

Package ‘luz’

July 22, 2025

Title Higher Level 'API' for 'torch'

Version 0.4.0

Description A high level interface for 'torch' providing utilities to reduce 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) <[doi:10.48550/arXiv.2002.04688](https://doi.org/10.48550/arXiv.2002.04688)>, 'Keras' by Chollet et al. (2015) and 'PyTorch Lightning' by Falcon et al. (2019) <[doi:10.5281/zenodo.3828935](https://doi.org/10.5281/zenodo.3828935)>.

License MIT + file LICENSE

URL <https://mlverse.github.io/luz/>, <https://github.com/mlverse/luz>

Encoding UTF-8

RoxygenNote 7.2.3

Imports torch (>= 0.9.0), magrittr, zeallot, rlang (>= 1.0.0), coro, glue, progress, R6, generics, purrr, ellipsis, fs, prettyunits, cli

Suggests knitr, rmarkdown, testthat (>= 3.0.0), covr, Metrics, withr, vdiff, ggplot2 (>= 3.0.0), dplyr, torchvision, tfevents (>= 0.0.2), tidyr

VignetteBuilder knitr

Config/testthat/edition 3

Collate 'accelerator.R' 'as_dataloader.R' 'utils.R' 'callbacks.R' 'callbacks-interrupt.R' 'callbacks-mixup.R' 'callbacks-monitor-metrics.R' 'callbacks-profile.R' 'callbacks-resume.R' 'callbacks-tfevents.R' 'context.R' 'losses.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 2023-04-17 10:00:05 UTC

Contents

accelerator	3
as_dataloader	3
context	5
ctx	9
evaluate	10
fit.luz_module_generator	11
get_metrics	13
lr_finder	13
luz_callback	15
luz_callback_auto_resume	18
luz_callback_csv_logger	20
luz_callback_early_stopping	21
luz_callback_gradient_clip	22
luz_callback_interrupt	22
luz_callback_keep_best_model	23
luz_callback_lr_scheduler	24
luz_callback_metrics	25
luz_callback_mixup	25
luz_callback_model_checkpoint	27
luz_callback_profile	28
luz_callback_progress	29
luz_callback_resume_from_checkpoint	29
luz_callback_tfevents	30
luz_callback_train_valid	31
luz_load	32
luz_load_checkpoint	32
luz_load_model_weights	33
luz_metric	34
luz_metric_accuracy	36
luz_metric_binary_accuracy	37
luz_metric_binary_accuracy_with_logits	38
luz_metric_binary_auroc	39
luz_metric_mae	40
luz_metric_mse	41
luz_metric_multiclass_auroc	41
luz_metric_rmse	43
luz_metric_set	43
luz_save	44
nnf_mixup	44
nn_mixup_loss	45
predict.luz_module_fitted	46
setup	47

<i>accelerator</i>	3
set_hparams	48
set_opt_hparams	49
Index	50

<i>accelerator</i>	<i>Create an accelerator</i>
--------------------	------------------------------

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().

<i>as_dataloader</i>	<i>Creates a dataloader from its input</i>
----------------------	--

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 <code>torch::dataloader()</code> .
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)

- `as_dataloader(dataset)`: Converts a `torch::dataset()` to a `torch::dataloader()`.
- `as_dataloader(list)`: Converts a list of tensors or arrays with the same size in the first dimension to a `torch::dataloader()`
- `as_dataloader(dataloader)`: Returns the same dataloader
- `as_dataloader(matrix)`: Converts the matrix to a dataloader
- `as_dataloader(numeric)`: Converts the numeric vector to a dataloader
- `as_dataloader(array)`: Converts the array to a dataloader
- `as_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, its 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 wether the process is in verbose mode or not.

`handlers` List of error handlers that can be used. See [rlang::try_fetch\(\)](#) for more info.

`epoch_handlers` List of error handlers that can be used. See [rlang::try_fetch\(\)](#) 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_fn` Loss function used to train the model
`loss` Last computed loss values. Detached from the graph.
`loss_grad` Last computed loss value, not detached, so you can do additional tranformation.
`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$unsafe_set_records()`
- `context$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
`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 given 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

index 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 look 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 metrics 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 look 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 look 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 look 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:

```
context$clean()
```

Method `call_callbacks()`: 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 `unsafe_set_records()`: Are you sure you know what you are doing?

Usage:

```
context$unsafe_set_records(records)
```

Arguments:

records New set of records to be set.

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	<i>Context object</i>
-----	-----------------------

Description

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

Details

The ctx 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.

