Package 'kindling'

July 10, 2026

Type Package

Title Higher-Level Interface of 'torch' Package to Auto-Train Neural Networks

Version 0.3.2

Description Provides a higher-level interface to the 'torch' package for defining, training, and fine-tuning neural networks through code generation. The package supports several architectures, including feedforward (multi-layer perceptron) and recurrent neural networks (RNN, LSTM, GRU), while reducing boilerplate 'torch' code. Model training methods also bridge to machine learning frameworks in R, particularly the 'tidymodels' ecosystem, including 'parsnip' model specifications, workflows, recipes, and tuning tools.

License MIT + file LICENSE

Encoding UTF-8

Imports purrr, torch, rlang, cli, glue, vctrs, parsnip (>= 1.0.0), tibble, tidyr, dplyr, stats, NeuralNetTools, ggplot2, tune, dials, hardhat, lifecycle, coro

Suggests testthat (>= 3.0.0), magrittr, box, vip, recipes, workflows, rsample, yardstick, mlbench, modeldata, knitr, rmarkdown, DiceDesign, lhs, sfd, covr, fansi

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

Config/testthat/edition 3

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

URL https://kindling.joshuamarie.com, https://github.com/joshuamarie/kindling

BugReports https://github.com/joshuamarie/kindling/issues

VignetteBuilder knitr

NeedsCompilation no

Author Joshua Marie [aut, cre],
Antoine Soetewey [aut] (ORCID: <https://orcid.org/0000-0001-8159-0804>)

Maintainer Joshua Marie <joshua.marie.k@gmail.com>

Repository CRAN

Date/Publication 2026-07-10 05:50:13 UTC

Contents

act_funs . . . 2
args . . . 3
autoplot_diagnostics . . . 4
early_stop . . . 5
gen-nn-train . . . 6
grid_depth . . . 14
kindling-basemodels . . . 18
kindling-varimp . . . 21
layer_prs . . . 23
mlp_kindling . . . 25
new_act_fn . . . 27
nn_arch . . . 28
nn_gens . . . 30
nn_module_generator . . . 34
ordinal_gen . . . 40
rnn_kindling . . . 40
table_summary . . . 43
train_nnsnip . . . 45

Index **49**

`act_funs` *Activation Functions Specification Helper*

Description

This function is a DSL function, kind of like `ggplot2::aes()`, that helps to specify activation functions for neural network layers. It validates that activation functions exist in `torch` and that any parameters match the function's formal arguments.

Usage

```
act_funs(...)
```

Arguments

...	Activation function specifications. Can be: • Bare symbols: `relu`, `tanh` • Character strings (simple): `"relu"`, `"tanh"` • Character strings (with params): `"softshrink(lambda = 0.1)"`, `"rrelu(lower = 1/5, upper = 1/4)"` • Named with parameters: `softmax = args(dim = 2L)` • Indexed syntax (named): `softshrink[lambd = 0.2]`, `rrelu[lower = 1/5, upper = 1/4]` • Indexed syntax (unnamed): `softshrink[0.5]`, `elu[0.5]`

Value

A `vctrs` vector with class "activation_spec" containing validated activation specifications.

Examples

```
act_funs(relu, sigmoid)
act_funs(relu, softshrink[lambd = 0.5], elu)
act_funs(softmax = args(dim = 2L))
```

`args` *Activation Function Arguments Helper*

Description

[Superseded]

This is superseded in v0.3.0 in favour of indexed syntax, e.g. `<act_fn[param = 0]>` type. Type-safe helper to specify parameters for activation functions. All parameters must be named and match the formal arguments of the corresponding `{torch}` activation function.

Usage

```
args(...)
```

Arguments

...	Named arguments for the activation function.

Value

A list with class "activation_args" containing the parameters.

Examples

```
act_funs(softmax = args(dim = 2L))
```

`autoplot_diagnostics` *Plot prediction diagnostics for a fitted neural network*

Description

Produces diagnostic plots comparing fitted values against actual (true) response values for a fitted `nn_fit` model.

- **Regression (single output)**: returns a named list with two panels – residuals vs fitted and actual vs fitted.
- **Regression (multi-output)**: returns a named list with one actual vs fitted panel per output column.
- **Classification**: returns a single confusion matrix heatmap.

