Package ‘luz’

June 17, 2021

Title  Higher Level 'API' for 'torch'

Version  0.1.0

Description  A high level interface for 'torch' providing utilities to reduce the
the amount of code needed for common tasks, abstract away torch details and
make the same code work on both the 'CPU' and 'GPU'. It's flexible enough to
support expressing a large range of models. It's heavily inspired by 'fastai' by
Howard et al. (2020) <arXiv:2002.04688>, 'Keras' by Chollet et al. (2015) and

License  MIT + file LICENSE


Encoding  UTF-8

RoxygenNote  7.1.1

Imports  torch (>= 0.4.0), magrittr, zeallot, rlang, glue, 
progress, R6, generics, purrr, ellipsis, fs, prettyunits, cli

Suggests  knitr, rmarkdown, testthat (>= 3.0.0), covr, Metrics, withr, 
vdiffr, ggplot2, dplyr

VignetteBuilder  knitr

Config/testthat/edition  3

Collate  'accelerator.R' 'utils.R' 'callbacks.R'
'callbacks-interrupt.R' 'callbacks-profile.R' 'context.R'
'metrics.R' 'metrics-auc.R' 'module-plot.R' 'module-print.R'
'module.R' 'reexports.R' 'serialization.R'

NeedsCompilation  no

Author  Daniel Falbel [aut, cre, cph],
        RStudio [cph]

Maintainer  Daniel Falbel <daniel@rstudio.com>

Repository  CRAN

Date/Publication  2021-06-17 08:40:04 UTC
## R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>accelerator</td>
<td>2</td>
</tr>
<tr>
<td>context</td>
<td>3</td>
</tr>
<tr>
<td>ctx</td>
<td>5</td>
</tr>
<tr>
<td>fit.luz_module_generator</td>
<td>6</td>
</tr>
<tr>
<td>luz_callback</td>
<td>7</td>
</tr>
<tr>
<td>luz_callback_csv_logger</td>
<td>10</td>
</tr>
<tr>
<td>luz_callback_early_stopping</td>
<td>11</td>
</tr>
<tr>
<td>luz_callback_interrupt</td>
<td>12</td>
</tr>
<tr>
<td>luz_callback_lr_scheduler</td>
<td>13</td>
</tr>
<tr>
<td>luz_callback_metrics</td>
<td>14</td>
</tr>
<tr>
<td>luz_callback_model_checkpoint</td>
<td>14</td>
</tr>
<tr>
<td>luz_callback_profile</td>
<td>16</td>
</tr>
<tr>
<td>luz_callback_progress</td>
<td>17</td>
</tr>
<tr>
<td>luz_callback_train_valid</td>
<td>17</td>
</tr>
<tr>
<td>luz_load</td>
<td>18</td>
</tr>
<tr>
<td>luz_load_model_weights</td>
<td>18</td>
</tr>
<tr>
<td>luz_metric</td>
<td>19</td>
</tr>
<tr>
<td>luz_metric_accuracy</td>
<td>21</td>
</tr>
<tr>
<td>luz_metric_binary_accuracy</td>
<td>22</td>
</tr>
<tr>
<td>luz_metric_binary_accuracy_with_logits</td>
<td>23</td>
</tr>
<tr>
<td>luz_metric_binary_auroc</td>
<td>24</td>
</tr>
<tr>
<td>luz_metric_mae</td>
<td>25</td>
</tr>
<tr>
<td>luz_metric_mse</td>
<td>26</td>
</tr>
<tr>
<td>luz_metric_multiclass_auroc</td>
<td>26</td>
</tr>
<tr>
<td>luz_metric_rmse</td>
<td>28</td>
</tr>
<tr>
<td>luz_save</td>
<td>28</td>
</tr>
<tr>
<td>setup</td>
<td>29</td>
</tr>
<tr>
<td>set_hpams</td>
<td>30</td>
</tr>
<tr>
<td>set_opt_hpams</td>
<td>30</td>
</tr>
</tbody>
</table>

## Index

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>accelerator</td>
<td>2</td>
</tr>
</tbody>
</table>

## accelerator

Create an accelerator

### Usage

```r
accelerator(device_placement = TRUE, cpu = FALSE)
```
Arguments

- **device_placement**
  (logical) whether the accelerator object should handle device placement. Default: TRUE
- **cpu**
  (logical) whether the training procedure should run on the CPU.

---

**context**

*Context object*

Description

Context object storing information about the model training context. See also `ctx`.

Active bindings

- **records** stores information about values logged with `self$log`.
- **device** allows querying the current accelerator device

Methods

**Public methods:**

- `context$log()`
- `context$log_metric()`
- `context$get_log()`
- `context$get_metrics()`
- `context$get_metric()`
- `context$get_formatted_metrics()`
- `context$get_metrics_df()`
- `context$set_verbose()`
- `context$clone()`

**Method** `log()`:

Allows logging arbitrary information in the `ctx`.

