Package ‘luz’

October 7, 2021

Title Higher Level 'API' for 'torch'

Version 0.2.0

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 (>= 0.5.0), 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' 'as_dataloader.R' 'utils.R' 'callbacks.R'
    'callbacks-interrupt.R' 'callbacks-monitor-metrics.R'
    'callbacks-profile.R' 'context.R' 'lr-finder.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 2021-10-07 20:50:02 UTC
## R topics documented:

- `accelerator` .......................................................... 2
- `as_dataloader` ...................................................... 3
- `context` ............................................................... 4
- `ctx` ........................................................................ 8
- `evaluate` ................................................................. 9
- `fit.luz_module_generator` ........................................... 11
- `get_metrics` ............................................................ 12
- `lr_finder` ............................................................... 13
- `luz_callback` ........................................................... 14
- `luz_callback_csv_logger` ............................................ 16
- `luz_callback_early_stopping` ..................................... 17
- `luz_callback_interrupt` .............................................. 18
- `luz_callback_lr_scheduler` ......................................... 19
- `luz_callback_metrics` ................................................ 20
- `luz_callback_model_checkpoint` ................................... 20
- `luz_callback_profile` ............................................... 22
- `luz_callback_progress` .............................................. 23
- `luz_callback_train_valid` .......................................... 23
- `luz_load` ............................................................... 24
- `luz_load_model_weights` ........................................... 24
- `luz_metric` ............................................................. 25
- `luz_metric_accuracy` ................................................ 27
- `luz_metric_binary_accuracy` ........................................ 28
- `luz_metric_binary_accuracy_with_logits` ...................... 29
- `luz_metric_binary_auroc` .......................................... 30
- `luz_metric_mae` .................................................... 31
- `luz_metric_mse` .................................................... 32
- `luz_metric_multiclass_auroc` ..................................... 32
- `luz_metric_rmse` .................................................... 34
- `luz_save` ............................................................. 34
- `predict.luz_module_fitted` ........................................ 35
- `setup` ...................................................................... 36
- `set_hparams` .......................................................... 37
- `set_opt_hparams` ..................................................... 37

### Index

<table>
<thead>
<tr>
<th>accelerator</th>
<th>Create an accelerator</th>
</tr>
</thead>
</table>

**Description**

Create an accelerator
Usage

accelerator(
    device_placement = TRUE,
    cpu = FALSE,
    cuda_index = torch::cuda_current_device()
)

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.

cuda_index
    (integer) index of the CUDA device to use if multiple GPUs are available. Default: the result of torch::cuda_current_device().

---

as_dataloader

Creates a dataloader from its input

Description

as_dataloader is used internally by luz to convert input data and valid_data as passed to fit.luz_module_generator() to a torch::dataloader

Usage

as_dataloader(x, ...)

## S3 method for class 'dataset'
as_dataloader(x, ..., batch_size = 32)

## S3 method for class 'list'
as_dataloader(x, ...)

## S3 method for class 'dataloader'
as_dataloader(x, ...)

## S3 method for class 'matrix'
as_dataloader(x, ...)

## S3 method for class 'numeric'
as_dataloader(x, ...)

## S3 method for class 'array'
as_dataloader(x, ...)

## S3 method for class 'torch_tensor'
as_dataloader(x, ...)
Arguments

x      the input object.
...    Passed to torch::dataloader().
batch_size (int, optional): how many samples per batch to load (default: 1).

Details

as_dataloader methods should have sensible defaults for batch_size, parallel workers, etc.
It allows users to quickly experiment with fit.luz_module_generator() by not requiring to create a torch::dataset and a torch::dataloader in simple experiments.

Methods (by class)

• dataset: Converts a torch::dataset() to a torch::dataloader().
• list: Converts a list of tensors or arrays with the same size in the first dimension to a torch::dataloader()
• dataloader: Returns the same dataloader
• matrix: Converts the matrix to a dataloader
• numeric: Converts the numeric vector to a dataloader
• array: Converts the array to a dataloader
• torch_tensor: Converts the tensor to a dataloader

Overriding

You can implement your own as_dataloader S3 method if you want your data structure to be automatically supported by luz's fit.luz_module_generator(). The method must satisfy the following conditions:

• The method should return a torch::dataloader().
• The only required argument is x. You have good default for all other arguments.

