Package: brulee (via r-universe)

July 2, 2026

Title High-Level Modeling Functions with 'torch'

Version 1.1.0

Description Provides high-level modeling functions to define and train models using the 'torch' R package. Models include linear, logistic, and multinomial regression as well as multilayer perceptrons.

License MIT + file LICENSE

URL https://github.com/tidymodels/brulee, https://brulee.tidymodels.org/

BugReports https://github.com/tidymodels/brulee/issues

Depends R (>= 4.1)

Imports cli, coro (>= 1.0.1), curl, dplyr, generics, ggplot2, hardhat, jsonlite, purrr, rlang (>= 1.1.1), safetensors, stats, tibble, tidyselect, torch (>= 0.13.0), utils, withr

Suggests covr, lubridate, modeldata, recipes, spelling, testthat, yardstick

Config/Needs/website tidyverse/tidytemplate

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Encoding UTF-8

Language en-US

RoxygenNote 8.0.0

NeedsCompilation no

Author Max Kuhn [aut, cre] (ORCID: <https://orcid.org/0000-0003-2402-136X>), Daniel Falbel [aut], Posit Software, PBC [cph, fnd]

Maintainer Max Kuhn <max@posit.co>

Repository https://cran.r-universe.dev

Date/Publication 2026-07-02 12:40:02 UTC

RemoteUrl https://github.com/cran/brulee

RemoteRef HEAD

RemoteSha 36e7c386740aadd1c0c40f110913d20d4a87aa22

Contents

`brulee-autoplot` *Plot model loss over epochs*

Description

Plot model loss over epochs

Usage

```
## S3 method for class 'brulee_mlp'
autoplot(object, ...)

## S3 method for class 'brulee_logistic_reg'
autoplot(object, ...)

## S3 method for class 'brulee_multinomial_reg'
autoplot(object, ...)

## S3 method for class 'brulee_linear_reg'
autoplot(object, ...)

## S3 method for class 'brulee_resnet'
autoplot(object, ...)

## S3 method for class 'brulee_auto_int'
autoplot(object, ...)

## S3 method for class 'brulee_saint'
autoplot(object, ...)

## S3 method for class 'brulee_rln'
autoplot(object, ...)
```

Arguments

object	A `brulee_mlp`, `brulee_logistic_reg`, `brulee_multinomial_reg`, or `brulee_linear_reg` object.
...	Not currently used

Details

This function plots the loss function across the available epochs. A vertical line shows the epoch with the best loss value.

Value

A `ggplot` object.

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "yardstick", "modeldata"))) {
 library(ggplot2)
 library(recipes)
 theme_set(theme_bw())

 data(ames, package = "modeldata")
```

```
 ames$Sale_Price <- log10(ames$Sale_Price)

 set.seed(1)
 in_train <- sample(seq_len(nrow(ames)), 2000)
 ames_train <- ames[ in_train,]
 ames_test  <- ames[-in_train,]

 ames_rec <-
  recipe(Sale_Price ~ Longitude + Latitude, data = ames_train) |>
    step_normalize(all_numeric_predictors())

 set.seed(2)
 fit <- brulee_mlp(ames_rec, data = ames_train, epochs = 50, batch_size = 32)

 autoplot(fit)
}
```

---

brulee-coefs *Extract Model Coefficients*

Description

Extract Model Coefficients

Usage

```
## S3 method for class 'brulee_logistic_reg'
coef(object, epoch = NULL, ...)

## S3 method for class 'brulee_linear_reg'
coef(object, epoch = NULL, ...)

## S3 method for class 'brulee_mlp'
coef(object, epoch = NULL, ...)

## S3 method for class 'brulee_multinomial_reg'
coef(object, epoch = NULL, ...)

## S3 method for class 'brulee_resnet'
coef(object, epoch = NULL, ...)

## S3 method for class 'brulee_rln'
coef(object, epoch = NULL, ...)
```

Arguments

object	A model fit from **brulee**.
epoch	A single integer for the training iteration. If left NULL, the estimates from the best model fit (via internal performance metrics).
...	Not currently used.

Value

For logistic/linear regression, a named vector. For neural networks, a list of arrays.

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "modeldata"))) {

 data(ames, package = "modeldata")

 ames$Sale_Price <- log10(ames$Sale_Price)

 set.seed(1)
 in_train <- sample(seq_len(nrow(ames)), 2000)
 ames_train <- ames[ in_train,]
 ames_test  <- ames[-in_train,]

 # Using recipe
 library(recipes)

 ames_rec <-
  recipe(Sale_Price ~ Longitude + Latitude, data = ames_train) |>
    step_normalize(all_numeric_predictors())

 set.seed(2)
 fit <- brulee_linear_reg(ames_rec, data = ames_train, epochs = 50)

 coef(fit)
 coef(fit, epoch = 1)
}
```

---

brulee_activations *Activation functions for neural networks in brulee*

Description

Activation functions for neural networks in brulee

Usage

```
brulee_activations()
```

Value

A character vector of values.

`brulee_auto_int` *Fit AutoInt models for tabular data*

Description

`brulee_auto_int()` fits AutoInt from Song *at al* (2019) that use multi-head columnar self-attention to help exploit how combinations of embeddings can be used to improve specific predictions.

Usage

```
brulee_auto_int(x, ...)

## Default S3 method:
brulee_auto_int(x, ...)

## S3 method for class 'data.frame'
brulee_auto_int(
  x,
  y,
  epochs = 100L,
  num_embedding = 16L,
  num_attn_feat = 16L,
  num_attn_heads = 2L,
  num_attn_blocks = 3L,
  activation = "relu",
  hidden_units = NULL,
  hidden_activations = NULL,
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  dropout_attn = 0,
  dropout_embedding = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
```

```
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'matrix'
brulee_auto_int(
  x,
  y,
  epochs = 100L,
  num_embedding = 16L,
  num_attn_feat = 16L,
  num_attn_heads = 2L,
  num_attn_blocks = 3L,
  activation = "relu",
  hidden_units = NULL,
  hidden_activations = NULL,
  dropout = 0,
  penalty = 0.001,
  mixture = 0,
  dropout_attn = 0,
  dropout_embedding = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'formula'
brulee_auto_int(
  formula,
  data,
  epochs = 100L,
  num_embedding = 16L,
  num_attn_feat = 16L,
  num_attn_heads = 2L,
  num_attn_blocks = 3L,
```

```
  activation = "relu",
  hidden_units = NULL,
  hidden_activations = NULL,
  dropout = 0,
  penalty = 0.001,
  mixture = 0,
  dropout_attn = 0,
  dropout_embedding = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'recipe'
brulee_auto_int(
  x,
  data,
  epochs = 100L,
  num_embedding = 16L,
  num_attn_feat = 16L,
  num_attn_heads = 2L,
  num_attn_blocks = 3L,
  activation = "relu",
  hidden_units = NULL,
  hidden_activations = NULL,
  dropout = 0,
  penalty = 0.001,
  mixture = 0,
  dropout_attn = 0,
  dropout_embedding = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
```

```
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)
```

Arguments

x	Depending on the context: • A **data frame** of predictors. • A **matrix** of predictors. • A **recipe** specifying a set of preprocessing steps created from `recipes::recipe()`. The predictor data should be standardized (e.g. centered or scaled).
...	Options to pass to the learning rate schedulers via `set_learn_rate()`. For example, the `reduction` or `steps` arguments to `schedule_step()` could be passed here.
y	When `x` is a **data frame** or **matrix**, `y` is the outcome specified as: • A **data frame** with 1 column (numeric or factor). • A **matrix** with numeric column (numeric or factor). • A **vector** (numeric or factor).
epochs	An integer for the number of epochs of training.
num_embedding	An integer for the embedding dimension. Each feature (categorical or continuous) is mapped to a vector of this dimension. Must be >= 1.
num_attn_feat	An integer for the per-head attention dimension. The total attention dimension is `num_attn_feat * num_attn_heads`. Must be >= 1.
num_attn_heads	An integer for the number of attention heads. Each head learns different interaction patterns in parallel. Must be >= 1.
num_attn_blocks	An integer for the number of stacked self-attention layers. More layers capture higher-order interactions. Must be >= 1.
activation	A single character string for the activation function used in the self-attention backbone (applied after each residual connection in each attention block). This does not affect the optional hidden layers; use `hidden_activations` for those. See `brulee_activations()` for options.
hidden_units	An integer vector for the number of units in optional hidden layers between the attention backbone and the output head. For example, `c(64L, 32L)` adds two hidden layers with 64 and 32 units. When `NULL` (the default), no hidden layers are added.
hidden_activations	A character vector of activation functions for the hidden layers. Must be the same length as `hidden_units` or a single value that will be recycled. When `NULL` (the default), no hidden layers are added. See `brulee_activations()` for options.

penalty — The amount of weight decay (i.e., L2 regularization).

mixture — Proportion of Lasso Penalty (type: double, default: 0.0). A value of mixture = 1 corresponds to a pure lasso model, while mixture = 0 indicates ridge regression (a.k.a weight decay). Must be zero for optimizers "ADAMw", "RMSprop", "Adadelta".

dropout — A number in [0, 1) for the dropout rate applied between the last hidden layer and the output head. Only has effect when hidden_units is not NULL. Default is 0 (no dropout).

dropout_attn — A number in [0, 1) for the dropout rate applied to attention weights during training.

dropout_embedding — A number in [0, 1) for the dropout rate applied to the embedding layer during training.

validation — The proportion of the data randomly assigned to a validation set.

optimizer — The method used in the optimization procedure. Possible choices are "SGD", "ADAMw", "Adadelta", "Adagrad", "RMSprop", and "LBFGS". "LBFGS" is the only second-order method and does not use batches.

learn_rate — A positive number that controls the initial rapidity that the model moves along the descent path. Values around 0.1 or less are typical.

rate_schedule — A single character value for how the learning rate should change as the optimization proceeds. Possible values are "none" (the default), "decay_time", "decay_expo", "cyclic" and "step". See schedule_decay_time() for more details.

momentum — A positive number usually on [0.50, 0.99] for the momentum parameter in gradient descent. (optimizers "SGD", and "RMSprop" only, ignored otherwise).

batch_size — An integer for the number of training set points in each batch. (optimizer != "LBFGS" only, ignored otherwise)

class_weights — Numeric class weights (classification only). The value can be:

- A named numeric vector (in any order) where the names are the outcome factor levels.
- An unnamed numeric vector assumed to be in the same order as the outcome factor levels.
- A single numeric value for the least frequent class in the training data and all other classes receive a weight of one.

stop_iter — A non-negative integer for how many iterations with no improvement before stopping.

grad_norm_clip, grad_value_clip — Two numeric values, possibly Inf, that prevents the gradient's values or norm(s) from exceeding the specified value. This can be helpful if training stops early with the message that "Loss is NaN at epoch x Training is stopped."

verbose — A logical that prints out the iteration history.

device — A single character string for the device to train on (e.g., "cpu" or "cuda" for GPU). If NULL, the function will use the GPU if available, otherwise CPU. See training_efficiency.

`formula` A formula specifying the outcome term(s) on the left-hand side, and the predictor term(s) on the right-hand side.

`data` When a **recipe** or **formula** is used, `data` is specified as:

- A **data frame** containing both the predictors and the outcome.

Details

What is Being Estimated:

In statistics, an interaction occurs when two or more predictors jointly predict the outcome. You need to know the values of all predictors within the interaction effect to appropriately model the data. AutoInt is often described as "automatically learning feature interactions," but that is not an accurate description.

In neural networks, the original predictors are converted to *embeddings*, which are often the hidden units of the network.

AutoInt uses *column attention* to change how embeddings are represented. It learns how to make the embeddings more relevant to the outcome by creating mixtures of them. For example, if we predict a data point in one part of the predictor space, attention will refocus (i.e., transform) the embedding to be more relevant to that part of the space.

Architecture:

The AutoInt architecture has three stages:

1. **Embedding layer**: Maps every feature (categorical or continuous) into a shared vector space of dimension `num_embedding`.
2. **Self-attention backbone**: A stack of `num_attn_blocks` multi-head self-attention layers. After all blocks, a residual connection from the original embeddings is added and an activation is applied.
3. **Hidden layers** (optional): If `hidden_units` is specified, one or more fully-connected layers with activations process the flattened attention output before the output head.
4. **Output head**: Projects to the output dimension via a linear layer.

Unlike other **brulee** models, `brulee_auto_int()` natively handles factor predictors via learned embeddings. Factor columns are automatically detected and embedded, while numeric columns use a scaled embedding. There is *no need to pre-encode factors as indicators*.

Attention Parameters:

The self-attention backbone has several tuning parameters that control its capacity and regularization:

- `num_attn_heads`: The number of attention heads that operate **in parallel** within each attention block. Each head independently learns which features interact, giving the model multiple "views" of the feature relationships. The total attention dimension per block is `num_attn_feat * num_attn_heads`.
- `num_attn_feat`: The per-head attention dimension. Each head projects features into a space of this size to compute attention scores. Larger values give each head more capacity to represent complex interactions.
- `num_attn_blocks`: The number of attention layers stacked **sequentially**. Each block's output feeds into the next, allowing the model to build higher-order interactions (e.g., block 1 captures pairwise interactions, block 2 can combine those into three-way interactions, etc.).

- `activation`: The activation function applied after the residual connection at the end of the attention backbone.
- `dropout_attn`: Dropout applied to the attention weight matrix within each block, which randomly zeroes out attention connections during training.

Learning Rates:

The learning rate can be set to constant (the default) or dynamically set via a learning rate scheduler (via the `rate_schedule`). Using `rate_schedule = 'none'` uses the `learn_rate` argument. Otherwise, any arguments to the schedulers can be passed via ....

Other Notes:

When the outcome is a number, the function internally standardizes the outcome data to have mean zero and a standard deviation of one. The prediction function creates predictions on the original scale.

By default, training halts when the validation loss increases for at least `stop_iter` iterations. If `validation = 0` the training set loss is used.

The model objects are saved for each epoch so that the number of epochs can be efficiently tuned. Both the `predict()` method for this model has an epoch argument (which defaults to the epoch with the best loss value).

The use of the L1 penalty (a.k.a. the lasso penalty) does *not* force parameters to be strictly zero (as it does in packages such as **glmnet**). The zeroing out of parameters is a specific feature the optimization method used in those packages.

Value

A `brulee_auto_int` object with elements:

- `models_obj`: a serialized raw vector for the torch module.
- `estimates`: a list of matrices with the model parameter estimates per epoch. The first element is epoch zero (the randomly initialized parameters before training), so the list has `epochs + 1` elements.
- `best_epoch`: an integer for the epoch with the smallest loss. Since `estimates` and `loss` include epoch zero, this epoch's values are at position `best_epoch + 1` in those objects.
- `loss`: A vector of loss values (MSE for regression, negative log- likelihood for classification) at each epoch, starting with epoch zero.
- `dim`: A list of data dimensions and feature metadata.
- `top_interactions`: A tibble containing the top 10 two-way feature interactions.
- `y_stats`: A list of summary statistics for numeric outcomes.
- `parameters`: A list of some tuning parameter values.
- `device`: A character string for the device used during training.
- `blueprint`: The `hardhat` blueprint data.

References

Song, W., Shi, C., Xiao, Z., Duan, Z., Xu, Y., Zhang, M., & Tang, J. (2019). AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM)*.

