scale = 1).
kernel_divergence_fn

Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are `tfd$Distribution`-like instances and the sample is a Tensor.

bias_posterior_fn

Function which creates a `tfd$Distribution` instance representing the surrogate posterior of the bias parameter. Default value: `default_mean_field_normal_fn(is_singular = TRUE)` (which creates an instance of `tfd_deterministic`).

bias_posterior_tensor_fn

Function which takes a `tfd$Distribution` instance and returns a representative value. Default value: `function(d) d %>% tfd_sample()`.

bias_prior_fn

Function which creates `tfd` instance. See `default_mean_field_normal_fn` docstring for required parameter signature. Default value: NULL (no prior, no variational inference)

bias_divergence_fn

Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are `tfd$Distribution`-like instances and the sample is a Tensor.

Additional keyword arguments passed to the `keras::layer_dense` constructed by this layer.

Details

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

\[
\text{outputs} = f(\text{inputs}; \text{kernel}, \text{bias}), \text{kernel}, \text{bias} \sim \text{posterior}
\]

where \( f \) denotes the layer's calculation. It uses the reparameterization estimator (Kingma and Welling, 2014), which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias.

The arguments permit separate specification of the surrogate posterior \( (q(W|X)) \), prior \( (p(W)) \), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the `losses` property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if \( \text{kl} \) is the sum of losses for each element of the batch, you should pass \( \text{kl} / \text{num_examples_per_epoch} \) to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the `kernel_posterior`, `kernel_prior`, `bias_posterior` and `bias_prior` properties.

Value

a Keras layer
layer.conv.3d.flipout

References


See Also

Other layers: layer.autoregressive(), layer.conv.1d.flipout(), layer.conv.1d.reparameterization(), layer.conv.2d.flipout(), layer.conv.3d.flipout(), layer.conv.3d.reparameterization(), layer.dense.flipout(), layer.dense.local.reparameterization(), layer.dense.reparameterization(), layer.dense.variational(), layer.variable()

layer.conv.3d.flipout 3D convolution layer (e.g. spatial convolution over volumes) with Flipout

Description

This layer creates a convolution kernel that is convolved (actually cross-correlated) with the layer input to produce a tensor of outputs. It may also include a bias addition and activation function on the outputs. It assumes the kernel and/or bias are drawn from distributions.

Usage

```python
layer.conv.3d.flipout(
    object,
    filters,
    kernel_size,
    strides = 1,
    padding = "valid",
    data_format = "channels_last",
    dilation_rate = 1,
    activation = NULL,
    activity_regularizer = NULL,
    trainable = TRUE,
    kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
    kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
    kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
    bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    bias_prior_fn = NULL,
    bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    ...
)
```
**Arguments**

- **object**: Model or layer object
- **filters**: Integer, the dimensionality of the output space (i.e. the number of filters in the convolution).
- **kernel_size**: An integer or list of a single integer, specifying the length of the 1D convolution window.
- **strides**: An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
- **padding**: One of "valid" or "same" (case-insensitive).
- **data_format**: A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, length, channels) while channels_first corresponds to inputs with shape (batch, channels, length).
- **dilation_rate**: An integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.
- **activation**: Activation function. Set it to None to maintain a linear activation.
- **activity_regularizer**: Regularizer function for the output.
- **trainable**: Whether the layer weights will be updated during training.
- **kernel_posterior_fn**: Function which creates `tfd$Distribution` instance representing the surrogate posterior of the kernel parameter. Default value: `default_mean_field_normal_fn()`.
- **kernel_posterior_tensor_fn**: Function which takes a `tfd$Distribution` instance and returns a representative value. Default value: `function(d) d %>% tfd_sample()`.
- **kernel_prior_fn**: Function which creates `tfd$Distribution` instance. See `default_mean_field_normal_fn` docstring for required parameter signature. Default value: `tfd_normal(loc = 0, scale = 1)`.
- **kernel_divergence_fn**: Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are `tfd$Distribution-like` instances and the sample is a Tensor.
- **bias_posterior_fn**: Function which creates a `tfd$Distribution` instance representing the surrogate posterior of the bias parameter. Default value: `default_mean_field_normal_fn(is_singular = TRUE)` (which creates an instance of `tfd_deterministic`).
- **bias_posterior_tensor_fn**: Function which takes a `tfd$Distribution` instance and returns a representative value. Default value: `function(d) d %>% tfd_sample()`.
bias_prior_fn Function which creates tfd instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: NULL (no prior, no variational inference)

bias_divergence_fn Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

Additional keyword arguments passed to the keras::layer_dense constructed by this layer.

Details

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

\[
\text{outputs} = f(\text{inputs}; \text{kernel, bias}), \text{kernel, bias} \sim \text{posterior}
\]

where f denotes the layer’s calculation. It uses the Flipout estimator (Wen et al., 2018), which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias. Flipout uses roughly twice as many floating point operations as the reparameterization estimator but has the advantage of significantly lower variance.

The arguments permit separate specification of the surrogate posterior (q(W|x)), prior (p(W)), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the losses property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if kl is the sum of losses for each element of the batch, you should pass kl / num_examples_per_epoch to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the kernel_posterior, kernel_prior, bias_posterior and bias_prior properties.

Value

a Keras layer

References


See Also

Other layers: layer_autoregressive(), layer_conv_1d_flipout(), layer_conv_1d_reparameterization(), layer_conv_2d_flipout(), layer_conv_2d_reparameterization(), layer_conv_3d_reparameterization(),
layer_conv_3d_reparameterization

3D convolution layer (e.g. spatial convolution over volumes)

Description

This layer creates a convolution kernel that is convolved (actually cross-correlated) with the layer
input to produce a tensor of outputs. It may also include a bias addition and activation function on
the outputs. It assumes the kernel and/or bias are drawn from distributions.

Usage

layer_conv_3d_reparameterization(
  object,
  filters,
  kernel_size,
  strides = 1,
  padding = "valid",
  data_format = "channels_last",
  dilation_rate = 1,
  activation = NULL,
  activity_regularizer = NULL,
  trainable = TRUE,
  kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
  kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
  kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
  bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  bias_prior_fn = NULL,
  bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  ...)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Model or layer object</td>
</tr>
<tr>
<td>filters</td>
<td>Integer, the dimensionality of the output space (i.e. the number of filters in the convolution).</td>
</tr>
<tr>
<td>kernel_size</td>
<td>An integer or list of a single integer, specifying the length of the 1D convolution window.</td>
</tr>
<tr>
<td>strides</td>
<td>An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.</td>
</tr>
</tbody>
</table>
padding
A string, one of "valid" or "same" (case-insensitive).

data_format
A string, one of channels_last (default) or channels_first. The ordering
of the dimensions in the inputs. channels_last corresponds to inputs with
shape (batch, length, channels) while channels_first corresponds to inputs
with shape (batch, channels, length).

dilation_rate
An integer or tuple/list of a single integer, specifying the dilation rate to use
for dilated convolution. Currently, specifying any dilation_rate value != 1 is
incompatible with specifying any strides value != 1.

activation
Activation function. Set it to None to maintain a linear activation.

activity_regularizer
Regularizer function for the output.

trainable
Whether the layer weights will be updated during training.

kernel_posterior_fn
Function which creates tfd$Distribution instance representing the surrogate
posterior of the kernel parameter. Default value: default_mean_field_normal_fn().

kernel_posterior_tensor_fn
Function which takes a tfd$Distribution instance and returns a representative
value. Default value: function(d) d %>% tfd_sample().

kernel_prior_fn
Function which creates tfd$Distribution instance. See default_mean_field_normal_fn
docstring for required parameter signature. Default value: tfd_normal(loc =
0, scale = 1).

kernel_divergence_fn
Function which takes the surrogate posterior distribution, prior distribution and
random variate sample(s) from the surrogate posterior and computes or approx-
imates the KL divergence. The distributions are tfd$Distribution-like in-
stances and the sample is a Tensor.

bias_posterior_fn
Function which creates a tfd$Distribution instance representing the surro-
gate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular
= TRUE) (which creates an instance of tfd_deterministic).

bias_posterior_tensor_fn
Function which takes a tfd$Distribution instance and returns a representative
value. Default value: function(d) d %>% tfd_sample().

bias_divergence_fn
Function which takes the surrogate posterior distribution, prior distribution and
random variate sample(s) from the surrogate posterior and computes or approx-
imates the KL divergence. The distributions are tfd$Distribution-like in-
stances and the sample is a Tensor.

... Additional keyword arguments passed to the keras::layer_dense constructed
by this layer.
This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

\[
\text{outputs} = f(\text{inputs}; \text{kernel, bias}), \text{kernel, bias} \sim \text{posterior}
\]

where \(f\) denotes the layer’s calculation. It uses the reparameterization estimator (Kingma and Welling, 2014), which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias.

The arguments permit separate specification of the surrogate posterior \(q(W|x)\), prior \(p(W)\), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the `losses` property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if `kl` is the sum of losses for each element of the batch, you should pass `kl / num_examples_per_epoch` to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the `kernel_posterior`, `kernel_prior`, `bias_posterior` and `bias_prior` properties.

**Value**

a Keras layer

**References**


**See Also**

- `layer_autoregressive()`, `layer_conv_1d_flipout()`, `layer_conv_1d_reparameterization()`, `layer_conv_2d_flipout()`, `layer_conv_2d_reparameterization()`, `layer_conv_3d_flipout()`, `layer_dense_flipout()`, `layer_dense_local_reparameterization()`, `layer_dense_reparameterization()`, `layer_dense_variational()`, `layer_variable()`

---

**Description**

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.
Usage

layer_dense_flipout(
    object,
    units,
    activation = NULL,
    activity_regularizer = NULL,
    trainable = TRUE,
    kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
    kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
    kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
    bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    bias_prior_fn = NULL,
    bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    seed = NULL,
    ...
)

Arguments

object Model or layer object
units integer dimensionality of the output space
activation Activation function. Set it to None to maintain a linear activation.
activity_regularizer Regularizer function for the output.
trainable Whether the layer weights will be updated during training.
kernel_posterior_fn Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().
kernell_posterior_tensor_fn Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().
kernell_prior_fn Function which creates tfd$Distribution instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: tfd_normal(loc = 0, scale = 1).
kernell_divergence_fn Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.
bias_posterior_fn Function which creates a tfd$Distribution instance representing the surrogate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular = TRUE) (which creates an instance of tfd_deterministic).
bias_posterior_tensor_fn
Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().

bias_prior_fn
Function which creates tfd instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: NULL (no prior, no variational inference)

bias_divergence_fn
Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

seed
scalar integer which initializes the random number generator. Default value: NULL (i.e., use global seed).

... Additional keyword arguments passed to the keras::layer_dense constructed by this layer.

Details
By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

\[
\text{kernel, bias} \sim \text{posterior} \\
\text{outputs} = \text{activation(matmul(inputs, kernel) + bias)}
\]

It uses the Flipout estimator (Wen et al., 2018), which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias. Flipout uses roughly twice as many floating point operations as the reparameterization estimator but has the advantage of significantly lower variance.

The arguments permit separate specification of the surrogate posterior \((q(W|x))\), prior \((p(W))\), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the losses property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if kl is the sum of losses for each element of the batch, you should pass kl / num_examples_per_epoch to your optimizer).

Value

a Keras layer

References

layer_dense_local_reparameterization

Densely-connected layer class with local reparameterization estimator.

Description

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

Usage

layer_dense_local_reparameterization(
  object,
  units,
  activation = NULL,
  activity_regularizer = NULL,
  trainable = TRUE,
  kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
  kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
  kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
  bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  bias_prior_fn = NULL,
  bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  ...
)

Arguments

object Model or layer object
units integer dimensionality of the output space
activation Activation function. Set it to None to maintain a linear activation.
activity_regularizer Regularizer function for the output.
trainable Whether the layer weights will be updated during training.
kernelposterior_fn Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().
layer_dense_local_reparameterization

`kernel_posterior_tensor_fn`
Function which takes a `tfd$Distribution` instance and returns a representative value. Default value: `function(d) d %>% tfd_sample()`.

`kernel_prior_fn`
Function which creates `tfd$Distribution` instance. See `default_mean_field_normal_fn` docstring for required parameter signature. Default value: `tfd_normal(loc = 0, scale = 1)`.

`kernel_divergence_fn`
Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are `tfd$Distribution`-like instances and the sample is a `Tensor`.

`bias_posterior_fn`
Function which creates a `tfd$Distribution` instance representing the surrogate posterior of the bias parameter. Default value: `default_mean_field_normal_fn(is_singular = TRUE)` (which creates an instance of `tfd_deterministic`).

`bias_posterior_tensor_fn`
Function which takes a `tfd$Distribution` instance and returns a representative value. Default value: `function(d) d %>% tfd_sample()`.

`bias_prior_fn`
Function which creates `tfd` instance. See `default_mean_field_normal_fn` docstring for required parameter signature. Default value: `NULL` (no prior, no variational inference)

`bias_divergence_fn`
Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are `tfd$Distribution`-like instances and the sample is a `Tensor`.

... Additional keyword arguments passed to the `keras::layer_dense` constructed by this layer.

Details
By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

```
kernel, bias ~ posterior
outputs = activation(matmul(inputs, kernel) + bias)
```

It uses the local reparameterization estimator (Kingma et al., 2015), which performs a Monte Carlo approximation of the distribution on the hidden units induced by the kernel and bias. The default `kernel_posterior_fn` is a normal distribution which factorizes across all elements of the weight matrix and bias vector. Unlike that paper’s multiplicative parameterization, this distribution has trainable location and scale parameters which is known as an additive noise parameterization (Molchanov et al., 2017).

The arguments permit separate specification of the surrogate posterior ($q(W|x)$), prior ($p(W)$), and divergence for both the kernel and bias distributions.
Upon being built, this layer adds losses (accessible via the `losses` property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if $\text{kl}$ is the sum of losses for each element of the batch, you should pass $\text{kl} / \text{num\_examples\_per\_epoch}$ to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the `kernel\_posterior`, `kernel\_prior`, `bias\_posterior` and `bias\_prior` properties.

Value

a Keras layer

References


See Also

Other layers: `layer\_autoregressive()`, `layer\_conv\_1d\_flipout()`, `layer\_conv\_1d\_reparameterization()`, `layer\_conv\_2d\_flipout()`, `layer\_conv\_2d\_reparameterization()`, `layer\_conv\_3d\_flipout()`, `layer\_conv\_3d\_reparameterization()`, `layer\_dense\_flipout()`, `layer\_dense\_reparameterization()`, `layer\_dense\_variational()`, `layer\_variable()`

---

`layer\_dense\_reparameterization`

*Densely-connected layer class with reparameterization estimator.*

Description

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

Usage

```python
layer\_dense\_reparameterization(
    object,
    units,
    activation = NULL,
    activity\_regularizer = NULL,
    trainable = TRUE,
    kernel\_posterior\_fn = tfp\$layers\$util\$default\_mean\_field\_normal\_fn(),
    kernel\_posterior\_tensor\_fn = function(d) d %>% tfd\_sample(),
    kernel\_prior\_fn = tfp\$layers\$util\$default\_multivariate\_normal\_fn,
    kernel\_divergence\_fn = function(q, p, ignore) tfd\_kl\_divergence(q, p),
```
layer_dense_reparameterization

bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
bias_prior_fn = NULL,
bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
...

Arguments

object Model or layer object
units integer dimensionality of the output space
activation Activation function. Set it to None to maintain a linear activation.
activity_regularizer Regularizer function for the output.
trainable Whether the layer weights will be updated during training.
kernel_posterior_fn Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().
kernel_posterior_tensor_fn Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().
kernels prior_fn Function which creates tfd$Distribution instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: tfd_normal(loc = 0, scale = 1).
kernel_divergence_fn Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.
bias_posterior_fn Function which creates a tfd$Distribution instance representing the surrogate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular = TRUE) (which creates an instance of tfd_deterministic).
bias_posterior_tensor_fn Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().
bias_prior_fn Function which creates tfd instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: NULL (no prior, no variational inference)
bias_divergence_fn Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.
...

Additional keyword arguments passed to the keras::layer_dense constructed by this layer.
Details

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

$$\text{kernel, bias} \sim \text{posterior}$$

$$\text{outputs} = \text{activation(matmul(inputs, kernel) + bias)}$$

It uses the reparameterization estimator (Kingma and Welling, 2014) which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias.

The arguments permit separate specification of the surrogate posterior ($q(W|x)$), prior ($p(W)$), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the losses property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if $\text{kl}$ is the sum of losses for each element of the batch, you should pass $\text{kl} / \text{num_examples_per_epoch}$ to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the kernel_posterior, kernel_prior, bias_posterior and bias_prior properties.

Value

a Keras layer

References


See Also

Other layers: layer_autoregressive(), layer_conv_1d_flipout(), layer_conv_1d_reparameterization(), layer_conv_2d_flipout(), layer_conv_2d_reparameterization(), layer_conv_3d_flipout(), layer_conv_3d_reparameterization(), layer_dense_flipout(), layer_dense_local_reparameterization(), layer_dense_variational(), layer_variable()

layer_dense_variational

**Dense Variational Layer**

Description

This layer uses variational inference to fit a "surrogate" posterior to the distribution over both the kernel matrix and the bias terms which are otherwise used in a manner similar to layer_dense(). This layer fits the "weights posterior" according to the following generative process:

$$[K, b] \sim \text{Prior()}$$

$$M = \text{matmul}(X, K) + b$$

$$Y \sim \text{Likelihood}(M)$$
**Usage**

```r
dl = layer_dense_variational(
  object,
  units,
  make_posterior_fn,
  make_prior_fn,
  kl_weight = NULL,
  kl_use_exact = FALSE,
  activation = NULL,
  use_bias = TRUE,
  ...
)
```

**Arguments**

- `object`: Model or layer object
- `units`: Positive integer, dimensionality of the output space.
- `make_posterior_fn`: function taking `tf$size(kernel), tf$size(bias), dtype` and returns another callable which takes an input and produces a `tfd$Distribution` instance.
- `make_prior_fn`: function taking `tf$size(kernel), tf$size(bias), dtype` and returns another callable which takes an input and produces a `tfd$Distribution` instance.
- `kl_weight`: Amount by which to scale the KL divergence loss between prior and posterior.
- `kl_use_exact`: Logical indicating that the analytical KL divergence should be used rather than a Monte Carlo approximation.
- `activation`: An activation function. See `keras::layer_dense`. Default: `NULL`.
- `use_bias`: Whether or not the dense layers constructed in this layer should have a bias term. See `keras::layer_dense`. Default: `TRUE`.
- `...`: Additional keyword arguments passed to the `keras::layer_dense` constructed by this layer.

**Value**

a Keras layer

**See Also**

Other layers: `layer_autoregressive()`, `layer_conv_1d_flipout()`, `layer_conv_1d_reparameterization()`, `layer_conv_2d_flipout()`, `layer_conv_2d_reparameterization()`, `layer_conv_3d_flipout()`, `layer_conv_3d_reparameterization()`, `layer_dense_flipout()`, `layer_dense_local_reparameterization()`, `layer_dense_reparameterization()`, `layer_variable()`
layer_distribution_lambda

Keras layer enabling plumbing TFP distributions through Keras models

Description

Keras layer enabling plumbing TFP distributions through Keras models

Usage

layer_distribution_lambda(
  object,
  make_distribution_fn,
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  ...
)

Arguments

object                  Model or layer object
make_distribution_fn    A callable that takes previous layer outputs and returns a tfd$distributions$Distribution instance.
convert_to_tensor_fn    A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
...                     Additional arguments passed to args of keras::create_layer.

Value

a Keras layer

See Also

For an example how to use in a Keras model, see layer_independent_normal().

Other distribution_layers: layer_categorical_mixture_of_one_hot_categorical(), layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_add_loss(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_normal(), layer_mixture_same_family(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()
layer_independent_bernoulli

An Independent-Bernoulli Keras layer from \( \text{prod}(\text{event}\_\text{shape}) \) params

Description

An Independent-Bernoulli Keras layer from \( \text{prod}(\text{event}\_\text{shape}) \) params

Usage

layer_independent_bernoulli(
  object,
  event_shape,
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  sample_dtype = NULL,
  validate_args = FALSE,
  ...
)

Arguments

object Model or layer object

event_shape Scalar integer representing the size of single draw from this distribution.

convert_to_tensor_fn A callable that takes \( \text{tfd} $\text{Distribution}$\) instance and returns \( \text{tf} $\text{Tensor}$\)-like object. Default value: \( \text{tfd} $\text{distributions} $\text{Distribution} $\text{sample}\).

sample_dtype dtype of samples produced by this distribution. Default value: NULL (i.e., previous layer’s dtype).

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ...

Additional arguments passed to args of \( \text{keras} :: \text{create} _\text{layer} \).

Value

a Keras layer

See Also

For an example how to use in a Keras model, see \( \text{layer} _\text{independent} _\text{normal} () \).

Other distribution layers: \( \text{layer} _\text{categorical} _\text{mixture} _\text{of} _\text{one} _\text{hot} _\text{categorical} () , \text{layer} _\text{distribution} _\text{lambda} () , \text{layer} _\text{independent} _\text{logistic} () , \text{layer} _\text{independent} _\text{normal} () , \text{layer} _\text{independent} _\text{poisson} () , \text{layer} _\text{kl} _\text{divergence} _\text{add} _\text{loss} () , \text{layer} _\text{kl} _\text{divergence} _\text{regularizer} () , \text{layer} _\text{mixture} _\text{logistic} () , \text{layer} _\text{mixture} _\text{normal} () , \text{layer} _\text{mixture} _\text{same} _\text{family} () , \text{layer} _\text{multivariate} _\text{normal} _\text{tri} _\text{l} () , \text{layer} _\text{one} _\text{hot} _\text{categorical} () \)
layer_independent_logistic

An independent Logistic Keras layer.

Description

An independent Logistic Keras layer.

Usage

layer_independent_logistic(
  object,
  event_shape,
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  validate_args = FALSE,
  ...
)

Arguments

- object: Model or layer object
- event_shape: Scalar integer representing the size of single draw from this distribution.
- convert_to_tensor_fn: A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
- validate_args: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ...
- ... Additional arguments passed to args of keras::create_layer.

Value

a Keras layer

See Also

For an example how to use in a Keras model, see layer_independent_normal().

Other distribution_layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(), layer_independent_bernoulli(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_add_loss(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_normal(), layer_mixture_same_family(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()
layer_independent_normal

An independent Normal Keras layer.

Description

An independent Normal Keras layer.

Usage

```r
layer_independent_normal(
    object,
    event_shape,
    convert_to_tensor_fn = tfp$distributions$Distribution$sample,
    validate_args = FALSE,
    ...
)
```

Arguments

- `object` Model or layer object
- `event_shape` Scalar integer representing the size of single draw from this distribution.
- `convert_to_tensor_fn` A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
- `validate_args` Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ...
- `...` Additional arguments passed to args of keras::create_layer.

Value

a Keras layer

See Also

Other distribution_layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(), layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_poisson(), layer_kl_divergence_add_loss(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_normal(), layer_mixture_same_family(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()
Examples

```r
library(keras)
input_shape <- c(28, 28, 1)
encoded_shape <- 2
n <- 2
model <- keras_model_sequential(
  list(
    layer_input(shape = input_shape),
    layer_flatten(),
    layer_dense(units = n),
    layer_dense(units = params_size_independent_normal(encoded_shape)),
    layer_independent_normal(event_shape = encoded_shape)
  )
)
```

Description

An independent Poisson Keras layer.

Usage

```r
layer_independent_poisson(
  object, 
  event_shape, 
  convert_to_tensor_fn = tf$distributions$Distribution$sample, 
  validate_args = FALSE, 
  ...
)
```

Arguments

- `object`: Model or layer object
- `event_shape`: Scalar integer representing the size of single draw from this distribution.
- `convert_to_tensor_fn`: A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
- `validate_args`: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. Additional arguments passed to args of keras::create_layer.
- `...`: Additional arguments passed to args of keras::create_layer.
layer_kl_divergence_add_loss

Pass-through layer that adds a KL divergence penalty to the model loss

Description

Pass-through layer that adds a KL divergence penalty to the model loss

Usage

layer_kl_divergence_add_loss(
  object,
  distribution_b,
  use_exact_kl = FALSE,
  test_points_reduce_axis = NULL,
  test_points_fn = tf$convert_to_tensor,
  weight = NULL,
  ...
)

Arguments

object Model or layer object
distribution_b Distribution instance corresponding to b as in KL[a,b]. The previous layer’s output is presumed to be a Distribution instance and is a.
use_exact_kl Logical indicating if KL divergence should be calculated exactly via tfp$distributions$kl_divergence or via Monte Carlo approximation. Default value: FALSE.
test_points_reduce_axis Integer vector or scalar representing dimensions over which to reduce_mean while calculating the Monte Carlo approximation of the KL divergence. As is with all tf$reduce_* ops, NULL means reduce over all dimensions; () means reduce over none of them. Default value: () (i.e., no reduction).
test_points_fn  A callable taking a tfp$distributions$Distribution instance and returning a tensor used for random test points to approximate the KL divergence. Default value: tf$convert_to_tensor.

weight  Multiplier applied to the calculated KL divergence for each Keras batch member. Default value: NULL (i.e., do not weight each batch member).

Additional arguments passed to args of keras::create_layer.

Value

a Keras layer

See Also

For an example how to use in a Keras model, see layer_independent_normal().

Other distribution layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(), layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_normal(), layer_mixture_same_family(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()
Arguments

- **object**: Model or layer object
- **distribution_b**: Distribution instance corresponding to b as in KL[a,b]. The previous layer's output is presumed to be a Distribution instance and is a.
- **use_exact_kl**: Logical indicating if KL divergence should be calculated exactly via `tfp$distributions$kl_divergence` or via Monte Carlo approximation. Default value: FALSE.
- **test_points_reduce_axis**: Integer vector or scalar representing dimensions over which to reduce_mean while calculating the Monte Carlo approximation of the KL divergence. As is with all `tf$reduce_*` ops, NULL means reduce over all dimensions; () means reduce over none of them. Default value: () (i.e., no reduction).
- **test_points_fn**: A callable taking a `tfp$distributions$Distribution` instance and returning a tensor used for random test points to approximate the KL divergence. Default value: `tf$convernto_tensor`.
- **weight**: Multiplier applied to the calculated KL divergence for each Keras batch member. Default value: NULL (i.e., do not weight each batch member).
- **...**: Additional arguments passed to args of `keras::create_layer`.

Value

- a Keras layer

See Also

For an example how to use in a Keras model, see `layer_independent_normal()`.

Other distribution layers: `layer_categorical_mixture_of_one_hot_categorical()`, `layer_distribution_lambda()`, `layer_independent_bernoulli()`, `layer_independent_logistic()`, `layer_independent_normal()`, `layer_independent_poisson()`, `layer_kl_divergence_add_loss()`, `layer_mixture_logistic()`, `layer_mixture_normal()`, `layer_mixture_same_family()`, `layer_multivariate_normal_tri_l()`, `layer_one_hot_categorical()`

---

**layer_mixture_logistic**

A mixture distribution Keras layer, with independent logistic components.

---

**Description**

A mixture distribution Keras layer, with independent logistic components.
**layer_mixture_normal**

Usage

```r
layer_mixture_logistic(
  object,
  num_components,
  event_shape = list(),
  convert_to_tensor_fn = tfp$ distributions$ Distribution$ sample,
  validate_args = FALSE,
  ...
)
```

Arguments

- `object`: Model or layer object
- `num_components`: Number of component distributions in the mixture distribution.
- `event_shape`: integer vector Tensor representing the shape of single draw from this distribution.
- `convert_to_tensor_fn`: A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$ distributions$ Distribution$ sample.
- `validate_args`: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ...

Value

a Keras layer

See Also

For an example how to use in a Keras model, see `layer_independent_normal()`. Other distribution layers: `layer_categorical_mixture_of_one_hot_categorical()`, `layer_distribution_lambda()`, `layer_independent_bernoulli()`, `layer_independent_logistic()`, `layer_independent_normal()`, `layer_independent_poisson()`, `layer_kl_divergence_add_loss()`, `layer_kl_divergence_regularizer()`, `layer_mixture_normal()`, `layer_mixture_same_family()`, `layer_multivariate_normal_tri_l()`, `layer_one_hot_categorical()`

---

**layer_mixture_normal**  
A mixture distribution Keras layer, with independent normal components.

Description

A mixture distribution Keras layer, with independent normal components.
layer_mixture_same_family

Usage

layer_mixture_normal(
  object,
  num_components,
  event_shape = list(),
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  validate_args = FALSE,
  ...
)

Arguments

object Model or layer object

num_components Number of component distributions in the mixture distribution.

event_shape integer vector Tensor representing the shape of single draw from this distribution.

convert_to_tensor_fn A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ... Additional arguments passed to args of keras::create_layer.

Value

a Keras layer

See Also

For an example how to use in a Keras model, see layer_independent_normal().

Other distribution_layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(), layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_add_loss(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_same_family(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()

A mixture (same-family) Keras layer.

Description

A mixture (same-family) Keras layer.
layer_mixture_same_family

Usage

layer_mixture_same_family(
    object,
    num_components,
    component_layer,
    convert_to_tensor_fn = tfp$distributions$Distribution$sample,
    validate_args = FALSE,
    ...
)

Arguments

object  Model or layer object
num_components  Number of component distributions in the mixture distribution.
component_layer  Function that, given a tensor of shape batch_shape + [num_components, component_params_size], returns a tfd.Distribution-like instance that implements the component distribution (with batch shape batch_shape + [num_components]) – e.g., a TFP distribution layer.
convert_to_tensor_fn  A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ...
    Additional arguments passed to args of keras::create_layer.
...
    Additional arguments passed to args of keras::create_layer.

Value

a Keras layer

See Also

For an example how to use in a Keras model, see layer_independent_normal().

Other distribution layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(), layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_add_loss(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_normal(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()
layer_multivariate_normal_tri_l

A d-variate Multivariate Normal TriL Keras layer from $d + d(d+1)/2$ params

Description
A d-variate Multivariate Normal TriL Keras layer from $d + d(d+1)/2$ params

Usage
layer_multivariate_normal_tri_l(
  object,
  event_size,
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  validate_args = FALSE,
  ...
)

Arguments
object: Model or layer object
event_size: Integer vector tensor representing the shape of single draw from this distribution.
convert_to_tensor_fn: A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
validate_args: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
...

Value
a Keras layer

See Also
For an example how to use in a Keras model, see layer_independent_normal().
Other distribution_layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(), layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_add_loss(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_normal(), layer_mixture_same_family(), layer_one_hot_categorical().
layer_one_hot_categorical

A $d$-variate OneHotCategorical Keras layer from d params.

Description

Typical choices for convert_to_tensor_fn include:

- `tfp$distributions$Distribution$sample`
- `tfp$distributions$Distribution$mean`
- `tfp$distributions$Distribution$mode`
- `tfp$distributions$OneHotCategorical$logits`

Usage

```r
layer_one_hot_categorical(
  object,
  event_size,
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  sample_dtype = NULL,
  validate_args = FALSE,
  ...
)
```

Arguments

- **object**: Model or layer object
- **event_size**: Scalar integer representing the size of single draw from this distribution.
- **convert_to_tensor_fn**: A callable that takes a `tfd$Distribution` instance and returns a `tf$Tensor-like` object. Default value: `tfd$distributions$Distribution$sample`.
- **sample_dtype**: Dtype of samples produced by this distribution. Default value: `NULL` (i.e., previous layer’s dtype).
- **validate_args**: Logical, default `FALSE`. When `TRUE` distribution parameters are checked for validity despite possibly degrading runtime performance. When `FALSE` invalid inputs may silently render incorrect outputs. Default value: `FALSE`.
- **...**: Additional arguments passed to args of `keras::create_layer`.

Value

a Keras layer
**layer_variable**

**See Also**

For an example how to use in a Keras model, see `layer_independent_normal()`.

Other distribution layers: `layer_categorical_mixture_of_one_hot_categorical()`, `layer_distribution_lambda()`, `layer_independent_bernoulli()`, `layer_independent_logistic()`, `layer_independent_normal()`, `layer_independent_poisson()`, `layer_kl_divergence_add_loss()`, `layer_kl_divergence_regularizer()`, `layer_mixture_logistic()`, `layer_mixture_normal()`, `layer_mixture_same_family()`, `layer_multivariate_normal()`, `layer_variable`

---

**layer_variable**

**Variable Layer**

**Description**

Simply returns a (trainable) variable, regardless of input. This layer implements the mathematical function $f(x) = c$ where $c$ is a constant, i.e., unchanged for all $x$. Like other Keras layers, the constant is trainable. This layer can also be interpreted as the special case of `layer_dense()` when the kernel is forced to be the zero matrix (tf$\text{zeros}$).

**Usage**

```
layer_variable(
  object,  # Model or layer object
  shape,   # integer or integer vector specifying the shape of the output of this layer.
  dtype = NULL,    # TensorFlow dtype of the variable created by this layer.
  activation = NULL,  # An activation function. See keras::layer_dense. Default: NULL.
  initializer = "zeros",  # Initializer for the constant vector.
  regularizer = NULL,  # Regularizer function applied to the constant vector.
  constraint = NULL,  # Constraint function applied to the constant vector.
  ...
)
```

**Arguments**

- **object**: Model or layer object
- **shape**: integer or integer vector specifying the shape of the output of this layer.
- **dtype**: TensorFlow dtype of the variable created by this layer.
- **activation**: An activation function. See keras::layer_dense. Default: NULL.
- **initializer**: Initializer for the constant vector.
- **regularizer**: Regularizer function applied to the constant vector.
- **constraint**: Constraint function applied to the constant vector.
- **...**: Additional keyword arguments passed to the keras::layer_dense constructed by this layer.

**Value**

a Keras layer
A Variational Gaussian Process Layer.

Description

Create a Variational Gaussian Process distribution whose index_points are the inputs to the layer. Parameterized by number of inducing points and a kernel_provider, which should be a tf.keras.Layer with an @property that late-binds variable parameters to a tfp.positive_semidefinite_kernel.PositiveSemidefiniteKernel instance (this requirement has to do with the way that variables must be created in a keras model). The mean_fn is an optional argument which, if omitted, will be automatically configured to be a constant function with trainable variable output.

Usage

```python
layer_variational_gaussian_process(
    object,  # Model or layer object
    num_inducing_points,  # number of inducing points in the Variational Gaussian Process distribution.
    kernel_provider,  # a Layer instance equipped with an @property, which yields a PositiveSemidefiniteKernel instance. The latter is used to parametrize the constructed Variational Gaussian Process distribution returned by calling the layer.
    event_shape = 1,  # the shape of the output of the layer. This translates to a batch of underlying Variational Gaussian Process distributions. For example, event_shape = 3 means we are modelling a batch of 3 distributions over functions. We can think of this as a distribution over 3-dimensional vector-valued functions.
    inducing_index_points_initializer = NULL,  # default is to use tf.zeros
    unconstrained_observation_noise_variance_initializer = NULL,  # default is to use tf.zeros
    mean_fn = NULL,  # default is to use a constant function with trainable variable output
    jitter = 1e-06,  # default is 1e-06
    name = NULL
)
```

Arguments

- `object`: Model or layer object
- `num_inducing_points`: number of inducing points in the Variational Gaussian Process distribution.
- `kernel_provider`: a Layer instance equipped with an @property, which yields a PositiveSemidefiniteKernel instance. The latter is used to parametrize the constructed Variational Gaussian Process distribution returned by calling the layer.
- `event_shape`: the shape of the output of the layer. This translates to a batch of underlying Variational Gaussian Process distributions. For example, event_shape = 3 means we are modelling a batch of 3 distributions over functions. We can think of this as a distribution over 3-dimensional vector-valued functions.

See Also

Other layers: layer_autoregressive., layer_conv_1d_flipout(), layer_conv_1d_reparameterization(), layer_conv_2d_flipout(), layer_conv_2d_reparameterization(), layer_conv_3d_flipout(), layer_conv_3d_reparameterization(), layer_dense_flipout(), layer_dense_local_reparameterization(), layer_dense_reparameterization(), layer_dense_variational()
inducing_index_points_initializer

A `tf.keras.initializer.Initializer` used to initialize the trainable inducing_index_points variables. Training VGP’s is pretty sensitive to choice of initial inducing index point locations. A reasonable heuristic is to scatter them near the data, not too close to each other.

unconstrained_observation_noise_variance_initializer

A `tf.keras.initializer.Initializer` used to initialize the unconstrained observation noise variable. The observation noise variance is computed from this variable via the `tf.nn.softplus` function.

mean_fn

A callable that maps layer inputs to mean function values. Passed to the mean_fn parameter of Variational Gaussian Process distribution. If omitted, defaults to a constant function with trainable variable value.

jitter

A small term added to the diagonal of various kernel matrices for numerical stability.

name

Name to give to this layer and the scope of ops and variables it contains.

Value

A Keras layer

---

**mcmc_dual_averaging_step_size_adaptation**

Adapts the inner kernel’s step size based on log_accept_prob.

---

**Description**

The dual averaging policy uses a noisy step size for exploration, while averaging over tuning steps to provide a smoothed estimate of an optimal value. It is based on section 3.2 of Hoffman and Gelman (2013), which modifies the stochastic convex optimization scheme of Nesterov (2009). The modified algorithm applies extra weight to recent iterations while keeping the convergence guarantees of Robbins-Monro, and takes care not to make the step size too small too quickly when maintaining a constant trajectory length, to avoid expensive early iterations. A good target acceptance probability depends on the inner kernel. If this kernel is HamiltonianMonteCarlo, then 0.6-0.9 is a good range to aim for. For RandomWalkMetropolis this should be closer to 0.25. See the individual kernels’ docstrings for guidance.

**Usage**

```python
mcmc_dual_averaging_step_size_adaptation(
    inner_kernel,
    num_adaptation_steps,
    target_accept_prob = 0.75,
    exploration_shrinkage = 0.05,
    step_count_smoothing = 10,
    decay_rate = 0.75,
    step_size_setter_fn = NULL,
```

mcmc_dual_averaging_step_size_adaptation

```r
step_size_getter_fn = NULL,
log_accept_prob_getter_fn = NULL,
validate_args = FALSE,
name = NULL
```

Arguments

- **inner_kernel** TransitionKernel-like object.
- **num_adaptation_steps** Scalar integer Tensor number of initial steps to during which to adjust the step size. This may be greater, less than, or equal to the number of burnin steps.
- **target_accept_prob** A floating point Tensor representing desired acceptance probability. Must be a positive number less than 1. This can either be a scalar, or have shape [num_chains]. Default value: 0.75 (the center of asymptotically optimal rate for HMC).
- **exploration_shrinkage** Floating point scalar Tensor. How strongly the exploration rate is biased towards the shrinkage target.
- **step_count_smoothing** Int32 scalar Tensor. Number of "pseudo-steps" added to the number of steps taken to prevents noisy exploration during the early samples.
- **decay_rate** Floating point scalar Tensor. How much to favor recent iterations over earlier ones. A value of 1 gives equal weight to all history.
- **step_size_setter_fn** A function with the signature (kernel_results, new_step_size) -> new_kernel_results where kernel_results are the results of the inner_kernel, new_step_size is a Tensor or a nested collection of Tensors with the same structure as returned by the step_size_getter_fn, and new_kernel_results are a copy of kernel_results with the step size(s) set.
- **step_size_getter_fn** A callable with the signature (kernel_results) -> step_size where kernel_results are the results of the inner_kernel, and step_size is a floating point Tensor or a nested collection of such Tensors.
- **log_accept_prob_getter_fn** A callable with the signature (kernel_results) -> log_accept_prob where kernel_results are the results of the inner_kernel, and log_accept_prob is a floating point Tensor. log_accept_prob can either be a scalar, or have shape [num_chains]. If it's the latter, step_size should also have the same leading dimension.
- **validate_args** logical. When TRUE kernel parameters are checked for validity. When FALSE invalid inputs may silently render incorrect outputs.
- **name** name prefixed to Ops created by this function. Default value: NULL (i.e., 'dual_averaging_step_size_adaptation').
mcmc_dual_averaging_step_size_adaptation

Details

In general, adaptation prevents the chain from reaching a stationary distribution, so obtaining consistent samples requires `num_adaptation_steps` be set to a value somewhat smaller than the number of burnin steps. However, it may sometimes be helpful to set `num_adaptation_steps` to a larger value during development in order to inspect the behavior of the chain during adaptation. The step size is assumed to broadcast with the chain state, potentially having leading dimensions corresponding to multiple chains. When there are fewer of those leading dimensions than there are chain dimensions, the corresponding dimensions in the `log_accept_prob` are averaged (in the direct space, rather than the log space) before being used to adjust the step size. This means that this kernel can do both cross-chain adaptation, or per-chain step size adaptation, depending on the shape of the step size. For example, if your problem has a state with shape [S], your chain state has shape [C0, C1, S] (meaning that there are C0 * C1 total chains) and `log_accept_prob` has shape [C0, C1] (one acceptance probability per chain), then depending on the shape of the step size, the following will happen:

- Step size has shape [], [S] or [1], the `log_accept_prob` will be averaged across its C0 and C1 dimensions. This means that you will learn a shared step size based on the mean acceptance probability across all chains. This can be useful if you don’t have a lot of steps to adapt and want to average away the noise.
- Step size has shape [C1, 1] or [C1, S], the `log_accept_prob` will be averaged across its C0 dimension. This means that you will learn a shared step size based on the mean acceptance probability across chains that share the coordinate across the C1 dimension. This can be useful when the C1 dimension indexes different distributions, while C0 indexes replicas of a single distribution, all sampled in parallel.
- Step size has shape [C0, C1, 1] or [C0, C1, S], then no averaging will happen. This means that each chain will learn its own step size. This can be useful when all chains are sampling from different distributions. Even when all chains are for the same distribution, this can help during the initial warmup period.
- Step size has shape [C0, 1, 1] or [C0, 1, S], the `log_accept_prob` will be averaged across its C1 dimension. This means that you will learn a shared step size based on the mean acceptance probability across chains that share the coordinate across the C0 dimension. This can be useful when the C0 dimension indexes different distributions, while C1 indexes replicas of a single distribution, all sampled in parallel.

Value

a Monte Carlo sampling kernel

References

mcmc_effective_sample_size

See Also

For an example how to use see mcmc_no_u_turn_sampler().

Other mcmc_kernels: mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_random_walk_metropolis(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()

mcmc_effective_sample_size

Estimate a lower bound on effective sample size for each independent chain.

Description

Roughly speaking, "effective sample size" (ESS) is the size of an iid sample with the same variance as state.

Usage

mcmc_effective_sample_size(
  states,
  filter_threshold = 0,
  filter_beyond_lag = NULL,
  name = NULL
)

Arguments

states Tensor or list of Tensor objects. Dimension zero should index identically distributed states.

filter_threshold Tensor or list of Tensor objects. Must broadcast with state. The auto-correlation sequence is truncated after the first appearance of a term less than filter_threshold. Setting to NULL means we use no threshold filter. Since |R_k| <= 1, setting to any number less than -1 has the same effect.

filter_beyond_lag Tensor or list of Tensor objects. Must be int-like and scalar valued. The auto-correlation sequence is truncated to this length. Setting to NULL means we do not filter based on number of lags.

name name to prepend to created ops.
mcmc_hamiltonian_monte_carlo

Details

More precisely, given a stationary sequence of possibly correlated random variables \( X_1, X_2, \ldots, X_N \), each identically distributed ESS is the number such that \( \text{Variance}\left\{ N^{-1} \cdot \text{Sum}\{X_i\} \right\} = \text{ESS}^{-1} \cdot \text{Variance}\{X_1\} \).

If the sequence is uncorrelated, ESS = \( N \). In general, one should expect \( \text{ESS} \leq \ N \), with more highly correlated sequences having smaller ESS.

Value

Tensor or list of Tensor objects. The effective sample size of each component of states. Shape will be states$shape[1:].

See Also

Other mcmc_functions: mcmc_potential_scale_reduction(), mcmc_sample_annealed_importance_chain(), mcmc_sample_chain(), mcmc_sample_halton_sequence()

mcmc_hamiltonian_monte_carlo

Runs one step of Hamiltonian Monte Carlo.

Description

Hamiltonian Monte Carlo (HMC) is a Markov chain Monte Carlo (MCMC) algorithm that takes a series of gradient-informed steps to produce a Metropolis proposal. This class implements one random HMC step from a given current_state. Mathematical details and derivations can be found in Neal (2011).

Usage

mcmc_hamiltonian_monte_carlo(
    target_log_prob_fn,
    step_size,
    num_leapfrog_steps,
    state_gradients_are_stopped = FALSE,
    step_size_update_fn = NULL,
    seed = NULL,
    store_parameters_in_results = FALSE,
    name = NULL
)

Arguments

target_log_prob_fn

Function which takes an argument like current_state (if it's a list current_state will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.
step_size  Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of current_state. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it’s often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.

num_leapfrog_steps  Integer number of steps to run the leapfrog integrator for. Total progress per HMC step is roughly proportional to step_size * num_leapfrog_steps.

state_gradients_are_stopped  logical indicating that the proposed new state be run through tf$stop_gradient. This is particularly useful when combining optimization over samples from the HMC chain. Default value: FALSE (i.e., do not apply stop_gradient).

step_size_update_fn  Function taking current step_size (typically a tf$Variable) and kernel_results (typically collections$namedtuple) and returns updated step_size (Tensors). Default value: NULL (i.e., do not update step_size automatically).

seed  integer to seed the random number generator.

store_parameters_in_results  If TRUE, then step_size and num_leapfrog_steps are written to and read from eponymous fields in the kernel results objects returned from one_step and bootstrap_results. This allows wrapper kernels to adjust those parameters on the fly. This is incompatible with step_size_update_fn, which must be set to NULL.

name  string prefixed to Ops created by this function. Default value: NULL (i.e., 'hmc_kernel').

Details

The one_step function can update multiple chains in parallel. It assumes that all leftmost dimensions of current_state index independent chain states (and are therefore updated independently). The output of target_log_prob_fn(current_state) should sum log-probabilities across all event dimensions. Slices along the rightmost dimensions may have different target distributions; for example, current_state[0, :] could have a different target distribution from current_state[1, :]. These semantics are governed by target_log_prob_fn(current_state). (The number of independent chains is tf$size(target_log_prob_fn(current_state)).)

Value

a Monte Carlo sampling kernel

References


mcmc_metropolis_adjusted_langevin_algorithm

Runs one step of Metropolis-adjusted Langevin algorithm.

Description

Metropolis-adjusted Langevin algorithm (MALA) is a Markov chain Monte Carlo (MCMC) algorithm that takes a step of a discretised Langevin diffusion as a proposal. This class implements one step of MALA using Euler-Maruyama method for a given current_state and diagonal preconditioning volatility matrix.

Usage

mcmc_metropolis_adjusted_langevin_algorithm(
    target_log_prob_fn,
    step_size,
    volatility_fn = NULL,
    seed = NULL,
    parallel_iterations = 10,
    name = NULL
)

Arguments

target_log_prob_fn
  Function which takes an argument like current_state (if it’s a list current_state will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

step_size
  Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of current_state. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it’s often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.

volatility_fn
  function which takes an argument like current_state (or *current_state if it’s a list) and returns volatility value at current_state. Should return a Tensor or list of Tensors that must broadcast with the shape of current_state. Defaults to the identity function.

seed
  integer to seed the random number generator.

See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_random_walk_metropolis(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()
parallel_iterations
the number of coordinates for which the gradients of the volatility matrix volatility_fn can be computed in parallel.

name
String prefixed to Ops created by this function. Default value: NULL (i.e., 'mala_kernel').

Details
Mathematical details and derivations can be found in Roberts and Rosenthal (1998) and Xifara et al. (2013).
The one_step function can update multiple chains in parallel. It assumes that all leftmost dimensions of current_state index independent chain states (and are therefore updated independently). The output of target_log_prob_fn(current_state) should reduce log-probabilities across all event dimensions. Slices along the rightmost dimensions may have different target distributions; for example, current_state[0, :] could have a different target distribution from current_state[1, :]. These semantics are governed by target_log_prob_fn(current_state). (The number of independent chains is tf.size(target_log_prob_fn(current_state)).)

References

See Also
Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_random_walk_monte_carlo(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()

---

**mcmc_metropolis_hastings**

Runs one step of the Metropolis-Hastings algorithm.

---

**Description**

The Metropolis-Hastings algorithm is a Markov chain Monte Carlo (MCMC) technique which uses a proposal distribution to eventually sample from a target distribution.

**Usage**

mcmc_metropolis_hastings(inner_kernel, seed = NULL, name = NULL)
Arguments

- **inner_kernel**: TransitionKernel-like object which has collections\$namedtuple kernel_results and which contains a target_log_prob member and optionally a log_acceptance_correction member.
- **seed**: integer to seed the random number generator.
- **name**: string prefixed to Ops created by this function. Default value: NULL (i.e., "mh_kernel").

Details

Note: inner_kernel\$one_step must return kernel_results as a collections\$namedtuple which must:

- have a target_log_prob field,
- optionally have a log_acceptance_correction field, and,
- have only fields which are Tensor-valued.

The Metropolis-Hastings log acceptance-probability is computed as:

\[
\text{log_accept_ratio} = \left( \text{current_kernel_results.target_log_prob} - \text{previous_kernel_results.target_log_prob} \right) + \text{current_kernel_results.log_acceptance_correction}
\]

If current_kernel_results\$log_acceptance_correction does not exist, it is presumed 0 (i.e., that the proposal distribution is symmetric). The most common use-case for log_acceptance_correction is in the Metropolis-Hastings algorithm, i.e.,

\[
\text{accept_prob}(x' | x) = \frac{p(x')} {p(x)} \frac{g(x|x')} {g(x'|x)}
\]

where,
- \(p\) represents the target distribution,
- \(g\) represents the proposal (conditional) distribution,
- \(x'\) is the proposed state, and,
- \(x\) is current state

The log of the parenthetical term is the log_acceptance_correction. The log_acceptance_correction may not necessarily correspond to the ratio of proposal distributions, e.g., log_acceptance_correction has a different interpretation in Hamiltonian Monte Carlo.

Value

- a Monte Carlo sampling kernel

See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_no_u_turn_sampler(), mcmc_random_walk_metropolis(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()
Description

The No U-Turn Sampler (NUTS) is an adaptive variant of the Hamiltonian Monte Carlo (HMC) method for MCMC. NUTS adapts the distance traveled in response to the curvature of the target density. Conceptually, one proposal consists of reversibly evolving a trajectory through the sample space, continuing until that trajectory turns back on itself (hence the name, 'No U-Turn'). This class implements one random NUTS step from a given current_state. Mathematical details and derivations can be found in Hoffman & Gelman (2011).

Usage

mcmc_no_u_turn_sampler(
  target_log_prob_fn,  
  step_size,  
  max_tree_depth = 10,  
  max_energy_diff = 1000,  
  unrolled_leapfrog_steps = 1,  
  seed = NULL,  
  name = NULL
)

Arguments

target_log_prob_fn  
function which takes an argument like current_state and returns its (possibly unnormalized) log-density under the target distribution.

step_size  
Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of current_state. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it’s often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.

max_tree_depth  
Maximum depth of the tree implicitly built by NUTS. The maximum number of leapfrog steps is bounded by \(2^{\text{max\_tree\_depth}}\) i.e. the number of nodes in a binary tree \(\text{max\_tree\_depth}\) nodes deep. The default setting of 10 takes up to 1024 leapfrog steps.

max_energy_diff  
Scaler threshold of energy differences at each leapfrog, divergence samples are defined as leapfrog steps that exceed this threshold. Default to 1000.

unrolled_leapfrog_steps  
The number of leapfrogs to unroll per tree expansion step. Applies a direct linear multiplier to the maximum trajectory length implied by max_tree_depth. Defaults to 1.  

mcmc_no_u_turn_sampler

Runs one step of the No U-Turn Sampler
seed  integer to seed the random number generator.
name  name prefixed to Ops created by this function. Default value: NULL (i.e., 'nuts_kernel').

Details

The one_step function can update multiple chains in parallel. It assumes that a prefix of leftmost dimensions of current_state index independent chain states (and are therefore updated independently). The output of target_log_prob_fn(current_state) should sum log-probabilities across all event dimensions. Slices along the rightmost dimensions may have different target distributions; for example, current_state[0][0,...] could have a different target distribution from current_state[0][1,...]. These semantics are governed by target_log_prob_fn(*current_state).
(The number of independent chains is tf$size(target_log_prob_fn(current_state)).)

Value

a Monte Carlo sampling kernel

References


See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_random_walk_metropolis(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()

Examples

```r
predictors <- tf$cast(c(201,244,47,287,203,58,210,202,198,158,165,201,157,131,166,160,186,218,146),tf$float32)
obs <- tf$cast(c(592,401,583,402,495,173,479,504,510,416,393,442,317,311,400,337,423,334,533,344),tf$float32)
y_sigma <- tf$cast(c(61,25,15,50,4,15,21,15,27,14,30,16,14,25,16,34,31,42,26,16,22),tf$float32)

# Robust linear regression model
robust_lm <- tfd_joint_distribution_sequential(list(
  tfd_normal(loc = 0, scale = 1),
  # b0
  tfd_normal(loc = 0, scale = 1),
  # b1
  tfd_half_normal(5),
  # df
  function(df, b1, b0)
  tfd_independent(
    tfd_student_t(
```
# Likelihood
\[
df = df[, \text{NULL}], \\
loc = b0[, \text{NULL}] + b1[, \text{NULL}] \ast \text{predictors}[\text{NULL},] \\
scale = y_{\text{sigma}}[\text{NULL},] 
\]

\)
\text{validate}_\text{args} = \text{TRUE})

\text{log}_\text{prob} \leftarrow \text{function}(b0, b1, df) \{ \text{\textbf{robust}}_{\text{lm}} \%\%
  \text{tfd}_\text{log}_\text{prob}(\text{list}(b0, b1, df, obs)) \}

\text{step}_\text{size}0 \leftarrow \text{Map}(\text{function}(x) \text{tf}$\text{cast}(x, \text{tf}$\text{float32}), c(1, .2, .5))

\text{number}_\text{of}_\text{steps} \leftarrow 100
\text{burnin} \leftarrow 50
\text{nchain} \leftarrow 50

\text{run}_\text{chain} \leftarrow \text{function}() \{ 
  \# \text{random initialization of the starting position of each chain} 
  \text{samples} \leftarrow \text{\textbf{robust}}_{\text{lm}} \%\% \text{\textbf{tfd}}_\text{sample}(\text{nchain}) 
  b0 \leftarrow \text{\textbf{samples}}[[1]] 
  b1 \leftarrow \text{\textbf{samples}}[[2]] 
  df \leftarrow \text{\textbf{samples}}[[3]] 

  \# \text{bijector to map contrained parameters to real} 
  \text{unconstraining}_\text{bijectors} \leftarrow \text{list}(\)
    \text{tfb}_\text{identity}(), \text{tfb}_\text{identity}(), \text{tfb}_\text{identity}(), \text{tfb}_\text{exp}()) 

  \text{trace}_\text{fn} \leftarrow \text{function}(x, \text{pkr}) \{ 
    \text{\textbf{list}}(\text{pkr}$\text{inner}_\text{results}$\text{inner}_\text{results}$\text{inner}_\text{results}$\text{step}_\text{size}, 
    \text{pkr}$\text{inner}_\text{results}$\text{inner}_\text{results}$\text{inner}_\text{results}$\text{log}_\text{accept}_\text{ratio}) 
  \}

  \text{nuts} \leftarrow \text{\textbf{mcmc}}_\text{no}_\text{u}_\text{turn}_\text{ sampler}(\)
    \text{target}_\text{log}_\text{prob}_\text{fn} = \text{log}_\text{prob}, 
    \text{step}_\text{size} = \text{step}_\text{size}0 
  \) \%\%

  \text{mcmc}_\text{transformed}_\text{transition}_\text{kernel}(\text{bijector} = \text{unconstraining}_\text{bijectors}) \%\%

  \text{mcmc}_\text{dual}_\text{averaging}_\text{step}_\text{size}_\text{adaptation}(\)
    \text{num}_\text{adaptation}_\text{steps} = \text{burnin}, 
    \text{step}_\text{size}_\text{setter}_\text{fn} = \text{function}(\text{pkr}, \text{new}_\text{step}_\text{size}) 
  \) \%\%

  \text{inner}_\text{results} = \text{pkr}$\text{inner}_\text{results}$\text{replace}('step_size = new_step_size), 
  \text{step}_\text{size}_\text{getter}_\text{fn} = \text{function}(\text{pkr}) \text{pkr}$\text{inner}_\text{results}$\text{step}_\text{size}, 
  \text{log}_\text{accept}_\text{prob}_\text{getter}_\text{fn} = \text{function}(\text{pkr}) \text{pkr}$\text{inner}_\text{results}$\text{log}_\text{accept}_\text{ratio} 

  \text{nuts} \%\% \text{mcmc}_\text{sample}_\text{chain}(\)
    \text{num}_\text{results} = \text{number}_\text{of}_\text{steps}, 
    \text{num}_\text{burnin}_\text{steps} = \text{burnin}, 
    \text{current}_\text{state} = \text{list}(b0, b1, df), 
    \text{trace}_\text{fn} = \text{trace}_\text{fn}) \}

run_chain <- tensorflow::tf_function(run_chain)
res <- run_chain()

mcmc_potential_scale_reduction

Gelman and Rubin (1992)'s potential scale reduction for chain convergence.

Description

Given \( N > 1 \) states from each of \( C > 1 \) independent chains, the potential scale reduction factor, commonly referred to as R-hat, measures convergence of the chains (to the same target) by testing for equality of means.

Usage

mcmc_potential_scale_reduction(
  chains_states,
  independent_chain_ndims = 1,
  name = NULL
)

Arguments

chains_states Tensor or list of Tensors representing the state(s) of a Markov Chain at each result step. The \( i \)th state is assumed to have shape \([N_i, C_{i1}, C_{i2}, \ldots, C_{iD}] + A\). Dimension 0 indexes the \( N_i > 1 \) result steps of the Markov Chain. Dimensions 1 through \( D \) index the \( C_{i1} \times \ldots \times C_{iD} \) independent chains to be tested for convergence to the same target. The remaining dimensions, \( A \), can have any shape (even empty).

independent_chain_ndims Integer type Tensor with value \( \geq 1 \) giving the number of dimensions, from \( \text{dim} = 1 \) to \( \text{dim} = D \), holding independent chain results to be tested for convergence.

name name to prepend to created tf. Default: potential_scale_reduction.

Details

Specifically, R-hat measures the degree to which variance (of the means) between chains exceeds what one would expect if the chains were identically distributed. See Gelman and Rubin (1992), Brooks and Gelman (1998).

Some guidelines:

- The initial state of the chains should be drawn from a distribution overdispersed with respect to the target.
• If all chains converge to the target, then as $N \to \infty$, $R$-hat $\to 1$. Before that, $R$-hat $> 1$ (except in pathological cases, e.g. if the chain paths were identical).

• The above holds for any number of chains $C > 1$. Increasing $C$ improves effectiveness of the diagnostic.

• Sometimes, $R$-hat $< 1.2$ is used to indicate approximate convergence, but of course this is problem dependent. See Brooks and Gelman (1998).

• $R$-hat only measures non-convergence of the mean. If higher moments, or other statistics are desired, a different diagnostic should be used. See Brooks and Gelman (1998).

To see why $R$-hat is reasonable, let $X$ be a random variable drawn uniformly from the combined states (combined over all chains). Then, in the limit $N, C \to \infty$, with $E$, $\text{Var}$ denoting expectation and variance, $R$-hat $= (E[\text{Var}[X | \text{chain}]] + \text{Var}[E[X | \text{chain}]]) / E[\text{Var}[X | \text{chain}]]$. Using the law of total variance, the numerator is the variance of the combined states, and the denominator is the total variance minus the variance of the the individual chain means. If the chains are all drawing from the same distribution, they will have the same mean, and thus the ratio should be one.

**Value**

Tensor or list of Tensors representing the $R$-hat statistic for the state(s). Same dtype as state, and shape equal to state$\text{shape}[1 + \text{independent}_\text{chain}_\text{ndims}]$.

**References**


**See Also**

Other mcmc_functions: mcmc_effective_sample_size(), mcmc_sample_annealed_importance_chain(), mcmc_sample_chain(), mcmc_sample_halton_sequence()

---

**Description**

Random Walk Metropolis is a gradient-free Markov chain Monte Carlo (MCMC) algorithm. The algorithm involves a proposal generating step $\text{proposal}_\text{state} = \text{current}_\text{state} + \text{perturb}$ by a random perturbation, followed by Metropolis-Hastings accept/reject step. For more details see Section 2.1 of Roberts and Rosenthal (2004).
mcmc_random_walk_metropolis

Usage

```r
mcmc_random_walk_metropolis(
    target_log_prob_fn,
    new_state_fn = NULL,
    seed = NULL,
    name = NULL
)
```

Arguments

- **target_log_prob_fn**: Function which takes an argument like `current_state` (if it's a list `current_state` will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

- **new_state_fn**: Function which takes a list of state parts and a seed; returns a same-type list of Tensors, each being a perturbation of the input state parts. The perturbation distribution is assumed to be a symmetric distribution centered at the input state part. Default value: NULL which is mapped to `tfp$mcmc$random_walk_normal_fn()`.

- **seed**: integer to seed the random number generator.

- **name**: String name prefixed to Ops created by this function. Default value: NULL (i.e., 'rwm_kernel').

Details

The current class implements RWM for normal and uniform proposals. Alternatively, the user can supply any custom proposal generating function. The function `one_step` can update multiple chains in parallel. It assumes that all leftmost dimensions of `current_state` index independent chain states (and are therefore updated independently). The output of `target_log_prob_fn(current_state)` should sum log-probabilities across all event dimensions. Slices along the rightmost dimensions may have different target distributions; for example, `current_state[0, :]` could have a different target distribution from `current_state[1, :]`. These semantics are governed by `target_log_prob_fn(current_state)`. (The number of independent chains is `tf$size(target_log_prob_fn(current_state))`.)

Value

a Monte Carlo sampling kernel

See Also

Other mcmc_kernels: `mcmc_dual_averaging_step_size_adaptation()`, `mcmc_hamiltonian_monte_carlo()`, `mcmc_metropolis_adjusted_langevin_algorithm()`, `mcmc_metropolis_hastings()`, `mcmc_no_u_turn_sampler()`, `mcmc_replica_exchange_mc()`, `mcmc_simple_step_size_adaptation()`, `mcmc_slice_sampler()`, `mcmc_transformed_transition_kernel()`, `mcmc_uncalibrated_hamiltonian_monte_carlo()`, `mcmc_uncalibrated_langevin()`, `mcmc_uncalibrated_random_walk()`
Replica Exchange Monte Carlo is a Markov chain Monte Carlo (MCMC) algorithm that is also known as Parallel Tempering. This algorithm performs multiple sampling with different temperatures in parallel, and exchanges those samplings according to the Metropolis-Hastings criterion. The $K$ replicas are parameterized in terms of inverse temperature's, $(\beta[0], \beta[1], ..., \beta[K-1])$. If the target distribution has probability density $p(x)$, the $k$th replica has density $p(x)^{\beta_k}$.

Usage

```r
mcmc_replica_exchange_mc(
  target_log_prob_fn,
  inverse_temperatures,
  make_kernel_fn,
  swap_proposal_fn = tfp$mcmc$replica_exchange_mc$default_swap_proposal_fn(1),
  seed = NULL,
  name = NULL
)
```

Arguments

- `target_log_prob_fn`
  Function which takes an argument like `current_state` (if it's a list `current_state` will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

- `inverse_temperatures`
  1D Tensor of inverse temperatures to perform samplings with each replica. Must have statically known shape. `inverse_temperatures[0]` produces the states returned by samplers, and is typically == 1.

- `make_kernel_fn`
  Function which takes `target_log_prob_fn` and seed args and returns a TransitionKernel instance.

- `swap_proposal_fn`
  function which take a number of replicas, and return combinations of replicas for exchange.

- `seed`
  integer to seed the random number generator.

- `name`
  string prefixed to Ops created by this function. Default value: `NULL` (i.e., "remc_kernel").

Details

Typically $\beta[0] = 1.0$, and $1.0 > \beta[1] > \beta[2] > ... > 0.0$.

- $\beta[0] == 1 ==>$ First replicas samples from the target density, $p$. 
mcmc_sample_annealed_importance_chain

- $\beta[k] < 1$, for $k = 1, \ldots, K-1$ $\Rightarrow$ Other replicas sample from "flattened" versions of $p$ (peak is less high, valley less low). These distributions are somewhat closer to a uniform on the support of $p$. Samples from adjacent replicas $i, i+1$ are used as proposals for each other in a Metropolis step. This allows the lower $\beta$ samples, which explore less dense areas of $p$, to occasionally be used to help the $\beta = 1$ chain explore new regions of the support. Samples from replica 0 are returned, and the others are discarded.

Value

- list of next_state (Tensor or Python list of Tensors representing the state(s) of the Markov chain(s) at each result step. Has same shape as and current_state.) and kernel_results (collections$\$namedtuple of internal calculations used to 'advance the chain).

See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_random_walk_metropolis(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()

mcmc_sample_annealed_importance_chain

    Runs annealed importance sampling (AIS) to estimate normalizing constants.

Description

This function uses an MCMC transition operator (e.g., Hamiltonian Monte Carlo) to sample from a series of distributions that slowly interpolates between an initial "proposal" distribution: $\exp(\text{proposal\_log\_prob\_fn}(x) - \text{proposal\_log\_normalizer})$ and the target distribution: $\exp(\text{target\_log\_prob\_fn}(x) - \text{target\_log\_normalizer})$, accumulating importance weights along the way. The product of these importance weights gives an unbiased estimate of the ratio of the normalizing constants of the initial distribution and the target distribution: $E[\exp(\text{ais\_weights})] = \exp(\text{target\_log\_normalizer} - \text{proposal\_log\_normalizer})$.