*Usage:*

```r
context$log(what, set, value, index = NULL, append = TRUE)
```

*Arguments:*

- **what** (string) What you are logging.
- **set** (string) Usually ‘train’ or ‘valid’ indicating the set you want to log to. But can be arbitrary info.
- **value** value to log
- **value** Arbitrary value to log.
- **index** Index that this value should be logged. If `NULL` the value is added to the end of list, otherwise the index is used.
append If TRUE and a value in the corresponding index already exists, then value is appended to the current value. If FALSE value is overwritten in favor of the new value.

**Method** log_metric(): Log a metric gen its name and value. Metric values are indexed by epoch.

*Usage:*
```
context$log_metric(name, value)
```

*Arguments:*
- `name` name of the metric
- `value` value to log

**Method** get_log(): Get an specific value from the log.

*Usage:*
```
context$get_log(what, set, index = NULL)
```

*Arguments:*
- `what` (string) What you are logging.
- `set` (string) Usually 'train' or 'valid' indicating the set you want to lot to. But can be arbitrary info.
- `index` Index that this value should be logged. If NULL the value is added to the end of list, otherwise the index is used.

**Method** get_metrics(): Get all metric given an epoch and set.

*Usage:*
```
context$get_metrics(set, epoch = NULL)
```

*Arguments:*
- `set` (string) Usually 'train' or 'valid' indicating the set you want to lot to. But can be arbitrary info.
- `epoch` The epoch you want to extract metrics from.

**Method** get_metric(): Get the value of a metric given its name, epoch and set.

*Usage:*
```
context$get_metric(name, set, epoch = NULL)
```

*Arguments:*
- `name` name of the metric
- `set` (string) Usually 'train' or 'valid' indicating the set you want to lot to. But can be arbitrary info.
- `epoch` The epoch you want to extract metrics from.

**Method** get_formatted_metrics(): Get formatted metrics values

*Usage:*
```
context$get_formatted_metrics(set, epoch = NULL)
```

*Arguments:*

set (string) Usually ‘train’ or ‘valid’ indicating the set you want to lot to. But can be arbitrary info.

ePOCH The epoch you want to extract metrics from.

**Method** `get_metrics_df()`: Get a data.frame containing all metrics.

**Usage:**

```r
context$get_metrics_df()
```

**Method** `set_verbose()`: Allows setting the verbose attribute.

**Usage:**

```r
context$set_verbose(verbose = NULL)
```

**Arguments:**

- `verbose` boolean. If `TRUE` verbose mode is used. If `FALSE` non verbose. if `NULL` we use the result of `interactive()`.

**Method** `clone()`: The objects of this class are cloneable with this method.

**Usage:**

```r
context$clone(deep = FALSE)
```

**Arguments:**

- `deep` Whether to make a deep clone.

---

**ctx**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>verbose</td>
<td>The value (TRUE or FALSE) attributed to the verbose argument in fit.</td>
</tr>
<tr>
<td>accelerator</td>
<td>Accelerator object used to query the correct device to place models, data, and etc. It assumes the value passed to the accelerator parameter in fit.</td>
</tr>
<tr>
<td>model</td>
<td>Initialized nn_module object that will be trained during the fit procedure.</td>
</tr>
<tr>
<td>optimizers</td>
<td>A named list of optimizers used during training.</td>
</tr>
<tr>
<td>data</td>
<td>Dataloader passed to the data argument in fit. Modified to yield data in the selected device.</td>
</tr>
<tr>
<td>valid_data</td>
<td>Dataloader passed to the valid_data argument in fit. Modified to yield data in the selected device.</td>
</tr>
<tr>
<td>epochs</td>
<td>Total number of epochs the model will be trained on.</td>
</tr>
<tr>
<td>epoch</td>
<td>Current training epoch.</td>
</tr>
<tr>
<td>iter</td>
<td>Current training iteration. It’s reset every epoch and when going from training to validation.</td>
</tr>
<tr>
<td>training</td>
<td>Whether the model is in training or validation mode. See also help(&quot;luz_callback_train_valid&quot;)</td>
</tr>
</tbody>
</table>
callbacks

List of callbacks that will be called during the training procedure. It’s the union of the list passed to the callbacks parameter and the default callbacks.

step

Closure that will be used to do one step of the model. It’s used for both training and validation. Takes no arguments.

call_callbacks

Call callbacks by name. For example call_callbacks("on_train_begin") will call all callbacks that provide methods for this point.

batch

Last batch obtained by the dataloader. A batch is a list() with 2 elements, one that is used as input and the other as target.

input

First element of the last batch obtained by the current dataloader.

target

Second element of the last batch obtained by the current dataloader.

pred

Last predictions obtained by ctx$model$forward. Note: can be potentially modified by previously ran callbacks.

loss

Last computed loss from the model. Note: this might not be available if you modified the training or validation step.

opt

Current optimizer, i.e. optimizer that will be used to do the next step to update parameters.

opt_nm

Current optimizer name. By default it’s opt, but can change if your model uses more than one optimizer.

metrics

list() with current metric object that are updated at every on_train_batch_end() or on_valid_batch_end().

records

list() recording metric values for training and validation for each epoch. See also help("luz_callback_metrics").

handlers

A named list() of handlers that is passed to rlang::with_handlers() during the training loop and can be used to handle errors or conditions raised by other callbacks.