See Also

predict.brulee_auto_int(), autoplot.brulee_auto_int()

Examples

```
pkgs <- c("recipes", "yardstick", "modeldata")
if (torch::torch_is_installed() && rlang::is_installed(pkgs)) {

  set.seed(87261)
  tr_data <- modeldata::sim_regression(500)
  te_data <- modeldata::sim_regression(50)

  set.seed(2)
  fit <- brulee_auto_int(outcome ~ ., data = tr_data,
                         epochs = 50L, batch_size = 64L, stop_iter = 10L,
                         learn_rate = 0.01, penalty = 0.01)
  fit

  autoplot(fit)

  library(yardstick)
  predict(fit, te_data) |>
   dplyr::bind_cols(te_data) |>
   rmse(outcome, .pred)

}
```

---

brulee_chronos *Chronos-2 pretrained forecasting model*

Description

brulee_chronos() loads a pretrained Chronos-2 time series forecasting quantile regression model from HuggingFace and ingests historical ("context") data so that the returned object is ready to forecast. Unlike other brulee models, no training is performed; the network has fixed pretrained weights.

Usage

```
brulee_chronos(x, ...)

## Default S3 method:
brulee_chronos(x, ...)

## S3 method for class 'data.frame'
brulee_chronos(
```

```
  x,
  y,
  item_id = NULL,
  timestamp = NULL,
  id_column = ".id_column",
  timestamp_column = ".timestamp_column",
  model_id = "amazon/chronos-2",
  revision = chronos2_default_revision(),
  prediction_length = NULL,
  quantile_levels = (1:9)/10,
  device = NULL,
  cache_dir = file.path(Sys.getenv("HOME"), ".cache", "chronos-r"),
  ...
)

## S3 method for class 'formula'
brulee_chronos(
  formula,
  data,
  id_column = NULL,
  timestamp_column = NULL,
  model_id = "amazon/chronos-2",
  revision = chronos2_default_revision(),
  prediction_length = NULL,
  quantile_levels = (1:9)/10,
  device = NULL,
  cache_dir = file.path(Sys.getenv("HOME"), ".cache", "chronos-r"),
  ...
)

## S3 method for class 'recipe'
brulee_chronos(
  x,
  data,
  model_id = "amazon/chronos-2",
  revision = chronos2_default_revision(),
  prediction_length = NULL,
  quantile_levels = (1:9)/10,
  device = NULL,
  cache_dir = file.path(Sys.getenv("HOME"), ".cache", "chronos-r"),
  ...
)
```

Arguments

x	Depending on the context: • A **data frame** of past covariates. • A **recipe** specifying preprocessing and roles for `target`, `id`, and `time`

columns.

Pass an empty data frame when there are no covariates.

`...` Currently unused.

`y` A numeric vector of target values, of length `nrow(x)`.

`item_id` Optional vector of time series identifiers, of length `nrow(x)`. Default: `NULL`, which treats all rows as a single series.

`timestamp` Optional vector of timestamps (Date, POSIXct, or numeric), of length `nrow(x)`. Default: `NULL`, which uses row order within each series.

`id_column` For the formula method, a tidyselect expression selecting the id column in `data` (e.g. `c(series_id)`, `series_id`, or `"series_id"`). For the data frame `x_y` method, a character string is used as the output label only (the actual id values come from `item_id`). Default: `NULL` for the formula method and `".id_column"` for the `x_y` method. When omitted, all rows are treated as one series. For the recipe method, identify the id column with `recipes::update_role(..., new_role = "id")`.

`timestamp_column`
For the formula method, a tidyselect expression selecting the timestamp column in `data`. For the data frame `x_y` method, a character string is used as the output label only. Default: `NULL` for the formula method and `".timestamp_column"` for the `x_y` method. When omitted, row order is used as the time order. For the recipe method, identify the timestamp column with `recipes::update_role(..., new_role = "time")`.

`model_id` A character string identifying the HuggingFace model repository to download. Default: `"amazon/chronos-2"` (120M parameters).

`revision` A character string identifying which version of the weights to load. May be a 40-character commit SHA, a tag, or a branch name on the HuggingFace repo (e.g. `"main"`). Default: a commit SHA pinned by brulee so the weights cannot change without you opting in. The resolved SHA is recorded on the returned object as `object$revision` and printed by `print()`.

`prediction_length`
An integer for the number of future time steps to forecast. Default: `NULL` (uses the model maximum). Must not exceed the model maximum. Can be overridden at `predict()` time.

`quantile_levels`
A numeric vector of quantile levels to produce in predictions. Must be a subset of the model's trained quantiles. Default: `(1:9) / 10`. Can be overridden at `predict()` time.

`device` A character string for the computation device: `"cpu"`, `"cuda"`, or `"mps"`. Default: `NULL` (auto-detects best available).

`cache_dir` Path to a directory for caching downloaded model files. Default: `"~/.cache/chronos-r"`.

`formula` A formula of the form `target ~ cov1 + cov2`. Use `target ~ .` when there are no covariates. The id and timestamp columns (if named) are dropped before the formula is evaluated.

`data` When a **recipe** or **formula** is used, `data` is the training set with columns for the id, timestamp, target, and any covariates.

Details

Computing Requirements:

This model can be used with or without a graphics processing unit (GPU). However, it may be computationally slow when used with a CPU (and no GPU).

Model Weight File Download:

Keep in mind that, on the first usage of the fitting function, the package will attempt to download the model weights file. This file can require about 500MB and is locally cached.

Interface Overview:

Every Chronos-2 forecast needs at most four pieces of information about the historical (context) data:

- A **target** column with the values to forecast (always required),
- An optional **id** column that distinguishes one time series from another (e.g. a city, store, or sensor); when omitted, all rows are treated as a single series,
- An optional **timestamp** column with the time index of each observation; when omitted, rows are read in their existing order,
- Any number of **past covariates**, additional numeric columns measured alongside the target.

`brulee_chronos()` is a generic with three interfaces for supplying that information; this intended to add flexibility in how you declare the model as well as what data are given as inputs. All three produce an object that behaves the same way at predict time.

To contrast these approaches, consider the `Chicago` data contained in the **modeldata** package. The goal is to predict daily train `ridership`. There is a `date` column, as well as a set of 14-day lagged ridership data from our station of interest and from others in the Chicago system.

You could use Chronos in the simplest way by just passing in the column containing past ridership values. It assumes that there are no gaps in the data and that the data are arranged/sorted in the proper order (past to present). The simplest interfaces to use in this case are the formula and matrix ones.

We could add the `date` column, but this is primarily used to label the data. Here, we would want the formula or recipe interface.

In these data, only one station's ridership is modeled. Suppose we did this for all stations. In that case, we would *stack* the ridership data and use the `id` argument to specify which station corresponds to each row. In this implementation, that is equivalent to running the function separately for each station; it is just a simpler interface with some small computational gains.

If we wanted to use covariates in our model, such as lagged ridership data, we can do so with the formula or recipe interfaces (see below).

Formula interface:

Use a formula when your data is a single tidy data frame and you want to name the covariates inline. The `id_column` and `timestamp_column` arguments use tidyselect, so bare column names, `c()` selections, and character strings all work:

```
brulee_chronos(target ~ cov1 + cov2, data = df,
               id_column = c(series_id), timestamp_column = c(date))

# bare names also work
```

```
brulee_chronos(target ~ cov1 + cov2, data = df,
               id_column = series_id, timestamp_column = date)

# character strings still work for back compatibility
brulee_chronos(target ~ cov1 + cov2, data = df,
               id_column = "series_id", timestamp_column = "date")
```

If you have no covariates, use `target ~ .`. The id and timestamp columns are excluded automatically. Categorical covariates on the right hand side are converted to numeric dummy variables (just like `lm()`).

If you have a single series and no useful timestamp, you can omit both columns entirely:

```
brulee_chronos(target ~ ., data = df_single_series)
```

Recipe interface:

Use a `recipes::recipe()` when you want to apply preprocessing steps (e.g. normalizing or encoding columns) before the data reaches the model. With the recipe interface, the id and timestamp columns are identified by their **role**, not by name:

```
rec <- recipe(target ~ ., data = df) |>
  update_role(item_id,   new_role = "id") |>
  update_role(timestamp, new_role = "time") |>
  step_normalize(all_numeric_predictors())

brulee_chronos(rec, data = df)
```

Both the `id` and `time` roles are optional. If neither role is set, `brulee_chronos()` treats the recipe data as a single series in row order. All non numeric covariates must be encoded numerically by the recipe (e.g. with `recipes::step_dummy()`).

Data-frame (`x` and `y`) interface:

Use the `x_y` interface when you already have your covariates and target separated. `x` is a data frame of past covariates (zero columns is allowed when there are no covariates), `y` is the numeric target vector, and `item_id` / `timestamp` are optional vectors of length `nrow(x)`:

```
brulee_chronos(x = df[, c("cov1", "cov2")], y = df$target,
               item_id = df$item_id, timestamp = df$timestamp)

# single series, no timestamp:
brulee_chronos(x = df[, c("cov1", "cov2")], y = df$target)
```

Multiple time series:

All three interfaces support multiple series in one call. Stack the series end to end in a single long format data frame and let the id column distinguish them. `brulee_chronos()` sorts each series by timestamp before forecasting. When you omit the id column, every row is treated as part of one series called `"default"`.

Pre-sorted input:

When you omit the timestamp, `brulee_chronos()` uses each series' row order as its time order. Pre-sort each series before calling `brulee_chronos()` if you take this shortcut.

What happens at `predict()` time:

The model is pretrained and performs no training, so the historical context is always the data supplied at construction; `predict.brulee_chronos()` forecasts forward from that context. By default it returns the full `prediction_length` horizon. To forecast a specific future window—and to supply known future values of any covariate (e.g., holiday flags, planned promotions)—pass that window as `new_data`. Its per-series row count sets how many future steps are returned (at most `prediction_length`). When the model has no covariates, `new_data` only needs the id and timestamp columns that describe the future steps.

Value

A `brulee_chronos` object with elements:

- `model`: The torch `nn_module` (in eval mode, on the specified device).
- `config`: Parsed model configuration list.
- `device`: The torch device in use.
- `prediction_length`: Validated prediction length.
- `quantile_levels`: Validated quantile levels.
- `model_id`: The HuggingFace repository the weights came from.
- `revision`: The 40-character commit SHA of the weights actually loaded.
- `blueprint`: The hardhat blueprint for processing new data.
- `context`: A list with the per-series target, covariates, timestamps, and column-name metadata that `predict()` uses by default.

References

Ansari, A. F., Shchur, O., Küken, J., Auer, A., Han, B., Mercado, P., ... & Bohlke-Schneider, M. (2025). "Chronos-2: From univariate to universal forecasting." *arXiv preprint arXiv:2510.15821*.

Ansari, A. F., Shchur, O., Küken, J., Auer, A., Han, B., Mercado, P., ... & Bohlke-Schneider, M. (2026). "A foundation model for multivariate time series forecasting.", https://doi.org/10.21203/rs.3.rs-9096522/v1

Examples

```
pkgs <- c("recipes", "lubridate", "modeldata", "ggplot2")

## Not run:
if (torch::torch_is_installed() && rlang::is_installed(pkgs)) {
 library(dplyr)
 library(ggplot2)

 n <- nrow(modeldata::Chicago)

 prior_data <- modeldata::Chicago[-((n-13):n),]
 test_data <-
  modeldata::Chicago[(n-13):n,] |>
  mutate(day = lubridate::wday(date, label = TRUE))
```

```
 # ------------------------------------------------------------------------------
 # Simple, no covariate model

 mod_1 <-
  brulee_chronos(
   ridership ~ 1,
   data = prior_data,
   # Removing `timestamp_column` does not affect the fit
   timestamp_column = c(date),
   prediction_length = 14)

 # ------------------------------------------------------------------------------
 # Some covariates via the formula method

mod_2 <-
  brulee_chronos(
   ridership ~ Clark_Lake + Belmont + Harlem + Monroe,
   data = prior_data,
   timestamp_column = c(date),
   prediction_length = 14)

 # ------------------------------------------------------------------------------
 # Covariates using recipes

 rec <-
  recipe(ridership ~ ., data = prior_data) |>
  update_role(date, new_role = "time")

 mod_3 <- brulee_chronos(rec, data = prior_data, prediction_length = 14)
}

## End(Not run)
```

`brulee_linear_reg` *Fit a linear regression model*

Description

`brulee_linear_reg()` fits a linear regression model.

Usage

```
brulee_linear_reg(x, ...)

## Default S3 method:
brulee_linear_reg(x, ...)

## S3 method for class 'data.frame'
brulee_linear_reg(
```

```
  x,
  y,
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  stop_iter = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'matrix'
brulee_linear_reg(
  x,
  y,
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  stop_iter = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'formula'
brulee_linear_reg(
  formula,
  data,
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  stop_iter = 5,
  verbose = FALSE,
```

```
  device = NULL,
  ...
)

## S3 method for class 'recipe'
brulee_linear_reg(
  x,
  data,
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  stop_iter = 5,
  verbose = FALSE,
  device = NULL,
  ...
)
```

Arguments

x	Depending on the context: • A **data frame** of predictors. • A **matrix** of predictors. • A **recipe** specifying a set of preprocessing steps created from `recipes::recipe()`. The predictor data should be standardized (e.g. centered or scaled).
...	Options to pass to the learning rate schedulers via `set_learn_rate()`. For example, the `reduction` or `steps` arguments to `schedule_step()` could be passed here.
y	When `x` is a **data frame** or **matrix**, `y` is the outcome specified as: • A **data frame** with 1 numeric column. • A **matrix** with 1 numeric column. • A numeric **vector**.
epochs	An integer for the number of epochs of training.
penalty	The amount of weight decay (i.e., L2 regularization).
mixture	Proportion of Lasso Penalty (type: double, default: 0.0). A value of mixture = 1 corresponds to a pure lasso model, while mixture = 0 indicates ridge regression (a.k.a weight decay). Must be zero for optimizers `"ADAMw"`, `"RMSprop"`, `"Adadelta"`.
validation	The proportion of the data randomly assigned to a validation set.
optimizer	The method used in the optimization procedure. Possible choices are `"SGD"`, `"ADAMw"`, `"Adadelta"`, `"Adagrad"`, `"RMSprop"`, and `"LBFGS"`. `"LBFGS"` is the only second-order method, does not use batches, and is the default.

learn_rate	A positive number that controls the initial rapidity that the model moves along the descent path. Values around 0.1 or less are typical.
momentum	A positive number usually on [0.50, 0.99] for the momentum parameter in gradient descent. (optimizers "SGD", and "RMSprop" only, ignored otherwise).
batch_size	An integer for the number of training set points in each batch. (optimizer != "LBFGS" only, ignored otherwise)
stop_iter	A non-negative integer for how many iterations with no improvement before stopping.
verbose	A logical that prints out the iteration history.
device	A single character string for the device to train on (e.g., "cpu" or "cuda" for GPU). If NULL, the function will use the GPU if available, otherwise CPU. See training_efficiency.
formula	A formula specifying the outcome term(s) on the left-hand side, and the predictor term(s) on the right-hand side.
data	When a **recipe** or **formula** is used, data is specified as: • A **data frame** containing both the predictors and the outcome.

Details

This function fits a linear combination of coefficients and predictors to model the numeric outcome. The training process optimizes the mean squared error loss function.

The function internally standardizes the outcome data to have mean zero and a standard deviation of one. The prediction function creates predictions on the original scale.

By default, training halts when the validation loss increases for at least step_iter iterations. If validation = 0 the training set loss is used.

The *predictors* data should all be numeric and encoded in the same units (e.g. standardized to the same range or distribution). If there are factor predictors, use a recipe or formula to create indicator variables (or some other method) to make them numeric. Predictors should be in the same units before training.

The model objects are saved for each epoch so that the number of epochs can be efficiently tuned. Both the coef() and predict() methods for this model have an epoch argument (which defaults to the epoch with the best loss value).