Usage

```
autoplot_diagnostics(object, actual, ...)

plot_diagnostics(object, actual, ...)
```

Arguments

`object`	A fitted model of class `nn_fit`, as returned by `train_nn()`, `ffnn()`, or `rnn()`.
`actual`	A vector of true response values, the same length as the training data used to fit `object`.
`...`	Additional arguments (currently unused).

Value

For regression, a named list of `ggplot2::ggplot()` objects (one per diagnostic panel). For classification, a single `ggplot2::ggplot()` confusion matrix heatmap.

Examples

```
if (torch::torch_is_installed()) {
  # Regression
  m = train_nn(
    as.matrix(iris[, 2:4]), iris$Sepal.Length,
    epochs = 5
  )
  autoplot_diagnostics(m, actual = iris$Sepal.Length)

  # Classification
  m_cls = train_nn(
    as.matrix(iris[, 1:4]), iris$Species,
    epochs = 5
  )
  autoplot_diagnostics(m_cls, actual = iris$Species)
```

```
}
```

`early_stop` *Early Stopping Specification*

Description

`early_stop()` is a helper function to be supplied on `early_stopping` arguments.

Usage

```
early_stop(
  patience = 5L,
  min_delta = 1e-04,
  restore_best_weights = TRUE,
  monitor = "val_loss"
)
```

Arguments

`patience`	Integer. Epochs to wait after last improvement. Default `5`.
`min_delta`	Numeric. Minimum improvement to qualify as better. Default `1e-4`.
`restore_best_weights`	Logical. Restore weights from best epoch. Default `TRUE`.
`monitor`	Character. Metric to monitor. One of `"val_loss"` (default) or `"train_loss"`.

Value

An object of class `"early_stop_spec"`.

Examples

```
early_stop(patience = 10L, min_delta = 1e-3, monitor = "train_loss")
```

gen-nn-train *Generalized Neural Network Trainer*

Description

[Experimental]

`train_nn()` is a generic function for training neural networks with a user-defined architecture via `nn_arch()`. Dispatch is based on the class of `x`.

Recommended workflow:

1. Define architecture with `nn_arch()` (optional).
2. Train with `train_nn()`.
3. Predict with `predict.nn_fit()`.

All methods delegate to a shared implementation core after preprocessing. When `architecture = NULL`, the model falls back to a plain feed-forward neural network (`nn_linear`) architecture.

Usage

```
train_nn(x, ...)

## S3 method for class 'matrix'
train_nn(
  x,
  y,
  hidden_neurons = NULL,
  activations = NULL,
  output_activation = NULL,
  bias = TRUE,
  arch = NULL,
  architecture = NULL,
  early_stopping = NULL,
  epochs = 100,
  batch_size = 32,
  penalty = 0,
  mixture = 0,
  learn_rate = 0.001,
  optimizer = "adam",
  optimizer_args = list(),
  loss = "mse",
  validation_split = 0,
  device = NULL,
  verbose = FALSE,
  cache_weights = FALSE,
  ...
)
```

```
## S3 method for class 'data.frame'
train_nn(
  x,
  y,
  hidden_neurons = NULL,
  activations = NULL,
  output_activation = NULL,
  bias = TRUE,
  arch = NULL,
  architecture = NULL,
  early_stopping = NULL,
  epochs = 100,
  batch_size = 32,
  penalty = 0,
  mixture = 0,
  learn_rate = 0.001,
  optimizer = "adam",
  optimizer_args = list(),
  loss = "mse",
  validation_split = 0,
  device = NULL,
  verbose = FALSE,
  cache_weights = FALSE,
  ...
)

## S3 method for class 'formula'
train_nn(
  x,
  data,
  hidden_neurons = NULL,
  activations = NULL,
  output_activation = NULL,
  bias = TRUE,
  arch = NULL,
  architecture = NULL,
  early_stopping = NULL,
  epochs = 100,
  batch_size = 32,
  penalty = 0,
  mixture = 0,
  learn_rate = 0.001,
  optimizer = "adam",
  optimizer_args = list(),
  loss = "mse",
  validation_split = 0,
  device = NULL,
```

```
  verbose = FALSE,
  cache_weights = FALSE,
  ...
)

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

## S3 method for class 'dataset'
train_nn(
  x,
  y = NULL,
  hidden_neurons = NULL,
  activations = NULL,
  output_activation = NULL,
  bias = TRUE,
  arch = NULL,
  architecture = NULL,
  flatten_input = NULL,
  epochs = 100,
  batch_size = 32,
  penalty = 0,
  mixture = 0,
  learn_rate = 0.001,
  optimizer = "adam",
  optimizer_args = list(),
  loss = "mse",
  validation_split = 0,
  device = NULL,
  verbose = FALSE,
  cache_weights = FALSE,
  n_classes = NULL,
  ...
)
```