Context attributes

See Also

Context object: context

---

**fit.luz_module_generator**

Fit a nn_module

**Description**

Fit a nn_module

**Usage**

```r
## S3 method for class 'luz_module_generator'
fit(
  object,
  data,
  epochs = 10,
  callbacks = NULL,
  valid_data = NULL,
  accelerator = NULL,
  verbose = NULL,
  ...
)
```
**luz_callback**

**Arguments**

- **object** (nn_module) An nn_module that has been `setup()`.
- **data** (dataloader) A dataloader created with `torch::dataloader()` used for training the model. The dataloader must return a list with at most 2 items. The first item will be used as input for the module and the second will be used as target for the loss function.
- **epochs** (int) The number of epochs for training the model.
- **callbacks** (list, optional) A list of callbacks defined with `luz_callback()` that will be called during the training procedure. The callbacks `luz_callback_metrics()`, `luz_callback_progress()` and `luz_callback_train_valid()` are always added by default.
- **valid_data** (dataloader, optional) A dataloader created with `torch::dataloader()` that will be used during the validation procedure.
- **accelerator** (accelerator, optional) An optional `accelerator()` object used to configure device placement of the components like `nn_modules`, optimizers and batches of data.
- **verbose** (logical, optional) An optional boolean value indicating if the fitting procedure should emit output to the console during training. By default, it will produce output if `interactive()` is TRUE, otherwise it won’t print to the console.

**...**

Currently unused,

**Value**

A fitted object that can be saved with `luz_save()` and can be printed with `print()` and plotted with `plot()`.

---

**luz_callback**

Create a new callback

**Description**

Create a new callback

**Usage**

```r
luz_callback(
  name = NULL,
  ...,  
  private = NULL,
  active = NULL,
  parent_env = parent.frame(),
  inherit = NULL
)
```
Arguments

- name: name of the callback
- ...: Public methods of the callback. The name of the methods is used to know how they should be called. See the details section.
- private: An optional list of private members, which can be functions and non-functions.
- active: An optional list of active binding functions.
- parent_env: An environment to use as the parent of newly-created objects.
- inherit: A R6ClassGenerator object to inherit from; in other words, a superclass. This is captured as an unevaluated expression which is evaluated in parent_env each time an object is instantiated.

Details

Let's implement a callback that prints 'Iteration n' (where n is the iteration number) for every batch in the training set and 'Done' when an epoch is finished. For that task we use the luz_callback function:

```r
print_callback <- luz_callback(
  name = "print_callback",
  initialize = function(message) {
    self$message <- message
  },
  on_train_batch_end = function() {
    cat("Iteration ", ctx$iter, "\n")
  },
  on_epoch_end = function() {
    cat(self$message, "\n")
  }
)
```

luz_callback() takes a named list of function as argument where the name indicate the moment at which the callback should be called. For instance on_train_batch_end() is called for every batch at the end of the training procedure and on_epoch() end is called at the end of every epoch.

The returned value of luz_callback() is a function that initializes an instance of the callback. Callbacks can have initialization parameters, like the name of a file you want to log the results, in this case, you can pass an initialize method when creating the callback definition and save these parameters to the self object. In the above example, the callback has a message parameter that is printed at the end of each epoch.

Once a callback is defined it can be passed to the fit function via the callbacks parameter, eg:

```r
fitted <- net %>%
  setup(...) %>%
  fit(..., callbacks = list(
    print_callback(message = "Done!"))
)
Callbacks can be called in many different positions of the training loop, including a combinations of them. Here’s an overview of possible callback breakpoints:

Start Fit
- on_fit_begin
Start Epoch Loop
- on_epoch_begin
Start Train
- on_train_begin
Start Batch Loop
- on_train_batch_begin
Start Default Training Step
- on_train_batch_after_pred
- on_train_batch_after_loss
- on_train_batch_before.backward
- on_train_batch_before_step
- on_train_batch_after_step
End Default Training Step:
- on_train_batch_end
End Batch Loop
- on_train_end
End Train
Start Valid
- on_valid_begin
Start Batch Loop
- on_valid_batch_begin
Start Default Validation Step
- on_valid_batch_after_pred
- on_valid_batch_after_loss
End Default Validation Step
- on_valid_batch_end
End Batch Loop
- on_valid_end
End Valid
- on_epoch_end
End Epoch Loop
- on_fit_end
End Fit

Every step marked with a on_* is a point in the training procedure that is available for callbacks to be called.

The other important part of callbacks is the ctx (context) object. See help("ctx") for details.

**Value**

A luz_callback that can be passed to `fit.luz_module_generator()`.
See Also


Examples

```r
print_callback <- luz_callback(
  name = "print_callback",
  on_train_batch_end = function() {
    cat("Iteration ", ctx$iter, "\n")
  },
  on_epoch_end = function() {
    cat("Done!\n")
  }
)
```

---

**luz_callback_csv_logger**

CSV logger callback

Description

Logs metrics obtained during training a file on disk. The file will have 1 line for each epoch/validation.

Usage