Attribute	Description
verbose	The value (TRUE or FALSE) attributed to the verbose argument in <code>fit</code> .
accelerator	Accelerator object used to query the correct device to place models, data, etc. It assumes the value passed
model	Initialized <code>nn_module</code> object that will be trained during the <code>fit</code> procedure.
optimizers	A named list of optimizers used during training.
data	The currently in-use dataloader. When training it's <code>ctx\$train_data</code> , when doing validation its <code>ctx\$valid_data</code> .
train_data	Dataloader passed to the <code>data</code> argument in <code>fit</code> . Modified to yield data in the selected device.
valid_data	Dataloader passed to the <code>valid_data</code> argument in <code>fit</code> . Modified to yield data in the selected device.
min_epochs	Minimum number of epochs the model will be trained for.
max_epochs	Maximum number of epochs the model will be trained for.
epoch	Current training epoch.
iter	Current training iteration. It's reset every epoch and when going from training to validation.
training	Whether the model is in training or validation mode. See also <code>help("luz_callback_train_valid")</code>
callbacks	List of callbacks that will be called during the training procedure. It's the union of the list passed to the <code>callbacks</code>
step	Closure that will be used to do one step of the model. It's used for both training and validation. Takes no
call_callbacks	Call callbacks by name. For example <code>call_callbacks("on_train_begin")</code> will call all callbacks that p
batch	Last batch obtained by the dataloader. A batch is a <code>list()</code> with 2 elements, one that is used as input and
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 <code>ctx\$model\$forward</code> . Note: can be potentially modified by previously ran c
loss_fn	The active loss function that will be minimized during training.
loss	Last computed loss from the model. Note: this might not be available if you modified the training or valid
opt	Current optimizer, ie. the optimizer that will be used to do the next step to update parameters.
opt_nm	Current optimizer name. By default it's <code>opt</code> , but can change if your model uses more than one optimizer
metrics	<code>list()</code> with current metric objects that are updated at every <code>on_train_batch_end()</code> or <code>on_valid_batch_end()</code>
records	<code>list()</code> recording metric values for training and validation for each epoch. See also <code>help("luz_callback")</code>
handlers	A named <code>list()</code> of handlers that is passed to <code>rlang::with_handlers()</code> during the training loop and car
epoch_handlers	A named list of handlers that is used with <code>rlang::with_handlers()</code> . Those handlers are used inside the

Context attributes

See Also

Context object: [context](#)

evaluate	<i>Evaluates a fitted model on a dataset</i>
----------	--

Description

Evaluates a fitted model on a dataset

Usage

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

Arguments

object	A fitted model to evaluate.
data	(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 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 a target for the loss function.
...	Currently unused.
metrics	A list of luz metrics to be tracked during evaluation. If NULL (default) then the same metrics that were used during training are tracked.
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.

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.

Details

Once a model has been trained you might want to evaluate its performance on 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:

```
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
```

See Also

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

```
fit.luz_module_generator
```

```
Fit a nn_module
```

Description

Fit a `nn_module`

Usage

```
## S3 method for class 'luz_module_generator'
fit(
  object,
  data,
  epochs = 10,
  callbacks = NULL,
  valid_data = NULL,
  accelerator = NULL,
```

```

    verbose = NULL,
    ...,
    dataloader_options = NULL
)

```

Arguments

object	An nn_module that has been setup() .
data	(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 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 a 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.
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`*Get metrics from the object*

Description

Get metrics from the object

Usage

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

Arguments

<code>object</code>	The object to query for metrics.
<code>...</code>	Currently unused.

Value

A data.frame containing the metric values.

Methods (by class)

- `get_metrics(luz_module_fitted)`: Extract metrics from a luz fitted model.

`lr_finder`*Learning Rate Finder*

Description

Learning Rate Finder

Usage

```
lr_finder(
  object,
  data,
  steps = 100,
  start_lr = 1e-07,
  end_lr = 0.1,
  log_spaced_intervals = TRUE,
  ...,
  verbose = NULL
)
```

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.
start_lr	(float) The smallest learning rate. Default: 1e-7.
end_lr	(float) The highest learning rate. Default: 1e-1.
log_spaced_intervals	(logical) Whether to divide the range between start_lr and end_lr into log-spaced intervals (alternative: uniform intervals). Default: TRUE
...	Other arguments passed to fit.
verbose	Whether to show a progress bar during the process.

Value

A dataframe with two columns: learning rate and loss

Examples

```
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	<i>Create a new callback</i>
--------------	------------------------------

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 named functions as ... arguments, where the name indicates 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 where you want to log the results. In that 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:

```

fitted <- net %>%
  setup(...) %>%
  fit(..., callbacks = list(
    print_callback(message = "Done!")
  ))

```

Callbacks can be called in many different positions of the training loop, including 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 `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` (or `predict` or `evaluate`), but you can provide a `weight` attribute that will control the order in which it will be called. For example, if one callback has `weight = 10` and another 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_callback_early_stopping` has a `weight` of `Inf`, 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()`.

Prediction callbacks

You can also use callbacks when using `predict()`. In this case the supported callback methods are detailed above.