Arguments

x	Dispatch is based on its current class: • `matrix`: used directly, no preprocessing applied. • `data.frame`: preprocessed via `hardhat::mold()`. `y` may be a vector / factor / matrix of outcomes, or a formula describing the outcome–predictor relationship within `x`. • `formula`: combined with `data` and preprocessed via `hardhat::mold()`. • `dataset`: a `torch` dataset object; batched via `torch::dataloader()`. This is the recommended interface for sequence/time-series and image data.
...	Additional arguments passed to specific methods.
y	Outcome data. Interpretation depends on the method:

- For the matrix and data.frame methods: a numeric vector, factor, or matrix of outcomes.
- For the data.frame method only: alternatively a formula of the form outcome ~ predictors, evaluated against x.
- Ignored when x is a formula (outcome is taken from the formula) or a dataset (labels come from the dataset itself).

hidden_neurons — Integer vector specifying the number of neurons in each hidden layer, e.g. c(128, 64) for two hidden layers. When NULL or missing, no hidden layers are used and the model reduces to a single linear mapping from inputs to outputs.

activations — Activation function specification(s) for the hidden layers. See act_funs() for supported values. Recycled if a single value is given.

output_activation — Optional activation function for the output layer. Defaults to NULL (no activation / linear output).

bias — Logical. Whether to include bias terms in each layer. Default TRUE.

arch — Backward-compatible alias for architecture. If both are supplied, they must be identical.

architecture — An nn_arch() object specifying a custom architecture. Default NULL, which falls back to a standard feed-forward network.

early_stopping — An early_stop() object specifying early stopping behaviour, or NULL (default) to disable early stopping. When supplied, training halts if the monitored metric does not improve by at least min_delta for patience consecutive epochs. Example: early_stopping = early_stop(patience = 10).

epochs — Positive integer. Number of full passes over the training data. Default 100.

batch_size — Positive integer. Number of samples per mini-batch. Default 32.

penalty — Non-negative numeric. L1/L2 regularization strength (lambda). Default 0 (no regularization).

mixture — Numeric in [0, 1]. Elastic net mixing parameter: 0 = pure ridge (L2), 1 = pure lasso (L1). Default 0.

learn_rate — Positive numeric. Step size for the optimizer. Default 0.001.

optimizer — Character. Optimizer algorithm. One of "adam" (default), "sgd", or "rmsprop".

optimizer_args — Named list of additional arguments forwarded to the optimizer constructor (e.g. list(momentum = 0.9) for SGD). Default list().

loss — Character or function. Loss function used during training. Built-in options: "mse" (default), "mae", "cross_entropy", or "bce". For classification tasks with a scalar label, "cross_entropy" is set automatically. Alternatively, supply a custom function or formula with signature function(input, target) returning a scalar torch_tensor.

validation_split — Numeric in [0, 1). Proportion of training data held out for validation. Default 0 (no validation set).

device — Character. Compute device: "cpu", "cuda", or "mps". Default NULL, which auto-detects the best available device.

verbose	Logical. If TRUE, prints loss (and validation loss) at regular intervals during training. Default FALSE.
cache_weights	Logical. If TRUE, stores a copy of the trained weight matrices in the returned object under $cached_weights. Default FALSE.
data	A data frame. Required when x is a formula.
flatten_input	Logical or NULL (dataset method only). Controls whether each batch/sample is flattened to 2D before entering the model. NULL (default) auto-selects: TRUE when architecture = NULL, otherwise FALSE.
n_classes	Positive integer. Number of output classes. Required when x is a dataset with scalar (classification) labels; ignored otherwise.