It's better to avoid implementing as_dataloader methods for common S3 classes like data.frames. In this case, it's better to assign a different class to the inputs and implement as_dataloader for it.

---

context            Context object

Description

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

Public fields

buffers This is a list of buffers that callbacks can use to write temporary information into ctx.
Active bindings

- **records** stores information about values logged with `self$log`.
- **device** allows querying the current accelerator device.
- **callbacks** list of callbacks that will be called.
- **iter** current iteration
- **batch** the current batch data. A list with input data and targets.
- **input** a shortcut for `ctx$batch[[1]]`
- **target** a shortcut for `ctx$batch[[2]]`
- **min_epochs** the minimum number of epochs that the model will run on.
- **max_epochs** the maximum number of epochs that the model will run.
- **hparams** a list of hyperparameters that were used to initialize `ctx$model`.
- **opt_hparams** a list of hyperparameters used to initialize the `ctx$optimizers`.
- **train_data** a dataloader that is used for training the model.
- **valid_data** a dataloader using during model validation
- **accelerator** an `accelerator()` used to move data, model and etc the the correct device.
- **optimizers** a named list of optimizers that will be used during model training.
- **verbose** bool whether the process is in verbose mode or not.
- **handlers** List of error handlers that can be used. See `rlang::with_handlers()` for more info.
- **training** A bool indicating if the model is in training or validation mode.
- **model** The model being trained.
- **pred** Last predicted values.
- **opt** Current optimizer.
- **opt_name** Current optimizer name.
- **data** Current dataloader in use.
- **loss** Last computed loss values. Detached from the graph.
- **loss_grad** Last computed loss value, not detached, so you can do additional transformation.
- **epoch** Current epoch.
- **metrics** List of metrics that are tracked by the process.

Methods

Public methods:

- `context$new()`
- `context$log()`
- `context$log_metric()`
- `context$get_log()`
- `context$get_metrics()`
- `context$get_metric()`
- `context$get_formatted_metrics()`
• context$get_metrics_df()
• context$set_verbose()
• context$clean()
• context$call_callbacks()
• context$state_dict()
• context$clone()

Method new(): Initializes the context object with minimal necessary information.

Usage:
context$new(verbose, accelerator, callbacks, training)

Arguments:
verbose Whether the context should be in verbose mode or not.
accelerator A luz accelerator() that configures device placement and others.
callbacks A list of callbacks used by the model. See luz_callback().
training A boolean that indicates if the context is in training mode or not.

Method log(): Allows logging arbitrary information in the ctx.

Usage:
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
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
value Arbitrary value to log.

Method get_log(): Get a 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 log 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 log 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 log 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 log 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:*
context$get_metrics_df()

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

*Usage:*
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** clean(): Removes unnecessary information from the context object.

*Usage:*

ctx

Method clean(): Call the selected callbacks. Where name is the callback types to call, eg `on_epoch_begin`.

Usage:
`context$call_callbacks(name)`

Arguments:
name name of the metric

Method state_dict(): Returns a list containing minimal information from the context. Used to create the returned values.

Usage:
`context$state_dict()`

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

Usage:
`context$clone(deep = FALSE)`

Arguments:
deep Whether to make a deep clone.

---

# ctx

## Context object

### Description

Context objects used in luz to share information between model methods, metrics and callbacks.

### Details

The ctctx object is used in luz to share information between the training loop and callbacks, model methods and metrics. The table below describes information available in the ctx by default. Other callbacks could potentially modify these attributes or add new ones.