```

Start predict
- on_predict_begin
Start prediction loop
- on_predict_batch_begin
- on_predict_batch_end
End prediction loop
- on_predict_end
End predict

```

Evaluate callbacks

Callbacks can also be used with `evaluate()`, in this case, the callbacks that are used are equivalent to those of the validation loop when using `fit()`:

```

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

```

See Also

Other luz_callbacks: [luz_callback_auto_resume\(\)](#), [luz_callback_csv_logger\(\)](#), [luz_callback_early_stopping\(\)](#), [luz_callback_interrupt\(\)](#), [luz_callback_keep_best_model\(\)](#), [luz_callback_lr_scheduler\(\)](#), [luz_callback_metrics\(\)](#), [luz_callback_mixup\(\)](#), [luz_callback_model_checkpoint\(\)](#), [luz_callback_profile\(\)](#), [luz_callback_progress\(\)](#), [luz_callback_resume_from_checkpoint\(\)](#), [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")
  }
)

```

`luz_callback_auto_resume`

Resume training callback

Description

This callback allows you to resume training a model.

Usage

```
luz_callback_auto_resume(path = "./state.pt")
```

Arguments

`path` Path to save state files for the model.

Details

When using it, model weights, optimizer state are serialized at the end of each epoch. If something fails during training simply re-running the same script will restart the model training from the epoch right after the last epoch that was serialized.

Customizing serialization

By default model, optimizer state and records are serialized. Callbacks can be used to customize serialization by implementing the `state_dict()` and `load_state_dict()` methods. If those methods are implemented, then `state_dict()` is called at the end of each epoch and `load_state_dict()` is called when the model is resumed.

Note

In general you will want to add this callback as the last in the callbacks list, this way, the serialized state is likely to contain all possible changes that other callbacks could have made at 'on_epoch_end'. The default weight attribute of this callback is Inf.

Read the checkpointing article in the pkgdown website for more information.

See Also

Other luz_callbacks: `luz_callback_csv_logger()`, `luz_callback_early_stopping()`, `luz_callback_interrupt()`, `luz_callback_keep_best_model()`, `luz_callback_lr_scheduler()`, `luz_callback_metrics()`, `luz_callback_mixup()`, `luz_callback_model_checkpoint()`, `luz_callback_profile()`, `luz_callback_progress()`, `luz_callback_resume_from_checkpoint()`, `luz_callback_train_valid()`, `luz_callback()`

Examples

```
if (torch::torch_is_installed()) {
  library(torch)
  library(luz)

  x <- torch_randn(1000, 10)
  y <- torch_randn(1000, 1)

  model <- nn_linear %>%
    setup(optimizer = optim_sgd, loss = nnf_mse_loss) %>%
    set_hparams(in_features = 10, out_features = 1) %>%
    set_opt_hparams(lr = 0.01)

  # simulate a failure in the middle of epoch 5 happening only once.
  callback_stop <- luz_callback(
    "interrupt",
    failed = FALSE,
    on_epoch_end = function() {
      if (ctx$epoch == 5 && !self$failed) {
        self$failed <- TRUE
        stop("Error on epoch 5")
      }
    }
  )

  path <- tempfile()
  autoresume <- luz_callback_auto_resume(path = path)
  interrupt <- callback_stop()
```

```

# try once and the model fails
try({
  results <- model %>% fit(
    list(x, y),
    callbacks = list(autoresume, interrupt),
    verbose = FALSE
  )
})

# model resumes and completes
results <- model %>% fit(
  list(x, y),
  callbacks = list(autoresume, interrupt),
  verbose = FALSE
)

get_metrics(results)

}

```

luz_callback_csv_logger

CSV logger callback

Description

Logs metrics obtained during training a fiel 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

Other luz_callbacks: [luz_callback_auto_resume\(\)](#), [luz_callback_early_stopping\(\)](#), [luz_callback_interrupt\(\)](#), [luz_callback_keep_best_model\(\)](#), [luz_callback_lr_scheduler\(\)](#), [luz_callback_metrics\(\)](#), [luz_callback_mixup\(\)](#), [luz_callback_model_checkpoint\(\)](#), [luz_callback_profile\(\)](#), [luz_callback_progress\(\)](#), [luz_callback_resume_from_checkpoint\(\)](#), [luz_callback_train_valid\(\)](#), [luz_callback\(\)](#)

```
luz_callback_early_stopping
```

Early stopping callback

Description

Stops training when a monitored metric stops improving

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 stoping 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 begining. Model will stopm 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_auto_resume\(\)](#), [luz_callback_csv_logger\(\)](#), [luz_callback_interrupt\(\)](#), [luz_callback_keep_best_model\(\)](#), [luz_callback_lr_scheduler\(\)](#), [luz_callback_metrics\(\)](#), [luz_callback_mixup\(\)](#), [luz_callback_model_checkpoint\(\)](#), [luz_callback_profile\(\)](#), [luz_callback_progress\(\)](#), [luz_callback_resume_from_checkpoint\(\)](#), [luz_callback_train_valid\(\)](#), [luz_callback\(\)](#)

Examples

```
cb <- luz_callback_early_stopping()
```

```
luz_callback_gradient_clip
```

Gradient clipping callback

Description

By adding the GradientClip callback, the gradient norm_type (default:2) norm is clipped to at most max_norm (default:1) using `torch::nn_utils_clip_grad_norm()`, which can avoid loss divergence.