Details

The returned "nn_fit" object is a named list with the following components:

- model — the trained torch::nn_module object
- fitted — fitted values on the training data (or NULL for dataset fits)
- loss_history — numeric vector of per-epoch training loss, trimmed to actual epochs run (relevant when early stopping is active)
- val_loss_history — per-epoch validation loss, or NULL if validation_split = 0
- n_epochs — number of epochs actually trained
- stopped_epoch — epoch at which early stopping triggered, or NA if training ran to completion
- hidden_neurons, activations, output_activation — architecture spec
- penalty, mixture — regularization settings
- feature_names, response_name — variable names (tabular methods only)
- no_x, no_y — number of input features and output nodes
- is_classification — logical flag
- y_levels, n_classes — class labels and count (classification only)
- device — device the model is on
- cached_weights — list of weight matrices, or NULL
- arch — the nn_arch object used, or NULL

Value

An object of class "nn_fit", or one of its subclasses:

- c("nn_fit_tab", "nn_fit") — returned by the data.frame and formula methods
- c("nn_fit_ds", "nn_fit") — returned by the dataset method

All subclasses share a common structure. See **Details** for the list of components.

Supported tasks and input formats

`train_nn()` is task-agnostic by design (no explicit `task` argument). Task behavior is determined by your input interface and architecture:

- **Tabular data**: use `matrix`, `data.frame`, or `formula` methods.
- **Time series**: use the `dataset` method with per-item tensors shaped as `[time, features]` (or your preferred convention) and a recurrent architecture via `nn_arch()`.
- **Image classification**: use the `dataset` method with per-item tensors shaped for your first layer (commonly `[channels, height, width]` for `torch::nn_conv2d`). If your source arrays are channel-last, reorder in the dataset or via `input_transform`.

Matrix method

When `x` is supplied as a raw numeric matrix, no preprocessing is applied. Data is passed directly to the shared `train_nn_impl` core.

Data frame method

When `x` is a data frame, `y` can be either a vector / factor / matrix of outcomes, or a formula of the form `outcome ~ predictors` evaluated against `x`. Preprocessing is handled by `hardhat::mold()`.

Formula method

When `x` is a formula, `data` must be supplied as the data frame against which the formula is evaluated. Preprocessing is handled by `hardhat::mold()`.

Dataset method (`train_nn.dataset()`)

Trains a neural network directly on a `torch` dataset object. Batching and lazy loading are handled by `torch::dataloader()`, making this method well-suited for large datasets that do not fit entirely in memory.

Architecture configuration follows the same contract as other `train_nn()` methods via `architecture = nn_arch(...)` (or legacy `arch = ...`). For non-tabular inputs (time series, images), set `flatten_input = FALSE` to preserve tensor dimensions expected by recurrent or convolutional layers.

Labels are taken from the second element of each dataset item (i.e. `dataset[[i]][[2]]`), so `y` is ignored. When the label is a scalar tensor, a classification task is assumed and `n_classes` must be supplied. The loss is automatically switched to `"cross_entropy"` in that case.

Fitted values are **not** cached in the returned object. Use `predict.nn_fit_ds()` with `newdata` to obtain predictions after training.