The use of the L1 penalty (a.k.a. the lasso penalty) does *not* force parameters to be strictly zero (as it does in packages such as **glmnet**). The zeroing out of parameters is a specific feature the optimization method used in those packages.

Value

A brulee_linear_reg object with elements:

- models_obj: a serialized raw vector for the torch module.
- estimates: a list of matrices with the model parameter estimates per epoch. The first element is epoch zero (the randomly initialized parameters before training), so the list has epochs + 1 elements.

- best_epoch: an integer for the epoch with the smallest loss. Since estimates and loss include epoch zero, this epoch's values are at position best_epoch + 1 in those objects.
- loss: A vector of loss values (MSE) at each epoch, starting with epoch zero.
- dim: A list of data dimensions.
- y_stats: A list of summary statistics for numeric outcomes.
- parameters: A list of some tuning parameter values.
- blueprint: The hardhat blueprint data.

See Also

predict.brulee_linear_reg(), coef.brulee_linear_reg(), autoplot.brulee_linear_reg()

Examples

```
if (torch::torch_is_installed()  && rlang::is_installed(c("recipes", "yardstick", "modeldata"))) {

 ## -----------------------------------------------------------------------------

 library(recipes)
 library(yardstick)

 data(ames, package = "modeldata")

 ames$Sale_Price <- log10(ames$Sale_Price)

 set.seed(122)
 in_train <- sample(seq_len(nrow(ames)), 2000)
 ames_train <- ames[ in_train,]
 ames_test  <- ames[-in_train,]


 # Using matrices
 set.seed(1)
 brulee_linear_reg(x = as.matrix(ames_train[, c("Longitude", "Latitude")]),
                   y = ames_train$Sale_Price,
                   penalty = 0.10, epochs = 1, batch_size = 64)

 # Using recipe
 library(recipes)

 ames_rec <-
  recipe(Sale_Price ~ Bldg_Type + Neighborhood + Year_Built + Gr_Liv_Area +
         Full_Bath + Year_Sold + Lot_Area + Central_Air + Longitude + Latitude,
         data = ames_train) |>
    # Transform some highly skewed predictors
    step_BoxCox(Lot_Area, Gr_Liv_Area) |>
    # Lump some rarely occurring categories into "other"
    step_other(Neighborhood, threshold = 0.05)  |>
    # Encode categorical predictors as binary.
    step_dummy(all_nominal_predictors(), one_hot = TRUE) |>
```

```
   # Add an interaction effect:
   step_interact(~ starts_with("Central_Air"):Year_Built) |>
   step_zv(all_predictors()) |>
   step_normalize(all_numeric_predictors())

set.seed(2)
fit <- brulee_linear_reg(ames_rec, data = ames_train, epochs = 5)
fit

autoplot(fit)

library(ggplot2)

predict(fit, ames_test) |>
  bind_cols(ames_test) |>
  ggplot(aes(x = .pred, y = Sale_Price)) +
  geom_abline(col = "green") +
  geom_point(alpha = 0.3) +
  lims(x = c(4, 6), y = c(4, 6)) +
  coord_fixed(ratio = 1)

library(yardstick)
predict(fit, ames_test) |>
  bind_cols(ames_test) |>
  rmse(Sale_Price, .pred)

}
```

`brulee_logistic_reg` *Fit a logistic regression model*

Description

`brulee_logistic_reg()` fits a model.

Usage

```
brulee_logistic_reg(x, ...)

## Default S3 method:
brulee_logistic_reg(x, ...)

## S3 method for class 'data.frame'
brulee_logistic_reg(
  x,
  y,
```

```
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'matrix'
brulee_logistic_reg(
  x,
  y,
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'formula'
brulee_logistic_reg(
  formula,
  data,
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
```

```
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'recipe'
brulee_logistic_reg(
  x,
  data,
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  verbose = FALSE,
  device = NULL,
  ...
)
```

Arguments

x	Depending on the context: • A **data frame** of predictors. • A **matrix** of predictors. • A **recipe** specifying a set of preprocessing steps created from `recipes::recipe()`. The predictor data should be standardized (e.g. centered or scaled).
...	Options to pass to the learning rate schedulers via `set_learn_rate()`. For example, the `reduction` or `steps` arguments to `schedule_step()` could be passed here.
y	When `x` is a **data frame** or **matrix**, `y` is the outcome specified as: • A **data frame** with 1 factor column (with two levels). • A **matrix** with 1 factor column (with two levels). • A factor **vector** (with two levels).
epochs	An integer for the number of epochs of training.
penalty	The amount of weight decay (i.e., L2 regularization).
mixture	Proportion of Lasso Penalty (type: double, default: 0.0). A value of mixture = 1 corresponds to a pure lasso model, while mixture = 0 indicates ridge regression (a.k.a weight decay). Must be zero for optimizers `"ADAMw"`, `"RMSprop"`, `"Adadelta"`.
validation	The proportion of the data randomly assigned to a validation set.

`optimizer` The method used in the optimization procedure. Possible choices are `"SGD"`, `"ADAMw"`, `"Adadelta"`, `"Adagrad"`, `"RMSprop"`, and `"LBFGS"`. `"LBFGS"` is the only second-order method, does not use batches, and is the default.

`learn_rate` A positive number that controls the initial rapidity that the model moves along the descent path. Values around 0.1 or less are typical.

`momentum` A positive number usually on `[0.50, 0.99]` for the momentum parameter in gradient descent. (optimizers `"SGD"`, and `"RMSprop"` only, ignored otherwise).

`batch_size` An integer for the number of training set points in each batch. (`optimizer != "LBFGS"` only, ignored otherwise)

`class_weights` Numeric class weights (classification only). The value can be:

- A named numeric vector (in any order) where the names are the outcome factor levels.
- An unnamed numeric vector assumed to be in the same order as the outcome factor levels.
- A single numeric value for the least frequent class in the training data and all other classes receive a weight of one.

`stop_iter` A non-negative integer for how many iterations with no improvement before stopping.

`verbose` A logical that prints out the iteration history.

`device` A single character string for the device to train on (e.g., `"cpu"` or `"cuda"` for GPU). If `NULL`, the function will use the GPU if available, otherwise CPU. See training_efficiency.

`formula` A formula specifying the outcome term(s) on the left-hand side, and the predictor term(s) on the right-hand side.

`data` When a **recipe** or **formula** is used, `data` is specified as:

- A **data frame** containing both the predictors and the outcome.

Details

This function fits a linear combination of coefficients and predictors to model the log odds of the classes. The training process optimizes the cross-entropy loss function (a.k.a Bernoulli loss).

By default, training halts when the validation loss increases for at least `step_iter` iterations. If `validation = 0` the training set loss is used.

The *predictors* data should all be numeric and encoded in the same units (e.g. standardized to the same range or distribution). If there are factor predictors, use a recipe or formula to create indicator variables (or some other method) to make them numeric. Predictors should be in the same units before training.

The model objects are saved for each epoch so that the number of epochs can be efficiently tuned. Both the `coef()` and `predict()` methods for this model have an epoch argument (which defaults to the epoch with the best loss value).

The use of the L1 penalty (a.k.a. the lasso penalty) does *not* force parameters to be strictly zero (as it does in packages such as **glmnet**). The zeroing out of parameters is a specific feature the optimization method used in those packages.

Value

A `brulee_logistic_reg` object with elements:

- `models_obj`: a serialized raw vector for the torch module.
- `estimates`: a list of matrices with the model parameter estimates per epoch. The first element is epoch zero (the randomly initialized parameters before training), so the list has `epochs + 1` elements.
- `best_epoch`: an integer for the epoch with the smallest loss. Since `estimates` and `loss` include epoch zero, this epoch's values are at position `best_epoch + 1` in those objects.
- `loss`: A vector of loss values (MSE for regression, negative log- likelihood for classification) at each epoch, starting with epoch zero.
- `dim`: A list of data dimensions.
- `parameters`: A list of some tuning parameter values.
- `blueprint`: The `hardhat` blueprint data.

See Also

`predict.brulee_logistic_reg()`, `coef.brulee_logistic_reg()`, `autoplot.brulee_logistic_reg()`

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "yardstick", "modeldata"))) {

 library(recipes)
 library(yardstick)

 ## -----------------------------------------------------------------------------
 # increase # epochs to get better results

 data(cells, package = "modeldata")

 cells$case <- NULL

 set.seed(122)
 in_train <- sample(seq_len(nrow(cells)), 1000)
 cells_train <- cells[ in_train,]
 cells_test  <- cells[-in_train,]

 # Using matrices
 set.seed(1)
 brulee_logistic_reg(x = as.matrix(cells_train[, c("fiber_width_ch_1", "width_ch_1")]),
                     y = cells_train$class,
                     penalty = 0.10, epochs = 3)

 # Using recipe
 library(recipes)

 cells_rec <-
```

```
 recipe(class ~ ., data = cells_train) |>
 # Transform some highly skewed predictors
 step_YeoJohnson(all_numeric_predictors()) |>
 step_normalize(all_numeric_predictors()) |>
 step_pca(all_numeric_predictors(), num_comp = 10)

set.seed(2)
fit <- brulee_logistic_reg(cells_rec, data = cells_train,
                           penalty = 0.01, epochs = 5)
fit

autoplot(fit)

library(yardstick)
predict(fit, cells_test, type = "prob") |>
 bind_cols(cells_test) |>
 roc_auc(class, .pred_PS)
}
```

---

brulee_mlp *Fit neural networks*

Description

brulee_mlp() fits neural network models. Multiple layers can be used. For working with two-layer networks in tidymodels, brulee_mlp_two_layer() can be helpful for specifying tuning parameters as scalars.

Usage

```
brulee_mlp(x, ...)

## Default S3 method:
brulee_mlp(x, ...)

## S3 method for class 'data.frame'
brulee_mlp(
  x,
  y,
  epochs = 100L,
  hidden_units = 3L,
  activation = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  validation = 0.1,
  optimizer = "LBFGS",
```

```
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'matrix'
brulee_mlp(
  x,
  y,
  epochs = 100L,
  hidden_units = 3L,
  activation = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'formula'
brulee_mlp(
  formula,
  data,
  epochs = 100L,
  hidden_units = 3L,
  activation = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
```

```
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'recipe'
brulee_mlp(
  x,
  data,
  epochs = 100L,
  hidden_units = 3L,
  activation = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

brulee_mlp_two_layer(x, ...)

## Default S3 method:
brulee_mlp_two_layer(x, ...)

## S3 method for class 'data.frame'
brulee_mlp_two_layer(
  x,
```

```
  y,
  epochs = 100L,
  hidden_units = 3L,
  hidden_units_2 = 3L,
  activation = "relu",
  activation_2 = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'matrix'
brulee_mlp_two_layer(
  x,
  y,
  epochs = 100L,
  hidden_units = 3L,
  hidden_units_2 = 3L,
  activation = "relu",
  activation_2 = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
```

```
  ...
)

## S3 method for class 'formula'
brulee_mlp_two_layer(
  formula,
  data,
  epochs = 100L,
  hidden_units = 3L,
  hidden_units_2 = 3L,
  activation = "relu",
  activation_2 = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'recipe'
brulee_mlp_two_layer(
  x,
  data,
  epochs = 100L,
  hidden_units = 3L,
  hidden_units_2 = 3L,
  activation = "relu",
  activation_2 = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
```