Usage

```r
mcmc_sample_annealed_importance_chain(
  num_steps, proposal_log_prob_fn, target_log_prob_fn, current_state, make_kernel_fn, parallel_iterations = 10, name = NULL
)```

Arguments

- **num_steps**: Integer number of Markov chain updates to run. More iterations means more expense, but smoother annealing between q and p, which in turn means exponentially lower variance for the normalizing constant estimator.

- **proposal_log_prob_fn**: function that returns the log density of the initial distribution.

- **target_log_prob_fn**: function which takes an argument like current_state and returns its (possibly unnormalized) log-density under the target distribution.

- **current_state**: Tensor or list of Tensors representing the current state(s) of the Markov chain(s). The first $r$ dimensions index independent chains, $r = \text{tf\$rank(target\_log\_prob\_fn(current\_state))}$.

- **make_kernel_fn**: function which returns a TransitionKernel-like object. Must take one argument representing the TransitionKernel’s target_log_prob_fn. The target_log_prob_fn argument represents the TransitionKernel’s target log distribution. Note: sample_annealed_importance_chain creates a new target_log_prob_fn which is an interpolation between the supplied target_log_prob_fn and proposal_log_prob_fn; it is this interpolated function which is used as an argument to make_kernel_fn.

- **parallel_iterations**: The number of iterations allowed to run in parallel. It must be a positive integer. See tf$\text{while\_loop}$ for more details.

- **name**: string prefixed to Ops created by this function. Default value: NULL (i.e., "sample_annealed_importance_chain").

Details

Note: When running in graph mode, proposal_log_prob_fn and target_log_prob_fn are called exactly three times (although this may be reduced to two times in the future).

Value

- **next_state**: Tensor or Python list of Tensors representing the state(s) of the Markov chain(s) at the final iteration. Has same shape as input current_state), ais_weights (Tensor with the estimated weight(s). Has shape matching target_log_prob_fn(current_state)), and kernel_results (collections.namedtuple of internal calculations used to advance the chain).

See Also

For an example how to use see mcmc_sample_chain().
mcmc_sample_chain

Implements Markov chain Monte Carlo via repeated TransitionKernel steps.

Description

This function samples from an Markov chain at current_state and whose stationary distribution
is governed by the supplied TransitionKernel instance (kernel).

Usage

mcmc_sample_chain(
    kernel = NULL,
    num_results,
    current_state,
    previous_kernel_results = NULL,
    num_burnin_steps = 0,
    num_steps_between_results = 0,
    trace_fn = NULL,
    return_final_kernel_results = FALSE,
    parallel_iterations = 10,
    name = NULL
)

Arguments

kernel An instance of tfp$mcmc$TransitionKernel which implements one step of
the Markov chain.

num_results Integer number of Markov chain draws.

current_state Tensor or list of Tensors representing the current state(s) of the Markov
chain(s).

previous_kernel_results A Tensor or a nested collection of Tensors representing internal calculations
made within the previous call to this function (or as returned by bootstrap_results).

num_burnin_steps Integer number of chain steps to take before starting to collect results. Default
value: 0 (i.e., no burn-in).

num_steps_between_results Integer number of chain steps between collecting a result. Only one out of every
num_steps_between_samples + 1 steps is included in the returned results. The
number of returned chain states is still equal to num_results. Default value: 0
(i.e., no thinning).

trace_fn A function that takes in the current chain state and the previous kernel results
and return a Tensor or a nested collection of Tensors that is then traced along
with the chain state.
mcmc_sample_chain

return_final_kernel_results
If TRUE, then the final kernel results are returned alongside the chain state and the trace specified by the trace_fn.

parallel_iterations
The number of iterations allowed to run in parallel. It must be a positive integer. See tf$while_loop for more details.

name
string prefixed to Ops created by this function. Default value: NULL, (i.e., "mcmc_sample_chain").

Details
This function can sample from multiple chains, in parallel. (Whether or not there are multiple chains is dictated by the kernel.)

The current_state can be represented as a single Tensor or a list of Tensors which collectively represent the current state. Since MCMC states are correlated, it is sometimes desirable to produce additional intermediate states, and then discard them, ending up with a set of states with decreased autocorrelation. See Owen (2017). Such "thinning" is made possible by setting num_steps_between_results > 0. The chain then takes num_steps_between_results extra steps between the steps that make it into the results. The extra steps are never materialized (in calls to sess$run), and thus do not increase memory requirements.

Warning: when setting a seed in the kernel, ensure that sample_chain's parallel_iterations=1, otherwise results will not be reproducible. In addition to returning the chain state, this function supports tracing of auxiliary variables used by the kernel. The traced values are selected by specifying trace_fn. By default, all kernel results are traced but in the future the default will be changed to no results being traced, so plan accordingly. See below for some examples of this feature.

Value
list of:

- checkpointable_states_and_trace: if return_final_kernel_results is TRUE. The return value is an instance of CheckpointableStatesAndTrace.
- all_states: if return_final_kernel_results is FALSE and trace_fn is NULL. The return value is a Tensor or Python list of Tensors representing the state(s) of the Markov chain(s) at each result step. Has same shape as input current_state but with a prepended num_results-size dimension.
- states_and_trace: if return_final_kernel_results is FALSE and trace_fn is not NULL. The return value is an instance of StatesAndTrace.

References

See Also
Other mcmc_functions: mcmc_effective_sample_size(), mcmc_potential_scale_reduction(), mcmc_sample_annealed_importance_chain(), mcmc_sample_halton_sequence()
Examples

```r
dims <- 10
true_stddev <- sqrt(seq(1, 3, length.out = dims))
likelihood <- tfd_multivariate_normal_diag(scale_diag = true_stddev)

kernel <- mcmc_hamiltonian_monte_carlo(
  target_log_prob_fn = likelihood$log_prob,
  step_size = 0.5,
  num_leapfrog_steps = 2
)

states <- kernel %>% mcmc_sample_chain(
  num_results = 1000,
  num_burnin_steps = 500,
  current_state = rep(0, dims),
  trace_fn = NULL
)

sample_mean <- tf$reduce_mean(states, axis = 0L)
sample_stddev <- tf$sqrt(
  tf$reduce_mean(tf$math$squared_difference(states, sample_mean), axis = 0L))
```

---

**mcmc_sample_halton_sequence**

*Returns a sample from the \( \dim \) dimensional Halton sequence.*

---

**Description**

Warning: The sequence elements take values only between 0 and 1. Care must be taken to appropriately transform the domain of a function if it differs from the unit cube before evaluating integrals using Halton samples. It is also important to remember that quasi-random numbers without randomization are not a replacement for pseudo-random numbers in every context. Quasi random numbers are completely deterministic and typically have significant negative autocorrelation unless randomization is used.

**Usage**

```r
mcmc_sample_halton_sequence(
  dim,
  num_results = NULL,
  sequence_indices = NULL,
  dtype = tf$float32,
  randomized = TRUE,
  seed = NULL,
  name = NULL
)
```
Arguments

- **dim**: Positive integer representing each sample's event_size. Must not be greater than 1000.
- **num_results**: (Optional) Positive scalar Tensor of dtype int32. The number of samples to generate. Either this parameter or sequence_indices must be specified but not both. If this parameter is None, then the behaviour is determined by the sequence_indices. Default value: NULL.
- **sequence_indices**: (Optional) Tensor of dtype int32 and rank 1. The elements of the sequence to compute specified by their position in the sequence. The entries index into the Halton sequence starting with 0 and hence, must be whole numbers. For example, sequence_indices=[0, 5, 6] will produce the first, sixth and seventh elements of the sequence. If this parameter is None, then the num_results parameter must be specified which gives the number of desired samples starting from the first sample. Default value: NULL.
- **dtype**: (Optional) The dtype of the sample. One of: float16, float32 or float64. Default value: tf$float32.
- **randomized**: (Optional) bool indicating whether to produce a randomized Halton sequence. If TRUE, applies the randomization described in Owen (2017). Default value: TRUE.
- **seed**: (Optional) integer to seed the random number generator. Only used if randomized is TRUE. If not supplied and randomized is TRUE, no seed is set. Default value: NULL.
- **name**: (Optional) string describing ops managed by this function. If not supplied the name of this function is used. Default value: "sample_halton_sequence".

Details

Computes the members of the low discrepancy Halton sequence in dimension dim. The dim-dimensional sequence takes values in the unit hypercube in dim dimensions. Currently, only dimensions up to 1000 are supported. The prime base for the k-th axes is the k-th prime starting from 2. For example, if dim = 3, then the bases will be [2, 3, 5] respectively and the first element of the non-randomized sequence will be: [0.5, 0.333, 0.2]. For a more complete description of the Halton sequences see here. For low discrepancy sequences and their applications see here.

If randomized is true, this function produces a scrambled version of the Halton sequence introduced by Owen (2017). For the advantages of randomization of low discrepancy sequences see here.

The number of samples produced is controlled by the num_results and sequence_indices parameters. The user must supply either num_results or sequence_indices but not both. The former is the number of samples to produce starting from the first element. If sequence_indices is given instead, the specified elements of the sequence are generated. For example, sequence_indices=tf$range(10) is equivalent to specifying n=10.

Value

halton_elements Elements of the Halton sequence. Tensor of supplied dtype and shape [num_results, dim] if num_results was specified or shape [s, dim] where s is the size of sequence_indices if sequence_indices were specified.
mcmc_simple_step_size_adaptation

References


See Also

For an example how to use see mcmc_sample_chain().

Other mcmc_functions: mcmc_effective_sample_size(), mcmc_potential_scale_reduction(), mcmc_sample_annealed_importance_chain(), mcmc_sample_chain()

mcmc_simple_step_size_adaptation

Adapts the inner kernel’s step size based on log_accept_prob.

Description

The simple policy multiplicatively increases or decreases the step_size of the inner kernel based on the value of log_accept_prob. It is based on equation 19 of Andrieu and Thoms (2008). Given enough steps and small enough adaptation_rate the median of the distribution of the acceptance probability will converge to the target_accept_prob. A good target acceptance probability depends on the inner kernel. If this kernel is HamiltonianMonteCarlo, then 0.6-0.9 is a good range to aim for. For RandomWalkMetropolis this should be closer to 0.25. See the individual kernels’ docstrings for guidance.

Usage

mcmc_simple_step_size_adaptation(
  inner_kernel,
  num_adaptation_steps,
  target_accept_prob = 0.75,
  adaptation_rate = 0.01,
  step_size_setter_fn = NULL,
  step_size_getter_fn = NULL,
  log_accept_prob_getter_fn = NULL,
  validate_args = FALSE,
  name = NULL
)

Arguments

inner_kernel TransitionKernel-like object.
num_adaptation_steps
  Scalar integer Tensor number of initial steps to during which to adjust the step size. This may be greater, less than, or equal to the number of burnin steps.
target_accept_prob
  A floating point Tensor representing desired acceptance probability. Must be a positive number less than 1. This can either be a scalar, or have shape list(num_chains). Default value: 0.75 (the center of asymptotically optimal rate for HMC).
adaptation_rate
Tensor representing amount to scale the current step_size.

step_size_setter_fn
A function with the signature (kernel_results, new_step_size) -> new_kernel_results
where kernel_results are the results of the inner_kernel, new_step_size
is a Tensor or a nested collection of Tensors with the same structure as re-
turned by the step_size_getter_fn, and new_kernel_results are a copy of
kernel_results with the step size(s) set.

step_size_getter_fn
A function with the signature (kernel_results) -> step_size where kernel_results
are the results of the inner_kernel, and step_size is a floating point Tensor
or a nested collection of such Tensors.

log_accept_prob_getter_fn
A function with the signature (kernel_results) -> log_accept_prob where
kernel_results are the results of the inner_kernel, and log_accept_prob
is a floating point Tensor. log_accept_prob can either be a scalar, or have
shape list(num_chains). If it’s the latter, step_size should also have the
same leading dimension.

validate_args Logical. When True kernel parameters are checked for validity. When False
invalid inputs may silently render incorrect outputs.

name string prefixed to Ops created by this class. Default: "simple_step_size_adaptation".

Details
In general, adaptation prevents the chain from reaching a stationary distribution, so obtaining con-
sistent samples requires num_adaptation_steps be set to a value somewhat smaller than the num-
ber of burnin steps. However, it may sometimes be helpful to set num_adaptation_steps to a
larger value during development in order to inspect the behavior of the chain during adaptation.

The step size is assumed to broadcast with the chain state, potentially having leading dimensions
 corresponding to multiple chains. When there are fewer of those leading dimensions than there
 are chain dimensions, the corresponding dimensions in the log_accept_prob are averaged (in the
direct space, rather than the log space) before being used to adjust the step size. This means that
this kernel can do both cross-chain adaptation, or per-chain step size adaptation, depending on the
shape of the step size.

For example, if your problem has a state with shape [S], your chain state has shape [C0, C1, Y]
(meaning that there are C0 * C1 total chains) and log_accept_prob has shape [C0, C1] (one ac-
ceptance probability per chain), then depending on the shape of the step size, the following will
happen:

• Step size has shape [], [S] or [1], the log_accept_prob will be averaged across its C0 and C1
dimensions. This means that you will learn a shared step size based on the mean acceptance
probability across all chains. This can be useful if you don’t have a lot of steps to adapt and
want to average away the noise.

• Step size has shape [C1, 1] or [C1, S], the log_accept_prob will be averaged across its C0
dimension. This means that you will learn a shared step size based on the mean acceptance
probability across chains that share the coordinate across the C1 dimension. This can be useful
when the C1 dimension indexes different distributions, while C0 indexes replicas of a single
distribution, all sampled in parallel.
• Step size has shape [C0, C1, 1] or [C0, C1, S], then no averaging will happen. This means that each chain will learn its own step size. This can be useful when all chains are sampling from different distributions. Even when all chains are for the same distribution, this can help during the initial warmup period.
• Step size has shape [C0, 1, 1] or [C0, 1, S], the log_accept_prob will be averaged across its C1 dimension. This means that you will learn a shared step size based on the mean acceptance probability across chains that share the coordinate across the C0 dimension. This can be useful when the C0 dimension indexes different distributions, while C1 indexes replicas of a single distribution, all sampled in parallel.

Value

A Monte Carlo sampling kernel

References


See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_random_walk_metropolis(), mcmc_replica_exchange_mc(), mcmc_slice_sampler(), mcmc_transformed_transformation(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()
trace_fn = function(x, pkr) {
  list (
    pkr$inner_results$accepted_results$step_size,
    pkr$inner_results$log_accept_ratio
  )
}

samples <- res$all_states
step_size <- res$trace[[1]]
log_accept_ratio <- res$trace[[2]]

Description

Slice Sampling is a Markov Chain Monte Carlo (MCMC) algorithm based, as stated by Neal (2003), on the observation that "...one can sample from a distribution by sampling uniformly from the region under the plot of its density function. A Markov chain that converges to this uniform distribution can be constructed by alternately uniform sampling in the vertical direction with uniform sampling from the horizontal slice defined by the current vertical position, or more generally, with some update that leaves the uniform distribution over this slice invariant". Mathematical details and derivations can be found in Neal (2003). The one dimensional slice sampler is extended to n-dimensions through use of a hit-and-run approach: choose a random direction in n-dimensional space and take a step, as determined by the one-dimensional slice sampling algorithm, along that direction (Belisle et al. 1993).

Usage

mcmc_slice_sampler(
  target_log_prob_fn,
  step_size,
  max_doublings,
  seed = NULL,
  name = NULL
)

Arguments

target_log_prob_fn
  Function which takes an argument like current_state (if it’s a list current_state will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.
step_size: Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of current_state. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it’s often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.

max_doublings: Scalar positive int32 tf$Tensor. The maximum number of doublings to consider.

seed: integer to seed the random number generator.

name: string prefixed to Ops created by this function. Default value: NULL (i.e., 'slice_sampler_kernel').

Details

The one_step function can update multiple chains in parallel. It assumes that all leftmost dimensions of current_state index independent chain states (and are therefore updated independently). The output of target_log_prob_fn(*current_state) should sum log-probabilities across all event dimensions. Slices along the rightmost dimensions may have different target distributions; for example, current_state[0, :] could have a different target distribution from current_state[1, :]. These semantics are governed by target_log_prob_fn(*current_state). (The number of independent chains is tf$size(target_log_prob_fn(*current_state))).

Note that the sampler only supports states where all components have a common dtype.

Value

list of next_state (Tensor or Python list of Tensors representing the state(s) of the Markov chain(s) at each result step. Has same shape as and current_state.) and kernel_results (collections$namedtuple of internal calculations used to advance the chain).

References


See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_random_walk_metropolis(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()
**mcmc_transformed_transition_kernel**

Applies a bijector to the MCMC’s state space

**Description**

The transformed transition kernel enables fitting a bijector which serves to decorrelate the Markov chain Monte Carlo (MCMC) event dimensions thus making the chain mix faster. This is particularly useful when the geometry of the target distribution is unfavorable. In such cases it may take many evaluations of the `target_log_prob_fn` for the chain to mix between faraway states.

**Usage**

```r
mcmc_transformed_transition_kernel(inner_kernel, bijector, name = NULL)
```

**Arguments**

- `inner_kernel` TransitionKernel-like object which has a `target_log_prob_fn` argument.
- `bijector` bijector or list of bijectors. These bijectors use forward to map the `inner_kernel` state space to the state expected by `inner_kernel$target_log_prob_fn`.
- `name` string prefixed to Ops created by this function. Default value: NULL (i.e., "transformed_kernel").

**Details**

The idea of training an affine function to decorrelate chain event dims was presented in Parno and Marzouk (2014). Used in conjunction with the Hamiltonian Monte Carlo transition kernel, the Parno and Marzouk (2014) idea is an instance of Riemannian manifold HMC (Girolami and Calderhead, 2011).

The transformed transition kernel enables arbitrary bijective transformations of arbitrary transition kernels, e.g., one could use bijectors `tfb_affine`, `tfb_real_nvp`, etc. with transition kernels `mcmc_hamiltonian_monte_carlo, mcmc_random_walk_metropolis`, etc.

**Value**

a Monte Carlo sampling kernel

**References**

mcmc_uncalibrated_hamiltonian_monte_carlo

Runs one step of Uncalibrated Hamiltonian Monte Carlo

Description

Warning: this kernel will not result in a chain which converges to the target_log_prob. To get a convergent MCMC, use mcmc_hamiltonian_monte_carlo(...) or mcmc_metropolis_hastings(mcmc_uncalibrated_hamiltonian_monte_carlo(...)).

For more details on UncalibratedHamiltonianMonteCarlo, see HamiltonianMonteCarlo.

Usage

mcmc_uncalibrated_hamiltonian_monte_carlo(
  target_log_prob_fn,
  step_size,
  num_leapfrog_steps,
  state_gradients_are_stopped = FALSE,
  seed = NULL,
  store_parameters_in_results = FALSE,
  name = NULL
)

Arguments

target_log_prob_fn
  Function which takes an argument like current_state (if it's a list current_state will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

step_size
  Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of current_state. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it's often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.

num_leapfrog_steps
  Integer number of steps to run the leapfrog integrator for. Total progress per HMC step is roughly proportional to step_size * num_leapfrog_steps.

state_gradients_are_stopped
  Logical indicating that the proposed new state be run through tf$stop_gradient. This is particularly useful when combining optimization over samples from the HMC chain. Default value: FALSE (i.e., do not apply stop_gradient).
mcmc_uncalibrated_langevin

Runs one step of Uncalibrated Langevin discretized diffusion.

Description

The class generates a Langevin proposal using _euler_method function and also computes helper UncalibratedLangevinKernelResults for the next iteration. Warning: this kernel will not result in a chain which converges to the target_log_prob. To get a convergent MCMC, use MetropolisAdjustedLangevinAlgorithm or MetropolisHastings(UncalibratedLangevin(...)).

Usage

mcmc_uncalibrated_langevin(
  target_log_prob_fn,
  step_size,
  volatility_fn = NULL,
  parallel_iterations = 10,
  compute_acceptance = TRUE,
  seed = NULL,
  name = NULL
)
Arguments

- **target_log_prob_fn**
  Function which takes an argument like `current_state` (if it’s a list `current_state` will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

- **step_size**
  Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of `current_state`. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it’s often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.

- **volatility_fn**
  Function which takes an argument like `current_state` (or *current_state if it’s a list) and returns volatility value at `current_state`. Should return a Tensor or list of Tensors that must broadcast with the shape of `current_state`. Defaults to the identity function.

- **parallel_iterations**
  The number of coordinates for which the gradients of the volatility matrix `volatility_fn` can be computed in parallel.

- **compute_acceptance**
  Logical indicating whether to compute the Metropolis log-acceptance ratio used to construct `MetropolisAdjustedLangevinAlgorithm` kernel.

- **seed**
  Integer to seed the random number generator.

- **name**
  String prefixed to Ops created by this function. Default value: NULL (i.e., 'mala_kernel').

Value

- list of `next_state` (Tensor or Python list of Tensors representing the state(s) of the Markov chain(s) at each result step. Has same shape as and `current_state`.) and `kernel_results` (collections$namedtuple of internal calculations used to ‘advance the chain).

See Also

Other `mcmc_kernels`: `mcmc_dual_averaging_step_size_adaptation()`, `mcmc_hamiltonian_monte_carlo()`, `mcmc_metropolis_adjusted_langevin_algorithm()`, `mcmc_metropolis_hastings()`, `mcmc_no_u_turn_sampler()`, `mcmc_random_walk_metropolis()`, `mcmc_replica_exchange_mc()`, `mcmc_simple_step_size_adaptation()`, `mcmc_slice_sampler()`, `mcmc_transformed_transition_kernel()`, `mcmc_uncalibrated_hamiltonian_monte_carlo()`, `mcmc_uncalibrated_random_walk()`

---

**mcmc_uncalibrated_random_walk**

Generate proposal for the Random Walk Metropolis algorithm.

---

**Description**

Warning: this kernel will not result in a chain which converges to the `target_log_prob`. To get a convergent MCMC, use `mcmc_random_walk_metropolis(...)` or `mcmc_metropolis_hastings(mcmc_uncalibrated_random_walk(...))`.
Usage

mcmc_uncalibrated_random_walk(
    target_log_prob_fn,
    new_state_fn = NULL,
    seed = NULL,
    name = NULL
)

Arguments

**target_log_prob_fn**
Function which takes an argument like `current_state` (if it's a list `current_state` will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

**new_state_fn**
Function which takes a list of state parts and a seed; returns a same-type list of Tensors, each being a perturbation of the input state parts. The perturbation distribution is assumed to be a symmetric distribution centered at the input state part. Default value: NULL which is mapped to `tfp$mcmc$random_walk_normal_fn()`.

**seed**
integer to seed the random number generator.

**name**
String name prefixed to Ops created by this function. Default value: NULL (i.e., 'rwm_kernel').

Value

a Monte Carlo sampling kernel

See Also

Other `mcmc_kernels`: `mcmc_dual_averaging_step_size_adaptation()`, `mcmc_hamiltonian_monte_carlo()`, `mcmc_metropolis_adjusted_langevin_algorithm()`, `mcmc_metropolis_hastings()`, `mcmc_no_u_turn_sampler()`, `mcmc_random_walk_metropolis()`, `mcmc_replica_exchange_mc()`, `mcmc_simple_step_size_adaptation()`, `mcmc_slice_sampler()`, `mcmc_transformed_transition_kernel()`, `mcmc_uncalibrated_hamiltonian_monte_carlo()`, `mcmc_uncalibrated_langevin()`

---

**params_size_categorical_mixture_of_one_hot_categorical**

*number of params needed to create a CategoricalMixtureOfOneHotCategorical distribution*

---

**Description**

number of params needed to create a CategoricalMixtureOfOneHotCategorical distribution
Usage

```cpp
params_size_categorical_mixture_of_one_hot_categorical(
    event_size,
    num_components
)
```

Arguments

- `event_size`: event size of this distribution
- `num_components`: number of components in the mixture

Value

`a scalar`

---

Description

number of params needed to create an IndependentBernoulli distribution

Usage

```cpp
params_size_independent_bernoulli(event_size)
```

Arguments

- `event_size`: event size of this distribution

Value

`a scalar`
**params_size_independent_logistic**

*number of params needed to create an IndependentLogistic distribution*

**Description**

number of params needed to create an IndependentLogistic distribution

**Usage**

`params_size_independent_logistic(event_size)`

**Arguments**

`event_size` event size of this distribution

**Value**

a scalar

---

**params_size_independent_normal**

*number of params needed to create an IndependentNormal distribution*

**Description**

number of params needed to create an IndependentNormal distribution

**Usage**

`params_size_independent_normal(event_size)`

**Arguments**

`event_size` event size of this distribution

**Value**

a scalar
**params_size_independent_poisson**

*number of params needed to create an IndependentPoisson distribution*

---

**Description**

number of params needed to create an IndependentPoisson distribution

**Usage**

```python
params_size_independent_poisson(event_size)
```

**Arguments**

- `event_size` event size of this distribution

**Value**

a scalar

---

**params_size_mixture_logistic**

*number of params needed to create a MixtureLogistic distribution*

---

**Description**

number of params needed to create a MixtureLogistic distribution

**Usage**

```python
params_size_mixture_logistic(num_components, event_shape)
```

**Arguments**

- `num_components` Number of component distributions in the mixture distribution.
- `event_shape` Number of parameters needed to create a single component distribution.

**Value**

a scalar
**params_size_mixture_normal**

number of params needed to create a MixtureNormal distribution

**Description**

number of params needed to create a MixtureNormal distribution

**Usage**

```python
params_size_mixture_normal(num_components, event_shape)
```

**Arguments**

- `num_components`  Number of component distributions in the mixture distribution.
- `event_shape`  Number of parameters needed to create a single component distribution.

**Value**

a scalar

---

**params_size_mixture_same_family**

number of params needed to create a MixtureSameFamily distribution

**Description**

number of params needed to create a MixtureSameFamily distribution

**Usage**

```python
params_size_mixture_same_family(num_components, component_params_size)
```

**Arguments**

- `num_components`  Number of component distributions in the mixture distribution.
- `component_params_size`  Number of parameters needed to create a single component distribution.

**Value**

a scalar
**params_size_multivariate_normal_tri_l**

*number of params needed to create a MultivariateNormalTriL distribution*

---

**Description**

number of params needed to create a MultivariateNormalTriL distribution

**Usage**

`params_size_multivariate_normal_tri_l(event_size)`

**Arguments**

- `event_size` event size of this distribution

**Value**

a scalar

---

**params_size_one_hot_categorical**

*number of params needed to create a OneHotCategorical distribution*

---

**Description**

number of params needed to create a OneHotCategorical distribution

**Usage**

`params_size_one_hot_categorical(event_size)`

**Arguments**

- `event_size` event size of this distribution

**Value**

a scalar
A state space model representing a sum of component state space models.

Description

A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with dynamics specified by a probabilistic transition model $p(z[t+1] | z[t])$. At each timestep, we observe a value sampled from an observation model conditioned on the current state, $p(x[t] | z[t])$. The special case where both the transition and observation models are Gaussians with mean specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see `tfd_linear_gaussian_state_space_model` for details.

Usage

```r
sts_additive_state_space_model(
  component_ssms,
  constant_offset = 0,
  observation_noise_scale = NULL,
  initial_state_prior = NULL,
  initial_step = 0,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = NULL
)
```

Arguments

- `component_ssms` list containing one or more `tfd_linear_gaussian_state_space_model` instances. The components will in general implement different time-series models, with possibly different latent size, but they must have the same `dtype`, event shape (`num_timesteps` and `observation_size`), and their batch shapes must broadcast to a compatible batch shape.
- `constant_offset` scalar `float` tensor, or batch of scalars, specifying a constant value added to the sum of outputs from the component models. This allows the components to model the shifted series `observed_time_series - constant_offset`. Default value: `0`.
- `observation_noise_scale` Optional scalar `float` tensor indicating the standard deviation of the observation noise. May contain additional batch dimensions, which must broadcast with the batch shape of elements in `component_ssms`. If `observation_noise_scale` is specified for the `sts_additive_state_space_model`, the observation noise scales of component models are ignored. If `NULL`, the observation noise scale is
The `sts_additive_state_space_model` represents a sum of component state space models. Each of the N components describes a random process generating a distribution on observed time series x1[t], x2[t], ..., xN[t]. The additive model represents the sum of these processes, y[t] = x1[t] + x2[t] + ... + xN[t] + eps[t], where eps[t] ~ N(0, observation_noise_scale) is an observation noise term.

Mathematical Details

The additive model concatenates the latent states of its component models. The generative process runs each component’s dynamics in its own subspace of latent space, and then observes the sum of the observation models from the components.

Formally, the transition model is linear Gaussian:

\[ p(z[t+1] | z[t]) \sim \text{Normal}(\text{loc} = \text{transition_matrix}.\text{matmul}(z[t]), \text{cov} = \text{transition_cov}) \]

where each z[t] is a latent state vector concatenating the component state vectors, z[t] = [z1[t], z2[t], ..., zN[t]], so it has size latent_size = sum([c.latent_size for c in components]).

The transition matrix is the block-diagonal composition of transition matrices from the component processes:

\[
\begin{bmatrix}
  c0.transition_matrix, & 0, & \ldots, & 0, \\
  0, & c1.transition_matrix, & \ldots, & 0, \\
  \ldots & \ldots & \ldots \\
  0, & 0, & \ldots, & cN.transition_matrix
\end{bmatrix}
\]
and the noise covariance is similarly the block-diagonal composition of component noise covariances:

$$
\text{transition}\_\text{cov} = \\
\begin{bmatrix}
\text{c0}.\text{transition}\_\text{cov}, 0., \ldots, 0. \\
0., \text{c1}.\text{transition}\_\text{cov}, \ldots, 0. \\
\ldots \\
0., 0., \ldots, \text{cN}.\text{transition}\_\text{cov}
\end{bmatrix}
$$

The observation model is also linear Gaussian,

$$
p(y[t] | z[t]) \sim \text{Normal(loc = observation}\_\text{matrix}.\text{matmul}(z[t]), \text{stddev = observation}\_\text{noise}\_\text{scale})
$$

This implementation assumes scalar observations, so observation\_matrix has shape [1, latent\_size]. The additive observation matrix simply concatenates the observation matrices from each component:

$$
\text{observation}\_\text{matrix} = \text{concat([c0.obs_matrix, c1.obs_matrix, \ldots, cN.obs_matrix], axis=-1)}
$$

The effect is that each component observation matrix acts on the dimensions of latent state corresponding to that component, and the overall expected observation is the sum of the expected observations from each component.

If observation\_noise\_scale is not explicitly specified, it is also computed by summing the noise variances of the component processes:

$$
\text{observation}\_\text{noise}\_\text{scale} = \text{sqrt(sum([c.observation}\_\text{noise}\_\text{scale}**2 for c in components]))}
$$

Value

an instance of LinearGaussianStateSpaceModel.

See Also

Other sts: sts_autoregressive\_state\_space\_model(), sts_autoregressive(), sts_constrained\_seasonal\_state\_space\_model(), sts_constrained\_seasonal(), sts_dynamic\_linear\_regression\_state\_space\_model(), sts_dynamic\_linear\_regression(), sts_linear\_regression(), sts_local\_level\_state\_space\_model(), sts_local\_level(), sts_local\_linear\_trend(), sts_local\_linear\_trend(), sts_seasonal\_state\_space\_model(), sts_seasonal(), sts_semi_local\_linear\_trend(), sts_semi_local\_linear\_trend(), sts_smooth\_seasonal\_state\_space\_model(), sts_smooth\_seasonal(), sts_sparse\_linear\_regression(), sts_sum()
ststs_autoregressive

Formal representation of an autoregressive model.

Description

An autoregressive (AR) model posits a latent level whose value at each step is a noisy linear combination of previous steps:

\[
\text{level}[t+1] = (\text{sum}(\text{coefficients} \times \text{levels}[t:t-order:-1]) + \text{Normal}(0., \text{level_scale}))
\]

Usage

\[
\text{sts_autoregressive}(\text{observed_time_series = NULL, order, coefficients_prior = NULL, level_scale_prior = NULL, initial_state_prior = NULL, coefficient_constraining_bijector = NULL, name = NULL})
\]

Arguments

- **observed_time_series**
  optional float tensor of shape batch_shape + [T, 1] (omitting the trailing unit dimension is also supported when T > 1), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of sts_masked_time_series, which includes a mask tensor to specify timesteps with missing observations. Default value: NULL.

- **order**
  scalar positive integer specifying the number of past timesteps to regress on.

- **coefficients_prior**
  optional Distribution instance specifying a prior on the coefficients parameter. If NULL, a default standard normal (tfd_multivariate_normal_diag(scale_diag = tf$ones(list(order)))) prior is used. Default value: NULL.

- **level_scale_prior**
  optional Distribution instance specifying a prior on the level_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

- **initial_state_prior**
  optional Distribution instance specifying a prior on the initial state, corresponding to the values of the process at a set of size order of imagined timesteps before the initial step. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.
sts_autoregressive_state_space_model

State space model for an autoregressive process.

Description

A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with dynamics specified by a probabilistic transition model \( p(z[t+1] | z[t]) \). At each timestep, we observe a value sampled from an observation model conditioned on the current state, \( p(x[t] | z[t]) \). The special case where both the transition and observation models are Gaussians with mean specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see `tfd_linear_gaussian_state_space_model` for details.

coefficient_constraining_bijector

optional Bijector instance representing a constraining mapping for the autoregressive coefficients. For example, `tfb_tanh()` constrains the coefficients to lie in \((-1, 1)\), while `tfb_softplus()` constrains them to be positive, and `tfb_identity()` implies no constraint. If `NULL`, the default behavior constrains the coefficients to lie in \((-1, 1)\) using a tanh bijector. Default value: `NULL`.

name

the name of this model component. Default value: 'Autoregressive'.

Details

The latent state is \( \text{levels}[t:t-order:-1] \). We observe a noisy realization of the current level:

\[
f[t] = \text{level}[t] + \text{Normal}(0, \text{observation_noise_scale})
\]

each timestep.

If coefficients=\([1.0]\), the AR process is a simple random walk, equivalent to a LocalLevel model. However, a random walk's variance increases with time, while many AR processes (in particular, any first-order process with \( \text{abs(coefficient)} < 1 \)) are stationary, i.e., they maintain a constant variance over time. This makes AR processes useful models of uncertainty.

Value

an instance of StructuralTimeSeries.

See Also

For usage examples see `sts_fit_with_hmc()`, `sts_forecast()`, `sts_decompose_by_component()`.

Other sts: `sts_additive_state_space_model()`, `sts_autoregressive_state_space_model()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_linear_regression()`, `sts_local_level_state_space_model()`, `sts_local_level()`, `sts_local_linear_trend_state_space_model()`, `sts_local_linear_trend()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_linear_trend_state_space_model()`, `sts_semi_local_linear_trend()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sparse_linear_regression()`, `sts_sum()`
Usage

```r
sts_autoregressive_state_space_model(
  num_timesteps,
  coefficients,
  level_scale,
  initial_state_prior,
  observation_noise_scale = 0,
  initial_step = 0,
  validate_args = FALSE,
  name = NULL
)
```

Arguments

- `num_timesteps`: Scalar integer tensor number of timesteps to model with this distribution.
- `coefficients`: float tensor of shape `tf$concat(batch_shape, list(order))` defining the autoregressive coefficients. The coefficients are defined backwards in time:
  ```latex
  \text{coefficients}[0] \times \text{level}[t] + \text{coefficients}[1] \times \text{level}[t-1] + \ldots + \text{coefficients}[\text{order}-1] \times \text{level}[t-\text{order}+1].
  ```
- `level_scale`: Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the transition noise at each step.
- `initial_state_prior`: instance of `tfd_multivariate_normal` representing the prior distribution on latent states. Must have event shape `list(order)`.
- `observation_noise_scale`: Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise. Default value: 0.
- `initial_step`: Optional scalar int tensor specifying the starting timestep. Default value: 0.
- `validate_args`: logical. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed. Default value: `FALSE`.
- `name`: name prefixed to ops created by this class. Default value: "AutoregressiveStateSpaceModel".

Details

In an autoregressive process, the expected level at each timestep is a linear function of previous levels, with added Gaussian noise:

```latex
\text{level}[t+1] = (\text{sum(coefficients} \times \text{levels}[t:t-\text{order}:1]) + \text{Normal}(0., \text{level_scale}))
```

The process is characterized by a vector `coefficients` whose size determines the order of the process (how many previous values it looks at), and by `level_scale`, the standard deviation of the noise added at each step. This is formulated as a state space model by letting the latent state encode the most recent values; see 'Mathematical Details' below.
The parameters level_scale and observation_noise_scale are each (a batch of) scalars, and coefficients is a (batch) vector of size list(order). The batch shape of this Distribution is the broadcast batch shape of these parameters and of the initial_state_prior.

Mathematical Details

The autoregressive model implements a `tfd_linear_gaussian_state_space_model` with latent_size = order and observation_size = 1. The latent state vector encodes the recent history of the process, with the current value in the topmost dimension. At each timestep, the transition sums the previous values to produce the new expected value, shifts all other values down by a dimension, and adds noise to the current value. This is formally encoded by the transition model:

\[
\text{transition_matrix} = \begin{bmatrix}
\text{coefs}[0], \text{coefs}[1], \ldots, \text{coefs}[\text{order}]
1., & 0, & \ldots, & 0.
0., & 1., & \ldots, & 0.
\ldots
0., & 0., & \ldots, & 1., & 0.
\end{bmatrix}
\]

\[
\text{transition_noise} \sim N(\text{loc}=0., \text{scale} = \text{diag([level_scale, 0., 0., \ldots, 0.])})
\]

The observation model simply extracts the current (topmost) value, and optionally adds independent noise at each step:

\[
\text{observation_matrix} = \begin{bmatrix} 1., & 0., & \ldots, & 0. \end{bmatrix}
\]

\[
\text{observation_noise} \sim N(\text{loc}=0, \text{scale}=\text{observation_noise_scale})
\]

Models with observation_noise_scale = 0 are AR processes in the formal sense. Setting observation_noise_scale to a nonzero value corresponds to a latent AR process observed under an iid noise model.

Value

an instance of LinearGaussianStateSpaceModel.

See Also

Other sts: `sts_additive_state_space_model()`, `sts_autoregressive()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_linear_regression()`, `sts_local_level_state_space_model()`, `sts_local_level()`, `sts_local_linear_trend()`, `sts_semi_local_linear_trend()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_linear_trend()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sparse_linear_regression()`, `sts_sum()`
Build a variational posterior that factors over model parameters.

Description

The surrogate posterior consists of independent Normal distributions for each parameter with trainable loc and scale, transformed using the parameter's bijector to the appropriate support space for that parameter.

Usage

```r
sts_build_factored_surrogate_posterior(
  model,
  batch_shape = list(),
  seed = NULL,
  name = NULL
)
```

Arguments

- `model`: An instance of StructuralTimeSeries representing a time-series model. This represents a joint distribution over time-series and their parameters with batch shape `[b1, ..., bN]`.
- `batch_shape`: Batch shape (list, or integer) of initial states to optimize in parallel. Default value: list(). (i.e., just run a single optimization).
- `seed`: integer to seed the random number generator.
- `name`: string prefixed to ops created by this function. Default value: NULL (i.e., 'build_factored_surrogate_posterior').

Value

`variational_posterior tfd_joint_distribution_named` defining a trainable surrogate posterior over model parameters. Samples from this distribution are named lists with character parameter names as keys.

See Also

Other sts-functions: `sts_build_factored_variational_loss()`, `sts_decompose_by_component()`, `sts_decompose_forecast_by_component()`, `sts_fit_with_hmc()`, `sts_forecast()`, `sts_one_step_predictive()`, `sts_sample_uniform_initial_state()`
Variational inference searches for the distribution within some family of approximate posteriors that minimizes a divergence between the approximate posterior $q(z)$ and true posterior $p(z|\text{observed\_time\_series})$. By converting inference to optimization, it's generally much faster than sampling-based inference algorithms such as HMC. The tradeoff is that the approximating family rarely contains the true posterior, so it may miss important aspects of posterior structure (in particular, dependence between variables) and should not be blindly trusted. Results may vary; it's generally wise to compare to HMC to evaluate whether inference quality is sufficient for your task at hand.

**Usage**

```r
sts_build_factored_variational_loss(
  observed_time_series,  
  model, 
  init_batch_shape = list(), 
  seed = NULL, 
  name = NULL
)
```

**Arguments**

- `observed_time_series` float tensor of shape concat([sample_shape, model.batch_shape, [num_timesteps, 1]]) where `sample_shape` corresponds to i.i.d. observations, and the trailing [1] dimension may (optionally) be omitted if `num_timesteps > 1`. May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations.

- `model` An instance of `StructuralTimeSeries` representing a time-series model. This represents a joint distribution over time-series and their parameters with batch shape $[b_1, ..., b_N]$.

- `init_batch_shape` Batch shape (list) of initial states to optimize in parallel. Default value: `list()`. (i.e., just run a single optimization).

- `seed` integer to seed the random number generator.

- `name` name prefixed to ops created by this function. Default value: `NULL` (i.e., 'build_factored_variational_loss').

**Details**

This method constructs a loss function for variational inference using the Kullback-Liebler divergence $\text{KL}[q(z) \| p(z|\text{observed\_time\_series})]$, with an approximating family given by independent Normal distributions transformed to the appropriate parameter space for each parameter.
Minimizing this loss (the negative ELBO) maximizes a lower bound on the log model evidence \(-\log p(\text{observed_time_series})\). This is equivalent to the 'mean-field' method implemented in Kucukelbir et al. (2017) and is a standard approach. The resulting posterior approximations are unimodal; they will tend to underestimate posterior uncertainty when the true posterior contains multiple modes (the \(\text{KL}[q||p]\) divergence encourages choosing a single mode) or dependence between variables.

Value

list of:

- variational_loss: float Tensor of shape \(\text{tf}\concat([\text{init\_batch\_shape}, \text{model\_batch\_shape}])\), encoding a stochastic estimate of an upper bound on the negative model evidence \(-\log p(y)\). Minimizing this loss performs variational inference; the gap between the variational bound and the true (generally unknown) model evidence corresponds to the divergence \(\text{KL}[q||p]\) between the approximate and true posterior.

- variational_distributions: a named list giving the approximate posterior for each model parameter. The keys are character parameter names in order, corresponding to \([\text{param\_name for param in model\_parameters}]\). The values are \(\text{tfd\_Distribution}\) instances with batch shape \(\text{tf}\concat([\text{init\_batch\_shape}, \text{model\_batch\_shape}])\); these will typically be of the form \(\text{tfd\_TransformedDistribution}(\text{tfd\_Normal}(\ldots), \text{bijector=param\_bijector})\).

References


See Also

Other \texttt{sts}-functions: \texttt{sts\_build\_factored\_surrogate\_posterior()}, \texttt{sts\_decompose\_by\_component()}, \texttt{sts\_decompose\_forecast\_by\_component()}, \texttt{sts\_fit\_with\_hmc()}, \texttt{sts\_forecast()}, \texttt{sts\_one\_step\_predictive()}, \texttt{sts\_sample\_uniform\_initial\_state()}

\texttt{sts\_constrained\_seasonal\_state\_space\_model}

\textit{Seasonal state space model with effects constrained to sum to zero.}

Description

Seasonal state space model with effects constrained to sum to zero.

Usage

\texttt{sts\_constrained\_seasonal\_state\_space\_model(}
num\_timesteps,
num\_seasons,
drift\_scale,
initial\_state\_prior,
observation\_noise\_scale = 1e\_\text{\textendash}04,
)
```
num_steps_per_season = 1,
initial_step = 0,
validate_args = FALSE,
allow_nan_stats = TRUE,
name = NULL
```

**Arguments**

- `num_timesteps` Scalar integer tensor number of timesteps to model with this distribution.
- `num_seasons` Scalar integer number of seasons.
- `drift_scale` Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the change in effect between consecutive occurrences of a given season. This is assumed to be the same for all seasons.
- `initial_state_prior` instance of `tfd_multivariate_normal` representing the prior distribution on latent states; must have event shape `[num_seasons]`.
- `observation_noise_scale` Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise.
- `num_steps_per_season` integer number of steps in each season. This may be either a scalar (shape []), in which case all seasons have the same length, or an array of shape `[num_seasons]`, in which seasons have different length, but remain constant around different cycles, or an array of shape `[num_cycles, num_seasons]`, in which `num_steps_per_season` for each season also varies in different cycle (e.g., a 4 years cycle with leap day). Default value: 1.
- `initial_step` Optional scalar integer tensor specifying the starting timestep. Default value: 0.
- `validate_args` logical. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed. Default value: FALSE.
- `allow_nan_stats` logical. If FALSE, raise an exception if a statistic (e.g. mean/mode/etc..) is undefined for any batch member. If TRUE, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: TRUE.
- `name` string prefixed to ops created by this class. Default value: "SeasonalStateSpaceModel".

**Value**

an instance of `LinearGaussianStateSpaceModel`.

---

**sts_constrained_seasonal_state_space_model**

105
See Also

sts_seasonal_state_space_model().

Mathematical details

The constrained model implements a reparameterization of the naive SeasonalStateSpaceModel. Instead of directly representing the seasonal effects in the latent space, the latent space of the constrained model represents the difference between each effect and the mean effect. The following discussion assumes familiarity with the mathematical details of SeasonalStateSpaceModel.

Reparameterization and constraints: let the seasonal effects at a given timestep be \( E = [e_1, ..., e_N] \). The difference between each effect \( e_i \) and the mean effect is \( z_i = e_i - \frac{1}{N} \sum e_i / N \). By itself, this transformation is not invertible because recovering the absolute effects requires that we know the mean as well. To fix this, we'll define \( z_N = \frac{1}{N} \sum e_i / N \) as the mean effect. It's easy to see that this is invertible: given the mean effect and the differences of the first \( N-1 \) effects from the mean, it's easy to solve for all \( N \) effects. Formally, we've defined the invertible linear reparameterization \( Z = R E \), where

\[
R = \begin{bmatrix} 1 - 1/N, & -1/N, & \ldots, & -1/N \\
-1/N, & 1 - 1/N, & \ldots, & -1/N \\
\vdots & \vdots & \ddots & \vdots \\
1/N, & 1/N, & \ldots, & 1/N \\
\end{bmatrix}
\]

represents the change of basis from 'effect coordinates' \( E \) to 'residual coordinates' \( Z \). The \( Z \)'s form the latent space of the ConstrainedSeasonalStateSpaceModel. To constrain the mean effect \( z_N \) to zero, we fix the prior to \( p(z_N) \sim N(0, \theta) \), and after the transition at each timestep we project \( z_N \) back to zero. Note that this projection is linear: to set the \( N \)th dimension to zero, we simply multiply by the identity matrix with a missing element in the bottom right, i.e., \( Z_{\text{constrained}} = P Z \), where \( P = \text{eye}(N) - \text{scatter}((N-1, N-1), 1) \).

Model: concretely, suppose a naive seasonal effect model has initial state prior \( N(m, S) \), transition matrix \( F \) and noise covariance \( Q \), and observation matrix \( H \). Then the corresponding constrained seasonal effect model has initial state prior \( N(P R m, P R S R' P') \), transition matrix \( P R F R^{-1} \) and noise covariance \( F R Q R' F \), and observation matrix \( H R^{-1} \), where the change-of-basis matrix \( R \) and constraint projection matrix \( P \) are as defined above. This follows directly from applying the reparameterization \( Z = R E \), and then enforcing the zero-sum constraint on the prior and transition noise covariances. In practice, because the sum of effects \( z_N \) is constrained to be zero, it will never contribute a term to any linear operation on the latent space, so we can drop that dimension from the model entirely. ConstrainedSeasonalStateSpaceModel does this, so that it implements the \( N-1 \) dimension latent space \( z_1, ..., z_{N-1} \). Note that since we constrained the mean effect to be zero, the latent \( z_i \)'s now recover their interpretation as the actual effects, \( z_i = e_i \) for \( i = 1, ..., N-1 \), even though they were originally defined as residuals. The \( N \)th effect is represented only implicitly, as the nonzero mean of the first \( N-1 \) effects. Although the computational representation is not symmetric across all \( N \) effects, we derived the ConstrainedSeasonalStateSpaceModel by starting with a symmetric representation and imposing only a symmetric constraint (the zero-sum constraint), so the probability model remains symmetric over all \( N \) seasonal effects.

Other sts: sts_additive_state_space_model(), sts_ar_state_space_model(),
sts_ar(), sts_dlr_state_space_model(), sts_dlr(),
sts_lrt().
sts_decompose_by_component

Decompose an observed time series into contributions from each component.

Description

This method decomposes a time series according to the posterior representation of a structural time series model. In particular, it:

- Computes the posterior marginal mean and covariances over the additive model’s latent space.
- Decomposes the latent posterior into the marginal blocks for each model component.
- Maps the per-component latent posteriors back through each component’s observation model, to generate the time series modeled by that component.

Usage

sts_decompose_by_component(observed_time_series, model, parameter_samples)

Arguments

- observed_time_series
  float tensor of shape concat([sample_shape, model.batch_shape, [num_timesteps, 1]])
  where sample_shape corresponds to i.i.d. observations, and the trailing [1] dimension may (optionally) be omitted if num_timesteps > 1. May optionally be an instance of sts_masked_time_series, which includes a mask tensor to specify timesteps with missing observations.

- model
  An instance of sts_sum representing a structural time series model.

- parameter_samples
  list of tensors representing posterior samples of model parameters, with shapes list(tf$concat(list(list(num_posterior_draws), param<1>$prior$batch_shape, param<1>$prior$event_shape), list(list(num_posterior_draws), param<2>$prior$batch_shape, param<2>$prior$event_shape), ... ) ) for all model parameters. This may optionally also be a named list mapping parameter names to tensor values.

Value

- component_dists A named list mapping component StructuralTimeSeries instances (elements of model$components) to Distribution instances representing the posterior marginal distributions on the process modeled by each component. Each distribution has batch shape matching that of posterior_means/posterior_covs, and event shape of list(num_timesteps).
See Also

Other sts-functions: `sts_build_factored_surrogate_posterior()`, `sts_build_factored_variational_loss()`, `sts_decompose_forecast_by_component()`, `sts_fit_with_hmc()`, `sts_forecast()`, `sts_one_step_predictive()`, `sts_sample_uniform_initial_state()`

Examples

```r
observed_time_series <- array(rnorm(2 * 1 * 12), dim = c(2, 1, 12))
day_of_week <- observed_time_series %>% sts_seasonal(num_seasons = 7, name = "seasonal")
local_linear_trend <- observed_time_series %>% sts_local_linear_trend(name = "local_linear")
model <- observed_time_series %>%
  sts_sum(components = list(day_of_week, local_linear_trend))
states_and_results <- observed_time_series %>%
  sts_fit_with_hmc(
    model,
    num_results = 10,
    num_warmup_steps = 5,
    num_variational_steps = 15
  )
samples <- states_and_results[[1]]

component_dists <- observed_time_series %>%
  sts_decompose_by_component(model = model, parameter_samples = samples)
```

---

`sts_decompose_forecast_by_component`

Decompose a forecast distribution into contributions from each component.

**Description**

Decompose a forecast distribution into contributions from each component.

**Usage**

`sts_decompose_forecast_by_component(model, forecast_dist, parameter_samples)`

**Arguments**

- `model`: An instance of `sts_sum` representing a structural time series model.
- `forecast_dist`: A `Distribution` instance returned by `sts_forecast()` (specifically, must be `tfd.MixtureSameFamily` over a `tfd_linear_gaussian_state_space_model` parameterized by posterior samples).
**Parameter samples**

- list of tensors representing posterior samples of model parameters, with shapes list(tf$concat(list(list(num_posterior_draws), param<1>$prior$batch_shape, param<1>$prior$event_shape), list(list(num_posterior_draws), param<2>$prior$batch_shape, param<2>$prior$event_shape), ... ) for all model parameters. This may optionally also be a named list mapping parameter names to tensor values.

**Value**

component_dists A named list mapping component StructuralTimeSeries instances (elements of model$components) to Distribution instances representing the marginal forecast for each component. Each distribution has batch shape matching forecast_dist (specifically, the event shape is [num_steps_forecast]).

**See Also**

Other sts-functions: sts_build_factored_surrogate_posterior(), sts_build_factored_variational_loss(), sts_decompose_by_component(), sts_fit_with_hmc(), sts_forecast(), sts_one_step_predictive(), sts_sample_uniform_initial_state()

---

**sts_dynamic_linear_regression**

*Formal representation of a dynamic linear regression model.*

**Description**

The dynamic linear regression model is a special case of a linear Gaussian SSM and a generalization of typical (static) linear regression. The model represents regression weights with a latent state which evolves via a Gaussian random walk:

**Usage**

```r
sts_dynamic_linear_regression(
  observed_time_series = NULL,
  design_matrix,
  drift_scale_prior = NULL,
  initial_weights_prior = NULL,
  name = NULL
)
```

**Arguments**

- observed_time_series: optional float tensor of shape batch_shape + [T, 1] (omitting the trailing unit dimension is also supported when T > 1), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of sts_masked_time_series, which includes a mask tensor to specify timesteps with missing observations. Default value: NULL.
**design_matrix**  
float tensor of shape `tf$concat(list(batch_shape, list(num_timesteps, num_features)))`. This may also optionally be an instance of `tf$linear$LinearOperator`.

**drift_scale_prior**  
instance of Distribution specifying a prior on the drift_scale parameter. If `NULL`, a heuristic default prior is constructed based on the provided observed_time_series. Default value: `NULL`.

**initial_weights_prior**  
instance of `tfd$multivariate_normal` representing the prior distribution on the latent states (the regression weights). Must have event shape `list(num_features)`. If `NULL`, a weakly-informative Normal(0, 10) prior is used. Default value: `NULL`.

**name**  
the name of this component. Default value: 'DynamicLinearRegression'.

### Details

weights[t] ~ Normal(weights[t-1], drift_scale)

The latent state has dimension `num_features`, while the parameters `drift_scale` and `observation_noise_scale` are each (a batch of) scalars. The batch shape of this distribution is the broadcast batch shape of these parameters, the `initial_state_prior`, and the `design_matrix`. `num_features` is determined from the last dimension of `design_matrix` (equivalent to the number of columns in the design matrix in linear regression).

### Value

an instance of `StructuralTimeSeries`.

### See Also

For usage examples see `sts$fit_with_hmc()`, `sts$forecast()`, `sts$decompose_by_component()`. Other sts: `sts$additive_state_space_model()`, `sts$autoregressive_state_space_model()`, `sts$autoregressive()`, `sts$constrained_seasonal_state_space_model()`, `sts$dynamic_linear_regression_state_space_model()`, `sts$linear_regression()`, `sts$local_level_state_space_model()`, `sts$local_level()`, `sts$local_linear_trend()`, `sts$local_linear_trend()`, `sts$seasonal_state_space_model()`, `sts$seasonal()`, `sts$semi_local_linear_trend()`, `sts$semi_local_linear_trend()`, `sts$smooth_seasonal_state_space_model()`, `sts$smooth_seasonal()`, `sts$sparse_linear_regression()`, `sts$sum()`
specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see `tfd_linear_gaussian_state_space_model` for details.

Usage

```r
sts_dynamic_linear_regression_state_space_model(
  num_timesteps,
  design_matrix,
  drift_scale,
  initial_state_prior,
  observation_noise_scale = 0,
  initial_step = 0,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = NULL
)
```

Arguments

- **num_timesteps**: Scalar integer tensor, number of timesteps to model with this distribution.
- **design_matrix**: float tensor of shape `tf$concat(list(batch_shape,list(num_timesteps,num_features)))`. This may also optionally be an instance of `tf$linalg$LinearOperator`.
- **drift_scale**: Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the latent state transitions.
- **initial_state_prior**: instance of `tfd_multivariate_normal` representing the prior distribution on latent states. Must have event shape `list(num_features)`.
- **observation_noise_scale**: Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise. Default value: 0.
- **initial_step**: scalar integer tensor specifying the starting timestep. Default value: 0.
- **validate_args**: logical. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed. Default value: `FALSE`.
- **allow_nan_stats**: logical. If `FALSE`, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If `TRUE`, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: `TRUE`.
- **name**: name prefixed to ops created by this class. Default value: 'DynamicLinearRegressionStateSpaceModel'.

Details

The dynamic linear regression model is a special case of a linear Gaussian SSM and a generalization of typical (static) linear regression. The model represents regression weights with a latent state which evolves via a Gaussian random walk: `weights[t] ~ Normal(weights[t-1],drift_scale)`
The latent state (the weights) has dimension `num_features`, while the parameters `drift_scale` and `observation_noise_scale` are each (a batch of) scalars. The batch shape of this `Distribution` is the broadcast batch shape of these parameters, the `initial_state_prior`, and the `design_matrix`. `num_features` is determined from the last dimension of `design_matrix` (equivalent to the number of columns in the design matrix in linear regression).

**Mathematical Details**

The dynamic linear regression model implements a `tfd_linear_gaussian_state_space_model` with `latent_size = num_features` and `observation_size = 1` following the transition model:

\[
\text{transition_matrix} = \text{eye}(\text{num_features}) \\
\text{transition_noise} \sim \text{Normal}(0, \text{diag}([\text{drift_scale}]))
\]

which implements the evolution of weights described above. The observation model is:

\[
\text{observation_matrix}[t] = \text{design_matrix}[t] \\
\text{observation_noise} \sim \text{Normal}(0, \text{observation_noise_scale})
\]

**Value**

an instance of `LinearGaussianStateSpaceModel`.

**See Also**

Other `sts`: `sts_additive_state_space_model()`, `sts_autoregressive_state_space_model()`, `sts_autoregressive()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_linear_regression()`, `sts_local_level_state_space_model()`, `sts_local_level()`, `sts_local_linear_trend()`, `sts_local_linear_trend()`, `sts_semi_local_linear_trend()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sparse_linear_regression()`, `sts_sum()

---

**sts_fit_with_hmc**

*Draw posterior samples using Hamiltonian Monte Carlo (HMC)*

**Description**

Markov chain Monte Carlo (MCMC) methods are considered the gold standard of Bayesian inference; under suitable conditions and in the limit of infinitely many draws they generate samples from the true posterior distribution. HMC (Neal, 2011) uses gradients of the model’s log-density function to propose samples, allowing it to exploit posterior geometry. However, it is computationally more expensive than variational inference and relatively sensitive to tuning.
Usage

```python
sts_fit_with_hmc(
    observed_time_series,
    model,
    num_results = 100,
    num_warmup_steps = 50,
    num_leapfrog_steps = 15,
    initial_state = NULL,
    initial_step_size = NULL,
    chain_batch_shape = list(),
    num_variational_steps = 150,
    variational_optimizer = NULL,
    variational_sample_size = 5,
    seed = NULL,
    name = NULL
)
```

Arguments

- **observed_time_series**
  - float tensor of shape `concat([sample_shape, model.batch_shape, [num_timesteps, 1]])` where `sample_shape` corresponds to i.i.d. observations, and the trailing `[1]` dimension may (optionally) be omitted if `num_timesteps > 1`. May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations.

- **model**
  - An instance of `StructuralTimeSeries` representing a time-series model. This represents a joint distribution over time-series and their parameters with batch shape `[b1, ..., bN]`.

- **num_results**
  - Integer number of Markov chain draws. Default value: 100.

- **num_warmup_steps**
  - Integer number of steps to take before starting to collect results. The warmup steps are also used to adapt the step size towards a target acceptance rate of 0.75. Default value: 50.

- **num_leapfrog_steps**
  - Integer number of steps to run the leapfrog integrator for. Total progress per HMC step is roughly proportional to `step_size * num_leapfrog_steps`. Default value: 15.

- **initial_state**
  - Optional Python list of Tensors, one for each model parameter, representing the initial state(s) of the Markov chain(s). These should have shape `tf$concat(list(chain_batch_shape) + prior$batch_shape + prior$event_shape)`.
  - If NULL, the initial state is set automatically using a sample from a variational posterior. Default value: NULL.

- **initial_step_size**
  - list of tensors, one for each model parameter, representing the step size for the leapfrog integrator. Must broadcast with the shape of `initial_state`.
  - Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. If NULL, the step size is set automatically using the standard deviation of a variational posterior. Default value: NULL.
chain_batch_shape

Batch shape (list or int) of chains to run in parallel. Default value: list() (i.e., a single chain).

num_variational_steps

int number of steps to run the variational optimization to determine the initial state and step sizes. Default value: 150.

variational_optimizer

Optional tf$train$Optimizer instance to use in the variational optimization. If NULL, defaults to tf$train$AdamOptimizer(0.1). Default value: NULL.

variational_sample_size

integer number of Monte Carlo samples to use in estimating the variational divergence. Larger values may stabilize the optimization, but at higher cost per step in time and memory. Default value: 1.

seed

integer to seed the random number generator.

name

name prefixed to ops created by this function. Default value: NULL (i.e., 'fit_with_hmc').

Details

This method attempts to provide a sensible default approach for fitting StructuralTimeSeries models using HMC. It first runs variational inference as a fast posterior approximation, and initializes the HMC sampler from the variational posterior, using the posterior standard deviations to set per-variable step sizes (equivalently, a diagonal mass matrix). During the warmup phase, it adapts the step size to target an acceptance rate of 0.75, which is thought to be in the desirable range for optimal mixing (Betancourt et al., 2014).

Value

list of:

- samples: list of Tensors representing posterior samples of model parameters, with shapes [concat([num_results], chain_batch_shape, param.prior.batch_shape, param.prior.event_shape)] for param in model.parameters.

- kernel_results: A (possibly nested) list of Tensors representing internal calculations made within the HMC sampler.

References


- M.J. Betancourt, Simon Byrne, and Mark Girolami. Optimizing The Integrator Step Size for Hamiltonian Monte Carlo.

See Also

Other sts-functions: sts_build_factored_surrogate_posterior(), sts_build_factored_variational_loss(), sts_decompose_by_component(), sts_decompose_forecast_by_component(), sts_forecast(), sts_one_step_predictive(), sts_sample_uniform_initial_state()
Examples

```r
observed_time_series <-
  rep(c(3.5, 4.1, 4.5, 3.9, 2.4, 2.1, 1.2), 5) +
  rep(c(1.1, 1.5, 2.4, 3.1, 4.0), each = 7) %>%
  tensorflow::tf$convert_to_tensor(dtype = tensorflow::tf$float64)

day_of_week <- observed_time_series %>%
  sts_seasonal(num_seasons = 7)

local_linear_trend <- observed_time_series %>%
  sts_local_linear_trend()

model <- observed_time_series %>%
  sts_sum(components = list(day_of_week, local_linear_trend))

states_and_results <- observed_time_series %>%
  sts_fit_with_hmc(
    model,
    num_results = 10,
    num_warmup_steps = 5,
    num_variational_steps = 15)
```

---

**sts_forecast**

*Construct predictive distribution over future observations*

**Description**

Given samples from the posterior over parameters, return the predictive distribution over future observations for `num_steps_forecast` timesteps.

**Usage**

```r
sts_forecast(
  observed_time_series,
  model,
  parameter_samples,
  num_steps_forecast
)
```

**Arguments**

- `observed_time_series` float tensor of shape `concat([sample_shape, model.batch_shape, [num_timesteps, 1]])` where `sample_shape` corresponds to i.i.d. observations, and the trailing `[1]` dimension may (optionally) be omitted if `num_timesteps > 1`. May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations.

- `model` An instance of `StructuralTimeSeries` representing a time-series model. This represents a joint distribution over time-series and their parameters with batch shape `[b1, ..., bN]`.
parameter_samples

list of tensors representing posterior samples of model parameters, with shapes list(tf$concat(list(list(num_posterior_draws), param<1>$prior$batch_shape, param<1>$prior$event_shape), list(list(num_posterior_draws), param<2>$prior$batch_shape, param<2>$prior$event_shape), ... ) ) for all model parameters. This may optionally also be a named list mapping parameter names to tensor values.

num_steps_forecast

scalar integer tensor number of steps to forecast

Value

forecast_dist a tfd_mixture_same_family instance with event shape list(num_steps_forecast,1) and batch shape tf$concat(list(sample_shape,model$batch_shape)), with num_posterior_draws mixture components.

See Also

Other sts-functions: sts_build_factored_surrogate_posterior(), sts_build_factored_variational_loss(), sts_decompose_by_component(), sts_decompose_forecast_by_component(), sts_fit_with_hmc(), sts_one_step_predictive(), sts_sample_uniform_initial_state()

Examples

observed_time_series <-
  rep(c(3.5, 4.1, 4.5, 3.9, 2.4, 2.1, 1.2), 5) +
  rep(c(1.1, 1.5, 2.4, 3.1, 4.0), each = 7) %>%
  tensorflow::tf$convert_to_tensor(dtype = tensorflow::tf$float64)
day_of_week <- observed_time_series %>% sts_seasonal(num_seasons = 7)
local_linear_trend <- observed_time_series %>% sts_local_linear_trend()
model <- observed_time_series %>%
  sts_sum(components = list(day_of_week, local_linear_trend))
states_and_results <- observed_time_series %>%
  sts_fit_with_hmc(model,
  num_results = 10,
  num_warmup_steps = 5,
  num_variational_steps = 15)
samples <- states_and_results[[1]]
preds <- observed_time_series %>%
  sts_forecast(model,
    parameter_samples = samples,
    num_steps_forecast = 50)
predictions <- preds %>% tfd_sample(10)
**sts_linear_regression**  
*Formal representation of a linear regression from provided covariates.*

**Description**

This model defines a time series given by a linear combination of covariate time series provided in a design matrix:

```r
observed_time_series <- tf$matmul(design_matrix, weights)
```

**Usage**

```r
sts_linear_regression(design_matrix, weights_prior = NULL, name = NULL)
```

**Arguments**

- **design_matrix**  
  float tensor of shape `tf$concat(list(batch_shape,list(num_timesteps,num_features)))`. This may also optionally be an instance of `tf$linalg$LinearOperator`.

- **weights_prior**  
  Distribution representing a prior over the regression weights. Must have event shape `list(num_features)` and batch shape broadcastable to the design matrix’s batch_shape. Alternately, event_shape may be scalar (`list()`), in which case the prior is internally broadcast as `tfd_transformed_distribution(weights_prior,tfb_identity(),event_shape = list(num_features),batch_shape = design_matrix$batch_shape)`. If `NULL`, defaults to `tfd_student_t(df = 5,loc = 0,scale = 10)`, a weakly-informative prior loosely inspired by the Stan prior choice recommendations. Default value: `NULL`.

- **name**  
  the name of this model component. Default value: ‘LinearRegression’.

**Details**

The design matrix has shape `list(num_timesteps,num_features)`. The weights are treated as an unknown random variable of size `list(num_features)` (both components also support batch shape), and are integrated over using the same approximate inference tools as other model parameters, i.e., generally HMC or variational inference.

This component does not itself include observation noise; it defines a deterministic distribution with mass at the point `tf$matmul(design_matrix,weights)`. In practice, it should be combined with observation noise from another component such as `sts_sum`, as demonstrated below.

**Value**

an instance of StructuralTimeSeries.
sts_local_level

Formal representation of a local level model

Description

The local level model posits a level evolving via a Gaussian random walk:

\[
\text{level}[t] = \text{level}[t-1] + \text{Normal}(0., \text{level_scale})
\]

Usage

sts_local_level(
    observed_time_series = NULL,
    level_scale_prior = NULL,
    initial_level_prior = NULL,
    name = NULL
)

Arguments

observed_time_series

optional float tensor of shape batch_shape + [T, 1] (omitting the trailing unit
dimension is also supported when T > 1), specifying an observed time series.
Any priors not explicitly set will be given default values according to the scale
of the observed time series (or batch of time series). May optionally be an in-
stance of sts_masked_time_series, which includes a mask tensor to specify
timesteps with missing observations. Default value: NULL.

level_scale_prior

optional tfp$distribution instance specifying a prior on the level_scale
parameter. If NULL, a heuristic default prior is constructed based on the provided
observed_time_series. Default value: NULL.

initial_level_prior

optional tfp$distribution instance specifying a prior on the initial level. If
NULL, a heuristic default prior is constructed based on the provided observed_time_series.
Default value: NULL.

name

the name of this model component. Default value: 'LocalLevel'.
Details

The latent state is [level]. We observe a noisy realization of the current level: \( f(t) = \text{level}[t] + \text{Normal}(0, \text{observation_noise_scale}) \) at each timestep.

Value

an instance of StructuralTimeSeries.

See Also

For usage examples see \texttt{sts_fit_with_hmc()}, \texttt{sts_forecast()}, \texttt{sts_decompose_by_component()}.

Other \texttt{sts}: \texttt{sts_additive_state_space_model()}, \texttt{sts_autoregressive_state_space_model()}, \texttt{sts_autoregressive()}, \texttt{sts_constrained_seasonal_state_space_model()}, \texttt{sts_dynamic_linear_regression_state_space_model()}, \texttt{sts_dynamic_linear_regression()}, \texttt{sts_linear_regression()}, \texttt{sts_local_level_state_space_model()}, \texttt{sts_local_linear_trend_state_space_model()}, \texttt{sts_local_linear_trend()}, \texttt{sts_seasonal_state_space_model()}, \texttt{sts_seasonal()}, \texttt{sts_semi_local_linear_trend_state_space_model()}, \texttt{sts_semi_local_linear_trend()}, \texttt{sts_smooth_seasonal_state_space_model()}, \texttt{sts_smooth_seasonal()}, \texttt{sts_sparse_linear_regression()}, \texttt{sts_sum()}

\begin{verbatim}
sts_local_level_state_space_model

State space model for a local level
\end{verbatim}

Description

A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with dynamics specified by a probabilistic transition model \( p(z[t+1] | z[t]) \). At each timestep, we observe a value sampled from an observation model conditioned on the current state, \( p(x[t] | z[t]) \). The special case where both the transition and observation models are Gaussians with mean specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see \texttt{tfd_linear_gaussian_state_space_model} for details. The local level model is a special case of a linear Gaussian SSM, in which the latent state posits a level evolving via a Gaussian random walk:

\[
\text{level}[t] = \text{level}[t-1] + \text{Normal}(0, \text{level_scale})
\]

Usage

\begin{verbatim}
sts_local_level_state_space_model(
    num_timesteps,
    level_scale,
    initial_state_prior,
    observation_noise_scale = 0,
    initial_step = 0,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = NULL
)
\end{verbatim}
Arguments

num_timesteps  Scalar integer tensor number of timesteps to model with this distribution.

level_scale  Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the level transitions.

initial_state_prior  instance of tfd_multivariate_normal representing the prior distribution on latent states. Must have event shape [1] (as tfd_linear_gaussian_state_space_model requires a rank-1 event shape).

observation_noise_scale  Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise.

initial_step  Optional scalar integer tensor specifying the starting timestep. Default value: 0.

validate_args  logical. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed. Default value: FALSE.

allow_nan_stats  logical. If FALSE, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If TRUE, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: TRUE.

name  string name prefixed to ops created by this class. Default value: "LocalLevel-StateSpaceModel".

Details

The latent state is [level] and [level] is observed (with noise) at each timestep. The parameters level_scale and observation_noise_scale are each (a batch of) scalars. The batch shape of this Distribution is the broadcast batch shape of these parameters and of the initial_state_prior.

Mathematical Details

The local level model implements a tfp$distributions$LinearGaussianStateSpaceModel with latent_size = 1 and observation_size = 1, following the transition model:

\[
\text{transition_matrix} = [[1]] \\
\text{transition_noise \sim N(loc = 0, scale = diag([level_scale]))}
\]

which implements the evolution of level described above, and the observation model:

\[
\text{observation_matrix} = [[1]] \\
\text{observation_noise \sim N(loc = 0, scale = observation_noise_scale)}
\]

Value

an instance of LinearGaussianStateSpaceModel.
See Also

Other sts: `sts_additive_state_space_model()`, `sts_autoregressive_state_space_model()`, `sts_autoregressive()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_linear_regression()`, `sts_local_level()`, `sts_local_linear_trend()`, `sts_local_linear_trend_state_space_model()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_linear_trend()`, `sts_semi_local_linear_trend_state_space_model()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sparse_linear_regression()`, `sts_sum()

sts_local_linear_trend

Formal representation of a local linear trend model

Description

The local linear trend model posits a level and slope, each evolving via a Gaussian random walk:

\[
\text{level}[t] = \text{level}[t-1] + \text{slope}[t-1] + \text{Normal}(0., \text{level_scale})
\]
\[
\text{slope}[t] = \text{slope}[t-1] + \text{Normal}(0., \text{slope_scale})
\]

Usage

```r
sts_local_linear_trend(
  observed_time_series = NULL,
  level_scale_prior = NULL,
  slope_scale_prior = NULL,
  initial_level_prior = NULL,
  initial_slope_prior = NULL,
  name = NULL
)
```

Arguments

- **observed_time_series**: optional float tensor of shape batch_shape + [T, 1] (omitting the trailing unit dimension is also supported when T > 1), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations. Default value: NULL.

- **level_scale_prior**: optional `tfp$Distribution` instance specifying a prior on the level_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

- **slope_scale_prior**: optional `tfd$Distribution` instance specifying a prior on the slope_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.
initial_level_prior
optional tfp$distribution instance specifying a prior on the initial level. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

initial_slope_prior
optional tfd$Distribution instance specifying a prior on the initial slope. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

ame
the name of this model component. Default value: 'LocalLinearTrend'.

Details
The latent state is the two-dimensional tuple [level, slope]. At each timestep we observe a noisy realization of the current level: \( f[t] = level[t] + \text{Normal}(0., \text{observation_noise_scale}) \). This model is appropriate for data where the trend direction and magnitude (latent slope) is consistent within short periods but may evolve over time.

Note that this model can produce very high uncertainty forecasts, as uncertainty over the slope compounds quickly. If you expect your data to have nonzero long-term trend, i.e. that slopes tend to revert to some mean, then the SemiLocalLinearTrend model may produce sharper forecasts.

Value
an instance of StructuralTimeSeries.

See Also
For usage examples see sts_fit_with_hmc(), sts_forecast(), sts_decompose_by_component().
Other sts: sts_additive_state_space_model(), sts_autoregressive_state_space_model(), sts_autoregressive(), sts_constrained_seasonal_state_space_model(), sts_dynamic_linear_regression_state_space_model(), sts_dynamic_linear_regression(), sts_linear_regression(), sts_local_level_state_space_model(), sts_local_level(), sts_local_linear_trend_state_space_model(), sts_seasonal_state_space_model(), sts_seasonal(), sts_semi_local_linear_trend_state_space_model(), sts_semi_local_linear_trend(), sts_smooth_seasonal_state_space_model(), sts_smooth_seasonal(), sts_sparse_linear_regression(), sts_sum()
Usage

sts_local_linear_trend_state_space_model(
    num_timesteps,
    level_scale,
    slope_scale,
    initial_state_prior,
    observation_noise_scale = 0,
    initial_step = 0,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = NULL
)

Arguments

num_timesteps  Scalar integer tensor number of timesteps to model with this distribution.
level_scale    Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the level transitions.
slope_scale    Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the slope transitions.
initial_state_prior
    instance of tfd_multivariate_normal representing the prior distribution on latent states. Must have event shape [1] (as tfd_linear_gaussian_state_space_model requires a rank-1 event shape).
observation_noise_scale
    Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise.
initial_step    Optional scalar integer tensor specifying the starting timestep. Default value: 0.
validate_args   logical. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed. Default value: FALSE.
allow_nan_stats
    logical. If FALSE, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If TRUE, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: TRUE.
name
    string prefixed to ops created by this class. Default value: "LocalLinearTrendStateSpaceModel".

Details

The local linear trend model is a special case of a linear Gaussian SSM, in which the latent state posits a level and slope, each evolving via a Gaussian random walk:

level[t] = level[t-1] + slope[t-1] + Normal(0., level_scale)
slope[t] = slope[t-1] + Normal(0., slope_scale)
The latent state is the two-dimensional tuple [level, slope]. The level is observed at each timestep. The parameters level_scale, slope_scale, and observation_noise_scale are each (a batch of) scalars. The batch shape of this Distribution is the broadcast batch shape of these parameters and of the initial_state_prior.

Mathematical Details

The linear trend model implements a `tfd_linear_gaussian_state_space_model` with latent_size = 2 and observation_size = 1, following the transition model:

\[
\text{transition_matrix} = \begin{bmatrix} 1. & 1. \\ 0. & 1. \end{bmatrix}
\]

\[
\text{transition_noise} \sim \text{N(loc = 0, scale = diag([level_scale, slope_scale])})
\]

which implements the evolution of [level, slope] described above, and the observation model:

\[
\text{observation_matrix} = \begin{bmatrix} 1. & 0. \end{bmatrix}
\]

\[
\text{observation_noise} \sim \text{N(loc= 0 , scale = observation_noise_scale)}
\]

which picks out the first latent component, i.e., the level, as the observation at each timestep.

Value

an instance of `LinearGaussianStateSpaceModel`.

See Also

Other sts: `sts_additive_state_space_model()`, `sts_autoregressive_state_space_model()`, `sts_autoregressive()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_linear_regression()`, `sts_local_level_state_space_model()`, `sts_local_level()`, `sts_local_linear_trend()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_linear_trend_state_space_model()`, `sts_semi_local_linear_trend()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_semi_local_linear_trend_state_space_model()`, `sts_semi_local_linear_trend()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sparse_linear_regression()`, `sts_sum()`

---

`sts_one_step_predictive`

Compute one-step-ahead predictive distributions for all timesteps

Description

Given samples from the posterior over parameters, return the predictive distribution over observations at each time T, given observations up through time T-1.

Usage

`sts_one_step_predictive(observed_time_series, model, parameter_samples)`
Arguments

observed_time_series

float tensor of shape concat([sample_shape, model.batch_shape, [num_timesteps, 1]])
where sample_shape corresponds to i.i.d. observations, and the trailing [1] di-

model

An instance of StructuralTimeSeries representing a time-series model. This

param

list of tensors representing posterior samples of model parameters, with

Value

forecast_dist a tfd_mixture_same_family instance with event shape list(num_timesteps) and

See Also

Other sts-functions: sts_build_factored_surrogate_posterior(), sts_build_factored_variational_loss(),
sts_decompose_by_component(), sts_decompose_forecast_by_component(), sts_fit_with_hmc(),
sts_forecast(), sts_sample_uniform_initial_state()
**sts_seasonal**

*Formal representation of a seasonal effect model.*

**Description**

A seasonal effect model posits a fixed set of recurring, discrete 'seasons', each of which is active for a fixed number of timesteps and, while active, contributes a different effect to the time series. These are generally not meteorological seasons, but represent regular recurring patterns such as hour-of-day or day-of-week effects. Each season lasts for a fixed number of timesteps. The effect of each season drifts from one occurrence to the next following a Gaussian random walk:

**Usage**

```r
sts_seasonal(
  observed_time_series = NULL,
  num_seasons,
  num_steps_per_season = 1,
  drift_scale_prior = NULL,
  initial_effect_prior = NULL,
  constrain_mean_effect_to_zero = TRUE,
  name = NULL
)
```

**Arguments**

- `parameter`: `sts$Parameter` named tuple instance.
- `return_constrained`: if `TRUE`, re-applies the constraining bijector to return initializations in the original domain. Otherwise, returns initializations in the unconstrained space. Default value: `TRUE`.
- `init_sample_shape`: sample_shape of the sampled initializations. Default value: `list()`.
- `seed`: integer to seed the random number generator.

**Value**

`uniform_initializer` Tensor of shape `concat([init_sample_shape, parameter.prior.batch_shape, transformed_event_shape])`, where `transformed_event_shape` is `parameter.prior.event_shape`, if `return_constrained=TRUE`, and otherwise it is `parameter$bijector$inverse_event_shape(parameter$prior$event_shape)`.

**See Also**

Other `sts`-functions: `sts_build_factored_surrogate_posterior()`, `sts_build_factored_variational_loss()`, `sts_decompose_by_component()`, `sts_decompose_forecast_by_component()`, `sts_fit_with_hmc()`, `sts_forecast()`, `sts_one_step_predictive()`
Arguments

observed_time_series
optional float tensor of shape batch_shape + [T, 1] (omitting the trailing unit dimension is also supported when T > 1), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of sts_masked_time_series, which includes a mask tensor to specify timesteps with missing observations. Default value: NULL.

num_seasons
Scalar integer number of seasons.

drift_scale_prior
optional tfd$Distribution instance specifying a prior on the drift_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

initial_effect_prior
optional tfd$Distribution instance specifying a normal prior on the initial effect of each season. This may be either a scalar tfd_normal prior, in which case it applies independently to every season, or it may be multivariate normal (e.g., tfd_multivariate_normal_diag) with event shape [num_seasons], in which case it specifies a joint prior across all seasons. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

constrain_mean_effect_to_zero
if TRUE, use a model parameterization that constrains the mean effect across all seasons to be zero. This constraint is generally helpful in identifying the contributions of different model components and can lead to more interpretable posterior decompositions. It may be undesirable if you plan to directly examine the latent space of the underlying state space model. Default value: TRUE.

name
the name of this model component. Default value: ‘Seasonal’.

Details

effects[season, occurrence[i]] = (effects[season, occurrence[i-1]] + Normal(loc=0., scale=drift_scale))

The drift_scale parameter governs the standard deviation of the random walk; for example, in a day-of-week model it governs the change in effect from this Monday to next Monday.

Value

an instance of StructuralTimeSeries.
State space model for a seasonal effect.

Description

A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with
dynamics specified by a probabilistic transition model \( p(z[t+1] | z[t]) \). At each timestep, we
observe a value sampled from an observation model conditioned on the current state, \( p(x[t] | z[t]) \). The special case where both the transition and observation models are Gaussians with mean
specified as a linear function of the inputs, is known as a linear Gaussian state space model and sup-
ports tractable exact probabilistic calculations; see `tfd_linear_gaussian_state_space_model`
for details.

Usage

```r
sts_seasonal_state_space_model(
  num_timesteps,
  num_seasons,
  drift_scale,
  initial_state_prior,
  observation_noise_scale = 0,
  num_steps_per_season = 1,
  initial_step = 0,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = NULL
)
```

Arguments

- `num_timesteps` Scalar integer tensor number of timesteps to model with this distribution.
- `num_seasons` Scalar integer number of seasons.
- `drift_scale` Scalar (any additional dimensions are treated as batch dimensions) float tensor
indicating the standard deviation of the change in effect between consecutive oc-
currences of a given season. This is assumed to be the same for all seasons.

See Also

For usage examples see `sts_fit_with_hmc(), sts_forecast(), sts_decompose_by_component()`.
Other `sts`: `sts_additive_state_space_model(), sts_autoregressive_state_space_model(),
sts_autoregressive(), sts_constrained_seasonal_state_space_model(), sts_dynamic_linear_regression_state_space_model(),
sts_dynamic_linear_regression(), sts_linear_regression(),
sts_local_level_state_space_model(),
sts_local_level(),
sts_local_linear_trend_state_space_model(),
sts_local_linear_trend(),
sts_seasonal_state_space_model(),
sts_semi_local_linear_trend_state_space_model(),
sts_semi_local_linear_trend(),
sts_smooth_seasonal_state_space_model(),
sts_smooth_seasonal(),
sts_semi_local_linear_trend(),
sts_smooth_seasonal_state_space_model(),
sts_smooth_seasonal().

sts_seasonal_state_space_model

State space model for a seasonal effect.
initial_state_prior

instance of tfd_multivariate_normal representing the prior distribution on latent states; must have event shape [num_seasons].

observation_noise_scale

Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise.

num_steps_per_season

integer number of steps in each season. This may be either a scalar (shape []), in which case all seasons have the same length, or an array of shape [num_seasons], in which seasons have different length, but remain constant around different cycles, or an array of shape [num_cycles, num_seasons], in which num_steps_per_season for each season also varies in different cycle (e.g., a 4 years cycle with leap day). Default value: 1.

initial_step

Optional scalar integer tensor specifying the starting timestep. Default value: 0.

validate_args

logical. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed. Default value: FALSE.

allow_nan_stats

logical. If FALSE, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If TRUE, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: TRUE.

name

string prefixed to ops created by this class. Default value: "SeasonalStateSpaceModel".

Details

A seasonal effect model is a special case of a linear Gaussian SSM. The latent states represent an unknown effect from each of several 'seasons'; these are generally not meteorological seasons, but represent regular recurring patterns such as hour-of-day or day-of-week effects. The effect of each season drifts from one occurrence to the next, following a Gaussian random walk:

effects[season, occurrence[i]] = (effects[season, occurrence[i-1]] + Normal(loc=0., scale=drift_scale))

The latent state has dimension num_seasons, containing one effect for each seasonal component. The parameters drift_scale and observation_noise_scale are each (a batch of) scalars. The batch shape of this Distribution is the broadcast batch shape of these parameters and of the initial_state_prior. Note: there is no requirement that the effects sum to zero.

Mathematical Details

The seasonal effect model implements a tfd_linear_gaussian_state_space_model with latent_size = num_seasons and observation_size = 1. The latent state is organized so that the current seasonal effect is always in the first (zeroth) dimension. The transition model rotates the latent state to shift to a new effect at the end of each season:

transition_matrix[t] = (permutation_matrix([1, 2, ..., num_seasons-1, 0]))

if season_is_changing(t)
else eye(num_seasons)
transition_noise[t] ~ Normal(loc=0., scale_diag=
    [drift_scale, 0, ..., 0]
    if season_is_changing(t)
    else [0, 0, ..., 0])

where season_is_changing(t) is True if t `mod` sum(num_steps_per_season) is in the set of final days for each season, given by cumsum(num_steps_per_season) - 1. The observation model always picks out the effect for the current season, i.e., the first element of the latent state:

observation_matrix = [[1., 0., ..., 0.]]
o

Value

an instance of LinearGaussianStateSpaceModel.

See Also

Other sts: sts_additive_state_space_model(), sts_autoregressive_state_space_model().
mts_autoregressive(), sts_constrained_seasonal_state_space_model(), sts_dynamic_linear_regression_state_space_model(),
sts_dynamic_linear_regression(), sts_linear_regression().sts_local_level_state_space_model().
sts_local_level(), sts_local_linear_trend_state_space_model(), sts_local_linear_trend().
sts_seasonal(), sts_semi_local_linear_trend_state_space_model(), sts_semi_local_linear_trend().
sts_smooth_seasonal_state_space_model(), sts_smooth_seasonal().sts_sparse_linear_regression().
sts_sum()
Arguments

- **observed_time_series**
  optional float tensor of shape `batch_shape + [T, 1]` (omitting the trailing unit dimension is also supported when `T > 1`), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations. Default value: NULL.

- **level_scale_prior**
  optional `tfp$distribution` instance specifying a prior on the `level_scale` parameter. If NULL, a heuristic default prior is constructed based on the provided `observed_time_series`. Default value: NULL.

- **slope_mean_prior**
  optional `tfd$Distribution` instance specifying a prior on the `slope_mean` parameter. If NULL, a heuristic default prior is constructed based on the provided `observed_time_series`. Default value: NULL.

- **slope_scale_prior**
  optional `tfd$Distribution` instance specifying a prior on the `slope_scale` parameter. If NULL, a heuristic default prior is constructed based on the provided `observed_time_series`. Default value: NULL.

- **autoregressive_coef_prior**
  optional `tfd$Distribution` instance specifying a prior on the `autoregressive_coef` parameter. If NULL, the default prior is a standard Normal(0, 1). Note that the prior may be implicitly truncated by `constrain_ar_coef_stationary` and/or `constrain_ar_coef_positive`. Default value: NULL.

- **initial_level_prior**
  optional `tfp$distribution` instance specifying a prior on the initial level. If NULL, a heuristic default prior is constructed based on the provided `observed_time_series`. Default value: NULL.

- **initial_slope_prior**
  optional `tfd$Distribution` instance specifying a prior on the initial slope. If NULL, a heuristic default prior is constructed based on the provided `observed_time_series`. Default value: NULL.

- **constrain_ar_coef_stationary**
  if TRUE, perform inference using a parameterization that restricts `autoregressive_coef` to the interval (-1, 1), or (0, 1) if `force_positive_ar_coef` is also TRUE, corresponding to stationary processes. This will implicitly truncate the support of `autoregressive_coef_prior`. Default value: TRUE.

- **constrain_ar_coef_positive**
  if TRUE, perform inference using a parameterization that restricts `autoregressive_coef` to be positive, or in (0, 1) if `constrain_ar_coef_stationary` is also TRUE. This will implicitly truncate the support of `autoregressive_coef_prior`. Default value: FALSE.

- **name**
  the name of this model component. Default value: 'SemiLocalLinearTrend'.

Details

\[
\text{level}[t] = \text{level}[t-1] + \text{slope}[t-1] + \text{Normal}(0., \text{level_scale})
\]
The slope component in a \texttt{sts_semi_local_linear_trend} model evolves according to a first-order autoregressive (AR1) process with potentially nonzero mean:

\[ \text{slope}[t] = (\text{slope}\_\text{mean} + \text{autoregressive}\_\text{coef} \times (\text{slope}[t-1] - \text{slope}\_\text{mean}) + \text{Normal}(0., \text{slope}\_\text{scale})) \]

Unlike the random walk used in \texttt{LocalLinearTrend}, a stationary AR1 process (coefficient in (-1, 1)) maintains bounded variance over time, so a \texttt{SemiLocalLinearTrend} model will often produce more reasonable uncertainties when forecasting over long timescales.

\textbf{Value}

an instance of \texttt{StructuralTimeSeries}.

\textbf{See Also}

For usage examples see \texttt{sts_fit_with_hmc()}, \texttt{sts_forecast()}, \texttt{sts_decompose_by_component()}. Other \texttt{sts}: \texttt{sts_additive_state_space_model()}, \texttt{sts_autoregressive_state_space_model()}, \texttt{sts_autoregressive()}, \texttt{sts_constrained_seasonal_state_space_model()}, \texttt{sts_dynamic_linear_regression_state_space_model()}, \texttt{sts_dynamic_linear_regression()}, \texttt{sts_local_level_state_space_model()}, \texttt{sts_local_level()}, \texttt{sts_local_linear_trend_state_space_model()}, \texttt{sts_local_linear_trend()}, \texttt{sts_seasonal_state_space_model()}, \texttt{sts_seasonal()}, \texttt{sts_semi_local_linear_trend_state_space_model()}, \texttt{sts_smooth_seasonal_state_space_model()}, \texttt{sts_smooth_seasonal()}, \texttt{sts_sparse_linear_regression()}, \texttt{sts_sum()}

\begin{verbatim}
sts_semi_local_linear_trend_state_space_model

State space model for a semi-local linear trend.
\end{verbatim}

\textbf{Description}

A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with dynamics specified by a probabilistic transition model \( p(z[t+1] | z[t]) \). At each timestep, we observe a value sampled from an observation model conditioned on the current state, \( p(x[t] | z[t]) \). The special case where both the transition and observation models are Gaussians with mean specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see \texttt{tfd_linear_gaussian_state_space_model} for details.

\textbf{Usage}

\begin{verbatim}
sts_semi_local_linear_trend_state_space_model(
    num_timesteps,
    level_scale,
    slope_mean,
    slope_scale,
    autoregressive_coef,
    initial_state_prior,
\end{verbatim}


observation_noise_scale = 0,
initial_step = 0,
validate_args = FALSE,
allow_nan_stats = TRUE,
name = NULL)

Arguments

num_timesteps  Scalar integer tensor number of timesteps to model with this distribution.
level_scale  Scalar (any additional dimensions are treated as batch dimensions) float tensor
indicating the standard deviation of the level transitions.
slope_mean  Scalar (any additional dimensions are treated as batch dimensions) float tensor
indicating the expected long-term mean of the latent slope.
slope_scale  Scalar (any additional dimensions are treated as batch dimensions) float tensor
indicating the standard deviation of the slope transitions.
autoregressive_coef
Scalar (any additional dimensions are treated as batch dimensions) float tensor
defining the AR1 process on the latent slope.
initial_state_prior
instance of tfd_multivariate_normal representing the prior distribution on
latent states. Must have event shape [1] (as tfd_linear_gaussian_state_space_model
requires a rank-1 event shape).
observation_noise_scale
Scalar (any additional dimensions are treated as batch dimensions) float tensor
indicating the standard deviation of the observation noise.
initial_step
Optional scalar integer tensor specifying the starting timestep. Default value: 0.
validate_args
logical. Whether to validate input with asserts. If validate_args is FALSE,
and the inputs are invalid, correct behavior is not guaranteed. Default value: FALSE.
allow_nan_stats
logical. If FALSE, raise an exception if a statistic (e.g. mean/mode/etc...) is
undefined for any batch member. If TRUE, batch members with valid parameters
leading to undefined statistics will return NaN for this statistic. Default value: TRUE.
name
string' prefixed to ops created by this class. Default value: "SemiLocalLinearTrend-
StateSpaceModel".

Details

The semi-local linear trend model is a special case of a linear Gaussian SSM, in which the latent
state posits a level and slope. The level evolves via a Gaussian random walk centered at
the current slope, while the slope follows a first-order autoregressive (AR1) process with mean
slope_mean:
level[t] = level[t-1] + slope[t-1] + Normal(0, level_scale)
slope[t] = (slope_mean + autoregressive_coef * (slope[t-1] - slope_mean) +
        Normal(0., slope_scale))

The latent state is the two-dimensional tuple [level, slope]. The level is observed at each timestep.
The parameters level_scale, slope_mean, slope_scale, autoregressive_coef, and observation_noise_scale
are each (a batch of) scalars. The batch shape of this Distribution is the broadcast batch shape of
these parameters and of the initial_state_prior.

Mathematical Details

The semi-local linear trend model implements a tfp.distributions.LinearGaussianStateSpaceModel
with latent_size = 2 and observation_size = 1, following the transition model:

transition_matrix = [[1., 1.]
                      [0., autoregressive_coef]]
transition_noise ~ N(loc=slope_mean - autoregressive_coef * slope_mean,
                      scale=diag([level_scale, slope_scale]))

which implements the evolution of [level, slope] described above, and the observation model:

observation_matrix = [[1., 0.]]
observation_noise ~ N(loc=0, scale=observation_noise_scale)

which picks out the first latent component, i.e., the level, as the observation at each timestep.

Value

an instance of LinearGaussianStateSpaceModel.

See Also

Other sts: sts_additive_state_space_model(), sts_autoregressive_state_space_model(),
sts_autoregressive(), sts_constrained_seasonal_state_space_model(), sts_dynamic_linear_regression_state_space_model(),
sts_dynamic_linear_regression(), sts_linear_regression(), sts_local_level_state_space_model(),
sts_local_level(), sts_local_linear_trend_state_space_model(), sts_local_linear_trend(),
sts_seasonal_state_space_model(), sts_seasonal(), sts_semi_local_linear_trend(), sts_smooth_seasonal_state_space_model(),
sts_smooth_seasonal(), sts_sparse_linear_regression(), sts_sum()
sts_smooth_seasonal

Usage

sts_smooth_seasonal(
  period,
  frequency_multipliers,
  allow_drift = TRUE,
  drift_scale_prior = NULL,
  initial_state_prior = NULL,
  observed_time_series = NULL,
  name = NULL
)

Arguments

period positive scalar float Tensor giving the number of timesteps required for the longest cyclic effect to repeat.

frequency_multipliers One-dimensional float Tensor listing the frequencies (cyclic components) included in the model, as multipliers of the base/fundamental frequency 2. * pi / period. Each component is specified by the number of times it repeats per period, and adds two latent dimensions to the model. A smooth seasonal model that can represent any periodic function is given by frequency_multipliers = [1,2,...,floor(period / 2)]. However, it is often desirable to enforce a smoothness assumption (and reduce the computational burden) by dropping some of the higher frequencies.

allow_drift optional logical specifying whether the seasonal effects can drift over time. Setting this to FALSE removes the drift_scale parameter from the model. This is mathematically equivalent to drift_scale_prior = tfd.Deterministic(0.), but removing drift directly is preferred because it avoids the use of a degenerate prior. Default value: TRUE.

drift_scale_prior optional tfd$Distribution instance specifying a prior on the drift_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

initial_state_prior instance of tfd$MultivariateNormal representing the prior distribution on the latent states. Must have event shape [2 * len(frequency_multipliers)]. If NULL, a heuristic default prior is constructed based on the provided observed_time_series.

observed_time_series optional float Tensor of shape batch_shape + [T, 1] (omitting the trailing unit dimension is also supported when T > 1), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of tfp$sts$MaskedTimeSeries, which includes a mask Tensor to specify timesteps with missing observations. Default value: NULL.

name the name of this model component. Default value: 'LocalLinearTrend'.

Details

frequencies[j] = 2. * pi * frequency_multipliers[j] / period
and then posits two latent states for each frequency. The two latent states associated with frequency \( j \) drift over time via:

\[
\begin{align*}
effect[t] &= (effect[t-1] \times \cos(frequencies[j]) + \\
&\quad auxiliary[t-1] \times \sin(frequencies[j]) + \\
&\quad \text{Normal}(0., drift_scale)) \\
\text{auxiliary}[t] &= (-effect[t-1] \times \sin(frequencies[j]) + \\
&\quad auxiliary[t-1] \times \cos(frequencies[j]) + \\
&\quad \text{Normal}(0., drift_scale))
\end{align*}
\]

where \( effect \) is the smooth seasonal effect and \( auxiliary \) only appears as a matter of construction. The interpretation of \( auxiliary \) is thus not particularly important.

**Value**

An instance of \texttt{StructuralTimeSeries}.

**See Also**

For usage examples see \texttt{sts_fit_with_hmc()}, \texttt{sts_forecast()}, \texttt{sts_decompose_by_component()}.

Other \texttt{sts}: \texttt{sts_additive_state_space_model()}, \texttt{sts_autoregressive_state_space_model()}, \texttt{sts_autoregressive()}, \texttt{sts_constrained_seasonal_state_space_model()}, \texttt{sts_dynamic_linear_regression_state_space_model()}, \texttt{sts_dynamic_linear_regression()}, \texttt{sts_local_level_state_space_model()}, \texttt{sts_local_level()}, \texttt{sts_local_linear_trend_state_space_model()}, \texttt{sts_local_linear_trend()}, \texttt{sts_seasonal_state_space_model()}, \texttt{sts_seasonal()}, \texttt{sts_semi_local_linear_trend_state_space_model()}, \texttt{sts_semi_local_linear_trend()}, \texttt{sts_smooth_seasonal_state_space_model()}, \texttt{sts_sparse_linear_regression()}, \texttt{sts_sum()}

---

**Description**

A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with dynamics specified by a probabilistic transition model \( p(z[t+1] \mid z[t]) \). At each timestep, we observe a value sampled from an observation model conditioned on the current state, \( p(x[t] \mid z[t]) \). The special case where both the transition and observation models are Gaussians with mean specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see \texttt{tfp$\text{distributions}$LinearGaussianStateSpaceModel} for details. A smooth seasonal effect model is a special case of a linear Gaussian SSM. It is the sum of a set of "cyclic" components, with one component for each frequency:

\[
frequencies[j] = 2. \times \pi \times frequency\_multipliers[j] / period
\]

Each cyclic component contains two latent states which we denote \( effect \) and \( auxiliary \). The two latent states for component \( j \) drift over time via:
\[
effect[t] = (\effect[t-1] \times \cos(\text{frequencies}[j]) + \\auxiliary[t-1] \times \sin(\text{frequencies}[j]) + \\
\text{Normal}(0, \text{drift_scale})
\]
\[
\auxiliary[t] = (-\effect[t-1] \times \sin(\text{frequencies}[j]) + \\auxiliary[t-1] \times \cos(\text{frequencies}[j]) + \\
\text{Normal}(0, \text{drift_scale})
\]

Usage

\text{sts\_smooth\_seasonal\_state\_space\_model}(\\num\_timesteps, \\
period, \\
frequency\_multipliers, \\
drift\_scale, \\
initial\_state\_prior, \\
observation\_noise\_scale = 0, \\
initial\_step = 0, \\
validate\_args = \text{FALSE}, \\
allow\_nan\_stats = \text{TRUE}, \\
name = \text{NULL})

Arguments

\text{num\_timesteps} \quad \text{Scalar integer Tensor number of timesteps to model with this distribution.}

\text{period} \quad \text{positive scalar float Tensor giving the number of timesteps required for the longest cyclic effect to repeat.}

\text{frequency\_multipliers} \quad \text{One-dimensional float Tensor listing the frequencies (cyclic components) included in the model, as multipliers of the base/fundamental frequency } 2 \times \pi / \text{period}. \text{ Each component is specified by the number of times it repeats per period, and adds two latent dimensions to the model. A smooth seasonal model that can represent any periodic function is given by frequency\_multipliers = \{1,2, \ldots, \text{floor(period / 2)}\}. However, it is often desirable to enforce a smoothness assumption (and reduce the computational burden) by dropping some of the higher frequencies.}

\text{drift\_scale} \quad \text{Scalar (any additional dimensions are treated as batch dimensions) float Tensor indicating the standard deviation of the latent state transitions.}

\text{initial\_state\_prior} \quad \text{instance of tfd\$MultivariateNormal representing the prior distribution on latent states. Must have event shape [num\_features].}

\text{observation\_noise\_scale} \quad \text{Scalar (any additional dimensions are treated as batch dimensions) float Tensor indicating the standard deviation of the observation noise. Default value: 0.}

\text{initial\_step} \quad \text{scalar integer Tensor specifying the starting timestep. Default value: 0.}

\text{validate\_args} \quad \text{logical. Whether to validate input with asserts. If validate\_args is \text{FALSE}, and the inputs are invalid, correct behavior is not guaranteed. Default value: \text{FALSE}.}
allow_nan_stats

logical. If FALSE, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If TRUE, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: TRUE.

name

string prefixed to ops created by this class. Default value: "LocalLinearTrend-StateSpaceModel".

Details

The auxiliary latent state only appears as a matter of construction and thus its interpretation is not particularly important. The total smooth seasonal effect is the sum of the effect values from each of the cyclic components. The parameters drift_scale and observation_noise_scale are each (a batch of) scalars. The batch shape of this Distribution is the broadcast batch shape of these parameters and of the initial_state_prior.

Mathematical Details

The smooth seasonal effect model implements a tfp$\text{distributions}$LinearGaussianStateSpaceModel with latent_size = 2 * len(frequency_multipliers) and observation_size = 1. The latent state is the concatenation of the cyclic latent states which themselves comprise an effect and an auxiliary state. The transition matrix is a block diagonal matrix where block j is:

\[
\text{transition_matrix}[j] = \begin{bmatrix}
\cos(\text{frequencies}[j]), \sin(\text{frequencies}[j]), \\
-sin(\text{frequencies}[j]), \cos(\text{frequencies}[j])
\end{bmatrix}
\]

The observation model picks out the cyclic effect values from the latent state:

\[
\text{observation_matrix} = \begin{bmatrix}
1., 0., 1., 0., ..., 1., 0.
\end{bmatrix}
\]

\[
\text{observation_noise} \sim \text{Normal(loc=0, scale=observation_noise_scale)}
\]

For further mathematical details please see Harvey (1990).

Value

an instance of LinearGaussianStateSpaceModel.

references


See Also

Other sts: sts_additive_state_space_model(), sts_autoregressive_state_space_model(),
sts_autoregressive(), sts_constrained_seasonal_state_space_model(), sts_dynamic_linear_regression_state_space_model(),
sts_dynamic_linear_regression(), sts_linear_regression(),
sts_local_level_state_space_model(),
sts_local_level(),
sts_local_linear_trend_state_space_model(),
sts_local_linear_trend(),
sts_seasonal_state_space_model(),
sts_seasonal(),
sts_semi_local_linear_trend_state_space_model(),
sts_semi_local_linear_trend(),
sts_smooth_seasonal(),
sts_sparse_linear_regression(),
sts_sum()
Formal representation of a sparse linear regression.

Description

This model defines a time series given by a sparse linear combination of covariate time series provided in a design matrix:

Usage

```r
sts_sparse_linear_regression(
  design_matrix,
  weights_prior_scale = 0.1,
  weights_batch_shape = NULL,
  name = NULL
)
```

Arguments

- `design_matrix`: float tensor of shape `tf$concat(list(batch_shape,list(num_timesteps,num_features)))`. This may also optionally be an instance of `tf$linalg$LinearOperator`.
- `weights_prior_scale`: float Tensor defining the scale of the Horseshoe prior on regression weights. Small values encourage the weights to be sparse. The shape must broadcast with `weights_batch_shape`. Default value: 0.1.
- `weights_batch_shape`: if NULL, defaults to `design_matrix$batch_shape_tensor()`. Must broadcast with the batch shape of `design_matrix`. Default value: NULL.
- `name`: the name of this model component. Default value: 'LinearRegression'.

Details

```r
observed_time_series <- tf$matmul(design_matrix, weights)
```

This is identical to `sts_linear_regression`, except that `sts_sparse_linear_regression` uses a parameterization of a Horseshoe prior to encode the assumption that many of the weights are zero, i.e., many of the covariate time series are irrelevant. See the mathematical details section below for further discussion. The prior parameterization used by `sts_sparse_linear_regression` is more suitable for inference than that obtained by simply passing the equivalent `tfd_horseshoe` prior to `sts_linear_regression`; when sparsity is desired, `sts_sparse_linear_regression` will likely yield better results.

This component does not itself include observation noise; it defines a deterministic distribution with mass at the point `tf$matmul(design_matrix, weights)`. In practice, it should be combined with observation noise from another component such as `sts_sum`.

Mathematical Details

The basic horseshoe prior Carvalho et al. (2009) is defined as a Cauchy-normal scale mixture:

```r
horseshoe_scale() ~ Normal(0, 10)
```
scales[i] ~ HalfCauchy(loc=0, scale=1)
weights[i] ~ Normal(loc=0., scale=scales[i] * global_scale)

The Cauchy scale parameters puts substantial mass near zero, encouraging weights to be sparse, but their heavy tails allow weights far from zero to be estimated without excessive shrinkage. The horseshoe can be thought of as a continuous relaxation of a traditional 'spike-and-slab' discrete sparsity prior, in which the latent Cauchy scale mixes between 'spike' (scales[i] ~ 0) and 'slab' (scales[i] >> 0) regimes.

Following the recommendations in Piironen et al. (2017), SparseLinearRegression implements a horseshoe with the following adaptations:

- The Cauchy prior on scales[i] is represented as an InverseGamma-Normal compound.
- The global_scale parameter is integrated out following a Cauchy(0., scale=weights_prior_scale) hyperprior, which is also represented as an InverseGamma-Normal compound.
- All compound distributions are implemented using a non-centered parameterization. The compound, non-centered representation defines the same marginal prior as the original horseshoe (up to integrating out the global scale), but allows samplers to mix more efficiently through the heavy tails; for variational inference, the compound representation implicitly expands the representational power of the variational model.

Note that we do not yet implement the regularized ('Finnish') horseshoe, proposed in Piironen et al. (2017) for models with weak likelihoods, because the likelihood in STS models is typically Gaussian, where it’s not clear that additional regularization is appropriate. If you need this functionality, please email tfprobability@tensorflow.org.

The full prior parameterization implemented in SparseLinearRegression is as follows:

Sample global_scale from Cauchy(0, scale=weights_prior_scale).
global_scale_variance ~ InverseGamma(alpha=0.5, beta=0.5)
global_scale_noncentered ~ HalfNormal(loc=0, scale=1)
global_scale = (global_scale_noncentered * sqrt(global_scale_variance)) * weights_prior_scale
Sample local_scales from Cauchy(0, 1).
local_scale_variances[i] ~ InverseGamma(alpha=0.5, beta=0.5)
local_scales_noncentered[i] ~ HalfNormal(loc=0, scale=1)
local_scales[i] = local_scales_noncentered[i] * sqrt(local_scale_variances[i])
weights[i] ~ Normal(loc=0., scale=local_scales[i] * global_scale)

Value

an instance of StructuralTimeSeries.

References

- Carvalho, C., Polson, N. and Scott, J. Handling Sparsity via the Horseshoe. AISTATS (2009).
See Also

For usage examples see `sts_fit_with_hmc()`, `sts_forecast()`, `sts_decompose_by_component()`.

Other `sts`: `sts_additive_state_space_model()`, `sts_autoregressive_state_space_model()`, `sts_autoregressive()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_linear_regression()`, `sts_local_level_state_space_model()`, `sts_local_level()`, `sts_local_linear_trend_state_space_model()`, `sts_local_linear_trend()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_linear_trend_state_space_model()`, `sts_semi_local_linear_trend()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sum()`

---

**sts_sum**

*Sum of structural time series components.*

**Description**

This class enables compositional specification of a structural time series model from basic components. Given a list of component models, it represents an additive model, i.e., a model of time series that may be decomposed into a sum of terms corresponding to the component models.

**Usage**

```r
sts_sum(
  observed_time_series = NULL,
  components,
  constant_offset = NULL,
  observation_noise_scale_prior = NULL,
  name = NULL
)
```

**Arguments**

- `observed_time_series` (optional float tensor of shape batch_shape + [T, 1]) (omitting the trailing unit dimension is also supported when T > 1), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations. Default value: NULL.

- `components` (list of one or more `StructuralTimeSeries` instances. These must have unique names.

- `constant_offset` (optional scalar float tensor, or batch of scalars, specifying a constant value added to the sum of outputs from the component models. This allows the components to model the shifted series `observed_time_series - constant_offset`. If NULL, this is set to the mean of the provided `observed_time_series`. Default value: NULL.)
observation_noise_scale_prior
optional tfd$Distribution instance specifying a prior on observation_noise_scale.
If NULL, a heuristic default prior is constructed based on the provided observed_time_series.
Default value: NULL.

name
string name of this model component; used as name_scope for ops created by
this class. Default value: 'Sum'.

Details
Formally, the additive model represents a random process $g[t] = f1[t] + f2[t] + ... + fN[t] + \text{eps}[t]$, where the $f$’s are the random processes represented by the components, and $\text{eps}[t] \sim \text{Normal(loc=0, scale=observation_noise_scale)}$ is an observation noise term. See the AdditiveStateSpaceModel documentation for mathematical details.

This model inherits the parameters (with priors) of its components, and adds an observation_noise_scale parameter governing the level of noise in the observed time series.

Value
an instance of StructuralTimeSeries.

See Also
For usage examples see sts_fit_with_hmc(), sts_forecast(), sts_decompose_by_component().

Other sts: sts_additive_state_space_model(), sts_autoregressive_state_space_model().
sts_autoregressive(), sts_constrained_seasonal_state_space_model(), sts_dynamic_linear_regression_state_space_model(),
sts_dynamic_linear_regression(), sts_linear_regression(), sts_local_level_state_space_model(),
sts_local_level(), sts_local_linear_trend_state_space_model(), sts_local_linear_trend(),
sts_seasonal_state_space_model(), sts_seasonal(), sts_semi_local_linear_trend_state_space_model(),
sts_semi_local_linear_trend(), sts_smooth_seasonal_state_space_model(), sts_smooth_seasonal(),
sts_sparse_linear_regression()

---

tfb_absolute_value

Computes $Y = g(X) = \text{Abs}(X)$, element-wise

Description
This non-injective bijector allows for transformations of scalar distributions with the absolute value function, which maps (-inf, inf) to [0, inf).

- For $y$ in (0, inf), tfb_absolute_value$inverse(y)$ returns the set inverse $\{x \in (-\text{inf, inf}) : |x| = y\}$ as a tuple, $-y, y$. tfb_absolute_value$inverse(0)$ returns 0, 0, which is not the set inverse (the set inverse is the singleton \{0\}), but "works" in conjunction with TransformedDistribution to produce a left semi-continuous pdf. For $y < 0$, tfb_absolute_value$inverse(y)$ happily returns the wrong thing, $-y$. This is done for efficiency. If validate_args == TRUE, $y < 0$ will raise an exception.
tfb_affine

Usage

tfb_absolute_value(validate_args = FALSE, name = "absolute_value")

Arguments

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
name name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expml(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(), tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(), tfb_rational_quadratic_spline(), tfb_real_nvp_default_template(), tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(), tfb_scale_matvec_lu(), tfb_scale_matvec_tri_l(), tfb_scale_tril(), tfb_scale(), tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_softmax_centered(), tfb_softplus(), tfb_softsign(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()
scale_perturb_factor = NULL,
scale_perturb_diag = NULL,
adjoint = FALSE,
validate_args = FALSE,
name = "affine",
dtype = NULL
)

Arguments

shift  
Floating-point Tensor. If this is set to NULL, no shift is applied.

scale_identity_multiplier  
Floating point rank 0 Tensor representing a scaling done to the identity matrix.  
When scale_identity_multiplier = scale_diag = scale_tril = NULL then scale += IdentityMatrix. Otherwise no scaled-identity-matrix is added to scale.

scale_diag  
Floating-point Tensor representing the diagonal matrix. scale_diag has shape [N1, N2, ... k], which represents a k x k diagonal matrix. When NULL no diagonal term is added to scale.

scale_tril  
Floating-point Tensor representing the lower triangular matrix. scale_tril has shape [N1, N2, ... k, k], which represents a k x k lower triangular matrix. When NULL no scale_tril term is added to scale. The upper triangular elements above the diagonal are ignored.

scale_perturb_factor  
Floating-point Tensor representing factor matrix with last two dimensions of shape (k, r) When NULL, no rank-r update is added to scale.

scale_perturb_diag  
Floating-point Tensor representing the diagonal matrix. scale_perturb_diag has shape [N1, N2, ... r], which represents an r x r diagonal matrix. When NULL low rank updates will take the form scale_perturb_factor * scale_perturb_factor.T.

adjoint  
Logical indicating whether to use the scale matrix as specified or its adjoint. Default value: FALSE.

validate_args  
Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name  
named prefixed to Ops created by this class.

dtype  
$tf$DType to prefer when converting args to Tensors. Else, we fall back to a common dtype inferred from the args, finally falling back to float32.

Details

If NULL of scale_identity_multiplier, scale_diag, or scale_tril are specified then scale += IdentityMatrix. Otherwise specifying a scale argument has the semantics of scale += Expand(arg), i.e., scale_diag != NULL means scale += tf$diag(scale_diag).

Value

a bijector instance.
tfb_affine_linear_operator

Computes
\[ Y = g(X; \text{shift}, \text{scale}) = \text{scale} @ X + \text{shift} \]

**Description**

shift is a numeric Tensor and scale is a LinearOperator. If X is a scalar then the forward transformation is: scale * X + shift where * denotes broadcasted elementwise product.

**Usage**

```python
tfb_affine_linear_operator(
    shift = NULL,
    scale = NULL,
    adjoint = FALSE,
    validate_args = FALSE,
    name = "affine_linear_operator"
)
```

**Arguments**

- **shift**  
  Floating-point Tensor.

- **scale**  
  Subclass of LinearOperator. Represents the (batch) positive definite matrix M in \( \mathbb{R}^{k \times k} \).

- **adjoint**  
  Logical indicating whether to use the scale matrix as specified or its adjoint. Default value: FALSE.

- **validate_args**  
  Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

- **name**  
  name prefixed to Ops created by this class.

**See Also**

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expm1(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(), tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(), tfb_rational_quadratic_spline(), tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(), tfb_scale_matvec_lu(), tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_softmax_centered(), tfb_softplus(), tfb.softsign(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()
Value

A bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull`

tfb_affine_scalar

AffineScalar bijector

Description

This Bijector is initialized with shift Tensor and scale arguments, giving the forward operation: \( Y = g(X) = \text{scale} \times X + \text{shift} \). If \text{scale} is not specified, then the bijector has the semantics of \text{scale} = 1.. Similarly, if \text{shift} is not specified, then the bijector has the semantics of \text{shift} = 0..

Usage

tfb_affine_scalar(
    shift = NULL,
    scale = NULL,
    validate_args = FALSE,
    name = "affine_scalar"
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift</td>
<td>Floating-point Tensor. If this is set to NULL, no shift is applied.</td>
</tr>
<tr>
<td>scale</td>
<td>Floating-point Tensor. If this is set to NULL, no scale is applied.</td>
</tr>
<tr>
<td>validate_args</td>
<td>Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.</td>
</tr>
<tr>
<td>name</td>
<td>name prefixed to Ops created by this class.</td>
</tr>
</tbody>
</table>
**tfb_batch_normalization**

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull`

---

**Description**

Applies Batch Normalization (Ioffe and Szegedy, 2015) to samples from a data distribution. This can be used to stabilize training of normalizing flows (Papamakarios et al., 2016; Dinh et al., 2017)

**Usage**

```r
tfb_batch_normalization(
  batchnorm_layer = NULL,
  training = TRUE,
  validate_args = FALSE,
  name = "batch_normalization"
)
```

**Arguments**

- `batchnorm_layer` : `tf laiders BatchNormalization` layer object. If `NULL`, defaults to `tf layers BatchNormalization(... + 1e-6)`. This ensures positivity of the scale variable.
- `training` : If `TRUE`, updates running-average statistics during call to `inverse()`.
validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.

Details

When training Deep Neural Networks (DNNs), it is common practice to normalize or whiten features by shifting them to have zero mean and scaling them to have unit variance.

The inverse() method of the BatchNormalization bijection, which is used in the log-likelihood computation of data samples, implements the normalization procedure (shift-and-scale) using the mean and standard deviation of the current minibatch.

Conversely, the forward() method of the bijection de-normalizes samples (e.g. \( X \times \text{std}(Y) + \text{mean}(Y) \)) with the running-average mean and standard deviation computed at training-time. De-normalization is useful for sampling.

During training time, BatchNormalization.inverse and BatchNormalization.forward are not guaranteed to be inverses of each other because inverse(y) uses statistics of the current minibatch, while forward(x) uses running-average statistics accumulated from training. In other words, tfb_batch_normalization().inverse(tfb_batch_normalization().forward(...)) and tfb_batch_normalization().forward(tfb_batch_normalization().inverse(...)) will be identical when training=FALSE but may be different when training=TRUE.

Value

a bijector instance.

References


See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expm1(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inplace(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(), tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(), tfb_rational_quadratic_spline(), tfb_real_nvp_default_template(), tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(),
tfb_blockwise

Bijector which applies a list of bijectors to blocks of a Tensor

Description

More specifically, given \([F_0, F_1, ..., F_n]\) which are scalar or vector bijectors this bijector creates a transformation which operates on the vector \([x_0, ..., x_n]\) with the transformation \([F_0(x_0), F_1(x_1), ..., F_n(x_n)]\) where \(x_0, ..., x_n\) are blocks (partitions) of the vector.

Usage

```r
tfb_blockwise(
  bijectors,
  block_sizes = NULL,
  validate_args = FALSE,
  name = NULL
)
```

Arguments

- **bijectors**: A non-empty list of bijectors.
- **block_sizes**: A 1-D integer Tensor with each element signifying the length of the block of the input vector to pass to the corresponding bijector. The length of block_sizes must be equal to the length of bijectors. If left as NULL, a vector of 1’s is used.
- **validate_args**: Logical indicating whether arguments should be checked for correctness.
- **name**: String, name given to ops managed by this object. Default: E.g., `tfb_blockwise(list(tfb_exp(), tfb_softplus()))$name` == 'blockwise_of_exp_and_softplus'.

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`...
tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(),
tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(),
tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(),
tfb_rational_quadratic_spline(), tfb_real_nvp_default_template(), tfb_real_nvp(),
tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(),
tfb_scale_matvec_lu(), tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shift(),
tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_softmax_centered(), tfb_softplus(), tfb_softsign(),
tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf(),
tfb_weibull()

---

tfb_chain

**Bijector which applies a sequence of bijectors**

**Description**

Bijector which applies a sequence of bijectors.

**Usage**

```r
tfb_chain(bijectors = NULL, validate_args = FALSE, name = NULL)
```

**Arguments**

- **bijectors**: list of bijector instances. An empty list makes this bijector equivalent to the Identity bijector.
- **validate_args**: Logical indicating whether arguments should be checked for correctness.
- **name**: String, name given to ops managed by this object. Default: E.g., `tfb_chain(list(tfb_exp(), tfb_softplus()))$name` == "chain_of_exp_of_softplus".

**Value**

A bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`,
`tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_cholesky_outer_product()`,
`tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`,
`tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`,
`tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`,
`tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`,
`tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`,
`tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`,
`tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`,
`tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`,
`tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`,
`tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`,
`tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`,
`tfb_weibull()`
tfb_cholesky_outer_product

Computes \( g(X) = X @ X^T \) where \( X \) is lower-triangular, positive-diagonal matrix.

**Description**

Note: the upper-triangular part of \( X \) is ignored (whether or not its zero).

**Usage**

```r
tfb_cholesky_outer_product(
  validate_args = FALSE,
  name = "cholesky_outer_product"
)
```

**Arguments**

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**Details**

The surjectivity of \( g \) as a map from the set of \( n \times n \) positive-diagonal lower-triangular matrices to the set of SPD matrices follows immediately from executing the Cholesky factorization algorithm on an SPD matrix \( A \) to produce a positive-diagonal lower-triangular matrix \( L \) such that \( A = L @ L^T \).

To prove the injectivity of \( g \), suppose that \( L_1 \) and \( L_2 \) are lower-triangular with positive diagonals and satisfy \( A = L_1 @ L_1^T = L_2 @ L_2^T \). Then \( \text{inv}(L_1) @ A @ \text{inv}(L_1)^T = \text{inv}(L_1) @ L_1 @ L_1^T = I \).

Setting \( L_3 := \text{inv}(L_1) @ L_2 \), that \( L_3 \) is a positive-diagonal lower-triangular matrix follows from \( \text{inv}(L_1) \) being positive-diagonal lower-triangular (which follows from the diagonal of a triangular matrix being its spectrum), and that the product of two positive-diagonal lower-triangular matrices is another positive-diagonal lower-triangular matrix. A simple inductive argument (proceeding one column of \( L_3 \) at a time) shows that, if \( I = L_3 @ L_3^T \), with \( L_3 \) being lower-triangular with positive diagonal, then \( L_3 = I \). Thus, \( L_1 = L_2 \), proving injectivity of \( g \).

**Value**

a bijector instance.
Maps the Cholesky factor of $M$ to the Cholesky factor of $M^{-1}$

The forward and inverse calculations are conceptually identical to:

```r
forward <- function(x) tf$cholesky(tf$linalg$inv(tf$matmul(x, x, adjoint_b=TRUE)))
```

However, the actual calculations exploit the triangular structure of the matrices.

**Usage**

```r
tfb_cholesky_to_inv_cholesky(
  validate_args = FALSE,
  name = "cholesky_to_inv_cholesky"
)
```

**Arguments**

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**Value**

- a bijector instance.
Maps unconstrained reals to Cholesky-space correlation matrices.

This bijector is a mapping between $\mathbb{R}^n$ and the $n$-dimensional manifold of Cholesky-space correlation matrices embedded in $\mathbb{R}^{m^2}$, where $n$ is the $(m-1)$th triangular number; i.e. $n = 1 + 2 + \ldots + (m-1)$.

Usage

tfb_correlation_cholesky(validate_args = FALSE, name = "correlation_cholesky")

Arguments

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

ame name prefixed to Ops created by this class.

Details

Mathematical Details

The image of unconstrained reals under the CorrelationCholesky bijector is the set of correlation matrices which are positive definite. A correlation matrix can be characterized as a symmetric positive semidefinite matrix with 1s on the main diagonal. However, the correlation matrix is positive definite if no component can be expressed as a linear combination of the other components. For a lower triangular matrix $L$ to be a valid Cholesky-factor of a positive definite correlation matrix, it is necessary and sufficient that each row of $L$ have unit Euclidean norm. To see this, observe that
if $L_i$ is the $i$th row of the Cholesky factor corresponding to the correlation matrix $R$, then the $i$th diagonal entry of $R$ satisfies:

$$1 = R_{i,i} = L_i \cdot L_i = ||L_i||^2$$

where $\cdot$ is the dot product of vectors and $||\cdot||$ denotes the Euclidean norm. Furthermore, observe that $R_{i,j}$ lies in the interval $[-1, 1]$. By the Cauchy-Schwarz inequality:

$$|R_{i,j}| = |L_i \cdot L_j| \leq ||L_i|| \cdot ||L_j|| = 1$$

This is a consequence of the fact that $R$ is symmetric positive definite with 1s on the main diagonal. The LKJ distribution with `input_output_cholesky=TRUE` generates samples from (and computes log-densities on) the set of Cholesky factors of positive definite correlation matrices. The `CorrelationCholesky` bijector provides a bijective mapping from unconstrained reals to the support of the LKJ distribution.

**Value**

a bijector instance.

**References**


**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tril()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordering()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_tril()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_cumsum

Computes the cumulative sum of a tensor along a specified axis.

Description
Computes the cumulative sum of a tensor along a specified axis.

Usage

```r
tfb_cumsum(axis = -1, validate_args = FALSE, name = "cumsum")
```

Arguments

- **axis** int indicating the axis along which to compute the cumulative sum. Note that positive (and zero) values are not supported.
- **validate_args** Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name** name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfbTranspose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_discrete_cosine_transform

Computes $Y = g(X) = DCT(X)$, where $DCT$ type is indicated by the type arg.

Description

The discrete cosine transform efficiently applies a unitary DCT operator. This can be useful for mixing and decorrelating across the innermost event dimension. The inverse $X = g^{-1}(Y) = IDCT(Y)$, where IDCT is DCT-III for type==2. This bijector can be interleaved with Affine bijectors to build a cascade of structured efficient linear layers as in Moczulski et al., 2016. Note that the operator applied is orthonormal (i.e. norm='ortho').

Usage

```r
tfb_discrete_cosine_transform(
  validate_args = FALSE,
  dct_type = 2,
  name = "dct"
)
```

Arguments

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `dct_type` integer, the DCT type performed by the forward transformation. Currently, only 2 and 3 are supported.
- `name` name prefixed to Ops created by this class.

Value

a bijector instance.

References


See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`.
Computes \( Y = g(X) = \exp(X) \)

### Usage

```r
tfb_exp(validate_args = FALSE, name = "exp")
```

### Arguments

- `validate_args`: Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name`: name prefixed to Ops created by this class.

### Value

A bijector instance.

### See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb.softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfbTranspose()`, `tfb_weibull_cdf()`, `tfb_weibull()"
tfb_expm1  \quad Computes Y = g(X) = \exp(X) - 1

Description

This Bijector is no different from tfb_chain([tfb_affine_scalar(shift=-1), tfb_exp()]). However, this makes use of the more numerically stable routines tf$math$expm1 and tf$log1p$.

Usage

```r
tfb_expm1(validate_args = FALSE, name = "expm1")
```

Arguments

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

Details

Note: the `expm1(.)` is applied element-wise but the Jacobian is a reduction over the event space.

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumuuniform()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tf_gumbel_cdf()`, `tf_gumbel()`, `tf_identity()`, `tf_inline()`, `tf_invert()`, `tf_iterated_sigmoid_centered()`, `tf_kumaraswamy_cdf()`, `tf_kumaraswamy()`, `tf_lambert_w_tail()`, `tf_masked_autoregressive_default_template()`, `tfb_minit.inverse_logistic()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_ffjord

Implements a continuous normalizing flow \( X \rightarrow Y \) defined via an ODE.

Description

This bijector implements a continuous dynamics transformation parameterized by a differential equation, where initial and terminal conditions correspond to domain (X) and image (Y) i.e.

Usage

```python
tfb_ffjord(
    state_time_derivative_fn,
    ode_solve_fn = NULL,
    trace_augmentation_fn = tfp$bijectors$ffjord$trace_jacobian_hutchinson,
    initial_time = 0,
    final_time = 1,
    validate_args = FALSE,
    dtype = tf$float32,
    name = "ffjord"
)
```

Arguments

- `state_time_derivative_fn` function taking arguments `time` (a scalar representing time) and `state` (a Tensor representing the state at given time) returning the time derivative of the state at given time.

- `ode_solve_fn` function taking arguments `ode_fn` (same as `state_time_derivative_fn` above), `initial_time` (a scalar representing the initial time of integration), `initial_state` (a Tensor of floating dtype represents the initial state) and `solution_times` (1D Tensor of floating dtype representing time at which to obtain the solution) returning a Tensor of shape \([\text{time\_axis}, \text{initial\_state\_shape}]\). Will take \(\text{final\_time}\) as the `solution_times` argument and `state_time_derivative_fn` as `ode_fn` argument. If NULL a DormandPrince solver from `tfp$math$ode` is used. Default value: NULL

- `trace_augmentation_fn` function taking arguments `ode_fn` (function same as `state_time_derivative_fn` above), `state_shape` (TensorShape of a the state), `dtype` (same as dtype of the state) and returning a function taking arguments `time` (a scalar representing the time at which the function is evaluated), `state` (a Tensor representing the state at given time) that computes a tuple \((\text{ode\_fn}(\text{time}, \text{state}), \text{jacobian\_trace\_estimation})\). \text{jacobian\_trace\_estimation} should represent trace of the jacobian of \text{ode\_fn} with respect to \text{state}. \text{state\_time\_derivative\_fn} will be passed as \text{ode\_fn} argument. Default value: tfp$bijectors$ffjord$trace_jacobian_hutchinson

- `initial_time` Scalar float representing time to which the x value of the bijector corresponds to. Passed as `initial_time` to `ode_solve_fn`. For default solver can be float or floating scalar Tensor. Default value: 0.
final_time Scalar float representing time to which the y value of the bijector corresponds to. Passed as solution_times to ode_solve_fn. For default solver can be float or floating scalar Tensor. Default value: 1.

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

dtype tf$DType to prefer when converting args to Tensors. Else, we fall back to a common dtype inferred from the args, finally falling back to float32.

name name prefixed to Ops created by this class.

Details

d/dt[state(t)] = state_time_derivative_fn(t, state(t))
state(initial_time) = X
state(final_time) = Y

For this transformation the value of log_det_jacobian follows another differential equation, reducing it to computation of the trace of the jacobian along the trajectory

state_time_derivative = state_time_derivative_fn(t, state(t))
d/dt[log_det_jac(t)] = Tr(jacobian(state_time_derivative, state(t)))

FFJORD constructor takes two functions ode_solve_fn and trace_augmentation_fn arguments that customize integration of the differential equation and trace estimation.

Differential equation integration is performed by a call to ode_solve_fn.

Custom ode_solve_fn must accept the following arguments:

- ode_fn(time, state): Differential equation to be solved.
- initial_time: Scalar float or floating Tensor representing the initial time.
- initial_state: Floating Tensor representing the initial state.
- solution_times: 1D floating Tensor of solution times.

And return a Tensor of shape [solution_times$shape, initial_state$shape] representing state values evaluated at solution_times. In addition ode_solve_fn must support nested structures. For more details see the interface of tfp$math$ode$Solver$solve().

Trace estimation is computed simultaneously with state_time_derivative using augmented_state_time_derivative_fn that is generated by trace_augmentation_fn. trace_augmentation_fn takes state_time_derivative_fn, state.shape and state.dtype arguments and returns a augmented_state_time_derivative_fn callable that computes both state_time_derivative and unreduced trace_estimation.

Custom ode_solve_fn and trace_augmentation_fn examples:

```r
# custom_solver_fn: 'function(f, t_initial, t_solutions, y_initial, ...)' # ... : Additional arguments to pass to custom_solver_fn.
ode_solve_fn <- function(ode_fn, initial_time, initial_state, solution_times) {
  custom_solver_fn(ode_fn, initial_time, solution_times, initial_state, ...)
}
ffjord <- tfb_ffjord(state_time_derivative_fn, ode_solve_fn = ode_solve_fn)
```
tfb_ffjord

# state_time_derivative_fn: `function(time, state)`
# trace_jac_fn: `function(time, state)` unreduced jacobian trace function
trace_augmentation_fn <- function(ode_fn, state_shape, state_dtype) {
  augmented_ode_fn <- function(time, state) {
    list(ode_fn(time, state), trace_jac_fn(time, state))
  }
  augmented_ode_fn
}

ffjord <- tfb_ffjord(state_time_derivative_fn, trace_augmentation_fn = trace_augmentation_fn)

For more details on FFJORD and continuous normalizing flows see Chen et al. (2018), Grathwohl et al. (2018).

Value

a bijector instance.

References


See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(),
tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform,
tfb_expml(), tfb_exp(), tfb_fill_scale_tril(), tfb_fill_triangular(), tfb_gumbel_cdf(),
tfb_gumbel(), tfb_identity(), tfb_inverse(), tfb_invert(), tfb_iterated_sigmoid_centered(),
tfb_kumaraswamy_pdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(),
tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tril(),
tfb_matvec(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(),
tfb_rational_quadratic_spline(), tfb_real_nvp_default_template(), tfb_real_nvp(),
tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(),
tfb_scale_matvec(), tfb_scale_matvec_tril(), tfb_scale_tril(), tfb_shift(),
tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_softmax_centered(), tfb_softplus(), tfb_softsign(),
tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf(),
tfb_weibull()
tfb_fill_scale_tri_l  
Transforms unconstrained vectors to TriL matrices with positive diagonal

Description

This is implemented as a simple tfb_chain of tfb_fill_triangular followed by tfb_transform_diagonal, and provided mostly as a convenience. The default setup is somewhat opinionated, using a Softplus transformation followed by a small shift (1e-5) which attempts to avoid numerical issues from zeros on the diagonal.

Usage

```python
tfb_fill_scale_tri_l(
    diag_bijector = NULL,
    diag_shift = 1e-05,
    validate_args = FALSE,
    name = "fill_scale_tril"
)
```

Arguments

- **diag_bijector**: Bijector instance, used to transform the output diagonal to be positive. Default value: NULL (i.e., tfb_softplus()).
- **diag_shift**: Float value broadcastable and added to all diagonal entries after applying the diag_bijector. Setting a positive value forces the output diagonal entries to be positive, but prevents inverting the transformation for matrices with diagonal entries less than this value. Default value: 1e-5.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expm1(), tfb_exp(), tfb_ffjord(), tfb_fill_triangular(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(),
tfb_fill_triangular

Transforms vectors to triangular

Description

Triangular matrix elements are filled in a clockwise spiral. Given input with shape batch_shape + [d], produces output with shape batch_shape + [n, n], where \( n = (-1 + \sqrt{1 + 8 \times d})/2 \). This follows by solving the quadratic equation \( d = 1 + 2 + \ldots + n = n \times (n + 1)/2 \).

Usage