Usage

```
luz_callback_gradient_clip(max_norm = 1, norm_type = 2)
```

Arguments

max_norm	(float or int): max norm of the gradients
norm_type	(float or int): type of the used p-norm. Can be Inf for infinity norm.

References

See FastAI [documentation](#) for the GradientClip callback.

```
luz_callback_interrupt
```

Interrupt callback

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

```
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

Other luz_callbacks: `luz_callback_auto_resume()`, `luz_callback_csv_logger()`, `luz_callback_early_stopping()`, `luz_callback_keep_best_model()`, `luz_callback_lr_scheduler()`, `luz_callback_metrics()`, `luz_callback_mixup()`, `luz_callback_model_checkpoint()`, `luz_callback_profile()`, `luz_callback_progress()`, `luz_callback_resume_from_checkpoint()`, `luz_callback_train_valid()`, `luz_callback()`

Examples

```
interrupt_callback <- luz_callback_interrupt()
```

```
luz_callback_keep_best_model
```

Keep the best model

Description

Each epoch, if there's improvement in the monitored metric we serialize the model weights to a temp file. When training is done, we reload weights from the best model.

Usage

```
luz_callback_keep_best_model(  
  monitor = "valid_loss",  
  mode = "min",  
  min_delta = 0  
)
```

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.
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 improvement to reset the patience counter.

See Also

Other luz_callbacks: `luz_callback_auto_resume()`, `luz_callback_csv_logger()`, `luz_callback_early_stopping()`, `luz_callback_interrupt()`, `luz_callback_lr_scheduler()`, `luz_callback_metrics()`, `luz_callback_mixup()`, `luz_callback_model_checkpoint()`, `luz_callback_profile()`, `luz_callback_progress()`, `luz_callback_resume_from_checkpoint()`, `luz_callback_train_valid()`, `luz_callback()`

Examples

```
cb <- luz_callback_keep_best_model()
```

```
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

<code>lr_scheduler</code>	A <code>torch::lr_scheduler()</code> that will be initialized with the optimizer and the ... parameters.
<code>...</code>	Additional arguments passed to <code>lr_scheduler</code> together with the optimizers.
<code>call_on</code>	The callback breakpoint that <code>scheduler\$step()</code> is called. Default is 'on_epoch_end'. See <code>luz_callback()</code> for more information.
<code>opt_name</code>	name of the optimizer that will be affected by this callback. Should match the name given in <code>set_optimizers</code> . If your module has a single optimizer, <code>opt_name</code> is not used.

Value

A `luz_callback()` generator.

See Also

Other `luz_callbacks`: `luz_callback_auto_resume()`, `luz_callback_csv_logger()`, `luz_callback_early_stopping()`, `luz_callback_interrupt()`, `luz_callback_keep_best_model()`, `luz_callback_metrics()`, `luz_callback_mixup()`, `luz_callback_model_checkpoint()`, `luz_callback_profile()`, `luz_callback_progress()`, `luz_callback_resume_from_checkpoint()`, `luz_callback_train_valid()`, `luz_callback()`

Examples

```
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

```
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

Other `luz_callbacks`: [luz_callback_auto_resume\(\)](#), [luz_callback_csv_logger\(\)](#), [luz_callback_early_stopping\(\)](#), [luz_callback_interrupt\(\)](#), [luz_callback_keep_best_model\(\)](#), [luz_callback_lr_scheduler\(\)](#), [luz_callback_mixup\(\)](#), [luz_callback_model_checkpoint\(\)](#), [luz_callback_profile\(\)](#), [luz_callback_progress\(\)](#), [luz_callback_resume_from_checkpoint\(\)](#), [luz_callback_train_valid\(\)](#), [luz_callback\(\)](#)

luz_callback_mixup *Mixup callback*

Description

Implementation of '[mixup: Beyond Empirical Risk Minimization](#)'. As of today, tested only for categorical data, where targets are expected to be integers, not one-hot encoded vectors. This callback is supposed to be used together with [nn_mixup_loss\(\)](#).

Usage

```
luz_callback_mixup(alpha = 0.4, ..., run_valid = FALSE, auto_loss = FALSE)
```

Arguments

alpha	parameter for the beta distribution used to sample mixing coefficients
...	currently unused. Just to force named arguments.
run_valid	Should it run during validation
auto_loss	Should it automatically modify the loss function? This will wrap the loss function to create the mixup loss. If TRUE make sure that your loss function does not apply reductions. If run_valid=FALSE, then loss will be mean reduced during validation.