```
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)
```

Arguments

x	Depending on the context: • A **data frame** of predictors. • A **matrix** of predictors. • A **recipe** specifying a set of preprocessing steps created from `recipes::recipe()`. The predictor data should be standardized (e.g. centered or scaled).
...	Options to pass to the learning rate schedulers via `set_learn_rate()`. For example, the `reduction` or `steps` arguments to `schedule_step()` could be passed here.
y	When `x` is a **data frame** or **matrix**, `y` is the outcome specified as: • A **data frame** with 1 column (numeric or factor). • A **matrix** with numeric column (numeric or factor). • A **vector** (numeric or factor).
epochs	An integer for the number of epochs of training.
hidden_units	An integer for the number of hidden units, or a vector of integers. If a vector of integers, the model will have `length(hidden_units)` layers each with `hidden_units[i]` hidden units.
activation	A character vector for the activation function (such as "relu", "tanh", "sigmoid", and so on). See `brulee_activations()` for a list of possible values. If `hidden_units` is a vector, `activation` can be a character vector with length equals to `length(hidden_units)` specifying the activation for each hidden layer.
penalty	The amount of weight decay (i.e., L2 regularization).
mixture	Proportion of Lasso Penalty (type: double, default: 0.0). A value of mixture = 1 corresponds to a pure lasso model, while mixture = 0 indicates ridge regression (a.k.a weight decay). Must be zero for optimizers `"ADAMw"`, `"RMSprop"`, `"Adadelta"`.
dropout	The proportion of parameters set to zero.
validation	The proportion of the data randomly assigned to a validation set.
optimizer	The method used in the optimization procedure. Possible choices are `"SGD"`, `"ADAMw"`, `"Adadelta"`, `"Adagrad"`, `"RMSprop"`, and `"LBFGS"`. `"LBFGS"` is the only second-order method and does not use batches.
learn_rate	A positive number that controls the initial rapidity that the model moves along the descent path. Values around 0.1 or less are typical.

`rate_schedule` A single character value for how the learning rate should change as the optimization proceeds. Possible values are `"none"` (the default), `"decay_time"`, `"decay_expo"`, `"cyclic"` and `"step"`. See `schedule_decay_time()` for more details.

`momentum` A positive number usually on `[0.50, 0.99]` for the momentum parameter in gradient descent. (optimizers `"SGD"`, and `"RMSprop"` only, ignored otherwise).

`batch_size` An integer for the number of training set points in each batch. (`optimizer != "LBFGS"` only, ignored otherwise)

`class_weights` Numeric class weights (classification only). The value can be:

- A named numeric vector (in any order) where the names are the outcome factor levels.
- An unnamed numeric vector assumed to be in the same order as the outcome factor levels.
- A single numeric value for the least frequent class in the training data and all other classes receive a weight of one.

`stop_iter` A non-negative integer for how many iterations with no improvement before stopping.

`grad_norm_clip, grad_value_clip` Two numeric values, possibly `Inf`, that prevents the gradient's values or norm(s) from exceeding the specified value. This can be helpful if training stops early with the message that `"Loss is NaN at epoch x Training is stopped."`

`verbose` A logical that prints out the iteration history.

`device` A single character string for the device to train on (e.g., `"cpu"` or `"cuda"` for GPU). If `NULL`, the function will use the GPU if available, otherwise CPU. See training_efficiency.

`formula` A formula specifying the outcome term(s) on the left-hand side, and the predictor term(s) on the right-hand side.

`data` When a **recipe** or **formula** is used, `data` is specified as:

- A **data frame** containing both the predictors and the outcome.

`hidden_units_2` An integer for the number of hidden units for a second layer.

`activation_2` A character vector for the activation function for a second layer.

Details

This function fits feed-forward neural network models for regression (when the outcome is a number) or classification (a factor). For regression, the mean squared error is optimized and cross-entropy is the loss function for classification.

When the outcome is a number, the function internally standardizes the outcome data to have mean zero and a standard deviation of one. The prediction function creates predictions on the original scale.

By default, training halts when the validation loss increases for at least `step_iter` iterations. If `validation = 0` the training set loss is used.

The *predictors* data should all be numeric and encoded in the same units (e.g. standardized to the same range or distribution). If there are factor predictors, use a recipe or formula to create indicator

variables (or some other method) to make them numeric. Predictors should be in the same units before training.

The model objects are saved for each epoch so that the number of epochs can be efficiently tuned. Both the `coef()` and `predict()` methods for this model have an `epoch` argument (which defaults to the epoch with the best loss value).

The use of the L1 penalty (a.k.a. the lasso penalty) does *not* force parameters to be strictly zero (as it does in packages such as **glmnet**). The zeroing out of parameters is a specific feature the optimization method used in those packages.

Learning Rates:

The learning rate can be set to constant (the default) or dynamically set via a learning rate scheduler (via the `rate_schedule`). Using `rate_schedule = 'none'` uses the `learn_rate` argument. Otherwise, any arguments to the schedulers can be passed via `...`.

Value

A `brulee_mlp` object with elements:

- `models_obj`: a serialized raw vector for the torch module.
- `estimates`: a list of matrices with the model parameter estimates per epoch. The first element is epoch zero (the randomly initialized parameters before training), so the list has `epochs + 1` elements.
- `best_epoch`: an integer for the epoch with the smallest loss. Since `estimates` and `loss` include epoch zero, this epoch's values are at position `best_epoch + 1` in those objects.
- `loss`: A vector of loss values (MSE for regression, negative log- likelihood for classification) at each epoch, starting with epoch zero.
- `dim`: A list of data dimensions.
- `y_stats`: A list of summary statistics for numeric outcomes.
- `parameters`: A list of some tuning parameter values.
- `blueprint`: The `hardhat` blueprint data.

References

adagrad (adaptive gradient algorithm): Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7).

adadelta: Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. arXiv preprint arXiv:1212.5701.

ADAMw: Loshchilov, I., & Hutter, F. (2017). Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101.

See Also

`predict.brulee_mlp()`, `coef.brulee_mlp()`, `autoplot.brulee_mlp()`

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "yardstick", "modeldata"))) {

 ## -----------------------------------------------------------------------------
 # regression examples (increase # epochs to get better results)

 data(ames, package = "modeldata")

 ames$Sale_Price <- log10(ames$Sale_Price)

 set.seed(122)
 in_train <- sample(seq_len(nrow(ames)), 2000)
 ames_train <- ames[ in_train,]
 ames_test  <- ames[-in_train,]


 # Using matrices
 set.seed(1)
 fit <-
   brulee_mlp(x = as.matrix(ames_train[, c("Longitude", "Latitude")]),
              y = ames_train$Sale_Price, penalty = 0.10)

 # Using recipe
 library(recipes)

 ames_rec <-
  recipe(Sale_Price ~ Bldg_Type + Neighborhood + Year_Built + Gr_Liv_Area +
         Full_Bath + Year_Sold + Lot_Area + Central_Air + Longitude + Latitude,
         data = ames_train) |>
   # Transform some highly skewed predictors
   step_BoxCox(Lot_Area, Gr_Liv_Area) |>
   # Lump some rarely occurring categories into "other"
   step_other(Neighborhood, threshold = 0.05)  |>
   # Encode categorical predictors as binary.
   step_dummy(all_nominal_predictors(), one_hot = TRUE) |>
   # Add an interaction effect:
   step_interact(~ starts_with("Central_Air"):Year_Built) |>
   step_zv(all_predictors()) |>
   step_normalize(all_numeric_predictors())

 set.seed(2)
 fit <- brulee_mlp(ames_rec, data = ames_train, hidden_units = 20,
                   dropout = 0.05, rate_schedule = "cyclic", step_size = 4)
 fit

 autoplot(fit)

 library(ggplot2)

 predict(fit, ames_test) |>
   bind_cols(ames_test) |>
```

```
  ggplot(aes(x = .pred, y = Sale_Price)) +
  geom_abline(col = "green") +
  geom_point(alpha = 0.3) +
  lims(x = c(4, 6), y = c(4, 6)) +
  coord_fixed(ratio = 1)

library(yardstick)
predict(fit, ames_test) |>
  bind_cols(ames_test) |>
  rmse(Sale_Price, .pred)

# Using multiple hidden layers and activation functions
set.seed(2)
hidden_fit <- brulee_mlp(ames_rec, data = ames_train,
                  hidden_units = c(15L, 17L), activation = c("relu", "elu"),
                  dropout = 0.05, rate_schedule = "cyclic", step_size = 4)

predict(hidden_fit, ames_test) |>
  bind_cols(ames_test) |>
  rmse(Sale_Price, .pred)

# ------------------------------------------------------------------------------
# classification

library(dplyr)
library(ggplot2)

data("parabolic", package = "modeldata")

set.seed(1)
in_train <- sample(seq_len(nrow(parabolic)), 300)
parabolic_tr <- parabolic[ in_train,]
parabolic_te <- parabolic[-in_train,]

set.seed(2)
cls_fit <- brulee_mlp(class ~ ., data = parabolic_tr, hidden_units = 2,
                      epochs = 200L, learn_rate = 0.1, activation = "elu",
                      penalty = 0.1, batch_size = 2^8, optimizer = "SGD")

summary(cls_fit)

autoplot(cls_fit)

grid_points <- seq(-4, 4, length.out = 100)

grid <- expand.grid(X1 = grid_points, X2 = grid_points)

predict(cls_fit, grid, type = "prob") |>
 bind_cols(grid) |>
 ggplot(aes(X1, X2)) +
 geom_contour(aes(z = .pred_Class1), breaks = 1/2, col = "black") +
 geom_point(data = parabolic_te, aes(col = class))
```

```
}
```

`brulee_multinomial_reg` *Fit a multinomial regression model*

Description

`brulee_multinomial_reg()` fits a model.

Usage

```
brulee_multinomial_reg(x, ...)

## Default S3 method:
brulee_multinomial_reg(x, ...)

## S3 method for class 'data.frame'
brulee_multinomial_reg(
  x,
  y,
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'matrix'
brulee_multinomial_reg(
  x,
  y,
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
```

```
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'formula'
brulee_multinomial_reg(
  formula,
  data,
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'recipe'
brulee_multinomial_reg(
  x,
  data,
  epochs = 20L,
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "LBFGS",
  learn_rate = 1,
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  verbose = FALSE,
  device = NULL,
  ...
)
```

Arguments

x	Depending on the context: • A **data frame** of predictors. • A **matrix** of predictors. • A **recipe** specifying a set of preprocessing steps created from `recipes::recipe()`. The predictor data should be standardized (e.g. centered or scaled).
...	Options to pass to the learning rate schedulers via `set_learn_rate()`. For example, the `reduction` or `steps` arguments to `schedule_step()` could be passed here.
y	When `x` is a **data frame** or **matrix**, `y` is the outcome specified as: • A **data frame** with 1 factor column (with three or more levels). • A **matrix** with 1 factor column (with three or more levels). • A factor **vector** (with three or more levels).
epochs	An integer for the number of epochs of training.
penalty	The amount of weight decay (i.e., L2 regularization).
mixture	Proportion of Lasso Penalty (type: double, default: 0.0). A value of mixture = 1 corresponds to a pure lasso model, while mixture = 0 indicates ridge regression (a.k.a weight decay). Must be zero for optimizers `"ADAMw"`, `"RMSprop"`, `"Adadelta"`.
validation	The proportion of the data randomly assigned to a validation set.
optimizer	The method used in the optimization procedure. Possible choices are `"SGD"`, `"ADAMw"`, `"Adadelta"`, `"Adagrad"`, `"RMSprop"`, and `"LBFGS"`. `"LBFGS"` is the only second-order method, does not use batches, and is the default.
learn_rate	A positive number that controls the initial rapidity that the model moves along the descent path. Values around 0.1 or less are typical.
momentum	A positive number usually on `[0.50, 0.99]` for the momentum parameter in gradient descent. (optimizers `"SGD"`, and `"RMSprop"` only, ignored otherwise).
batch_size	An integer for the number of training set points in each batch. (`optimizer != "LBFGS"` only, ignored otherwise)
class_weights	Numeric class weights (classification only). The value can be: • A named numeric vector (in any order) where the names are the outcome factor levels. • An unnamed numeric vector assumed to be in the same order as the outcome factor levels. • A single numeric value for the least frequent class in the training data and all other classes receive a weight of one.
stop_iter	A non-negative integer for how many iterations with no improvement before stopping.
verbose	A logical that prints out the iteration history.
device	A single character string for the device to train on (e.g., `"cpu"` or `"cuda"` for GPU). If `NULL`, the function will use the GPU if available, otherwise CPU. See training_efficiency.

`formula` A formula specifying the outcome term(s) on the left-hand side, and the predictor term(s) on the right-hand side.

`data` When a **recipe** or **formula** is used, `data` is specified as:

- A **data frame** containing both the predictors and the outcome.

Details

This function fits a linear combination of coefficients and predictors to model the log of the class probabilities. The training process optimizes the cross-entropy loss function.

By default, training halts when the validation loss increases for at least `step_iter` iterations. If `validation = 0` the training set loss is used.

The *predictors* data should all be numeric and encoded in the same units (e.g. standardized to the same range or distribution). If there are factor predictors, use a recipe or formula to create indicator variables (or some other method) to make them numeric. Predictors should be in the same units before training.

The model objects are saved for each epoch so that the number of epochs can be efficiently tuned. Both the `coef()` and `predict()` methods for this model have an `epoch` argument (which defaults to the epoch with the best loss value).

The use of the L1 penalty (a.k.a. the lasso penalty) does *not* force parameters to be strictly zero (as it does in packages such as **glmnet**). The zeroing out of parameters is a specific feature the optimization method used in those packages.

Value

A `brulee_multinomial_reg` object with elements:

- `models_obj`: a serialized raw vector for the torch module.
- `estimates`: a list of matrices with the model parameter estimates per epoch. The first element is epoch zero (the randomly initialized parameters before training), so the list has `epochs + 1` elements.
- `best_epoch`: an integer for the epoch with the smallest loss. Since `estimates` and `loss` include epoch zero, this epoch's values are at position `best_epoch + 1` in those objects.
- `loss`: A vector of loss values (MSE for regression, negative log- likelihood for classification) at each epoch, starting with epoch zero.
- `dim`: A list of data dimensions.
- `parameters`: A list of some tuning parameter values.
- `blueprint`: The `hardhat` blueprint data.

See Also

`predict.brulee_multinomial_reg()`, `coef.brulee_multinomial_reg()`, `autoplot.brulee_multinomial_reg()`

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "yardstick", "modeldata"))) {

  library(recipes)
  library(yardstick)

  data(penguins, package = "modeldata")

  penguins <- penguins |> na.omit()

  set.seed(122)
  in_train <- sample(seq_len(nrow(penguins)), 200)
  penguins_train <- penguins[ in_train,]
  penguins_test  <- penguins[-in_train,]

  rec <- recipe(island ~ ., data = penguins_train) |>
    step_dummy(species, sex) |>
    step_normalize(all_predictors())

  set.seed(3)
  fit <- brulee_multinomial_reg(rec, data = penguins_train, epochs = 5)
  fit

  predict(fit, penguins_test) |>
    bind_cols(penguins_test) |>
    conf_mat(island, .pred_class)
}
```

---

brulee_resnet *Fit residual neural networks (ResNet)*

Description

`brulee_resnet()` fits residual network models with skip connections.

Usage

```
brulee_resnet(x, ...)

## Default S3 method:
brulee_resnet(x, ...)

## S3 method for class 'data.frame'
brulee_resnet(
  x,
  y,
```

```
  epochs = 100L,
  hidden_units = 3L,
  bottleneck_units = hidden_units,
  residual_at = NULL,
  activation = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'matrix'
brulee_resnet(
  x,
  y,
  epochs = 100L,
  hidden_units = 3L,
  bottleneck_units = hidden_units,
  residual_at = NULL,
  activation = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
```

```
)

## S3 method for class 'formula'
brulee_resnet(
  formula,
  data,
  epochs = 100L,
  hidden_units = 3L,
  bottleneck_units = hidden_units,
  residual_at = NULL,
  activation = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'recipe'
brulee_resnet(
  x,
  data,
  epochs = 100L,
  hidden_units = 3L,
  bottleneck_units = hidden_units,
  residual_at = NULL,
  activation = "relu",
  penalty = 0.001,
  mixture = 0,
  dropout = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.01,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
```