```r
tfb_fill_triangular(
    upper = FALSE,
    validate_args = FALSE,
    name = "fill_triangular"
)
```

Arguments

- **upper**: Logical representing whether output matrix should be upper triangular (TRUE) or lower triangular (FALSE, default).
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expml(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(),...
tfb_forward

Returns the forward Bijector evaluation, i.e., $X = g(Y)$.

Description

Returns the forward Bijector evaluation, i.e., $X = g(Y)$.

Usage

tfb_forward(bijector, x, name = "forward")

Arguments

bijector The bijector to apply
x Tensor. The input to the "forward" evaluation.
name name of the operation

Value

a tensor

See Also

Other bijector_methods: tfb_forward_log_det_jacobian(), tfb_inverse_log_det_jacobian(), tfb_inverse()

Examples

b <- tfb_affine_scalar(shift = 1, scale = 2)
x <- 10
b %>% tfb_forward(x)
tfb_forward_log_det_jacobian

Returns the result of the forward evaluation of the log determinant of the Jacobian

Description

Returns the result of the forward evaluation of the log determinant of the Jacobian

Usage

```
tfb_forward_log_det_jacobian(
    bijector,
    x,
    event_ndims,
    name = "forward_log_det_jacobian"
)
```

Arguments

- **bijector**: The bijector to apply
- **x**: Tensor. The input to the "forward" Jacobian determinant evaluation.
- **event_ndims**: Number of dimensions in the probabilistic events being transformed. Must be greater than or equal to `bijector$forward_min_event_ndims`. The result is summed over the final dimensions to produce a scalar Jacobian determinant for each event, i.e. it has shape `x$shape$ndims - event_ndims` dimensions.
- **name**: name of the operation

Value

a tensor

See Also

Other bijector methods: `tfb_forward()`, `tfb_inverse_log_det_jacobian()`, `tfb_inverse()`

Examples

```
b <- tfb_affine_scalar(shift = 1, scale = 2)
x <- 10
b %>% tfb_forward_log_det_jacobian(x, event_ndims = 0)
```
tfb_gumbel

Computes \( Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale})) \)

**Description**

This bijector maps inputs from \([-\infty, \infty]\) to \([0, 1]\). The inverse of the bijector applied to a uniform random variable \( X \sim U(0, 1) \) gives back a random variable with the Gumbel distribution:

**Usage**

```python
tfb_gumbel(loc = 0, scale = 1, validate_args = FALSE, name = "gumbel")
```

**Arguments**

- **loc**: Float-like Tensor that is the same dtype and is broadcastable with scale. This is loc in \( Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale})) \).
- **scale**: Positive Float-like Tensor that is the same dtype and is broadcastable with loc. This is scale in \( Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale})) \).
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

**Details**

\( Y \sim \text{Gumbel}(\text{loc}, \text{scale}) \)

\[
\text{pdf}(y; \text{loc}, \text{scale}) = \exp(-((y - \text{loc}) / \text{scale} + \exp(-(y - \text{loc}) / \text{scale}))) / \text{scale}
\]

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matveclu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matveclu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_gumbel_cdf

Compute $Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale}))$, the Gumbel CDF.

Description

This bijector maps inputs from $[-\infty, \infty]$ to $[0, 1]$. The inverse of the bijector applied to a uniform random variable $X \sim U(0, 1)$ gives back a random variable with the Gumbel distribution:

Usage

```r
tfb_gumbel_cdf(loc = 0, scale = 1, validate_args = FALSE, name = "gumbel_cdf")
```

Arguments

- **loc**: Float-like Tensor that is the same dtype and is broadcastable with scale. This is loc in $Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale}))$.
- **scale**: Positive Float-like Tensor that is the same dtype and is broadcastable with loc. This is scale in $Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale}))$.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Details

$Y \sim \text{GumbelCDF}(\text{loc}, \text{scale})$

$$
\text{pdf}(y; \text{loc}, \text{scale}) = \exp(-\left(\frac{y - \text{loc}}{\text{scale}} + \exp\left(-\frac{y - \text{loc}}{\text{scale}}\right)\right)) / \text{scale}
$$

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_flow()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_flow()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`,
tfb_identity

Computes \( Y = g(X) = X \)

**Description**

Computes \( Y = g(X) = X \)

**Usage**

```r
tfb_identity(validate_args = FALSE, name = "identity")
```

**Arguments**

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
Bijector constructed from custom functions

**Description**

Bijector constructed from custom functions

**Usage**

```r
tfb_inline(
  forward_fn = NULL,
  inverse_fn = NULL,
  inverse_log_det_jacobian_fn = NULL,
  forward_log_det_jacobian_fn = NULL,
  forward_event_shape_fn = NULL,
  forward_event_shape_tensor_fn = NULL,
  inverse_event_shape_fn = NULL,
  inverse_event_shape_tensor_fn = NULL,
  is_constant_jacobian = NULL,
  validate_args = FALSE,
  forward_min_event_ndims = NULL,
  inverse_min_event_ndims = NULL,
  name = "inline"
)
```

**Arguments**

- `forward_fn` Function implementing the forward transformation.
- `inverse_fn` Function implementing the inverse transformation.
- `inverse_log_det_jacobian_fn` Function implementing the log_det_jacobian of the forward transformation.
- `forward_log_det_jacobian_fn` Function implementing the log_det_jacobian of the inverse transformation.
- `forward_event_shape_fn` Function implementing non-identical static event shape changes. Default: shape is assumed unchanged.
- `forward_event_shape_tensor_fn` Function implementing non-identical event shape changes. Default: shape is assumed unchanged.
- `inverse_event_shape_fn` Function implementing non-identical static event shape changes. Default: shape is assumed unchanged.
- `inverse_event_shape_tensor_fn` Function implementing non-identical event shape changes. Default: shape is assumed unchanged.
tfb_inverse

is_constant_jacobian
Logical indicating that the Jacobian is constant for all input arguments.

validate_args
Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

forward_min_event_ndims
Integer indicating the minimal dimensionality this bijector acts on.

inverse_min_event_ndims
Integer indicating the minimal dimensionality this bijector acts on.

name
name prefixed to Ops created by this class.

Value
a bijector instance.

See Also
For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
  tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_out
  er_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(),
  tfb_discrete_cosine_transform(), tfb_expml(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tr
  i_l(), tfb_fill_triangular(),
  tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_invert(), tfb_iterated_sigmoid_centered(),
  tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressiv
  e_default_template(),
  tfb_masked_autoregressive_flow(), tfb_masked_density(), tfb_matrix_inverse_tri_l(),
  tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(),
  tfb_rational_quadratic_spline(), tfb_real_nvp_default_template(), tfb_real_nvp(),
  tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(),
  tfb_scale_matvec_lu(), tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shift(),
  tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_softmax_centered(), tfb_softplus(), tfb_softsign(),
  tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf(),
  tfb_weibull()

________________________
tfb_inverse

Returns the inverse Bijector evaluation, i.e., $X = g^{-1}(Y)$.

Description
Returns the inverse Bijector evaluation, i.e., $X = g^{-1}(Y)$.

Usage
tfb_inverse(bijector, y, name = "inverse")

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bijector</td>
<td>The bijector to apply</td>
</tr>
<tr>
<td>y</td>
<td>Tensor. The input to the &quot;inverse&quot; evaluation.</td>
</tr>
<tr>
<td>name</td>
<td>name of the operation</td>
</tr>
</tbody>
</table>


tbf_inverse_log_det_jacobian

Value
  a tensor

See Also
  Other bijector methods: tbf_forward_log_det_jacobian(), tbf_forward(), tbf_inverse_log_det_jacobian()

Examples

b <- tbf_affine_scalar(shift = 1, scale = 2)
x <- 10
y <- b %>% tbf_forward(x)
b %>% tbf_inverse(y)

tfb_inverse_log_det_jacobian
  Returns the result of the inverse evaluation of the log determinant of the Jacobian

Description
  Returns the result of the inverse evaluation of the log determinant of the Jacobian

Usage
  tbf_inverse_log_det_jacobian(
    bijector,
    y,
    event_ndims,
    name = "inverse_log_det_jacobian"
  )

Arguments
  bijector  The bijector to apply
  y  Tensor. The input to the "inverse" Jacobian determinant evaluation.
  event_ndims  Number of dimensions in the probabilistic events being transformed. Must be greater than or equal to bijector$inverse_min_event_ndims. The result is summed over the final dimensions to produce a scalar Jacobian determinant for each event, i.e. it has shape x$shape$ndims - event_ndims dimensions.
  name  name of the operation

Value
  a tensor
See Also

Other bijector methods: `tfb_forward_log_det_jacobian()`, `tfb_forward()`, `tfb_inverse()`

Examples

```r
b <- tfb_affine_scalar(shift = 1, scale = 2)
x <- 10
y <- b %>% tfb_forward(x)
b %>% tfb_inverse_log_det_jacobian(y, event_ndims = 0)
```

---

**tfb_invert**

*Bijector which inverts another Bijector*

Description

Creates a Bijector which swaps the meaning of inverse and forward. Note: An inverted bijector’s `inverse_log_det_jacobian` is often more efficient if the base bijector implements `_forward_log_det_jacobian`. If `_forward_log_det_jacobian` is not implemented then the following code is used: `y = b$inverse(x)`

```r
-b$inverse_log_det_jacobian(y)
```

Usage

```r
tfb_invert(bijector, validate_args = FALSE, name = NULL)
```

Arguments

- **bijector**: Bijector instance.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`,
Bijector which applies a Stick Breaking procedure.

**Description**

Bijector which applies a Stick Breaking procedure.

**Usage**

```r
tfb_iterated_sigmoid_centered(validate_args = FALSE, name = "iterated_sigmoid")
```

**Arguments**

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_kumaraswamy

Computes $Y = g(X) = (1 - (1 - X)**(1 / b))**(1 / a)$, with $X$ in $[0, 1]$

Description

This bijector maps inputs from $[0, 1]$ to $[0, 1]$. The inverse of the bijector applied to a uniform random variable $X \sim U(0, 1)$ gives back a random variable with the Kumaraswamy distribution: $Y \sim \text{Kumaraswamy}(a, b)$

pdf$(y; a, b, 0 <= y <= 1) = a * b * y^{(a - 1)} * (1 - y^a)^{(b - 1)}$

Usage

```
tfb_kumaraswamy(
    concentration1 = NULL,
    concentration0 = NULL,
    validate_args = FALSE,
    name = "kumaraswamy"
)
```

Arguments

- `concentration1` float scalar indicating the transform power, i.e., $Y = g(X) = (1 - (1 - X)**(1 / b))**(1 / a)$ where $a$ is concentration$1$.
- `concentration0` float scalar indicating the transform power, i.e., $Y = g(X) = (1 - (1 - X)**(1 / b))**(1 / a)$ where $b$ is concentration$0$.
- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inplace()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tril()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_tanh()`, `tfb_warning()`. 
tfb_kumaraswamy_cdf

Computes

\[ Y = g(X) = (1 - (1 - X)^{1/b})^{1/a}, \]

with \( X \) in \([0, 1]\)

Description

This bijector maps inputs from \([0, 1]\) to \([0, 1]\). The inverse of the bijector applied to a uniform random variable \( X \sim U(0, 1) \) gives back a random variable with the Kumaraswamy distribution: \( Y \sim \text{Kumaraswamy}(a, b) \)

\[
\text{pdf}(y; a, b, 0 \leq y \leq 1) = a \cdot b \cdot y^{(a - 1)} \cdot (1 - y^a)^{(b - 1)}
\]

Usage

```r
tfb_kumaraswamy_cdf(
    concentration1 = 1,
    concentration0 = 1,
    validate_args = FALSE,
    name = "kumaraswamy_cdf"
)
```

Arguments

- `concentration1` float scalar indicating the transform power, i.e., \( Y = g(X) = (1 - (1 - X)^{1/b})^{1/a} \) where a is concentration1.
- `concentration0` float scalar indicating the transform power, i.e., \( Y = g(X) = (1 - (1 - X)^{1/b})^{1/a} \) where b is concentration0.
- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_exp1(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb_iterated_sigmoid_centered()
tfb_lambert_w_tail

LambertWTail transformation for heavy-tail Lambert W x F random variables.

Description

A random variable Y has a Lambert W x F distribution if W_t(Y) = X has distribution F, where tau = (shift, scale, tail) parameterizes the inverse transformation.

Usage

tfb_lambert_w_tail(
  shift = NULL,
  scale = NULL,
  tailweight = NULL,
  validate_args = FALSE,
  name = "lambertw_tail"
)

Arguments

shift
  Floating point tensor; the shift for centering (uncentering) the input (output) random variable(s).

scale
  Floating point tensor; the scaling (unscaling) of the input (output) random variable(s). Must contain only positive values.

tailweight
  Floating point tensor; the tail behaviors of the output random variable(s). Must contain only non-negative values.

validate_args
  Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name
  name prefixed to Ops created by this class.

Details

This bijector defines the transformation underlying Lambert W x F distributions that transform an input random variable to an output random variable with heavier tails. It is defined as Y = (U * exp(0.5 * tail * U^2)) * scale + shift, tail >= 0 where U = (X - shift) / scale is a shifted/scaled input random variable, and tail >= 0 is the tail parameter.
Attributes: shift: shift to center (uncenter) the input data. scale: scale to normalize (de-normalize) the input data. tailweight: Tail parameter \( \delta \) of heavy-tail transformation; must be \( \geq 0 \).

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb.softsign()`, `tfb_square()`, `tfb.tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

tfb_masked_autoregressive_default_template

Masked Autoregressive Density Estimator

Description

This will be wrapped in a make_template to ensure the variables are only created once. It takes the input and returns the loc ("mu" in Germain et al. (2015)) and log_scale ("alpha" in Germain et al. (2015)) from the MADE network.

Usage

tfb_masked_autoregressive_default_template(
    hidden_layers,
    shift_only = FALSE,
    activation = tf$nn$relu,
    log_scale_min_clip = -5,
    log_scale_max_clip = 3,
    log_scale_clip_gradient = FALSE,
    name = NULL,
    ...)
)
Arguments

hidden_layers: list-like of non-negative integer, scalars indicating the number of units in each hidden layer. Default: list(512,512).

shift_only: logical indicating if only the shift term shall be computed. Default: FALSE.

activation: Activation function (callable). Explicitly setting to NULL implies a linear activation.

log_scale_min_clip: float-like scalar Tensor, or a Tensor with the same shape as log_scale. The minimum value to clip by. Default: -5.

log_scale_max_clip: float-like scalar Tensor, or a Tensor with the same shape as log_scale. The maximum value to clip by. Default: 3.

log_scale_clip_gradient: logical indicating that the gradient of tf$clip_by_value should be preserved. Default: FALSE.

name: A name for ops managed by this function. Default: "tfb_masked_autoregressive_default_template".

... tf$layers$dense arguments

Details

Warning: This function uses masked_dense to create randomly initialized tf$Variables. It is presumed that these will be fit, just as you would any other neural architecture which uses tf$layers$dense.

About Hidden Layers Each element of hidden_layers should be greater than the input_depth (i.e., input_depth = tf$shape(input)[-1] where input is the input to the neural network). This is necessary to ensure the autoregressivity property.

About Clipping This function also optionally clips the log_scale (but possibly not its gradient). This is useful because if log_scale is too small/large it might underflow/overflow making it impossible for the MaskedAutoregressiveFlow bijector to implement a bijection. Additionally, the log_scale_clip_gradient bool indicates whether the gradient should also be clipped. The default does not clip the gradient; this is useful because it still provides gradient information (for fitting) yet solves the numerical stability problem. I.e., log_scale_clip_gradient = FALSE means grad[exp(clip(x))] = grad[x] exp(clip(x)) rather than the usual grad[clip(x)] exp(clip(x)).

Value

list of:

- shift: Float-like Tensor of shift terms
- log_scale: Float-like Tensor of log(scale) terms

References

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
    tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(),
    tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform
    tfb_expm1(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(),
    tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inverse(), tfb_iterated_sigmoid_centered
    tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_flow(),
    tfb_masked_dense(), tfb_matrix_inverse_tri_l(), tfb_matrix_tri_l(), tfb_normal_cdf(),
    tfb_ortho_matrix(), tfb_pad(), tfb_permute(), tfb_power_transform(), tfb_rational_quadratic_spline(),
    tfb_real_nvp_default_template(), tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_linear_operator(),
    tfb_scale_matvec_linear_operator(), tfb_scale_matvec_linear_operator(), tfb_scale_matvec_tri_l(),
    tfb_shift(), tfb_sigmoid(), tfb_sigmoid_centered(), tfb_softplus(), tfb_softsign(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(),
    tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()
tfb_masked_autoregressive_flow

is_constant_jacobian
Logical, default: FALSE. When TRUE the implementation assumes log_scale does not depend on the forward domain (x) or inverse domain (y) values. (No validation is made; is_constant_jacobian=FALSE is always safe but possibly computationally inefficient.)

unroll_loop
Logical indicating whether the tf$while_loop in _forward should be replaced with a static for loop. Requires that the final dimension of x be known at graph construction time. Defaults to FALSE.

event_ndims
integer, the intrinsic dimensionality of this bijector. 1 corresponds to a simple vector autoregressive bijector as implemented by the tfb_masked_autoregressive_default_template, 2 might be useful for a 2D convolutional shift_and_log_scale_fn and so on.

validate_args
Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name
name prefixed to Ops created by this class.

Details

"Autoregressive models decompose the joint density as a product of conditionals, and model each conditional in turn. Normalizing flows transform a base density (e.g. a standard Gaussian) into the target density by an invertible transformation with tractable Jacobian." (Papamakarios et al., 2016)

In other words, the "autoregressive property" is equivalent to the decomposition, p(x) = \prod_{i=0}^{d} p(x[i] | x[0:i]) where perm is some permutation of \{0, ..., d\}. In the simple case where the permutation is identity this reduces to:

p(x) = \prod_{i=0}^{d} p(x[i] | x[0:i])

The provided shift_and_log_scale_fn, tfb_masked_autoregressive_default_template, achieves this property by zeroing out weights in its masked_dense layers. In TensorFlow Probability, "normalizing flows" are implemented as tfp.bijectors.Bijectors. The forward "autoregression" is implemented using a tf.while_loop and a deep neural network (DNN) with masked weights such that the autoregressive property is automatically met in the inverse. A TransformedDistribution using MaskedAutoregressiveFlow(...) uses the (expensive) forward-mode calculation to draw samples and the (cheap) reverse-mode calculation to compute log-probabilities. Conversely, a TransformedDistribution using Invert(MaskedAutoregressiveFlow(...)) uses the (expensive) forward-mode calculation to compute log-probabilities and the (cheap) reverse-mode calculation to compute samples.

Given a shift_and_log_scale_fn, the forward and inverse transformations are (a sequence of) affine transformations. A "valid" shift_and_log_scale_fn must compute each shift (aka loc or "mu" in Germain et al. (2015)) and log(scale) (aka "alpha" in Germain et al. (2015)) such that each are broadcastable with the arguments to forward and inverse, i.e., such that the calculations in forward, inverse below are possible.

For convenience, tfb_masked_autoregressive_default_template is offered as a possible shift_and_log_scale_fn function. It implements the MADE architecture (Germain et al., 2015). MADE is a feed-forward network that computes a shift and log(scale) using masked_dense layers in a deep neural network. Weights are masked to ensure the autoregressive property. It is possible that this architecture is suboptimal for your task. To build alternative networks, either change the arguments to tfb_masked_autoregressive_default_template, use the masked_dense function to roll-out your own, or use some other architecture, e.g., using tf.layers. Warning: no attempt is made to validate that the shift_and_log_scale_fn enforces the "autoregressive property".

Assuming shift_and_log_scale_fn has valid shape and autoregressive semantics, the forward transformation is
```python
def forward(x):
    y = zeros_like(x)
    event_size = x.shape[-event_dims:].num_elements()
    for _ in range(event_size):
        shift, log_scale = shift_and_log_scale_fn(y)
        y = x * tf.exp(log_scale) + shift
    return y
```

and the inverse transformation is

```python
def inverse(y):
    shift, log_scale = shift_and_log_scale_fn(y)
    return (y - shift) / tf.exp(log_scale)
```

Notice that the inverse does not need a for-loop. This is because in the forward pass each calculation of shift and log_scale is based on the y calculated so far (not x). In the inverse, the y is fully known, thus is equivalent to the scaling used in forward after event_size passes, i.e., the "last" y used to compute shift, log_scale. (Roughly speaking, this also proves the transform is bijective.)

Value

a bijector instance.

References


See Also

For usage examples see `tfb.forward()`, `tfb.inverse()`, `tfb.inverse_log_det_jacobian()`. Other bijectors: `tfb.absolute_value()`, `tfb.affine_linear_operator()`, `tfb.affine_scalar()`, `tfb.affine()`, `tfb.batch_normalization()`, `tfb.blockwise()`, `tfb.chain()`, `tfb.cholesky_outer_product()`, `tfb.cholesky_to_inv_cholesky()`, `tfb.correlation_cholesky()`, `tfb.cumsum()`, `tfb.discrete_cosine_transform()`, `tfb.expm1()`, `tfb.exp()`, `tfb.ffjord()`, `tfb.fill_scale_tri_l()`, `tfb.fill_triangular()`, `tfb.gumbel_cdf()`, `tfb.gumbel()`, `tfb.identity()`, `tfb.inline()`, `tfb.invert()`, `tfb.iterated_sigmoid_centered()`, `tfb.kumaraswamy_cdf()`, `tfb.kumaraswamy()`, `tfb.lambert_w_tail()`, `tfb.masked_autoregressive_default_template()`, `tfb.masked_dense()`, `tfb.matrix_inverse_tri_l()`, `tfb.matvec_lu()`, `tfb.normal_cdf()`, `tfb.ordered()`, `tfb.pad()`., `tfb.permute()`, `tfb.power_transform()`, `tfb.rational_quadratic_spline()`, `tfb.real_nvp_default_template()`, `tfb.real_nvp()`, `tfb.reciprocal()`, `tfb.reshape()`, `tfb.scale_matvec_diagonal()`, `tfb.scale_matvec_linear_operator()`, `tfb.scale_matvec_lu()`, `tfb.scale_matvec_tri_l()`, `tfb.scale_tri_l()`, `tfb.scale()`, `tfb.shift()`, `tfb.sigmoid()`, `tfb.sinh_arcsinh()`, `tfb.softmax_centered()`, `tfb.softplus()`, `tfb.softsign()`, `tfb.square()`, `tfb.tanh()`, `tfb.transform_diagonal()`, `tfb.transpose()`, `tfb.weibull_cdf()`, `tfb.weibull()`
tfb_masked_dense  

Autoregressively masked dense layer

Description

Analogous to `tf$layers$dense`.

Usage

```r
tfb_masked_dense(
  inputs,
  units,
  num_blocks = NULL,
  exclusive = FALSE,
  kernel_initializer = NULL,
  reuse = NULL,
  name = NULL,
  ...
)
```

Arguments

- **inputs**: Tensor input.
- **units**: integer scalar representing the dimensionality of the output space.
- **num_blocks**: integer scalar representing the number of blocks for the MADE masks.
- **exclusive**: logical scalar representing whether to zero the diagonal of the mask, used for the first layer of a MADE.
- **kernel_initializer**:Initializer function for the weight matrix. If NULL (default), weights are initialized using the `tf$glorot_random_initializer`
- **reuse**: logical scalar representing whether to reuse the weights of a previous layer by the same name.
- **name**: string used to describe ops managed by this function.
- **...**: `tf$layers$dense` arguments

Details

See Germain et al. (2015) for detailed explanation.

Value

- **tensor**
Computes \( g(L) = \text{inv}(L) \), where \( L \) is a lower-triangular matrix

**Description**

\( L \) must be nonsingular; equivalently, all diagonal entries of \( L \) must be nonzero. The input must have rank \( \geq 2 \). The input is treated as a batch of matrices with batch shape \( \text{input.shape[:-2]} \), where each matrix has dimensions \( \text{input.shape[-2]} \) by \( \text{input.shape[-1]} \) (hence \( \text{input.shape[-2]} \) must equal \( \text{input.shape[-1]} \)).

**Usage**

```r
tfb_matrix_inverse_tril(validate_args = FALSE, name = "matrix_inverse_tril")
```

**Arguments**

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**Value**

a bijector instance.
See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diagonal()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()

---

**tfb_matvec_lu**

Matrix-vector multiply using LU decomposition

**Description**

This bijector is identical to the "Convolution1x1" used in Glow (Kingma and Dhariwal, 2018).

**Usage**

```r
tfb_matvec_lu(lower_upper, permutation, validate_args = FALSE, name = NULL)
```

**Arguments**

- **lower_upper**: The LU factorization as returned by `tf$linalg$lu`
- **permutation**: The LU factorization permutation as returned by `tf$linalg$lu`
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

**Details**

Warning: this bijector never verifies the scale matrix (as parameterized by LU decomposition) is invertible. Ensuring this is the case is the caller’s responsibility.

**Value**

a bijector instance.
tfb_normal_cdf

Computes \( Y = g(X) = \text{NormalCDF}(x) \)

Description
This bijector maps inputs from \([-\infty, \infty]\) to \([0, 1]\). The inverse of the bijector applied to a uniform random variable \( X \sim U(0, 1) \) gives back a random variable with the Normal distribution:

Usage
```
tfb_normal_cdf(validate_args = FALSE, name = "normal")
```

Arguments
- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

Details
\( Y \sim \text{Normal}(0, 1) \) \( \text{pdf}(y; 0, 1) = 1 / \sqrt{2 \pi} * \exp(-y^2 / 2) \)

Value
a bijector instance.

References
See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diagonal()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

---

tfb_ordered

*Bijector which maps a tensor $x_k$ that has increasing elements in the last dimension to an unconstrained tensor $y_k$*

---

**Description**

Both the domain and the codomain of the mapping is $[-\infty, \infty]$, however, the input of the forward mapping must be strictly increasing. The inverse of the bijector applied to a normal random vector $y \sim N(0,1)$ gives back a sorted random vector with the same distribution $x \sim N(0,1)$ where $x = \text{sort}(y)$

**Usage**

```r
tfb_ordered(validate_args = FALSE, name = "ordered")
```

**Arguments**

- `validate_args` Logical, default `FALSE`. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**Details**

On the last dimension of the tensor, Ordered bijector performs: $y[0] = x[0]$ $y[1:] = \text{tf}\log(x[1:] - x[:-1])$

**Value**

a bijector instance.
See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`.

`tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_squared()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

---

**tfb_pad**

*Pads a value to the event_shape of a Tensor.*

---

**Description**

The semantics of `bijector_pad` generally follow that of `tf$pad()` except that `bijector_pad`'s `paddings` argument applies to the rightmost dimensions. Additionally, the new argument `axis` enables overriding the dimensions to which `paddings` is applied. Like `paddings`, the `axis` argument is also relative to the rightmost dimension and must therefore be negative. The argument `paddings` is a vector of integer pairs each representing the number of left and/or right constant_values to pad to the corresponding rightmost dimensions. That is, unless `axis` is specified, specifying `k` different `paddings` means the rightmost `k` dimensions will be "grown" by the sum of the respective `paddings` row. When `axis` is specified, it indicates the dimension to which the corresponding `paddings` element is applied. By default, `axis` is `NULL` which means it is logically equivalent to `range(start=-length(paddings), limit=0)` i.e., the rightmost dimensions.

**Usage**

```r
tfb_pad(
  paddings = list(c(0, 1)),
  mode = "CONSTANT",
  constant_values = 0,
  axis = NULL,
  validate_args = FALSE,
  name = NULL
)
```
tfb_permute

Permutes the rightmost dimension of a Tensor

Description

Permutes the rightmost dimension of a Tensor.
Usage

tfb_permute(permutation, axis = -1L, validate_args = FALSE, name = NULL)

Arguments

permutation  An integer-like vector-shaped Tensor representing the permutation to apply to
the axis dimension of the transformed Tensor.

axis         Scalar integer Tensor representing the dimension over which to tf$gather. axis
must be relative to the end (reading left to right) thus must be negative. Default
value: -1 (i.e., right-most).

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args
is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name         name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(),
tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform,
tfb_exp1(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(),
tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb iterated_sigmoid_centered,
tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template,
tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(),
tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_power_transform(),
tfb_rational_quadrature_spline(), tfb_real_nvp_default_template(), tfb_real_nvp(),
tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(),
tfb_scale_matvec_lu(), tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shift(),
tfb_sigmoid(), tfb_sinhsinh(), tfb_softmax_centered(), tfb_sofmaxplus(), tfb_sofsign(),
tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf(),
tfb_weibull()

---

tfb_power_transform  Computes \( Y = g(X) = (1 + X \times c)^{(1 / c)}, \) where \( X \geq -1 / c \)

Description

The power transform maps inputs from \([0, \infty]\) to \([-1/c, \infty]\); this is equivalent to the inverse of this
bijector. This bijector is equivalent to the Exp bijector when \( c=0 \).

Usage

tfb_power_transform(power, validate_args = FALSE, name = "power_transform")
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>power</td>
<td>float scalar indicating the transform power, i.e., $Y = g(X) = (1 + X \cdot c)^{(1 / c)}$ where $c$ is the power.</td>
</tr>
<tr>
<td>validate_args</td>
<td>Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.</td>
</tr>
<tr>
<td>name</td>
<td>name prefixed to Ops created by this class.</td>
</tr>
</tbody>
</table>

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diagonal()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

```

```
**Arguments**

- **bin_widths**: The widths of the spans between subsequent knot $x$ positions, a floating point Tensor. Must be positive, and at least 1-D. Innermost axis must sum to the same value as `bin_heights`. The knot $x$ positions will be a first at `range_min`, followed by knots at `range_min + cumsum(bin_widths,axis=-1)`.

- **bin_heights**: The heights of the spans between subsequent knot $y$ positions, a floating point Tensor. Must be positive, and at least 1-D. Innermost axis must sum to the same value as `bin_widths`. The knot $y$ positions will be a first at `range_min`, followed by knots at `range_min + cumsum(bin_heights,axis=-1)`.

- **knot_slopes**: The slope of the spline at each knot, a floating point Tensor. Must be positive. 1s are implicitly padded for the first and last implicit knots corresponding to `range_min` and `range_min + sum(bin_widths,axis=-1)`. Innermost axis size should be 1 less than that of `bin_widths/bin_heights`, or 1 for broadcasting.

- **range_min**: The $x/y$ position of the first knot, which has implicit slope 1. `range_max` is implicit, and can be computed as `range_min + sum(bin_widths,axis=-1)`. Scalar floating point Tensor.

- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

- **name**: name prefixed to Ops created by this class.

**Details**

Typically this bijector will be used as part of a chain, with splines for trailing $x$ dimensions conditioned on some of the earlier $x$ dimensions, and with the inverse then solved first for unconditioned dimensions, then using conditioning derived from those inverses, and so forth.

For each argument, the innermost axis indexes bins/knots and batch axes index axes of $x/y$ spaces. A RationalQuadraticSpline with a separate transform for each of three dimensions might have `bin_widths` shaped [3, 32]. To use the same spline for each of $x$'s three dimensions we may broadcast against $x$ and use a `bin_widths` parameter shaped [32]. Parameters will be broadcast against each other and against the input $x/y$s, so if we want fixed slopes, we can use kwarg `knot_slopes=1`. A typical recipe for acquiring compatible bin widths and heights would be:

```r
nbins <- unconstrained_vector$shape[-1]
range_min <- 1
range_max <- 1
min_bin_size = 1e-2
scale <- range_max - range_min - nbins * min_bin_size
bin_widths = tf$math$softmax(unconstrained_vector) * scale + min_bin_size
```

**Value**

a bijector instance.

**References**

RealNVP affine coupling layer for vector-valued events

Real NVP models a normalizing flow on a D-dimensional distribution via a single D-d-dimensional conditional distribution (Dinh et al., 2017): 
$$y[d:D] = x[d:D] \cdot \exp(\log\_scale\_fn(x[0:d])) + \text{shift\_fn}(x[0:d]) y[0:d] = x[0:d]$$ 
The last D-d units are scaled and shifted based on the first d units only, while the first d units are 'masked' and left unchanged. Real NVP's shift_and_log_scale_fn computes vector-valued quantities. For scale-and-shift transforms that do not depend on any masked units, i.e. d=0, use the tfb_affine bijector with learned parameters instead. Masking is currently only supported for base distributions with event_ndims=1. For more sophisticated masking schemes like checkerboard or channel-wise masking (Papamakarios et al., 2016), use the tfb_permute bijector to re-order desired masked units into the first d units. For base distributions with event_ndims > 1, use the tfb_reshape bijector to flatten the event shape.

Usage
```
tfb_real_nvp( 
    num_masked,
    shift_and_log_scale_fn,
    is_constant_jacobian = FALSE,
    validate_args = FALSE,
    name = NULL
)```

Arguments

- **num_masked**: integer indicating that the first d units of the event should be masked. Must be in the closed interval [1, D-1], where D is the event size of the base distribution.
shift_and_log_scale_fn

Function which computes shift and log_scale from both the forward domain (x) and the inverse domain (y). Calculation must respect the "autoregressive property". Suggested default: tfb_real_nvp_default_template(hidden_layers=...). Typically the function contains tf$Variables and is wrapped using tf$make_template. Returning NULL for either (both) shift, log_scale is equivalent to (but more efficient than) returning zero.

is_constant_jacobian

Logical, default: FALSE. When TRUE the implementation assumes log_scale does not depend on the forward domain (x) or inverse domain (y) values. (No validation is made; is_constant_jacobian=FALSE is always safe but possibly computationally inefficient.)

validate_args

Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name

name prefixed to Ops created by this class.

Details

Recall that the MAF bijector (Papamakarios et al., 2016) implements a normalizing flow via an autoregressive transformation. MAF and IAF have opposite computational tradeoffs - MAF can train all units in parallel but must sample units sequentially, while IAF must train units sequentially but can sample in parallel. In contrast, Real NVP can compute both forward and inverse computations in parallel. However, the lack of an autoregressive transformations makes it less expressive on a per-bijector basis.

A "valid" shift_and_log_scale_fn must compute each shift (aka loc or "mu" in Papamakarios et al. (2016) and log(scale) (aka "alpha" in Papamakarios et al. (2016)) such that each are broadcastable with the arguments to forward and inverse, i.e., such that the calculations in forward, inverse below are possible. For convenience, real_nvp_default_nvp is offered as a possible shift_and_log_scale_fn function.

NICE (Dinh et al., 2014) is a special case of the Real NVP bijector which discards the scale transformation, resulting in a constant-time inverse-log-determinant-Jacobian. To use a NICE bijector instead of Real NVP, shift_and_log_scale_fn should return (shift, NULL), and is_constant_jacobian should be set to TRUE in the RealNVP constructor. Calling tfb_real_nvp_default_template with shift_only=TRUE returns one such NICE-compatible shift_and_log_scale_fn function.

Caching: the scalar input depth D of the base distribution is not known at construction time. The first call to any of forward(x), inverse(x), inverse_log_det_jacobian(x), or forward_log_det_jacobian(x) memoizes D, which is re-used in subsequent calls. This shape must be known prior to graph execution (which is the case if using tf$Layers).

Value

a bijector instance.

References

tfb_real_nvp_default_template

Build a scale-and-shift function using a multi-layer neural network

Description

This will be wrapped in a make_template to ensure the variables are only created once. It takes the d-dimensional input \( x[0:d] \) and returns the \( D-d \) dimensional outputs \( \text{loc} \) ("mu") and \( \text{log_scale} \) ("alpha").

Usage

```r
tfb_real_nvp_default_template(  
  hidden_layers,  
  shift_only = FALSE,  
  activation = tf$nn$relu,  
  name = NULL,  
  ...)
```

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_real_nvp_default_template

Arguments

- **hidden_layers**: list-like of non-negative integer. scalars indicating the number of units in each hidden layer. Default: `list(512,512)`.
- **shift_only**: logical indicating if only the shift term shall be computed (i.e. NICE bijector). Default: `FALSE`.
- **activation**: Activation function (callable). Explicitly setting to NULL implies a linear activation.
- **name**: A name for ops managed by this function. Default: "tfb_real_nvp_default_template".
- ... tf$layers$dense arguments

Details

The default template does not support conditioning and will raise an exception if condition_kwargs are passed to it. To use conditioning in real nvp bijector, implement a conditioned shift/scale template that handles the condition_kwargs.

Value

list of:

- **shift**: Float-like Tensor of shift terms
- **log_scale**: Float-like Tensor of log(scale) terms

References


See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`.

- `tfb_matvec_lu()`
- `tfb_normal_cdf()`
- `tfb_ordered()`
- `tfb_pad()`
- `tfb_permute()`
- `tfb_power_transform()`
- `tfb_rational_quadratic_spline()`
- `tfb_real_nvp()`
- `tfb_reciprocal()`
- `tfb_reshape()`
- `tfb_scale_matvec_diagonal()`
- `tfb_scale_matvec_linear_operator()`
- `tfb_scale_matvec_tri_l()`
- `tfb_scale_tri_l()`
- `tfb_shift()`
- `tfb_sigmoid()`
- `tfb_sinh_arcsinh()`
- `tfb_softmax_centered()`
- `tfb_softplus()`
- `tfb_softsign()`
- `tfb_square()`
- `tfb_tanh()`
- `tfb_transform_diagonal()`
- `tfb_transpose()`
- `tfb_weibull_cdf()`
- `tfb_weibull()`
tfb_reciprocal

A Bijector that computes \( b(x) = \frac{1}{x} \)

Description

A Bijector that computes \( b(x) = \frac{1}{x} \)

Usage

```r
tfb_reciprocal(validate_args = FALSE, name = "reciprocal")
```

Arguments

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

Value

A bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
Reshapes the event_shape of a Tensor

Description

The semantics generally follow that of tf$reshape(), with a few differences:

- The user must provide both the input and output shape, so that the transformation can be inverted. If an input shape is not specified, the default assumes a vector-shaped input, i.e., event_shape_in = list(-1).
- The Reshape bijector automatically broadcasts over the leftmost dimensions of its input (sample_shape and batch_shape); only the rightmost event_ndims_in dimensions are reshaped. The number of dimensions to reshape is inferred from the provided event_shape_in (event_ndims_in = length(event_shape_in)).

Usage

```r
tfb_reshape(
  event_shape_out,
  event_shape_in = c(-1),
  validate_args = FALSE,
  name = NULL
)
```

Arguments

- event_shape_out: An integer-like vector-shaped Tensor representing the event shape of the transformed output.
- event_shape_in: An optional integer-like vector-shape Tensor representing the event shape of the input. This is required in order to define inverse operations; the default of list(-1) assumes a vector-shaped input.
- validate_args: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- name: name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expml(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular().
tfb_scale

Compute \( Y = g(X; \text{scale}) = \text{scale} \times X \).

**Description**

Examples:

\[
Y \leftarrow 2 \times X \\
b \leftarrow \text{tfb\_scale}(\text{scale} = 2)
\]

**Usage**

```

tfb\_scale(scale, validate\_args = FALSE, name = "scale")
```

**Arguments**

- `scale`
  - Floating-point Tensor.
- `validate\_args`
  - Logical, default FALSE. Whether to validate input with asserts. If `validate\_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name`
  - name prefixed to Ops created by this class.

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb\_forward()`, `tfb\_inverse()`, `tfb\_inverse\_log\_det\_jacobian()`.

Other bijectors: `tfb\_absolute\_value()`, `tfb\_affine\_linear\_operator()`, `tfb\_affine\_scalar()`, `tfb\_affine()`, `tfb\_batch\_normalization()`, `tfb\_blockwise()`, `tfb\_chain()`, `tfb\_cholesky\_outer\_product()`, `tfb\_cholesky\_to\_inv\_cholesky()`, `tfb\_correlation\_cholesky()`, `tfb\_cumsum()`, `tfb\_discrete\_cosine\_transform()`, `tfb\_expml()`, `tfb\_exp()`, `tfb\_ffjord()`, `tfb\_fill\_scale\_tri\_l()`, `tfb\_fill\_triangular()`, `tfb\_gumbel\_cdf()`, `tfb\_gumbel()`, `tfb\_identity()`, `tfb\_inverse()`, `tfb\_iterated\_sigmoid\_centered()`, `tfb\_kumaraswamy\_cdf()`, `tfb\_kumaraswamy()`}, `tfb\_lambert\_w\_tail()`, `tfb\_masked\_autoregressive\_default\_template()`, `tfb\_masked\_autoregressive\_flow()`, `tfb\_masked\_dense()`, `tfb\_matrix\_inverse\_tri\_l()`, `tfb\_matvec\_lu()`, `tfb\_normal\_cdf()`, `tfb\_ordered()`, `tfb\_pad()`}, `tfb\_permute()`, `tfb\_power\_transform()`, `tfb\_rational\_quadratic\_spline()`, `tfb\_real\_nvp\_default\_template()`, `tfb\_real\_nvp()`, `tfb\_relu()`, `tfb\_scale\_matvec\_diag()`, `tfb\_scale\_matvec\_linear\_operator()`, `tfb\_scale\_matvec\_lu()`}, `tfb\_scale\_matvec\_tri\_l()`}, `tfb\_scale\_tri\_l()`}, `tfb\_scale()`}, `tfb\_shift()`}, `tfb\_sigmoid()`}, `tfb\_sinh\_arcsinh()`}, `tfb\_softmax\_centered()`}, `tfb\_softplus()`}, `tfb\_softsign()`}, `tfb\_square()`}, `tfb\_tanh()`}, `tfb\_transform\_diagonal()`}, `tfb\_transpose()`}, `tfb\_weibull\_cdf()`}, `tfb\_weibull()`}
tfb_scale_matvec_diag  Compute $Y = g(X; \text{scale}) = \text{scale} \ @ X$

Description

In TF parlance, the scale term is logically equivalent to:

$$\text{scale} = \text{tf$diag$(scale_{diag})}$$

The scale term is applied without materializing a full dense matrix.

Usage

```r
tfb_scale_matvec_diag(
  scale_diag,
  adjoint = FALSE,
  validate_args = FALSE,
  name = "scale_matvec_diag",
  dtype = NULL
)
```

Arguments

- `scale_diag` Floating-point Tensor representing the diagonal matrix. `scale_diag` has shape [N1, N2, ... k], which represents a k x k diagonal matrix.
- `adjoint` logical indicating whether to use the scale matrix as specified or its adjoint. Default value: FALSE.
- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.
- `dtype` tf$DType to prefer when converting args to Tensors. Else, we fall back to a common dtype inferred from the args, finally falling back to float32.

Value

A bijector instance.
See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

Compute $Y = g(X; \text{scale}) = \text{scale} @ X$.

Description

`scale` is a `LinearOperator`. If $X$ is a scalar then the forward transformation is: $\text{scale} \times X$ where $\times$ denotes broadcasted elementwise product.

Usage

```r
tfb_scale_matvec_linear_operator(
  scale,
  adjoint = FALSE,
  validate_args = FALSE,
  name = "scale_matvec_linear_operator"
)
```

Arguments

- **scale**: Subclass of `LinearOperator`. Represents the (batch, non-singular) linear transformation by which the `Bijector` transforms inputs.
- **adjoint**: logical indicating whether to use the scale matrix as specified or its adjoint. Default value: `FALSE`.
- **validate_args**: Logical, default `FALSE`. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.
tfb_scale_matvec_lu

Matrix-vector multiply using LU decomposition.

Description
This bijector is identical to the "Convolution1x1" used in Glow (Kingma and Dhariwal, 2018).

Usage
```r
tfb_scale_matvec_lu(
  lower_upper,
  permutation,
  validate_args = FALSE,
  name = NULL
)
```

Arguments
- `lower_upper`: The LU factorization as returned by `tf$linalg$lu`.
- `permutation`: The LU factorization permutation as returned by `tf$linalg$lu`.
- `validate_args`: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name`: name prefixed to Ops created by this class.

Value
a bijector instance.
tfb_scale_matvec_tri_l

Compute $Y = g(X; \text{scale}) = \text{scale} \odot X$.

Description

The scale term is presumed lower-triangular and non-singular (i.e., no zeros on the diagonal), which permits efficient determinant calculation (linear in matrix dimension, instead of cubic).

Usage

```r
tfb_scale_matvec_tri_l(
  scale_tril,
  adjoint = FALSE,
  validate_args = FALSE,
  name = "scale_matvec_tril",
  dtype = NULL
)
```

Arguments

- `scale_tril`: Floating-point Tensor representing the lower triangular matrix. `scale_tril` has shape `[N1, N2, ..., k, k]`, which represents a `k x k` lower triangular matrix. When `NULL` no `scale_tril` term is added to `scale`. The upper triangular elements above the diagonal are ignored.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tril()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tril()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_tril()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

References

tfb_scale_tri_l

adjoint logical indicating whether to use the scale matrix as specified or its adjoint. Note that lower-triangularity is taken into account first: the region above the diagonal of scale_tril is treated as zero (irrespective of the adjoint setting). A lower-triangular input with adjoint=TRUE will behave like an upper triangular transform. Default value: FALSE.

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.

dtype tf$DType$ to prefer when converting args to Tensors. Else, we fall back to a common dtype inferred from the args, finally falling back to float32.

Value a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(), tfb матрица линейного обратного произведения (), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(), tfb_rational_quadratic_spline(), tfb_real_nvp_default_template(), tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(), tfb_scale_matvec_diag(), tfb_scale_tri_l(), tfb_scale(), tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_softmax_centered(), tfb_sofplus(), tfb_log_sigmoid(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()

tfb_scale_tri_l  Transforms unconstrained vectors to TriL matrices with positive diagonal

Description

This is implemented as a tfb_chain of tfb_fill_triangular followed by tfb_transform_diagonal, and provided mostly as a convenience. The default setup is somewhat opinionated, using a Softplus transformation followed by a small shift (1e-5) which attempts to avoid numerical issues from zeros on the diagonal.
tfb_scale_tri_l

Usage

tfb_scale_tri_l(
    diag_bijector = NULL,
    diag_shift = 1e-05,
    validate_args = FALSE,
    name = "scale_tril"
)

Arguments

diag_bijector Bijector instance, used to transform the output diagonal to be positive. Default value: NULL (i.e., tfb_softplus()).
diag_shift Float value broadcastable and added to all diagonal entries after applying the diag_bijector. Setting a positive value forces the output diagonal entries to be positive, but prevents inverting the transformation for matrices with diagonal entries less than this value. Default value: 1e-5.
validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
name name prefixed to Ops created by this class.

Value

A bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(),
tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(),
tfb_expm1(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(),
tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inplace(), tfb_invert(), tfb_iterated_sigmoid_centered(),
tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(),
tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(),
tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(),
tfb_rational_quadratic_spline(), tfb_real_nvp_default_template(), tfb_real_nvp(),
tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(),
tfb_scale_matvec_lu(), tfb_scale_matvec_tri_l(), tfb_scale(), tfb_shift(), tfb_sigmoid(),
tfb_sinh_arcsinh(), tfb_softmax_centered(), tfb_softplus(), tfb_softsign(), tfb_square(),
tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()
Compute $Y = g(X; \text{shift}) = X + \text{shift}$. 

Description

where \text{shift} is a numeric Tensor.

Usage

```r
tfb_shift(shift, validate_args = FALSE, name = "shift")
```

Arguments

- **shift**: floating-point tensor
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_chol()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_invariant()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_orthogonal()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_sigmoid

Computes \( Y = g(X) = 1 / (1 + \exp(-X)) \)

**Description**

Computes \( Y = g(X) = 1 / (1 + \exp(-X)) \)

**Usage**

```
tfb_sigmoid(validate_args = FALSE, name = "sigmoid")
```

**Arguments**

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**Value**

A bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_cholesky_to_inverse_cholesky()`, `tfb_corr_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_expml()`, `tfb_ffjord()`, `tfb_fill_scale_tril()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_independent()`, `tfb_inverse()`, `tfb_interated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tril()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tril()`, `tfb_scale_tril()`, `tfb_shift()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_sinh_arcsinh

Computes \( Y = g(X) = \sinh\left( (\text{Arcsinh}(X) + \text{skewness}) * \text{tailweight} \right) \)

Description

For skewness in (-inf, inf) and tailweight in (0, inf), this transformation is a diffeomorphism of the real line (-inf, inf). The inverse transform is \( X = g^{-1}(Y) = \sinh\left( \text{Arcsinh}(Y) / \text{tailweight} - \text{skewness} \right) \). The SinhArcsinh transformation of the Normal is described in Sinh-arcsinh distributions.

Usage

```r
tfb_sinh_arcsinh(
  skewness = NULL,
  tailweight = NULL,
  validate_args = FALSE,
  name = "SinhArcsinh"
)
```

Arguments

- **skewness**: Skewness parameter. Float-type Tensor. Default is 0 of type float32.
- **tailweight**: Tailweight parameter. Positive Tensor of same dtype as skewness and broadcastable shape. Default is 1 of type float32.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Details

This Bijector allows a similar transformation of any distribution supported on (-inf, inf).

Value

a bijector instance.

Meaning of the parameters

- If skewness = 0 and tailweight = 1, this transform is the identity.
- Positive (negative) skewness leads to positive (negative) skew.
- Positive skew means, for unimodal X centered at zero, the mode of Y is "tilted" to the right.
- Larger (smaller) tailweight leads to fatter (thinner) tails.
• Fatter tails mean larger values of \(|Y|\) become more likely.
• If \(X\) is a unit Normal, \(\text{tailweight} < 1\) leads to a distribution that is “flat” around \(Y = 0\), and a very steep drop-off in the tails.
• If \(X\) is a unit Normal, \(\text{tailweight} > 1\) leads to a distribution more peaked at the mode with heavier tails. To see the argument about the tails, note that for \(|X| \gg 1\) and \(|X| \gg (|\text{skewness}| \ast \text{tailweight})\),

\[
Y \approx 0.5 \ast X^{\text{tailweight}} \ast e^{(\text{sign}(X) \ast \text{skewness} \ast \text{tailweight})}.
\]

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inverse()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

tfb_softmax_centered  Computes \(Y = g(X) = \exp([X 0]) / \sum(\exp([X 0]))\)

Description

To implement `softmax` as a bijection, the forward transformation appends a value to the input and the inverse removes this coordinate. The appended coordinate represents a pivot, e.g., \(\text{softmax}(x) = \exp(x-c) / \sum(\exp(x-c))\) where \(c\) is the implicit last coordinate.

Usage

```r
tfb_softmax_centered(validate_args = FALSE, name = "softmax_centered")
```

Arguments

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

Details

At first blush it may seem like the Invariance of domain theorem implies this implementation is not a bijection. However, the appended dimension makes the (forward) image non-open and the theorem does not directly apply.
Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inverse()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

```
| tfb_softplus | Computes \(Y = g(X) = \log(1 + \exp(X))\) |
```

Description

The softplus Bijector has the following two useful properties:

- The domain is the positive real numbers
- \(\text{softplus}(x)\) approx \(x\), for large \(x\), so it does not overflow as easily as the Exp Bijector.

Usage

```
tfb_softplus(hinge_softness = NULL, validate_args = FALSE, name = "softplus")
```

Arguments

- **hinge_softness** Nonzero floating point Tensor. Controls the softness of what would otherwise be a kink at the origin. Default is 1.0.
- **validate_args** Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name** name prefixed to Ops created by this class.
Details

The optional nonzero hinge\_softness parameter changes the transition at zero. With hinge\_softness = c, the bijector is:

\[
f_c(x) := c \times g(x / c) = c \times \log[1 + \exp(x / c)].
\]

For large \(x \gg 1\),

\[
c \times \log[1 + \exp(x / c)] \approx c \times \log[\exp(x / c)] = x
\]

so the behavior for large \(x\) is the same as the standard softplus.

As \(c > 0\) approaches 0 from the right, \(f_c(x)\) becomes less and less soft, approaching \(\max(0, x)\).

* \(c = 1\) is the default.
* \(c > 0\) but small means \(f(x)\) approx ReLu\((x) = \max(0, x)\).
* \(c < 0\) flips sign and reflects around the y-axis: \(f_{-c}(x) = -f_c(-x)\).
* \(c = 0\) results in a non-bijective transformation and triggers an exception.

Note: log(.) and exp(.) are applied element-wise but the Jacobian is a reduction over the event space.

\[
[1 + \exp(x / c)]: R:10\%20+\exp(x/20/%20c)
\]
\[
[1 + \exp(x / c)]: R:10\%20+\exp(x/20/%20c)
\]
\[
[\exp(x / c)]: R:\exp(x/20/%20c)
\]

Value

a bijector instance.

See Also

For usage examples see tfb\_forward(), tfb\_inverse(), tfb\_inverse\_log\_det\_jacobian().

Other bijectors: tfb\_absolute\_value(), tfb\_affine\_linear\_operator(), tfb\_affine\_scalar(),

tfb\_affine\_inverse(), tfb\_batch\_normalization\(), tfb\_blockwise\(), tfb\_chain\(), tfb\_cholesky\_outer\_product\(),

tfb\_cholesky\_to\_inv\_cholesky\(), tfb\_correlation\_cholesky\(), tfb\_cumsum\(), tfb\_discrete\_cosine\_transform\(),

tfb\_exp\_1\(), tfb\_exp\(), tfb\_ffjord\(), tfb\_fill\_scale\_tri\_l\(),

tfb\_fill\_triangular\(), tfb\_gumbel\_cdf\(), tfb\_gumbel\(), tfb\_identity\(),

tfb\_inverse\(), tfb\_invert\(), tfb\_iterated\_sigmoid\_centered\(),

tfb\_kumaraswamy\_cdf\(),

tfb\_kumaraswamy\(),
tfb\_lambert\_w\_tail\(),

tfb\_masked\_autoregressive\_default\_template\(),

tfb\_masked\_autoregressive\_flow\(),

tfb\_masked\_dense\(),

tfb\_matrix\_inverse\_tri\_l\(),

tfb\_matvec\_lu\(),

tfb\_normal\_cdf\(),

tfb\_ordered\(),

tfb\_pad\(),

tfb\_permute\(),

tfb\_power\_transform\(),

tfb\_rational\_quadratic\_spline\(),

tfb\_real\_nvp\_default\_template\(),

tfb\_real\_nvp\(),

tfb\_reciprocal\(),

tfb\_reshape\(),

tfb\_scale\_matvec\_diag\(),

tfb\_scale\_matvec\_linear\_operator\(),

tfb\_scale\_matvec\_lu\(),

tfb\_scale\_matvec\_tri\_l\(),

tfb\_scale\_tri\_l\(),

tfb\_shift\(),

tfb\_sigmoid\(),

tfb\_sinh\_arcsinh\(),

tfb\_softmax\_centered\(),

tfb\_softsign\(),

tfb\_square\(),

tfb\_tanh\(),

tfb\_transform\_diagonal\(),

tfb\_transpose\(),

tfb\_weibull\_cdf\(),

tfb\_weibull\()
tfb_softsign

Computes \( Y = \text{g}(X) = X / (1 + |X|) \)

**Description**

The softsign Bijector has the following two useful properties:

- The domain is all real numbers
- softsign(x) approx sgn(x), for large |x|.

**Usage**

```r
tfb_softsign(validate_args = FALSE, name = "softsign")
```

**Arguments**

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matrix_normal_diag()`, `tfb_matrix_diag()`, `tfb_matrix_tril()`, `tfb_matrix_inverse()`, `tfb_matrix_square()`, `tfb_moyal_cdf()`, `tfb_moyal()`, `tfb_permute()`, `tfb_dropout()`, `tfb_dropout_with啜()`, `tfb_power_transform()`, `tfb_rank()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_smoothmax_centered()`, `tfb_softplus()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_square

**Description**

$g$ is a bijection between the non-negative real numbers ($\mathbb{R}_+$) and the non-negative real numbers.

**Usage**

```r
tfb_square(validate_args = FALSE, name = "square")
```

**Arguments**

- `validate_args`: Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name`: name prefixed to Ops created by this class.

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`,
`tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`,
`tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`,
`tfb_exp()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`,
`tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_infinite()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`,
`tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`,
`tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`,
`tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`,
`tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`,
`tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`,
`tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shift()`,
`tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`,
`tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_tanh

**Computes** Y = tanh(X)

---

**Description**

Y = tanh(X), therefore Y in (-1, 1).

**Usage**

tfb_tanh(validate_args = FALSE, name = "tanh")

**Arguments**

- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

**Details**

This can be achieved by an affine transform of the Sigmoid bijector, i.e., it is equivalent to
tfb_chain(list(tfb_affine(shift = -1, scale = 2), tfb_sigmoid(), tfb_affine(scale = 2)))

However, using the Tanh bijector directly is slightly faster and more numerically stable.

**Value**

A bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_infer()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_maskeddense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
Applies a Bijector to the diagonal of a matrix

Description
Applies a Bijector to the diagonal of a matrix

Usage
```
import tensorflow as tf

tfb_transform_diagonal(
    diag_bijector,
    validate_args = FALSE,
    name = "transform_diagonal"
)
```

Arguments
- `diag_bijector`: Bijector instance used to transform the diagonal.
- `validate_args`: Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name`: name prefixed to Ops created by this class.

Value
a bijector instance.

See Also
For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_square()`, `tfb_tanh()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
Computes $Y = g(X) = \text{transpose}_\text{rightmost}_d(X, \text{rightmost}_\text{perm})$

**Description**

This bijector is semantically similar to `tf.transpose` except that it transposes only the rightmost "event" dimensions. That is, unlike `tf.transpose` the perm argument is itself a permutation of `tf.range(rightmost_transposed_ndims)` rather than `tf.range(tf.rank(x))`, i.e., users specify the (rightmost) dimensions to permute, not all dimensions.

**Usage**

```python
tfb_transpose(
    perm = NULL,
    rightmost_transposed_ndims = NULL,
    validate_args = FALSE,
    name = "transpose"
)
```

**Arguments**

- `perm` Positive integer vector-shaped Tensor representing permutation of rightmost dims (for forward transformation). Note that the 0th index represents the first of the rightmost dims and the largest value must be `rightmost_transposed_ndims - 1` and corresponds to `tf.rank(x) - 1`. Only one of `perm` and `rightmost_transposed_ndims` can (and must) be specified. Default value: `tf.range(start=rightmost_transposed_ndims, limit=-1, delta=-1)`.

- `rightmost_transposed_ndims` Positive integer scalar-shaped Tensor representing the number of rightmost dimensions to permute. Only one of `perm` and `rightmost_transposed_ndims` can (and must) be specified. Default value: `tf.size(perm)`.

- `validate_args` Logical, default `FALSE`. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed.

- `name` name prefixed to Ops created by this class.

**Details**

The actual (forward) transformation is:

```python
sample_batch_ndims <- tf.rank(x) - tf.size(perm)
perm = tf.concat(list(tf.range(sample_batch_ndims), sample_batch_ndims + perm), axis=0) tf.transpose(x, perm)
```

**Value**

a bijector instance.
tfb_weibull

Computes $Y = g(X) = 1 - \exp((-X / \text{scale}) ** \text{concentration})$ where $X >= 0$

Description

This bijector maps inputs from $[0, \infty)$ to $[0, 1]$. The inverse of the bijector applied to a uniform random variable $X \sim U(0, 1)$ gives back a random variable with the Weibull distribution:

Usage

tfb_weibull(
    scale = 1,
    concentration = 1,
    validate_args = FALSE,
    name = "weibull"
)

Arguments

scale Positive Float-type Tensor that is the same dtype and is broadcastable with concentration. This is $l$ in $Y = g(X) = 1 - \exp((-x / \text{scale}) ** \text{concentration})$.

concentration Positive Float-type Tensor that is the same dtype and is broadcastable with scale. This is $k$ in $Y = g(X) = 1 - \exp((-x / \text{scale}) ** \text{concentration})$.

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.
tfb_weibull_cdf

Details

\[ Y \sim \text{Weibull}(\text{scale}, \text{concentration}) \]
\[ \text{pdf}(y; \text{scale}, \text{concentration}, y \geq 0) = \frac{(\text{concentration} / \text{scale}) \cdot (y / \text{scale})^{\text{concentration} - 1} \cdot \exp(-(y / \text{scale})^{\text{concentration}})}{\text{scale}^{\text{concentration}}} \]

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expm1(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triaangular(), tfb_gambel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_masked_triangular(), tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(), tfb_rational_quadratic_spline(), tfb_real_nvp_default_template(), tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(), tfb_scale_matvec_lu(), tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_softmax_centered(), tfb_softplus(), tfb.softsign(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf()

---

tfb_weibull_cdf

Compute \( Y = g(X) = 1 - \exp(-X / \text{scale})^{\text{concentration}} \), \( X \geq 0 \).