Details

Overall, we follow the [fastai implementation](#) described [here](#). Namely,

- We work with a single dataloader only, randomly mixing two observations from the same batch.
- We linearly combine losses computed for both targets: $\text{loss}(\text{output}, \text{new_target}) = \text{weight} * \text{loss}(\text{output}, \text{target1}) + (1 - \text{weight}) * \text{loss}(\text{output}, \text{target2})$
- We draw different mixing coefficients for every pair.
- We replace weight with $\text{weight} = \max(\text{weight}, 1 - \text{weight})$ to avoid duplicates.

Value

A luz_callback

See Also

`nn_mixup_loss()`, `nnf_mixup()`

Other luz_callbacks: `luz_callback_auto_resume()`, `luz_callback_csv_logger()`, `luz_callback_early_stopping()`, `luz_callback_interrupt()`, `luz_callback_keep_best_model()`, `luz_callback_lr_scheduler()`, `luz_callback_metrics()`, `luz_callback_model_checkpoint()`, `luz_callback_profile()`, `luz_callback_progress()`, `luz_callback_resume_from_checkpoint()`, `luz_callback_train_valid()`, `luz_callback()`

Examples

```
if (torch::torch_is_installed()) {
  mixup_callback <- luz_callback_mixup()
}
```

```
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 <code>ctx</code> by using ' <code>{ctx\$epoch}</code> '. Specially the epoch and monitor quantities are already in the environment. If the specified path is a path to a directory (ends with <code>/</code> or <code>\</code>), then models are saved with the name given by <code>epoch-{epoch:02d}-{self\$monitor}-{monitor:.3f}.pt</code> . See more in the examples. You can use <code>sprintf()</code> to quickly format quantities, for example: ' <code>{epoch:02d}</code> '.
monitor	A string in the format <code><set>_<metric></code> where <code><set></code> can be 'train' or 'valid' and <code><metric></code> 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 <code>save_best_only=TRUE</code> .

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.

Read the checkpointing article in the pkgdown website for more information.

See Also

Other luz_callbacks: [luz_callback_auto_resume\(\)](#), [luz_callback_csv_logger\(\)](#), [luz_callback_early_stopping\(\)](#), [luz_callback_interrupt\(\)](#), [luz_callback_keep_best_model\(\)](#), [luz_callback_lr_scheduler\(\)](#), [luz_callback_metrics\(\)](#), [luz_callback_mixup\(\)](#), [luz_callback_profile\(\)](#), [luz_callback_progress\(\)](#), [luz_callback_resume_from_checkpoint\(\)](#), [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")
```

luz_callback_profile	<i>Profile callback</i>
----------------------	-------------------------

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

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

Other luz_callbacks: [luz_callback_auto_resume\(\)](#), [luz_callback_csv_logger\(\)](#), [luz_callback_early_stopping\(\)](#), [luz_callback_interrupt\(\)](#), [luz_callback_keep_best_model\(\)](#), [luz_callback_lr_scheduler\(\)](#), [luz_callback_metrics\(\)](#), [luz_callback_mixup\(\)](#), [luz_callback_model_checkpoint\(\)](#), [luz_callback_progress\(\)](#), [luz_callback_resume_from_checkpoint\(\)](#), [luz_callback_train_valid\(\)](#), [luz_callback\(\)](#)

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_auto_resume()`, `luz_callback_csv_logger()`, `luz_callback_early_stopping()`, `luz_callback_interrupt()`, `luz_callback_keep_best_model()`, `luz_callback_lr_scheduler()`, `luz_callback_metrics()`, `luz_callback_mixup()`, `luz_callback_model_checkpoint()`, `luz_callback_profile()`, `luz_callback_resume_from_checkpoint()`, `luz_callback_train_valid()`, `luz_callback()`

```
luz_callback_resume_from_checkpoint
```

Allow resume model training from a specific checkpoint

Description

Allow resume model training from a specific checkpoint

Usage

```
luz_callback_resume_from_checkpoint(
    path,
    ...,
    restore_model_state = TRUE,
    restore_records = FALSE,
    restore_optimizer_state = FALSE,
    restore_callbacks_state = FALSE
)
```

Arguments

<code>path</code>	Path to the checkpoint that you want to resume.
<code>...</code>	currently unused.
<code>restore_model_state</code>	Whether to restore the model state from the callback.
<code>restore_records</code>	Whether to restore records from the checkpoint.
<code>restore_optimizer_state</code>	Whether to restore the optimizer state from the checkpoint.
<code>restore_callbacks_state</code>	Whether to restore the callbacks state from the checkpoint.

Note

Read the checkpointing article in the pkgdown website for more information.