```r
luz_callback_csv_logger(path)
```

Arguments

- **path**
  
  path to a file on disk.

See Also

**luz_callback_early_stopping**

*Early stopping callback*

**Description**

Stops training when a monitored metric stops improving.

**Usage**

```r
luz_callback_early_stopping(
  monitor = "valid_loss",
  min_delta = 0,
  patience = 0,
  mode = "min",
  baseline = NULL
)
```

**Arguments**

- **monitor**: A string in the format `<set>_<metric>` where `<set>` can be 'train' or 'valid' and `<metric>` can be the abbreviation of any metric that you are tracking during training. The metric name is case insensitive.
- **min_delta**: Minimum improvement to reset the patience counter.
- **patience**: Number of epochs without improving until stopping training.
- **mode**: Specifies the direction that is considered an improvement. By default 'min' is used. Can also be 'max' (higher is better) and 'zero' (closer to zero is better).
- **baseline**: An initial value that will be used as the best seen value in the beginning. Model will stop training if no better than baseline value is found in the first patience epochs.

**Value**

A `luz_callback` that does early stopping.

**Note**

This callback adds a `on_early_stopping` callback that can be used to call callbacks after as soon as the model stopped training.

If `verbose=TRUE` in `fit.luz_module_generator()` a message is printed when early stopping.

**See Also**

**Description**

Adds a handler that allows interrupting the training loop using `ctrl + c`. Also registers a `on_interrupt` breakpoint so users can register callbacks to be run on training loop interruption.

**Usage**

```r
luz_callback_interrupt()
```

**Value**

A `luz_callback`

**Note**

In general you don’t need to use these callback by yourself because it’s always included by default in `fit.luz_module_generator()`.

**See Also**


**Examples**

```r
interrupt_callback <- luz_callback_interrupt()
```
luz_callback_lr_scheduler

Learning rate scheduler callback

Description

Initializes and runs torch::lr_scheduler()s.

Usage

luz_callback_lr_scheduler(
    lr_scheduler,
    ...,
    call_on = "on_epoch_end",
    opt_name = NULL
)

Arguments

lr_scheduler  A torch::lr_scheduler() that will be initialized with the optimizer and the ...
parameters.

...  Additional arguments passed to lr_scheduler together with the optimizers.

call_on  The callback breakpoint that scheduler$step() is called. Default is 'on_epoch_end'.
See luz_callback() for more information.

opt_name  name of the optimizer that will be affected by this callback. Should match the
name given in set_optimizers. If your module has a single optimizer, opt_name is not used.

Value

A luz_callback() generator.

See Also

Other luz_callbacks: luz_callback_csv_logger(), luz_callback_early_stopping(), luz_callback_interrupt(), luz_callback_metrics(), luz_callback_model_checkpoint(), luz_callback_profile(), luz_callback_progress(), luz_callback_train_valid(), luz_callback()
luz_callback_metrics   *Metrics callback*

**Description**

Tracks metrics passed to `setup()` during training and validation.

**Usage**