Description

This bijector maps inputs from \([0, \infty]\) to \([0, 1]\). The inverse of the bijector applied to a uniform random variable \( X \sim U(0, 1) \) gives back a random variable with the Weibull distribution:

\[ Y \sim \text{Weibull}(\text{scale}, \text{concentration}) \]
\[ \text{pdf}(y; \text{scale}, \text{concentration}, y \geq 0) = \frac{(\text{concentration} / \text{scale}) \cdot (y / \text{scale})^{\text{concentration} - 1} \cdot \exp(-(y / \text{scale})^{\text{concentration}})}{\text{scale}^{\text{concentration}}} \]

Usage

```python
tfb_weibull_cdf(
    scale = 1,
    concentration = 1,
    validate_args = FALSE,
    name = "weibull_cdf"
)```
Arguments

scale  Positive Float-type Tensor that is the same dtype and is broadcastable with concentration. This is $l$ in $Y = g(X) = 1 - \exp((-x / l)^{k})$.

concentration  Positive Float-type Tensor that is the same dtype and is broadcastable with scale. This is $k$ in $Y = g(X) = 1 - \exp((-x / l)^{k})$.

validate_args  Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name  name prefixed to Ops created by this class.

Details

Likewise, the forward of this bijector is the Weibull distribution CDF.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform
tfb_expml(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(),
tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb_iterated_sigmoid_centered,
tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template,
tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(),
tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(),
tfb_rational_quadratic_spline(), tfb_real_nvp_default_template(), tfb_real_nvp(),
tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(),
tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull()
tfd_autoregressive

Usage

tfd_autoregressive(
    distribution_fn,
    sample0 = NULL,
    num_steps = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Autoregressive"
)

Arguments

distribution_fn
    Function which constructs a tfd$Distribution-like instance from a Tensor
    (e.g., sample0). The function must respect the "autoregressive property", i.e.,
    there exists a permutation of event such that each coordinate is a diffeomorphic
    function of on preceding coordinates.

sample0
    Initial input to distribution_fn; used to build the distribution in __init__
    which in turn specifies this distribution’s properties, e.g., event_shape, batch_shape,
    dtype. If unspecified, then distribution_fn should be default constructable.

num_steps
    Number of times distribution_fn is composed from samples, e.g., num_steps=2
    implies distribution_fn(distribution_fn(sample0)$sample(n))$sample().

validate_args
    Logical, default FALSE. When TRUE distribution parameters are checked for
    validity despite possibly degrading runtime performance. When FALSE invalid
    inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
    Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
    the value NaN to indicate the result is undefined. When FALSE, an exception is
    raised if one or more of the statistic’s batch members are undefined.

name
    name prefixed to Ops created by this class.

Details

Regarding terminology, "Autoregressive models decompose the joint density as a product of con-
ditionals, and model each conditional in turn. Normalizing flows transform a base density (e.g. a
standard Gaussian) into the target density by an invertible transformation with tractable Jacobian."
(Papamakarios et al., 2016)

In other words, the "autoregressive property" is equivalent to the decomposition, p(x) = prod{ p(x[i] | x[0:i]) : i=0, ..., d }. The provided shift_and_log_scale_fn, tfb_masked_autoregressive_default_template, achieves
this property by zeroing out weights in its masked_dense layers. Practically speaking the autore-
gressive property means that there exists a permutation of the event coordinates such that each
coordinate is a diffeomorphic function of only preceding coordinates (van den Oord et al., 2016).

Mathematical Details

The probability function is

prob(x; fn, n) = fn(x).prob(x)
And a sample is generated by

\[ x = \text{fn}(\ldots\text{fn}(\text{fn}(x_0).\text{sample()}).\text{sample()}).\text{sample()} \]

where the ellipses (...) represent \( n-2 \) composed calls to \text{fn}, \text{fn} constructs a \text{tfd$Distribution}-\text{like instance}, and \( x_0 \) is a fixed initializing Tensor.

Value

a distribution instance.

References


See Also

For usage examples see e.g. \text{tfd_sample()}, \text{tfd_log_prob()}, \text{tfd_mean()}.

Other distributions: \text{tfd_batch_reshape()}, \text{tfd_bernoulli()}, \text{tfd_beta_binomial()}, \text{tfd_beta()}, \text{tfd_binomial()}, \text{tfd_categorical()}, \text{tfd_cauchy()}, \text{tfd_chi2()}, \text{tfd_chi()}, \text{tfd_cholesky_lkj()}, \text{tfd_deterministic()}, \text{tfd_dirichlet_multinomial()}, \text{tfd_dirichlet()}, \text{tfd_empirical()}, \text{tfd_exponential()}, \text{tfd_gamma_gamma()}, \text{tfd_gamma()}, \text{tfd_gaussian_process_regression_model()}, \text{tfd_gaussian_process()}, \text{tfd_geometric()}, \text{tfd_gumbel()}, \text{tfd_half_cauchy()}, \text{tfd_half_normal()}, \text{tfd_hidden_markov_model()}, \text{tfd_horseshoe()}, \text{tfd_independent()}, \text{tfd_inverse_gamma()}, \text{tfd_inverse_gaussian()}, \text{tfd_joint_distribution_named()}, \text{tfd_joint_distribution_sequential()}, \text{tfd_kumaraswamy()}, \text{tfd_laplace()}, \text{tfd_linear_gaussian_state_space_model()}, \text{tfd_lkj()}, \text{tfd_log_normal()}, \text{tfd_logistic()}, \text{tfd_mixture_same_family()}, \text{tfd_mixture()}, \text{tfd_multinomial()}, \text{tfd_multivariate_normal_diag_plus_low_rank()}, \text{tfd_multivariate_normal_diag()}, \text{tfd_multivariate_normal()} ,\text{tfd_multivariate_normal_linear_operator()}, \text{tfd_multivariate_normal_tri_l()}, \text{tfd_multivariate_student_t()}, \text{tfd_negative_binomial()}, \text{tfd_normal()}, \text{tfd_one_hot_categorical()}, \text{tfd_pareto()}, \text{tfd_pixel_cnn()}, \text{tfd_poisson_log_normal_quadrature_compound()}, \text{tfd_probit_bernoulli()}, \text{tfd_quantized()}, \text{tfd_rela...
tfd_batch_reshape  Batch-Reshaping distribution

Description

This "meta-distribution" reshares the batch dimensions of another distribution.

Usage

```r
tfd_batch_reshape(
  distribution,
  batch_shape,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = NULL
)
```

Arguments

- **distribution**: The base distribution instance to reshape. Typically an instance of `Distribution`.
- **batch_shape**: Positive integer-like vector-shaped Tensor representing the new shape of the batch dimensions. Up to one dimension may contain -1, meaning the remainder of the batch size.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Value

A distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`, `tfd_inverse_gaussian()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential()`.
Description

The Bernoulli distribution with `probs` parameter, i.e., the probability of a 1 outcome (vs a 0 outcome).

Usage

tfd_bernoulli(
    logits = NULL,
    probs = NULL,
    dtype = tf$int32,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Bernoulli"
)

Arguments

logits
An N-D Tensor representing the log-odds of a 1 event. Each entry in the Tensor parametrizes an independent Bernoulli distribution where the probability of an event is \text{sigmoid}(logits). Only one of logits or probs should be passed in.

probs
An N-D Tensor representing the probability of a 1 event. Each entry in the Tensor parameterizes an independent Bernoulli distribution. Only one of logits or probs should be passed in.

dtype
The type of the event samples. Default: int32.

validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic's batch members are undefined.

name
name prefixed to Ops created by this class.

Value
a distribution instance.

See Also
For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.
Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_beta_binomial()`, `tfd_beta()`,
`tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chi()`, `tfd_cholesky_lkj()`,
`tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`,
`tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`,
`tfd_gaussian_process()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`,
`tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`,
`tfd_inverse_gaussian()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential()`,
`tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`,
`tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multinomial()`,
`tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`, `tfd_multivariate_normal_linear_operator()`,
`tfd_multivariate_normal_tri_l()`, `tfd_multivariate_student_t()`,
`tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`,
`tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`,
`tfd_sinh_arcsinh()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`,
`tfd_triangular()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_von_mises_fisher()`, `tfd_von_mises()`,
`tfd_wishart()`, `tfd_zipf()`

tfd_beta

<table>
<thead>
<tr>
<th>Beta distribution</th>
</tr>
</thead>
</table>

Description
The Beta distribution is defined over the (0, 1) interval using parameters concentration1 (aka
"alpha") and concentration0 (aka "beta").

Usage
```r
  tfd_beta(
    concentration1 = NULL,
    concentration0 = NULL,
```
validate_args = FALSE,
allow_nan_stats = TRUE,
name = "Beta"
)

Arguments

concentration1 Positive floating-point Tensor indicating mean number of successes; aka "alpha". Implies self$dtype and self$batch_shape, i.e., concentration1$shape = [N1, N2, ..., Nm] = self$batch_shape.

concentration0 Positive floating-point Tensor indicating mean number of failures; aka "beta". Otherwise has same semantics as concentration1.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

$$
pdf(x; \alpha, \beta) = x^{\alpha - 1}(1 - x)^{\beta - 1} / Z
$$

$$
Z = \text{Gamma}(\alpha) \text{Gamma}(\beta) / \text{Gamma}(\alpha + \beta)
$$

where:

- concentration1 = \alpha,
- concentration0 = \beta,
- Z is the normalization constant, and,
- Gamma is the gamma function. The concentration parameters represent mean total counts of a 1 or a 0, i.e.,

$$
\text{concentration1} = \alpha = \text{mean} \ast \text{total_concentration}
\text{concentration0} = \beta = (1. - \text{mean}) \ast \text{total_concentration}
$$

where mean in (0, 1) and total_concentration is a positive real number representing a mean total_count = concentration1 + concentration0. Distribution parameters are automatically broadcast in all functions; see examples for details. Warning: The samples can be zero due to finite precision. This happens more often when some of the concentrations are very small. Make sure to round the samples to np$finfo(dtype)$tiny before computing the density. Samples of this distribution are reparameterized (pathwise differentiable). The derivatives are computed using the approach described in the paper Michael Figurnov, Shakir Mohamed, Andriy Mnih. Implicit Reparameterization Gradients, 2018
The Beta-Binomial distribution is parameterized by (a batch of) `total_count` parameters, the number of trials per draw from Binomial distributions where the probabilities of success per trial are drawn from underlying Beta distributions; the Beta distributions are parameterized by `concentration1` (aka 'alpha') and `concentration0` (aka 'beta'). Mathematically, it is (equivalent to) a special case of the Dirichlet-Multinomial over two classes, although the computational representation is slightly different: while the Beta-Binomial is a distribution over the number of successes in `total_count` trials, the two-class Dirichlet-Multinomial is a distribution over the number of successes and failures.
concentration0,
validate_args = FALSE,
allow_nan_stats = TRUE,
name = "BetaBinomial"
)

Arguments

total_count  Non-negative integer-valued tensor, whose dtype is the same as concentration1 and concentration0. The shape is broadcastable to \([N1,..., Nm]\) with \(m \geq 0\). When total_count is broadcast with concentration1 and concentration0, it defines the distribution as a batch of \(N1 \times ... \times Nm\) different Beta-Binomial distributions. Its components should be equal to integer values.

concentration1  Positive floating-point Tensor indicating mean number of successes. Specifically, the expected number of successes is \(\text{total_count} \times \text{concentration1} / (\text{concentration1} + \text{concentration0})\).

concentration0  Positive floating-point Tensor indicating mean number of failures; see description of concentration1 for details.

validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

name  name prefixed to Ops created by this class.

Details

Mathematical Details

The Beta-Binomial is a distribution over the number of successes in total_count independent Binomial trials, with each trial having the same probability of success, the underlying probability being unknown but drawn from a Beta distribution with known parameters. The probability mass function (pmf) is,

\[
\text{pmf}(k; n, a, b) = \frac{\text{Beta}(k + a, n - k + b)}{Z}
\]

\[
Z = \frac{(k! \cdot (n - k)!)}{n!} \cdot \text{Beta}(a, b)
\]

where:

\begin{itemize}
  \item concentration1 = \(a > 0\),
  \item concentration0 = \(b > 0\),
  \item total_count = \(n, n\) a positive integer,
  \item \(n!\) is \(n\) factorial,
  \item \text{Beta}(x, y) = \frac{\text{Gamma}(x) \cdot \text{Gamma}(y)}{\text{Gamma}(x + y)}\) is the beta function, and
  \item \text{Gamma} is the gamma function.
\end{itemize}
Dirichlet-Multinomial is a compound distribution, i.e., its samples are generated as follows.

1. Choose success probabilities: \( \text{probs} \sim \text{Beta}(\text{concentration}_1, \text{concentration}_0) \)
2. Draw integers representing the number of successes: \( \text{counts} \sim \text{Binomial}(\text{total\_count}, \text{probs}) \)

Distribution parameters are automatically broadcast in all functions; see examples for details.

Value

A distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

probs = NULL,
validate_args = FALSE,
allow_nan_stats = TRUE,
name = "Beta"
)

**Arguments**

**total_count**
Non-negative floating point tensor with shape broadcastable to \([N1, ..., Nm]\) with \(m \geq 0\) and the same dtype as \(probs\) or \(logits\). Defines this as a batch of \(N1 \times ... \times Nm\) different Binomial distributions. Its components should be equal to integer values.

**logits**
Floating point tensor representing the log-odds of a positive event with shape broadcastable to \([N1, ..., Nm]\) \(m \geq 0\), and the same dtype as \(total\_count\). Each entry represents logits for the probability of success for independent Binomial distributions. Only one of \(logits\) or \(probs\) should be passed in.

**probs**
Positive floating point tensor with shape broadcastable to \([N1, ..., Nm]\) \(m \geq 0\), \(probs\) in \([0, 1]\). Each entry represents the probability of success for independent Binomial distributions. Only one of \(logits\) or \(probs\) should be passed in.

**validate_args**
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

**allow_nan_stats**
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

**name**
name prefixed to Ops created by this class.

**Details**

**Mathematical Details**
The Binomial is a distribution over the number of 1's in \(total\_count\) independent trials, with each trial having the same probability of 1, i.e., \(probs\).

The probability mass function (pmf) is,

\[
\text{pmf}(k; n, p) = p^k \cdot (1 - p)^{n - k} / Z
\]

\[
Z = k! \cdot (n - k)! / n!
\]

where:

- \(total\_count = n\),
- \(probs = p\),
- \(Z\) is the normalizing constant, and,
- \(n!\) is the factorial of \(n\).

**Value**
a distribution instance.
See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batchreshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`,
`tfd_beta()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_cholesky_lkj()`,
`tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`,
`tfd_exponential()`, `tfd_gaussian()`, `tfd_gaussian_process_regression_model()`,
`tfd_gaussian_process()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`,
`tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gaussian()`,
`tfd_inverse_gaussian()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequence()`,
`tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`,
`tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_mixture()`,
`tfd_mixture()`, `tfd_multinomial()`,
`tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`,
`tfd_multivariate_normal()`,
`tfd_negative_binomial()`, `tfd_normal()`,
`tfd_pareto()`, `tfd_pixel_cnn()`,
`tfd_poisson_log_normal_quadrature_compound()`, `tfd_poisson()`,
`tfd_probit_bernoulli()`,
`tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`,
`tfd_sample_distribution()`,
`tfd_sinh_arcsinh()`, `tfd_student_t_process()`,
`tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`,
`tfd_triangular()`, `tfd_truncated_normal()`, `tfd_uniform()`,
`tfd_variable()`, `tfd_vector_diffeomixture()`,
`tfd_vector_exp()`, `tfd_vector_exponential_diag()`,
`tfd_vector_exponential_linear_operator()`,
`tfd_vector_laplace()`, `tfd_vector_laplace_linear_operator()`,
`tfd_vector_sinh_arcsinh()`,
`tfd_von_mises_fisher()`, `tfd_von_mises()`,
`tfd_wishart_linear_operator()`, `tfd_wishart_tri_l()`,
`tfd_wishart()`, `tfd_zipf()`
samples of distributions will be cast to this dtype. If unspecified, all distributions must have the same dtype. Default value: NULL (i.e., do not cast).

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

tfd_categorical

Categorical distribution over integers

Description

The Categorical distribution is parameterized by either probabilities or log-probabilities of a set of K classes. It is defined over the integers \{0, 1, ..., K-1\}.

Usage

tfd_categorical(
  logits = NULL,
  probs = NULL,
  dtype = tf$int32,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Categorical"
)

Arguments

logits An N-D Tensor, N >= 1, representing the log probabilities of a set of Categorical distributions. The first N-1 dimensions index into a batch of independent distributions and the last dimension represents a vector of logits for each class. Only one of logits or probs should be passed in.
probs
An N-D Tensor, $N \geq 1$, representing the probabilities of a set of Categorical distributions. The first $N-1$ dimensions index into a batch of independent distributions and the last dimension represents a vector of probabilities for each class. Only one of logits or probs should be passed in.

dtype
The type of the event samples (default: int32).

validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

name
name prefixed to Ops created by this class.

Details
The Categorical distribution is closely related to the OneHotCategorical and Multinomial distributions. The Categorical distribution can be intuited as generating samples according to \( \arg\max\{ \text{OneHotCategorical}(\text{probs}) \} \) itself being identical to \( \arg\max\{ \text{Multinomial}(\text{probs}, \text{total_count}=1) \} \).

Mathematical Details
The probability mass function (pmf) is,

\[
\text{pmf}(k; \pi) = \prod_j \pi_j^{[k == j]}
\]

Pitfalls
The number of classes, $K$, must not exceed:

- the largest integer representable by self$\text{dtype}$, i.e., $2^{(\text{mantissa_bits}+1)}$ (IEEE 754),
- the maximum Tensor index, i.e., $2^{31}-1$.

Note: This condition is validated only when validate_args = TRUE.

Value
a distribution instance.

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(),
tfd_beta(), tfd_binomial(), tfd_cauchy(), tfd_chi2(), tfd_chisquare(), tfd_cholesky_lkj(),
tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(),
tfd_exponential(), tfd_gamma(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(),
tfd_lkj(), tfd_mixture_same_family(), tfd_multinomial(), tfd_normal(), tfd_one_hot_categorical(),
tfd_ova(), tfd_poisson(), tfd_student_t(), tfd_tower(), tfd_uniform().
tfd_cauchy

Cauchy distribution with location loc and scale scale

Description
Mathematical details

Usage

```r
tfd_cauchy(
  loc,
  scale,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Cauchy"
)
```

Arguments

- **loc**
  Floating point tensor; the modes of the distribution(s).

- **scale**
  Floating point tensor; the locations of the distribution(s). Must contain only positive values.

- **validate_args**
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**
  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

- **name**
  name prefixed to Ops created by this class.
**Details**

The probability density function (pdf) is,

\[
    \text{pdf}(x; \text{loc}, \text{scale}) = \frac{1}{\pi \text{scale} (1 + z^2)}
    \]

\[
    z = \frac{x - \text{loc}}{\text{scale}}
    \]

where \(\text{loc}\) is the location, and \(\text{scale}\) is the scale. The Cauchy distribution is a member of the location-scale family, i.e. \(Y \sim \text{Cauchy}(\text{loc}, \text{scale})\) is equivalent to,

\[
    X \sim \text{Cauchy}(\text{loc}=0, \text{scale}=1)
    \]

\[
    Y = \text{loc} + \text{scale} \times X
    \]

**Value**

A distribution instance.

**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


---

**tfd_cdf**

*Cumulative distribution function. Given random variable \(X\), the cumulative distribution function \(\text{cdf}\) is: \(\text{cdf}(x) := P[X \leqslant x]\)*

---

**Description**

Cumulative distribution function. Given random variable \(X\), the cumulative distribution function \(\text{cdf}\) is: \(\text{cdf}(x) := P[X \leqslant x]\)
tfd_chi

Chi distribution

Description

The Chi distribution is defined over nonnegative real numbers and uses a degrees of freedom ("df") parameter.

Usage

tfd_chi(df, validate_args = FALSE, allow_nan_stats = TRUE, name = "Chi")

Arguments

df

Floating point tensor, the degrees of freedom of the distribution(s). df must contain only positive values.

validate_args

Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
Mathematical Details
The probability density function (pdf) is,

\[
pdf(x; df, x \geq 0) = x^{(df - 1)} \exp(-0.5 x^2) / Z
\]

\[
Z = 2^{(0.5 df - 1)} \Gamma(0.5 df)
\]

where:

- df denotes the degrees of freedom,
- Z is the normalization constant, and,
- \(\Gamma\) is the gamma function.

The Chi distribution is a transformation of the Chi2 distribution; it is the distribution of the positive
square root of a variable obeying a Chi distribution.

Value
a distribution instance.

See Also
For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.
Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`,
`tfd_beta()`, `tfd_binomial()`, `tfd_catgorical()`, `tfd_cauchy()`, `tfd_cholesky_lkj()`,
`tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`,
`tfd_exponential()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`,
`tfd_gaussian_process()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`,
`tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`,
`tfd_inverse_gaussian()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential()`,
`tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`,
`tfd_log_normal()`, `tfd_logistic()`, `tfd لوحة البيانات عائلة`.
`tfd_mixture()`, `tfd_mixture_same_family()`, `tfd_multivariate_normal_diag()`,
`tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`,
`tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`,
`tfd_multivariate_normal_tril()`, `tfd_multivariate_student_t()`,
`tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`,
`tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature_compound()`, `tfd_poisson()`,
`tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`,
`tfd_sample_distribution()`,
`tfd_sinh_arcsinh()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`,
`tfd_triangular()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_variational_gaussian_process()`,
`tfd_vector_diffeomixture()`, `tfd_vector_exponential_diag()`, `tfd_vector_exponential_linear_operator()`,
`tfd_vector_laplace_diag()`, `tfd_vector_laplace_linear_operator()`, `tfd_vector_sinh_arcsinh_diag()`,
`tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_wishart_linear_operator()`, `tfd_wishart_tri_l()`,
`tfd_wishart()`, `tfd_zipf()`
tfd_chi2

Chi Square distribution

Description
The Chi2 distribution is defined over positive real numbers using a degrees of freedom ("df") parameter.

Usage
tfd_chi2(df, validate_args = FALSE, allow_nan_stats = TRUE, name = "Chi2")

Arguments
df Floating point tensor, the degrees of freedom of the distribution(s). df must contain only positive values.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name name prefixed to Ops created by this class.

Details
Mathematical Details
The probability density function (pdf) is,

\[
pdf(x; df, x > 0) = x^{(0.5 \cdot df - 1)} \exp(-0.5 \cdot x) / Z
\]
\[
Z = 2^{(0.5 \cdot df)} \Gamma(0.5 \cdot df)
\]

where

* df denotes the degrees of freedom,
* Z is the normalization constant, and,
* \( Gamma \) is the gamma function. The Chi2 distribution is a special case of the Gamma distribution, i.e.,

\[
\chi^2(df) = \Gamma(\text{concentration}=0.5 \cdot df, \text{rate}=0.5)
\]

Value
a distribution instance.
**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


---

**tfd_cholesky_lkj**  
The CholeskyLKJ distribution on cholesky factors of correlation matrices

**Description**

This is a one-parameter family of distributions on cholesky factors of correlation matrices. In other words, if $X \sim \text{CholeskyLKJ}(c)$, then $X @ X^T \sim \text{LKJ}(c)$. For more details on the LKJ distribution, see `tfd_lkj`.

**Usage**

```python
  tfd_cholesky_lkj(
      dimension,
      concentration,
      validate_args = FALSE,
      allow_nan_stats = TRUE,
      name = "CholeskyLKJ"
  )
```
Arguments

dimension integer. The dimension of the correlation matrices to sample.

concentration float or double Tensor. The positive concentration parameter of the CholeskyLKJ distributions.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chisq(), tfd_chi(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tril(), tfd_multivariate_student_t(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature_compound(), tfd_poisson(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_normal(), tfd_uniform(), tfd_von_mises_fisher(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_von_mises_fisher(), tfd_von_mises(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()
Description

Covariance is (possibly) defined only for non-scalar-event distributions. For example, for a length-$k$, vector-valued distribution, it is calculated as, $\text{Cov}[i, j] = \text{Covariance}(X_i, X_j) = \mathbb{E}[(X_i - \mathbb{E}[X_i])(X_j - \mathbb{E}[X_j])]$ where $\text{Cov}$ is a (batch of) $k \times k$ matrix, $0 \leq (i, j) < k$, and $\mathbb{E}$ denotes expectation.

Usage

tfd_c covariance(distribution, ...)

Arguments

distribution  The distribution being used.

...  Additional parameters passed to Python.

Details

Alternatively, for non-vector, multivariate distributions (e.g., matrix-valued, Wishart), Covariance shall return a (batch of) matrices under some vectorization of the events, i.e., $\text{Cov}[i, j] = \text{Covariance}(\text{Vec}(X)_i, \text{Vec}(X)_j) = [\text{as above}]$ where $\text{Cov}$ is a (batch of) $k \times k$ matrices, $0 \leq (i, j) < k = \text{reduce_prod(event_shape)}$, and $\text{Vec}$ is some function mapping indices of this distribution’s event dimensions to indices of a length-$k$ vector.

Value

Floating-point Tensor with shape $[B_1, ..., B_n, k, k]$ where the first $n$ dimensions are batch coordinates and $k = \text{reduce_prod(self.event_shape)}$.

See Also

Other distribution methods: tfd_cdf(), tfd_cross_entropy(), tfd_entropy(), tfd_kl_divergence(), tfd_log_cdf(), tfd_log_prob(), tfd_log_survival_function(), tfd_mean(), tfd_mode(), tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_survival_function(), tfd_variance()

Examples

d <- tfd_vector_diffeomixture(
  mix_loc = list(c(0, 1)),
  temperature = list(1),
  distribution = tfd_normal(loc = 0, scale = 1),
  loc = list(NULL, rep(2, 5)),
  scale = list(
    tf$linear$LinearOperatorScaledIdentity(
      num_rows = 5L,
    )
  )
)
multiplier = 1.1,
is_positive_definite = TRUE),
tf$linalg$LinearOperatorDiag(
  diag = seq(2.5, 3.5, length.out = 5),
is_positive_definite = TRUE)))
d %>% tfd_covariance()

---

tfd_cross_entropy

**Computes the (Shannon) cross entropy.**

**Description**

Denote this distribution (self) by P and the other distribution by Q. Assuming P, Q are absolutely continuous with respect to one another and permit densities \( p(x) \, dr(x) \) and \( q(x) \, dr(x) \), (Shannon) cross entropy is defined as:

\[
H[P, Q] = E_p[-\log q(X)] = -\int_F p(x) \log q(x) \, dr(x)
\]

where \( F \) denotes the support of the random variable \( X \sim P \).

**Usage**

tfd_cross_entropy(distribution, other, name = "cross_entropy")

**Arguments**

distribution  The distribution being used.
other tfp$distributions$Distribution instance.
name String prepended to names of ops created by this function.

**Value**

cross_entropy: self.dtype Tensor with shape \([B_1, ..., B_n]\) representing n different calculations of (Shannon) cross entropy.

**See Also**

Other distribution_methods: tfd_cdf(), tfd_covariance(), tfd_entropy(), tfd_kl_divergence(), tfd_log_cdf(), tfd_log_prob(), tfd_log_survival_function(), tfd_mean(), tfd_mode(), tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_survival_function(), tfd_variance()

**Examples**

d1 <- tfd_normal(loc = 1, scale = 1)
d2 <- tfd_normal(loc = 2, scale = 1)
d1 %>% tfd_cross_entropy(d2)
The scalar Deterministic distribution is parameterized by a (batch) point $\text{loc}$ on the real line. The distribution is supported at this point only, and corresponds to a random variable that is constant, equal to $\text{loc}$. See Degenerate rv.

Usage

```r
  tfd_deterministic(
    loc,
    atol = NULL,
    rtol = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Deterministic"
  )
```

Arguments

- **loc**: Numeric Tensor of shape $[B_1, ..., B_b]$, with $b >= 0$. The point (or batch of points) on which this distribution is supported.
- **atol**: Non-negative Tensor of same dtype as `loc` and broadcastable shape. The absolute tolerance for comparing closeness to `loc`. Default is 0.
- **rtol**: Non-negative Tensor of same dtype as `loc` and broadcastable shape. The relative tolerance for comparing closeness to `loc`. Default is 0.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

Mathematical Details

The probability mass function (pmf) and cumulative distribution function (cdf) are

\[
  \text{pmf}(x; \text{loc}) = 1, \text{ if } x == \text{loc}, \text{ else } 0
\]
\[
  \text{cdf}(x; \text{loc}) = 1, \text{ if } x >= \text{loc}, \text{ else } 0
\]
tfd_dirichlet

Dirichlet distribution

Description

The Dirichlet distribution is defined over the \((k-1)\)-simplex using a positive, length-k vector concentration \((k > 1)\). The Dirichlet is identically the Beta distribution when \(k = 2\).

Usage

```python
tfd_dirichlet(
    concentration,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Dirichlet"
)
```
Arguments

- **concentration**: Positive floating-point Tensor indicating mean number of class occurrences; aka "alpha". Implies self$dtype, and self$batch_shape, self$event_shape, i.e., if concentration$shape = [N1, N2, ..., Nm, k] then batch_shape = [N1, N2, ..., Nm] and event_shape = [k].

- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**: name prefixed to Ops created by this class.

Details

Mathematical Details

The Dirichlet is a distribution over the open (k-1)-simplex, i.e.,

\[ S^{(k-1)} = \{ (x_0, \ldots, x_{(k-1)}) \in \mathbb{R}^k : \sum_j x_j = 1 \text{ and } \forall_j x_j > 0 \} \]

The probability density function (pdf) is,

\[
\text{pdf}(x; \alpha) = \frac{\prod_j x_j^{\alpha_j - 1}}{Z} \\
Z = \frac{\prod_j \Gamma(\alpha_j)}{\Gamma(\sum_j \alpha_j)}
\]

where:

- \( x \) in \( S^{(k-1)} \), i.e., the (k-1)-simplex,
- \( \text{concentration} = \alpha = [\alpha_0, \ldots, \alpha_{(k-1)}] \), \( \alpha_j > 0 \),
- \( Z \) is the normalization constant aka the multivariate beta function, and,
- \( \Gamma \) is the gamma function.

The concentration represents mean total counts of class occurrence, i.e.,

\[
\text{concentration} = \alpha = \text{mean} \times \text{total_concentration}
\]

where mean in \( S^{(k-1)} \) and total_concentration is a positive real number representing a mean total count. Distribution parameters are automatically broadcast in all functions; see examples for details. Warning: Some components of the samples can be zero due to finite precision. This happens more often when some of the concentrations are very small. Make sure to round the samples to np$finfo(dtype)$tiny before computing the density. Samples of this distribution are reparameterized (pathwise differentiable). The derivatives are computed using the approach described in the paper Michael Figurnov, Shakir Mohamed, Andriy Mnih. Implicit Reparameterization Gradients, 2018

Value

a distribution instance.
**tfd_dirichlet_multinomial**

**Dirichlet-Multinomial compound distribution**

**Description**

The Dirichlet-Multinomial distribution is parameterized by a (batch of) length-K concentration vectors (K > 1) and a total_count number of trials, i.e., the number of trials per draw from the DirichletMultinomial. It is defined over a (batch of) length-K vector counts such that \( \text{tf}\text{reduce}_\text{sum}(\text{counts}, -1) = \text{total}_\text{count} \). The Dirichlet-Multinomial is identically the Beta-Binomial distribution when K = 2.

**Usage**

```python
tfd_dirichlet_multinomial(
    total_count,
    concentration,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "DirichletMultinomial"
)
```

**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`. Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_chisq()`, `tfd_chisq2()`, `tfd_cholesky_lkj()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exponential()`, `tfd_gaussian()`, `tfd_gaussian_process()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process_state_space_model()`, `tfd_gaussiandm()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`, `tfd_inverse_gaussian()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`, `tfd_multivariate_normal()`, `tfd_multivariate_normal_linear_operator()`, `tfd_multivariate_normal_tri_l()`, `tfd_multivariate_student_t()`, `tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature_compound()`, `tfd_poisson()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`, `tfd_triangular()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_variational_gaussian_process()`, `tfd_vector_diffeomixture()`, `tfd_vector_exponential_diag()`, `tfd_vector_exponential_linear_operator()`, `tfd_vector_laplace_diag()`,
**Arguments**

**total_count**
Non-negative floating point tensor, whose dtype is the same as `concentration`. The shape is broadcastable to `[N1,...,Nm]` with `m >= 0`. Defines this as a batch of `N1 x ... x Nm` different Dirichlet multinomial distributions. Its components should be equal to integer values.

**concentration**
Positive floating point tensor, whose dtype is the same as `n` with shape broadcastable to `[N1,...,Nm,K]` for `m >= 0`. Defines this as a batch of `N1 x ... x Nm` different `K` class Dirichlet multinomial distributions.

**validate_args**
Logical, default `FALSE`. When `TRUE` distribution parameters are checked for validity despite possibly degrading runtime performance. When `FALSE` invalid inputs may silently render incorrect outputs. Default value: `FALSE`.

**allow_nan_stats**
Logical, default `TRUE`. When `TRUE`, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When `FALSE`, an exception is raised if one or more of the statistic’s batch members are undefined.

**name**
name prefixed to Ops created by this class.

**Details**

**Mathematical Details**

The Dirichlet-Multinomial is a distribution over `K`-class counts, i.e., a length-`K` vector of non-negative integer counts `n = [n_0, ..., n_{K-1}]`.

The probability mass function (pmf) is,

```
  pmf(n; alpha, N) = Beta(alpha + n) / (prod_j n_j!) / Z
  Z = Beta(alpha) / N!
```

where:
- `concentration = alpha = [alpha_0, ..., alpha_{K-1}]`, `alpha_j > 0`,
- `total_count = N`, `N` a positive integer,
- `N!` is `N` factorial, and,
- `Beta(x) = prod_j Gamma(x_j) / Gamma(sum_j x_j)` is the multivariate beta function, and,
- `Gamma` is the gamma function.

Dirichlet-Multinomial is a [compound distribution](https://en.wikipedia.org/wiki/Compound_distribution), i.e., its samples are generated as follows.

1. Choose class probabilities: `probs = [p_0,...,p_{(K-1)}] ~ Dir(concentration)`
2. Draw integers: `counts = [n_0,...,n_{(K-1)}] ~ Multinomial(total_count, probs)`

The last concentration dimension parametrizes a single Dirichlet-Multinomial distribution. When calling distribution functions (e.g., `dist$prob(counts)`), `concentration`, `total_count` and `counts` are broadcast to the same shape. The last dimension of `counts` corresponds single Dirichlet-Multinomial distributions. Distribution parameters are automatically broadcast in all functions; see examples for details.

**Pitfalls**
The number of classes, `K`, must not exceed:
• the largest integer representable by self$\text{dtype}$, i.e., $2^{\text{mantissa_bits}+1}$ (IEEE754),
• the maximum Tensor index, i.e., $2^{31}-1$.

Note: This condition is validated only when validate_args = TRUE.

Value

A distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chisq()`, `tfd_chisq_lkj()`, `tfd_deterministic()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exponential()`, `tfd_bernoulli()`, `tfd_beta()`,
`tfd_cauchy()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chi()`,
tfd_doublesided_maxwell()  
Double-sided Maxwell distribution.

tfd_doublesided_maxwell

Description

This distribution is useful to compute measure valued derivatives for Gaussian distributions. See Mohamed et al. (2019) for more details.

Usage

tfd_doublesided_maxwell()
  loc,
  scale,
  validate_args = FALSE,
allow_nan_stats = TRUE,
name = "doublesided_maxwell"
)

Arguments

loc  Floating point tensor; location of the distribution
scale  Floating point tensor; the scales of the distribution. Must contain only positive values.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name  string prefixed to Ops created by this class. Default value: 'doublesided_maxwell'.

Details

Mathematical details
The double-sided Maxwell distribution generalizes the Maxwell distribution to the entire real line.

\[
pdf(x; \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} \left(\frac{x-\mu}{\sigma}\right)^2 \exp\left(-0.5 \left(\frac{x-\mu}{\sigma}\right)^2\right)
\]

where \( \text{loc} = \mu \) and \( \text{scale} = \sigma \). The DoublesidedMaxwell distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{DoublesidedMaxwell}(\text{loc}=0, \text{scale}=1)
Y = \text{loc} + \text{scale} \times X
\]

The double-sided Maxwell is a symmetric distribution that extends the one-sided maxwell from \( \mathbb{R}^+ \) to the entire real line. Their densities are therefore the same up to a factor of 0.5.

It has several methods for generating random variates from it. The version here uses 3 Gaussian variates and a uniform variate to generate the samples. The sampling path is:

\[
\mu + \sigma \times \text{sgn}(U-0.5) \times \sqrt{X^2 + Y^2 + Z^2}, U \sim \text{Unif}, X,Y,Z \sim \text{N}(0,1)
\]

In the sampling process above, the random variates generated by \( \sqrt{X^2 + Y^2 + Z^2} \) are samples from the one-sided Maxwell (or Maxwell-Boltzmann) distribution.

Value

a distribution instance.

References

- Mohamed, et all, "Monte Carlo Gradient Estimation in Machine Learning.", 2019
See Also

For usage examples see e.g. `tfd_sample().tfd_log_prob().tfd_mean()`.

---

tfd_empirical  
*Empirical distribution*

### Description

The Empirical distribution is parameterized by a (batch) multiset of samples. It describes the empirical measure (observations) of a variable. Note: some methods (log_prob, prob, cdf, mode, entropy) are not differentiable with regard to samples.

### Usage

```python
tfd_empirical(
    samples,
    event_ndims = 0,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Empirical"
)
```

### Arguments

- **samples**  
  Numeric Tensor of shape \([B1, ..., Bk, S, E1, ..., En]\), \(k, n \geq 0\). Samples or batches of samples on which the distribution is based. The first \(k\) dimensions index into a batch of independent distributions. Length of \(S\) dimension determines number of samples in each multiset. The last \(n\) dimension represents samples for each distribution. \(n\) is specified by argument `event_ndims`.

- **event_ndims**  
  int32, default 0. number of dimensions for each event. When 0 this distribution has scalar samples. When 1 this distribution has vector-like samples.

- **validate_args**  
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**  
  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**  
  name prefixed to Ops created by this class.

### Details

**Mathematical Details**

The probability mass function (pmf) and cumulative distribution function (cdf) are
\[
\text{pmf}(k; s_1, \ldots, s_n) = \sum_i I(k)^{k == s_i} / n
\]
\[
I(k)^{k == s_i} = 1, \text{ if } k == s_i, \text{ else } 0.
\]
\[
\text{cdf}(k; s_1, \ldots, s_n) = \sum_i I(k)^{k >= s_i} / n
\]
\[
I(k)^{k >= s_i} = 1, \text{ if } k >= s_i, \text{ else } 0.
\]

**Value**

A distribution instance.

**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


---

**tfd_entropy**

Shannon entropy in nats.

**Description**

Shannon entropy in nats.

**Usage**

`tfd_entropy(distribution, ...)`

**Arguments**

distribution The distribution being used.

... Additional parameters passed to Python.
Value

a Tensor of shape `sample_shape(x) + self$batch_shape` with values of type `self$dtype`.

See Also

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_mode()`, `tfd_prob()`, `tfd_quantile()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`

Examples

d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_entropy()

Description

The Exponential distribution is parameterized by an event `rate` parameter.

Usage

tfd_exponential(
  rate,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Exponential"
)

Arguments

rate Floating point tensor, equivalent to 1 / mean. Must contain only positive values.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name name prefixed to Ops created by this class.
Details

Mathematical Details

The probability density function (pdf) is,

$$\text{pdf}(x; \lambda, x > 0) = \exp(-\lambda x) / Z$$

$$Z = 1 / \lambda$$

where rate = lambda and Z is the normalizing constant.

The Exponential distribution is a special case of the Gamma distribution, i.e.,

$$\text{Exponential}(rate) = \text{Gamma}(\text{concentration}=1., \text{rate})$$

The Exponential distribution uses a rate parameter, or "inverse scale", which can be intuited as,

$$X \sim \text{Exponential}(\text{rate}=1)$$

$$Y = X / \text{rate}$$

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()` , `tfd_log_prob()` , `tfd_mean()`.

`tfd_exp_relaxed_one_hot_categorical`  
`ExpRelaxedOneHotCategorical distribution with temperature and logits.`

**Description**

`ExpRelaxedOneHotCategorical` distribution with temperature and logits.

**Usage**

```r
library(tfd)
tfd_exp_relaxed_one_hot_categorical(
  temperature,
  logits = NULL,
  probs = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "ExpRelaxedOneHotCategorical"
)
```

**Arguments**

- **temperature**
  An 0-D Tensor, representing the temperature of a set of `ExpRelaxedCategorical` distributions. The temperature should be positive.

- **logits**
  An N-D Tensor, N >= 1, representing the log probabilities of a set of `ExpRelaxedCategorical` distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of logits for each class. Only one of logits or probs should be passed in.

- **probs**
  An N-D Tensor, N >= 1, representing the probabilities of a set of `ExpRelaxedCategorical` distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of probabilities for each class. Only one of logits or probs should be passed in.

- **validate_args**
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**
  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**
  name prefixed to Ops created by this class.

**Value**

A distribution instance.
The finite discrete distribution.

**Description**

The FiniteDiscrete distribution is parameterized by either probabilities or log-probabilities of a set of \( K \) possible outcomes, which is defined by a strictly ascending list of \( K \) values.

**Usage**

```r
tfd_finite_discrete(
  outcomes,
  logits = NULL,
  probs = NULL,
  rtol = NULL,
  atol = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "FiniteDiscrete"
)
```

**Arguments**

- **outcomes** A 1-D floating or integer Tensor, representing a list of possible outcomes in strictly ascending order.
- **logits** A floating N-D Tensor, \( N \geq 1 \), representing the log probabilities of a set of FiniteDiscrete distributions. The first \( N - 1 \) dimensions index into a batch of independent distributions and the last dimension represents a vector of logits for each discrete value. Only one of logits or probs should be passed in.
- **probs** A floating N-D Tensor, \( N \geq 1 \), representing the probabilities of a set of FiniteDiscrete distributions. The first \( N - 1 \) dimensions index into a batch of independent distributions and the last dimension represents a vector of probabilities for each discrete value. Only one of logits or probs should be passed in.
- **rtol** Tensor with same dtype as outcomes. The relative tolerance for floating number comparison. Only effective when outcomes is a floating Tensor. Default is \( 10 \times \text{eps} \).
- **atol** Tensor with same dtype as outcomes. The absolute tolerance for floating number comparison. Only effective when outcomes is a floating Tensor. Default is \( 10 \times \text{eps} \).
- **validate_args** Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

name
string prefixed to Ops created by this class.

Details
Note: log_prob, prob, cdf, mode, and entropy are differentiable with respect to logits or probs but not with respect to outcomes.
Mathematical Details
The probability mass function (pmf) is,
\[ \text{pmf}(x; \pi, q_i) = \prod_j \pi_j^{[x == q_i_j]} \]

Value
a distribution instance.

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().
validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid
inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[
pdf(x; \alpha, \beta, x > 0) = x^{(\alpha - 1)} \exp(-x \beta) / Z
\]
\[
Z = \text{Gamma}(\alpha) \beta^{-(\alpha + 1)}
\]

where

- \(\text{concentration} = \alpha, \alpha > 0\),
- \(\text{rate} = \beta, \beta > 0\),
- \(Z\) is the normalizing constant, and,
- \(\text{Gamma}\) is the gamma function.

The cumulative density function (cdf) is,

\[
cdf(x; \alpha, \beta, x > 0) = \text{GammaInc}(\alpha, \beta x) / \text{Gamma}(\alpha)
\]

where \(\text{GammaInc}\) is the lower incomplete Gamma function. The parameters can be intuited via their
relationship to mean and stddev,

\[
\text{concentration} = \alpha = (\text{mean} / \text{stddev})^2
\]
\[
\text{rate} = \beta = \text{mean} / \text{stddev}^2 = \text{concentration} / \text{mean}
\]

Distribution parameters are automatically broadcast in all functions; see examples for details.

Warning: The samples of this distribution are always non-negative. However, the samples that are
smaller than np\$\text{finfo(dtype)}\$\$\tiny are rounded to this value, so it appears more often than it
should. This should only be noticeable when the concentration is very small, or the rate is very
large. See note in tf\$\text{random\_gamma}\$ docstring. Samples of this distribution are reparameterized
(pathwise differentiable). The derivatives are computed using the approach described in the paper
Michael Figurnov, Shakir Mohamed, Andriy Mnih. Implicit Reparameterization Gradients, 2018

Value

a distribution instance.
Gamma-Gamma distribution

Description

Gamma-Gamma is a compound distribution defined over positive real numbers using parameters concentration, mixing_concentration and mixing_rate.

Usage

```r
tfd_gamma_gamma(
  concentration,
  mixing_concentration,
  mixing_rate,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "GammaGamma"
)
```

Arguments

- **concentration**: Floating point tensor, the concentration params of the distribution(s). Must contain only positive values.
mixing_concentration: Floating point tensor, the concentration params of the mixing Gamma distribution(s). Must contain only positive values.

mixing_rate: Floating point tensor, the rate params of the mixing Gamma distribution(s). Must contain only positive values.

validate_args: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name: name prefixed to Ops created by this class.

Details

This distribution is also referred to as the beta of the second kind (B2), and can be useful for transaction value modeling, as in Fader and Hardi, 2013.

Mathematical Details

It is derived from the following Gamma-Gamma hierarchical model by integrating out the random variable beta.

\[ \begin{align*}
\beta & \sim \text{Gamma}(\alpha_0, \beta_0) \\
X | \beta & \sim \text{Gamma}(\alpha, \beta)
\end{align*} \]

where

- concentration = \(\alpha\)
- mixing_concentration = \(\alpha_0\)
- mixing_rate = \(\beta_0\)

The probability density function (pdf) is

\[ x^{(\alpha - 1)} \]

\[ \text{pdf}(x; \alpha, \alpha_0, \beta_0) = Z \times (x + \beta_0)^{(\alpha + \alpha_0)} \]

where the normalizing constant \(Z = \text{Beta}(\alpha, \alpha_0) \times \beta_0^{\alpha} (\beta_0)^{\alpha} \). Samples of this distribution are reparameterized as samples of the Gamma distribution are reparameterized using the technique described in (Figurnov et al., 2018).

@section References:

tfd_gaussian_process

Marginal distribution of a Gaussian process at finitely many points.

Description

A Gaussian process (GP) is an indexed collection of random variables, any finite collection of which are jointly Gaussian. While this definition applies to finite index sets, it is typically implicit that the index set is infinite; in applications, it is often some finite dimensional real or complex vector space. In such cases, the GP may be thought of as a distribution over (real- or complex-valued) functions defined over the index set.

Usage

```python
tfd_gaussian_process(
    kernel,
    index_points,
    mean_fn = NULL,
    observation_noise_variance = 0,
    jitter = 1e-06,
)```
Arguments

kernel PositiveSemidefiniteKernel-like instance representing the GP's covariance function.

index_points float Tensor representing finite (batch of) vector(s) of points in the index set over which the GP is defined. Shape has the form \([b1, ..., bB, e1, f1, ..., fF]\) where \(F\) is the number of feature dimensions and must equal \(\text{kernel.feature_ndims}\) and \(e1\) is the number (size) of index points in each batch (we denote it \(e1\) to distinguish it from the number of inducing index points, denoted \(e2\) below). Ultimately the GaussianProcess distribution corresponds to an \(e1\)-dimensional multivariate normal. The batch shape must be broadcastable with \(\text{kernel.batch_shape}\), the batch shape of \(\text{inducing_index_points}\), and any batch dims yielded by \(\text{mean_fn}\).

mean_fn function that acts on index points to produce a (batch of) vector(s) of mean values at those index points. Takes a Tensor of shape \([b1, ..., bB, f1, ..., fF]\) and returns a Tensor whose shape is (broadcastable with) \([b1, ..., bB]\). Default value: NULL implies constant zero function.

observation_noise_variance float Tensor representing the variance of the noise in the Normal likelihood distribution of the model. May be batched, in which case the batch shape must be broadcastable with the shapes of all other batched parameters (\(\text{kernel.batch_shape}\), \(\text{index_points}\), etc.). Default value: 0.

jitter float scalar Tensor added to the diagonal of the covariance matrix to ensure positive definiteness of the covariance matrix. Default value: 1e-6.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

Just as Gaussian distributions are fully specified by their first and second moments, a Gaussian process can be completely specified by a mean and covariance function. Let \(S\) denote the index set and \(K\) the space in which each indexed random variable takes its values (again, often \(\mathbb{R}\) or \(\mathbb{C}\)). The mean function is then a map \(m: S \rightarrow K\), and the covariance function, or kernel, is a positive-definite function \(k: (S \times S) \rightarrow K\). The properties of functions drawn from a GP are entirely dictated (up to translation) by the form of the kernel function.
This Distribution represents the marginal joint distribution over function values at a given finite collection of points \([x[1], \ldots, x[N]]\) from the index set \(S\). By definition, this marginal distribution is just a multivariate normal distribution, whose mean is given by the vector \([m(x[1]), \ldots, m(x[N])]\) and whose covariance matrix is constructed from pairwise applications of the kernel function to the given inputs:

\[
\begin{bmatrix}
k(x[1], x[1]) & k(x[1], x[2]) & \ldots & k(x[1], x[N]) \\
k(x[2], x[1]) & k(x[2], x[2]) & \ldots & k(x[2], x[N]) \\
\vdots & \vdots & \ddots & \vdots \\
k(x[N], x[1]) & k(x[N], x[2]) & \ldots & k(x[N], x[N])
\end{bmatrix}
\]

For this to be a valid covariance matrix, it must be symmetric and positive definite; hence the requirement that \(k\) be a positive definite function (which, by definition, says that the above procedure will yield PD matrices).

We also support the inclusion of zero-mean Gaussian noise in the model, via the \(\text{observation\_noise\_variance}\) parameter. This augments the generative model to

\[
f \sim \text{GP}(m, k)
\]

\((y[i] \mid f, x[i]) \sim \text{Normal}(f(x[i]), s)\)

where

- \(m\) is the mean function
- \(k\) is the covariance kernel function
- \(f\) is the function drawn from the GP
- \(x[i]\) are the index points at which the function is observed
- \(y[i]\) are the observed values at the index points
- \(s\) is the scale of the observation noise.

Note that this class represents an unconditional Gaussian process; it does not implement posterior inference conditional on observed function evaluations. This class is useful, for example, if one wishes to combine a GP prior with a non-conjugate likelihood using MCMC to sample from the posterior.

Mathematical Details

The probability density function (pdf) is a multivariate normal whose parameters are derived from the GP’s properties:

\[
\text{pdf}(x; \text{index\_points}, \text{mean\_fn}, \text{kernel}) = \exp(-0.5 \cdot y) / Z
\]

\[K = (\text{kernel.matrix}(\text{index\_points}, \text{index\_points}) +
(\text{observation\_noise\_variance} + \text{jitter}) \cdot \text{eye}(N))\]

\[y = (x - \text{mean\_fn}(\text{index\_points}))^T @ K @ (x - \text{mean\_fn}(\text{index\_points}))\]

\[Z = (2 \cdot \pi)^{(0.5 \cdot N)} \cdot |\text{det}(K)|^{0.5}\]

where:

- \(\text{index\_points}\) are points in the index set over which the GP is defined,
- \(\text{mean\_fn}\) is a callable mapping the index set to the GP’s mean values,
- kernel is PositiveSemidefiniteKernel-like and represents the covariance function of the GP.
- observation_noise_variance represents (optional) observation noise.
- jitter is added to the diagonal to ensure positive definiteness up to machine precision (otherwise Cholesky-decomposition is prone to failure).
- eye(N) is an N-by-N identity matrix.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batchreshape(), tfd_bernoulli(), tfd_beta(), tfd_beta_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exponential(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_joint_distribution(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space(), tfd_lkj(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_student_t(), tfd_student_t_process(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_normal(), tfd_uniform(), tfd_von_mises_fisher(), tfd_von_mises(), tfd_wishart_linear_operator(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_wishart(), tfd_zipf()
Usage

tfd_gaussian_process_regression_model(
    kernel,
    index_points = NULL,
    observation_index_points = NULL,
    observations = NULL,
    observation_noise_variance = 0,
    predictive_noise_variance = NULL,
    mean_fn = NULL,
    jitter = 1e-06,
    validate_args = FALSE,
    allow_nan_stats = FALSE,
    name = "GaussianProcessRegressionModel"
)

Arguments

kernel PositiveSemidefiniteKernel-like instance representing the GP’s covariance function.

index_points float Tensor representing finite (batch of) vector(s) of points in the index set over which the GP is defined. Shape has the form \([b_1, ..., b_B, e_1, f_1, ..., f_F]\) where \(F\) is the number of feature dimensions and must equal \(\text{kernel.feature_ndims}\) and \(e_1\) is the number (size) of index points in each batch (we denote it \(e_1\) to distinguish it from the number of inducing index points, denoted \(e_2\) below). Ultimately the GaussianProcess distribution corresponds to an \(e_1\)-dimensional multivariate normal. The batch shape must be broadcastable with \(\text{kernel.batch_shape}\), the batch shape of \(\text{inducing_index_points}\), and any batch dims yielded by \(\text{mean_fn}\).

observation_index_points Tensor representing finite collection, or batch of collections, of points in the index set for which some data has been observed. Shape has the form \([b_1, ..., b_B, e, f_1, ..., f_F]\) where \(F\) is the number of feature dimensions and must equal \(\text{kernel.feature_ndims}\), and \(e\) is the number (size) of index points in each batch. \([b_1, ..., b_B, e]\) must be broadcastable with the shape of observations, and \([b_1, ..., b_B]\) must be broadcastable with the shapes of all other batched parameters (\(\text{kernel.batch_shape}\), \(\text{index_points}\), etc). The default value is None, which corresponds to the empty set of observations, and simply results in the prior predictive model (a GP with noise of variance \(\text{predictive_noise_variance}\)).

observations Tensor representing collection, or batch of collections, of observations corresponding to \(\text{observation_index_points}\). Shape has the form \([b_1, ..., b_B, e]\), which must be broadcastable with the batch and example shapes of \(\text{observation_index_points}\). The batch shape \([b_1, ..., b_B]\) must be broadcastable with the shapes of all other batched parameters (\(\text{kernel.batch_shape}\), \(\text{index_points}\), etc.). The default value is None, which corresponds to the empty set of observations, and simply results in the prior predictive model (a GP with noise of variance \(\text{predictive_noise_variance}\)).

observation_noise_variance float Tensor representing the variance of the noise in the Normal likelihood
distribution of the model. May be batched, in which case the batch shape must be broadcastable with the shapes of all other batched parameters (kernel$batch_shape, index_points, etc.). Default value: 0.

**predictive_noise_variance**

Tensor representing the variance in the posterior predictive model. If None, we simply re-use observation_noise_variance for the posterior predictive noise. If set explicitly, however, we use this value. This allows us, for example, to omit predictive noise variance (by setting this to zero) to obtain noiseless posterior predictions of function values, conditioned on noisy observations.

**mean_fn**

Callable that acts on index_points to produce a collection, or batch of collections, of mean values at index_points. Takes a Tensor of shape $[b1, ..., bB, f1, ..., fF]$ and returns a Tensor whose shape is broadcastable with $[b1, ..., bB]$. Default value: None implies the constant zero function.

**jitter**

Float scalar Tensor added to the diagonal of the covariance matrix to ensure positive definiteness of the covariance matrix. Default value: $1e-6$.

**validate_args**

Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

**allow_nan_stats**

Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

**name**

Name prefixed to Ops created by this class.

Value

A distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_joint_distribution(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tril(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_poisson(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_student_t(), tfd_student_t_process(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_process(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag().
The Generalized Pareto distribution.

Description

The Generalized Pareto distributions are a family of continuous distributions on the reals. Special cases include Exponential (when \( \text{loc} = 0 \), \( \text{concentration} = 0 \)), Pareto (when \( \text{concentration} > 0 \), \( \text{loc} = \text{scale} / \text{concentration} \)), and Uniform (when \( \text{concentration} = -1 \)).

Usage

tfd_generalized_pareto(
  loc,
  scale,
  concentration,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = NULL
)

Arguments

loc The location / shift of the distribution. GeneralizedPareto is a location-scale distribution. This parameter lower bounds the distribution’s support. Must broadcast with scale, concentration. Floating point Tensor.

scale The scale of the distribution. GeneralizedPareto is a location-scale distribution, so doubling the scale doubles a sample and halves the density. Strictly positive floating point Tensor. Must broadcast with loc, concentration.

concentration The shape parameter of the distribution. The larger the magnitude, the more the distribution concentrates near \( \text{loc} \) (for \( \text{concentration} \geq 0 \)) or near \( \text{loc} - (\text{scale}/\text{concentration}) \) (for \( \text{concentration} < 0 \)). Floating point Tensor.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.
Details

This distribution is often used to model the tails of other distributions. As a member of the location-scale family, $X \sim \text{GeneralizedPareto}(\text{loc}=\text{loc}, \text{scale}=\text{scale}, \text{concentration}=\text{conc})$ maps to $Y \sim \text{GeneralizedPareto}(\text{loc}=0, \text{scale}=1, \text{concentration}=\text{conc})$ via $Y = (X - \text{loc}) / \text{scale}$.

For positive concentrations, the distribution is equivalent to a hierarchical Exponential-Gamma model with $X | \text{rate} \sim \text{Exponential}(\text{rate})$ and $\text{rate} \sim \text{Gamma}(\text{concentration}=1 / \text{concentration}, \text{scale}=\text{scale} / \text{concentration})$. In the following, samps1 and samps2 are identically distributed:

```r
  genp <- tfd_generalized_pareto(loc = 0, scale = scale, concentration = conc)
  samps1 <- genp %>% tfd_sample(1000)
  jd <- tfd_joint_distribution_named(
    list(
      rate = tfd_gamma(1 / genp$concentration, genp$scale / genp$concentration),
      x = function(rate) tfd_exponential(rate)))
  samps2 <- jd %>% tfd_sample(1000) %>% .$x
```

The support of the distribution is always lower bounded by $\text{loc}$. When $\text{concentration} < 0$, the support is also upper bounded by $\text{loc} + \text{scale} / \text{abs(\text{concentration})}$.

Mathematical Details

The probability density function (pdf) is,

$$\text{pdf}(x; \mu, \sigma, \text{shp}, x > \mu) = \frac{(1 + \text{shp} \times (x - \mu) / \sigma)^{-1 / \text{shp} - 1}}{\sigma}$$

where:

- $\text{concentration} = \text{shp}$, any real value.
- $\text{scale} = \sigma$, $\sigma > 0$.
- $\text{loc} = \mu$.

The cumulative density function (cdf) is,

$$\text{cdf}(x; \mu, \sigma, \text{shp}, x > \mu) = 1 - (1 + \text{shp} \times (x - \mu) / \sigma)^{-1 / \text{shp}}$$

Distribution parameters are automatically broadcast in all functions; see examples for details. Samples of this distribution are reparameterized (pathwise differentiable).

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.
tfd_geometric

Geometric distribution

Description

The Geometric distribution is parameterized by p, the probability of a positive event. It represents
the probability that in k + 1 Bernoulli trials, the first k trials failed, before seeing a success. The
pmf of this distribution is:

Usage

```r
  tfd_geometric(
    logits = NULL,
    probs = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Geometric"
  )
```

Arguments

- **logits**: Floating-point Tensor with shape \([B_1, ..., B_b]\) where \(b \geq 0\) indicates the num-
  ber of batch dimensions. Each entry represents logits for the probability of suc-
  cess for independent Geometric distributions and must be in the range (-inf, inf].
  Only one of `logits` or `probs` should be specified.
- **probs**: Positive floating-point Tensor with shape \([B_1, ..., B_b]\) where \(b \geq 0\) indicates the
  number of batch dimensions. Each entry represents the probability of success
  for independent Geometric distributions and must be in the range (0, 1]. Only
  one of `logits` or `probs` should be specified.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for
  validity despite possibly degrading runtime performance. When FALSE invalid
  inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
  the value NaN to indicate the result is undefined. When FALSE, an exception is
  raised if one or more of the statistic's batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

Mathematical Details

\[
  \text{pmf}(k; p) = (1 - p)^{k} \times p
\]

where:

- p is the success probability, 0 < p <= 1, and,
- k is a non-negative integer.
Scalar Gumbel distribution with location \( \text{loc} \) and scale parameters.

### Description

Mathematical details

### Usage

```r
tfd_gumbel(
  loc,
  scale,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Gumbel"
)
```
Arguments

loc
Floating point tensor, the means of the distribution(s).

scale
Floating point tensor, the scales of the distribution(s). ‘scale’ must contain only positive values.

validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details

The probability density function (pdf) of this distribution is,

\[ \text{pdf}(x; \mu, \sigma) = \exp(-((x - \mu) / \sigma - \exp(-(x - \mu) / \sigma))) / \sigma \]

where \( \text{loc} = \mu \) and \( \text{scale} = \sigma \).

The cumulative density function of this distribution is, \( \text{cdf}(x; \mu, \sigma) = \exp(-\exp(-(x - \mu) / \sigma)) \)

The Gumbel distribution is a member of the location-scale family, i.e., it can be constructed as,

\[ X \sim \text{Gumbel}(\text{loc}=0, \text{scale}=1) \]
\[ Y = \text{loc} + \text{scale} \times X \]

Value

a distribution instance.

See Also

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chisq(), tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_geometric(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_poisson(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_normal(), tfd_uniform(),
tfd_half_cauchy

Half-Cauchy distribution

Description
The half-Cauchy distribution is parameterized by a \( \text{loc} \) and a \( \text{scale} \) parameter. It represents the right half of the two symmetric halves in a Cauchy distribution.

Usage
```r
tfd_half_cauchy(
  loc,
  scale,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "HalfCauchy"
)
```

Arguments
- \( \text{loc} \): Floating-point Tensor; the location(s) of the distribution(s).
- \( \text{scale} \): Floating-point Tensor; the scale(s) of the distribution(s). Must contain only positive values.
- \( \text{validate_args} \): Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- \( \text{allow_nan_stats} \): Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.
- \( \text{name} \): name prefixed to Ops created by this class.

Details
Mathematical Details
The probability density function (pdf) for the half-Cauchy distribution is given by

\[
\text{pdf}(x; \text{loc}, \text{scale}) = \frac{2}{\pi \text{scale} (1 + z^2)}
\]
\[
z = \frac{(x - \text{loc})}{\text{scale}}
\]

where \( \text{loc} \) is a scalar in \( \mathbb{R} \) and \( \text{scale} \) is a positive scalar in \( \mathbb{R} \). The support of the distribution is given by the interval \([\text{loc}, \infty)\).
Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chisq()`, `tfd_cholesky_lkj()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gaussian()`, `tfd_inv_gamma()`, `tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`, `tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`, `tfd_multivariate_normal_tri_l()`, `tfd_multivariate_student_t_linear_operator()`, `tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature_tower()`, `tfd_poisson()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relayed_bernoulli()`, `tfd_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`, `tfd_student_t_process()`, `tfd_student_t()`.

Usage

```python
import tensorflow_probability as tfd

tfd_half_normal(scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "HalfNormal"
)
```

Arguments

- `scale` : Floating point tensor; the scales of the distribution(s). Must contain only positive values.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name  name prefixed to Ops created by this class.

Details
The half normal is a transformation of a centered normal distribution. If some random variable $X$ has normal distribution,

$$X \sim \text{Normal}(0,0, \text{scale})$$

$$Y = |X|$$

Then $Y$ will have half normal distribution. The probability density function (pdf) is:

$$\text{pdf}(x; \text{scale}, x > 0) = \sqrt{2} / (\text{scale} \times \sqrt{\pi}) \times \exp(-1/2 \times (x / \text{scale})^2))$$

Where $\text{scale} = \sigma$ is the standard deviation of the underlying normal distribution.

Value
a distribution instance.

See Also
For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

tfd_hidden_markov_model

Hidden Markov model distribution

Description

The HiddenMarkovModel distribution implements a (batch of) hidden Markov models where the initial states, transition probabilities and observed states are all given by user-provided distributions.

Usage

tfd_hidden_markov_model(
    initial_distribution,
    transition_distribution,
    observation_distribution,
    num_steps,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "HiddenMarkovModel"
)

Arguments

initial_distribution
A Categorical-like instance. Determines probability of first hidden state in Markov chain. The number of categories must match the number of categories of transition_distribution as well as both the rightmost batch dimension of transition_distribution and the rightmost batch dimension of observation_distribution.

transition_distribution
A Categorical-like instance. The rightmost batch dimension indexes the probability distribution of each hidden state conditioned on the previous hidden state.

observation_distribution
A tfp$distributions$Distribution-like instance. The rightmost batch dimension indexes the distribution of each observation conditioned on the corresponding hidden state.

num_steps
The number of steps taken in Markov chain. An integer.

validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.
Details

This model assumes that the transition matrices are fixed over time. In this model, there is a sequence of integer-valued hidden states: \(z[0], z[1], \ldots, z[num\_steps - 1]\) and a sequence of observed states: \(x[0], \ldots, x[num\_steps - 1]\).

The distribution of \(z[0]\) is given by initial_distribution. The conditional probability of \(z[i + 1]\) given \(z[i]\) is described by the batch of distributions in transition_distribution. For a batch of hidden Markov models, the coordinates before the rightmost one of the transition_distribution batch correspond to indices into the hidden Markov model batch. The rightmost coordinate of the batch is used to select which distribution \(z[i + 1]\) is drawn from. The distributions corresponding to the probability of \(z[i + 1]\) conditional on \(z[i] = k\) is given by the elements of the batch whose rightmost coordinate is \(k\).

Similarly, the conditional distribution of \(z[i]\) given \(x[i]\) is given by the batch of observation_distribution. When the rightmost coordinate of observation_distribution is \(k\) it gives the conditional probabilities of \(x[i]\) given \(z[i] = k\). The probability distribution associated with the HiddenMarkovModel distribution is the marginal distribution of \(x[0]..., x[num\_steps - 1]\).

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

The so-called 'horseshoe' distribution is a Cauchy-Normal scale mixture, proposed as a sparsity-inducing prior for Bayesian regression. It is symmetric around zero, has heavy (Cauchy-like) tails, so that large coefficients face relatively little shrinkage, but an infinitely tall spike at 0, which pushes small coefficients towards zero. It is parameterized by a positive scalar \( \text{scale} \) parameter: higher values yield a weaker sparsity-inducing effect.

```
tfd_horseshoe(scale, validate_args = FALSE, allow_nan_stats = TRUE, name = "Horseshoe")
```

- **scale**: Floating point tensor; the scales of the distribution(s). Must contain only positive values.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

### Details

**Mathematical details**

The Horseshoe distribution is centered at zero, with scale parameter \( \text{scale} \). It is defined by:

\[
\text{horseshoe}(\text{scale} = \text{lambda}) \sim \text{Normal}(0, \lambda \ast \text{sigma})
\]

where \( \text{sigma} \sim \text{half_cauchy}(0,1) \)

**Value**

a distribution instance.
tfd_independent

Independent distribution from batch of distributions

Description

This distribution is useful for regarding a collection of independent, non-identical distributions as a single random variable. For example, the Independent distribution composed of a collection of Bernoulli distributions might define a distribution over an image (where each Bernoulli is a distribution over each pixel).

Usage

```r
tfd_independent(
    distribution,
    reinterpreted_batch_ndims = NULL,
    validate_args = FALSE,
    name = paste0("Independent", distribution$name)
)
```
Arguments

distribution  The base distribution instance to transform. Typically an instance of Distribution
reinterpreted_batch_ndims  Scalar, integer number of rightmost batch dims which will be regarded as event
dims. When NULL all but the first batch axis (batch axis 0) will be transferred
to event dimensions (analogous to tf$\text{layers}$flatten).
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid
inputs may silently render incorrect outputs. Default value: FALSE.
name  The name for ops managed by the distribution. Default value: Independent +
distribution.name.

Details

More precisely, a collection of $B$ (independent) $E$-variate random variables (rv) \{X_1, ..., X_B\}, can
be regarded as a \[B, E\]-variate random variable (X_1, ..., X_B) with probability
$p(x_1, ..., x_B) = p_1(x_1) \times ... \times p_B(x_B)$ where $p_b(x_b)$ is the probability of the \(b\)-th rv. More generally
$B, E$ can be arbitrary shapes. Similarly, the Independent distribution specifies a distribution over
[B, E]-shaped events. It operates by reinterpreting the rightmost batch dims as part of the event
dimensions. The reinterpreted_batch_ndims parameter controls the number of batch dims which
are absorbed as event dims; $reinterpreted_batch_ndims <= len(batch_shape)$. For example,
the log_prob function entails a reduce_sum over the rightmost reinterpreted_batch_ndims af-
ther calling the base distribution’s log_prob. In other words, since the batch dimension(s) index
independent distributions, the resultant multivariate will have independent components.

Mathematical Details

The probability function is,

\[
\text{prob}(x; \text{reinterpreted_batch_ndims}) = \text{tf.reduce_prod}(\text{dist.prob}(x), \text{axis}=-1-\text{range}(\text{reinterpreted_batch_ndims}))
\]

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(),
tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi,
tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(),
tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(),
tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
tfd_hidden_markov_model(), tfd_horseshoe(), tfd_inverse_gamma(), tfd_inverse_gaussian(),
tfd_joint_distribution_named(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(),
tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_normal(),
tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag(),
tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag().
The InverseGamma distribution is defined over positive real numbers using parameters concentration (aka "alpha") and scale (aka "beta").

Usage

tfd_inverse_gamma(
    concentration,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "InverseGamma"
)

Arguments

- **concentration**: Floating point tensor, the concentration params of the distribution(s). Must contain only positive values.
- **scale**: Floating point tensor, the scale params of the distribution(s). Must contain only positive values. This parameter was called rate before release 0.8.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.
Details

Mathematical Details

The probability density function (pdf) is,

\[
\text{pdf}(x; \alpha, \beta, x > 0) = x^{-(\alpha - 1)} \exp(-\beta / x) / Z
\]

\[
Z = \Gamma(\alpha) \beta^{-\alpha}
\]

where:

- concentration = \(\alpha\),
- scale = \(\beta\),
- \(Z\) is the normalizing constant, and,
- \(\Gamma\) is the gamma function.

The cumulative density function (cdf) is,

\[
\text{cdf}(x; \alpha, \beta, x > 0) = \Gamma_{\text{inc}}(\alpha, \beta / x) / \Gamma(\alpha)
\]

where \(\Gamma_{\text{inc}}\) is the [upper incomplete Gamma function](https://en.wikipedia.org/wiki/Incomplete_gamma_function).

The parameters can be intuited via their relationship to mean and variance when these moments exist,

\[
\text{mean} = \beta / (\alpha - 1) \text{ when } \alpha > 1 \quad \text{variance} = \beta^{\star 2} / (\alpha - 1)^{\star 2} / (\alpha - 2) \text{ when } \alpha > 2
\]

i.e., under the same conditions:

\[
\alpha = \text{mean}^2 / \text{variance} + 2 \quad \beta = \text{mean} \ast (\text{mean}^2 / \text{variance} + 1)
\]

Distribution parameters are automatically broadcast in all functions; see examples for details.

Samples of this distribution are reparameterized (pathwise differentiable).

The derivatives are computed using the approach described in the paper


[gamma function]: R:gamma

[upper incomplete Gamma function]: R:upper complete Gamma

[Michael Figurnov, Shakir Mohamed, Andriy Mnih. Implicit Reparameterization Gradients, 2018]: R:Michael

Value

a distribution instance.
**tfd_inverse_gaussian**  
*Inverse Gaussian distribution*

**Description**

The inverse Gaussian distribution is parameterized by a `loc` and a concentration parameter. It’s also known as the Wald distribution. Some, e.g., the Python scipy package, refer to the special case when `loc` is 1 as the Wald distribution.

**Usage**

```python
import tensorflow_probability as tfd

loc, concentration, validate_args = FALSE, allow_nan_stats = TRUE, name = "InverseGaussian"
```

**Arguments**

- `loc` Floatlng-point Tensor, the loc params. Must contain only positive values.
- `concentration` Floatlng-point Tensor, the concentration params. Must conta...
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

name  name prefixed to Ops created by this class.

Details

The "inverse" in the name does not refer to the distribution associated to the multiplicative inverse of a random variable. Rather, the cumulant generating function of this distribution is the inverse to that of a Gaussian random variable.

Mathematical Details

The probability density function (pdf) is,

\[
pdf(x; \mu, \lambda) = \left[ \frac{\lambda}{2 \pi x^3} \right]^{0.5} \exp\left\{ -\frac{\lambda(x - \mu)^2}{2 \mu^2 x} \right\}
\]

where

- \(loc = \mu\)
- \(concentration = \lambda\).

The support of the distribution is defined on (0, infinity). Mapping to R and Python scipy's parameterization:

- R: statmod::invgauss
  - mean = loc
  - shape = concentration
  - dispersion = 1 / concentration. Used only if shape is NULL.

- Python: scipy.stats.invgauss
  - \(mu = \mu / \lambda\)
  - scale = concentration

Value

a distribution instance.
Joint distribution parameterized by named distribution-making functions.

Description

This distribution enables both sampling and joint probability computation from a single model specification. A joint distribution is a collection of possibly interdependent distributions. Like JointDistributionSequential, JointDistributionNamed is parameterized by several distribution-making functions. Unlike JointDistributionNamed, each distribution-making function must have its own key. Additionally every distribution-making function’s arguments must refer to only specified keys.

Usage

tfd_joint_distribution_named(model, validate_args = FALSE, name = NULL)

Arguments

model

named list of distribution-making functions each with required args corresponding only to other keys in the named list.

validate_args

Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
name

The name for ops managed by the distribution. Default value: "JointDistributionNamed".