See Also

[luz_callback_model_checkpoint\(\)](#)

Other `luz_callbacks`: [luz_callback_auto_resume\(\)](#), [luz_callback_csv_logger\(\)](#), [luz_callback_early_stopping\(\)](#), [luz_callback_interrupt\(\)](#), [luz_callback_keep_best_model\(\)](#), [luz_callback_lr_scheduler\(\)](#), [luz_callback_metrics\(\)](#), [luz_callback_mixup\(\)](#), [luz_callback_model_checkpoint\(\)](#), [luz_callback_profile\(\)](#), [luz_callback_progress\(\)](#), [luz_callback_train_valid\(\)](#), [luz_callback\(\)](#)

`luz_callback_tfevents` *tfevents callback*

Description

Logs metrics and other model information in the tfevents file format. Assuming tensorboard is installed, result can be visualized with

Usage

```
luz_callback_tfevents(logdir = "logs", histograms = FALSE, ...)
```

Arguments

logdir	A directory to where log will be written to.
histograms	A boolean specifying if histograms of model weights should be logged. It can also be a character vector specifying the name of the parameters that should be logged (names are the same as names(model\$parameters)).
...	Currently not used. For future expansion.

Details

```
tensorboard --logdir=logs
```

Examples

```
if (torch::torch_is_installed()) {
  library(torch)
  x <- torch_randn(1000, 10)
  y <- torch_randn(1000, 1)

  model <- nn_linear %>%
    setup(loss = nnf_mse_loss, optimizer = optim_adam) %>%
    set_hparams(in_features = 10, out_features = 1) %>%
    set_opt_hparams(lr = 1e-4)

  tmp <- tempfile()

  model %>% fit(list(x, y), valid_data = 0.2, callbacks = list(
    luz_callback_tfevents(tmp, histograms = TRUE)
  ))
}
```

```
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_auto_resume()`, `luz_callback_csv_logger()`, `luz_callback_early_stopping()`, `luz_callback_interrupt()`, `luz_callback_keep_best_model()`, `luz_callback_lr_scheduler()`, `luz_callback_metrics()`, `luz_callback_mixup()`, `luz_callback_model_checkpoint()`, `luz_callback_profile()`, `luz_callback_progress()`, `luz_callback_resume_from_checkpoint()`, `luz_callback()`

luz_load	<i>Load trained model</i>
----------	---------------------------

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_checkpoint	<i>Loads a checkpoint</i>
---------------------	---------------------------

Description

Works with checkpoints created typically with `luz_callback_model_checkpoint()`.

Usage

`luz_load_checkpoint(obj, path, ...)`

Arguments

obj	Object to which we want to load the checkpoint.
path	Path of the checkpoint on disk.
...	unused. Is there to allow future extensions.

`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)
```

Arguments

obj	luz object to which you want to copy the new weights.
path	path to saved model in disk.
...	other arguments passed to <code>torch.load()</code> .

Value

Returns NULL invisibly.

Warning

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

luz_metric	<i>Creates a new luz metric</i>
------------	---------------------------------

Description

Creates a new luz metric

Usage

```
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 an `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:

```
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\(\)](#)

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.

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
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

```

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

```

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

```
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	<i>Mean absolute error</i>
----------------	----------------------------

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

```
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	<i>Mean squared error</i>
----------------	---------------------------

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
<i>Computes the multi-class AUROC</i>

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")  
)
```

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 <code>torch::nnf_softmax()</code> in the predictions before computing the metric. |
| average | The averaging method: <ul style="list-style-type: none"> • '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

```
if (torch::torch_is_installed()) {
  library(torch)
  actual <- c(1, 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	<i>Root mean squared error</i>
-----------------	--------------------------------

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_metric_set	<i>Creates a metric set</i>
----------------	-----------------------------

Description

A metric set can be used to specify metrics that are only evaluated during training, validation or both.

Usage

```
luz_metric_set(metrics = NULL, train_metrics = NULL, valid_metrics = NULL)
```

Arguments

metrics	A list of luz_metrics that are meant to be used in both training and validation.
train_metrics	A list of luz_metrics that are only used during training.
valid_metrics	A list of luz_metrics that are only used for validation.

luz_save	<i>Saves luz objects to disk</i>
----------	----------------------------------

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\(\)](#)

nnf_mixup	<i>Mixup logic</i>
-----------	--------------------

Description

Logic underlying [luz_callback_mixup\(\)](#).

Usage

```
nnf_mixup(x, y, weight)
```

Arguments

x	an input batch
y	a target batch
weight	weighting coefficient to be used by torch_lerp()

Details

Based on the passed-in input and target batches, as well as applicable mixing weights, we return new tensors intended to replace the current batch. The new input batch is a weighted linear combination of input batch items, while the new target batch bundles the original targets, as well as the mixing weights, in a nested list.