See Also

`predict.nn_fit()`, `nn_arch()`, `act_funs()`, `early_stop()`

Examples

```
if (torch::torch_is_installed()) {
    # Matrix method — no preprocessing
    model = train_nn(
        x = as.matrix(iris[, 2:4]),
        y = iris$Sepal.Length,
        hidden_neurons = c(64, 32),
        activations = "relu",
        epochs = 50
    )

    # Data frame method — y as a vector
    model = train_nn(
        x = iris[, 2:4],
        y = iris$Sepal.Length,
        hidden_neurons = c(64, 32),
        activations = "relu",
        epochs = 50
    )

    # Data frame method — y as a formula evaluated against x
    model = train_nn(
        x = iris,
        y = Sepal.Length ~ . - Species,
        hidden_neurons = c(64, 32),
        activations = "relu",
        epochs = 50
    )

    # Formula method — outcome derived from formula
    model = train_nn(
        x = Sepal.Length ~ .,
        data = iris[, 1:4],
        hidden_neurons = c(64, 32),
        activations = "relu",
        epochs = 50
    )

    # No hidden layers — linear model
    model = train_nn(
        x = Sepal.Length ~ .,
        data = iris[, 1:4],
        epochs = 50
    )

    # Architecture object (nn_arch -> train_nn)
    mlp_arch = nn_arch(nn_name = "mlp_model")
    model = train_nn(
        x = Sepal.Length ~ .,
        data = iris[, 1:4],
        hidden_neurons = c(64, 32),
        activations = "relu",
```

```
    architecture = mlp_arch,
    epochs = 50
)

# Custom layer architecture
custom_linear = torch::nn_module(
    "CustomLinear",
    initialize = function(in_features, out_features, bias = TRUE) {
        self$layer = torch::nn_linear(in_features, out_features, bias = bias)
    },
    forward = function(x) self$layer(x)
)
custom_arch = nn_arch(
    nn_name = "custom_linear_mlp",
    nn_layer = ~ custom_linear
)
model = train_nn(
    x = Sepal.Length ~ .,
    data = iris[, 1:4],
    hidden_neurons = c(16, 8),
    activations = "relu",
    architecture = custom_arch,
    epochs = 50
)

# With early stopping
model = train_nn(
    x = Sepal.Length ~ .,
    data = iris[, 1:4],
    hidden_neurons = c(64, 32),
    activations = "relu",
    epochs = 200,
    validation_split = 0.2,
    early_stopping = early_stop(patience = 10)
)
}



if (torch::torch_is_installed()) {
    # torch dataset method — labels come from the dataset itself
    iris_cls_dataset = torch::dataset(
        name = "iris_cls_dataset",

        initialize = function(data = iris) {
            self$x = torch::torch_tensor(
                as.matrix(data[, 1:4]),
                dtype = torch::torch_float32()
            )
        # Species is a factor; convert to integer (1-indexed -> keep as-is for cross_entropy)
            self$y = torch::torch_tensor(
                as.integer(data$Species),
                dtype = torch::torch_long()
```

```
            )
        },

        .getitem = function(i) {
            list(self$x[i, ], self$y[i])
        },

        .length = function() {
            self$x$size(1)
        }
    )()

    model_nn_ds = train_nn(
        x = iris_cls_dataset,
        hidden_neurons = c(32, 10),
        activations = "relu",
        epochs = 80,
        batch_size = 16,
        learn_rate = 0.01,
        n_classes = 3, # Iris dataset has only 3 species
        validation_split = 0.2,
        verbose = TRUE
    )

    pred_nn = predict(model_nn_ds, iris_cls_dataset)
   class_preds = c("Setosa", "Versicolor", "Virginica")[predict(model_nn_ds, iris_cls_dataset)]

    # Confusion Matrix
    table(actual = iris$Species, pred = class_preds)
}
```

`grid_depth` *Depth-Aware Grid Generation for Neural Networks*

Description

`grid_depth()` extends standard grid generation to support multi-layer neural network architectures. It creates heterogeneous layer configurations by generating list columns for `hidden_neurons` and `activations`.

Usage

```
grid_depth(
  x,
  ...,
  n_hlayer = 2L,
  size = 5L,
 type = c("regular", "random", "latin_hypercube", "max_entropy", "audze_eglais"),
```

```
  original = TRUE,
  levels = 3L,
  variogram_range = 0.5,
  iter = 1000L
)

## S3 method for class 'parameters'
grid_depth(
  x,
  ...,
  n_hlayer = 2L,
  size = 5L,
 type = c("regular", "random", "latin_hypercube", "max_entropy", "audze_eglais"),
  original = TRUE,
  levels = 3L,
  variogram_range = 0.5,
  iter = 1000L
)

## S3 method for class 'list'
grid_depth(
  x,
  ...,
  n_hlayer = 2L,
  size = 5L,
 type = c("regular", "random", "latin_hypercube", "max_entropy", "audze_eglais"),
  original = TRUE,
  levels = 3L,
  variogram_range = 0.5,
  iter = 1000L
)

## S3 method for class 'workflow'
grid_depth(
  x,
  ...,
  n_hlayer = 2L,
  size = 5L,
 type = c("regular", "random", "latin_hypercube", "max_entropy", "audze_eglais"),
  original = TRUE,
  levels = 3L,
  variogram_range = 0.5,
  iter = 1000L
)

## S3 method for class 'model_spec'
grid_depth(
  x,
```

```
  ...,
  n_hlayer = 2L,
  size = 5L,
 type = c("regular", "random", "latin_hypercube", "max_entropy", "audze_eglais"),
  original = TRUE,
  levels = 3L,
  variogram_range = 0.5,
  iter = 1000L
)

## S3 method for class 'param'
grid_depth(
  x,
  ...,
  n_hlayer = 2L,
  size = 5L,
 type = c("regular", "random", "latin_hypercube", "max_entropy", "audze_eglais"),
  original = TRUE,
  levels = 3L,
  variogram_range = 0.5,
  iter = 1000L
)

## Default S3 method:
grid_depth(
  x,
  ...,
  n_hlayer = 2L,
  size = 5L,
 type = c("regular", "random", "latin_hypercube", "max_entropy", "audze_eglais"),
  original = TRUE,
  levels = 3L,
  variogram_range = 0.5,
  iter = 1000L
)
```