Details

Mathematical Details

Internally `JointDistributionNamed` implements the chain rule of probability. That is, the probability function of a length-`d` vector `x` is,

\[
p(x) = \prod_{i=0}^{d-1} p(x[i] \mid x[:i])
\]

The `JointDistributionNamed` is parameterized by a dict (or `namedtuple`) composed of either:

1. `tfp$\text{distributions}$Distribution`-like instances or,
2. functions which return a `tfp$\text{distributions}$Distribution`-like instance. The "conditioned on" elements are represented by the function's required arguments; every argument must correspond to a key in the named distribution-making functions. Distribution-makers which are directly a `Distribution`-like instance are allowed for convenience and semantically identical a zero argument function. When the maker takes no arguments it is preferable to directly provide the distribution instance.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Description

This distribution enables both sampling and joint probability computation from a single model specification.

Usage

tfd_joint_distribution_sequential(model, validate_args = FALSE, name = NULL)

Arguments

- **model**: list of either `tfp$Distribution` instances and/or functions which take the k previous distributions and returns a new `tfp$Distribution` instance.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **name**: name prefixed to Ops created by this class.

Details

A joint distribution is a collection of possibly interdependent distributions. Like `tf$Sequential`, the `JointDistributionSequential` can be specified via a list of functions (each responsible for making a `tfp$Distribution`-like instance). Unlike `tf$Sequential`, each function can depend on the output of all previous elements rather than only the immediately previous.

Mathematical Details

The `JointDistributionSequential` implements the chain rule of probability. That is, the probability function of a length-d vector x is,

\[ p(x) = \prod (p(x[i] | x[:i]) : i = 0, \ldots, (d - 1) \) \]

The `JointDistributionSequential` is parameterized by a list comprised of either:

1. `tfp$Distribution`-like instances or,
2. callables which return a `tfp$Distribution`-like instance. Each list element implements the i-th full conditional distribution, \( p(x[i] | x[:i]) \). The "conditioned on" elements are represented by the callable's required arguments. Directly providing a `Distribution` instance is a convenience and is semantically identical a zero argument callable. Denote the i-th callables non-default arguments as `args[i]`. Since the callable is the conditional manifest, 0 <= len(`args[i]`) <= i - 1. When len(`args[i]`) < i - 1, the callable only depends on a subset of the previous distributions, specifically those at indexes: \( \text{range}(i - 1, i - 1 - \text{num_args}[i], -1) \).
**tfd_kl_divergence**

Computes the Kullback–Leibler divergence.

**Description**

Denote this distribution by \( p \) and the other distribution by \( q \). Assuming \( p, q \) are absolutely continuous with respect to reference measure \( r \), the KL divergence is defined as:

\[
KL[p, q] = E_p[\log(p(X)/q(X))] = -\int_F p(x) \log q(x) \, dr(x) + \int_F p(x) \log p(x) \, dr(x) = H[p, q] - H[p]
\]

where \( F \) denotes the support of the random variable \( X \sim p \), \( H[\cdot, \cdot] \) denotes (Shannon) cross entropy, and \( H[\cdot] \) denotes (Shannon) entropy.

**Usage**

\[
tfd_kl_divergence(distribution, other, name = "kl_divergence")
\]
**Arguments**

- **distribution** The distribution being used.
- **other** `tfp$distribution$Distribution` instance.
- **name** String prepended to names of ops created by this function.

**Value**

`self$dtype` Tensor with shape `[B1, ..., Bn]` representing n different calculations of the Kullback-Leibler divergence.

**See Also**

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_mode()`, `tfd_prob()`, `tfd_quantile()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`

**Examples**

```r
  d1 <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
  d2 <- tfd_normal(loc = c(1.5, 2), scale = c(1, 0.5))
  d1 %>% tfd_kl_divergence(d2)
```

---

**tfd_kumaraswamy  Kumaraswamy distribution**

**Description**

The Kumaraswamy distribution is defined over the `(0, 1)` interval using parameters `concentration1` (aka "alpha") and `concentration0` (aka "beta"). It has a shape similar to the Beta distribution, but is easier to reparameterize.

**Usage**

```r
tfd_kumaraswamy(
  concentration1 = 1,
  concentration0 = 1,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Kumaraswamy"
)
```
Arguments

concentration1 Positive floating-point Tensor indicating mean number of successes; aka "alpha". Implies self$dtype and self$batch_shape, i.e., concentration1$shape = [N1, N2, ..., Nm] = self$batch_shape.

concentration0 Positive floating-point Tensor indicating mean number of failures; aka "beta". Otherwise has same semantics as concentration1.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

name name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

$$pdf(x; \alpha, \beta) = \alpha \times \beta \times x^{(\alpha - 1)} \times (1 - x^{\alpha})^{(\beta - 1)}$$

where:

- concentration1 = alpha,
- concentration0 = beta, Distribution parameters are automatically broadcast in all functions.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gaussian(), tfd_inverse_gaussian(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_lci, tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_bernoulli(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_poisson(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_student_t_process(), tfd_student_t,
tfd_laplace

Description

Laplace distribution with location \( \text{loc} \) and scale parameters

Usage

tfd_laplace(
    loc,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Laplace"
)

Arguments

- **loc**: Floating point tensor which characterizes the location (center) of the distribution.
- **scale**: Positive floating point tensor which characterizes the spread of the distribution.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

The probability density function (pdf) of this distribution is,

\[
pdf(x; \mu, \sigma) = \frac{\exp(-|x - \mu| / \sigma)}{Z}
\]

\[
Z = 2 \sigma
\]

where \( \text{loc} = \mu \), \( \text{scale} = \sigma \), and \( Z \) is the normalization constant.

Note that the Laplace distribution can be thought of two exponential distributions spliced together “back-to-back.” The Laplace distribution is a member of the location-scale family, i.e., it can be constructed as,
X ~ Laplace(loc=0, scale=1)
Y = loc + scale * X

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`,
`tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chi()`,
tfd_cholesky_lkj()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`,
tfd_empirical()`, `tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`,
tfd_gaussian_process()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`,
tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`,
tfd_inverse_gaussian()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential()`,
tfd_kumaraswamy()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`, `tfd_log_normal()`,
tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag()`,
tfd_multivariate_normal_full_c covariance()`, `tfd_multivariate_normal_linear_operator()`,
tfd_negative_binomial()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature()`,
tfd_poisson()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`,
tfd_sample_distribution()`, `tfd_sinh_arcsinh()`, `tfd_student_t_process()`, `tfd_student_t()`,
tfd_truncated_normal()`, `tfd_transformed_distribution()`, `tfd_triangular()`,
tfd_transformed_distribution()`, `tfd_triangular()`, `tfd_truncated_normal()`, `tfd_uniform()`,
tfd_variational_gaussian_process()`, `tfd_vector_diffeomixture()`, `tfd_vector_exponential_diag()`,
tfd_vector_exponential_linear_operator()`, `tfd_vector_laplace_diag()`, `tfd_vector_laplace_linear_operator()`,
tfd_vector_sinh_arcsinh_diag()`, `tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_wishart_linear_operator()`,
tfd_wishart_tril()`, `tfd_wishart()`, `tfd_zipf()`

tfd_linear_gaussian_state_space_model

Observation distribution from a linear Gaussian state space model

Description

The state space model, sometimes called a Kalman filter, posits a latent state vector \( z_t \) of dimension \( \text{latent}\_size \) that evolves over time following linear Gaussian transitions, \( z_{t+1} = F * z_t + N(b; Q) \) for transition matrix \( F \), bias \( b \) and covariance matrix \( Q \). At each timestep, we observe a noisy projection of the latent state \( x_t = H \* z_t + N(c; R) \). The transition and observation models may be fixed or may vary between timesteps.

Usage

```python
tfd_linear_gaussian_state_space_model(
    num_timesteps,
    transition_matrix,
    ...)```
transition_noise,
observation_matrix,
observation_noise,
initial_state_prior,
initial_step = 0L,
validate_args = FALSE,
allow_nan_stats = TRUE,
name = "LinearGaussianStateSpaceModel"
)

Arguments

num_timesteps  Integer Tensor total number of timesteps.
transition_matrix  A transition operator, represented by a Tensor or LinearOperator of shape [latent_size, latent_size], or by a callable taking as argument a scalar integer Tensor \( t \) and returning a Tensor or LinearOperator representing the transition operator from latent state at time \( t \) to time \( t + 1 \).
transition_noise  An instance of `tfd.MultivariateNormalLinearOperator` with event shape [latent_size], representing the mean and covariance of the transition noise model, or a callable taking as argument a scalar integer Tensor \( t \) and returning such a distribution representing the noise in the transition from time \( t \) to time \( t + 1 \).
observation_matrix  An observation operator, represented by a Tensor or LinearOperator of shape [observation_size, latent_size], or by a callable taking as argument a scalar integer Tensor \( t \) and returning a timestep-specific Tensor or LinearOperator.
observation_noise  An instance of `tfd.MultivariateNormalLinearOperator` with event shape [observation_size], representing the mean and covariance of the observation noise model, or a callable taking as argument a scalar integer Tensor \( t \) and returning a timestep-specific noise model.
initial_state_prior  An instance of `MultivariateNormalLinearOperator` representing the prior distribution on latent states; must have event shape [latent_size].
initial_step  optional integer specifying the time of the first modeled timestep. This is added as an offset when passing timesteps \( t \) to (optional) callables specifying timestep-specific transition and observation models.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name  name prefixed to Ops created by this class.
Details

This Distribution represents the marginal distribution on observations, \(p(x)\). The marginal \texttt{log_prob} is computed by Kalman filtering, and \texttt{sample} by an efficient forward recursion. Both operations require time linear in \(T\), the total number of timesteps.

Shapes

The event shape is \([\text{num\_timesteps}, \text{observation\_size}]\), where \text{observation\_size} is the dimension of each observation \(x_t\). The observation and transition models must return consistent shapes. This implementation supports vectorized computation over a batch of models. All of the parameters (prior distribution, transition and observation operators and noise models) must have a consistent batch shape.

Time-varying processes

Any of the model-defining parameters (prior distribution, transition and observation operators and noise models) may be specified as a callable taking an integer timestep \(t\) and returning a time-dependent value. The dimensionality (latent\_size and observation\_size) must be the same at all timesteps.

Importantly, the timestep is passed as a \texttt{Tensor}, not a Python integer, so any conditional behavior must occur \texttt{inside} the TensorFlow graph. For example, suppose we want to use a different transition model on even days than odd days. It does \textit{not} work to write

```plaintext
transition_matrix <- function(t) {
  if(t %% 2 == 0) even_day_matrix else odd_day_matrix
}
```

since the value of \(t\) is not fixed at graph-construction time. Instead we need to write

```plaintext
transition_matrix <- function(t) {
  tf$cond(tf$equal(tf$mod(t, 2), 0), function() even_day_matrix, function() odd_day_matrix)
}
```

so that TensorFlow can switch between operators appropriately at runtime.

Value

a distribution instance.

See Also

For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean()}.

Other distributions: \texttt{tfd_autoregressive()}, \texttt{tfd_batch_reshape()}, \texttt{tfd_bernoulli()}, \texttt{tfd_beta_binomial()}, \texttt{tfd_beta()}, \texttt{tfd_binomial()}, \texttt{tfd_categorical()}, \texttt{tfd_cauchy()}, \texttt{tfd_chi2()}, \texttt{tfd_chol}, \texttt{tfd_cholesky_lkj()}, \texttt{tfd_deterministic()}, \texttt{tfd_dirichlet_multinomial()}, \texttt{tfd_dirichlet()}, \texttt{tfd_empirical()}, \texttt{tfd_exponential()}, \texttt{tfd_gamma_gamma()}, \texttt{tfd_gamma()}, \texttt{tfd_gaussian_process_regression_model()}, \texttt{tfd_gaussian_process()}, \texttt{tfd_geometric()}, \texttt{tfd_gumbel()}, \texttt{tfd_half_cauchy()}, \texttt{tfd_half_normal()}, \texttt{tfd_half_normal()}.
**tfd_lkj**

**LJK distribution on correlation matrices**

**Description**

This is a one-parameter of distributions on correlation matrices. The probability density is proportional to the determinant raised to the power of the parameter: \( \text{pdf}(X; \eta) = Z(\eta) \cdot \det(X)^{\eta - 1} \), where \( Z(\eta) \) is a normalization constant. The uniform distribution on correlation matrices is the special case \( \eta = 1 \).

**Usage**

```python
import tensorflow_probability as tfd

tfd_lkj(
    dimension,
    concentration,
    input_output_cholesky = FALSE,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "LKJ"
)
```

**Arguments**

- **dimension** integer. The dimension of the correlation matrices to sample.
- **concentration** float or double Tensor. The positive concentration parameter of the LKJ distributions. The pdf of a sample matrix \( X \) is proportional to \( \det(X)^{\eta - 1} \).
- **input_output_cholesky** Logical. If TRUE, functions whose input or output have the semantics of samples assume inputs are in Cholesky form and return outputs in Cholesky form. In particular, if this flag is TRUE, input to log_prob is presumed of Cholesky form and output from sample is of Cholesky form. Setting this argument to TRUE is purely a computational optimization and does not change the underlying distribution. Additionally, validation checks which are only defined on the multiplied-out form are omitted, even if validate_args is TRUE. Default value: FALSE (i.e., input/output does not have Cholesky semantics).
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

The distribution is named after Lewandowski, Kurowicka, and Joe, who gave a sampler for the distribution in Lewandowski, Kurowicka, Joe, 2009.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


`tfd_logistic` Logistic distribution with location `loc` and scale parameters

Description

Mathematical details
Usage

tfd_logistic(
    loc,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Logistic"
)

Arguments

loc
Floating point tensor, the means of the distribution(s).

scale
Floating point tensor, the scales of the distribution(s). Must contain only positive
values.

validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid
inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details

The cumulative density function of this distribution is:

cdf(x; mu, sigma) = 1 / (1 + exp(-(x - mu) / sigma))

where loc = mu and scale = sigma.

The Logistic distribution is a member of the location-scale family, i.e., it can be constructed as,

X ~ Logistic(loc=0, scale=1)
Y = loc + scale * X

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(),
tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(),
tfd_cholesky_1k(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(),
tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(),
tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
The Logit-Normal distribution

The Logit-Normal distribution models positive-valued random variables whose logit (i.e., sigmoid_inverse, i.e., \( \log(p) - \log(1-p) \)) is normally distributed with mean \( \text{loc} \) and standard deviation \( \text{scale} \). It is constructed as the sigmoid transformation, (i.e., \( 1 / (1 + \exp(-x)) \)) of a Normal distribution.

Usage

```r
tfd_logit_normal(  
  loc,  
  scale,  
  validate_args = FALSE,  
  allow_nan_stats = TRUE,  
  name = "LogitNormal"  
)
```

Arguments

- **loc**: Floating point tensor; the means of the distribution(s).
- **scale**: Floating point tensor; the stddevs of the distribution(s). Must contain only positive values.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.
tfd_log_cdf

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

---

tfd_log_cdf  
*Log cumulative distribution function.*

Description

Given random variable X, the cumulative distribution function cdf is: \( \text{tfd_log_cdf}(x) := \log[P[X \leq x]] \) Often, a numerical approximation can be used for \( \text{tfd_log_cdf}(x) \) that yields a more accurate answer than simply taking the logarithm of the cdf when \( x \ll 1 \).

Usage

```
tfd_log_cdf(distribution, value, ...)
```

Arguments

- `distribution`  
The distribution being used.
- `value`  
float or double Tensor.
- `...`  
Additional parameters passed to Python.

Value

a Tensor of shape `sample_shape(x) + self$batch_shape` with values of type `self$dtype`.

See Also

Other distribution methods: tfd_cdf(), tfd_covariance(), tfd_cross_entropy(), tfd_entropy(), tfd_kl_divergence(), tfd_log_prob(), tfd_log_survival_function(), tfd_mean(), tfd_mode(), tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_survival_function(), tfd_variance()

Examples

```
d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
x <- d %>% tfd_sample()
d %>% tfd_log_cdf(x)
```
**Description**

The LogNormal distribution models positive-valued random variables whose logarithm is normally distributed with mean `loc` and standard deviation `scale`. It is constructed as the exponential transformation of a Normal distribution.

The LogNormal distribution models positive-valued random variables whose logarithm is normally distributed with mean `loc` and standard deviation `scale`. It is constructed as the exponential transformation of a Normal distribution.

**Usage**

```python
import tensorflow as tf

tfd_log_normal(
    loc,  # Floating-point Tensor; the means of the underlying Normal distribution(s).
scale,  # Floating-point Tensor; the std devs of the underlying Normal distribution(s).
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "LogNormal"
)

tfd_log_normal(
    loc,  # Floating-point Tensor; the means of the underlying Normal distribution(s).
scale,  # Floating-point Tensor; the std devs of the underlying Normal distribution(s).
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "LogNormal"
)
```

**Arguments**

- `loc`: Floating-point Tensor; the means of the underlying Normal distribution(s).
- `scale`: Floating-point Tensor; the std devs of the underlying Normal distribution(s).
- `validate_args`: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- `allow_nan_stats`: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- `name`: name prefixed to Ops created by this class.

**Value**

A distribution instance.
See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_choloeysky_lkj()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gama()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gama()`, `tfd_inverse_gaussian()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`, `tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag()`, `tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`, `tfd_negative_binomial()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature_compound()`, `tfd_poisson()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`.

Value

A Tensor of shape `sample_shape(x) + self$batch_shape` with values of type `self$dtype`.
See Also

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_mode()`, `tfd_prob()`, `tfd_quantile()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`

Examples

d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
x <- d %>% tfd_sample()
d %>% tfd_log_prob(x)

tfd_log_survival_function

Log survival function.

Description

Given random variable X, the survival function is defined: `tfd_log_survival_function(x) = Log[ P[X > x] ] = Log[ 1 - P[X <= x] ] = Log[ 1 - cdf(x) ]`

Usage

`tfd_log_survival_function(distribution, value, ...)`

Arguments

- `distribution`: The distribution being used.
- `value`: float or double Tensor.
- `...`: Additional parameters passed to Python.

Details

Typically, different numerical approximations can be used for the log survival function, which are more accurate than `1 - cdf(x)` when `x ≈ 1`.

Value

a Tensor of shape `sample_shape(x) + self$batch_shape` with values of type `self$dtype`.

See Also

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_mode()`, `tfd_prob()`, `tfd_quantile()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`
**tfd_mean**

## Examples

```r
d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
x <- d %>% tfd_sample()
d %>% tfd_log_survival_function(x)
```

<table>
<thead>
<tr>
<th>tfd_mean</th>
<th>Mean.</th>
</tr>
</thead>
</table>

## Description

Mean.

## Usage

```r
tfd_mean(distribution, ...)
```

## Arguments

- `distribution` The distribution being used.
- `...` Additional parameters passed to Python.

## Value

A Tensor of shape `sample_shape(x) + self$batch_shape` with values of type `self$dtype`.

## See Also

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mode()`, `tfd_prob()`, `tfd_quantile()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`

## Examples

```r
d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_mean()
```
Description

The `Mixture` object implements batched mixture distributions. The mixture model is defined by a `Categorical` distribution (the mixture) and a list of `Distribution` objects.

Usage

```r
tfd_mixture(
  cat,
  components,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  use_static_graph = FALSE,
  name = "Mixture"
)
```

Arguments

- `cat`: A `Categorical` distribution instance, representing the probabilities of distributions.
- `components`: A list or tuple of `Distribution` instances. Each instance must have the same type, be defined on the same domain, and have matching `event_shape` and `batch_shape`.
- `validate_args`: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- `allow_nan_stats`: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- `use_static_graph`: Calls to `sample` will not rely on dynamic tensor indexing, allowing for some static graph compilation optimizations, but at the expense of sampling all underlying distributions in the mixture. (Possibly useful when running on TPUs). Default value: FALSE (i.e., use dynamic indexing).
- `name`: name prefixed to Ops created by this class.

Details

Methods supported include `tfd_log_prob`, `tfd_prob`, `tfd_mean`, `tfd_sample`, and `entropy_lower_bound`.

Value

a distribution instance.
Mixture (same-family) distribution

Description

The MixtureSameFamily distribution implements a (batch of) mixture distribution where all components are from different parameterizations of the same distribution type. It is parameterized by a Categorical “selecting distribution” (over k components) and a components distribution, i.e., a Distribution with a rightmost batch shape (equal to [k]) which indexes each (batch of) component.

Usage

```r
tfd_mixture_same_family(
  mixture_distribution,
  components_distribution,
  reparameterize = FALSE,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "MixtureSameFamily"
)
```
**Arguments**

- `mixture_distribution`: `tfp$ distributions$ Categorical-like instance. Manages the probability of selecting components. The number of categories must match the rightmost batch dimension of the components$ distribution. Must have either scalar batch$ shape or batch$ shape matching components$ distribution$ batch$ shape$ [-1].

- `components_distribution`: `tfp$ distributions$ Distribution-like instance. Right-most batch dimension indexes components.

- `reparameterize`: Logical, default FALSE. Whether to reparameterize samples of the distribution using implicit reparameterization gradients (Figurnov et al., 2018). The gradients for the mixture logits are equivalent to the ones described by (Graves, 2016). The gradients for the components parameters are also computed using implicit reparameterization (as opposed to ancestral sampling), meaning that all components are updated every step. Only works when: (1) components$ distribution is fully reparameterized; (2) components$ distribution is either a scalar distribution or fully factorized (tfd$ Independent applied to a scalar distribution); (3) batch shape has a known rank. Experimental, may be slow and produce infs/NaNs.

- `validate_args`: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- `allow_nan_stats`: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

- `name`: name prefixed to Ops created by this class.

**Value**

a distribution instance.

**References**


**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`.
tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gaussian(),
tfd_inverse_gaussian(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(),
tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(),
tfd_log_normal(), tfd_logistic(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(),
tfd_multivariate_normal_diag(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_normal_tril(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(),
tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature_compound(),
tfd_poisson(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(),
tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_student_t(),
tfd_student_t_process(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_normal(), tfd_uniform(),
tfd_von_mises_fisher(), tfd_von_mises(), tfd_wishart_linear_operator(),
tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()

<table>
<thead>
<tr>
<th>tfd_mode</th>
<th>Mode</th>
</tr>
</thead>
</table>

**Description**

Mode.

**Usage**

tfd_mode(distribution, ...)

**Arguments**

distribution The distribution being used.
...
  Additional parameters passed to Python.

**Value**

A Tensor of shape `sample_shape(x) + self$batch_shape` with values of type `self$dtype`.

**See Also**

Other distribution methods: tfd_cdf(), tfd_covariance(), tfd_cross_entropy(), tfd_entropy(),
tfd_kl_divergence(), tfd_log_cdf(), tfd_log_prob(), tfd_log_survival_function(), tfd_mean(),
tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_survival_function(), tfd_variance()

**Examples**

d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_mode()
Multinomial distribution

Description

This Multinomial distribution is parameterized by `probs`, a (batch of) length-\(K\) prob (probability) vectors (\(K > 1\)) such that \(\text{tf.reduce_sum}(\text{probs}, -1) = 1\), and a `total_count` number of trials, i.e., the number of trials per draw from the Multinomial. It is defined over a (batch of) length-\(K\) vector counts such that \(\text{tf.reduce_sum}(\text{counts}, -1) = \text{total_count}\). The Multinomial is identically the Binomial distribution when \(K = 2\).

Usage

```r
tfd_multinomial(
  total_count,
  logits = NULL,
  probs = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Multinomial"
)
```

Arguments

- **total_count**
  Non-negative floating point tensor with shape broadcastable to \([N_1, \ldots, N_m]\) with \(m \geq 0\). Defines this as a batch of \(N_1 \times \ldots \times N_m\) different Multinomial distributions. Its components should be equal to integer values.

- **logits**
  Floating point tensor representing unnormalized log-probabilities of a positive event with shape broadcastable to \([N_1, \ldots, N_m, K]\) \(m \geq 0\), and the same dtype as `total_count`. Defines this as a batch of \(N_1 \times \ldots \times N_m\) different \(K\) class Multinomial distributions. Only one of `logits` or `probs` should be passed in.

- **probs**
  Positive floating point tensor with shape broadcastable to \([N_1, \ldots, N_m, K]\) \(m \geq 0\) and same dtype as `total_count`. Defines this as a batch of \(N_1 \times \ldots \times N_m\) different \(K\) class Multinomial distributions. `probs`'s components in the last portion of its shape should sum to 1. Only one of `logits` or `probs` should be passed in.

- **validate_args**
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**
  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

- **name**
  name prefixed to Ops created by this class.
The Multinomial is a distribution over \( K \)-class counts, i.e., a length-\( K \) vector of non-negative integer counts \( n = [n_0, \ldots, n_{K-1}] \). The probability mass function (pmf) is,

\[
\text{pmf}(n; \pi, N) = \prod_j (\pi_j)^{n_j} / Z \\
Z = (\prod_j n_j!) / N!
\]

where:

- \( \text{probs} = \pi = [\pi_0, \ldots, \pi_{K-1}] \), \( \pi_j > 0 \), \( \text{sum}_j \pi_j = 1 \),
- \( \text{total}_\text{count} = N \), \( N \) a positive integer,
- \( Z \) is the normalization constant, and,
- \( N! \) denotes \( N \) factorial.

Distribution parameters are automatically broadcast in all functions; see examples for details.

**Pitfalls**

The number of classes, \( K \), must not exceed:

- the largest integer representable by \( \text{self}.$\text{dtype} \), i.e., \( 2^{\text{mantissa_bits}+1} \) (IEE754),
- the maximum Tensor index, i.e., \( 2^{31}-1 \).

Note: This condition is validated only when \( \text{validate}\_	ext{args} = \text{TRUE} \).

**Value**

a distribution instance.

**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

The Multivariate Normal distribution is defined over \( \mathbb{R}^k \) and parameterized by a (batch of) length- \( k \) loc vector (aka "mu") and a (batch of) \( k \times k \) scale matrix; \( \text{covariance} = \text{scale} \times \text{scale}^T \). Where \( @ \) denotes matrix-multiplication.

### Usage

```python
tfd_multivariate_normal_diag(  
    loc = NULL,  
    scale_diag = NULL,  
    scale_identity_multiplier = NULL,  
    validate_args = FALSE,  
    allow_nan_stats = TRUE,  
    name = "MultivariateNormalDiag"
)
```

### Arguments

- **loc**
  - Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape \([B_1, ..., B_b, k]\) where \( b \geq 0 \) and \( k \) is the event size.

- **scale_diag**
  - Non-zero, floating-point Tensor representing a diagonal matrix added to scale. May have shape \([B_1, ..., B_b, k]\), \( b \geq 0 \), and characterizes \( b \)-batches of \( k \times k \) diagonal matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

- **scale_identity_multiplier**
  - Non-zero, floating-point Tensor representing a scaled-identity-matrix added to scale. May have shape \([B_1, ..., B_b]\), \( b \geq 0 \), and characterizes \( b \)-batches of scaled \( k \times k \) identity matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

- **validate_args**
  - Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**
  - Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**
  - name prefixed to Ops created by this class.
Mathematical Details

The probability density function (pdf) is,

\[
\text{pdf}(x; \text{loc}, \text{scale}) = \exp(-0.5 \|y\|^2) / Z \\
y = \text{inv}(\text{scale}) @ (x - \text{loc}) \\
Z = (2 \pi)^{(0.5 \cdot k)} |\det(\text{scale})|
\]

where:
- \text{loc} is a vector in \( \mathbb{R}^k \),
- \text{scale} is a linear operator in \( \mathbb{R}^{k \times k} \), \( \text{cov} = \text{scale} @ \text{scale}.T \),
- \( Z \) denotes the normalization constant, and,
- \( \|y\|^2 \) denotes the squared Euclidean norm of \( y \).

A (non-batch) scale matrix is:

\[
\text{scale} = \text{diag} (\text{scale}_\text{diag} + \text{scale}_\text{identity}_\text{multiplier} \cdot \text{ones}(k))
\]

where:
- \text{scale}_\text{diag}.shape = [k], and,
- \text{scale}_\text{identity}_\text{multiplier}.shape = [1].

Additional leading dimensions (if any) will index batches.

If both \text{scale}_\text{diag} and \text{scale}_\text{identity}_\text{multiplier} are NULL, then \text{scale} is the Identity matrix. The MultivariateNormal distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{MultivariateNormal}(\text{loc}=0, \text{scale}=1) \quad \# \text{Identity scale, zero shift.} \\
Y = \text{scale} @ X + \text{loc}
\]

Value

a distribution instance.

See Also

For usage examples see e.g. \text{tfd_sample()}, \text{tfd_log_prob()}, \text{tfd_mean()}.

Other distributions: \text{tfd_autoregressive()}, \text{tfd_batch_reshape()}, \text{tfd_bernoulli()}, \text{tfd_beta_binomial()}, \text{tfd_beta()}, \text{tfd_binomial()}, \text{tfd_categorical()}, \text{tfd_cauchy()}, \text{tfd_chi2()}, \text{tfd_chi}(), \text{tfd_cholesky_lkj()}, \text{tfd_deterministic()}, \text{tfd_dirichlet_multinomial()}, \text{tfd_dirichlet()}, \text{tfd_empirical()}, \text{tfd_exponential()}, \text{tfd_gamma_gamma()}, \text{tfd_gamma()}, \text{tfd_gaussian_process_regression_model()}, \text{tfd_gaussian_process()}, \text{tfd_geometric()}, \text{tfd_gumbel()}, \text{tfd_half_cauchy()}, \text{tfd_half_normal()}, \text{tfd_hidden_markov_model()}, \text{tfd_horseshoe()}, \text{tfd_independent()}, \text{tfd_inverse_gamma()}, \text{tfd_inverse_gaussian()}, \text{tfd_joint_distribution_named()}, \text{tfd_joint_distribution_sequential()}, \text{tfd_kumaraswamy()}, \text{tfd_laplace()}, \text{tfd_linear_gaussian_state_space_model()}, \text{tfd_lkj()}, \text{tfd_log_normal()}, \text{tfd_logistic()}, \text{tfd_mixture_same_family()}, \text{tfd_mixture()}, \text{tfd_multinomial}().
Multivariate normal distribution on $\mathbb{R}^k$

**Description**

The Multivariate Normal distribution is defined over $\mathbb{R}^k$ and parameterized by a (batch of) length-$k$ loc vector (aka "mu") and a (batch of) $k \times k$ scale matrix; covariance = scale @ scale. Where @ denotes matrix-multiplication.

**Usage**

```r

tfd_multivariate_normal_diag_plus_low_rank(
  loc = NULL,
  scale_diag = NULL,
  scale_identity_multiplier = NULL,
  scale_perturb_factor = NULL,
  scale_perturb_diag = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "MultivariateNormalDiagPlusLowRank"
)
```

**Arguments**

- **loc**: Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape $[B1, ..., Bb, k]$ where $b \geq 0$ and $k$ is the event size.

- **scale_diag**: Non-zero, floating-point Tensor representing a diagonal matrix added to scale. May have shape $[B1, ..., Bb, k]$, $b \geq 0$, and characterizes $b$-batches of $k \times k$ diagonal matrices added to scale. When both `scale_identity_multiplier` and `scale_diag` are NULL then scale is the Identity.

- **scale_identity_multiplier**: Non-zero, floating-point Tensor representing a scaled-identity-matrix added to scale. May have shape $[B1, ..., Bb, k]$, $b \geq 0$, and characterizes $b$-batches of scaled $k \times k$ identity matrices added to scale. When both `scale_identity_multiplier` and `scale_diag` are NULL then scale is the Identity.
scale_perturb_factor
Floating-point Tensor representing a rank-$r$ perturbation added to scale. May have shape $[B_1, ..., B_b, k, r]$, $b \geq 0$, and characterizes $b$-batches of rank-$r$ updates to scale. When NULL, no rank-$r$ update is added to scale.

scale_perturb_diag
Floating-point Tensor representing a diagonal matrix inside the rank-$r$ perturbation added to scale. May have shape $[B_1, ..., B_b, r]$, $b \geq 0$, and characterizes $b$-batches of $r \times r$ diagonal matrices inside the perturbation added to scale. When NULL, an identity matrix is used inside the perturbation. Can only be specified if scale_perturb_factor is also specified.

validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
Mathematical Details
The probability density function (pdf) is,

$$
\text{pdf}(x; \text{loc}, \text{scale}) = \exp(-0.5 \|y\|**2) / Z
$$

$$
y = \text{inv}(\text{scale}) @ (x - \text{loc})
$$

$$
Z = (2 \pi)**(0.5 \ k) \ |\text{det}(%scale)|
$$

where:
- $\text{loc}$ is a vector in $\mathbb{R}^k$,
- $\text{scale}$ is a linear operator in $\mathbb{R}^{k \times k}$, $\text{cov} = \text{scale} @ \text{scale}.T$,
- $Z$ denotes the normalization constant, and,
- $\|y\|**2$ denotes the squared Euclidean norm of $y$.

A (non-batch) scale matrix is:

$$
\text{scale} = \text{diag}(%scale\_diag + %scale\_identity\_multiplier \ \text{ones}(k)) + \text{scale\_perturb\_factor} @ \text{diag}(%scale\_perturb\_diag) @ \text{scale\_perturb\_factor}.T
$$

where:
- $%scale\_diag.\text{shape} = [k]$,
- $%scale\_identity\_multiplier.\text{shape} = []$,
- $%scale\_perturb\_factor.\text{shape} = [k, r]$, typically $k \gg r$, and,
- $%scale\_perturb\_diag.\text{shape} = [r]$.
Additional leading dimensions (if any) will index batches. If both scale_diag and scale_identity_multiplier are NULL, then scale is the Identity matrix. The MultivariateNormal distribution is a member of the location-scale family, i.e., it can be constructed as,

\[ X \sim \text{MultivariateNormal}(\text{loc}=0, \text{scale}=1) \quad \# \text{Identity scale, zero shift.} \]
\[ Y = \text{scale} \odot X + \text{loc} \]

Value

a distribution instance.

See Also

For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean()}.

Other distributions: \texttt{tfd_autoregressive()}, \texttt{tfd_batch_reshape()}, \texttt{tfd_bernoulli()}, \texttt{tfd_beta_binomial()}, \texttt{tfd_beta()}, \texttt{tfd_binomial()}, \texttt{tfd_categorical()}, \texttt{tfd_cauchy()}, \texttt{tfd_chisquare()}, \texttt{tfd_cholesky()}, \texttt{tfd_deterministic()}, \texttt{tfd_dirichlet_multinomial()}, \texttt{tfd_dirichlet()}, \texttt{tfd_empirical()}, \texttt{tfd_exponential()}, \texttt{tfd_gamma()}, \texttt{tfd_gaussian_process_regression_model()}, \texttt{tfd_gaussian_process()}, \texttt{tfd_geometric()}, \texttt{tfd_half_cauchy()}, \texttt{tfd_half_normal()}, \texttt{tfd_hidden_markov_model()}, \texttt{tfd_horseshoe()}, \texttt{tfd_independent()}, \texttt{tfd_inverse_gamma()}, \texttt{tfd_inverse_gaussian()}, \texttt{tfd_inverse_gaussian()}, \texttt{tfd_joint_distribution_named()}, \texttt{tfd_joint_distribution_sequential()}, \texttt{tfd_kumaraswamy()}, \texttt{tfd_linear_gaussian_state_space_model()}, \texttt{tfd_lkj()}, \texttt{tfd_log_normal()}, \texttt{tfd_logistic()}, \texttt{tfd_mixture_same_family()}, \texttt{tfd_mixture()}, \texttt{tfd_multinomial()}, \texttt{tfd_multivariate_normal_diag()}, \texttt{tfd_multivariate_normal_full_covariance()}, \texttt{tfd_multivariate_normal_linear_operator()}, \texttt{tfd_multivariate_student_t_linear_operator()}, \texttt{tfd_negatively_binomial()}, \texttt{tfd_negatively_binomial()}, \texttt{tfd_negatively_binomial()}, \texttt{tfd_normal()}, \texttt{tfd_one_hot_categorical()}, \texttt{tfd_pareto()}, \texttt{tfd_pixel_cnn()}, \texttt{tfd_poisson_log_normal_quadrature_compound()}, \texttt{tfd_probit_bernoulli()}, \texttt{tfd_quantized()}, \texttt{tfd_relaxed_bernoulli()}, \texttt{tfd_relaxed_categorical()}, \texttt{tfd_sample_distribution()}, \texttt{tfd_sinh_arcsinh()}, \texttt{tfd_student_t_process()}, \texttt{tfd_student_t()}, \texttt{tfd_transformed_distribution()}, \texttt{tfd_triangular()}, \texttt{tfd_truncated_normal()}, \texttt{tfd_uniform()}, \texttt{tfd_vibrational_gaussian_process()}, \texttt{tfd_vector_diffeomixture()}, \texttt{tfd_vector_exponential_diagonal()}, \texttt{tfd_vector_exponential_linear_operator()}, \texttt{tfd_vector_laplace_diagonal()}, \texttt{tfd_vector_laplace_linear_operator()}, \texttt{tfd_vector_sinh_arcsinh_diagonal()}, \texttt{tfd_von_mises_fisher()}, \texttt{tfd_von_mises()}, \texttt{tfd_wishart_linear_operator()}, \texttt{tfd_wishart_tri_l()}, \texttt{tfd_wishart()}, \texttt{tfd_zipf()}

---

tfd_multivariate_normal_full_covariance

\textit{Multivariate normal distribution on } \mathbb{R}^k

Description

The Multivariate Normal distribution is defined over \( \mathbb{R}^k \) and parameterized by a (batch of) length-
\( k \) \texttt{loc} vector (aka "mu") and a (batch of) \( k \times k \) \texttt{scale} matrix; \texttt{covariance} = \texttt{scale} \odot \texttt{scale\_T}. Where \( \odot \) denotes matrix-multiplication.
tfd_multivariate_normal_full_covariance

Usage

tfd_multivariate_normal_full_covariance(
    loc = NULL,
    covariance_matrix = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "MultivariateNormalFullCovariance"
)

Arguments

loc Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape [B1, ..., Bb, k] where b >= 0 and k is the event size.
covariance_matrix Floating-point, symmetric positive definite Tensor of same dtype as loc. The strict upper triangle of covariance_matrix is ignored, so if covariance_matrix is not symmetric no error will be raised (unless validate_args is TRUE). covariance_matrix has shape [B1, ..., Bb, k, k] where b >= 0 and k is the event size.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[ \text{pdf}(x; \text{loc}, \text{scale}) = \exp(-0.5 \ ||y||^2) / Z \]
\[ y = \text{inv}(\text{scale}) @ (x - \text{loc}) \]
\[ Z = (2 \pi)^{0.5 k} |\text{det}(\text{scale})| \]

where:

- \text{loc} is a vector in \( R^k \),
- \text{scale} is a linear operator in \( R^{k \times k} \), \text{cov} = \text{scale} @ \text{scale}.T,
- \( ||y||^2 \) denotes the normalization constant, and,
- \( ||y||^2 \) denotes the squared Euclidean norm of y.

The MultivariateNormal distribution is a member of the location-scale family, i.e., it can be constructed as,

\[ X \sim \text{MultivariateNormal} (\text{loc}=0, \text{scale}=1) \quad \# \text{Identity scale, zero shift.} \]
\[ Y = \text{scale} @ X + \text{loc} \]
The multivariate normal distribution is defined over \( \mathbb{R}^k \) and parameterized by a (batch of) length- \( k \) \text{loc} vector (aka "mu") and a (batch of) \( k \times k \) scale matrix; \( \text{covariance} = \text{scale} \times \text{scale}^\top \). Where \( \times \) denotes matrix-multiplication.

**Value**

A distribution instance.

**See Also**

For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean()}.

Other distributions: \texttt{tfd_autoregressive()}, \texttt{tfd_batch_reshape()}, \texttt{tfd_bernoulli()}, \texttt{tfd_beta_binomial()}, \texttt{tfd_beta()}, \texttt{tfd_binomial()}, \texttt{tfd_categorical()}, \texttt{tfd_cauchy()}, \texttt{tfd_chi2()}, \texttt{tfd_chi()}, \texttt{tfd_cholesky_lkj()}, \texttt{tfd_deterministic()}, \texttt{tfd_dirichlet_multinomial()}, \texttt{tfd_dirichlet()}, \texttt{tfd_empirical()}, \texttt{tfd_exponential()}, \texttt{tfd_gamma_gamma()}, \texttt{tfd_gamma()}, \texttt{tfd_gaussian_process_regression_model()}, \texttt{tfd_gaussian_process()}, \texttt{tfd_geometric()}, \texttt{tfd_gumbel()}, \texttt{tfd_half_cauchy()}, \texttt{tfd_half_normal()}, \texttt{tfd_hidden_markov_model()}, \texttt{tfd_horseshoe()}, \texttt{tfd_independent()}, \texttt{tfd_inverse_gamma()}, \texttt{tfd_inverse_gaussian()}, \texttt{tfd_joint_distribution_named()}, \texttt{tfd_joint_distribution_sequential()}, \texttt{tfd_kumaraswamy()}, \texttt{tfd_laplace()}, \texttt{tfd_linear_gaussian_state_space_model()}, \texttt{tfd_lkj()}, \texttt{tfd_log_normal()}, \texttt{tfd_logistic()}, \texttt{tfd_mixture_same_family()}, \texttt{tfd_mixture()}, \texttt{tfd_multinomial()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}, \texttt{tfd_multivariate_normal_diag()}, \texttt{tfd_multivariate_normal_tri_l()}, \texttt{tfd_multivariate_student_t_linear_operator()}, \texttt{tfd_negative_binomial()}, \texttt{tfd_normal()}, \texttt{tfd_one_hot_categorical()}, \texttt{tfd_pareto()}, \texttt{tfd_pixel_cnn()}, \texttt{tfd_poisson_log_normal_quadrature_compound()}, \texttt{tfd_poisson()}, \texttt{tfd_probit_bernoulli()}, \texttt{tfd_quantized()}, \texttt{tfd_relaxed_bernoulli()}, \texttt{tfd_relaxed_one_hot_categorical()}, \texttt{tfd_sample_distribution()}, \texttt{tfd_sinh_arcsinh()}, \texttt{tfd_student_t_process()}, \texttt{tfd_student_t()}, \texttt{tfd_transformed_distribution()}, \texttt{tfd_triangular()}, \texttt{tfd_truncated_normal()}, \texttt{tfd_uniform()}, \texttt{tfd_variational_gaussian_process()}, \texttt{tfd_vector_diffeomixture()}, \texttt{tfd_vector_exponential_diag()}, \texttt{tfd_vector_exponential_linear_operator()}, \texttt{tfd_vector_laplace_diag()}, \texttt{tfd_vector_laplace_linear_operator()}, \texttt{tfd_vector_sinh_arcsinh_diag()}, \texttt{tfd_von_mises_fisher()}, \texttt{tfd_von_mises()}, \texttt{tfd_wishart_linear_operator()}, \texttt{tfd_wishart_tri_l()}, \texttt{tfd_wishart()}, \texttt{tfd_zipf()}

---

**Description**

The Multivariate Normal distribution is defined over \( \mathbb{R}^k \) and parameterized by a (batch of) length- \( k \) \text{loc} vector (aka "mu") and a (batch of) \( k \times k \) scale matrix; \( \text{covariance} = \text{scale} \times \text{scale}^\top \). Where \( \times \) denotes matrix-multiplication.
tfd_multivariate_normal_linear_operator

Usage

```python
tfd_multivariate_normal_linear_operator(
    loc = NULL,
    scale = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "MultivariateNormalLinearOperator"
)
```

Arguments

- **loc**: Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape \([B1, \ldots, Bb, k]\) where \(b \geq 0\) and \(k\) is the event size.
- **scale**: Instance of LinearOperator with same dtype as loc and shape \([B1, \ldots, Bb, k, k]\).
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[
pdf(x; \text{loc}, \text{scale}) = \exp(-0.5 \ ||y||^2) / Z
\]

\[
y = \text{inv}(\text{scale}) @ (x - \text{loc})
\]

\[
Z = (2 \pi)^{0.5 k} |\text{det}(\text{scale})|
\]

where:

- \(\text{loc}\) is a vector in \(\mathbb{R}^k\),
- \(\text{scale}\) is a linear operator in \(\mathbb{R}^{k \times k}\), \(\text{cov} = \text{scale} @ \text{scale.T}\),
- \(Z\) denotes the normalization constant, and,
- \(\|y\|^2\) denotes the squared Euclidean norm of \(y\).

The MultivariateNormal distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{MultivariateNormal}(\text{loc}=0, \text{scale}=I) \quad \# \text{Identity scale, zero shift.}
\]

\[
Y = \text{scale} @ X + \text{loc}
\]

The batch_shape is the broadcast shape between loc and scale arguments. The event_shape is given by last dimension of the matrix implied by scale. The last dimension of loc (if provided) must broadcast with this. Recall that \(\text{covariance} = \text{scale} @ \text{scale.T}\). Additional leading dimensions (if any) will index batches.
Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


tfd_multivariate_normal_tri_l

The multivariate normal distribution on \( \mathbb{R}^k \)

Description

The Multivariate Normal distribution is defined over \( \mathbb{R}^k \) and parameterized by a (batch of) length-k `loc` vector (aka "mu") and a (batch of) k x k `scale` matrix; covariance = scale @ scale. Where `@` denotes matrix-multiplication.

Usage

tfd_multivariate_normal_tri_l(
    loc = NULL,
    scale_tril = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "MultivariateNormalTriL"
)
Arguments

- **loc**: Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape [B1, ..., Bb, k] where b >= 0 and k is the event size.

- **scale_tril**: Floating-point, lower-triangular Tensor with non-zero diagonal elements. scale_tril has shape [B1, ..., Bb, k, k] where b >= 0 and k is the event size.

- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**: name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[
\text{pdf}(x; \text{loc}, \text{scale}) = \exp(-0.5 ||y||^2) / Z
\]

\[
y = \text{inv}(\text{scale}) @ (x - \text{loc})
\]

\[
Z = (2 \pi)^{(0.5 \times k)} |\det(\text{scale})|
\]

where:

- `loc` is a vector in \(\mathbb{R}^k\),
- `scale` is a linear operator in \(\mathbb{R}^{k \times k}\), \(\text{cov} = \text{scale} @ \text{scale}.\text{T}\),
- \(Z\) denotes the normalization constant, and,
- \(||y||^2\) denotes the squared Euclidean norm of \(y\).

A (non-batch) scale matrix is:

\[
\text{scale} = \text{scale_tril}
\]

where `scale_tril` is lower-triangular \(k \times k\) matrix with non-zero diagonal, i.e., `tf$\text{diag}_\text{part}(\text{scale_tril})` != 0. Additional leading dimensions (if any) will index batches.

The MultivariateNormal distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{MultivariateNormal}(\text{loc}=0, \text{scale}=1) \quad \# \text{Identity scale, zero shift.}
\]

\[
Y = \text{scale} @ X + \text{loc}
\]

Value

a distribution instance.
See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


```
tfd_multivariate_student_t_linear_operator

Multivariate Student's t-distribution on R^k

Description

Mathematical Details

Usage

tfd_multivariate_student_t_linear_operator(  
df,  
loc,  
scale,  
validate_args = FALSE,  
allow_nan_stats = TRUE,  
name = "MultivariateStudentTLinearOperator"
)
```

Arguments

df
  A positive floating-point Tensor. Has shape [B1, ..., Bb] where b >= 0.

loc
  Floating-point Tensor. Has shape [B1, ..., Bb, k] where k is the event size.
scale
Instance of LinearOperator with a floating dtype and shape [B1, ..., Bb, k, k].

validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
The probability density function (pdf) is,

$$pdf(x; df, loc, Sigma) = \frac{(1 + ||y||^2 / df)^{(-0.5 (df + k))}}{Z}$$

where,

$$y = \text{inv}(Sigma) (x - loc)$$
$$Z = \text{abs}(\text{det}(Sigma)) \sqrt{df \ pi}^{0.5 k} \Gamma(0.5 df) / \Gamma(0.5 (df + k))$$

where:

- df is a positive scalar.
- loc is a vector in \(\mathbb{R}^k\).
- Sigma is a positive definite shape matrix in \(\mathbb{R}^{k \times k}\), parameterized as \(\text{scale} \circ \text{scale.T}\) in this class.
- Z denotes the normalization constant, and,
- \(||y||^2\) denotes the squared Euclidean norm of \(y\).

The Multivariate Student's t-distribution distribution is a member of the location-scale family, i.e., it can be constructed as,

\(X \sim \text{MultivariateT}(\text{loc} = 0, \text{scale} = 1)\) # Identity scale, zero shift.
\(Y = \text{scale} \circ X + \text{loc}\)

Value
a distribution instance.

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().
The NegativeBinomial distribution is related to the experiment of performing Bernoulli trials in sequence. Given a Bernoulli trial with probability $p$ of success, the NegativeBinomial distribution represents the distribution over the number of successes $s$ that occur until we observe $f$ failures.

**Usage**

```python
tfd_negative_binomial(
    total_count,
    logits = NULL,
    probs = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "NegativeBinomial"
)
```

**Arguments**

- `total_count`: Non-negative floating-point Tensor with shape broadcastable to `[B1,..., Bb]` with $b >= 0$ and the same dtype as `probs` or `logits`. Defines this as a batch of $N1 x ... x Nm$ different Negative Binomial distributions. In practice, this represents the number of negative Bernoulli trials to stop at (the `total_count` of failures), but this is still a valid distribution when `total_count` is a non-integer.

- `logits`: Floating-point Tensor with shape broadcastable to `[B1, ..., Bb]` where $b >= 0$ indicates the number of batch dimensions. Each entry represents logits for the probability of success for independent Negative Binomial distributions and must be in the open interval (-inf, inf). Only one of `logits` or `probs` should be specified.
probs

Positive floating-point Tensor with shape broadcastable to \([B1, ..., Bb]\) where \(b \geq 0\) indicates the number of batch dimensions. Each entry represents the probability of success for independent Negative Binomial distributions and must be in the open interval \((0, 1)\). Only one of \texttt{logits} or \texttt{probs} should be specified.

validate_args

Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats

Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name

name prefixed to Ops created by this class.

Details

The probability mass function (pmf) is,

\[
\text{pmf}(s; f, p) = p^s \cdot (1 - p)^f / Z \\
Z = s! \cdot (f - 1)! / (s + f - 1)!
\]

where:

- total_count = f,
- probs = p,
- \(Z\) is the normalizing constant, and,
- \(n!\) is the factorial of \(n\).

Value

a distribution instance.

See Also

For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean()}. Other distributions: \texttt{tfd_autoregressive()}, \texttt{tfd_batch_reshape()}, \texttt{tfd_bernoulli()}, \texttt{tfd_beta_binomial()}, \texttt{tfd_beta()}, \texttt{tfd_binomial()}, \texttt{tfd_categorical()}, \texttt{tfd_cauchy()}, \texttt{tfd_chi2()}, \texttt{tfd_chi()}, \texttt{tfd_cholesky_lkj()}, \texttt{tfd_deterministic()}, \texttt{tfd_dirichlet_multinomial()}, \texttt{tfd_dirichlet()}, \texttt{tfd_empirical()}, \texttt{tfd_exponential()}, \texttt{tfd_gamma_gamma()}, \texttt{tfd_gaussian_process_regression_model()}, \texttt{tfd_geometric()}, \texttt{tfd_gumbel()}, \texttt{tfd_half_cauchy()}, \texttt{tfd_half_normal()}, \texttt{tfd_hidden_markov_model()}, \texttt{tfd_horseshoe()}, \texttt{tfd_independent()}, \texttt{tfd_inverse_gamma()}, \texttt{tfd_inverse_gaussian()}, \texttt{tfd_joint_distribution_named()}, \texttt{tfd_joint_distribution_sequential()}, \texttt{tfd_kumaraswamy()}, \texttt{tfd_laplace()}, \texttt{tfd_linear_gaussian_state_space_model()}, \texttt{tfd_lkj()}, \texttt{tfd_log_normal()}, \texttt{tfd_logistic()}, \texttt{tfd_mixture_same_family()}, \texttt{tfd_mixture()}, \texttt{tfd_multinomial()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}, \texttt{tfd_multivariate_normal_diag()}, \texttt{tfd_multivariate_normal()}, \texttt{tfd_multivariate_normal_linear_operator()}, \texttt{tfd_multivariate_normal_tri_l()}, \texttt{tfd_multivariate_student_t()}, \texttt{tfdNormals()}, \texttt{tfd_one_hot_categorical()}, \texttt{tfd_pareto()}, \texttt{tfd_pixel_cnn()}, \texttt{tfd_poisson_log_normal_quadrature_compound()}, \texttt{tfd_poisson()}, \texttt{tfd_probit_bernoulli()}, \texttt{tfd_quantized()}, \texttt{tfd_relaxed_bernoulli()}, \texttt{tfd_relaxed_one_hot_categorical()}, \texttt{tfd_sample_distribution()}, \texttt{tfd_sinh_arcsinh()}, \texttt{tfd_student_t_process()}, \texttt{tfd_student_t()}, \texttt{tfd_student_t()}.
tfd_normal

Normal distribution with loc and scale parameters

Description
Mathematical details

Usage

tfd_normal(
    loc,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Normal"
)

Arguments

loc Floating point tensor; the means of the distribution(s).
scale floating point tensor; the std devs of the distribution(s). Must contain only pos

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid
inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details
The probability density function (pdf) is,

$$pdf(x; \mu, \sigma) = \exp(-0.5 (x - \mu)**2 / \sigma**2) / Z$$

$$Z = (2 \pi \sigma**2)**0.5$$

where loc = mu is the mean, scale = sigma is the std. deviation, and, Z is the normalization con-
stant. The Normal distribution is a member of the location-scale family, i.e., it can be constructed as,
**Description**

The categorical distribution is parameterized by the log-probabilities of a set of classes. The difference between OneHotCategorical and Categorical distributions is that OneHotCategorical is a discrete distribution over one-hot bit vectors whereas Categorical is a discrete distribution over positive integers. OneHotCategorical is equivalent to Categorical except Categorical has event_dim=() while OneHotCategorical has event_dim=K, where K is the number of classes.
tfd_one_hot_categorical

Usage

tfd_one_hot_categorical(
    logits = NULL,
    probs = NULL,
    dtype = tf$int32,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "OneHotCategorical"
)

Arguments

logits An N-D Tensor, N >= 1, representing the log probabilities of a set of Categorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of logits for each class. Only one of logits or probs should be passed in.

probs An N-D Tensor, N >= 1, representing the probabilities of a set of Categorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of probabilities for each class. Only one of logits or probs should be passed in.

dtype The type of the event samples (default: int32).

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

This class provides methods to create indexed batches of OneHotCategorical distributions. If the provided logits or probs is rank 2 or higher, for every fixed set of leading dimensions, the last dimension represents one single OneHotCategorical distribution. When calling distribution functions (e.g. dist.prob(x)), logits and x are broadcast to the same shape (if possible). In all cases, the last dimension of logits, x represents single OneHotCategorical distributions.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi1,.
**tfd_pareto**

Pareto distribution

**Description**

The Pareto distribution is parameterized by a scale and a concentration parameter.

**Usage**

```r
  tfd_pareto(
    concentration,
    scale = 1,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Pareto"
  )
```

**Arguments**

- **concentration**: Floating point tensor. Must contain only positive values.
- **scale**: Floating point tensor, equivalent to mode. scale also restricts the domain of this distribution to be in [scale, inf). Must contain only positive values. Default value: 1.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
Mathematical Details
The probability density function (pdf) is,

\[
pdf(x; \alpha, \text{scale}, x \geq \text{scale}) = \alpha \times \text{scale}^{\alpha} / x^{(\alpha + 1)}
\]

where \`\text{concentration} = \alpha\`.

Note that \`\text{scale}\` acts as a scaling parameter, since
\`
Pareto(c, \text{scale}).pdf(x) = Pareto(c, 1.).pdf(x / \text{scale})\`.

The support of the distribution is defined on \`[\text{scale}, \infty)\`.

Value
a distribution instance.

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(),
tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(),
tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(),
tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_mode(),
tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(),
tfd_inverse_gaussian(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(),
tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(),
tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(),
tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal(),
tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tril(), tfd_multivariate_student_t(),
tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pixel_cnn(),
tfd_poisson_log_normal_quadrature_compound(), tfd_poisson(), tfd_probit_bernoulli(),
tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(),
tfd_sinh_arcsinh(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(),
tfd_triangular(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_process(),
tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(),
tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(),
tfd_von_mises_fisher(), tfd_von_mises(), tfd_wishart_linear_operator(), tfd_wishart_tril(),
tfd_wishart(), tfd_zipf()
tfd_pert

*Modified PERT distribution for modeling expert predictions.*

**Description**

The PERT distribution is a loc-scale family of Beta distributions fit onto a real interval between low and high values set by the user, along with a peak to indicate the expert’s most frequent prediction, and temperature to control how sharp the peak is.

**Usage**

```r
tfd_pert(
  low,
  peak,
  high,
  temperature = 4,
  validate_args = FALSE,
  allow_nan_stats = FALSE,
  name = "Pert"
)
```

**Arguments**

- `low` lower bound
- `peak` most frequent value
- `high` upper bound
- `temperature` controls the shape of the distribution
- `validate_args` Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- `allow_nan_stats` Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- `name` name prefixed to Ops created by this class.

**Details**

The distribution is similar to a Triangular distribution (i.e. `tfd.Triangular`) but with a smooth peak.

**Mathematical Details**

In terms of a Beta distribution, PERT can be expressed as

\[ \text{PERT} \sim \text{loc} + \text{scale} \ast \text{Beta}(\text{concentration}1, \text{concentration}0) \]
where

loc = low
scale = high - low
concentration1 = 1 + temperature \times (peak - low)/(high - low)
concentration0 = 1 + temperature \times (high - peak)/(high - low)
temperature > 0

The support is [low, high]. The peak must fit in that interval: low < peak < high. The temperature is a positive parameter that controls the shape of the distribution. Higher values yield a sharper peak. The standard PERT distribution is obtained when temperature = 4.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample(), tfd_log_prob(), tfd_mean()`.

tfd_pixel_cnn

Pixel CNN++ (Salimans et al., 2017) models a distribution over image data, parameterized by a neural network. It builds on Pixel CNN and Conditional Pixel CNN, as originally proposed by (van den Oord et al., 2016). The model expresses the joint distribution over pixels as the product of conditional distributions: $p(x|h) = \prod \{ p(x[i]|x[0:i], h) : i=0, ..., d \}$, in which $p(x[i]|x[0:i], h) : i=0, ..., d$ is the probability of the $i$-th pixel conditional on the pixels that preceded it in raster order (color channels in RGB order, then left to right, then top to bottom). $h$ is optional additional data on which to condition the image distribution, such as class labels or VAE embeddings. The Pixel CNN++ network enforces the dependency structure among pixels by applying a mask to the kernels of the convolutional layers that ensures that the values for each pixel depend only on other pixels up and to the left. Pixel values are modeled with a mixture of quantized logistic distributions, which can take on a set of distinct integer values (e.g. between 0 and 255 for an 8-bit image). Color intensity $v$ of each pixel is modeled as: $v \sim \sum q[i] \times \text{quantized\_logistic}(\text{loc}[i], \text{scale}[i]) : i = 0, ..., k \}$, in which $k$ is the number of mixture components and the $q[i]$ are the Categorical probabilities over the components.

Usage

tfd_pixel_cnn(
    image_shape,
    conditional_shape = NULL,
    num_resnet = 5,
    num_hierarchies = 3,
```r
num_filters = 160,
num_logistic_mix = 10,
receptive_field_dims = c(3, 3),
dropout_p = 0.5,
resnet_activation = "concat_elu",
use_weight_norm = TRUE,
use_data_init = TRUE,
high = 255,
low = 0,
dtype = tf$float32,
name = "PixelCNN"
)
```

**Arguments**

- **image_shape**: 3D TensorShape or tuple for the [height, width, channels] dimensions of the image.
- **conditional_shape**: TensorShape or tuple for the shape of the conditional input, or NULL if there is no conditional input.
- **num_resnet**: integer, the number of layers (shown in Figure 2 of https://arxiv.org/abs/1606.05328) within each highest-level block of Figure 2 of https://pdfs.semanticscholar.org/9e90/6792f67cbdda7b7777b69284a81044857656.pdf.
- **num_hierarchies**: integer, the number of highest-level blocks (separated by expansions/contractions of dimensions in Figure 2 of https://pdfs.semanticscholar.org/9e90/6792f67cbdda7b7777b69284a81044857656.pdf.
- **num_filters**: integer, the number of convolutional filters.
- **num_logistic_mix**: integer, number of components in the logistic mixture distribution.
- **receptive_field_dims**: tuple, height and width in pixels of the receptive field of the convolutional layers above and to the left of a given pixel. The width (second element of the tuple) should be odd. Figure 1 (middle) of https://arxiv.org/abs/1606.05328 shows a receptive field of (3, 5) (the row containing the current pixel is included in the height). The default of (3, 3) was used to produce the results in https://pdfs.semanticscholar.org/9e90/6792f67cbdda7b7777b69284a81044857656.pdf.
- **dropout_p**: float, the dropout probability. Should be between 0 and 1.
- **resnet_activation**: string, the type of activation to use in the resnet blocks. May be ‘concat_elu’, ‘elu’, or ‘relu’.
- **use_weight_norm**: logical, if TRUE then use weight normalization (works only in Eager mode).
- **use_data_init**: logical, if TRUE then use data-dependent initialization (has no effect if use_weight_norm is FALSE).
- **high**: integer, the maximum value of the input data (255 for an 8-bit image).
- **low**: integer, the minimum value of the input data.
- **dtype**: Data type of the Distribution.
- **name**: string, the name of the Distribution.
Plackett-Luce distribution over permutations.

Description

The Plackett-Luce distribution is defined over permutations of fixed length. It is parameterized by a positive score vector of same length. This class provides methods to create indexed batches of PlackettLuce distributions. If the provided scores is rank 2 or higher, for every fixed set of leading dimensions, the last dimension represents one single PlackettLuce distribution. When calling
distribution functions (e.g. dist.log_prob(x)), scores and x are broadcast to the same shape (if possible). In all cases, the last dimension of scores, x represents single PlackettLuce distributions.

Usage

tfd_plackett_luce(
  scores,
  dtype = tf$int32,
  validate_args = FALSE,
  allow_nan_stats = FALSE,
  name = "PlackettLuce"
)

Arguments

scores An N-D Tensor, N >= 1, representing the scores of a set of elements to be permuted. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of scores for the elements.

dtype The type of the event samples (default: int32).

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

Mathematical Details

The Plackett-Luce is a distribution over permutation vectors p of length k where the permutation p is an arbitrary ordering of k indices {0, 1, ..., k-1}.

The probability mass function (pmf) is,

pmf(p; s) = prod_i s_{p_i} / (Z - Z_i)
Z = sum_{j=0}^{k-1} s_j
Z_i = sum_{j=0}^{i-1} s_{p_j} for i>0 and 0 for i=0

where scores = s = [s_0, ..., s_{k-1}], s_i >= 0.

Samples from Plackett-Luce distribution are generated sequentially as follows.

Initialize normalization 'N_0 = Z'
For 'i' in '{0, 1, ..., k-1}'
  1. Sample i-th element of permutation
     'p_i ~ Categorical(probs=[s_0/N_i, ..., s_{k-1}/N_i])'
  2. Update normalization
3. Mask out sampled index for subsequent rounds
   \( s_{p_i} = 0 \)

Return \( p \)

Alternately, an equivalent way to sample from this distribution is to sort Gumbel perturbed log-scores (Aditya et al. 2019)

\[
p = \text{argsort}(\log s + g) \sim \text{PlackettLuce}(s)
g = [g_0, \ldots, g_{k-1}], g_i \sim \text{Gumbel}(0, 1)
\]

**Value**

a distribution instance.

**References**


**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

---

**tfd_poisson**

**Poisson distribution**

**Description**

The Poisson distribution is parameterized by an event rate parameter.

**Usage**

```python
  tfd_poisson(
      rate = NULL,
      log_rate = NULL,
      interpolate_nondiscrete = TRUE,
      validate_args = FALSE,
      allow_nan_stats = TRUE,
      name = "Poisson"
  )
```
Arguments

rate Floating point tensor, the rate parameter. rate must be positive. Must specify exactly one of rate and log_rate.

log_rate Floating point tensor, the log of the rate parameter. Must specify exactly one of rate and log_rate.

interpolate_nondiscrete Logical. When FALSE, log_prob returns -inf (and prob returns 0) for non-integer inputs. When TRUE, log_prob evaluates the continuous function k * log_rate - lgamma(k+1) - rate, which matches the Poisson pmf at integer arguments k (note that this function is not itself a normalized probability log-density). Default value: TRUE.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

Mathematical Details
The probability mass function (pmf) is,

\[ \text{pmf}(k; \lambda, k \geq 0) = \frac{\lambda^k / k!}{Z} \]

where rate = lambda and Z is the normalizing constant.

Value
a distribution instance.

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chisq(), tfd_chi(), tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_...
**tfd_poisson_log_normal_quadrature_compound**

### Description

The **PoissonLogNormalQuadratureCompound** is an approximation to a Poisson-LogNormal compound distribution, i.e.,

\[
p(k | \text{loc}, \text{scale}) \approx \int_{\mathbb{R}_+} d\log \text{Normal}(l | \text{loc}, \text{scale}) \text{ Poisson}(k | l)
\]

**Usage**

```r

tfd_poisson_log_normal_quadrature_compound(
  loc,
  scale,
  quadrature_size = 8,
  quadrature_fn = tfp$distributions$quadrature_scheme_lognormal_quantiles,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "PoissonLogNormalQuadratureCompound"
)
```

### Arguments

- **loc** float-like (batch of) scalar Tensor; the location parameter of the LogNormal prior.
- **scale** float-like (batch of) scalar Tensor; the scale parameter of the LogNormal prior.
- **quadrature_size** integer scalar representing the number of quadrature points.
- **quadrature_fn** Function taking `loc`, `scale`, `quadrature_size`, `validate_args` and returning `tuple(grid,probs)` representing the LogNormal grid and corresponding normalized weight. Default value: `quadrature_scheme_lognormal_quantiles`. 

---

**tfd_poisson_log_normal_quadrature_compound**

**PoissonLogNormalQuadratureCompound** distribution

---

### Description

The PoissonLogNormalQuadratureCompound is an approximation to a Poisson-LogNormal compound distribution, i.e.,

\[
p(k | \text{loc}, \text{scale}) = \int_{\mathbb{R}_+} d\log \text{Normal}(l | \text{loc}, \text{scale}) \text{ Poisson}(k | l)
\]

**approx= sum{ prob[d] Poisson(k | lambda(grid[d])) : d=0, ..., deg-1 }**

### Usage

```r

tfd_poisson_log_normal_quadrature_compound(
  loc,
  scale,
  quadrature_size = 8,
  quadrature_fn = tfp$distributions$quadrature_scheme_lognormal_quantiles,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "PoissonLogNormalQuadratureCompound"
)
```

### Arguments

- **loc** float-like (batch of) scalar Tensor; the location parameter of the LogNormal prior.
- **scale** float-like (batch of) scalar Tensor; the scale parameter of the LogNormal prior.
- **quadrature_size** integer scalar representing the number of quadrature points.
- **quadrature_fn** Function taking `loc`, `scale`, `quadrature_size`, `validate_args` and returning `tuple(grid,probs)` representing the LogNormal grid and corresponding normalized weight. Default value: `quadrature_scheme_lognormal_quantiles`. 

---
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name  name prefixed to Ops created by this class.

Details
By default, the grid is chosen as quantiles of the LogNormal distribution parameterized by loc, scale and the prob vector is [1. / quadrature_size]*quadrature_size.

In the non-approximation case, a draw from the LogNormal prior represents the Poisson rate parameter. Unfortunately, the non-approximate distribution lacks an analytical probability density function (pdf). Therefore the PoissonLogNormalQuadratureCompound class implements an approximation based on quadrature. Note: although the PoissonLogNormalQuadratureCompound is approximately the Poisson-LogNormal compound distribution, it is itself a valid distribution. Viz., it possesses a sample, log_prob, mean, variance, etc. which are all mutually consistent.

Mathematical Details
The PoissonLogNormalQuadratureCompound approximates a Poisson-LogNormal compound distribution. Using variable-substitution and numerical quadrature (default: based on LogNormal quantiles) we can redefine the distribution to be a parameter-less convex combination of deg different Poisson samples. That is, defined over positive integers, this distribution is parameterized by a (batch of) loc and scale scalars.

The probability density function (pdf) is,

\[\text{pdf}(k | \text{loc}, \text{scale}, \text{deg}) = \sum \{ \text{prob}[d] \text{Poisson}(k | \text{lambda} = \exp(\text{grid}[d])) : d=0, \ldots, \text{deg}-1 \}\]

Note: probs returned by (optional) quadrature_fn are presumed to be either a length-quadrature_size vector or a batch of vectors in 1-to-1 correspondence with the returned grid. (I.e., broadcasting is only partially supported.)

Value
a distribution instance.

See Also
For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chisq()`, `tfd_cholesky_lkj()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`, `tfd_inverse_gaussian()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential()`.
tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(),
tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(),
tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t(),
tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(),
tfd_poisson(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(),
tfd_relaxed_one_hot_categorical(),
tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_student_t_process(), tfd_student_t(),
tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_normal(),
tfd_uniform(),
tfd_variational_gaussian_process(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(),
tfd_vector_exponential_linear_operator(),
tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(),
tfd_vector_sinh_arcsinh_diag(),
tfd_von_mises_fisher(), tfd_von_mises(),
tfd_wishart_linear_operator(),
tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()

| tfd_prob | Probability density/mass function. |

**Description**

Probability density/mass function.

**Usage**

```r
tfd_prob(distribution, value, ...)
```

**Arguments**

- `distribution` The distribution being used.
- `value` float or double Tensor.
- `...` Additional parameters passed to Python.

**Value**

a Tensor of shape `sample_shape(x) + self$batch_shape` with values of type `self$dtype`.

**See Also**

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`,
`tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`,
`tfd_mode()`, `tfd_quantile()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`

**Examples**

```r
d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
x <- d %>% tfd_sample()
d %>% tfd_prob(x)
```
ProbitBernoulli distribution.

Description

The ProbitBernoulli distribution with probs parameter, i.e., the probability of a 1 outcome (vs a 0 outcome). Unlike a regular Bernoulli distribution, which uses the logistic (aka 'sigmoid') function to go from the un-constrained parameters to probabilities, this distribution uses the CDF of the standard normal distribution:

\[
p(x=1; \text{probits}) = 0.5 * (1 + \text{erf}(\text{probits} / \sqrt{2}))
\]

\[
p(x=0; \text{probits}) = 1 - p(x=1; \text{probits})
\]

Where \(\text{erf}\) is the error function. A typical application of this distribution is in probit regression.

Usage

`tfd_probit_bernoulli(
  probits = NULL,
  probs = NULL,
  dtype = tf$int32,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "ProbitBernoulli"
)
```

Arguments

probits An N-D Tensor representing the probit-odds of a 1 event. Each entry in the Tensor parameterizes an independent ProbitBernoulli distribution where the probability of an event is normal_cdf(probits). Only one of probits or probs should be passed in.

probs An N-D Tensor representing the probability of a 1 event. Each entry in the Tensor parameterizes an independent ProbitBernoulli distribution. Only one of probits or probs should be passed in.

dtype The type of the event samples. Default: int32.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.
tfd_quantile

Quantile function. Aka "inverse cdf" or "percent point function".