<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 in milieu.</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>Current in use dataloader. When training it’s ctx$train_data, when doing validation its ctx$valid_data.</td>
</tr>
<tr>
<td>train_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>min_epochs</td>
<td>Minimum number of epochs the model will be trained for.</td>
</tr>
<tr>
<td>max_epochs</td>
<td>Maximum number of epochs the model will be trained for.</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>
</tbody>
</table>
evaluate

Evaluates a fitted model on a dataset

Description

Evaluates a fitted model on a dataset

Usage

```r
evaluate(
  object,
  data,
  ...
  callbacks = list(),
  accelerator = NULL,
  verbose = NULL,
  dataloader_options = NULL
)
```

Arguments

- `object` A fitted model to evaluate.
Evaluate returns a `luz_module_evaluation` object that you can query for metrics using the `get_metrics` function or simply print to see the results.

For example:

```r
evaluation <- fitted %>% evaluate(data = valid_dl)
metrics <- get_metrics(evaluation)
print(evaluation)
```

## A `luz_module_evaluation`
## -- Results -----------------------------------------------
## loss: 1.5146
## mae: 1.0251
## mse: 1.5159
## rmse: 1.2312

### Details

Once a model has been trained you might want to evaluate its performance in a different dataset. For that reason, luz provides the `evaluate` function that takes a fitted model and a dataset and computes the metrics attached to the model.

Evaluate returns a `luz_module_evaluation` object that you can query for metrics using the `get_metrics` function or simply print to see the results.

For example:

```r
evaluation <- fitted %>% evaluate(data = valid_dl)
metrics <- get_metrics(evaluation)
print(evaluation)
```

## See Also

Other training: `fit.luz_module_generator()`, `predict.luz_module_fitted()`, `setup()`
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,
  ...
)
```

Arguments

- **object**: An nn_module that has been `setup()`. (dataloader, dataset or list) A dataloader created with `torch::dataloader()` used for training the model, or a dataset created with `torch::dataset()` or a list. Dataloaders and datasets must return 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 maximum number of epochs for training the model. If a single value is provided, this is taken to be the `max_epochs` and `min_epochs` is set to 0. If a vector of two numbers is provided, the first value is `min_epochs` and the second value is `max_epochs`. The minimum and maximum number of epochs are included in the context object as `ctx$min_epochs` and `ctx$max_epochs`, respectively.

- **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, dataset, list or scalar value; optional) A dataloader created with `torch::dataloader()` or a dataset created with `torch::dataset()` that will be used during the validation procedure. They must return a list with (input, target). If `data` is a torch dataset or a list, then you can also supply a numeric value between 0 and 1 - and in this case a random sample with size corresponding to that proportion from `data` will be used for validation.
get_metrics

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.

dataloader_options Options used when creating a dataloader. See torch::dataloader(). shuffle=TRUE by default for the training data and batch_size=32 by default. It will error if not NULL and data is already a dataloader.

Value

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

See Also

predict.luz_module_fitted() for how to create predictions. setup() to find out how to create modules that can be trained with fit.

Other training: evaluate(), predict.luz_module_fitted(), setup

get_metrics

Description

Get metrics from the object

Usage

get_metrics(object, ...)

## S3 method for class 'luz_module_fitted'
get_metrics(object, ...)

Arguments

object The object to query for metrics.

... Currently unused.

Value

A data.frame containing the metric values.
Methods (by class)

- luz_module_fitted: Extract metrics from a luz fitted model.

---

### lr_finder

**Description**

Learning Rate Finder

**Usage**

```r
lr_finder(object, data, steps = 100, start_lr = 1e-07, end_lr = 0.1, ...)
```

**Arguments**

- `object`: An nn_module that has been setup().
- `data`: (dataloader) A dataloader created with torch::dataloader() used for learning rate finding.
- `steps`: (integer) The number of steps to iterate over in the learning rate finder. Default: 100.
- `end_lr`: (float) The highest learning rate. Default: 1e-1.
- `...`: Other arguments passed to `fit`.

**Value**

A dataframe with two columns: learning rate and loss

**Examples**

```r
if (torch::torch_is_installed()) {
  library(torch)
  ds <- torch::tensor_dataset(x = torch_randn(100, 10), y = torch_randn(100, 1))
  dl <- torch::dataloader(ds, batch_size = 32)
  model <- torch::nn_linear
  model <- model %>% setup(
    loss = torch::nn_mse_loss(),
    optimizer = torch::optim_adam
  ) %>%
    set_hparams(in_features = 10, out_features = 1)
  records <- lr_finder(model, dl, verbose = FALSE)
  plot(records)
}
```
luz_callback

Create a new callback

Description

Create a new callback

Usage

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:

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
```
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.