```r
luz_callback_metrics()
```

**Details**

This callback takes care of 2 `ctx` attributes:

- `ctx$metrics`: stores the current metrics objects that are initialized once for epoch, and are further `update()`d and `compute()`d every batch. You will rarely need to work with these metrics.
- `ctx$records$metrics`: Stores metrics per training/validation and epoch. The structure is very similar to `ctx$losses`.

**Value**

A `luz_callback`

**Note**

In general you won’t need to explicitly use the metrics callback as it’s used by default in `fit.luz_module_generator()`.

**See Also**


---

luz_callback_model_checkpoint   *Checkpoints model weights*

**Description**

This saves checkpoints of the model according to the specified metric and behavior.
luz_callback_model_checkpoint

Usage

luz_callback_model_checkpoint(
    path,
    monitor = "valid_loss",
    save_best_only = FALSE,
    mode = "min",
    min_delta = 0
)

Arguments

path Path to save the model on disk. The path is interpolated with glue, so you can use any attribute within the ctx by using '{ctx$epoch}'. Specially the epoch and monitor quantities are already in the environment. If the specified path is a path to a directory (ends with / or \), then models are saved with the name given by epoch-{epoch:02d}-{self$monitor}-{monitor:.3f}.pt. See more in the examples. You can use sprintf() to quickly format quantities, for example:'{epoch:02d}'.

monitor A string in the format <set>_<metric> where <set> can be 'train' or 'valid' and <metric> can be the abbreviation of any metric that you are tracking during training. The metric name is case insensitive.

save_best_only if TRUE models are only saved if they have an improvement over a previously saved model.

mode Specifies the direction that is considered an improvement. By default 'min' is used. Can also be 'max' (higher is better) and 'zero' (closer to zero is better).

min_delta Minimum difference to consider as improvement. Only used when save_best_only=TRUE.

Note

mode and min_delta are only used when save_best_only=TRUE. save_best_only will overwrite the saved models if the path parameter don’t differentiate by epochs.

See Also

Other luz_callbacks: luz_callback_csv_logger(), luz_callback_early_stopping(), luz_callback_interrupt(), luz_callback_lr_scheduler(), luz_callback_metrics(), luz_callback_profile(), luz_callback_progress(), luz_callback_train_valid(), luz_callback()

Examples

luz_callback_model_checkpoint(path= "path/to/dir")
luz_callback_model_checkpoint(path= "path/to/dir/epoch-{epoch:02d}/model.pt")
luz_callback_model_checkpoint(path= "path/to/dir/epoch-{epoch:02d}/model-{monitor:.2f}.pt")
Profile callback

Description
Computes the times for high-level operations in the training loops.

Usage
luz_callback_profile()

Details
Records are saved in ctx$records$profile. Times are stored as seconds. Data is stored in the following structure:

- `fit` time for the entire fit procedure.
- `epoch` times per epoch
- `(train/valid)_batch` time per batch of data processed, including data acquisition and step.
- `(train/valid)_step` time per step (training or validation step) - only the model step. (not including data acquisition and preprocessing)

Value
A luz_callback

Note
In general you don’t need to use these callback by yourself because it’s always included by default in fit.luz_module_generator().

See Also

Examples
profile_callback <- luz_callback_profile()
luz_callback_progress  

**Progress callback**

**Description**

Responsible for printing progress during training.

**Usage**

luz_callback_progress()

**Value**

A luz_callback

**Note**

In general you don’t need to use these callback by yourself because it’s always included by default in `fit.luz_module_generator()`. Printing can be disabled by passing `verbose=FALSE` to `fit.luz_module_generator()`.

**See Also**

Other luz_callbacks: luz_callback_csv_logger(), luz_callback_early_stopping(), luz_callback_interrupt(), luz_callback_lr_scheduler(), luz_callback_metrics(), luz_callback_model_checkpoint(), luz_callback_profile(), luz_callback_train_valid(), luz_callback()

luz_callback_train_valid  

**Train-eval callback**

**Description**

Switches important flags for training and evaluation modes.

**Usage**

luz_callback_train_valid()

**Details**

It takes care of the three ctx attributes:

- `ctx$model`: Responsible for calling `ctx$model$train()` and `ctx$model$eval()`, when appropriate.
- `ctx$training`: Sets this flag to TRUE when training and FALSE when in validation mode.
- `ctx$loss`: Resets the loss attribute to list() when finished training/ or validating.
luz_load_model_weights

Value
A luz_callback

Note
In general you won’t need to explicitly use the metrics callback as it’s used by default in `fit.luz_module_generator()`.

See Also
Other luz_callbacks: luz_callback_csv_logger(), luz_callback_early_stopping(), luz_callback_interrupt(), luz_callback_lr_scheduler(), luz_callback_metrics(), luz_callback_model_checkpoint(), luz_callback_profile(), luz_callback_progress(), luz_callback()

luz_load

Load trained model

Description
 Loads a fitted model. See documentation in luz_save().

Usage
luz_load(path)

Arguments
path path in file system so save the object.

See Also
Other luz_save: luz_save()

luz_load_model_weights

 Loads model weights into a fitted object.

Description
This can be useful when you have saved model checkpoints during training and want to reload the best checkpoint in the end.

Usage
luz_load_model_weights(obj, path, ...)
luz_save_model_weights(obj, path)
**luz_metric**

**Arguments**

- **obj**  luz object to which you want to copy the new weights.
- **path**  path to saved model in disk.
- **...**  other arguments passed to `torch_load()`.

**Value**

Returns NULL invisibly.

**Warning**

`luz_save_model_weights` operates inplace, ie modifies the model object to contain the new weights.

---

**luz_metric**  \textit{Creates a new luz metric}

**Description**

Creates a new luz metric

**Usage**

```r
luz_metric(
    name = NULL,
    ..., 
    private = NULL,
    active = NULL,
    parent_env = parent.frame(),
    inherit = NULL
)
```

**Arguments**

- **name**  string naming the new metric.
- **...**  named list of public methods. You should implement at least `initialize`, `update` and `compute`. See the details section for more information.
- **private**  An optional list of private members, which can be functions and non-functions.
- **active**  An optional list of active binding functions.
- **parent_env**  An environment to use as the parent of newly-created objects.
- **inherit**  A R6ClassGenerator object to inherit from; in other words, a superclass. This is captured as an unevaluated expression which is evaluated in `parent_env` each time an object is instantiated.
Details

In order to implement a new `luz_metric` we need to implement 3 methods:

- **initialize**: defines the metric initial state. This function is called for each epoch for both training and validation loops.

- **update**: updates the metric internal state. This function is called at every training and validation step with the predictions obtained by the model and the target values obtained from the dataloader.

- **compute**: uses the internal state to compute metric values. This function is called whenever we need to obtain the current metric value. Eg, it’s called every training step for metrics displayed in the progress bar, but only called once per epoch to record it’s value when the progress bar is not displayed.

Optionally, you can implement a `abbrev` field that gives the metric an abbreviation that will be used when displaying metric information in the console or tracking record. If no `abbrev` is passed, the class name will be used.

Let’s take a look at the implementation of `luz_metric_accuracy` so you can see how to implement a new one:

```r
luz_metric_accuracy <- luz_metric(
  # An abbreviation to be shown in progress bars, or
  # when printing progress
  abbrev = "Acc",
  # Initial setup for the metric. Metrics are initialized
  # every epoch, for both training and validation
  initialize = function() {
    self$correct <- 0
    self$total <- 0
  },
  # Run at every training or validation step and updates
  # the internal state. The update function takes `preds`
  # and `target` as parameters.
  update = function(preds, target) {
    pred <- torch::torch_argmax(preds, dim = 2)
    self$correct <- self$correct + (pred == target)$
      to(dtype = torch::torch_float())$sum()$
      item()
    self$total <- self$total + pred$numel()
  },
  # Use the internal state to query the metric value
  compute = function() {
    self$correct/self$total
  }
)
```

**Note**: It’s good practice that the compute metric returns regular R values instead of torch tensors and other parts of luz will expect that.
Value

Returns new Luz metric.

See Also

Other luz_metrics: luz_metric_accuracy(), luz_metric_binary_accuracy_with_logits(), luz_metric_binary_accuracy(), luz_metric_binary_auroc(), luz_metric_mae(), luz_metric_mse(), luz_metric_multiclass_auroc(), luz_metric_rmse()

Examples

luz_metric_accuracy <- luz_metric(
  # An abbreviation to be shown in progress bars, or
  # when printing progress
  abbrev = "Acc",
  # Initial setup for the metric. Metrics are initialized
  # every epoch, for both training and validation
  initialize = function() {
    self$correct <- 0
    self$total <- 0
  },
  # Run at every training or validation step and updates
  # the internal state. The update function takes `preds`
  # and `target` as parameters.
  update = function(preds, target) {
    pred <- torch::torch_argmax(preds, dim = 2)
    self$correct <- self$correct + (pred == target)$
    to(dtype = torch::torch_float())$sum()$item()
    self$total <- self$total + pred$numel()
  },
  # Use the internal state to query the metric value
  compute = function() {
    self$correct/self$total
  }
)

——

luz_metric_accuracy  Accuracy

Description

Computes accuracy for multi-class classification problems.

Usage

luz_metric_accuracy()
Details

This metric expects to take logits or probabilities at every update. It will then take the columnwise argmax and compare to the target.

Value

Returns new Luz metric.

See Also

Other luz_metrics: luz_metric_binary_accuracy_with_logits(), luz_metric_binary_accuracy(), luz_metric_binary_auroc(), luz_metric_mae(), luz_metric_mse(), luz_metric_multiclass_auroc(), luz_metric_rmse(), luz_metric()
luz_metric_binary_accuracy_with_logits

See Also

Other luz_metrics: luz_metric_accuracy(), luz_metric_binary_accuracy_with_logits(), luz_metric_binary_auroc(), luz_metric_mae(), luz_metric_mse(), luz_metric_multiclass_auroc(), luz_metric_rmse(), luz_metric()

Examples

if (torch::torch_is_installed()) {
library(torch)
metric <- luz_metric_binary_accuracy(threshold = 0.5)
metric <- metric$new()
metric$update(torch_rand(100), torch::torch_randint(0, 1, size = 100))
metric$compute()
}

luz_metric_binary_accuracy_with_logits

Description

Computes accuracy for binary classification problems where the model return logits. Commonly used together with torch::nn_bce_with_logits_loss().

Usage

luz_metric_binary_accuracy_with_logits(threshold = 0.5)

Arguments

threshold value used to classify observations between 0 and 1.

Details

Probabilities are generated using torch::nnf_sigmoid() and threshold is used to classify between 0 or 1.

Value

Returns new Luz metric.

See Also

Other luz_metrics: luz_metric_accuracy(), luz_metric_binary_accuracy(), luz_metric_binary_auroc(), luz_metric_mae(), luz_metric_mse(), luz_metric_multiclass_auroc(), luz_metric_rmse(), luz_metric()
Examples

```r
if (torch::torch_is_installed()) {
library(torch)
metric <- luz_metric_binary_accuracy_with_logits(threshold = 0.5)
metric <- metric$new()
metric$update(torch_randn(100), torch::torch_randint(0, 1, size = 100))
metric$compute()
}
```

luz_metric_binary_auroc

*Computes the area under the ROC*

Description

To avoid storing all predictions and targets for an epoch we compute confusion matrices across a range of pre-established thresholds.

Usage

```r
luz_metric_binary_auroc(
    num_thresholds = 200,
    thresholds = NULL,
    from_logits = FALSE
)
```

Arguments

- `num_thresholds`: Number of thresholds used to compute confusion matrices. In that case, thresholds are created by getting `num_thresholds` values linearly spaced in the unit interval.
- `thresholds`: (optional) If threshold are passed, then those are used to compute the confusion matrices and `num_thresholds` is ignored.
- `from_logits`: Boolean indicating if predictions are logits, in that case we use sigmoid to put them in the unit interval.

See Also

Other luz_metrics: `luz_metric_accuracy()`, `luz_metric_binary_accuracy_with_logits()`, `luz_metric_binary_accuracy()`, `luz_metric_mae()`, `luz_metric_mse()`, `luz_metric_multiclass_auroc()`, `luz_metric_rmse()`, `luz_metric()`
Examples