```
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  ...
)
```

Arguments

x	Depending on the context: • A **data frame** of predictors. • A **matrix** of predictors. • A **recipe** specifying a set of preprocessing steps created from `recipes::recipe()`. The predictor data should be standardized (e.g. centered or scaled).
...	Options to pass to the learning rate schedulers via `set_learn_rate()`. For example, the `reduction` or `steps` arguments to `schedule_step()` could be passed here.
y	When `x` is a **data frame** or **matrix**, `y` is the outcome specified as: • A **data frame** with 1 column (numeric or factor). • A **matrix** with numeric column (numeric or factor). • A **vector** (numeric or factor).
epochs	An integer for the number of epochs of training.
hidden_units	An integer vector specifying the number of hidden units in each layer. The length of this vector determines the number of layers. Each value must be >= 1.
bottleneck_units	An integer vector specifying the intermediate dimension within each layer. Must have the same length as `hidden_units`. Each value must be >= 2.
residual_at	An integer vector specifying which layer indices should have residual (skip) connections. For example, `residual_at = c(2, 4)` creates residual connections after layers 2 and 4, forming two residual blocks (layers 1-2 and 3-4). If `NULL` (default), every layer gets its own skip connection. Use `integer(0)` for no residual connections (i.e., a purely feed-forward model only).
activation	A character vector for the activation function (such as "relu", "tanh", "sigmoid", and so on). See `brulee_activations()` for a list of possible values. If `hidden_units` is a vector, `activation` can be a character vector with length equals to `length(hidden_units)` specifying the activation for each hidden layer.
penalty	The amount of weight decay (i.e., L2 regularization).
mixture	Proportion of Lasso Penalty (type: double, default: 0.0). A value of mixture = 1 corresponds to a pure lasso model, while mixture = 0 indicates ridge regression (a.k.a weight decay). Must be zero for optimizers `"ADAMw"`, `"RMSprop"`, `"Adadelta"`.
dropout	The proportion of parameters set to zero.
validation	The proportion of the data randomly assigned to a validation set.

`optimizer` The method used in the optimization procedure. Possible choices are `"SGD"`, `"ADAMw"`, `"Adadelta"`, `"Adagrad"`, `"RMSprop"`, and `"LBFGS"`. `"LBFGS"` is the only second-order method and does not use batches.

`learn_rate` A positive number that controls the initial rapidity that the model moves along the descent path. Values around 0.1 or less are typical.

`rate_schedule` A single character value for how the learning rate should change as the optimization proceeds. Possible values are `"none"` (the default), `"decay_time"`, `"decay_expo"`, `"cyclic"` and `"step"`. See `schedule_decay_time()` for more details.

`momentum` A positive number usually on `[0.50, 0.99]` for the momentum parameter in gradient descent. (optimizers `"SGD"`, and `"RMSprop"` only, ignored otherwise).

`batch_size` An integer for the number of training set points in each batch. (`optimizer != "LBFGS"` only, ignored otherwise)

`class_weights` Numeric class weights (classification only). The value can be:

- A named numeric vector (in any order) where the names are the outcome factor levels.
- An unnamed numeric vector assumed to be in the same order as the outcome factor levels.
- A single numeric value for the least frequent class in the training data and all other classes receive a weight of one.

`stop_iter` A non-negative integer for how many iterations with no improvement before stopping.

`grad_norm_clip, grad_value_clip`
Two numeric values, possibly `Inf`, that prevents the gradient's values or norm(s) from exceeding the specified value. This can be helpful if training stops early with the message that `"Loss is NaN at epoch x Training is stopped."`

`verbose` A logical that prints out the iteration history.

`device` A single character string for the device to train on (e.g., `"cpu"` or `"cuda"` for GPU). If `NULL`, the function will use the GPU if available, otherwise CPU. See training_efficiency.

`formula` A formula specifying the outcome term(s) on the left-hand side, and the predictor term(s) on the right-hand side.

`data` When a **recipe** or **formula** is used, `data` is specified as:

- A **data frame** containing both the predictors and the outcome.

Details

This function fits residual network (ResNet) models for regression (when the outcome is a number) or classification (a factor). ResNets use skip connections that add the input of a block to its output, allowing gradients to flow more easily through deep networks. For regression, the mean squared error is optimized and cross-entropy is the loss function for classification.

Architecture:
The network consists of a sequence of layers, each with batch normalization, two linear transformations (with an intermediate bottleneck dimension), and activation functions. Residual (skip) connections can be placed at specified layers via the `residual_at` parameter.

Each layer follows this pattern:

- Batch normalization (input dimension)
- Linear transformation (input dimension -> `bottleneck_units[i]`)
- Activation function (ReLU by default)
- Dropout (if specified)
- Linear transformation (`bottleneck_units[i]` -> `hidden_units[i]`)
- Dropout (if specified)

When a residual connection is specified at layer `i` via `residual_at`, the output of layer `i` is added to the input from the start of that residual block. If dimensions don't match, a linear projection is automatically added.

Residual Blocks:

The `residual_at` parameter defines where skip connections occur:

- `residual_at = 3` creates one block spanning layers 1-3
- `residual_at = c(2, 4)` creates two blocks: layers 1-2 and layers 3-4
- `residual_at = NULL` (default) places a skip connection at every layer
- `residual_at = integer(0)` creates no residual connections (a purely feed-forward model)

Learning Rates:

The learning rate can be set to constant (the default) or dynamically set via a learning rate scheduler (via the `rate_schedule`). Using `rate_schedule = 'none'` uses the `learn_rate` argument. Otherwise, any arguments to the schedulers can be passed via `...`.

Other Notes:

When the outcome is a number, the function internally standardizes the outcome data to have mean zero and a standard deviation of one. The prediction function creates predictions on the original scale.

By default, training halts when the validation loss increases for at least `step_iter` iterations. If `validation = 0` the training set loss is used.

The *predictors* data should all be numeric and encoded in the same units (e.g. standardized to the same range or distribution). If there are factor predictors, use a recipe or formula to create indicator variables (or some other method) to make them numeric. Predictors should be in the same units before training.

The model objects are saved for each epoch so that the number of epochs can be efficiently tuned. Both the `coef()` and `predict()` methods for this model have an `epoch` argument (which defaults to the epoch with the best loss value).

The use of the L1 penalty (a.k.a. the lasso penalty) does *not* force parameters to be strictly zero (as it does in packages such as **glmnet**). The zeroing out of parameters is a specific feature the optimization method used in those packages.

Value

A `brulee_resnet` object with elements:

- `models_obj`: a serialized raw vector for the torch module.

- `estimates`: a list of matrices with the model parameter estimates per epoch. The first element is epoch zero (the randomly initialized parameters before training), so the list has `epochs + 1` elements.
- `best_epoch`: an integer for the epoch with the smallest loss. Since `estimates` and `loss` include epoch zero, this epoch's values are at position `best_epoch + 1` in those objects.
- `loss`: A vector of loss values (MSE for regression, negative log- likelihood for classification) at each epoch, starting with epoch zero.
- `dim`: A list of data dimensions.
- `y_stats`: A list of summary statistics for numeric outcomes.
- `parameters`: A list of some tuning parameter values.
- `blueprint`: The `hardhat` blueprint data.

References

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Identity mappings in deep residual networks. In *European conference on computer vision* (pp. 630-645). Springer, Cham.

Gorishniy, Y., Rubachev, I., Khrulkov, V., & Babenko, A. (2021). Revisiting deep learning models for tabular data. *Advances in neural information processing systems*, 34, 18932-18943.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4510-4520).

See Also

`predict.brulee_resnet()`, `coef.brulee_resnet()`, `autoplot.brulee_resnet()`

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "yardstick", "modeldata"))) {

 ## -----------------------------------------------------------------------------
 # regression examples (increase # epochs to get better results)

 data(ames, package = "modeldata")

 ames$Sale_Price <- log10(ames$Sale_Price)

 set.seed(122)
 in_train <- sample(seq_len(nrow(ames)), 2000)
 ames_train <- ames[ in_train,]
 ames_test  <- ames[-in_train,]

 # Using recipe
 library(recipes)
```

```
ames_rec <-
 recipe(Sale_Price ~ Longitude + Latitude, data = ames_train) |>
   step_normalize(all_numeric_predictors())

set.seed(2)
fit <- brulee_resnet(ames_rec, data = ames_train,
                     hidden_units = c(20, 10), bottleneck_units = c(15, 8),
                     residual_at = 2,
                     epochs = 50, batch_size = 32)
fit

summary(fit)

autoplot(fit)

library(yardstick)
predict(fit, ames_test) |>
  bind_cols(ames_test) |>
  rmse(Sale_Price, .pred)

# ------------------------------------------------------------------------------
# classification

library(dplyr)

data("parabolic", package = "modeldata")

set.seed(1)
in_train <- sample(seq_len(nrow(parabolic)), 300)
parabolic_tr <- parabolic[ in_train,]
parabolic_te <- parabolic[-in_train,]

set.seed(2)
cls_fit <- brulee_resnet(class ~ ., data = parabolic_tr,
                         hidden_units = c(8, 5), bottleneck_units = c(6, 4),
                         residual_at = 1:2,
                         epochs = 200L, learn_rate = 0.1, activation = "elu",
                         penalty = 0.1, batch_size = 2^8)
autoplot(cls_fit)

predict(cls_fit, parabolic_te, type = "prob") |>
  bind_cols(parabolic_te) |>
  roc_auc(class, .pred_Class1)

}
```

---

brulee_rln *Fit Regularization Learning Networks (RLN)*

Description

brulee_rln() fits a single-hidden-layer neural network where each weight learns its own adaptive regularization coefficient.

Usage

```
brulee_rln(x, ...)

## Default S3 method:
brulee_rln(x, ...)

## S3 method for class 'data.frame'
brulee_rln(
  x,
  y,
  epochs = 100L,
  hidden_units = 5L,
  penalty_type = "L1",
  penalty_average = 1e-10,
  step_rate = 1e+06,
  activation = "relu",
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.001,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  stop_iter = 20,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'matrix'
brulee_rln(
  x,
  y,
  epochs = 100L,
  hidden_units = 5L,
  penalty_type = "L1",
  penalty_average = 1e-10,
  step_rate = 1e+06,
  activation = "relu",
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.001,
  rate_schedule = "none",
  momentum = 0,
```

```
  batch_size = NULL,
  stop_iter = 20,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'formula'
brulee_rln(
  formula,
  data,
  epochs = 100L,
  hidden_units = 5L,
  penalty_type = "L1",
  penalty_average = 1e-10,
  step_rate = 1e+06,
  activation = "relu",
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.001,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  stop_iter = 20,
  verbose = FALSE,
  device = NULL,
  ...
)

## S3 method for class 'recipe'
brulee_rln(
  x,
  data,
  epochs = 100L,
  hidden_units = 5L,
  penalty_type = "L1",
  penalty_average = 1e-10,
  step_rate = 1e+06,
  activation = "relu",
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 0.001,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  stop_iter = 20,
  verbose = FALSE,
  device = NULL,
```

```
  ...
)
```

Arguments

x	Depending on the context: • A **data frame** of predictors. • A **matrix** of predictors. • A **recipe** specifying a set of preprocessing steps created from `recipes::recipe()`. The predictor data should be standardized (e.g. centered or scaled).
...	Options to pass to the learning rate schedulers via `set_learn_rate()`. For example, the `reduction` or `steps` arguments to `schedule_step()` could be passed here.
y	When `x` is a **data frame** or **matrix**, `y` is the outcome specified as: • A **data frame** with 1 column (numeric or factor). • A **matrix** with numeric column (numeric or factor). • A **vector** (numeric or factor).
epochs	An integer for the number of epochs of training.
hidden_units	An integer for the number of units in the single hidden layer. Must be >= 1.
penalty_type	A string for the regularization norm: `"L1"` (default) or `"L2"`. L1 is recommended by the original paper.
penalty_average	A positive numeric value for the target geometric mean of the per-weight regularization coefficients (Theta in Shavitt and Segal (2018)), on the natural scale. Converted to log10 scale internally. Default is `1e-10` (i.e., `10^-10`).
step_rate	A positive numeric value for the step size used to update the per-weight regularization coefficients (nu in Shavitt and Segal (2018)), on the natural scale. Converted to log10 scale internally; the multiplier applied is `10^log10(step_rate)`. Default is `1e6` (i.e., `10^6`). Both parameters are best tuned on the log10 scale.
activation	A character vector for the activation function (such as "relu", "tanh", "sigmoid", and so on). See `brulee_activations()` for a list of possible values. If `hidden_units` is a vector, `activation` can be a character vector with length equals to `length(hidden_units)` specifying the activation for each hidden layer.
validation	The proportion of the data randomly assigned to a validation set.
optimizer	The method used in the optimization procedure. Possible choices are `"SGD"`, `"ADAMw"`, `"Adadelta"`, `"Adagrad"`, `"RMSprop"`, and `"LBFGS"`. `"LBFGS"` is the only second-order method and does not use batches.
learn_rate	A positive number that controls the initial rapidity that the model moves along the descent path. Values around 0.1 or less are typical.
rate_schedule	A single character value for how the learning rate should change as the optimization proceeds. Possible values are `"none"` (the default), `"decay_time"`, `"decay_expo"`, `"cyclic"` and `"step"`. See `schedule_decay_time()` for more details.

momentum	A positive number usually on [0.50, 0.99] for the momentum parameter in gradient descent. (optimizers "SGD", and "RMSprop" only, ignored otherwise).
batch_size	An integer for the number of training set points in each batch. (optimizer != "LBFGS" only, ignored otherwise)
stop_iter	A non-negative integer for how many iterations with no improvement before stopping.
verbose	A logical that prints out the iteration history.
device	A single character string for the device to train on (e.g., "cpu" or "cuda" for GPU). If NULL, the function will use the GPU if available, otherwise CPU. See training_efficiency.
formula	A formula specifying the outcome term(s) on the left-hand side, and the predictor term(s) on the right-hand side.
data	When a **recipe** or **formula** is used, data is specified as: • A **data frame** containing both the predictors and the outcome.

Details

This function fits Regularization Learning Network (RLN) models for regression (numeric outcomes only). Unlike standard regularization, which applies a single global penalty, RLN learns a separate regularization coefficient for each weight in the hidden layer. After each gradient step, the per-weight coefficients (lambdas) are updated and projected to keep their mean at log10(penalty_average) * log(10).

Why Use RLN?:

RLNs are designed for tabular datasets where interpretability matters. The per-weight regularization tends to produce very sparse networks. The original paper reports eliminating up to ~99.8% of network edges and ~82% of input features. This sparsity makes it easier to identify which inputs the network considers important, and the resulting models are competitive with gradient boosted trees. The best results in the paper are achieved by ensembling RLNs with gradient boosting tree ensembles.

Architecture:

The network is a single-hidden-layer MLP:

- Linear transformation (predictors -> hidden_units)
- Activation function
- Linear transformation (hidden_units -> 1 output)

Weights are initialized with Xavier normal initialization.

RLN Update:

After each optimizer step, the per-weight regularization coefficients are updated using the gradient of the Counterfactual Loss with respect to the coefficients, then projected onto a simplex so that mean(lambda) == log10(penalty_average) * log(10). The ADAMw optimizer is the default.

Other Notes:

The outcome is internally standardized to have mean zero and standard deviation one. Predictions are returned on the original scale.

By default, training halts when the validation loss increases for at least `stop_iter` consecutive iterations. If `validation = 0` the training set loss is used. The default for `stop_iter` is higher for RLN than for other brulee models (20 vs 5) because the sparsification process takes approximately 10-20 epochs to stabilize (Shavitt & Segal, 2018); stopping too early prevents the per-weight regularization from taking effect.

Predictors should all be numeric and on comparable scales. Categorical predictors must be converted to dummy variables.

Model parameters are saved each epoch so that epoch can be tuned efficiently via the epoch argument of `predict.brulee_rln()` and `coef.brulee_rln()`.

Value

A `brulee_rln` object with elements:

- `model_obj`: a serialized raw vector for the torch module.
- `estimates`: a list of model parameter matrices per epoch. The first element is epoch zero (the randomly initialized parameters before training), so the list has `epochs + 1` elements.
- `best_epoch`: an integer for the epoch with the smallest loss. Since `estimates` and `loss` include epoch zero, this epoch's values are at position `best_epoch + 1` in those objects.
- `loss`: a numeric vector of loss values (scaled MSE) at each epoch, starting with epoch zero.
- `dims`: a list of data dimensions.
- `y_stats`: a list of mean and standard deviation for the outcome.
- `parameters`: a list of tuning parameter values.
- `device`: a character string for the device used during training.
- `blueprint`: the `hardhat` blueprint data.

References

Shavitt, I., & Segal, E. (2018). Regularization learning networks: Deep learning for tabular datasets. In *Advances in neural information processing systems* (pp. 1379-1389).

See Also

`predict.brulee_rln()`, `coef.brulee_rln()`, `autoplot.brulee_rln()`

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "yardstick", "modeldata"))) {

 data(ames, package = "modeldata")
 ames$Sale_Price <- log10(ames$Sale_Price)

 set.seed(122)
 in_train <- sample(seq_len(nrow(ames)), 2000)
 ames_train <- ames[ in_train,]
 ames_test  <- ames[-in_train,]
```

```
library(recipes)

ames_rec <-
 recipe(Sale_Price ~ Longitude + Latitude, data = ames_train) |>
   step_normalize(all_numeric_predictors())

set.seed(2)
fit <- brulee_rln(ames_rec, data = ames_train, hidden_units = 20L, epochs = 50L)
fit

autoplot(fit)

library(yardstick)
predict(fit, ames_test) |>
  bind_cols(ames_test) |>
  rmse(Sale_Price, .pred)

}
```

`brulee_saint` *Fit SAINT models for tabular data*

Description

`brulee_saint()` fits the SAINT (Self-Attention and Inter-sample Attention Transformer) model from Somepalli *et al* (2021). SAINT applies multi-head self-attention across both features (column attention) and samples within a batch (row/inter-sample attention) to learn complex feature interactions.

Usage

```
brulee_saint(x, ...)