Description

Given random variable X and p in [0, 1], the quantile is: \( \text{tfd_quantile}(p) := x \) such that \( P[X <= x] == p \)

Usage

tfd_quantile(distribution, value, ...)

Arguments

distribution The distribution being used.
value float or double Tensor.
... Additional parameters passed to Python.
tfd_quantized

Value

A Tensor of shape `sample_shape(x) + self$batch_shape` with values of type `self$dtype`.

See Also

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`,
`tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`,
`tfd_mode()`, `tfd_prob()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`

Examples

d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_quantile(0.5)

tfd_quantized

Distribution representing the quantization \( Y = \text{ceiling}(X) \)

Description

Definition in Terms of Sampling

Usage

tfd_quantized(
  distribution,
  low = NULL,
  high = NULL,
  validate_args = FALSE,
  name = "QuantizedDistribution"
)

Arguments

distribution The base distribution class to transform. Typically an instance of Distribution.
low Tensor with same dtype as this distribution and shape able to be added to samples. Should be a whole number. Default NULL. If provided, base distribution’s prob should be defined at low.
high Tensor with same dtype as this distribution and shape able to be added to samples. Should be a whole number. Default NULL. If provided, base distribution’s prob should be defined at high - 1. high must be strictly greater than low.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
name name prefixed to Ops created by this class.
Details

1. Draw $X$
2. Set $Y \leftarrow \text{ceiling}(X)$
3. If $Y < \text{low}$, reset $Y \leftarrow \text{low}$
4. If $Y > \text{high}$, reset $Y \leftarrow \text{high}$
5. Return $Y$

Definition in Terms of the Probability Mass Function

Given scalar random variable $X$, we define a discrete random variable $Y$ supported on the integers as follows:

$$P[Y = j] := P[X \leq \text{low}], \text{ if } j = \text{low},$$
$$:= P[X > \text{high} - 1], \text{ if } j = \text{high},$$
$$:= 0, \text{ if } j < \text{low} \text{ or } j > \text{high},$$
$$:= P[j - 1 < X \leq j], \text{ all other } j.$$  

Conceptually, without cutoffs, the quantization process partitions the real line $\mathbb{R}$ into half open intervals, and identifies an integer $j$ with the right endpoints:

$$R = \ldots (-\infty, -1\)[-1, 0](0, 1](1, \infty) \ldots$$
$$j = \ldots -1 \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \ldots$$

$P[Y = j]$ is the mass of $X$ within the $j$th interval. If $\text{low} = 0$, and $\text{high} = 2$, then the intervals are redrawn and $j$ is re-assigned:

$$R = (-\infty, 0](0, 1](1, \infty)$$
$$j = \quad 0 \quad 1 \quad 2$$

$P[Y = j]$ is still the mass of $X$ within the $j$th interval.

@section References:


Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`...
The RelaxedBernoulli is a distribution over the unit interval (0,1), which continuously approximates a Bernoulli. The degree of approximation is controlled by a temperature: as the temperature goes to 0 the RelaxedBernoulli becomes discrete with a distribution described by the logits or probs parameters, as the temperature goes to infinity the RelaxedBernoulli becomes the constant distribution that is identically 0.5.

Usage

```r
tfd_relaxed_bernoulli(
  temperature,
  logits = NULL,
  probs = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "RelaxedBernoulli"
)
```

Arguments

- **temperature**: An 0-D Tensor, representing the temperature of a set of RelaxedBernoulli distributions. The temperature should be positive.

- **logits**: An N-D Tensor representing the log-odds of a positive event. Each entry in the Tensor parametrizes an independent RelaxedBernoulli distribution where the probability of an event is sigmoid(logits). Only one of logits or probs should be passed in.
probs

A N-D Tensor representing the probability of a positive event. Each entry in the Tensor parameterizes an independent Bernoulli distribution. Only one of logits or probs should be passed in.

validate_args

Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats

Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name

name prefixed to Ops created by this class.

Details

The RelaxedBernoulli distribution is a reparameterized continuous distribution that is the binary special case of the RelaxedOneHotCategorical distribution (Maddison et al., 2016; Jang et al., 2016). For details on the binary special case see the appendix of Maddison et al. (2016) where it is referred to as BinConcrete. If you use this distribution, please cite both papers.

Some care needs to be taken for loss functions that depend on the log-probability of RelaxedBernoullis, because computing log-probabilities of the RelaxedBernoulli can suffer from underflow issues. In many cases loss functions such as these are invariant under invertible transformations of the random variables. The KL divergence, found in the variational autoencoder loss, is an example. Because RelaxedBernoullis are sampled by a Logistic random variable followed by a \textit{tf$sigmoid} operation, one solution is to treat the Logistic as the random variable and \textit{tf$sigmoid} as downstream. The KL divergences of two Logistices, which are always followed by a \textit{tf.sigmoid} op, is equivalent to evaluating KL divergences of RelaxedBernoulli samples. See Maddison et al., 2016 for more details where this distribution is called the BinConcrete. An alternative approach is to evaluate Bernoulli log probability or KL directly on relaxed samples, as done in Jang et al., 2016. In this case, guarantees on the loss are usually violated. For instance, using a Bernoulli KL in a relaxed ELBO is no longer a lower bound on the log marginal probability of the observation. Thus care and early stopping are important.

Value

a distribution instance.

See Also

For usage examples see e.g. \textit{tfd_sample()}, \textit{tfd_log_prob()}, \textit{tfd_mean()}.

The RelaxedOneHotCategorical is a distribution over random probability vectors, vectors of positive real values that sum to one, which continuously approximates a OneHotCategorical. The degree of approximation is controlled by a temperature: as the temperature goes to 0 the RelaxedOneHotCategorical becomes discrete with a distribution described by the logits or probs parameters, as the temperature goes to infinity the RelaxedOneHotCategorical becomes the constant distribution that is identically the constant vector of (1/event_size, ..., 1/event_size). The RelaxedOneHotCategorical distribution was concurrently introduced as the Gumbel-Softmax (Jang et al., 2016) and Concrete (Maddison et al., 2016) distributions for use as a reparameterized continuous approximation to the Categorical one-hot distribution. If you use this distribution, please cite both papers.

Usage

tfd_relaxed_one_hot_categorical(
  temperature,
  logits = NULL,
  probs = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "RelaxedOneHotCategorical"
)

Arguments

- **temperature**: An 0-D Tensor, representing the temperature of a set of RelaxedOneHotCategorical distributions. The temperature should be positive.
- **logits**: An N-D Tensor, N >= 1, representing the log probabilities of a set of RelaxedOneHotCategorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of logits for each class. Only one of logits or probs should be passed in.
An N-D Tensor, N >= 1, representing the probabilities of a set of RelaxedOne-HotCategorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of probabilities for each class. Only one of logits or probs should be passed in.

Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Value
a distribution instance.

References

- Eric Jang, Shixiang Gu, and Ben Poole. Categorical Reparameterization with Gumbel-Softmax. 2016.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

tfd_sample

Generate samples of the specified shape.

Description

Note that a call to `tfd_sample()` without arguments will generate a single sample.

Usage

```
tfd_sample(distribution, sample_shape = list(), ...)
```

Arguments

- `distribution`: The distribution being used.
- `sample_shape`: 0D or 1D int32 Tensor. Shape of the generated samples.
- `...`: Additional parameters passed to Python.

Value

A Tensor with prepended dimensions `sample_shape`.

See Also

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_mode()`, `tfd_prob()`, `tfd_quantile()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`

Examples

```
d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_sample()
```

tfd_sample_distribution

Sample distribution via independent draws.

Description

This distribution is useful for reducing over a collection of independent, identical draws. It is otherwise identical to the input distribution.
tfd_sample_distribution

Usage

tfd_sample_distribution(
    distribution,
    sample_shape = list(),
    validate_args = FALSE,
    name = NULL
  )

Arguments

distribution  The base distribution instance to transform. Typically an instance of Distribution.
sample_shape  integer scalar or vector Tensor representing the shape of a single sample.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
name  The name for ops managed by the distribution. Default value: NULL (i.e., 'Sample' + distribution$name).

Details

Mathematical Details The probability function is,

\[
p(x) = \prod_{i=0}^{(n-1)} p(x[i])
\]

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet(), tfd_dirichlet_multinomial(), tfd_empirical(), tfd_exponential(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_multinomial(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature_compound(), tfd_poisson(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sinh_arcsinh(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_process(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(),...
The SinhArcsinh transformation of a distribution on (-inf, inf)

Description

This distribution models a random variable, making use of a SinhArcsinh transformation (which has adjustable tailweight and skew), a rescaling, and a shift. The SinhArcsinh transformation of the Normal is described in great depth in Sinh-arcsinh distributions. Here we use a slightly different parameterization, in terms of tailweight and skewness. Additionally we allow for distributions other than Normal, and control over scale as well as a "shift" parameter loc.

Usage

```r
tfd_sinh_arcsinh(
  loc,
  scale,
  skewness = NULL,
  tailweight = NULL,
  distribution = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "SinhArcsinh"
)
```

Arguments

- **loc**: Floating-point Tensor.
- **scale**: Tensor of same dtype as loc.
- **skewness**: Skewness parameter. Default is 0.0 (no skew).
- **tailweight**: Tailweight parameter. Default is 1.0 (unchanged tailweight)
- **distribution**: `tf$Distributions$Distribution-like instance. Distribution that is transformed to produce this distribution. Default is `tfd_normal(0,1)`. Must be a scalar-batch, scalar-event distribution. Typically distribution$reparameterization_type = FULLY_REPARAMETERIZED or it is a function of non-trainable parameters. Warning: If you backprop through a SinhArcsinh sample and distribution is not FULLY_REPARAMETERIZED yet is a function of trainable variables, then the gradient will be incorrect!
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
ame prefixed to Ops created by this class.

Details
Mathematical Details
Given random variable \( Z \), we define the SinhArcsinh transformation of \( Z, Y \), parameterized by
(\( \text{loc} \), \( \text{scale} \), \( \text{skewness} \), \( \text{tailweight} \)), via the relation:

\[
Y := \text{loc} + \text{scale} \times F(Z) \times \left(2 / F_0(2)\right)
\]

\[
F(Z) := \sinh\left( (\text{Arcsinh}(Z) + \text{skewness}) \times \text{tailweight} \right)
\]

\[
F_0(Z) := \sinh\left( \text{Arcsinh}(Z) \times \text{tailweight} \right)
\]

This distribution is similar to the location-scale transformation \( L(Z) := \text{loc} + \text{scale} \times Z \)
in the following ways:

- If \( \text{skewness} = 0 \) and \( \text{tailweight} = 1 \) (the defaults), \( F(Z) = Z \), and then \( Y = L(Z) \) exactly.
- \( \text{loc} \) is used in both to shift the result by a constant factor.
- The multiplication of \( \text{scale} \) by \( 2 / F_0(2) \) ensures that if \( \text{skewness} = 0 \) \( P[Y - \text{loc} <= 2 \times \text{scale}] = P[L(Z) - \text{loc} <= 2 \times \text{scale}] \). Thus it can be said that the weights in the tails of \( Y \) and \( L(Z) \) beyond \( \text{loc} + 2 \times \text{scale} \) are the same.

This distribution is different than \( \text{loc} + \text{scale} \times Z \) due to the reshaping done by \( F \):

- Positive (negative) \( \text{skewness} \) leads to positive (negative) skew.
- Positive skew means, the mode of \( F(Z) \) is "tilted" to the right.
- Positive skew means positive values of \( F(Z) \) become more likely, and negative values become
  less likely.
- Larger (smaller) \( \text{tailweight} \) leads to fatter (thinner) tails.
- Fatter tails mean larger values of \( |F(Z)| \) become more likely.
- \( \text{tailweight} < 1 \) leads to a distribution that is "flat" around \( Y = \text{loc} \), and a very steep drop-off
  in the tails.
- \( \text{tailweight} > 1 \) leads to a distribution more peaked at the mode with heavier tails.

To see the argument about the tails, note that for \( |Z| >> 1 \) and \( |Z| >> (|\text{skewness}| \times \text{tailweight})^{\text{tailweight}} \), we have \( Y \approx 0.5 \times Z^{\text{tailweight}} e^{(\text{sign}(Z) \times \text{skewness} \times \text{tailweight})} \).

To see the argument regarding multiplying \( \text{scale} \) by \( 2 / F_0(2) \),

\[
P[(Y - \text{loc}) / \text{scale} <= 2] = P[2 / F_0(Z) <= 2]
= P[F(Z) <= F_0(2)]
= P[Z <= 2] \quad (\text{if } F = F_0).
\]

Value
a distribution instance.
See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


---

tfd_stddev  

**Standard deviation.**

**Description**

Standard deviation is defined as, \( \text{stddev} = E[(X - E[X])**2]**0.5 \) where \( X \) is the random variable associated with this distribution, \( E \) denotes expectation, and \( \text{Var}$shape = \text{batch_shape} + \text{event_shape}. \)

**Usage**

```python
  tfd_stddev(distribution, ...)  
```

**Arguments**

- **distribution**  
  The distribution being used.

- **...**  
  Additional parameters passed to Python.

**Value**

A Tensor of shape \( \text{sample_shape}(x) + \text{self$batch_shape} \) with values of type \( \text{self$dtype} \).
See Also

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_mode()`, `tfd_prob()`, `tfd_quantile()`, `tfd_sample()`, `tfd_survival_function()`, `tfd_variance()`

Examples

```r
  d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
  d %>% tfd_stddev()
```

```r
tfd_student_t  Student's t-distribution
```

Description

This distribution has parameters: degree of freedom `df`, location `loc`, and `scale`.

Usage

```r
tfd_student_t(
  df,
  loc,
  scale,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "StudentT"
)
```

Arguments

- `df` Floating-point `Tensor`. The degrees of freedom of the distribution(s). `df` must contain only positive values.
- `loc` Floating-point `Tensor`. The mean(s) of the distribution(s).
- `scale` Floating-point `Tensor`. The scaling factor(s) for the distribution(s). Note that `scale` is not technically the standard deviation of this distribution but has semantics more similar to standard deviation than variance.
- `validate_args` Logical, default `FALSE`. When `TRUE` distribution parameters are checked for validity despite possibly degrading runtime performance. When `FALSE` invalid inputs may silently render incorrect outputs. Default value: `FALSE`.
- `allow_nan_stats` Logical, default `TRUE`. When `TRUE`, statistics (e.g., mean, mode, variance) use the value `NaN` to indicate the result is undefined. When `FALSE`, an exception is raised if one or more of the statistic’s batch members are undefined.
- `name` name prefixed to Ops created by this class.
Details

Mathematical details

The probability density function (pdf) is,

\[ \text{pdf}(x; \text{df}, \text{mu}, \text{sigma}) = \frac{1 + y^2 / \text{df}}{Z} \]

where,

\[ y = \frac{x - \text{mu}}{\text{sigma}} \]

\[ Z = \text{abs}(\text{sigma}) \sqrt{\text{df} \pi} \Gamma(0.5 \text{df}) / \Gamma(0.5 (\text{df} + 1)) \]

where:

- loc = mu,
- scale = sigma, and,
- Z is the normalization constant, and,
- Gamma is the gamma function. The StudentT distribution is a member of the location-scale family, i.e., it can be constructed as,

\[ X \sim \text{StudentT}(|df|, \text{loc}=0, \text{scale}=1) \]

\[ Y = \text{loc} + \text{scale} \times X \]

Notice that scale has semantics more similar to standard deviation than variance. However it is not actually the std. dev.; the Student’s t-distribution std. dev. is scale \( \sqrt{\text{df}/(\text{df}-2)} \) when \( \text{df} > 2 \).

Samples of this distribution are reparameterized (pathwise differentiable). The derivatives are computed using the approach described in the paper Michael Figurnov, Shakir Mohamed, Andriy Mnih. Implicit Reparameterization Gradients, 2018

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature_compound(), tfd_poisson(), tfd_pareto(), tfd_probit_bernoulli().
Marginal distribution of a Student’s T process at finitely many points

Description
A Student’s T process (TP) is an indexed collection of random variables, any finite collection of which are jointly Multivariate Student’s T. While this definition applies to finite index sets, it is typically implicit that the index set is infinite; in applications, it is often some finite dimensional real or complex vector space. In such cases, the TP may be thought of as a distribution over (real- or complex-valued) functions defined over the index set.

Usage
tfd_student_t_process(
    df,
    kernel,
    index_points,
    mean_fn = NULL,
    jitter = 1e-06,
    validate_args = FALSE,
    allow_nan_stats = FALSE,
    name = "StudentTProcess"
)

Arguments

- **df**: Positive Floating-point Tensor representing the degrees of freedom. Must be greater than 2.
- **kernel**: PositiveSemidefiniteKernel-like instance representing the TP’s covariance function.
- **index_points**: float Tensor representing finite (batch of) vector(s) of points in the index set over which the TP is defined. Shape has the form [b1, ..., bB, e, f1, ..., fF] where F is the number of feature dimensions and must equal kernel.feature_ndims and e is the number (size) of index points in each batch. Ultimately this distribution corresponds to a e-dimensional multivariate Student’s T. The batch shape must be broadcastable with kernel.batch_shape and any batch dims yielded by mean_fn.
Function that acts on `index_points` to produce a (batch of) vector(s) of mean values at `index_points`. Takes a `Tensor` of shape `[b1, ..., bB, f1, ..., fF]` and returns a `Tensor` whose shape is broadcastable with `[b1, ..., bB]`. Default value: `NULL` implies constant zero function.

A `float` scalar `Tensor` added to the diagonal of the covariance matrix to ensure positive definiteness of the covariance matrix. Default value: `1e-6`.

Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name prefixed to Ops created by this class.

Details

Just as Student’s T distributions are fully specified by their degrees of freedom, location and scale, a Student’s T process can be completely specified by a degrees of freedom parameter, mean function and covariance function.

Let `S` denote the index set and `K` the space in which each indexed random variable takes its values (again, often R or C). The mean function is then a map `m: S -> K`, and the covariance function, or kernel, is a positive-definite function `k: (S x S) -> K`. The properties of functions drawn from a TP are entirely dictated (up to translation) by the form of the kernel function.

This Distribution represents the marginal joint distribution over function values at a given finite collection of points `[x[1], ..., x[N]]` from the index set `S`. By definition, this marginal distribution is just a multivariate Student’s T distribution, whose mean is given by the vector `[m(x[1]), ..., m(x[N])]` and whose covariance matrix is constructed from pairwise applications of the kernel function to the given inputs:

\[
\begin{bmatrix}
  k(x[1], x[1]) & k(x[1], x[2]) & \ldots & k(x[1], x[N]) \\
  k(x[2], x[1]) & k(x[2], x[2]) & \ldots & k(x[2], x[N]) \\
  \vdots & \vdots & \ddots & \vdots \\
  k(x[N], x[1]) & k(x[N], x[2]) & \ldots & k(x[N], x[N])
\end{bmatrix}
\]

For this to be a valid covariance matrix, it must be symmetric and positive definite; hence the requirement that `k` be a positive definite function (which, by definition, says that the above procedure will yield PD matrices). Note also we use a parameterization as suggested in Shat et al. (2014), which requires `df` to be greater than 2. This allows for the covariance for any finite dimensional marginal of the TP (a multivariate Student’s T distribution) to just be the PD matrix generated by the kernel.

Mathematical Details

The probability density function (pdf) is a multivariate Student’s T whose parameters are derived from the TP’s properties:
pdf(x; df, index_points, mean_fn, kernel) = MultivariateStudentT(df, loc, K)
K = (df - 2) / df * (kernel.matrix(index_points, index_points) + jitter * eye(N))
loc = (x - mean_fn(index_points))^T @ K @ (x - mean_fn(index_points))

where:
- df is the degrees of freedom parameter for the TP,
- index_points are points in the index set over which the TP is defined,
- mean_fn is a callable mapping the index set to the TP’s mean values,
- kernel is PositiveSemidefiniteKernel-like and represents the covariance function of the
  TP,
- jitter is added to the diagonal to ensure positive definiteness up to machine precision (oth-
  erwise Cholesky-decomposition is prone to failure),
- eye(N) is an N-by-N identity matrix.

Value
a distribution instance.

References
- Amar Shah, Andrew Gordon Wilson, and Zoubin Ghahramani. Student-t Processes as Alter-

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(),
tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(),
tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(),
tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(),
tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(),
tfd_inverse_gaussian(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(),
tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(),
tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(),
tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t(),
tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(),
tfd_poisson_log_normal_quadrature_compound(), tfd_poisson(), tfd_probit_bernoulli(),
tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(),
tfd_sinh_arcsinh(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(),
tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_process(), tfd_vector_diffeomixture(),
tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diagonal(),
tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_von_mises_fisher(),
tfd_von_mises(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(),
tfd_zipf())
tfd_survival_function

Description

Given random variable X, the survival function is defined: \( \text{tfd_survival_function}(x) = P[X > x] = 1 - P[X \leq x] = 1 - \text{cdf}(x) \).

Usage

\[
\text{tfd_survival_function}(\text{distribution, value, ...})
\]

Arguments

- \text{distribution} \quad \text{The distribution being used.}
- \text{value} \quad \text{float or double Tensor.}
- \text{...} \quad \text{Additional parameters passed to Python.}

Value

\text{a Tensor of shape} \ \text{sample_shape(x)} + \text{self$batch_shape} \ \text{with values of type} \ \text{self$dtype}.

See Also

Other distribution\_methods: \text{tfd\_cdf()}, \text{tfd\_covariance()}, \text{tfd\_cross\_entropy()}, \text{tfd\_entropy()}, \text{tfd\_kl\_divergence()}, \text{tfd\_log\_cdf()}, \text{tfd\_log\_prob()}, \text{tfd\_log\_survival\_function()}, \text{tfd\_mean()}, \text{tfd\_mode()}, \text{tfd\_prob()}, \text{tfd\_quantile()}, \text{tfd\_sample()}, \text{tfd\_stddev()}, \text{tfd\_variance()}

Examples

\[
d <- \text{tfd} \text{\_normal}(\text{loc} = \text{c}(1, 2), \text{scale} = \text{c}(1, 0.5))
x <- d \%>\% \text{tfd} \text{\_sample()}
d \%>\% \text{tfd} \text{\_survival\_function(x)}
\]

tfd_transformed_distribution

A Transformed Distribution

Description

A TransformedDistribution models \( p(y) \) given a base distribution \( p(x) \), and a deterministic, invertible, differentiable transform, \( Y = g(X) \). The transform is typically an instance of the Bijector class and the base distribution is typically an instance of the Distribution class.
Usage

tfd_transformed_distribution(
distribution,
bijector,
batch_shape = NULL,
event_shape = NULL,
kwags_split_fn = NULL,
validate_args = FALSE,
parameters = NULL,
name = NULL
)

Arguments

distribution  The base distribution instance to transform. Typically an instance of Distribution.
bijector      The object responsible for calculating the transformation. Typically an instance of Bijector.
batch_shape   integer vector Tensor which overrides distribution batch_shape; valid only if distribution.is_scalar_batch().
event_shape   integer vector Tensor which overrides distribution event_shape; valid only if distribution.is_scalar_event().
kwags_split_fn Python callable which takes a kwags dict and returns a tuple of kwags dicts for each of the distribution and bijector parameters respectively. Default value: _default_kwags_split_fn (i.e., lambda kwags: (kwags.get('distribution_kwags', {}), kwags.get('bijector_kwags', {}))).
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
parameters    Locals dict captured by subclass constructor, to be used for copy/slice re-instantiation operations.
name          The name for ops managed by the distribution. Default value: bijector.name + distribution.name.

Details

A Bijector is expected to implement the following functions:

- forward,
- inverse,
- inverse_log_det_jacobian.

The semantics of these functions are outlined in the Bijector documentation.

We now describe how a TransformedDistribution alters the input/outputs of a Distribution associated with a random variable (rv) X. Write cdf(Y=y) for an absolutely continuous cumulative distribution function of random variable Y; write the probability density function pdf(Y=y) := d^k / (dy_1,...,dy_k) cdf(Y=y)
for its derivative wrt to \( Y \) evaluated at \( y \). Assume that \( Y = g(X) \) where \( g \) is a deterministic diffeomorphism, i.e., a non-random, continuous, differentiable, and invertible function. Write the inverse of \( g \) as \( X = g^{-1}(Y) \) and \((J \circ g)(x)\) for the Jacobian of \( g \) evaluated at \( x \).

A TransformedDistribution implements the following operations:

- **sample**: Mathematically: \( Y = g(X) \) Programmatically: \( \text{bijector}.\text{forward} (\text{distribution}.\text{sample}(...)) \)
- **log_prob**: Mathematically: \((\log \circ \text{pdf})(Y=y) = (\log \circ \text{pdf} \circ g^{-1})(y) + (\log \circ \text{abs} \circ \text{det} \circ J \circ g^{-1})(y)\) Programmatically: \( \text{distribution}.\text{log_prob}(\text{bijector}.\text{inverse}(y)) + \text{bijector}.\text{inverse_log_det_jacobian}(y) \)
- **log_cdf**: Mathematically: \((\log \circ \text{cdf})(Y=y) = (\log \circ \text{cdf} \circ g^{-1})(y)\) Programmatically: \( \text{distribution}.\text{log_cdf}(\text{bijector}.\text{inverse}(x)) \)
- and similarly for: cdf, prob, log_survival_function, survival_function.

**Value**

a distribution instance.

**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


---

**tfd_triangular**

*Triangular distribution with low, high and peak parameters*

**Description**

The parameters low, high and peak must be shaped in a way that supports broadcasting (e.g., high ~ low is a valid operation).
tfd_triangular

Usage

tfd_triangular(
    low = 0,
    high = 1,
    peak = 0.5,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Triangular"
)

Arguments

low Floating point tensor, lower boundary of the output interval. Must have low < high. Default value: 0.
high Floating point tensor, upper boundary of the output interval. Must have low < high. Default value: 1.
peak Floating point tensor, mode of the output interval. Must have low <= peak and peak <= high. Default value: 0.5.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name name prefixed to Ops created by this class.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tril(), tfd_multivariate_student_t(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature_compound(), tfd_poisson(), tfd_probit_bernoulli(), tfd_triangular
The truncated normal is a normal distribution bounded between low and high (the pdf is 0 outside these bounds and renormalized). Samples from this distribution are differentiable with respect to loc, scale as well as the bounds, low and high, i.e., this implementation is fully reparameterizable. For more details, see here.

**Usage**

tfd_truncated_normal(
    loc,
    scale,
    low,
    high,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "TruncatedNormal"
)

**Arguments**

- **loc**  
  Floating point tensor; the means of the distribution(s).

- **scale**  
  Floating point tensor; the stddevs of the distribution(s). Must contain only positive values.

- **low**  
  float Tensor representing lower bound of the distribution’s support. Must be such that low < high.

- **high**  
  float Tensor representing upper bound of the distribution’s support. Must be such that low < high.

- **validate_args**  
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**  
  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**  
  name prefixed to Ops created by this class.
Mathematical Details

The probability density function (pdf) of this distribution is:

\[
\text{pdf}(x; \text{loc}, \text{scale}, \text{low}, \text{high}) = \\
\begin{cases} 
(2 \pi)^{-0.5} \exp(-0.5 y^2) / (\text{scale} \times z) & \text{for } \text{low} \leq x \leq \text{high} \\
0 & \text{otherwise} 
\end{cases}
\]

\[
y = (x - \text{loc})/\text{scale} \\
z = \text{NormalCDF}((\text{high} - \text{loc}) / \text{scale}) - \text{NormalCDF}((\text{low} - \text{loc}) / \text{scale})
\]

where:

- \text{NormalCDF} is the cumulative density function of the Normal distribution with 0 mean and unit variance.

This is a scalar distribution so the event shape is always scalar and the dimensions of the parameters defined the batch_shape.

Value

a distribution instance.

See Also

For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean()}.

Other distributions: \texttt{tfd_autoregressive()}, \texttt{tfd_batch_reshape()}, \texttt{tfd_bernoulli()}, \texttt{tfd_beta_binomial()}, \texttt{tfd_beta()}, \texttt{tfd_binomial()}, \texttt{tfd_categorical()}, \texttt{tfd_cauchy()}, \texttt{tfd_chi2()}, \texttt{tfd_chi()}, \texttt{tfd_cholesky_lkj()}, \texttt{tfd_deterministic()}, \texttt{tfd_dirichlet_multinomial()}, \texttt{tfd_dirichlet()}, \texttt{tfd_empirical()}, \texttt{tfd_exponential()}, \texttt{tfd_gamma_gamma()}, \texttt{tfd_gamma()}, \texttt{tfd_gaussian_process_regression_model()}, \texttt{tfd_gaussian_process()}, \texttt{tfd_geometric()}, \texttt{tfd_gumbel()}, \texttt{tfd_half_cauchy()}, \texttt{tfd_half_normal()}, \texttt{tfd_hidden_markov_model()}, \texttt{tfd_horseshoe()}, \texttt{tfd_independent()}, \texttt{tfd_inverse_gamma()}, \texttt{tfd_inverse_gaussian()}, \texttt{tfd_joint_distribution_named()}, \texttt{tfd_joint_distribution_sequential()}, \texttt{tfd_kumaraswamy()}, \texttt{tfd_laplace()}, \texttt{tfd_linear_gaussian_state_space_model()}, \texttt{tfd_lkj()}, \texttt{tfd_log_normal()}, \texttt{tfd_logistic()}, \texttt{tfd_mixture_same_family()}, \texttt{tfd_mixture()}, \texttt{tfd_multinomial()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}, \texttt{tfd_multivariate_normal_diag()}, \texttt{tfd_multivariate_normal()},
**tfd_uniform**  
*Uniform distribution with low and high parameters*

**Description**

Mathematical Details

**Usage**

```r
tfd_uniform(
    low = 0,
    high = 1,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Uniform"
)
```

**Arguments**

- **low**
  - Floating point tensor, lower boundary of the output interval. Must have \( \text{low} < \text{high} \).

- **high**
  - Floating point tensor, upper boundary of the output interval. Must have \( \text{low} < \text{high} \).

- **validate_args**
  - Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**
  - Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**
  - name prefixed to Ops created by this class.

**Details**

The probability density function (pdf) is,

\[
\text{pdf}(x; \, a, \, b) = \frac{\mathbb{I}[\text{a} \leq x < \text{b}]}{\text{Z}}
\]

\[
\text{Z} = \text{b} - \text{a}
\]

where

- \( \text{low} = a \),
- \( \text{high} = b \),
- \( \text{Z} \) is the normalizing constant, and
- \( \mathbb{I}[\text{predicate}] \) is the indicator function for predicate.

The parameters low and high must be shaped in a way that supports broadcasting (e.g., high - low is a valid operation).
tfd_variance

**Description**

Variance is defined as, \( \text{Var} = E[(X - E[X])^2] \) where \( X \) is the random variable associated with this distribution, \( E \) denotes expectation, and \( \text{Var} \text{shape} = \text{batch_shape} + \text{event_shape} \).

**Usage**

\[
\text{tfd_variance}(\text{distribution}, ...)\]

**Arguments**

- **distribution** The distribution being used.
- ... Additional parameters passed to Python.

**Value**

A Tensor of shape \( \text{sample_shape}(x) + \text{self.batch_shape} \) with values of type \( \text{self.dtype} \).
See Also

Other distribution methods: 
- `tfd_cdf()`
- `tfd_covariance()`
- `tfd_cross_entropy()`
- `tfd_entropy()`
- `tfd_kl_divergence()`
- `tfd_log_cdf()`
- `tfd_log_prob()`
- `tfd_log_survival_function()`
- `tfd_mean()`
- `tfd_mode()`
- `tfd_prob()`
- `tfd_quantile()`
- `tfd_sample()`
- `tfd_stddev()`
- `tfd_survival_function()`

Examples

```r
  d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
  d %>% tfd_variance()
```

**tfd_variational_gaussian_process**

*Posterior predictive of a variational Gaussian process*

Description

This distribution implements the variational Gaussian process (VGP), as described in Titsias (2009) and Hensman (2013). The VGP is an inducing point-based approximation of an exact GP posterior. Ultimately, this Distribution class represents a marginal distribution over function values at a collection of `index_points`. It is parameterized by

- a kernel function,
- a mean function,
- the (scalar) observation noise variance of the normal likelihood,
- a set of index points,
- a set of inducing index points, and
- the parameters of the (full-rank, Gaussian) variational posterior distribution over function values at the inducing points, conditional on some observations.

Usage

```r
tfd_variational_gaussian_process(
  kernel,
  index_points,
  inducing_index_points,
  variational_inducing_observations_loc,
  variational_inducing_observations_scale,
  mean_fn = NULL,
  observation_noise_variance = 0,
  predictive_noise_variance = 0,
  jitter = 1e-06,
  validate_args = FALSE,
  allow_nan_stats = FALSE,
  name = "VariationalGaussianProcess"
)
```
Arguments

kernel  PositiveSemidefiniteKernel-like instance representing the GP’s covariance function.

index_points  float Tensor representing finite (batch of) vector(s) of points in the index set over which the VGP is defined. Shape has the form \([b1, ..., bB, e1, f1, ..., fF]\) where \(F\) is the number of feature dimensions and must equal kernel\$feature_ndims and \(e1\) is the number (size) of index points in each batch (we denote it \(e1\) to distinguish it from the number of inducing index points, denoted \(e2\) below). Ultimately the VariationalGaussianProcess distribution corresponds to an \(e1\)-dimensional multivariate normal. The batch shape must be broadcastable with kernel\$batch_shape, the batch shape of inducing_index_points, and any batch dims yielded by mean_fn.

inducing_index_points  float Tensor of locations of inducing points in the index set. Shape has the form \([b1, ..., bB, e2, f1, ..., fF]\), just like index_points. The batch shape components needn’t be identical to those of index_points, but must be broadcast compatible with them.

variational_inducing_observations_loc  float Tensor; the mean of the (full-rank Gaussian) variational posterior over function values at the inducing points, conditional on observed data. Shape has the form \([b1, ..., bB, e2]\), where \(b1, ..., bB\) is broadcast compatible with other parameters’ batch shapes, and \(e2\) is the number of inducing points.

variational_inducing_observations_scale  float Tensor; the scale matrix of the (full-rank Gaussian) variational posterior over function values at the inducing points, conditional on observed data. Shape has the form \([b1, ..., bB, e2, e2]\), where \(b1, ..., bB\) is broadcast compatible with other parameters and \(e2\) is the number of inducing points.

mean_fn  function that acts on index points to produce a (batch of) vector(s) of mean values at those index points. Takes a Tensor of shape \([b1, ..., bB, f1, ..., fF]\) and returns a Tensor whose shape is (broadcastable with) \([b1, ..., bB]\). Default value: NULL implies constant zero function.

observation_noise_variance  float Tensor representing the variance of the noise in the Normal likelihood distribution of the model. May be batched, in which case the batch shape must be broadcastable with the shapes of all other batched parameters (kernel\$batch_shape, index_points, etc.). Default value: 0.

predictive_noise_variance  float Tensor representing additional variance in the posterior predictive model. If NULL, we simply re-use observation_noise_variance for the posterior predictive noise. If set explicitly, however, we use the given value. This allows us, for example, to omit predictive noise variance (by setting this to zero) to obtain noiseless posterior predictions of function values, conditioned on noisy observations.

jitter  float scalar Tensor added to the diagonal of the covariance matrix to ensure positive definiteness of the covariance matrix. Default value: 1e-6.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid
inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name  name prefixed to Ops created by this class.

Details

A VGP is "trained" by selecting any kernel parameters, the locations of the inducing index points,
and the variational parameters. Titsias (2009) and Hensman (2013) describe a variational lower
bound on the marginal log likelihood of observed data, which this class offers through the variational_loss
method (this is the negative lower bound, for convenience when plugging into a TF Optimizer’s
minimize function). Training may be done in minibatches.

Titsias (2009) describes a closed form for the optimal variational parameters, in the case of suffi-
ciently small observational data (ie, small enough to fit in memory but big enough to warrant ap-
proximating the GP posterior). A method to compute these optimal parameters in terms of the full
observational data set is provided as a staticmethod, optimal_variational_posterior. It returns
a MultivariateNormalLinearOperator instance with optimal location and scale parameters.

Mathematical Details

Notation We will in general be concerned about three collections of index points, and it’ll be good
to give them names:

- x[1], ..., x[N]: observation index points – locations of our observed data.
- z[1], ..., z[M]: inducing index points – locations of the "summarizing" inducing points
- t[1], ..., t[P]: predictive index points – locations where we are making posterior predictions
  based on observations and the variational parameters.

To lighten notation, we’ll use X, Z, T to denote the above collections. Similarly, we’ll denote by f(X)
the collection of function values at each of the x[i], and by Y, the collection of (noisy) observed data at each x[i]. We’ll denote kernel matrices generated from pairs of index points as K_tt, K_xt, K_tz, etc, e.g.,

\[
K_{tz} = 
\begin{vmatrix}
  k(t[1], z[1]) & k(t[1], z[2]) & \ldots & k(t[1], z[M]) \\
  k(t[2], z[1]) & k(t[2], z[2]) & \ldots & k(t[2], z[M]) \\
  \vdots & \vdots & \ddots & \vdots \\
  k(t[P], z[1]) & k(t[P], z[2]) & \ldots & k(t[P], z[M])
\end{vmatrix}
\]

Preliminaries A Gaussian process is an indexed collection of random variables, any finite collection
of which are jointly Gaussian. Typically, the index set is some finite-dimensional, real vector space,
and indeed we make this assumption in what follows. The GP may then be thought of as a distri-
bution over functions on the index set. Samples from the GP are functions on the whole index set;
these can’t be represented in finite compute memory, so one typically works with the marginals at a
finite collection of index points. The properties of the GP are entirely determined by its mean function $m$ and covariance function $k$. The generative process, assuming a mean-zero normal likelihood with stddev $\sigma$, is

$$\mathbf{f} \sim \mathcal{GP}(m, k)$$

$$\mathbf{Y} | \mathbf{f}(\mathbf{X}) \sim \mathcal{N}(\mathbf{f}(\mathbf{X}), \sigma), \; i = 1, \ldots, N$$

In finite terms (i.e., marginalizing out all but a finite number of $\mathbf{f}(\mathbf{X}), \sigma$), we can write

$$\mathbf{f}(\mathbf{X}) \sim \mathcal{MVN}(\text{loc}=\mathbf{m}(\mathbf{X}), \text{cov} = \mathbf{K}_{xx})$$

$$\mathbf{Y} | \mathbf{f}(\mathbf{X}) \sim \mathcal{N}(\mathbf{f}(\mathbf{X}), \sigma), \; i = 1, \ldots, N$$

Posterior inference is possible in analytical closed form but becomes intractible as data sizes get large. See Rasmussen (2006) for details.

The VGP

The VGP is an inducing point-based approximation of an exact GP posterior, where two approximating assumptions have been made:

1. function values at non-inducing points are mutually independent conditioned on function values at the inducing points,

2. the (expensive) posterior over function values at inducing points conditional on observations is replaced with an arbitrary (learnable) full-rank Gaussian distribution,

$$q(\mathbf{f}(\mathbf{Z})) = \mathcal{MVN}(\text{loc}=\mathbf{m}, \text{scale} = \mathbf{S}),$$

where $\mathbf{m}$ and $\mathbf{S}$ are parameters to be chosen by optimizing an evidence lower bound (ELBO). The posterior predictive distribution becomes

$$q(\mathbf{f}(\mathbf{T})) = \text{integral } d\mathbf{f}(\mathbf{Z}) \; p(\mathbf{f}(\mathbf{T}) | \mathbf{f}(\mathbf{Z})) \; q(\mathbf{f}(\mathbf{Z})) = \mathcal{MVN}(\text{loc} = \mathbf{A} @ \mathbf{m}, \text{scale} = \mathbf{B}^{1/2})$$

where

$$\mathbf{A} = \mathbf{K}_{tz} @ \mathbf{K}_{zz}^{-1}$$

$$\mathbf{B} = \mathbf{K}_{tt} - \mathbf{A} @ (\mathbf{K}_{zz} - \mathbf{S} \mathbf{S}^T) \mathbf{A}^T$$

The approximate posterior predictive distribution $q(\mathbf{f}(\mathbf{T}))$ is what the VariationalGaussianProcess class represents.

Model selection in this framework entails choosing the kernel parameters, inducing point locations, and variational parameters. We do this by optimizing a variational lower bound on the marginal log likelihood of observed data. The lower bound takes the following form (see Titsias (2009) and Hensman (2013) for details on the derivation):

$$L(\mathbf{Z}, \mathbf{m}, \mathbf{S}, \mathbf{Y}) = \text{MVN}(\text{loc}=(\mathbf{K}_{zx} @ \mathbf{K}_{zz}^{-1}) @ \mathbf{m}, \text{scale\_diag} = \sigma).\log\_\text{prob}(\mathbf{Y}) -$$

$$\text{Tr}(\mathbf{K}_{xx} - \mathbf{K}_{zx} @ \mathbf{K}_{zz}^{-1} @ \mathbf{K}_{xz}) +$$

$$\text{Tr}(\mathbf{S} @ \mathbf{S}^T @ \mathbf{K}_{zz}^{-1} @ \mathbf{K}_{zx} @ \mathbf{K}_{xz} @ \mathbf{K}_{zz}^{-1})) / (2 * \sigma^2) -$$

$$\text{KL}(q(\mathbf{f}(\mathbf{Z})) || p(\mathbf{f}(\mathbf{Z})))$$
where in the final KL term, \( p(f(Z)) \) is the GP prior on inducing point function values. This variational lower bound can be computed on minibatches of the full data set \((X, Y)\). A method to compute the negative variational lower bound is implemented as `VariationalGaussianProcess.variational_loss`.

Optimal variational parameters

As described in Titsias (2009), a closed form optimum for the variational location and scale parameters, \( m \) and \( S \), can be computed when the observational data are not prohibitively voluminous. The `optimal_variational_posterior` function computes the optimal variational posterior distribution over inducing point function values in terms of the GP parameters (mean and kernel functions), inducing point locations, observation index points, and observations. Note that the inducing index points must still be optimized even when these parameters are known functions of the inducing index points. The optimal parameters are computed as follows:

\[
C = \sigma^{-2} \left( K_{zz} + K_{zx} @ K_{xz}\right)^{-1}
\]

optimal Gaussian covariance: \( K_{zz} @ C @ K_{zz} \)

optimal Gaussian location: \( \sigma^{-2} K_{zz} @ C @ K_{zx} @ Y \)

Value

a distribution instance.

References


See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

The VectorDeterministic distribution is parameterized by a batch point loc in \( \mathbb{R}^k \). The distribution is supported at this point only, and corresponds to a random variable that is constant, equal to loc.

### Usage

```
tfd_vector_deterministic(loc, atol = NULL, rtol = NULL, validate_args = FALSE, allow_nan_stats = TRUE, name = "VectorDeterministic")
```

### Arguments

- **loc**: Numeric Tensor of shape \([B_1, \ldots, B_b, k]\), with \( b \geq 0, k \geq 0 \) The point (or batch of points) on which this distribution is supported.
- **atol**: Non-negative Tensor of same dtype as loc and broadcastable shape. The absolute tolerance for comparing closeness to loc. Default is 0.
- **rtol**: Non-negative Tensor of same dtype as loc and broadcastable shape. The relative tolerance for comparing closeness to loc. Default is 0.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

### Details

See [Degenerate rv](#).

### Value

a distribution instance.
See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

tfd_vector_diffeomixture

*VectorDiffeomixture distribution*

Description

A vector diffeomixture (VDM) is a distribution parameterized by a convex combination of \( K \) component loc vectors, \( \text{loc}[k], k = 0,..,K-1 \), and \( K \) scale matrices \( \text{scale}[k], k = 0,.., K-1 \). It approximates the following compound distribution \( p(x) = \int p(x | z) p(z) \, dz \), where \( z \) is in the \( K \)-simplex, and \( p(x | z) := p(x | \text{loc} = \sum_k z[k] \text{loc}[k], \text{scale} = \sum_k z[k] \text{scale}[k]) \).

Usage

```r
  tfd_vector_diffeomixture(
    mix_loc,  # float-like Tensor with shape [b1, ..., bB, K-1]. In terms of samples, larger mix_loc[...,k] => Z is more likely to put more weight on its kth component.
    temperature,  # float-like Tensor. Broadcastable with mix_loc. In terms of samples, smaller temperature means one component is more likely to dominate. I.e., smaller temperature makes the VDM look more like a standard mixture of \( K \) components.
    distribution,  # tfp$distributions$Distribution-like instance. Distribution from which iid samples are used as input to the selected affine transformation. Must be a scalar-batch, scalar-event distribution. Typically \( \text{distribution$reparameterization_type} = \text{FULLY_REPARAMETERIZED} \) or it is a function of non-trainable parameters. WARNING: If you backprop through a VectorDiffeomixture sample and the distribution is not \text{FULLY_REPARAMETERIZED} yet is a function of trainable variables, then the gradient will be incorrect!
    loc = NULL,  # scale = NULL,  # quadrature_size = 8,  # quadrature_fn = tfp$distributions$quadrature_scheme_softmaxnormal_quantiles,  # validate_args = FALSE,  # allow_nan_stats = TRUE,  # name = "VectorDiffeomixture"
  )
```

Arguments

- `mix_loc` float-like Tensor with shape \([b1, ..., bB, K-1]\). In terms of samples, larger \( \text{mix}\_\text{loc} [..., k] \Rightarrow Z \) is more likely to put more weight on its \( k \)th component.
- `temperature` float-like Tensor. Broadcastable with `mix_loc`. In terms of samples, smaller temperature means one component is more likely to dominate. I.e., smaller temperature makes the VDM look more like a standard mixture of \( K \) components.
- `distribution` tfp$distributions$Distribution-like instance. Distribution from which iid samples are used as input to the selected affine transformation. Must be a scalar-batch, scalar-event distribution. Typically `distribution$reparameterization_type = \text{FULLY_REPARAMETERIZED}` or it is a function of non-trainable parameters. WARNING: If you backprop through a VectorDiffeomixture sample and the distribution is not \text{FULLY_REPARAMETERIZED} yet is a function of trainable variables, then the gradient will be incorrect!
loc
Length-K list of float-type Tensors. The k-th element represents the shift used for the k-th affine transformation. If the k-th item is NULL, loc is implicitly \(0\). When specified, must have shape \([B_1, \ldots, B_b, d]\) where \(b \geq 0\) and \(d\) is the event size.

scale
Length-K list of LinearOperators. Each should be positive-definite and operate on a d-dimensional vector space. The k-th element represents the scale used for the k-th affine transformation. LinearOperators must have shape \([B_1, \ldots, B_b, d, d]\), \(b \geq 0\), i.e., characterizes b-batches of d x d matrices.

quadrature_size
integer scalar representing number of quadrature points. Larger quadrature_size means \(q_N(x)\) better approximates \(p(x)\).

quadrature_fn
Function taking \(normal_loc, normal_scale, quadrature_size, validate_args\) and returning tuple(grid,probs) representing the SoftmaxNormal grid and corresponding normalized weight. Default value: \(quadrature_scheme_softmaxnormal_quantiles\).

validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

name
name prefixed to Ops created by this class.

Details
The integral \(\int p(x \mid z) p(z) \, dz\) is approximated with a quadrature scheme adapted to the mixture density \(p(z)\). The \(N\) quadrature points \(z_{[N,n]}\) and weights \(w_{[N,n]}\) (which are non-negative and sum to 1) are chosen such that \(q_N(x) := \sum_{n=1}^N w_{[n,N]} p(x \mid z_{[N,n]}) \rightarrow p(x)\) as \(N \rightarrow \infty\). Since \(q_N(x)\) is in fact a mixture (of \(N\) points), we may sample from \(q_N\) exactly. It is important to note that the VDM is defined as \(q_N\) above, and \textit{not} \(p(x)\). Therefore, sampling and pdf may be implemented as exact (up to floating point error) methods.

A common choice for the conditional \(p(x \mid z)\) is a multivariate Normal. The implemented marginal \(p(z)\) is the SoftmaxNormal, which is a \(K\)-1 dimensional Normal transformed by a SoftmaxCentered bijector, making it a density on the \(K\)-simplex. That is, \(Z = \text{SoftmaxCentered}(X), X = \text{Normal}(mix\_loc / temperature, 1 / temperature)\).

The default quadrature scheme chooses \(z_{[N,n]}\) as \(N\) midpoints of the quantiles of \(p(z)\) (generalized quantiles if \(K > 2\)). See Dillon and Langmore (2018) for more details.

About Vector distributions in TensorFlow.

The VectorDiffeomixture is a non-standard distribution that has properties particularly useful in \textit{variational Bayesian methods}. Conditioned on a draw from the SoftmaxNormal, \(x \mid z\) is a vector whose components are linear combinations of affine transformations, thus it is itself an affine transformation.

Note: The marginals \(X_1\mid v, \ldots, X_d\mid v\) are \textit{not} generally identical to some parameterization of distribution. This is due to the fact that the sum of draws from distribution are not generally itself the same distribution.
About Diffeomixtures and reparameterization.

The VectorDiffeomixture is designed to be reparameterized, i.e., its parameters are only used to transform samples from a distribution which has no trainable parameters. This property is important because backprop stops at sources of stochasticity. That is, as long as the parameters are used after the underlying source of stochasticity, the computed gradient is accurate. Reparameterization means that we can use gradient-descent (via backprop) to optimize Monte-Carlo objectives. Such objectives are a finite-sample approximation of an expectation and arise throughout scientific computing.

WARNING: If you backprop through a VectorDiffeomixture sample and the "base" distribution is both: not FULLY_REPARAMETERIZED and a function of trainable variables, then the gradient is not guaranteed correct!

Value

a distribution instance.

References


See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

tfd_vector_exponential_diag

The vectorization of the Exponential distribution on \( \mathbb{R}^k \)

Description

The vector exponential distribution is defined over a subset of \( \mathbb{R}^k \), and parameterized by a (batch of) length-\( k \) loc vector and a (batch of) \( k \times k \) scale matrix: covariance = scale @ scale.T, where @ denotes matrix-multiplication.

Usage

```python
tfd_vector_exponential_diag(
    loc = NULL,
    scale_diag = NULL,
    scale_identity_multiplier = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "VectorExponentialDiag"
)
```

Arguments

- **loc**
  
  Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape \([B1, ..., Bb, k]\) where \( b \geq 0 \) and \( k \) is the event size.

- **scale_diag**
  
  Non-zero, floating-point Tensor representing a diagonal matrix added to scale. May have shape \([B1, ..., Bb, k]\), \( b \geq 0 \), and characterizes \( b \)-batches of \( k \times k \) diagonal matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

- **scale_identity_multiplier**
  
  Non-zero, floating-point Tensor representing a scaled-identity-matrix added to scale. May have shape \([B1, ..., Bb, k]\), \( b \geq 0 \), and characterizes \( b \)-batches of scaled \( k \times k \) identity matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

- **validate_args**
  
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**
  
  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**
  
  name prefixed to Ops created by this class.
Details

The probability density function (pdf) is defined over the image of the scale matrix + loc, applied to the positive half-space: Supp = \{loc + scale @ x : x in \mathbb{R}^k, x_1 > 0, ..., x_k > 0\}. On this set,

$$pdf(y; \text{loc}, \text{scale}) = \frac{\exp(-||x||_1)}{Z}, \text{ for } y \in \text{Supp}$$

$$x = \text{inv}(\text{scale}) @ (y - \text{loc}),$$

$$Z = |\text{det}(\text{scale})|,$$

where:

- loc is a vector in \(\mathbb{R}^k\),
- scale is a linear operator in \(\mathbb{R}^{k \times k}\), \(\text{cov} = \text{scale} @ \text{scale}.T\),
- \(Z\) denotes the normalization constant, and,
- \(||x||_1\) denotes the \(l_1\) norm of \(x\), \(\sum_i |x_i|\). The VectorExponential distribution is a member of the location-scale family, i.e., it can be constructed as,

\[X = (X_1, ..., X_k), \text{ each } X_i \sim \text{Exponential}(rate=1)\]

\[Y = (Y_1, ..., Y_k) = \text{scale} @ X + \text{loc}\]

About VectorExponential and Vector distributions in TensorFlow.

The VectorExponential is a non-standard distribution that has useful properties. The marginals \(Y_1, ..., Y_k\) are not Exponential random variables, due to the fact that the sum of Exponential random variables is not Exponential. Instead, \(Y\) is a vector whose components are linear combinations of Exponential random variables. Thus, \(Y\) lives in the vector space generated by vectors of Exponential distributions. This allows the user to decide the mean and covariance (by setting \text{loc} and \text{scale}), while preserving some properties of the Exponential distribution. In particular, the tails of \(Y_i\) will be (up to polynomial factors) exponentially decaying. To see this last statement, note that the pdf of \(Y_i\) is the convolution of the pdf of \(k\) independent Exponential random variables. One can then show by induction that distributions with exponential (up to polynomial factors) tails are closed under convolution.

The batch_shape is the broadcast shape between \text{loc} and \text{scale} arguments. The event_shape is given by last dimension of the matrix implied by \text{scale}. The last dimension of \text{loc} (if provided) must broadcast with this. Recall that \(\text{covariance} = 2 * \text{scale} @ \text{scale}.T\). Additional leading dimensions (if any) will index batches. If both \text{scale}_\text{diag} and \text{scale}_\text{identity_multiplier} are NULL, then \text{scale} is the Identity matrix.

Value

- a distribution instance.

See Also

- For usage examples see e.g. \text{tfd_sample()}, \text{tfd_log_prob()}, \text{tfd_mean()}.
- Other distributions: \text{tfd_autoregressive()}, \text{tfd_batch_reshape()}, \text{tfd_bernoulli()}, \text{tfd_beta_binomial()}, \text{tfd_beta()}, \text{tfd_binomial()}, \text{tfd_categorical()}, \text{tfd_cauchy()}, \text{tfd_chi2()}, \text{tfd_chi()}, \text{tfd_cholesky_lkj()}, \text{tfd_deterministic()}, \text{tfd_dirichlet_multinomial()}, \text{tfd_dirichlet()},
The vectorization of the Exponential distribution on $\mathbb{R}^k$

**Description**

The vector exponential distribution is defined over a subset of $\mathbb{R}^k$, and parameterized by a (batch of) length-$k$ loc vector and a (batch of) $k \times k$ scale matrix: covariance = scale @ scale.T, where @ denotes matrix-multiplication.

**Usage**

```r
tf_vector_exponential_linear_operator(
  loc = NULL,
  scale = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "VectorExponentialLinearOperator"
)
```

**Arguments**

- `loc` (Floating point tensor; the means of the distribution(s).)
- `scale` (Instance of LinearOperator with same dtype as loc and shape [B1, ..., Bb, k, k].)
- `validate_args` (Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.)
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
Mathematical Details The probability density function (pdf) is

\[
pdf(y; \text{loc}, \text{scale}) = \exp(-||x||_1) / Z, \text{ for } y \in S(\text{loc}, \text{scale}),
\]

\[
x = \text{inv}(\text{scale}) @ (y - \text{loc}),
\]

\[
Z = |\det(\text{scale})|,
\]

where:

- \text{loc} is a vector in \(\mathbb{R}^k\),
- \text{scale} is a linear operator in \(\mathbb{R}^{k \times k}\), \(\text{cov} = \text{scale} @ \text{scale.T}\),
- \(S = \{\text{loc} + \text{scale} @ x : x \in \mathbb{R}^k, x_1 > 0, ..., x_k > 0\}\), is an image of the positive half-space,
- \(||x||_1\) denotes the \(l_1\) norm of \(x\), \(\sum_i |x_i|\),
- \(Z\) denotes the normalization constant.

The VectorExponential distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X = (X_1, ..., X_k), \text{ each } X_i \sim \text{Exponential}(rate=1)
\]

\[
Y = (Y_1, ..., Y_k) = \text{scale} @ X + \text{loc}
\]

About VectorExponential and Vector distributions in TensorFlow.
The VectorExponential is a non-standard distribution that has useful properties. The marginals \(Y_1, ..., Y_k\) are not Exponential random variables, due to the fact that the sum of Exponential random variables is not Exponential. Instead, \(Y\) is a vector whose components are linear combinations of Exponential random variables. Thus, \(Y\) lives in the vector space generated by vectors of Exponential distributions. This allows the user to decide the mean and covariance (by setting \text{loc and scale}), while preserving some properties of the Exponential distribution. In particular, the tails of \(Y_i\) will be (up to polynomial factors) exponentially decaying. To see this last statement, note that the pdf of \(Y_i\) is the convolution of the pdf of \(k\) independent Exponential random variables. One can then show by induction that distributions with exponential (up to polynomial factors) tails are closed under convolution.

The batch_shape is the broadcast shape between loc and scale arguments. The event_shape is given by last dimension of the matrix implied by scale. The last dimension of loc (if provided) must broadcast with this. Recall that \(\text{covariance} = 2 \times \text{scale} @ \text{scale.T}\). Additional leading dimensions (if any) will index batches.

Value
a distribution instance.
See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

tfd_vector_laplace_diag

Arguments

loc: Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape [B1, ..., Bb, k] where b >= 0 and k is the event size.

scale_diag: Non-zero, floating-point Tensor representing a diagonal matrix added to scale. May have shape [B1, ..., Bb, k], b >= 0, and characterizes b-batches of k x k diagonal matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

cscale_identity_multiplier: Non-zero, floating-point Tensor representing a scaled-identity-matrix added to scale. May have shape [B1, ..., Bb], b >= 0, and characterizes b-batches of scaled k x k identity matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

validate_args: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

name: name prefixed to Ops created by this class.

Details

Mathematical Details The probability density function (pdf) is,

\[
pdf(x; \text{loc}, \text{scale}) = \exp(-||y||_1) / Z \\
y = \text{inv}(\text{scale}) @ (x - \text{loc}) \\
Z = 2**k |\det(\text{scale})|
\]

where:

- \text{loc} is a vector in \(\mathbb{R}^k\),
- \text{scale} is a linear operator in \(\mathbb{R}^{k \times k}\), \(\text{cov} = \text{scale} @ \text{scale}.\top\),
- \(Z\) denotes the normalization constant, and,
- \(||y||_1\) denotes the \(l1\) norm of \(y\), \('\sum_i |y_i|'\).

A (non-batch) scale matrix is:

\[
\text{scale} = \text{diag}(\text{scale}_\text{diag} + \text{scale}_\text{identity}_\text{multiplier} * \text{ones}(k))
\]

where:

- \text{scale}_\text{diag}.\text{shape} = [k], and,
- \text{scale}_\text{identity}_\text{multiplier}.\text{shape} = []. Additional leading dimensions (if any) will index batches. If both \text{scale}_\text{diag} and \text{scale}_\text{identity}_\text{multiplier} are NULL, then scale is the Identity matrix.
The VectorLaplace is a non-standard distribution that has useful properties. The marginals $Y_1, ..., Y_k$ are not Laplace random variables, due to the fact that the sum of Laplace random variables is not Laplace. Instead, $Y$ is a vector whose components are linear combinations of Laplace random variables. Thus, $Y$ lives in the vector space generated by vectors of Laplace distributions. This allows the user to decide the mean and covariance (by setting loc and scale), while preserving some properties of the Laplace distribution. In particular, the tails of $Y_i$ will be (up to polynomial factors) exponentially decaying. To see this last statement, note that the pdf of $Y_i$ is the convolution of the pdf of $k$ independent Laplace random variables. One can then show by induction that distributions with exponential (up to polynomial factors) tails are closed under convolution.

Value

A distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


The vectorization of the Laplace distribution on $\mathbb{R}^k$.

Description

The vector laplace distribution is defined over $\mathbb{R}^k$, and parameterized by a (batch of) length-$k$ loc vector (the means) and a (batch of) $k \times k$ scale matrix: covariance = $2 \times$ scale @ scale.T, where @ denotes matrix-multiplication.
Usage

tfd_vector_laplace_linear_operator(
    loc = NULL,
    scale = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "VectorLaplaceLinearOperator"
)

Arguments

loc Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape [B1, ..., Bb, k] where b >= 0 and k is the event size.
scale Instance of LinearOperator with same dtype as loc and shape [B1, ..., Bb, k, k].
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name name prefixed to Ops created by this class.

Details

Mathematical Details The probability density function (pdf) is,

\[
pdf(x; \text{loc}, \text{scale}) = \frac{\exp(-||y||_1)}{Z},
\]

\[
y = \text{inv}(\text{scale}) @ (x - \text{loc}),
\]

\[
Z = 2^{**k} |\text{det}(\text{scale})|,
\]

where:

- \text{loc} is a vector in \(\mathbb{R}^k\),
- \text{scale} is a linear operator in \(\mathbb{R}^{k x k}\), \(\text{cov} = \text{scale} @ \text{scale.T}\),
- \(Z\) denotes the normalization constant, and,
- \(||y||_1\) denotes the l1 norm of \(y\), \(\sum_i |y_i|\).

The VectorLaplace distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X = (X_1, \ldots, X_k), \text{ each } X_i \sim \text{Laplace(loc=0, scale=1)}
\]

\[
Y = (Y_1, \ldots, Y_k) = \text{scale} @ X + \text{loc}
\]

About VectorLaplace and Vector distributions in TensorFlow

The VectorLaplace is a non-standard distribution that has useful properties. The marginals \(Y_1, ..., Y_k\) are not Laplace random variables, due to the fact that the sum of Laplace random variables is
The (diagonal) SinhArcsinh transformation of a distribution on $\mathbb{R}^{*k}$
Description

This distribution models a random vector $Y = (Y_1, ..., Y_k)$, making use of a SinhArcsinh transformation (which has adjustable tailweight and skew), a rescaling, and a shift. The SinhArcsinh transformation of the Normal is described in great depth in Sinh-arcsinh distributions. Here we use a slightly different parameterization, in terms of tailweight and skewness. Additionally we allow for distributions other than Normal, and control over scale as well as a "shift" parameter loc.