```r
if (torch::torch_is_installed()){
  library(torch)
  actual <- c(1, 1, 1, 0, 0, 0)
  predicted <- c(0.9, 0.8, 0.4, 0.5, 0.3, 0.2)

  y_true <- torch_tensor(actual)
  y_pred <- torch_tensor(predicted)

  m <- luz_metric_binary_auroc(thresholds = predicted)
  m <- m$new()
  m$update(y_pred[1:2], y_true[1:2])
  m$update(y_pred[3:4], y_true[3:4])
  m$update(y_pred[5:6], y_true[5:6])

  m$compute()
}
```

---

**luz_metric_mae**  
*Mean absolute error*

**Description**
Computes the mean absolute error.

**Usage**

```r
luz_metric_mae()
```

**Value**

Returns new Luz metric.

**See Also**

Other luz_metrics: **luz_metric_accuracy**, **luz_metric_binary_accuracy_with_logits**,  
**luz_metric_binary_accuracy**, **luz_metric_binary_auroc**, **luz_metric_mse**, **luz_metric_multiclass_auroc**,  
**luz_metric_rmse**, **luz_metric**

**Examples**

```r
if (torch::torch_is_installed()) {
  library(torch)
  metric <- luz_metric_mae()
  metric <- metric$new()
  metric$update(torch_randn(100), torch_randn(100))
  metric$compute()
}
```
**luz_metric_mse**  
*Mean squared error*

**Description**  
Computes the mean squared error

**Usage**  
luz_metric_mse()

**Value**  
A luz_metric object.

**See Also**  
Other luz_metrics: luz_metric_accuracy(), luz_metric_binary_accuracy_with_logits(), luz_metric_binary_accuracy(), luz_metric_binary_auroc(), luz_metric_mae(), luz_metric_multiclass_auroc(), luz_metric_rmse(), luz_metric()

**luz_metric_multiclass_auroc**  
*Computes the multi-class AUROC*

**Description**  
The same definition as Keras is used by default. This is equivalent to the 'micro' method in SciKit Learn too. See docs.

**Usage**  
luz_metric_multiclass_auroc(
    num_thresholds = 200,
    thresholds = NULL,
    from_logits = FALSE,
    average = c("micro", "macro", "weighted", "none")
)
**luz_metric_multiclass_auroc**

Arguments

num_thresholds  Number of thresholds used to compute confusion matrices. In that case, thresholds are created by getting num_thresholds values linearly spaced in the unit interval.

thresholds  (optional) If threshold are passed, then those are used to compute the confusion matrices and num_thresholds is ignored.

from_logits  If TRUE then we call `torch::nnf_softmax()` in the predictions before computing the metric.

average  The averaging method:

- 'micro': Stack all classes and computes the AUROC as if it was a binary classification problem.
- 'macro': Finds the AUCROC for each class and computes their mean.
- 'weighted': Finds the AUROC for each class and computes their weighted mean pondering by the number of instances for each class.
- 'none': Returns the AUROC for each class in a list.

Details

*Note* that class imbalance can affect this metric unlike the AUC for binary classification. Currently the AUC is approximated using the 'interpolation' method described in Keras.

See Also

Other luz_metrics: `luz_metric_accuracy()`, `luz_metric_binary_accuracy_with_logits()`, `luz_metric_binary_accuracy()`, `luz_metric_binary_auroc()`, `luz_metric_mae()`, `luz_metric_mse()`, `luz_metric_rmse()`, `luz_metric()`

Examples

```r
if (torch::torch_is_installed()) {
  library(torch)
  actual <- c(1, 1, 0, 0, 0) + 1L
  predicted <- c(0.9, 0.8, 0.4, 0.5, 0.3, 0.2)
  predicted <- cbind(1-predicted, predicted)
  y_true <- torch_tensor(as.integer(actual))
  y_pred <- torch_tensor(predicted)

  m <- luz_metric_multiclass_auroc(thresholds = as.numeric(predicted),
                                   average = "micro")
  m <- m$new()
  m$update(y_pred[1:2,], y_true[1:2])
  m$update(y_pred[3:4,], y_true[3:4])
  m$update(y_pred[5:6,], y_true[5:6])
  m$compute()
}
```
luz_metric_rmse  
Root mean squared error

Description
Computes the root mean squared error.

Usage
luz_metric_rmse()

Value
Returns new Luz metric.

See Also
Other luz_metrics: luz_metric_accuracy(), luz_metric_binary_accuracy_with_logits(), luz_metric_binary_accuracy(), luz_metric_binary_auroc(), luz_metric_mae(), luz_metric_mse(), luz_metric_multiclass_auroc(), luz_metric()

luz_save  
Saves luz objects to disk

Description
Allows saving luz fitted models to the disk. Objects can be loaded back with luz_load().

Usage
luz_save(obj, path, ...)

Arguments
obj  
an object of class 'luz_module_fitted' as returned by fit.luz_module_generator().

path  
path in file system so save the object.

...  
currently unused.

Warning
The ctx is naively serialized. Ie, we only use saveRDS() to serialize it. Don’t expect luz_save to work correctly if you have unserializable objects in the ctx like torch_tensors and external pointers in general.
Note

Objects are saved as plain .rds files but obj$model is serialized with torch_save before saving it.

See Also

Other luz_save: luz_load()

setup

Set’s up a nn_module to use with luz

Description

The setup function is used to set important attributes and method for nn_modules to be used with Luz.

Usage

setup(module, loss = NULL, optimizer = NULL, metrics = NULL)

Arguments

module (nn_module) The nn_module that you want set up.

loss (function, optional) An optional function with the signature function(input, target). It’s only requires if your nn_module doesn’t implement a method called loss.

optimizer (torch_optimizer, optional) A function with the signature function(parameters, ...) that is used to initialize an optimizer given the model parameters.

metrics (list, optional) A list of metrics to be tracked during the training procedure.

Details

It makes sure the module have all the necessary ingredients in order to be fitted.

Value

A luz module that can be trained with fit().
**set_hparams**  
*Set hyper-parameter of a module*

**Description**

This function is used to define hyper-parameters before calling `fit` for `luz_modules`.

**Usage**