Arguments

x	A parameters object, list, workflow, or model spec. Can also be a single param object if ... contains additional param objects.
...	One or more param objects (e.g., hidden_neurons(), epochs()). If x is a parameters object, ... is ignored. None of the objects can have unknown() values.
n_hlayer	Integer vector specifying number of hidden layers to generate (e.g., 2:4 for 2, 3, or 4 layers), or a param object created with n_hlayers(). Default is 2.
size	Integer. Number of parameter combinations to generate.
type	Character. Type of grid: "regular", "random", "latin_hypercube", "max_entropy", or "audze_eglais".

original	Logical. Should original parameter ranges be used?
levels	Integer. Levels per parameter for regular grids.
variogram_range	Numeric. Range for audze_eglais design.
iter	Integer. Iterations for max_entropy optimization.

Details

This function is specifically for {kindling} models. The n_hlayer parameter determines network depth and creates list columns for hidden_neurons and activations, where each element is a vector of length matching the sampled depth.

When n_hlayer is a parameter object (created with n_hlayers()), it will be treated as a tunable parameter and sampled according to its defined range.

Value

A tibble with list columns for hidden_neurons and activations, where each element is a vector of length n_hlayer.

Examples

```
## Not run:
library(dials)
library(workflows)
library(tune)

# Method 1: Fixed depth
grid = grid_depth(
    hidden_neurons(c(32L, 128L)),
    activations(c("relu", "elu")),
    epochs(c(50L, 200L)),
    n_hlayer = 2:3,
    type = "random",
    size = 20
)

# Method 2: Tunable depth using parameter object
grid = grid_depth(
    hidden_neurons(c(32L, 128L)),
    activations(c("relu", "elu")),
    epochs(c(50L, 200L)),
    n_hlayer = n_hlayers(range = c(2L, 4L)),
    type = "random",
    size = 20
)

# Method 3: From workflow
wf = workflow() |>
    add_model(mlp_kindling(hidden_neurons = tune(), activations = tune())) |>
    add_formula(y ~ .)
grid = grid_depth(wf, n_hlayer = 2:4, type = "latin_hypercube", size = 15)
```

```
## End(Not run)
```

`kindling-basemodels` *Base models for Neural Network Training in kindling*

Description

Base models for Neural Network Training in kindling

Usage

```
ffnn(
  formula = NULL,
  data = NULL,
  hidden_neurons,
  activations = NULL,
  output_activation = NULL,
  bias = TRUE,
  epochs = 100,
  batch_size = 32,
  penalty = 0,
  mixture = 0,
  learn_rate = 0.001,
  optimizer = "adam",
  optimizer_args = list(),
  loss = "mse",
  validation_split = 0,
  device = NULL,
  verbose = FALSE,
  cache_weights = FALSE,
  ...,
  x = NULL,
  y = NULL
)

rnn(
  formula = NULL,
  data = NULL,
  hidden_neurons,
  rnn_type = "lstm",
  activations = NULL,
  output_activation = NULL,
  bias = TRUE,
  bidirectional = TRUE,
```

```
  dropout = 0,
  epochs = 100,
  batch_size = 32,
  penalty = 0,
  mixture = 0,
  learn_rate = 0.001,
  optimizer = "adam",
  optimizer_args = list(),
  loss = "mse",
  validation_split = 0,
  device = NULL,
  verbose = FALSE,
  cache_weights = FALSE,
  ...,
  x = NULL,
  y = NULL
)
```