Value

A list of:

- x, the new, mixed-up input batch
- y, a list of:
 - ys, a list of:
 - * y1, the original target y1
 - * y2, the mixed-in target y2
 - weight, the mixing weights

See Also

[luz_callback_mixup\(\)](#)

Examples

```
if (torch::torch_is_installed()) {
  batch_x <- torch::torch_randn(c(10, 768))
  batch_y <- torch::torch_randn(10)
  weight <- torch::torch_tensor(rep(0.9, 10))$view(c(10, 1))
  nnf_mixup(batch_x, batch_y, weight)
}
```

nn_mixup_loss	<i>Loss to be used with callbacks_mixup().</i>
---------------	--

Description

In the training phase, computes individual losses with regard to two targets, weights them item-wise, and averages the linear combinations to yield the mean batch loss. For validation and testing, defers to the passed-in loss.

Usage

```
nn_mixup_loss(loss)
```

Arguments

`loss` the underlying loss nn_module to call. It must support the reduction field. During training the attribute will be changed to 'none' so we get the loss for individual observations. See for example documentation for the reduction argument in [torch::nn_cross_entropy_loss\(\)](#).

Details

It should be used together with [luz_callback_mixup\(\)](#).

See Also

[luz_callback_mixup\(\)](#)

```
predict.luz_module_fitted
```

Create predictions for a fitted model

Description

Create predictions for a fitted model

Usage

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

Arguments

<code>object</code>	(fitted model) the fitted model object returned from fit.luz_module_generator()
<code>newdata</code>	(dataloader, dataset, list or array) returning a list with at least 1 element. The other elements aren't used.
<code>...</code>	Currently unused.

callbacks	(list, optional) A list of callbacks defined with <code>luz_callback()</code> that will be called during the training procedure. The callbacks <code>luz_callback_metrics()</code> , <code>luz_callback_progress()</code> and <code>luz_callback_train_valid()</code> are always added by default.
accelerator	(accelerator, optional) An optional <code>accelerator()</code> object used to configure device placement of the components like <code>nn_modules</code> , 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 <code>interactive()</code> is TRUE, otherwise it won't print to the console.
dataloader_options	Options used when creating a dataloader. See <code>torch::dataloader()</code> . <code>shuffle=TRUE</code> by default for the training data and <code>batch_size=32</code> 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	<i>Set's up a nn_module to use with luz</i>
-------	---

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, backward = NULL)
```

Arguments

module	(nn_module) The nn_module that you want set up.
loss	(function, optional) An optional function with the signature <code>function(input, target)</code> . It's only requires if your nn_module doesn't implement a method called <code>loss</code> .
optimizer	(torch_optimizer, optional) A function with the signature <code>function(parameters, ...)</code> 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. Sometimes, you want some metrics to be evaluated only during training or validation, in this case you can pass a <code>luz_metric_set()</code> object to specify metrics used in each stage.
backward	(function) A functions that takes the loss scalar values as it's parameter. It must call <code>\$backward()</code> or <code>torch::autograd_backward()</code> . In general you don't need to set this parameter unless you need to customize how <code>luz</code> calls the <code>backward()</code> , for example, if you need to add additional arguments to the backward call. Note that this becomes a method of the <code>nn_module</code> thus can be used by your custom <code>step()</code> if you override it.

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()`.

Note

It also adds a device active field that can be used to query the current module device within methods, with eg `self$device`. This is useful when `ctx()` is not available, eg, when calling methods from outside the luz wrappers. Users can override the default by implementing a device active method in the input module.

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

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

Arguments

<code>module</code>	An nn_module that has been <code>setup()</code> .
<code>...</code>	The parameters set here will be used to initialize the nn_module, ie they are passed unchanged to the <code>initialize</code> method of the base nn_module.

Value

The same luz module

See Also

Other set_hparam: `set_opt_hparams()`

set_opt_hparams	<i>Set optimizer hyper-parameters</i>
-----------------	---------------------------------------

Description

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

Usage

```
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 <code>optim_adam</code> and you pass <code>lr=0.1</code> , then the <code>optim_adam</code> function is called with <code>optim_adam(parameters, lr=0.1)</code> when fitting the model.

Value

The same luz module

See Also

Other set_hparam: [set_hparams\(\)](#)