```python
set_hparams(module, ...)
```

**Arguments**

- `module`: An `nn_module` that has been `setup()`.  
- `...`: The parameters set here will be used to initialize the `nn_module`, i.e., they are passed unchanged to the `initialize` method of the base `nn_module`.

**Value**

The same `luz` module

**See Also**

Other `set_hparam`: `set_opt_hparams()`

---

**set_opt_hparams**  
*Set optimizer hyper-parameters*

**Description**

This function is used to define hyper-parameters for the optimizer initialization method.

**Usage**

```python
set_opt_hparams(module, ...)
```

**Arguments**

- `module`: An `nn_module` that has been `setup()`.  
- `...`: The parameters passed here will be used to initialize the optimizers. For example, if your optimizer is `optim_adam` and you pass `lr=0.1`, then the `optim_adam` function is called with `optim_adam(parameters, lr=0.1)` when fitting the model.

**Value**

The same `luz` module
See Also

Other set_hparam: `set_hparams()`
Index

* luz_callbacks
  luz_callback, 7
  luz_callback_csv_logger, 10
  luz_callback_early_stopping, 11
  luz_callback_interrupt, 12
  luz_callback_lr_scheduler, 13
  luz_callback_metrics, 14
  luz_callback_model_checkpoint, 14
  luz_callback_profile, 16
  luz_callback_progress, 17
  luz_callback_train_valid, 17

* luz_metrics
  luz_metric, 19
  luz_metric_accuracy, 21
  luz_metric_binary_accuracy, 22
  luz_metric_binary_accuracy_with_logits, 23
  luz_metric_binary_auroc, 24
  luz_metric_mae, 25
  luz_metric_mse, 26
  luz_metric_multiclass_auroc, 26
  luz_metric_rmse, 28

* luz_save
  luz_load, 18
  luz_save, 28

* set_hparam
  set_hparams, 30
  set_opt_hparams, 30

* training
  setup, 29

accelerator, 2
accelerator(), 7

ctx, 3, 5, 14, 15, 28

fit(), 29
fit.luz_module_generator, 6
fit.luz_module_generator(), 9, 11, 12, 14, 16–18, 28

interactive(), 5, 7

luz_callback, 7, 10–18
luz_callback(), 7, 13
luz_callback_csv_logger, 10, 10, 11–18
luz_callback_early_stopping, 10, 11, 12–18
luz_callback_interrupt, 10, 11, 12, 13–18
luz_callback_lr_scheduler, 10–12, 13, 14–18
luz_callback_metrics, 10–13, 14, 15–18
luz_callback_metrics(), 7
luz_callback_model_checkpoint, 10–14, 14, 16–18
luz_callback_profile, 10–15, 16, 17, 18
luz_callback_progress, 10–16, 17, 18
luz_callback_progress(), 7
luz_callback_train_valid, 10–17, 17
luz_callback_train_valid(), 7
luz_load, 18, 29
luz_load(), 28
luz_load_model_weights, 18
luz_metric, 19, 22–28
luz_metric_accuracy, 21, 21, 23–28
luz_metric_binary_accuracy, 21, 22, 22, 23–28
luz_metric_binary_accuracy_with_logits, 21–23, 23, 24–28
luz_metric_binary_auroc, 21–23, 24, 25–28
luz_metric_mae, 21–24, 25, 26–28
luz_metric_mse, 21–25, 26, 27, 28
luz_metric_multiclass_auroc, 21–26, 26, 28
luz_metric_rmse, 21–27, 28
luz_save, 18, 28
luz_save(), 7, 18

32
luz_save_model_weights
   (luz_load_model_weights). 18

nn_module, 7

plot(). 7
print(). 7

saveRDS(). 28
set_hpparams, 30, 31
set_opt_hpparams, 30, 30
setup, 29
setup(), 7, 14, 30
sprintf(), 15

torch::dataloader(). 7
torch::lr_scheduler(). 13
torch::nn_bce_loss(). 22
torch::nn_bce_with_logits_loss(). 23
torch::nnf_sigmoid(). 23
torch::nnf_softmax(). 27
torch_load(). 19