Usage

tfd_vector_sinh_arcsinh_diag(
    loc = NULL,
    scale_diag = NULL,
    scale_identity_multiplier = NULL,
    skewness = NULL,
    tailweight = NULL,
    distribution = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "VectorSinhArcsinhDiag"
)

Arguments

loc  Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape \([B_1, ..., B_b, k]\) where \(b \geq 0\) and \(k\) is the event size.

scale_diag  Non-zero, floating-point Tensor representing a diagonal matrix added to scale. May have shape \([B_1, ..., B_b, k]\), \(b \geq 0\), and characterizes \(b\)-batches of \(k \times k\) diagonal matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

scale_identity_multiplier  Non-zero, floating-point Tensor representing a scale-identity-matrix added to scale. May have shape \([B_1, ..., B_b]\), \(b \geq 0\), and characterizes \(b\)-batches of \(k \times k\) identity matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

skewness  Skewness parameter. floating-point Tensor with shape broadcastable with event_shape.

tailweight  Tailweight parameter. floating-point Tensor with shape broadcastable with event_shape.

distribution tf$distributions$Distribution-like instance. Distribution from which k iid samples are used as input to transformation $F$. Default is tfd_normal(loc = 0, scale = 1). Must be a scalar-batch, scalar-event distribution. Typically distribution$reparameterization_type = FULLY_REPARAMETERIZED or it is a function of non-trainable parameters. WARNING: If you backprop through a VectorSinhArcsinhDiag sample and distribution is not FULLY_REPARAMETERIZED yet is a function of trainable variables, then the gradient will be incorrect!

validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
Mathematical Details
Given iid random vector \( Z = (Z_1, \ldots, Z_k) \), we define the VectorSinhArcsinhDiag transformation of \( Z, Y \), parameterized by \( (\text{loc}, \text{scale}, \text{skewness}, \text{tailweight}) \), via the relation (with \( \oplus \) denoting matrix multiplication):

\[
Y := \text{loc} + \text{scale} \oplus F(Z) \star \left( \frac{2}{F_0(2)} \right)
\]

\[
F(Z) := \text{Sinh} \left( \text{Arcsinh}(Z) + \text{skewness} \right) \star \text{tailweight}
\]

\[
F_0(Z) := \text{Sinh} \left( \text{Arcsinh}(Z) \right) \star \text{tailweight}
\]

This distribution is similar to the location-scale transformation \( L(Z) := \text{loc} + \text{scale} \oplus Z \) in the following ways:

- If \( \text{skewness} = 0 \) and \( \text{tailweight} = 1 \) (the defaults), \( F(Z) = Z \), and then \( Y = L(Z) \) exactly.
- \( \text{loc} \) is used in both to shift the result by a constant factor.
- The multiplication of \( \text{scale} \) by \( \frac{2}{F_0(2)} \) ensures that if \( \text{skewness} = 0 \) \( P[Y - \text{loc} <= 2 \star \text{scale}] = P[L(Z) - \text{loc} <= 2 \star \text{scale}] \). Thus it can be said that the weights in the tails of \( Y \) and \( L(Z) \) beyond \( \text{loc} + 2 \star \text{scale} \) are the same. This distribution is different than \( \text{loc} + \text{scale} \oplus Z \) due to the reshaping done by \( F \):
  - Positive (negative) skewness leads to positive (negative) skew.
  - Positive skew means, the mode of \( F(Z) \) is "tilted" to the right.
  - Positive skew means positive values of \( F(Z) \) become more likely, and negative values become less likely.
  - Larger (smaller) \( \text{tailweight} \) leads to fatter (thinner) tails.
  - Fatter tails mean larger values of \( |F(Z)| \) become more likely.
  - \( \text{tailweight} < 1 \) leads to a distribution that is "flat" around \( Y = \text{loc} \), and a very steep drop-off in the tails.
  - \( \text{tailweight} > 1 \) leads to a distribution more peaked at the mode with heavier tails. To see the argument about the tails, note that for \( |Z| >> 1 \) and \( |Z| >> (\text{skewness} \star \text{tailweight})^{\text{tailweight}} \), we have \( Y \approx 0.5 Z^{\star \text{tailweight}} e^{(\text{sign}(Z) \ \text{skewness} \star \text{tailweight})} \). To see the argument regarding multiplying \( \text{scale} \) by \( \frac{2}{F_0(2)} \),

\[
P[(Y - \text{loc}) / \text{scale} <= 2] = P[F(Z) \star (2 / F_0(2)) <= 2]
\]

\[
= P[F(Z) <= F_0(2)]
\]

\[
= P[Z <= 2] \ (\text{if} \ F = F_0).
\]
See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Arguments

loc  Floating point tensor, the circular means of the distribution(s).
concentration  Floating point tensor, the level of concentration of the distribution(s) around loc.
               Must take non-negative values. concentration = 0 defines a Uniform distribution,
               while concentration = +inf indicates a Deterministic distribution at loc.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for
               validity despite possibly degrading runtime performance. When FALSE invalid
               inputs may silently render incorrect outputs. Default value: FALSE.
allow_nans_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
               the value NaN to indicate the result is undefined. When FALSE, an exception is
               raised if one or more of the statistic's batch members are undefined.
name  name prefixed to Ops created by this class.

Details

The von Mises distribution is a special case of von Mises-Fisher distribution for n=2. However,
the TFP's VonMisesFisher implementation represents the samples and location as (x, y) points on a
circle, while VonMises represents them as scalar angles.

Mathematical details The probability density function (pdf) of this distribution is,

\[
    \text{pdf}(x; \text{loc}, \text{concentration}) = \exp(\text{concentration} \cos(x - \text{loc})) / Z \\
    Z = 2 \times \pi \times I_0(\text{concentration})
\]

where:

- \(I_0(\text{concentration})\) is the modified Bessel function of order zero;
- \text{loc}  the circular mean of the distribution, a scalar. It can take arbitrary values, but it is 2pi-
  periodic: \text{loc} and \text{loc} + 2\pi result in the same distribution.
- \text{concentration} \geq 0 parameter is the concentration parameter. When \text{concentration} = 0,
  this distribution becomes a Uniform distribution on \([-\pi, \pi)\).

The parameters \text{loc} and \text{concentration} must be shaped in a way that supports broadcasting (e.g. \text{loc} + \text{concentration} is a valid operation).

Value

a distribution instance.

See Also

For usage examples see e.g. \text{tfd_sample()}, \text{tfd_log_prob()}, \text{tfd_mean()}.

Other distributions: \text{tfd_autoregressive()}, \text{tfd_batch_reshape()}, \text{tfd_bernoulli()}, \text{tfd_beta_binomial()},
\text{tfd_beta()}, \text{tfd_binomial()}, \text{tfd_categorical()}, \text{tfd_cauchy()}, \text{tfd_chi2()}, \text{tfd_chi()},
\text{tfd_cholesky_lkj()}, \text{tfd_deterministic()}, \text{tfd_dirichlet_multinomial()}, \text{tfd_dirichlet()},
\text{tfd_empirical()}, \text{tfd_exponential()}, \text{tfd_gamma_gamma()}, \text{tfd_gamma()}, \text{tfd_gaussian_process_regression_model()},
\text{tfd_gaussian_process()}, \text{tfd_geometric()}, \text{tfd_gumbel()}, \text{tfd_half_cauchy()}, \text{tfd_half_normal()}.\n
The von Mises-Fisher distribution is a directional distribution over vectors on the unit hypersphere $S^{n-1}$ embedded in $n$ dimensions ($\mathbb{R}^n$).

**Usage**

```r
tfd_von_mises_fisher(
  mean_direction,
  concentration,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "VonMisesFisher"
)
```

**Arguments**

- **mean_direction**
  - Floating-point Tensor with shape [B1, ..., Bn, D]. A unit vector indicating the mode of the distribution, or the unit-normalized direction of the mean. (This is not in general the mean of the distribution; the mean is not generally in the support of the distribution.) NOTE: D is currently restricted to $\leq 5$.

- **concentration**
  - Floating-point Tensor having batch shape [B1, ..., Bn] broadcastable with mean_direction. The level of concentration of samples around the mean_direction. concentration=0 indicates a uniform distribution over the unit hypersphere, and concentration=+inf indicates a Deterministic distribution (delta function) at mean_direction.

- **validate_args**
  - Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
Mathematical details The probability density function (pdf) is,
\[
pdf(x; \mu, \kappa) = C(\kappa) \exp(\kappa \cdot \mu^T x)
\]
where,
\[
C(\kappa) = (2 \pi)^{-n/2} \kappa^{n/2-1} / I_{n/2-1}(\kappa),
\]
\(I_v(z)\) being the modified Bessel function of the first kind of order \(v\)
where:
- mean_direction = \(\mu\); a unit vector in \(\mathbb{R}^k\),
- concentration = \(\kappa\); scalar real \(\geq 0\), concentration of samples around mean_direction,
  where 0 pertains to the uniform distribution on the hypersphere, and \(\infty\) indicates a delta
  function at mean_direction.

NOTE: Currently only n in 2, 3, 4, 5 are supported. For n=5 some numerical instability can occur
for low concentrations (<.01).

Value

a distribution instance.

See Also
For usage examples see e.g. \(\text{tfd_sample}\), \(\text{tfd_log_prob}\), \(\text{tfd_mean}\).

Other distributions: \(\text{tfd_autoregressive}\), \(\text{tfd_batch_reshape}\), \(\text{tfd_bernoulli}\), \(\text{tfd_beta_binomial}\),
\(\text{tfd_beta}\), \(\text{tfd_binomial}\), \(\text{tfd_categorical}\), \(\text{tfd_cauchy}\), \(\text{tfd_chi2}\), \(\text{tfd_chi}\),
\(\text{tfd_cholesky_lkj}\), \(\text{tfd_deterministic}\), \(\text{tfd_dirichlet_multinomial}\), \(\text{tfd_dirichlet}\),
\(\text{tfd_empirical}\), \(\text{tfd_exponential}\), \(\text{tfd_gamma_gamma}\), \(\text{tfd_gaussian_process_regression_model}\),
\(\text{tfd_gaussian_process}\), \(\text{tfd_geometric}\), \(\text{tfd_gumbel}\), \(\text{tfd_half_cauchy}\), \(\text{tfd_half_normal}\),
\(\text{tfd_hidden_markov_model}\), \(\text{tfd_horseshoe}\), \(\text{tfd_independent}\), \(\text{tfd_inverse_gaussian}\),
\(\text{tfd_inverse_gaussian}\), \(\text{tfd_joint_distribution_named}\), \(\text{tfd_joint_distribution_sequential}\),
\(\text{tfd_kumaraswamy}\), \(\text{tfd_laplace}\), \(\text{tfd_linear_gaussian_state_space_model}\), \(\text{tfd_lkj}\),
\(\text{tfd_log_normal}\), \(\text{tfd_logistic}\), \(\text{tfd_mixture_same_family}\), \(\text{tfd_mixture}\), \(\text{tfd_multinomial}\),
\(\text{tfd_multivariate_normal_diag_plus_low_rank}\), \(\text{tfd_multivariate_normal_diag}\), \(\text{tfd_multivariate_normal}\),
\(\text{tfd_multivariate_normal_linear_operator}\), \(\text{tfd_multivariate_normal_tril}\), \(\text{tfd_multivariate_student_t}\),
\(\text{tfd_negative_binomial}\), \(\text{tfd_normal}\), \(\text{tfd_one_hot_categorical}\), \(\text{tfd_pareto}\), \(\text{tfd_pixel_cnn}\),
\(\text{tfd_poisson_log_normal_quadrature_compound}\), \(\text{tfd_poisson}\), \(\text{tfd_probit_bernoulli}\),
\(\text{tfd_quantized}\), \(\text{tfd_relaxed_bernoulli}\), \(\text{tfd_relaxed_one_hot_categorical}\), \(\text{tfd_sample_distribution}\),
\(\text{tfd_sinh_arcsinh}\), \(\text{tfd_student_t_process}\), \(\text{tfd_student_t}\), \(\text{tfd_transformed_distribution}\),
\(\text{tfd_triangular}\), \(\text{tfd_truncated_normal}\), \(\text{tfd_uniform}\), \(\text{tfd_variational_gaussian_process}\),
\(\text{tfd_vector_diffeomixture}\), \(\text{tfd_vector_exponential_diag}\), \(\text{tfd_vector_exponential_linear_operator}\),
The matrix Wishart distribution on positive definite matrices

Description

This distribution is defined by a scalar number of degrees of freedom `df` and an instance of LinearOperator, which provides matrix-free access to a symmetric positive definite operator, which defines the scale matrix.

Usage

tfd_wishart(
  df,
  scale = NULL,
  scale_tril = NULL,
  input_output_cholesky = FALSE,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Wishart"
)

Arguments

- **df**: float or double tensor, the degrees of freedom of the distribution(s). `df` must be greater than or equal to `k`.
- **scale**: float or double Tensor. The symmetric positive definite scale matrix of the distribution. Exactly one of `scale` and `scale_tril` must be passed.
- **scale_tril**: float or double Tensor. The Cholesky factorization of the symmetric positive definite scale matrix of the distribution. Exactly one of `scale` and `scale_tril` must be passed.
- **input_output_cholesky**: Logical. If TRUE, functions whose input or output have the semantics of samples assume inputs are in Cholesky form and return outputs in Cholesky form. In particular, if this flag is TRUE, input to `log_prob` is presumed of Cholesky form and output from sample, mean, and mode are of Cholesky form. Setting this argument to TRUE is purely a computational optimization and does not change the underlying distribution; for instance, mean returns the Cholesky of the mean, not the mean of Cholesky factors. The variance and `stddev` methods are unaffected by this flag. Default value: FALSE (i.e., input/output does not have Cholesky semantics).
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
Mathematical Details
The probability density function (pdf) is,

\[
pdf(X; \text{df}, \text{scale}) = \det(X)^{0.5 (\text{df}-k-1)} \exp(-0.5 \text{tr}[\text{inv}(\text{scale}) X]) / Z
\]

\[
Z = 2^{0.5 \text{df} k} |\det(\text{scale})|^{0.5 \text{df}} \Gamma_k(0.5 \text{df})
\]

where:
• \(\text{df} \geq k\) denotes the degrees of freedom,
• \(\text{scale}\) is a symmetric, positive definite, \(k \times k\) matrix,
• \(Z\) is the normalizing constant, and,
• \(\Gamma_k\) is the multivariate Gamma function.

Value
a distribution instance.

See Also
For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`,
`tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chi()`,
`tfd_cholesky_lkj()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`,
`tfd_empirical()`, `tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`,
`tfd_gaussian_process()`, `tfd_geometric()`, `tfd_half_cauchy()`, `tfd_half_normal()`,
`tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gaussian()`,
`tfd_inverse_gaussian()`, `tfd_joystick_distribution_named()`, `tfd_joystick_distribution_sequential()`,
`tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`,
`tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multinomial()`,
`tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`, `tfd_multivariate_normal_linear_operator()`,
`tfd_multivariate_normal_tril()`, `tfd_multivariate_student_t()`,
`tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`,
`tfd_poisson_log_normal_quadrature_compound()`, `tfd_poisson()`, `tfd_probit_bernoulli()`,
`tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`,
`tfd_sinh_arcsinh()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`,
`tfd_triangular()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_variational_gaussian_process()`,
`tfd_vector_diffeomixture()`, `tfd_vector_exponential_diag()`, `tfd_vector_exponential_linear_operator()`,
`tfd_vector_laplace_diag()`, `tfd_vector_laplace_linear_operator()`, `tfd_vector_sin_arcsin()`,
`tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_wishart_linear_operator()`, `tfd_wishart_tril()`,
`tfd_zipf()`
tfd_wishart_linear_operator

The matrix Wishart distribution on positive definite matrices

Description

This distribution is defined by a scalar number of degrees of freedom df and an instance of LinearOperator, which provides matrix-free access to a symmetric positive definite operator, which defines the scale matrix.

Usage

```python
tfd_wishart_linear_operator(
  df,
  scale,
  input_output_cholesky = FALSE,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "WishartLinearOperator"
)
```

Arguments

- **df**: float or double tensor, the degrees of freedom of the distribution(s). df must be greater than or equal to k.
- **scale**: float or double instance of LinearOperator.
- **input_output_cholesky**: Logical. If TRUE, functions whose input or output have the semantics of samples assume inputs are in Cholesky form and return outputs in Cholesky form. In particular, if this flag is TRUE, input to log_prob is presumed of Cholesky form and output from sample, mean, and mode are of Cholesky form. Setting this argument to TRUE is purely a computational optimization and does not change the underlying distribution; for instance, mean returns the Cholesky of the mean, not the mean of Cholesky factors. The variance and stddev methods are unaffected by this flag. Default value: FALSE (i.e., input/output does not have Cholesky semantics).
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.
Mathematical Details

The probability density function (pdf) is,

$$pdf(X; df, \text{scale}) = det(X)^{0.5 (df-k-1)} \exp(-0.5 \ tr[\text{inv}(\text{scale}) X]) / Z$$

$$Z = 2^{0.5 df k} |det(\text{scale})|^{0.5 df} \text{Gamma}_k(0.5 df)$$

where:

- $df \geq k$ denotes the degrees of freedom,
- $\text{scale}$ is a symmetric, positive definite, $k \times k$ matrix,
- $Z$ is the normalizing constant, and,
- $\text{Gamma}_k$ is the multivariate Gamma function.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

The matrix Wishart distribution parameterized with Cholesky factors.

Description

This distribution is defined by a scalar degrees of freedom df and a scale matrix, expressed as a lower triangular Cholesky factor.

Usage

tfd_wishart_tri_l(
  df,
  scale_tril,
  input_output_cholesky = FALSE,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "WishartTriL"
)

Arguments

df float or double tensor, the degrees of freedom of the distribution(s). df must be greater than or equal to k.

scale_tril float or double Tensor. The Cholesky factorization of the symmetric positive definite scale matrix of the distribution.

input_output_cholesky Logical. If TRUE, functions whose input or output have the semantics of samples assume inputs are in Cholesky form and return outputs in Cholesky form. In particular, if this flag is TRUE, input to log_prob is presumed of Cholesky form and output from sample, mean, and mode are of Cholesky form. Setting this argument to TRUE is purely a computational optimization and does not change the underlying distribution; for instance, mean returns the Cholesky of the mean, not the mean of Cholesky factors. The variance and stddev methods are unaffected by this flag. Default value: FALSE (i.e., input/output does not have Cholesky semantics).

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.
Details

Mathematical Details

The probability density function (pdf) is,

\[
\text{pdf}(X; \text{df}, \text{scale}) = \det(X)^{0.5(\text{df}-k-1)} \exp(-0.5 \text{tr}[\text{inv(scale)} X]) / Z
\]

\[
Z = 2^{0.5 \text{df} k} |\det(\text{scale})|^{0.5 \text{df}} \Gamma_k(0.5 \text{df})
\]

where:

- \(\text{df} \geq k\) denotes the degrees of freedom,
- \(\text{scale}\) is a symmetric, positive definite, \(k \times k\) matrix,
- \(Z\) is the normalizing constant, and,
- \(\Gamma_k\) is the multivariate Gamma function.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chisq()`, `tfd_cholesky_lkj()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gaussian()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixtured_same_family()`, `tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`, `tfd_multivariate_normal_linear_operator()`, `tfd_multivariate_normal_tri_l()`, `tfd_multivariate_student_t()`, `tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature_compound()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`, `tfd_triangular()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_wishart_linear_operator()`, `tfd_wishart()`, `tfd_zipf()`
The Zipf distribution is parameterized by a power parameter.

Usage

tfd_zipf(
  power,
  dtype = tf$int32,
  interpolate_nondiscrete = TRUE,
  sample_maximum_iterations = 100,
  validate_args = FALSE,
  allow_nan_stats = FALSE,
  name = "Zipf"
)

Arguments

- **power**: Float like Tensor representing the power parameter. Must be strictly greater than 1.
- **dtype**: The dtype of Tensor returned by sample. Default value: tf$int32.
- **interpolate_nondiscrete**: Logical. When FALSE, log_prob returns -inf (and prob returns 0) for non-integer inputs. When TRUE, log_prob evaluates the continuous function -power log(k) - log(zeta(power)), which matches the Zipf pmf at integer arguments k (note that this function is not itself a normalized probability log-density). Default value: TRUE.
- **sample_maximum_iterations**: Maximum number of iterations of allowable iterations in sample. When validate_args=TRUE, samples which fail to reach convergence (subject to this cap) are masked out with self$dtype$min or nan depending on self$dtype$is_integer. Default value: 100.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Default value: FALSE.
- **name**: name prefixed to Ops created by this class.

Details

Mathematical Details The probability mass function (pmf) is,
pmf(k; alpha, k >= 0) = (k^(-alpha)) / Z
Z = zeta(alpha).

where power = alpha and Z is the normalization constant. zeta is the Riemann zeta function. Note that gradients with respect to the power parameter are not supported in the current implementation.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tril(), tfd_mixture(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature_compound(), tfd_poisson(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_normal(), tfd_uniform(), tfd_vonCDFspecial(); tfd_von_mises_fisher(), tfd_von_mises(), tfd_wishart_linear_operator(), tfd_wishart_tril(), tfd_wishart()

**tfp**

*Handle to the tensorflow_probability module*

**Description**

Handle to the tensorflow_probability module

**Usage**

tfp

**Format**

An object of class python.builtin.module (inherits from python.builtin.object) of length 5.
tfp_version

Value

Module(tensorflow_probability)

Description

TensorFlow Probability Version

Usage

tfp_version()

Value

the Python TFP version

vi_amari_alpha

The Amari-alpha Csiszar-function in log-space

Description

A Csiszar-function is a member of \( F = \{ f : \mathbb{R}_+ \to \mathbb{R} : f \text{ convex } \} \).

Usage

vi_amari_alpha(logu, alpha = 1, self_normalized = FALSE, name = NULL)

Arguments

logu float-like Tensor representing \( \log(u) \) from above.
alpha float-like scalar.
self_normalized logical indicating whether \( f'(u=1)=0 \). When \( f'(u=1)=0 \) the implied Csiszar f-Divergence remains non-negative even when \( p, q \) are unnormalized measures.
name name prefixed to Ops created by this function.
**vi_arithmetic_geometric**

Details

When `self_normalized = TRUE`, the Amari-alpha Csiszar-function is:

\[
\begin{align*}
  f(u) &= \begin{cases} 
    -\log(u) + (u - 1), & \text{alpha} = 0 \\
    u \log(u) - (u - 1), & \text{alpha} = 1 \\
    \left((u^\text{alpha} - 1) - \alpha (u - 1) / (\alpha (\alpha - 1))\right), & \text{otherwise}
  \end{cases}
\end{align*}
\]

When `self_normalized = FALSE` the `(u-1)` terms are omitted.

Warning: when `alpha != 0` and/or `self_normalized = True` this function makes non-log-space calculations and may therefore be numerically unstable for `|\log u| >> 0`.

Value

`amari_alpha_of_u` float-like Tensor of the Csiszar-function evaluated at `u = \exp(\log u)`.

References


See Also

Other vi-functions: `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_jensen_shannon()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_log1p_abs()`, `vi_modified_gan()`, `vi_monte_carlo_variational_loss()`, `vi_pearson()`, `vi_squared_hellinger()`, `vi_symmetrized_csiszar_function()`

---

**vi_arithmetic_geometric**

*The Arithmetic-Geometric Csiszar-function in log-space*

Description

A Csiszar-function is a member of \( F = \{ f:R_+ \to R : f \text{ convex } \} \).

Usage

`vi_arithmetic_geometric(logu, self_normalized = FALSE, name = NULL)`

Arguments

- `logu`: float-like Tensor representing \( \log(u) \) from above.
- `self_normalized`: logical indicating whether \( f(u=1)=0 \). When \( f(u=1)=0 \) the implied Csiszar f-Divergence remains non-negative even when \( p, q \) are unnormalized measures.
- `name`: name prefixed to Ops created by this function.
vi_chi_square

Details

When `self_normalized = True` the Arithmetic-Geometric Csiszar-function is:

\[ f(u) = (1 + u) \log \left( \frac{1 + u}{\sqrt{u}} \right) - (1 + u) \log(2) \]

When `self_normalized = False` the \((1 + u) \log(2)\) term is omitted.

Observe that as an f-Divergence, this Csiszar-function implies:

\[ D_f[p, q] = KL[m, p] + KL[m, q] \]
\[ m(x) = 0.5 \ p(x) + 0.5 \ q(x) \]

In a sense, this divergence is the "reverse" of the Jensen-Shannon f-Divergence. This Csiszar-function induces a symmetric f-Divergence, i.e., \( D_f[p, q] = D_f[q, p] \).

Warning: when `self_normalized = True` this function makes non-log-space calculations and may therefore be numerically unstable for \(|\log u| \gg 0\).

Value

`arithmetic_geometric_of_u`: float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

See Also


---

vi_chi_square The chi-square Csiszar-function in log-space

Description

A Csiszar-function is a member of \( F = \{ f: \mathbb{R}_+ \to \mathbb{R} : f \text{ convex} \} \).

Usage

`vi_chi_square(logu, name = NULL)`

Arguments

- `logu`: float-like Tensor representing \( \log(u) \) from above.
- `name`: name prefixed to Ops created by this function.
The Chi-square Csiszar-function is:

\[ f(u) = u^2 - 1 \]

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( \log u \gg 0 \).

Value

\texttt{chi\_square\_of\_u}: float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

See Also

Other vi-functions: \texttt{vi\_amari\_alpha()}, \texttt{vi\_arithmetic\_geometric()}, \texttt{vi\_csiszar\_vimco()}, \texttt{vi\_dual\_csiszar\_function()}, \texttt{vi\_fit\_surrogate\_posterior()}, \texttt{vi\_jeffreys()}, \texttt{vi\_jensen\_shannon()}, \texttt{vi\_kl\_forward()}, \texttt{vi\_kl\_reverse()}, \texttt{vi\_log1p\_abs()}, \texttt{vi\_modified\_gan()}, \texttt{vi\_monte\_carlo\_variational\_loss()}, \texttt{vi\_pearson()}, \texttt{vi\_squared\_hellinger()}, \texttt{vi\_symmetrized\_csiszar\_function()}

---

\texttt{vi\_csiszar\_vimco} \hspace{1cm} Use VIMCO to lower the variance of the gradient of \( \text{csiszar\_function(Avg(\log u))} \)

Description

This function generalizes VIMCO (Mnih and Rezende, 2016) to Csiszar f-Divergences.

Usage

\begin{verbatim}
vi_csiszar_vimco(
    f, p_log_prob, q,
    num_draws, num_batch_draws = 1,
    seed = NULL,
    name = NULL
)
\end{verbatim}

Arguments

\begin{itemize}
    \item \texttt{f} function representing a Csiszar-function in log-space.
    \item \texttt{p\_log\_prob} function representing the natural-log of the probability under distribution \( p \). (In variational inference \( p \) is the joint distribution.)
    \item \texttt{q} \texttt{tfd\$Distribution}-like instance; must implement: \texttt{sample(n, seed)}, and \texttt{log\_prob(x)}. (In variational inference \( q \) is the approximate posterior distribution.)
\end{itemize}
num_draws  Integer scalar number of draws used to approximate the f-Divergence expectation.

num_batch_draws  Integer scalar number of draws used to approximate the f-Divergence expectation.

seed  integer seed for q.sample.

name  String prefixed to Ops created by this function.

Details

Note: if q.reparameterization_type = tfd.FULLY_REPARAMETERIZED, consider using monte_carlo_csiszar_f_divergence.

The VIMCO loss is:

vimco = f(Avg(logu[i] : i=0,...,m-1))

where,

logu[i] = log( p(x, h[i]) / q(h[i] | x) )

h[i] iid~ q(H | x)

Interestingly, the VIMCO gradient is not the naive gradient of vimco. Rather, it is characterized by:

grad[vimco] - variance_reducing_term

where,

variance_reducing_term = Sum{ grad[log q(h[i] | x)] * (vimco - f(log Avg{h[j;i] : j=0,...,m-1})) / Var(h[i]) : i=0, ..., m-1 }

h[j;i] = u[j] for j!=i, GeometricAverage{ u[k] : k!=i} for j==i

(We omitted stop_gradient for brevity. See implementation for more details.) The Avg{h[j;i] : j} term is a kind of "swap-out average" where the i-th element has been replaced by the leave-i-out Geometric-average.

This implementation prefers numerical precision over efficiency, i.e., O(num_draws * num_batch_draws * prod(batch_shape) * prod(event_shape)). (The constant may be fairly large, perhaps around 12.)

Value

vimco The Csiszar f-Divergence generalized VIMCO objective

References


See Also

Other vi-functions: vi_amari_alpha(), vi_arithmetic_geometric(), vi_chi_square(), vi_dual_csiszar_function(), vi_fit_surrogate_posterior(), vi_jeffreys(), vi_jensen_shannon(), vi_kl_forward(), vi_kl_reverse(), vi_log1p_abs(), vi_modified_gan(), vi_monte_carlo_variational_loss(), vi_pearson(), vi_squared_hellinger(), vi_symmetrized_csiszar_function()
vi_dual_csiszar_function

Calculates the dual Csiszar-function in log-space

Description

A Csiszar-function is a member of \( F = \{ f:R_+ \rightarrow R : f \text{ convex} \} \).

Usage

\[
\text{vi_dual_csiszar_function}(\text{logu}, \text{csiszar_function}, \text{name} = \text{NULL})
\]

Arguments

- \text{logu} float-like Tensor representing \( \log(u) \) from above.
- \text{csiszar_function} function representing a Csiszar-function over log-domain.
- \text{name} name prefixed to Ops created by this function.

Details

The Csiszar-dual is defined as:

\[
f^*(u) = u \ f(1 / u)
\]

where \( f \) is some other Csiszar-function. For example, the dual of \( \text{kl_reverse} \) is \( \text{kl_forward} \), i.e.,

\[
f(u) = -\log(u)
\]

\[
f^*(u) = u \ f(1 / u) = -u \ \log(1 / u) = u \ \log(u)
\]

The dual of the dual is the original function:

\[
f^{**}(u) = (u \ f(1/u))^*(u) = u \ (1/u) \ f(1/(1/u)) = f(u)
\]

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( \text{logu} >> 0 \).

Value

\text{dual_f_of_u} float-like Tensor of the result of calculating the dual of \( f \) at \( u = \exp(\text{logu}) \).

See Also

Other vi-functions: \text{vi_amari_alpha()}, \text{vi_arithmetic_geometric()}, \text{vi_chi_square()}, \text{vi_csiszar_vimco()}, \text{vi_fit_surrogate_posterior()}, \text{vi_jeffreys()}, \text{vi_jensen_shannon()}, \text{vi_kl_forward()}, \text{vi_kl_reverse()}, \text{vi_log1p_abs()}, \text{vi_modified_gan()}, \text{vi_monte_carlo_variational_loss()}, \text{vi_pearson()}, \text{vi_squared_hellinger()}, \text{vi_symmetrized_csiszar_function()}
vi_fit_surrogate_posterior

Fit a surrogate posterior to a target (unnormalized) log density

Description

The default behavior constructs and minimizes the negative variational evidence lower bound (ELBO),
given by

\[
\text{q_samples} \leftarrow \text{surrogate_posterior}\$\text{sample}(\text{num_draws})
\]
\[
\text{elbo_loss} \leftarrow -\text{tf}\$\text{reduce}\_\text{mean}(\text{target_log_prob_fn}(\text{q_samples}) - \text{surrogate_posterior}\$\text{log_prob}(\text{q_samples}))
\]

Usage

\[
\text{vi_fit_surrogate_posterior}(\text{\text{target_log_prob_fn}}, \text{\text{surrogate_posterior}}, \text{\text{optimizer}}, \text{\text{num_steps}}, \text{\text{trace_fn}} = \text{\text{function}}(\text{\text{loss}}, \text{\text{grads}}, \text{\text{variables}}) \text{\text{loss}}, \text{\text{variational_loss_fn}} = \text{\text{NULL}}, \text{\text{sample_size}} = \text{\text{1}}, \text{\text{trainable_variables}} = \text{\text{NULL}}, \text{\text{seed}} = \text{\text{NULL}}, \text{\text{name}} = \text{\text{"fit_surrogate_posterior"}})
\]

Arguments

\text{target_log_prob_fn}

function that takes a set of Tensor arguments and returns a Tensor log-density.
Given \text{q_sample} \leftarrow \text{surrogate_posterior}\$\text{sample}(\text{sample_size}), this will be
(in Python) called as \text{target_log_prob_fn}(\text{q_sample}) if \text{q_sample} is a list or
a tuple, \text{target_log_prob_fn}(**\text{q_sample}) if \text{q_sample} is a dictionary, or \text{target_log_prob_fn}(\text{q_sample})
if \text{q_sample} is a Tensor. It should support batched evaluation, i.e., should return
a result of shape [\text{sample_size}].

\text{surrogate_posterior}

A \text{tfp}\$\text{distributions}\$\text{Distribution} instance defining a variational posterior
(could be a \text{tfp}\$\text{distributions}\$\text{JointDistribution}). Crucially, the distribution's
\text{log_prob} and (if reparameterized) \text{sample} methods must directly invoke
all ops that generate gradients to the underlying variables. One way to ensure
this is to use \text{tfp}\$\text{util}\$\text{DeferredTensor} to represent any parameters defined as
transformations of unconstrained variables, so that the transformations execute
at runtime instead of at distribution creation.

\text{optimizer}

Optimizer instance to use. This may be a TF1-style \text{tf}\$\text{train}\$\text{Optimizer}, TF2-
style \text{tf}\$\text{optimizers}\$\text{Optimizer}, or any Python-compatible object that imple-
ments \text{optimizer}\$\text{apply}\_\text{gradients}((\text{grads_and_vars}).
num_steps integer number of steps to run the optimizer.

trace_fn function with signature state = trace_fn(loss,grads,variables), where state may be a Tensor or nested structure of Tensors. The state values are accumulated (by tf$scan) and returned. The default trace_fn simply returns the loss, but in general can depend on the gradients and variables (if trainable_variables is not NULL then variables==trainable_variables; otherwise it is the list of all variables accessed during execution of loss_fn()), as well as any other quantities captured in the closure of trace_fn, for example, statistics of a variational distribution. Default value: function(loss,grads,variables) loss.

variational_loss_fn function with signature loss <- variational_loss_fn(target_log_prob_fn,surrogate_posterior,sample_size,seed) defining a variational loss function. The default is a Monte Carlo approximation to the standard evidence lower bound (ELBO), equivalent to minimizing the 'reverse' KL[q||p] divergence between the surrogate q and true posterior p. Default value: functools.partial(tfp.vi.monte_carlo_variational_loss,discrepancy_fn=tfp.vi.kl_reverse,use_reparameterization=True).

sample_size integer number of Monte Carlo samples to use in estimating the variational divergence. Larger values may stabilize the optimization, but at higher cost per step in time and memory. Default value: 1.

trainable_variables Optional list of tf$Variable instances to optimize with respect to. If NULL, defaults to the set of all variables accessed during the computation of the variational bound, i.e., those defining surrogate_posterior and the model target_log_prob_fn. Default value: NULL.

seed integer to seed the random number generator.

name name prefixed to ops created by this function. Default value: 'fit_surrogate_posterior'.

Details

This corresponds to minimizing the 'reverse' Kullback-Liebler divergence (KL[q||p]) between the variational distribution and the unnormalized target_log_prob_fn, and defines a lower bound on the marginal log likelihood, log p(x) >= -elbo_loss.

More generally, this function supports fitting variational distributions that minimize any Csiszar f-divergence.

Value

results Tensor or nested structure of Tensors, according to the return type of result_fn. Each Tensor has an added leading dimension of size num_steps, packing the trajectory of the result over the course of the optimization.

See Also

Other vi-functions: vi_amari_alpha(), vi_arithmetic_geometric(), vi_chi_square(), vi_csiszar_vimco(), vi_dual_csiszar_function(), vi_jeffreys(), vi_jensen_shannon(), vi_kl_forward(), vi_kl_reverse(), vi_log1p_abs(), vi_modified_gan(), vi_monte_carlo_variational_loss(), vi_pearson(), vi_squared_hellinger(), vi_symmetrized_csiszar_function()
vi_jeffreys

The Jeffreys Csiszar-function in log-space

Description

A Csiszar-function is a member of $F = \{ f : \mathbb{R}_+ \to \mathbb{R} : f \text{ convex} \}$.

Usage

```r
vi_jeffreys(logu, name = NULL)
```

Arguments

- `logu` float-like Tensor representing $\log(u)$ from above.
- `name` name prefixed to Ops created by this function.

Details

The Jeffreys Csiszar-function is:

$$
f(u) = 0.5 \left( u \log(u) - \log(u) \right)
= 0.5 \cdot \text{kl_forward} + 0.5 \cdot \text{kl_reverse}
= \text{symmetrized_csiszar_function}(\text{kl_reverse})
= \text{symmetrized_csiszar_function}(\text{kl_forward})
$$

This Csiszar-function induces a symmetric $f$-Divergence, i.e., $D_f[p, q] = D_f[q, p]$.

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for $\log(u) >> 0$.

Value

`jeffreys_of_u`: float-like Tensor of the Csiszar-function evaluated at $u = \exp(\logu)$.

See Also

Other vi-functions: `vi_amari_alpha()`, `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jensen_shannon()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_log1p_abs()`, `vi_modified_gan()`, `vi_monte_carlo_variational_loss()`, `vi_pearson()`, `vi_squared_hellinger()`, `vi_symmetrized_csiszar_function()`
vi_jensen_shannon

The Jensen-Shannon Csiszar-function in log-space

Description

A Csiszar-function is a member of \( F = \{ f:R_+ \to R : f \text{ convex } \} \).

Usage

\[
\text{vi_jensen_shannon}(\text{logu}, \text{self_normalized} = \text{FALSE}, \text{name} = \text{NULL})
\]

Arguments

- \text{logu} float-like Tensor representing \( \log(u) \) from above.
- \text{self_normalized} logical indicating whether \( f'(u=1)=0 \). When \( f'(u=1)=0 \) the implied Csiszar f-Divergence remains non-negative even when \( p, q \) are unnormalized measures.
- \text{name} name prefixed to Ops created by this function.

Details

When \( \text{self_normalized} = \text{True} \), the Jensen-Shannon Csiszar-function is:

\[
f(u) = u \log(u) - (1 + u) \log(1 + u) + (u + 1) \log(2)
\]

When \( \text{self_normalized} = \text{False} \) the \( (u + 1) \log(2) \) term is omitted.

Observe that as an f-Divergence, this Csiszar-function implies:

\[
D_f[p, q] = KL[p, m] + KL[q, m]
\]

\[
m(x) = 0.5 \ p(x) + 0.5 \ q(x)
\]

In a sense, this divergence is the "reverse" of the Arithmetic-Geometric f-Divergence.

This Csiszar-function induces a symmetric f-Divergence, i.e., \( D_f[p, q] = D_f[q, p] \).

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( \log(u) \gg 0 \).

Value

\( \text{jensen_shannon_of_u} \), float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\text{logu}) \).

References

### vi_kl_forward

**The forward Kullback-Leibler Csiszar-function in log-space**

#### Description

A Csiszar-function is a member of $F = \{ f : \mathbb{R}_+ \to \mathbb{R} : f \text{ convex} \}$.

#### Usage

```r
vi_kl_forward(logu, self_normalized = FALSE, name = NULL)
```

#### Arguments

- **logu**: float-like Tensor representing $\log(u)$ from above.
- **self_normalized**: logical indicating whether $f'(u=1)=0$. When $f'(u=1)=0$ the implied Csiszar f-Divergence remains non-negative even when $p, q$ are unnormalized measures.
- **name**: name prefixed to Ops created by this function.

#### Details

When `self_normalized = TRUE`, the KL-reverse Csiszar-function is $f(u) = u \log(u) - (u - 1)$. When `self_normalized = FALSE` the $(u - 1)$ term is omitted. Observe that as an f-Divergence, this Csiszar-function implies: $\mathbb{D}_f[p,q] = \text{KL}[q,p]$.

The KL is "forward" because in maximum likelihood we think of minimizing $q$ as in $\text{KL}[p,q]$.

**Warning**: when `self_normalized = TRUE` this function makes non-log-space calculations and may therefore be numerically unstable for $\log\|u\| > 0$.

#### Value

$\text{kl_forward_of_u}$: float-like Tensor of the Csiszar-function evaluated at $u = \exp(\logu)$.

#### See Also

Other vi-functions: `vi_amari_alpha()`, `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_log1p_abs()`, `vi_modified_gan()`, `vi_monte_carlo_variational_loss()`, `vi_pearson()`, `vi_squared_hellinger()`, `vi_symmetrized_csiszar_function()`
vi_kl_reverse  

The reverse Kullback-Leibler Csiszar-function in log-space

Description

A Csiszar-function is a member of \( F = \{ f:R_+ \to R : f \text{ convex} \} \).

Usage

\[
\text{vi_kl_reverse}(\log u, \text{self_normalized} = \text{FALSE}, \text{name} = \text{NULL})
\]

Arguments

- **logu**: float-like Tensor representing \( \log(u) \) from above.
- **self_normalized**: logical indicating whether \( f(u=1)=0 \). When \( f(u=1)=0 \) the implied Csiszar f-Divergence remains non-negative even when \( p, q \) are unnormalized measures.
- **name**: name prefixed to Ops created by this function.

Details

When \( \text{self_normalized} = \text{TRUE} \), the KL-reverse Csiszar-function is \( f(u) = -\log(u) + (u - 1) \). When \( \text{self_normalized} = \text{FALSE} \) the \((u - 1)\) term is omitted. Observe that as an f-Divergence, this Csiszar-function implies: \( D_f[p,q] = \text{KL}[q,p] \)

The KL is “reverse” because in maximum likelihood we think of minimizing \( q \) as in \( \text{KL}[p,q] \).

Warning: when \( \text{self_normalized} = \text{TRUE} \) this function makes non-log-space calculations and may therefore be numerically unstable for \( |\log u| \gg 0 \).

Value

\( \text{kl_reverse_of_u} \): float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

See Also

Other vi-functions: \( \text{vi_amari_alpha()}, \text{vi_arithmetic_geometric()}, \text{vi_chisquare()}, \text{vi_csiszar_vimco()}, \text{vi_dual_csiszar_function()}, \text{vi_fit_surrogate_posterior()}, \text{vi_jeffreys()}, \text{vi_jensen_shannon()}, \text{vi_kl_forward()}, \text{vi_log1p_abs()}, \text{vi_modified_gan()}, \text{vi_monte_carlo_variational_loss()}, \text{vi_pearson()}, \text{vi_squared_hellinger()}, \text{vi_symmetrized_csiszar_function()} \)
vi_log1p_abs

Description

A Csiszar-function is a member of \( F = \{ f: \mathbb{R}_+ \to \mathbb{R} : f \text{ convex} \} \).

Usage

\[
\text{vi_log1p_abs}(\log u, \text{name} = \text{NULL})
\]

Arguments

- \( \log u \): float-like Tensor representing \( \log(u) \) from above.
- \( \text{name} \): name prefixed to Ops created by this function.

Details

The Log1p-Abs Csiszar-function is:

\[
f(u) = u^{\text{sign}(u-1)} - 1
\]

This function is so-named because it was invented from the following recipe. Choose a convex function \( g \) such that \( g(0) = 0 \) and solve for \( f \):

\[
\log(1 + f(u)) = g(\log(u)).
\]

\[
\iff \quad f(u) = \exp(g(\log(u))) - 1
\]

That is, the graph is identically \( g \) when y-axis is log1p-domain and x-axis is log-domain.

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( \|\log u\| \gg 0 \).

Value

\( \log1p\_abs\_of\_u \): float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

See Also

Other vi-functions: \( \text{vi_amari_alpha()} \), \( \text{vi_arithmetic_geometric()} \), \( \text{vi_chi_square()} \), \( \text{vi_csiszar_vimco()} \), \( \text{vi_dual_csiszar_function()} \), \( \text{vi_fit_surrogate_posterior()} \), \( \text{vi_jeffreys()} \), \( \text{vi_jensen_shannon()} \), \( \text{vi_kl_forward()} \), \( \text{vi_kl_reverse()} \), \( \text{vi_modified_gan()} \), \( \text{vi_monte_carlo_variational_loss()} \), \( \text{vi_pearson()} \), \( \text{vi_squared_hellinger()} \), \( \text{vi_symmetrized_csiszar_function()} \)
The Modified-GAN Csiszar-function in log-space

Description

A Csiszar-function is a member of \( F = \{ f: \mathbb{R}_+ \to \mathbb{R} : f \text{ convex} \} \).

Usage

\[
\text{vi_modified_gan}(\log u, \text{self_normalized} = \text{FALSE}, \text{name} = \text{NULL})
\]

Arguments

- \( \log u \): float-like Tensor representing \( \log(u) \) from above.
- \( \text{self_normalized} \): logical indicating whether \( f'(u=1)=0 \). When \( f'(u=1)=0 \) the implied Csiszar f-Divergence remains non-negative even when \( p, q \) are unnormalized measures.
- \( \text{name} \): name prefixed to Ops created by this function.

Details

When \( \text{self_normalized} = \text{True} \) the modified-GAN (Generative/Adversarial Network) Csiszar-function is:

\[
f(u) = \log(1 + u) - \log(u) + 0.5 (u - 1)
\]

When \( \text{self_normalized} = \text{False} \) the \( 0.5 (u - 1) \) is omitted.

The unmodified GAN Csiszar-function is identical to Jensen-Shannon (with \( \text{self_normalized} = \text{False} \)).

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( |\log u| >> 0 \).

Value

\( \text{jensen_shannon_of_u} \), float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

See Also

Other vi-functions: \( \text{vi_amari_alpha()} \), \( \text{vi_arithmetic_geometric()} \), \( \text{vi_chi_square()} \), \( \text{vi_csiszar_vimco()} \), \( \text{vi_dual_csiszar_function()} \), \( \text{vi_fit_surrogate_posterior()} \), \( \text{vi_jeffreys()} \), \( \text{vi_jensen_shannon()} \), \( \text{vi_kl_forward()} \), \( \text{vi_kl_reverse()} \), \( \text{vi_log1p_abs()} \), \( \text{vi_monte_carlo_variational_loss()} \), \( \text{vi_pearson()} \), \( \text{vi_squared_hellinger()} \), \( \text{vi_symmetrized_csiszar_function()} \)
vi_monte_carlo_variational_loss

Monte-Carlo approximation of an f-Divergence variational loss

Description

Variational losses measure the divergence between an unnormalized target distribution \( p \) (provided via `target_log_prob_fn`) and a surrogate distribution \( q \) (provided as `surrogate_posterior`). When the target distribution is an unnormalized posterior from conditioning a model on data, minimizing the loss with respect to the parameters of `surrogate_posterior` performs approximate posterior inference.

Usage

```r
vi_monte_carlo_variational_loss(
  target_log_prob_fn,
  surrogate_posterior,
  sample_size = 1,
  discrepancy_fn = vi_kl_reverse,
  use_reparameterization = NULL,
  seed = NULL,
  name = NULL
)
```

Arguments

- `target_log_prob_fn` function that takes a set of Tensor arguments and returns a Tensor log-density. Given `q_sample <- surrogate_posterior$sample(sample_size)`, this will be (in Python) called as `target_log_prob_fn(q_sample)` if `q_sample` is a list or a tuple, `target_log_prob_fn(**q_sample)` if `q_sample` is a dictionary, or `target_log_prob_fn(q_sample)` if `q_sample` is a Tensor. It should support batched evaluation, i.e., should return a result of shape `[sample_size]`.

- `surrogate_posterior` A `tfp$Distributions$Distribution` instance defining a variational posterior (could be a `tfp$Distributions$JointDistribution`). Crucially, the distribution's `log_prob` and (if reparameterized) `sample` methods must directly invoke all ops that generate gradients to the underlying variables. One way to ensure this is to use `tfp$util$DeferredTensor` to represent any parameters defined as transformations of unconstrained variables, so that the transformations execute at runtime instead of at distribution creation.

- `sample_size` integer number of Monte Carlo samples to use in estimating the variational divergence. Larger values may stabilize the optimization, but at higher cost per step in time and memory. Default value: 1.

- `discrepancy_fn` function representing a Csiszar \( f \) function in log-space. That is, `discrepancy_fn(log(u)) = f(u)`, where \( f \) is convex in \( u \). Default value: `vi_kl_reverse`.

use_reparametrization

When NULL (the default), automatically set to: surrogate_posterior.reparameterization_type == tfp$distributions$FULLY_REPARAMETERIZED. When TRUE uses the standard Monte-Carlo average. When FALSE uses the score-gradient trick. (See above for details.) When FALSE, consider using csiszar_vimco.

seed

integer seed for surrogate_posterior$sample.

name

name prefixed to Ops created by this function.

Details

This function defines divergences of the form E_q[discrepancy_fn(log p(z) - log q(z))], sometimes known as f-divergences.

In the special case discrepancy_fn(logu) == -logu (the default vi_kl_reverse), this is the reverse Kullback-Liebler divergence KL[q||p], whose negation applied to an unnormalized p is the widely-used evidence lower bound (ELBO). Other cases of interest available under tfp$vi include the forward KL[p||q] (given by vi_kl_forward(logu) == exp(logu) * logu), total variation distance, Amari alpha-divergences, and more.

Csiszar f-divergences

A Csiszar function f is a convex function from R^+ (the positive reals) to R. The Csiszar f-Divergence is given by:

D_f[p(X), q(X)] := E_{q(X)}[ f( p(X) / q(X) ) ]

= m**-1 sum_j^m f( p(x_j) / q(x_j) ),
where x_j ~iid q(X)

For example, f = lambda u: -log(u) recovers KL[q||p], while f = lambda u: u * log(u) recovers the forward KL[p||q]. These and other functions are available in tfp$vi.

Tricks: Reparameterization and Score-Gradient

When q is "reparameterized", i.e., a diffeomorphic transformation of a parameterless distribution (e.g., Normal(Y; m, s) <=> Y = sX + m, X ~ Normal(0,1)), we can swap gradient and expectation, i.e., grad[Avg{ s_i : i=1...n }] = Avg{ grad[s_i] : i=1...n } where S_n=Avg{s_i} and s_i = f(x_i), x_i ~iid q(X).

However, if q is not reparameterized, TensorFlow's gradient will be incorrect since the chain-rule stops at samples of unreparameterized distributions. In this circumstance using the Score-Gradient trick results in an unbiased gradient, i.e.,

grad[ E_q[f(X)] ]
= grad[ int dx q(x) f(x) ]
= int dx grad[ q(x) f(x) ]
= int dx [ q'(x) f(x) + q(x) f'(x) ]
= int dx q(x) [ q'(x) / q(x) f(x) + f'(x) ]
= int dx q(x) grad[ f(x) q(x) / stop_grad[q(x)] ]
= E_q[ grad[ f(x) q(x) / stop_grad[q(x)] ] ]

Unless q.reparameterization_type != tfd.FULLY_REPARAMETERIZED it is usually preferable to set use_reparametrization = True.

Example Application: The Csiszar f-Divergence is a useful framework for variational inference. I.e., observe that,
The inequality follows from the fact that the "perspective" of $f$, i.e., $(s, t) \mapsto t f(s / t)$, is convex in $(s, t)$ when $s/t$ is in domain($f$) and $t$ is a real. Since the above framework includes the popular Evidence Lower Bound (ELBO) as a special case, i.e., $f(u) = -\log(u)$, we call this framework "Evidence Divergence Bound Optimization" (EDBO).

**Value**

`monte_carlo_variational_loss` float-like Tensor Monte Carlo approximation of the Csiszar f-Divergence.

**References**


**See Also**

Other vi-functions: `vi_amari_alpha()`, `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_jensen_shannon()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_log1p_abs()`, `vi_modified_gan()`, `vi_pearson()`, `vi_squared_hellinger()`, `vi_symmetrized_csiszar_function()`

---

### `vi_pearson`

*The Pearson Csiszar-function in log-space*

**Description**

A Csiszar-function is a member of $F = \{ f: \mathbb{R}_+ \to \mathbb{R} : f \text{ convex} \}$.

**Usage**

`vi_pearson(logu, name = NULL)`

**Arguments**

- `logu` float-like Tensor representing $\log(u)$ from above.
- `name` name prefixed to Ops created by this function.

**Details**

The Pearson Csiszar-function is:

$$f(u) = (u - 1)^2$$

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for $\logu \gg 0$. 
The Squared-Hellinger Csiszar-function in log-space

Description
A Csiszar-function is a member of \( F = \{ f : \mathbb{R}_+ \to \mathbb{R} : f \text{ convex} \} \).

Usage
\[
\text{vi_squared_hellinger}(\log u, \text{name} = \text{NULL})
\]

Arguments
- \( \log u \): float-like Tensor representing \( \log(u) \) from above.
- \( \text{name} \): name prefixed to Ops created by this function.

Details
The Squared-Hellinger Csiszar-function is:
\[
f(u) = (\sqrt{u} - 1)^2
\]
This Csiszar-function induces a symmetric f-Divergence, i.e., \( D_f[p, q] = D_f[q, p] \).
Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( |\log u| \gg 0 \).

Value
Squared-Hellinger_of_u: float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

See Also
Other vi-functions: vi_amari_alpha(), vi_arithmetic_geometric(), vi_chi_square(), vi_csiszar_vimco(), vi_dual_csiszar_function(), vi_fit_surrogate_posterior(), vi_jeffreys(), vi_jensen_shannon(), vi_kl_forward(), vi_kl_reverse(), vi_log1p_abs(), vi_modified_gan(), vi_monte_carlo_variational_loss(), vi_pearson(), vi_symmetrized_csiszar_function()
vi_symmetrized_csiszar_function

Symmetrizes a Csiszar-function in log-space

Description
A Csiszar-function is a member of F = { f:R_+ to R : f convex }.

Usage
vi_symmetrized_csiszar_function(logu, csiszar_function, name = NULL)

Arguments
logu float-like Tensor representing log(u) from above.
csiszar_function function representing a Csiszar-function over log-domain.
name name prefixed to Ops created by this function.

Details
The symmetrized Csiszar-function is defined as:

\[ f_g(u) = 0.5 \cdot g(u) + 0.5 \cdot u \cdot g\left(\frac{1}{u}\right) \]

where \( g \) is some other Csiszar-function. We say the function is "symmetrized" because:

\[ D_{f_g}[p, q] = D_{f_g}[q, p] \]

for all \( p \ll q \) (i.e., \( \text{support}(p) = \text{support}(q) \)). There exists alternatives for symmetrizing a Csiszar-function. For example,

\[ f_g(u) = \max(f(u), f^*(u)) \]

where \( f^* \) is the dual Csiszar-function, also implies a symmetric f-Divergence.

Example: When either of the following functions are symmetrized, we obtain the Jensen-Shannon Csiszar-function, i.e.,

\[
\begin{align*}
g(u) &= -\log(u) - (1 + u) \log((1 + u) / 2) + u - 1 \\
h(u) &= \log(4) + 2 \cdot u \cdot \log(u / (1 + u))
\end{align*}
\]

implies,

\[ f_g(u) = f_h(u) = u \cdot \log(u) - (1 + u) \cdot \log((1 + u) / 2) = \text{jensen_shannon}(\log(u)). \]

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( |\logu| \gg 0 \).
**Value**

symmetrized_g_of_u: float-like Tensor of the result of applying the symmetrization of g evaluated at \( u = \exp(\log u) \).

**See Also**

Other vi-functions: `vi_amari_alpha()`, `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_jensen_shannon()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_log1p_abs()`, `vi_modified_gan()`, `vi_monte_carlo_variational_loss()`, `vi_pearson()`, `vi_squared_hellinger()`

---

**vi_total_variation**  
*The Total Variation Csiszar-function in log-space*

**Description**

A Csiszar-function is a member of \( F = \{ f: R_+ \to R : f \text{ convex } \} \).

**Usage**

`vi_total_variation(logu, name = NULL)`

**Arguments**

- `logu` float-like Tensor representing \( \log(u) \) from above.
- `name` name prefixed to Ops created by this function.

**Details**

The Total-Variation Csiszar-function is:

\[
    f(u) = 0.5 \mid u - 1 \mid
\]

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( \log u \gg 0 \).

**Value**

`total_variation_of_u`: float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

**See Also**

Other vi-functions: `vi_t_power()`, `vi_triangular()`
**vi_triangular**

*The Triangular Csiszar-function in log-space*

**Description**

The Triangular Csiszar-function is:

**Usage**

\[
\text{vi_triangular}(\log u, \text{name} = \text{NULL})
\]

**Arguments**

- \( \log u \): float-like Tensor representing \( \log(u) \) from above.
- \( \text{name} \): name prefixed to Ops created by this function.

**Details**

\[
f(u) = \frac{(u - 1)^2}{1 + u}
\]

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( |\log u| \gg 0 \).

**Value**

\( \text{triangular_of}_u \): float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

**See Also**

- Other vi-functions: \( \text{vi_t_power}(), \text{vi_total_variation}() \)

---

**vi_t_power**

*The T-Power Csiszar-function in log-space*

**Description**

A Csiszar-function is a member of \( F = \{ f:R_+ \to R : f \text{ convex } \} \).

**Usage**

\[
\text{vi_t_power}(\log u, t, \text{self_normalized} = \text{FALSE}, \text{name} = \text{NULL})
\]
Arguments

- **logu**: float-like Tensor representing \( \log(u) \) from above.
- **t**: Tensor of same dtype as logu and broadcastable shape.
- **self_normalized**: logical indicating whether \( f(u=1) = 0 \). When \( f(u=1) = 0 \) the implied Csiszar f-Divergence remains non-negative even when \( p, q \) are unnormalized measures.
- **name**: name prefixed to Ops created by this function.

Details

When **self_normalized** = True the T-Power Csiszar-function is:

\[
f(u) = s \left[ u^t - 1 - t(u - 1) \right]
\]

\[
s = \begin{cases} 
-1 & 0 < t < 1 \\ 
+1 & \text{otherwise} 
\end{cases}
\]

When **self_normalized** = False the \(-t(u - 1)\) term is omitted.

This is similar to the amari_alpha Csiszar-function, with the associated divergence being the same up to factors depending only on \( t \).

Warning: when **self_normalized** = True this function makes non-log-space calculations and may therefore be numerically unstable for \( \logu \approx 0 \).

Value

- **t_power_of_u**: float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\logu) \).

See Also

Other vi-functions#': **vi_total_variation**, **vi_triangular**
Index

*Topic datasets
  tfp, 392

glm_families, 8, 10, 12
glm_fit, 8
glm_fit(), 8
glm_fit.tensorflow.tensor, 8, 9, 12
glm_fit.tensorflow.tensor(), 9
glm_fit_one_step, 11
glm_fit_one_step.tensorflow.tensor, 8, 10, 11
glm_fit_one_step.tensorflow.tensor(), 11

initializer_blockwise, 13
initializer_constant(), 13
install_tfprobability, 13

keras::initializer_glorot_uniform(), 13

layer_conv_3d_reparameterization, 15, 20, 23, 26, 29, 31, 32, 37, 39, 41, 42, 57
layer_dense_flipout, 15, 21, 23, 26, 29, 32, 34, 39, 41, 42, 57
layer_dense_local_reparameterization, 15, 21, 23, 26, 29, 32, 34, 37, 39, 41, 42, 57
layer_dense_reparameterization, 15, 21, 23, 26, 29, 32, 34, 37, 39, 41, 42, 57
layer_dense_variational, 15, 21, 23, 26, 29, 32, 34, 37, 39, 41, 42, 57
layer_distribution_lambda, 18, 43, 44–46, 48–54, 56
layer_independent_bernoulli, 18, 43, 44, 45, 46, 48–54, 56
layer_independent_logistic, 18, 43, 44, 45, 46, 48–54, 56
layer_independent_normal, 18, 43–45, 46, 48–54, 56
layer_independent_normal(), 18, 43–45, 48–54, 56
layer_independent_poisson, 18, 43–46, 47, 49–54, 56
layer_kl_divergence_add_loss, 18, 43–46, 48, 49, 50–54, 56
layer_kl_divergence_regularizer, 18, 43–46, 48, 49, 51–54, 56
layer_mixture_logistic, 18, 43–46, 48–50, 50, 52–54, 56
layer_mixture_normal, 18, 43–46, 48–51, 51, 53, 54, 56
layer_mixture_same_family, 18, 43–46, 48–52, 52, 54, 56
layer_multivariate_normal_tri_l, 18, 43–46, 48–53, 54, 56
layer_one_hot_categorical, 18, 43–46, 48–54, 55
layer_variable, 15, 21, 23, 26, 29, 32, 34,
layer_variational_gaussian_process, 57
mcmc_dual_averaging_step_size_adaptation, 58, 64–66, 68, 72, 74, 82, 84, 86–89
mcmc_effective_sample_size, 61, 71, 75, 77, 80
mcmc_hamiltonian_monte_carlo, 61, 62, 65, 66, 68, 72, 74, 82, 84, 86–89
mcmc_metropolis_adjusted_langevin_algorithm, 61, 64, 66, 68, 72, 74, 82, 84, 86–89
mcmc_metropolis_hastings, 61, 64, 65, 66, 68, 72, 74, 82, 84, 86–89
mcmc_no_u_turn_sampler, 61, 64–66, 67, 72, 74, 82, 84, 86–89
mcmc_replica_exchange_mc, 61, 64–66, 68, 72, 73, 82, 84, 86–89
mcmc_sample_annealed_importance_chain, 62, 71, 74, 77, 80
mcmc_sample_chain, 62, 71, 75, 76, 80
mcmc_sample_chain(), 75, 80
mcmc_sample_halton_sequence, 62, 71, 75, 77, 78
mcmc_simple_step_size_adaptation, 61, 64–66, 68, 72, 74, 80, 84, 86–89
mcmc_slice_sampler, 61, 64–66, 68, 72, 74, 82, 83, 86–89
mcmc_transformed_transition_kernel, 61, 64–66, 68, 72, 74, 82, 84, 85, 87–89
mcmc_uncalibrated_hamiltonian_monte_carlo, 61, 64–66, 68, 72, 74, 82, 84, 86, 88, 89
mcmc_uncalibrated_langevin, 61, 64–66, 68, 72, 74, 82, 84, 86, 87, 89
mcmc_uncalibrated_random_walk, 61, 64–66, 68, 72, 74, 82, 84, 86–88, 88

params_size_categorical_mixture_of_one_hot_categorical, 89
params_size_mixture_logistic, 92
params_size_mixture_normal, 93
params_size_mixture_same_family, 93
params_size_multivariate_normal_tri_l, 94
params_size_one_hot_categorical, 94

sts_build_factored_surrogate_posterior, 102, 104, 108, 109, 114, 116, 125, 126
sts_build_factored_variational_loss, 102, 103, 108, 109, 114, 116, 125, 126
sts_decompose_by_component, 102, 104, 107, 109, 114, 116, 125, 126
sts_decompose_by_component(), 99, 110, 118, 119, 122, 128, 132, 136, 141, 142
sts_decompose_forecast_by_component, 102, 104, 108, 108, 114, 116, 125, 126
sts_fit_with_hmc, 102, 104, 108, 109, 112, 116, 125, 126
sts_fit_with_hmc(), 99, 110, 118, 119, 122, 128, 132, 136, 141, 142


tfb_inverse, 164, 165, 170, 172


tfb_inverse_log_det_jacobian, 164, 165, 171, 172


tfb_iterated_sigmoid_centered, 143,
INDEX


tfb_masked_autoregressive_flow(). 16


INDEX

388, 390, 392


tfd_empirical, 220, 221, 223, 225, 227, 229,
INDEX


tfd_kl_divergence, 234, 239, 240, 250, 284, 295, 298, 299, 303, 334, 337, 343, 348, 353, 361


tfd_log_survival_function, 234, 239, 240,
250, 285, 295, 298, 299, 303,
334, 337, 343, 348, 353, 361

**tfd_logistic**, 220, 222, 223, 225, 227, 229,
232, 233, 235, 237, 238, 242, 244,
246, 249, 251, 256, 258, 261, 263,
267, 268, 270, 271, 273, 275, 276,
279, 281, 282, 284, 286, 288, 290,
292, 292, 297, 301, 303, 305, 307,
310, 312, 314, 316, 318, 319, 321,
323, 324, 328, 331, 334, 336, 339,
340, 342, 344, 347, 349, 352, 355,
356, 358, 360, 365, 369, 372, 374,
376, 378, 381, 383, 384, 386, 388,
390, 392

tfd_logit_normal, 294

tfd_mean, 234, 239, 240, 250, 285, 295, 298,
299, 303, 334, 337, 343, 348, 353, 361

**tfd_mean()**, 220, 221, 223, 225, 227,
229–231, 233, 235, 237, 238, 242,
244, 246, 248, 249, 251, 253, 254,
256, 258, 261, 263, 265, 267, 270,
271, 273, 275, 276, 279, 281, 282,
284, 286, 288, 290, 292, 293, 295,
297, 301, 302, 305, 307, 310, 312,
314, 316, 317, 319, 321, 322, 324,
326, 328, 330, 331, 333, 336, 338,
340, 342, 344, 347, 349, 352, 355,
356, 358, 360, 365, 367, 369, 371,
374, 376, 378, 381, 382, 384, 386,
388, 390, 392

tfd_mixture, 220, 222, 223, 225, 227, 229,
232, 233, 235, 237, 238, 242, 244,
246, 249, 251, 256, 258, 261, 263,
267, 268, 270, 271, 273, 275, 276,
279, 281, 282, 284, 286, 288, 290,
292, 294, 297, 300, 303, 305, 307,
310, 312, 314, 316, 318, 319, 321,
323, 324, 328, 331, 334, 336, 339,
340, 342, 344, 347, 349, 352, 355,
356, 358, 360, 365, 369, 372, 374,
INDEX


tfd_vector_deterministic, 366

INDEX

| tfp, 392 |
| tfp_version, 393 |
| vi_amari(alpha, 393, 395–398, 400, 401, 403–406, 409, 410, 412 |
| vi_arithmetic_geometric, 394, 394 |
432

vi_chi_square, 394, 395, 395, 397, 398, 400, 401, 403–406, 409, 410, 412
vi_csiszar_vimco, 394–396, 396, 398, 400, 401, 403–406, 409, 410, 412
vi_dual_csiszar_function, 394–397, 398, 400, 401, 403–406, 409, 410, 412
vi_fit_surrogate_posterior, 394–398, 399, 401, 403–406, 409, 410, 412
vi_jeffreys, 394–398, 400, 401, 403–406, 409, 410, 412
vi_jensen_shannon, 394–398, 400, 401, 402, 403–406, 409, 410, 412
vi_kl_forward, 394–398, 400, 401, 403, 403, 404–406, 409, 410, 412
vi_kl_reverse, 394–398, 400, 401, 403, 404, 405, 406, 409, 410, 412
vi_lg1p_abs, 394–398, 400, 401, 403, 404, 405, 406, 409, 410, 412
vi_modified_gan, 394–398, 400, 401, 403–405, 406, 409, 410, 412
vi_monte_carlo_variational_loss, 394–398, 400, 401, 403–406, 407, 410, 412
vi_pearson, 394–398, 400, 401, 403–406, 409, 409, 410, 412
vi_squared_hellinger, 394–398, 400, 401, 403–406, 409, 410, 412
vi_symmetrized_csiszar_function, 394–398, 400, 401, 403–406, 409, 410, 411
vi_t_power, 412, 413, 413
vi_total_variation, 412, 413, 414
vi_triangular, 412, 413, 414