## Default S3 method:
brulee_saint(x, ...)

## S3 method for class 'data.frame'
brulee_saint(
  x,
  y,
  epochs = 100L,
  num_embedding = 32L,
  attention_type = "both",
  num_attn_heads = 8L,
  num_attn_blocks = 6L,
  dropout_attn = 0.1,
  dropout_hidden = 0.1,
```

```
  dropout_last = 0,
  row_attention_on_predict = TRUE,
  hidden_units = 5,
  hidden_activations = "relu",
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 1e-04,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  target_token = TRUE,
  ...
)

## S3 method for class 'matrix'
brulee_saint(
  x,
  y,
  epochs = 100L,
  num_embedding = 32L,
  attention_type = "both",
  num_attn_heads = 8L,
  num_attn_blocks = 6L,
  dropout_attn = 0.1,
  dropout_hidden = 0.1,
  dropout_last = 0,
  row_attention_on_predict = TRUE,
  hidden_units = 5,
  hidden_activations = "relu",
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 1e-04,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
```

```
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  target_token = TRUE,
  ...
)

## S3 method for class 'formula'
brulee_saint(
  formula,
  data,
  epochs = 100L,
  num_embedding = 32L,
  attention_type = "both",
  num_attn_heads = 8L,
  num_attn_blocks = 6L,
  dropout_attn = 0.1,
  dropout_hidden = 0.1,
  dropout_last = 0,
  row_attention_on_predict = TRUE,
  hidden_units = 5,
  hidden_activations = "relu",
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 1e-04,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  target_token = TRUE,
  ...
)

## S3 method for class 'recipe'
brulee_saint(
  x,
  data,
  epochs = 100L,
  num_embedding = 32L,
  attention_type = "both",
  num_attn_heads = 8L,
```

```
  num_attn_blocks = 6L,
  dropout_attn = 0.1,
  dropout_hidden = 0.1,
  dropout_last = 0,
  row_attention_on_predict = TRUE,
  hidden_units = 5,
  hidden_activations = "relu",
  penalty = 0.001,
  mixture = 0,
  validation = 0.1,
  optimizer = "ADAMw",
  learn_rate = 1e-04,
  rate_schedule = "none",
  momentum = 0,
  batch_size = NULL,
  class_weights = NULL,
  stop_iter = 5,
  grad_value_clip = 5,
  grad_norm_clip = 5,
  verbose = FALSE,
  device = NULL,
  target_token = TRUE,
  ...
)
```

Arguments

x	Depending on the context: • A **data frame** of predictors. • A **matrix** of predictors. • A **recipe** specifying a set of preprocessing steps created from `recipes::recipe()`. The predictor data should be standardized (e.g. centered or scaled).
...	Options to pass to the learning rate schedulers via `set_learn_rate()`. For example, the `reduction` or `steps` arguments to `schedule_step()` could be passed here.
y	When `x` is a **data frame** or **matrix**, `y` is the outcome specified as: • A **data frame** with 1 column (numeric or factor). • A **matrix** with numeric column (numeric or factor). • A **vector** (numeric or factor).
epochs	An integer for the number of epochs of training.
num_embedding	An integer for the dimension of the initial embedding layer that encodes the original predictors. Each feature (categorical or continuous) is mapped to a vector of this dimension. Must be >= 1.
attention_type	A character string for the type of attention to use. Options are: • `"column"`: Column attention only (attends across features). This is the SAINT-s variant.

- "row": Row/inter-sample attention only (attends across samples within a batch). This is the SAINT-i variant.
- "both": Alternates between column and row attention in each transformer block. This is the full SAINT model.

num_attn_heads An integer for the number of parallel attention heads used in both column and row attention. Must be >= 1.

num_attn_blocks
An integer for the number of sequential transformer blocks (depth). Must be >= 1.

dropout_attn A number in [0, 1) for the dropout rate applied to attention weights during training.

dropout_hidden A number in [0, 1) for the dropout rate applied within the feed-forward layers of each transformer block.

dropout_last A number in [0, 1) for the dropout rate applied between the last hidden layer and the output head. Only has effect when hidden_units is not NULL. Default is 0 (no dropout).

row_attention_on_predict
A logical value. Should row (inter-sample) attention be applied during prediction? Default is TRUE, matching the training-time behavior. When FALSE, row attention is bypassed at predict time so that predictions for a given row do not depend on what other rows are in the prediction set; column attention is used on its own. This is only relevant when attention_type is "row" or "both".

hidden_units An integer vector for the number of units in optional hidden layers between the transformer backbone and the output head. When NULL (the default), no hidden layers are added and the pooled transformer output is projected directly to the output.

hidden_activations
A character vector of activation functions for the hidden layers. Must be the same length as hidden_units or a single value that will be recycled. See brulee_activations() for options.

penalty The amount of weight decay (i.e., L2 regularization).

mixture Proportion of Lasso Penalty (type: double, default: 0.0). A value of mixture = 1 corresponds to a pure lasso model, while mixture = 0 indicates ridge regression (a.k.a weight decay). Must be zero for optimizers "ADAMw", "RMSprop", "Adadelta".

validation The proportion of the data randomly assigned to a validation set.

optimizer The method used in the optimization procedure. Possible choices are "SGD", "ADAMw", "Adadelta", "Adagrad", "RMSprop", and "LBFGS". "LBFGS" is the only second-order method and does not use batches.

learn_rate A positive number that controls the initial rapidity that the model moves along the descent path. Values around 0.1 or less are typical.

rate_schedule A single character value for how the learning rate should change as the optimization proceeds. Possible values are "none" (the default), "decay_time", "decay_expo", "cyclic" and "step". See schedule_decay_time() for more details.

momentum — A positive number usually on [0.50, 0.99] for the momentum parameter in gradient descent. (optimizers "SGD", and "RMSprop" only, ignored otherwise).

batch_size — An integer for the number of training set points in each batch. (optimizer != "LBFGS" only, ignored otherwise)

class_weights — Numeric class weights (classification only). The value can be:

- A named numeric vector (in any order) where the names are the outcome factor levels.
- An unnamed numeric vector assumed to be in the same order as the outcome factor levels.
- A single numeric value for the least frequent class in the training data and all other classes receive a weight of one.

stop_iter — A non-negative integer for how many iterations with no improvement before stopping.

grad_norm_clip, grad_value_clip — Two numeric values, possibly Inf, that prevents the gradient's values or norm(s) from exceeding the specified value. This can be helpful if training stops early with the message that "Loss is NaN at epoch x Training is stopped."

verbose — A logical that prints out the iteration history.

device — A single character string for the device to train on (e.g., "cpu" or "cuda" for GPU). If NULL, the function will use the GPU if available, otherwise CPU. See training_efficiency.

target_token — A logical value. When TRUE (the default), a learnable target token ([CLS] in the SAINT paper) is prepended to each sample's feature sequence and only its final-layer embedding is fed to the head. This matches the architecture described in the SAINT paper (Section 3 and Figure 1); see the **Target Token Pooling** section in **Details**. When FALSE, the head instead consumes the concatenation of every feature token, which matches the SAINT reference implementation at https://github.com/somepago/saint.

formula — A formula specifying the outcome term(s) on the left-hand side, and the predictor term(s) on the right-hand side.

data — When a **recipe** or **formula** is used, data is specified as:

- A **data frame** containing both the predictors and the outcome.

Details

Architecture:

The SAINT architecture has three stages:

1. **Embedding layer**: Categorical features are mapped through per-feature embedding tables. Continuous features are passed through per-feature MLPs (1 -> 100 -> num_embedding). These initial embeddings are per-feature; there is a distinct embedding MLP for each predictor. Also, see the "Target Token Pooling" section below.
2. **Transformer backbone**: A stack of num_attn_blocks transformer layers. Each layer contains multi-head self-attention followed by a feed-forward network with GeGLU activation. For attention_type = "both", each block alternates between column attention (across features) and row attention (across samples within the batch).

3. **Output head**: Pools the transformer output (either the target token's embedding or the flattened concatenation of all feature embeddings, controlled by `target_token`) and projects it through optional hidden layers to the output dimension.

There is a `summary()` methods that can provide details of the architecture for a specific model fit.

Differences in this implementation and the original paper: pretraining isn't supported.

Attention Types:

- **Column attention** (`"column"`): Standard self-attention over features. Each feature embedding attends to all other feature embeddings.
- **Row attention** (`"row"`): inter-sample attention. Reshapes the batch so that each sample's full feature representation becomes a single token, then applies attention across all samples in the batch.
- **Both** (`"both"`): Alternates between column and row attention in each transformer block. This is the full SAINT model.

Target Token Pooling:

Borrowing from BERT, SAINT prepends a learnable target token (the paper calls it `[CLS]`) to each sample's feature sequence before the transformer. With embeddings `E(x_i^{(1)}), ..., E(x_i^{(n)})` for the `n` predictors of sample `i`, the input sequence becomes
`[target, E(x_i^{(1)}), E(x_i^{(2)}), ..., E(x_i^{(n)})]`
giving `n + 1` tokens of dimension `num_embedding`. The target token has no input value; it is a free parameter of the model that is trained alongside the rest of the network. Column attention lets every feature token attend to the target and vice versa, so the target slot accumulates a contextual summary of the sample. When `attention_type` is `"row"` or `"both"`, inter-sample attention sees the full `n + 1` token sequence per sample, so the target slot also exchanges information across samples in the batch.

After the transformer backbone, the head reads *only* the final-layer embedding of the target token (the first position) and feeds it through the optional `hidden_units` MLP and the output layer. This is what the paper describes in Figure 1: "We take the contextual embeddings from SAINT and pass only the embedding correspond to the CLS token through an MLP to obtain the final prediction."

With `target_token = FALSE`, no target token is added and the head instead consumes the concatenation of all `n` feature tokens. That option is provided because the SAINT reference Python implementation (https://github.com/somepago/saint) departs from the paper and uses flatten-pooling; it is kept available for compatibility with that code path and for users who want the original brulee behavior.

Row Attention at Prediction Time:

Row attention computations adjust the internal embeddings based on the rows that are available at any given time. During training, the other rows in the batch are used to compute attention. After training, when `predict()` is called, the default behavior is to keep row attention on, mirroring the training-time computation. Because row attention is computed across the samples present in a given call, predictions for a row depend on what other rows are passed alongside it. To get batch-independent predictions (where the prediction for a given row is the same regardless of what other rows are in the input), set `row_attention_on_predict` to `FALSE`; row attention is then bypassed at predict time and column attention is used on its own.

Learning Rates:

The learning rate can be set to constant (the default) or dynamically set via a learning rate scheduler (via the `rate_schedule`). Using `rate_schedule = 'none'` uses the `learn_rate` argument.

Other Notes:

Unlike other **brulee** models, `brulee_saint()` natively handles factor predictors via learned embeddings. Factor columns are automatically detected and embedded, while numeric columns pass through per-feature MLPs. There is *no need to pre-encode factors as indicators*.

When the outcome is a number, the function internally standardizes the outcome data to have mean zero and a standard deviation of one. The prediction function creates predictions on the original scale.

By default, training halts when the validation loss increases for at least `stop_iter` iterations. If `validation = 0` the training set loss is used.

The model objects are saved for each epoch so that the number of epochs can be efficiently tuned. The `predict()` method for this model has an `epoch` argument (which defaults to the epoch with the best loss value).

Value

A `brulee_saint` object with elements:

- `models_obj`: a serialized raw vector for the torch module.
- `estimates`: a list of matrices with the model parameter estimates per epoch. The first element is epoch zero (the randomly initialized parameters before training), so the list has `epochs + 1` elements.
- `best_epoch`: an integer for the epoch with the smallest loss. Since `estimates` and `loss` include epoch zero, this epoch's values are at position `best_epoch + 1` in those objects.
- `loss`: A vector of loss values (MSE for regression, negative log- likelihood for classification) at each epoch, starting with epoch zero.
- `dim`: A list of data dimensions and feature metadata.
- `y_stats`: A list of summary statistics for numeric outcomes.
- `parameters`: A list of some tuning parameter values.
- `device`: A character string for the device used during training.
- `blueprint`: The `hardhat` blueprint data.

References

Somepalli, G., Goldblum, M., Schwarzschild, A., Bruss, C. B., & Goldstein, T. (2021). SAINT: Improved Neural Networks for Tabular Data via Row Attention and Contrastive Pre-Training. arXiv preprint arXiv:2106.01342.

See Also

`predict.brulee_saint()`, `autoplot.brulee_saint()`

Examples

```
pkgs <- c("recipes", "yardstick", "modeldata")
if (torch::torch_is_installed() && rlang::is_installed(pkgs)) {

 set.seed(87261)
 tr_data <- modeldata::sim_regression(500, method = "worley_1987")
 te_data <- modeldata::sim_regression(50, method = "worley_1987")

 library(recipes)
 rec <- recipe(outcome ~ ., data = te_data) |>
  step_normalize(all_numeric_predictors())

 set.seed(389)
 fit <- brulee_saint(
  rec,
  data = te_data,
  hidden_unit = 5,
  dropout_hidden = 0.2,
  num_embedding = 3,
  num_attn_heads = 5,
  num_attn_blocks = 4,
  dropout_attn = 0.2,
  epochs = 50L,
  batch_size = 32L,
  learn_rate = 0.01,
  optimize = "SGD",
  verbose = TRUE
 )

 autoplot(fit)
 summary(fit)

 library(yardstick)
 predict(fit, te_data) |>
  dplyr::bind_cols(te_data) |>
  rsq(outcome, .pred)

}
```

`brulee_tab_icl` *Fit a TabICL tabular foundation model*

Description

`brulee_tab_icl()` prepares the pre-trained TabICL (Tabular In-Context Learning) foundation model from Qu *et al* (2025) for prediction. TabICL is a transformer that makes predictions on tabular data by *in-context learning*: it is not trained on your data. Instead, the released pre-trained

weights are loaded and the model conditions on your training rows at prediction time, much like a few-shot language model conditions on its prompt. Both classification and regression are supported.

Usage

```
brulee_tab_icl(x, ...)

## Default S3 method:
brulee_tab_icl(x, ...)

## S3 method for class 'data.frame'
brulee_tab_icl(
  x,
  y,
  num_estimators = 8L,
  normalization = c("none", "YeoJohnson"),
  softmax_temperature = 0.9,
  training_set_limit = Inf,
  device = NULL,
  ...
)

## S3 method for class 'matrix'
brulee_tab_icl(
  x,
  y,
  num_estimators = 8L,
  normalization = c("none", "YeoJohnson"),
  softmax_temperature = 0.9,
  training_set_limit = Inf,
  device = NULL,
  ...
)

## S3 method for class 'formula'
brulee_tab_icl(
  formula,
  data,
  num_estimators = 8L,
  normalization = c("none", "YeoJohnson"),
  softmax_temperature = 0.9,
  training_set_limit = Inf,
  device = NULL,
  ...
)