By default, callbacks are called in the same order as they were passed to fit (predict or evaluate) but you can provide a weight attribute that will control the order that it will be called. For example if a callback has weight = 10 and the other has weight = 1 then the first one is called after the second one. Callbacks that don’t specify a weight attribute are considered weight = 0. A few built-in callbacks in luz already provide a weight value, for example the ?luz_early_stopping_callback, since in general we want to run it as the last thing in the loop.

Value

A luz_callback that can be passed 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_progress(), luz_callback_train_valid()

Examples

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

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

**Usage**

```
luz_callback_csv_logger(path)
```

**Arguments**

- `path`  
  path to a file on disk.

**See Also**


---

**Description**

Early stopping callback

**Usage**

```
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 as soon as the model stops training.

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

See Also

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

Examples

cb <- luz_callback_early_stopping()
luz_callback_lr_scheduler

Learning rate scheduler callback

Description

Initializes and runs `torch::lr_scheduler()`s.

Usage

```r
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 ...
- `...` 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


Examples

```r
if (torch::torch_is_installed()) {
  cb <- luz_callback_lr_scheduler(torch::lr_step, step_size = 30)
}
```
## 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 updated and computed 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
Other `luz_callback`s: `luz_callback_csv_logger()`, `luz_callback_early_stopping()`, `luz_callback_interrupt()`, `luz_callback_lr_scheduler()`, `luz_callback_model_checkpoint()`, `luz_callback_profile()`, `luz_callback_progress()`, `luz_callback_train_valid()`, `luz_callback()`

## luz_callback_model_checkpoint  Checkpoints model weights

### Description
This saves checkpoints of the model according to the specified metric and behavior.
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")
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.
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**

*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()
luz_metric_binary_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()

Examples

if (torch::torch_is_installed()) {
  library(torch)
  metric <- luz_metric_accuracy()
  metric <- metric$new()
  metric$update(torch_randn(100, 10), torch::torch_randint(1, 10, size = 100))
  metric$compute()
}

luz_metric_binary_accuracy

Binary accuracy

Description

Computes the accuracy for binary classification problems where the model returns probabilities. Commonly used when the loss is torch::nn_bce_loss().

Usage

luz_metric_binary_accuracy(threshold = 0.5)

Arguments

threshold value used to classify observations between 0 and 1.

Value

Returns new 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

```r
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**

*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

```r
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()`
luz_metric_mae

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_mae()
  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**

```
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**

```r
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**

```r
luz_metric_multiclass_auroc(
  num_thresholds = 200,
  thresholds = NULL,
  from_logits = FALSE,
  average = c("micro", "macro", "weighted", "none")
)
```
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 thresholds 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()
### luz_metric_rmse

**Description**

Computes the root mean squared error.

**Usage**

```r
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

**Description**

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

**Usage**

```r
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. I.e., 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_tensor` 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()`

---

**predict.luz_module_fitted**

Create predictions for a fitted model

**Description**

Create predictions for a fitted model

**Usage**

```r
## S3 method for class 'luz_module_fitted'
predict(
  object,
  newdata,
  ...
  callbacks = list(),
  accelerator = NULL,
  verbose = NULL,
  dataloader_options = NULL
)
```

**Arguments**

- `object` (fitted model) the fitted model object returned from `fit.luz_module_generator()`
- `newdata` (dataloader, dataset, list or array) returning a list with at least 1 element. The other elements aren’t used.
- `...` Currently unused.
- `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.
- `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.
setup

dataloader_options
Options used when creating a dataloader. See `torch::dataloader()`. shuffle=TRUE by default for the training data and batch_size=32 by default. It will error if not NULL and data is already a dataloader.