Index

- * **luz_callbacks**
 - luz_callback, 15
 - luz_callback_auto_resume, 18
 - luz_callback_csv_logger, 20
 - luz_callback_early_stopping, 21
 - luz_callback_interrupt, 22
 - luz_callback_keep_best_model, 23
 - luz_callback_lr_scheduler, 24
 - luz_callback_metrics, 25
 - luz_callback_mixup, 25
 - luz_callback_model_checkpoint, 27
 - luz_callback_profile, 28
 - luz_callback_progress, 29
 - luz_callback_resume_from_checkpoint, 29
 - luz_callback_train_valid, 31
- * **luz_metrics**
 - luz_metric, 34
 - luz_metric_accuracy, 36
 - luz_metric_binary_accuracy, 37
 - luz_metric_binary_accuracy_with_logits, 38
 - luz_metric_binary_auroc, 39
 - luz_metric_mae, 40
 - luz_metric_mse, 41
 - luz_metric_multiclass_auroc, 41
 - luz_metric_rmse, 43
- * **luz_save**
 - luz_load, 32
 - luz_save, 44
- * **set_hparam**
 - set_hparams, 48
 - set_opt_hparams, 49
- * **training**
 - evaluate, 10
 - fit.luz_module_generator, 11
 - predict.luz_module_fitted, 46
 - setup, 47
- accelerator(), 3
- accelerator(), 5, 6, 10, 12, 47
- as_dataloader, 3
- context, 5, 10
- ctx, 5, 9, 25, 27, 44
- ctx(), 48
- evaluate, 10, 13, 47, 48
- evaluate(), 17
- fit(), 17, 48
- fit.luz_module_generator, 11, 11, 47, 48
- fit.luz_module_generator(), 3, 4, 17, 21, 23, 25, 28, 29, 32, 44, 46
- get_metrics, 13
- interactive(), 8, 10, 12, 47
- lr_finder, 13
- luz_callback, 15, 19–21, 23–26, 28–30, 32
- luz_callback(), 6, 10, 12, 24, 47
- luz_callback_auto_resume, 18, 18, 20, 21, 23–26, 28–30, 32
- luz_callback_csv_logger, 18, 19, 20, 21, 23–26, 28–30, 32
- luz_callback_early_stopping, 18–20, 21, 23–26, 28–30, 32
- luz_callback_gradient_clip, 22
- luz_callback_interrupt, 18–21, 22, 23–26, 28–30, 32
- luz_callback_keep_best_model, 18–21, 23, 23, 24–26, 28–30, 32
- luz_callback_lr_scheduler, 18–21, 23, 24, 25, 26, 28–30, 32
- luz_callback_metrics, 18–21, 23, 24, 25, 26, 28–30, 32
- luz_callback_metrics(), 10, 12, 47
- luz_callback_mixup, 18–21, 23–25, 25, 28–30, 32
- luz_callback_mixup(), 44–46

luz_callback_model_checkpoint, 18–21, 23–26, 27, 28–30, 32
 luz_callback_model_checkpoint(), 30, 32
 luz_callback_profile, 18–21, 23–26, 28, 28, 29, 30, 32
 luz_callback_progress, 18–21, 23–26, 28, 29, 30, 32
 luz_callback_progress(), 10, 12, 47
 luz_callback_resume_from_checkpoint, 18–21, 23–26, 28, 29, 29, 32
 luz_callback_tfevents, 30
 luz_callback_train_valid, 18–21, 23–26, 28–30, 31
 luz_callback_train_valid(), 10, 12, 47
 luz_load, 32, 44
 luz_load(), 44
 luz_load_checkpoint, 32
 luz_load_model_weights, 33
 luz_metric, 34, 37–43
 luz_metric_accuracy, 35, 36, 37–43
 luz_metric_binary_accuracy, 35, 37, 37, 38–43
 luz_metric_binary_accuracy_with_logits, 35, 37, 38, 39–43
 luz_metric_binary_auroc, 35, 37, 38, 39, 40–43
 luz_metric_mae, 35, 37–39, 40, 41–43
 luz_metric_mse, 35, 37–40, 41, 42, 43
 luz_metric_multiclass_auroc, 35, 37–41, 41, 43
 luz_metric_rmse, 35, 37–42, 43
 luz_metric_set, 43
 luz_metric_set(), 47
 luz_save, 32, 44
 luz_save(), 12, 32
 luz_save_model_weights
 (luz_load_model_weights), 33

 nn_mixup_loss, 45
 nn_mixup_loss(), 25, 26
 nn_module, 10, 12, 47
 nnf_mixup, 44
 nnf_mixup(), 26

 plot(), 12
 predict(), 17
 predict.luz_module_fitted, 11, 13, 46, 48
 predict.luz_module_fitted(), 13
 print(), 12

 rlang::try_fetch(), 5

 saveRDS(), 44
 set_hparams, 48, 49
 set_opt_hparams, 48, 49
 setup, 11, 13, 47, 47
 setup(), 12, 13, 25, 48, 49
 sprintf(), 27

 torch::autograd_backward(), 47
 torch::dataloader, 3, 4
 torch::dataloader(), 4, 10–12, 47
 torch::dataset, 4
 torch::dataset(), 4, 10, 12
 torch::lr_scheduler(), 24
 torch::nn_bce_loss(), 37
 torch::nn_bce_with_logits_loss(), 38
 torch::nn_cross_entropy_loss(), 46
 torch::nn_utils_clip_grad_norm_(), 22
 torch::nnf_sigmoid(), 38
 torch::nnf_softmax(), 42
 torch_load(), 33