## S3 method for class 'recipe'
brulee_tab_icl(
  x,
```

```
  data,
  num_estimators = 8L,
  normalization = c("none", "YeoJohnson"),
  softmax_temperature = 0.9,
  training_set_limit = Inf,
  device = NULL,
  ...
)
```

Arguments

x	Depending on the context: • A **data frame** of predictors. • A **matrix** of predictors. • A **recipe** specifying a set of preprocessing steps created from `recipes::recipe()`. The predictor data should be standardized (e.g. centered or scaled).
...	Not currently used.
y	When `x` is a **data frame** or **matrix**, `y` is the outcome specified as: • A **data frame** with 1 column (numeric or factor). • A **matrix** with numeric column (numeric or factor). • A **vector** (numeric or factor).
num_estimators	An integer for the number of ensemble members (default `8`). Each member preprocesses, permutes features, and (for classification) shuffles class labels differently; their predictions are averaged. Use `1` for a single, fully deterministic member.
normalization	A character vector of per-member normalization methods. Currently `"none"` (standardize only) and `"YeoJohnson"` (Yeo-Johnson power transform on top of standardization) are supported.
softmax_temperature	A number for the temperature applied to the classification softmax. Only used for classification.
training_set_limit	A single number giving the maximum number of training rows kept as in-context examples. When the training data has more rows than this, a subsample of exactly `training_set_limit` rows is drawn (stratified by the outcome for classification, simple random for regression). The default is `Inf`, which keeps every row. Useful for capping memory and prediction time on large training sets, since the entire (kept) training set is stored on the fitted object and re-sent through the network on every call to `predict()`.
device	A character string for the compute device: `"cpu"` (the default) or `"cuda"`. See the **Device support** section.
formula	A formula specifying the outcome term(s) on the left-hand side, and the predictor term(s) on the right-hand side.
data	When a **recipe** or **formula** is used, `data` is specified as: • A **data frame** containing both the predictors and the outcome.

Details

In-context learning:

Unlike the other **brulee** models, `brulee_tab_icl()` does not train any parameters. The pre-trained network is fixed; "fitting" simply validates and stores the (encoded) training predictors and outcomes together with a reference to the checkpoint. At `predict()` time, the model is given the training rows as labelled context alongside the new rows and produces predictions in a single forward pass. Because the training data are stored on the fitted object, larger training sets make the object larger and prediction slower.

Architecture:

TabICL processes a table through three transformer stages:

1. **Column embedding**: a per-column set transformer turns each cell into a distribution-aware embedding, optionally informed by the target.
2. **Row interaction**: a transformer with rotary position encoding mixes the feature embeddings within each row and aggregates them with learnable CLS tokens.
3. **In-context learning**: a dataset-level transformer lets the test rows attend to the labelled training rows to produce class logits (classification) or quantiles (regression).

Preprocessing:

TabICL applies its own preprocessing to mirror the reference implementation, so most data shaping that other tabular models require is unnecessary (and in some cases counter-productive). The pipeline runs in two stages.

Stage 1: numeric encoding (always, at fit time).

Each predictor column is converted to a numeric value:

- Factor and character columns are **ordinal-encoded**: the unique values seen during fitting are sorted lexicographically and mapped to 0-based integers. *Do not pre-encode factors as indicator (dummy) variables.* TabICL is a per-column tokenized transformer; one ordinal column gives the model one token per row, while a wide one-hot expansion bloats the sequence length, blows up the row-interaction stage, and degrades prediction quality.
- Numeric columns are taken as-is.

The training predictors are stored on the fitted object in this encoded form so that they can serve as context at `predict()` time.

Stage 2: per-member normalization (at predict time).

For each ensemble member, the encoded predictors pass through a small pipeline before being handed to the network:

1. **Standardization** — center by column mean and divide by the population standard deviation (with a small epsilon and a soft clip to ± 100). This always runs.
2. **Optional Yeo-Johnson** — when the member's `normalization` slot is `"YeoJohnson"`, a per-column Yeo-Johnson power transform is inserted between standardization and outlier clipping. The Yeo-Johnson λ for each column is fit on the standardized training data via maximum likelihood, then the transformed values are re-standardized so the downstream stages see the same mean/scale as the `"none"` path. The transform is helpful when individual columns are heavily skewed or heavy-tailed. The `normalization` argument is a vector because the default ensemble intentionally mixes `"none"` and `"YeoJohnson"` across members to boost predictive diversity.

3. **Outlier clipping** — a two-stage z-score clipper trims extreme values. This always runs.

All parameters in stage 2 (means, standard deviations, Yeo-Johnson lambdas, clip bounds) are estimated on the training rows alone and then applied to both training and new rows.

For regression, the outcome is standardized internally and predictions are returned on the original scale. For classification, the outcome is label-encoded.

Missing Values:

Missing values do not need to be imputed by the user.

- **Numeric columns**: at fit time the column mean (ignoring `NA`) is learned and reused to fill any `NA` in both the training context and the prediction rows.
- **Factor and character columns**: missing values, as well as any *new* factor levels seen at prediction time that were not present during fitting, are mapped to the sentinel code `-1` and treated as a distinct "unknown" category by the model.

Pre-imputation by the user is still allowed and is sometimes desirable (for example, when a domain-appropriate imputation outperforms a column mean), but it is not required for the model to run.

Ensembling:

With `num_estimators > 1`, several views of the data are created by permuting features and (for classification) shuffling class labels, each optionally with a different normalization. Class logits are averaged across members and converted to probabilities with a temperature softmax; regression means are averaged. `num_estimators = 1` uses a single deterministic member (no shuffles, `"none"` normalization). Note that with more than one member the feature permutations are drawn with R's random number generator, so results are a faithful reproduction of the reference ensemble but not bit-for-bit identical to it; set the seed for reproducibility across runs.

Device support:

Computation runs on CPU by default and on CUDA when `device = "cuda"` and a GPU is available. The Apple `"mps"` backend is **not** supported: the bundled libtorch MPS kernels crash on parts of the model, so requesting `"mps"` issues a warning and falls back to CPU.

Pre-trained weights:

The estimated parameters from the pre-trained Python model are used. On first use, the values are downloaded and cached locally and are more than 200MB.

Value

A `brulee_tab_icl` object with elements:

- `path`: the cached checkpoint directory the weights are loaded from.
- `config`: the parsed model configuration.
- `task`: `"classification"` or `"regression"`.
- `levels`: the outcome factor levels (classification only).
- `encoders`: the fitted per-column predictor encoders.
- `train_x`, `train_y`: the encoded training context.
- `num_estimators`, `normalization`, `softmax_temperature`: ensemble settings.
- `device`: the resolved compute device.
- `blueprint`: the `hardhat` blueprint.

References

Qu, J., Holzmüller, D., Varoquaux, G., & Le Morvan, M. (2025). TabICL: A Tabular Foundation Model for In-Context Learning on Large Data. arXiv preprint arXiv:2502.05564.

See Also

`predict.brulee_tab_icl()`

Examples

```
## Not run:
# Requires converted TabICL weights cached under ~/.cache/TabICL/ (see the
# "Pre-trained weights" section and dev/tabicl/).

if (torch::torch_is_installed() && rlang::is_installed("modeldata")) {
  data(penguins, package = "modeldata")
  penguins <- na.omit(penguins)

  in_train <- sample(seq_len(nrow(penguins)), 250)
  tr <- penguins[in_train, ]
  te <- penguins[-in_train, ]

  # Classification (uses the cached classification checkpoint)
  cls_fit <- brulee_tab_icl(species ~ ., data = tr)
  predict(cls_fit, te)
  predict(cls_fit, te, type = "prob")

  # Regression (uses the cached regression checkpoint)
  reg_fit <- brulee_tab_icl(body_mass_g ~ ., data = tr)
  predict(reg_fit, te)
}

## End(Not run)
```

---

matrix_to_dataset *Convert data to torch format*

Description

For an x/y interface, `matrix_to_dataset()` converts the data to proper encodings then formats the results for consumption by `torch`.

Usage

```
matrix_to_dataset(x, y, device = NULL)
```

Arguments

x	A numeric matrix of predictors.
y	A vector. If regression than y is numeric. For classification, it is a factor.
device	A single character string for the device to use (e.g., "cpu" or "cuda"). The default of NULL uses the CPU. See training_efficiency.

Details

Missing values should be removed before passing data to this function.

Value

An R6 index sampler object with classes "training_set", "dataset", and "R6".

Examples

```
if (torch::torch_is_installed()) {
  matrix_to_dataset(as.matrix(mtcars[, -1]), mtcars$mpg)
}
```

---

predict.brulee_auto_int

Predict from a brulee_auto_int

Description

Predict from a brulee_auto_int

Usage

```
## S3 method for class 'brulee_auto_int'
predict(object, new_data, type = NULL, epoch = NULL, ...)
```

Arguments

object	A brulee_auto_int object.
new_data	A data frame or matrix of new predictors.
type	A single character. The type of predictions to generate. Valid options are: • "numeric" for numeric predictions. • "class" for hard class predictions • "prob" for soft class predictions (i.e., class probabilities)
epoch	An integer for the epoch to make predictions. If this value is larger than the maximum number that was fit, a warning is issued and the parameters from the last epoch are used. If left NULL, the epoch associated with the smallest loss is used.
...	Not used, but required for extensibility.

Value

A tibble of predictions. The number of rows in the tibble is guaranteed to be the same as the number of rows in new_data.

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "modeldata"))) {
  set.seed(87261)
  tr_data <- modeldata::sim_classification(500)
  te_data <- modeldata::sim_classification(50)

  set.seed(2)
  fit <- brulee_auto_int(class ~ ., data = tr_data,
                         epochs = 50L, batch_size = 64L, stop_iter = 10L,
                         hidden_units = 5, hidden_activations = "relu",
                         learn_rate = 0.01, penalty = 0.01)
  fit

  autoplot(fit)

 predict(fit, te_data)
 predict(fit, te_data, type = "prob")
}
```

---

predict.brulee_chronos

Predict from a brulee_chronos *model*

Description

Predict from a brulee_chronos model

Usage

```
## S3 method for class 'brulee_chronos'
predict(
  object,
  new_data = NULL,
  type = "all",
  prediction_length = NULL,
  quantile_levels = NULL,
  ...
)
```

Arguments

object	A brulee_chronos object returned by brulee_chronos().
new_data	Optional data frame describing the future window to forecast for. It should contain the id and timestamp columns (when those were supplied at construction) plus any known future covariate values (a subset of the past covariates). The number of rows per series is the number of future time steps to return and may be at most prediction_length; supplying more is an error. When a series has fewer rows than prediction_length, the missing future covariates are treated as unknown and the forecast is truncated to the rows provided. If NULL (the default), the full prediction_length horizon is forecast from the context stored in object. The model is pretrained, so the historical context is always the data passed to brulee_chronos() and is never overridden here.
type	A single string for the type of prediction to return. The default "all" returns both the point forecast (.pred) and the quantile forecast (.pred_quantile). Use "numeric" for only .pred or "quantile" for only .pred_quantile.
prediction_length	Number of future time steps to forecast. Defaults to the value stored in object.
quantile_levels	Numeric vector of quantile levels. Defaults to the value stored in object.
...	Not used.

Value

A tibble with one row per forecast time step per series (up to nrow(new_data) rows per series, or prediction_length rows when new_data is NULL). Columns depend on type:

<id_column> The time series identifier. Omitted when the context contains a single series.

.pred Point forecast, i.e. the median of .pred_quantile. Returned when type is "all" or "numeric".

.pred_quantile A hardhat::quantile_pred() vector packing all requested quantile levels into a single column. Returned when type is "all" or "quantile".

Examples

```
pkgs <- c("recipes", "lubridate", "modeldata", "ggplot2")

## Not run:
if (torch::torch_is_installed() && rlang::is_installed(pkgs)) {
 library(dplyr)
 library(ggplot2)

 n <- nrow(modeldata::Chicago)

 prior_data <- modeldata::Chicago[-((n-13):n),]
 test_data <-
  modeldata::Chicago[(n-13):n,] |>
  mutate(day = lubridate::wday(date, label = TRUE))
```

```
# ------------------------------------------------------------------------------
# Simple, no covariate model

mod_1 <-
 brulee_chronos(
  ridership ~ 1,
  data = prior_data,
  # Removing `timestamp_column` does not affect the fit
  timestamp_column = c(date),
  prediction_length = 14)

pred_1 <- predict(mod_1, new_data = test_data)
pred_1

pred_1 |>
 bind_cols(test_data) |>
 ggplot(aes(date)) +
 geom_point(aes(y = ridership, col = day)) +
 geom_line(aes(y = .pred)) +
 labs(title = "No covariates: Meh") +
 theme_bw()

# ------------------------------------------------------------------------------
# Some covariates via the formula method

mod_2 <-
  brulee_chronos(
   ridership ~ Clark_Lake + Belmont + Harlem + Monroe,
   data = prior_data,
   timestamp_column = c(date),
   prediction_length = 14)

pred_2 <- predict(mod_2, new_data = test_data)

pred_2 |>
 bind_cols(test_data) |>
 ggplot(aes(date)) +
 geom_point(aes(y = ridership, col = day)) +
 geom_line(aes(y = .pred)) +
 labs(title = "Four covariates: Pretty good") +
 theme_bw()

# ------------------------------------------------------------------------------
# Covariates using recipes

rec <-
 recipe(ridership ~ ., data = prior_data) |>
 update_role(date, new_role = "time")

mod_3 <- brulee_chronos(rec, data = prior_data, prediction_length = 14)

pred_3 <- predict(mod_3, new_data = test_data)
```

```
 pred_3 |>
  bind_cols(test_data) |>
  ggplot(aes(date)) +
  geom_point(aes(y = ridership, col = day)) +
  geom_line(aes(y = .pred)) +
  labs(title = "All covariates: Better Saturdays") +
  theme_bw()
}

## End(Not run)
```

`predict.brulee_linear_reg` *Predict from a* `brulee_linear_reg`

Description

Predict from a `brulee_linear_reg`

Usage

```
## S3 method for class 'brulee_linear_reg'
predict(object, new_data, type = NULL, epoch = NULL, ...)
```

Arguments

`object`	A `brulee_linear_reg` object.
`new_data`	A data frame or matrix of new predictors.
`type`	A single character. The type of predictions to generate. Valid options are: • `"numeric"` for numeric predictions.
`epoch`	An integer for the epoch to make predictions. If this value is larger than the maximum number that was fit, a warning is issued and the parameters from the last epoch are used. If left `NULL`, the epoch associated with the smallest loss is used.
`...`	Not used, but required for extensibility.

Value

A tibble of predictions. The number of rows in the tibble is guaranteed to be the same as the number of rows in `new_data`.

Examples

```
if (torch::torch_is_installed() && rlang::is_installed("recipes")) {

 data(ames, package = "modeldata")

 ames$Sale_Price <- log10(ames$Sale_Price)

 set.seed(1)
 in_train <- sample(seq_len(nrow(ames)), 2000)
 ames_train <- ames[ in_train,]
 ames_test  <- ames[-in_train,]

 # Using recipe
 library(recipes)

 ames_rec <-
  recipe(Sale_Price ~ Longitude + Latitude, data = ames_train) |>
    step_normalize(all_numeric_predictors())

 set.seed(2)
 fit <- brulee_linear_reg(ames_rec, data = ames_train, epochs = 50)

 predict(fit, ames_test)
}
```

---

predict.brulee_logistic_reg

Predict from a `brulee_logistic_reg`

Description

Predict from a `brulee_logistic_reg`

Usage

```
## S3 method for class 'brulee_logistic_reg'
predict(object, new_data, type = NULL, epoch = NULL, ...)
```

Arguments

`object`	A `brulee_logistic_reg` object.
`new_data`	A data frame or matrix of new predictors.
`type`	A single character. The type of predictions to generate. Valid options are: • `"class"` for hard class predictions • `"prob"` for soft class predictions (i.e., class probabilities)

epoch — An integer for the epoch to make predictions. If this value is larger than the maximum number that was fit, a warning is issued and the parameters from the last epoch are used. If left NULL, the epoch associated with the smallest loss is used.

... — Not used, but required for extensibility.

Value

A tibble of predictions. The number of rows in the tibble is guaranteed to be the same as the number of rows in new_data.