Arguments

formula	Formula. Model formula (e.g., y ~ x1 + x2).
data	Data frame. Training data.
hidden_neurons	Integer vector. Number of neurons in each hidden layer.
activations	Activation function specifications. See act_funs().
output_activation	Optional. Activation for output layer.
bias	Logical. Use bias weights. Default TRUE.
epochs	Integer. Number of training epochs. Default 100.
batch_size	Integer. Batch size for training. Default 32.
penalty	Numeric. Regularization penalty (lambda). Default 0 (no regularization).
mixture	Numeric. Elastic net mixing parameter (0-1). Default 0.
learn_rate	Numeric. Learning rate for optimizer. Default 0.001.
optimizer	Character. Optimizer type ("adam", "sgd", "rmsprop"). Default "adam".
optimizer_args	Named list. Additional arguments passed to the optimizer. Default list().
loss	Character. Loss function ("mse", "mae", "cross_entropy", "bce"). Default "mse".
validation_split	Numeric. Proportion of data for validation (0-1). Default 0.
device	Character. Device to use ("cpu", "cuda", "mps"). Default NULL (auto-detect).
verbose	Logical. Print training progress. Default FALSE.
cache_weights	Logical. Cache weight matrices for faster variable importance. Default FALSE.
...	Additional arguments. Can be used to pass x and y for direct interface.
x	When not using formula: predictor data (data.frame or matrix).
y	When not using formula: outcome data (vector, factor, or matrix).

rnn_type	Character. Type of RNN ("rnn", "lstm", "gru"). Default "lstm".
bidirectional	Logical. Use bidirectional RNN. Default TRUE.
dropout	Numeric. Dropout rate between layers. Default 0.

Value

An object of class "ffnn_fit" containing the trained model and metadata.

FFNN

Train a feed-forward neural network using the torch package.

RNN

Train a recurrent neural network using the torch package.

Examples

```
if (torch::torch_is_installed()) {
    # Formula interface (original)
    model_reg = ffnn(
        Sepal.Length ~ .,
        data = iris[, 1:4],
        hidden_neurons = c(64, 32),
        activations = "relu",
        epochs = 50
    )

    # XY interface (new)
    model_xy = ffnn(
        hidden_neurons = c(64, 32),
        activations = "relu",
        epochs = 50,
        x = iris[, 2:4],
        y = iris$Sepal.Length
    )
}



if (torch::torch_is_installed()) {
    # Formula interface (original)
    model_rnn = rnn(
        Sepal.Length ~ .,
        data = iris[, 1:4],
        hidden_neurons = c(64, 32),
        rnn_type = "lstm",
        activations = "relu",
        epochs = 50
    )

    # XY interface (new)
```

```
    model_xy = rnn(
        hidden_neurons = c(64, 32),
        rnn_type = "gru",
        epochs = 50,
        x = iris[, 2:4],
        y = iris$Sepal.Length
    )
}
```

---

kindling-varimp *Variable Importance Methods for kindling Models*

Description

This file implements methods for variable importance generics from NeuralNetTools and vip packages.

Usage

```
## S3 method for class 'ffnn_fit'
garson(mod_in, bar_plot = FALSE, ...)

## S3 method for class 'ffnn_fit'
olden(mod_in, bar_plot = TRUE, ...)

vi_model.ffnn_fit(object, type = c("olden", "garson"), ...)
```

Arguments

mod_in	A fitted model object of class "ffnn_fit".
bar_plot	Logical. Whether to plot variable importance (default TRUE).
...	Additional arguments passed to methods.
object	A fitted model object of class "ffnn_fit".
type	Type of algorithm to extract the variable importance. This must be one of the strings: • 'olden' • 'garson'

Value

A data frame for both "garson" and "olden" classes with columns:

x_names	Character vector of predictor variable names
y_names	Character string of response variable name

`rel_imp` Numeric vector of relative importance scores (percentage)

The data frame is sorted by importance in descending order.

A tibble with columns "Variable" and "Importance" (via `vip::vi()` / `vip::vi_model()` only).

Garson's Algorithm for FFNN Models

{kindling} inherits `NeuralNetTools::garson` to extract the variable importance from the fitted `ffnn()` model.

Olden's Algorithm for FFNN Models

{kindling} inherits `NeuralNetTools::olden` to extract the variable importance from the fitted `ffnn()` model.