See Also
Other training: `evaluate()`, `fit.luz_module_generator()`, `setup()`

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()`.

See Also
Other training: `evaluate()`, `fit.luz_module_generator()`, `predict.luz_module_fitted()`
set_hparams

*Set hyper-parameter of a module*

---

**Description**

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

**Usage**

```r
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**

```r
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, 14
  luz_callback_csv_logger, 16
  luz_callback_early_stopping, 17
  luz_callback_interrupt, 18
  luz_callback_lr_scheduler, 19
  luz_callback_metrics, 20
  luz_callback_model_checkpoint, 20
  luz_callback_profile, 22
  luz_callback_progress, 23
  luz_callback_train_valid, 23

* **luz_metrics**
  luz_metric, 25
  luz_metric_accuracy, 27
  luz_metric_binary_accuracy, 28
  luz_metric_binary_accuracy_with_logits, 29
  luz_metric_binary_auroc, 30
  luz_metric_mae, 31
  luz_metric_mse, 32
  luz_metric_multiclass_auroc, 32
  luz_metric_rmse, 34

* **luz_save**
  luz_load, 24
  luz_save, 34

* **set_hparam**
  set_hparams, 37
  set_opt_hparams, 37

* **training**
  evaluate, 9, 12, 36
  fit(), 36
  fit.luz_module_generator, 10, 11, 36
  fit.luz_module_generator(), 3, 4, 16, 18, 20, 22–24, 34, 35
  get_metrics, 12
  interactive(), 7, 10, 12, 35
  lr_finder, 13
  luz_callback, 14, 17–24
  luz_callback(), 6, 10, 11, 19, 35
  luz_callback_csv_logger, 16, 16, 18–24
  luz_callback_early_stopping, 16, 17, 17, 18–24
  luz_callback_interrupt, 16–18, 18, 19–24
  luz_callback_lr_scheduler, 16–18, 19, 20–24
  luz_callback_metrics, 16–19, 20, 21–24
  luz_callback_metrics(), 10, 11, 35
  luz_callback_model_checkpoint, 16–20, 20, 22–24
  luz_callback_profile, 16–21, 22, 23, 24
  luz_callback_progress, 16–22, 23, 24
  luz_callback_progress(), 10, 11, 35
  luz_callback_train_valid, 16–23, 23
  luz_callback_train_valid(), 10, 11, 35
  luz_load, 24, 35
  luz_load(), 34
  luz_load_model_weights, 24
  luz_metric, 25, 28–34
  luz_metric_accuracy, 27, 27, 29–34
  luz_metric_binary_accuracy, 27, 28, 28, 29–34
  luz_metric_binary_accuracy_with_logits, 27–29, 29, 30–34
  luz_metric_binary_auroc, 27–29, 30, 31–34

accelerator, 2
accelerator(), 5, 6, 10, 12, 35
as_dataloader, 3
context, 4, 9
ctx, 4, 8, 20, 21, 34
luz_metric_mae, 27–30, 31, 32–34
luz_metric_mse, 27–31, 32, 33, 34
luz_metric_multiclass_auroc, 27–32, 32, 34
luz_metric_rmse, 27–33, 34
luz_save, 24, 34
luz_save(), 12, 24
luz_save_model_weights
   (luz_load_model_weights), 24

nn_module, 10, 12, 35

plot(), 12
predict.luz_module_fitted, 10, 12, 35, 36
predict.luz_module_fitted(), 12
print(), 12

rlang::with_handlers(), 5

saveRDS(), 34
set_hpargs, 37, 38
set_opt_hpargs, 37, 37
setup, 10, 12, 36, 36
setup(), 11, 12, 20, 37
sprintf(), 21

torch::dataloader, 3, 4
torch::dataloader(), 4, 10–12, 36
torch::dataset, 4
torch::dataset(), 4, 10, 11
torch::lr_scheduler(), 19
torch::nn_bce_loss(), 28
torch::nn_bce_with_logits_loss(), 29
torch::nnf_sigmoid(), 29
torch::nnf_softmax(), 33
torch_load(), 25