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "yardstick", "modeldata"))) {

  library(recipes)
  library(yardstick)

  data(penguins, package = "modeldata")

  penguins <- penguins |> na.omit()

  set.seed(122)
  in_train <- sample(seq_len(nrow(penguins)), 200)
  penguins_train <- penguins[ in_train,]
  penguins_test  <- penguins[-in_train,]

  rec <- recipe(sex ~ ., data = penguins_train) |>
    step_dummy(all_nominal_predictors()) |>
    step_normalize(all_numeric_predictors())

  set.seed(3)
  fit <- brulee_logistic_reg(rec, data = penguins_train, epochs = 5)
  fit

  predict(fit, penguins_test)

  predict(fit, penguins_test, type = "prob") |>
    bind_cols(penguins_test) |>
    roc_curve(sex, .pred_female) |>
    autoplot()

}
```

predict.brulee_mlp *Predict from a* brulee_mlp

Description

Predict from a brulee_mlp

Usage

```
## S3 method for class 'brulee_mlp'
predict(object, new_data, type = NULL, epoch = NULL, ...)
```

Arguments

object	A brulee_mlp object.
new_data	A data frame or matrix of new predictors.
type	A single character. The type of predictions to generate. Valid options are: • "numeric" for numeric predictions. • "class" for hard class predictions • "prob" for soft class predictions (i.e., class probabilities)
epoch	An integer for the epoch to make predictions. If this value is larger than the maximum number that was fit, a warning is issued and the parameters from the last epoch are used. If left NULL, the epoch associated with the smallest loss is used.
...	Not used, but required for extensibility.

Value

A tibble of predictions. The number of rows in the tibble is guaranteed to be the same as the number of rows in new_data.

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "modeldata"))) {
 # regression example:

 data(ames, package = "modeldata")

 ames$Sale_Price <- log10(ames$Sale_Price)

 set.seed(1)
 in_train <- sample(seq_len(nrow(ames)), 2000)
 ames_train <- ames[ in_train,]
 ames_test  <- ames[-in_train,]

 # Using recipe
```

```
library(recipes)

ames_rec <-
 recipe(Sale_Price ~ Longitude + Latitude, data = ames_train) |>
   step_normalize(all_numeric_predictors())

set.seed(2)
fit <- brulee_mlp(ames_rec, data = ames_train, epochs = 50, batch_size = 32)

predict(fit, ames_test)
}
```

`predict.brulee_multinomial_reg` *Predict from a* `brulee_multinomial_reg`

Description

Predict from a `brulee_multinomial_reg`

Usage

```
## S3 method for class 'brulee_multinomial_reg'
predict(object, new_data, type = NULL, epoch = NULL, ...)
```

Arguments

`object`	A `brulee_multinomial_reg` object.
`new_data`	A data frame or matrix of new predictors.
`type`	A single character. The type of predictions to generate. Valid options are: • `"class"` for hard class predictions • `"prob"` for soft class predictions (i.e., class probabilities)
`epoch`	An integer for the epoch to make predictions. If this value is larger than the maximum number that was fit, a warning is issued and the parameters from the last epoch are used. If left `NULL`, the epoch associated with the smallest loss is used.
`...`	Not used, but required for extensibility.

Value

A tibble of predictions. The number of rows in the tibble is guaranteed to be the same as the number of rows in `new_data`.

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "yardstick", "modeldata"))) {

  library(recipes)
  library(yardstick)

  data(penguins, package = "modeldata")

  penguins <- penguins |> na.omit()

  set.seed(122)
  in_train <- sample(seq_len(nrow(penguins)), 200)
  penguins_train <- penguins[ in_train,]
  penguins_test  <- penguins[-in_train,]

  rec <- recipe(island ~ ., data = penguins_train) |>
    step_dummy(species, sex) |>
    step_normalize(all_numeric_predictors())

  set.seed(3)
  fit <- brulee_multinomial_reg(rec, data = penguins_train, epochs = 5)
  fit

  predict(fit, penguins_test) |>
    bind_cols(penguins_test) |>
    conf_mat(island, .pred_class)
}
```

---

predict.brulee_resnet *Predict from a* brulee_resnet

Description

Predict from a brulee_resnet

Usage

```
## S3 method for class 'brulee_resnet'
predict(object, new_data, type = NULL, epoch = NULL, ...)
```

Arguments

object	A brulee_resnet object.
new_data	A data frame or matrix of new predictors.
type	A single character. The type of predictions to generate. Valid options are:

- "numeric" for numeric predictions.
- "class" for hard class predictions
- "prob" for soft class predictions (i.e., class probabilities)

epoch An integer for the epoch to make predictions. If this value is larger than the maximum number that was fit, a warning is issued and the parameters from the last epoch are used. If left NULL, the epoch associated with the smallest loss is used.

... Not used, but required for extensibility.

Value

A tibble of predictions. The number of rows in the tibble is guaranteed to be the same as the number of rows in new_data.

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "modeldata"))) {
 # regression example:

 data(ames, package = "modeldata")

 ames$Sale_Price <- log10(ames$Sale_Price)

 set.seed(1)
 in_train <- sample(seq_len(nrow(ames)), 2000)
 ames_train <- ames[ in_train,]
 ames_test  <- ames[-in_train,]

 # Using recipe
 library(recipes)

 ames_rec <-
  recipe(Sale_Price ~ Longitude + Latitude, data = ames_train) |>
    step_normalize(all_numeric_predictors())

 set.seed(2)
 fit <- brulee_resnet(ames_rec, data = ames_train,
                      hidden_units = 2, num_layers = 2, bottleneck_units = 10,
                      epochs = 50, batch_size = 32)

 predict(fit, ames_test)
}
```

predict.brulee_rln *Predict from a* brulee_rln

Description

Predict from a brulee_rln

Usage

```
## S3 method for class 'brulee_rln'
predict(object, new_data, type = NULL, epoch = NULL, ...)
```

Arguments

object	A brulee_rln object.
new_data	A data frame or matrix of new predictors.
type	A single character. The only valid option is "numeric" for numeric predictions.
epoch	An integer for the epoch to make predictions. If larger than the number of epochs fit, a warning is issued and the last epoch is used. If NULL (default), the epoch with the smallest loss is used.
...	Not used, but required for extensibility.

Value

A tibble of predictions with the same number of rows as new_data.

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "modeldata"))) {

 data(ames, package = "modeldata")
 ames$Sale_Price <- log10(ames$Sale_Price)

 set.seed(1)
 in_train <- sample(seq_len(nrow(ames)), 2000)
 ames_train <- ames[ in_train,]
 ames_test  <- ames[-in_train,]

 library(recipes)

 ames_rec <-
  recipe(Sale_Price ~ Longitude + Latitude, data = ames_train) |>
    step_normalize(all_numeric_predictors())

 set.seed(2)
 fit <- brulee_rln(ames_rec, data = ames_train, hidden_units = 20L, epochs = 30L)
```

```
 predict(fit, ames_test)
}
```

`predict.brulee_saint` *Predict from a* `brulee_saint`

Description

Predict from a `brulee_saint`

Usage

```
## S3 method for class 'brulee_saint'
predict(object, new_data, type = NULL, epoch = NULL, ...)
```

Arguments

`object`	A `brulee_saint` object.
`new_data`	A data frame or matrix of new predictors.
`type`	A single character. The type of predictions to generate. Valid options are: • `"numeric"` for numeric predictions. • `"class"` for hard class predictions • `"prob"` for soft class predictions (i.e., class probabilities)
`epoch`	An integer for the epoch to make predictions. If this value is larger than the maximum number that was fit, a warning is issued and the parameters from the last epoch are used. If left `NULL`, the epoch associated with the smallest loss is used.
`...`	Not used, but required for extensibility.

Value

A tibble of predictions. The number of rows in the tibble is guaranteed to be the same as the number of rows in `new_data`.

Examples

```
if (torch::torch_is_installed() && rlang::is_installed(c("recipes", "modeldata"))) {
  set.seed(87261)
  tr_data <- modeldata::sim_classification(500)
  te_data <- modeldata::sim_classification(50)

  set.seed(2)
  fit <- brulee_saint(class ~ ., data = tr_data,
                      epochs = 50L, batch_size = 64L, stop_iter = 10L,
```

```
                      learn_rate = 0.001)
  fit

  autoplot(fit)

 predict(fit, te_data)
 predict(fit, te_data, type = "prob")
}
```

`predict.brulee_tab_icl` *Predict from a* `brulee_tab_icl`

Description

Predict from a `brulee_tab_icl`

Usage

```
## S3 method for class 'brulee_tab_icl'
predict(object, new_data, type = NULL, ...)
```

Arguments

`object`	A `brulee_tab_icl` object from `brulee_tab_icl()`.
`new_data`	A data frame or matrix of new predictors.
`type`	A single character string for the type of prediction. Valid options are: • `"class"` for hard class predictions (classification). • `"prob"` for class probabilities (classification). • `"numeric"` for numeric predictions (regression). If `NULL` (the default), the natural type for the outcome is used: `"class"` for a factor outcome and `"numeric"` for a numeric one.
`...`	Not used, but required for extensibility.

Details

Because TabICL is an in-context learner, prediction reloads the pretrained weights from the checkpoint directory stored on `object` and conditions on the training rows captured at fit time. The same preprocessing and ensembling used for `object` are applied to `new_data`; see `brulee_tab_icl()` for details. For classification, `"prob"` returns one column per class (named `.pred_<level>`) and `"class"` returns the highest-probability class.

Value

A tibble of predictions. The number of rows is guaranteed to match `new_data`. For `type = "prob"` there is one column per outcome class; for `"class"` and `"numeric"` there is a single prediction column.

See Also

`brulee_tab_icl()`

Examples

```
## Not run:
if (torch::torch_is_installed() && rlang::is_installed("modeldata")) {
  data(penguins, package = "modeldata")
  penguins <- na.omit(penguins)

  fit <- brulee_tab_icl(
    species ~ .,
    data = penguins,
    path = "path/to/tabicl-classifier"
  )
  predict(fit, penguins)
  predict(fit, penguins, type = "prob")
}

## End(Not run)
```

---

schedule_decay_time *Change the learning rate over time*

Description

Learning rate schedulers alter the learning rate to adjust as training proceeds. In most cases, the learning rate decreases as epochs increase. The `schedule_*()` functions are individual schedulers and `set_learn_rate()` is a general interface.

Usage

```
schedule_decay_time(epoch, initial = 0.1, decay = 1)

schedule_decay_expo(epoch, initial = 0.1, decay = 1)

schedule_step(epoch, initial = 0.1, reduction = 1/2, steps = 5)

schedule_cyclic(epoch, initial = 0.001, largest = 0.1, step_size = 5)

set_learn_rate(epoch, learn_rate, type = "none", ...)
```

Arguments

epoch	An integer for the number of training epochs (zero being the initial value),
initial	A positive numeric value for the starting learning rate.
decay	A positive numeric constant for decreasing the rate (see Details below).
reduction	A positive numeric constant stating the proportional decrease in the learning rate occurring at every steps epochs.
steps	The number of epochs before the learning rate changes.
largest	The maximum learning rate in the cycle.
step_size	The half-length of a cycle.
learn_rate	A constant learning rate (when no scheduler is used),
type	A single character value for the type of scheduler. Possible values are: "decay_time", "decay_expo", "none", "cyclic", and "step".
...	Arguments to pass to the individual scheduler functions (e.g. reduction).

Details

The details for how the schedulers change the rates:

- schedule_decay_time(): $rate(epoch) = initial/(1 + decay \times epoch)$
- schedule_decay_expo(): $rate(epoch) = initial \exp(-decay \times epoch)$
- schedule_step(): $rate(epoch) = initial \times reduction^{floor(epoch/steps)}$
- schedule_cyclic(): $cycle = floor(1+(epoch/2/stepsize))$, $x = abs((epoch/stepsize) - (2 * cycle) + 1)$, and $rate(epoch) = initial + (largest - initial) * \max(0, 1 - x)$

Value

A numeric value for the updated learning rate.

See Also

brulee_mlp()

Examples

```
if (rlang::is_installed("purrr")) {
 library(ggplot2)
 library(dplyr)
 library(purrr)

 iters <- 0:50

 bind_rows(
  tibble(epoch = iters, rate = map_dbl(iters, schedule_decay_time), type = "decay_time"),
  tibble(epoch = iters, rate = map_dbl(iters, schedule_decay_expo), type = "decay_expo"),
  tibble(epoch = iters, rate = map_dbl(iters, schedule_step), type = "step"),
  tibble(epoch = iters, rate = map_dbl(iters, schedule_cyclic), type = "cyclic")
```

```
 ) |>
  ggplot(aes(epoch, rate)) +
  geom_line() +
  facet_wrap(~ type)

}
```

---

summary.brulee_mlp *Summarize the architecture of a brulee model*

Description

summary() methods **brulee** neural network models print a layer-by-layer description of the fitted torch module: each component's type, shape, and parameter count, followed by the total parameter count. For brulee_resnet, residual (skip) connections and their projection layers are shown at the block boundaries where they apply.

Usage

```
## S3 method for class 'brulee_mlp'
summary(object, ...)

## S3 method for class 'brulee_resnet'
summary(object, ...)

## S3 method for class 'brulee_rln'
summary(object, ...)

## S3 method for class 'brulee_auto_int'
summary(object, ...)

## S3 method for class 'brulee_saint'
summary(object, ...)
```

Arguments

object	A brulee_resnet, brulee_mlp, brulee_rln, brulee_auto_int, or brulee_saint object.
...	Not used.

Value

The model object, invisibly. Called for its side effect of printing the architecture.

Examples

```
if (torch::torch_is_installed() && rlang::is_installed("modeldata")) {
  data(ames, package = "modeldata")
  ames$Sale_Price <- log10(ames$Sale_Price)

  set.seed(1)
  fit <- brulee_resnet(Sale_Price ~ Longitude + Latitude, data = ames,
                       hidden_units = c(8, 4), bottleneck_units = c(6, 3),
                       residual_at = 2, epochs = 3)
  summary(fit)
}
```

---

tab_icl_download_weights

Download and cache pretrained TabICL weights

Description

brulee_tab_icl() needs pretrained weights that are not shipped with the package. tab_icl_download_weights() fetches them from a release of the tidymodels/tabicl-weights GitHub repository into the local cache. tab_icl_weights_available() reports whether the cache already holds usable weights.

Usage

```
tab_icl_download_weights(
  task = c("classification", "regression"),
  version = tabicl_default_version(),
  date = tabicl_default_date(),
  repo = tabicl_default_repo(),
  cache_dir = tabicl_cache_dir(),
  call = rlang::caller_env()
)

tab_icl_weights_available(
  task = c("classification", "regression"),
  cache_dir = tabicl_cache_dir()
)
```

Arguments

task	The task(s) to act on, one or both of "classification" and "regression". Both are fetched (and checked) by default.
version, date	The release to fetch, identifying the tag <version>-<date> (for example "v2" and "2026-02-12").

repo	The owner/name of the GitHub repository hosting the weights.
cache_dir	The root of the local weight cache. Defaults to the brulee.tabicl_cache_dir option or ~/.cache/TabICL.
call	The calling environment, used for error messages.

Details

Each release carries the two files brulee reads per task (a JSON config and a safetensors weight file) as individual assets. They are downloaded into <cache_dir>/<version>/<date>/<TaskLabel>/. A file already present and complete is left in place, so re-running resumes rather than re-downloads.

The cache location can be overridden with the brulee.tabicl_cache_dir option. When brulee is attached and the weights are missing, the package offers to download them in interactive sessions and downloads them otherwise; set options(brulee.tabicl_autodownload = FALSE) to disable that behavior.

Value

tab_icl_download_weights() invisibly returns the populated <cache_dir>/<version>/<date> directory. tab_icl_weights_available() returns a single logical.

Examples

```
tab_icl_download_weights()
tab_icl_weights_available()
```

---

training_efficiency *Training Efficiency*

Description

There are ways to speed up or slow down model training. Here are some notes.

Details

GPUs can perform calculations very fast, sometimes faster than the overhead of a high-level interface such as brulee. GPU utilization might be lower than expected because the model is not very large (i.e., with millions of parameters) and/or because the batch size is small.

For the latter, here is an example of a training set with 1K samples, one single hidden layer with 50 units, 200 epochs, and used ADAMw optimizer:

batch_size	CPU elapsed	CUDA elapsed	(CPU/CUDA) speedup
128	90.09s	111.54s	0.81x
512	26.22s	28.61s	0.92x
2048	11.07s	8.31s	1.33x
8192	4.42s	3.57s	1.24x

As batch sizes become larger, the GPU has a better chance of reducing training time.

Some optimizers are faster than others. Although we use `torch::optim_adamw()` directly, it can be much slower than others. For one benchmark:

optimizer	CPU elapsed	CUDA elapsed
ADAMw	66.22s	84.42s
SGD	30.12s	30.83s

Index