Variable Importance via {vip} Package

If the {vip} package (Suggests only) is installed, `kindling` registers a `vip::vi_model()` method for `"ffnn_fit"` objects, so `vip::vi()` and `vip::vi_model()` work out of the box.

References

Beck, M.W. 2018. NeuralNetTools: Visualization and Analysis Tools for Neural Networks. Journal of Statistical Software. 85(11):1-20.

Garson, G.D. 1991. Interpreting neural network connection weights. Artificial Intelligence Expert. 6(4):46-51.

Goh, A.T.C. 1995. Back-propagation neural networks for modeling complex systems. Artificial Intelligence in Engineering. 9(3):143-151.

Olden, J.D., Jackson, D.A. 2002. Illuminating the 'black-box': a randomization approach for understanding variable contributions in artificial neural networks. Ecological Modelling. 154:135-150.

Olden, J.D., Joy, M.K., Death, R.G. 2004. An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data. Ecological Modelling. 178:389-397.

Examples

```
if (torch::torch_is_installed()) {
    model_mlp = ffnn(
        Species ~ .,
        data = iris,
        hidden_neurons = c(64, 32),
        activations = "relu",
        epochs = 100,
        verbose = FALSE,
        cache_weights = TRUE
    )

    # Directly use `NeuralNetTools::garson`
    model_mlp |>
```

```
        garson()

    # Directly use `NeuralNetTools::olden`
    model_mlp |>
        olden()
} else {
    message("Torch not fully installed — skipping example")
}



# kindling also supports `vip::vi()` / `vip::vi_model()` when {vip} is installed
if (torch::torch_is_installed() && requireNamespace("vip", quietly = TRUE)) {
    model_mlp = ffnn(
        Species ~ .,
        data = iris,
        hidden_neurons = c(64, 32),
        activations = "relu",
        epochs = 100,
        verbose = FALSE,
        cache_weights = TRUE
    )

    model_mlp |>
        vip::vi(type = 'garson') |>
        vip::vip()
} else {
    message("Torch and/or {vip} not installed — skipping example")
}
```

`layer_prs` *Layer argument pronouns for formula-based specifications*

Description

These pronouns provide a cleaner, more readable way to reference layer parameters in formula-based specifications for `nn_module_generator()` and related functions. They work similarly to `rlang::.data` and `rlang::.env`.

Usage

```
.layer

.i

.in

.out

.is_output
```

Format

An object of class `layer_pr` (inherits from `list`) of length 0.

An object of class `layer_index_pr` (inherits from `layer_pr`, `list`) of length 0.

An object of class `layer_input_pr` (inherits from `layer_pr`, `list`) of length 0.

An object of class `layer_output_pr` (inherits from `layer_pr`, `list`) of length 0.

An object of class `layer_is_output_pr` (inherits from `layer_pr`, `list`) of length 0.

Details

Available pronouns:

- `.layer`: Access all layer parameters as a list-like object
- `.i`: Layer index (1-based integer)
- `.in`: Input dimension for the layer
- `.out`: Output dimension for the layer
- `.is_output`: Logical indicating if this is the output layer

These pronouns can be used in formulas passed to:

- `layer_arg_fn` parameter
- Custom layer configuration functions

Usage

```
# Using individual pronouns
layer_arg_fn = ~ list(
    input_size = .in,
    hidden_size = .out,
    num_layers = if (.i == 1) 2L else 1L
)

# Using .layer pronoun (alternative syntax)
layer_arg_fn = ~ list(
    input_size = .layer$ind,
    hidden_size = .layer$out,
    is_first = .layer$i == 1
)
```

mlp_kindling *Multi-Layer Perceptron (Feedforward Neural Network) via kindling*

Description

mlp_kindling() defines a feedforward neural network model that can be used for classification or regression. It integrates with the tidymodels ecosystem and uses the torch backend via kindling.

Usage

```
mlp_kindling(
  mode = "unknown",
  engine = "kindling",
  hidden_neurons = NULL,
  activations = NULL,
  output_activation = NULL,
  bias = NULL,
  epochs = NULL,
  batch_size = NULL,
  penalty = NULL,
  mixture = NULL,
  learn_rate = NULL,
  optimizer = NULL,
  validation_split = NULL,
  optimizer_args = NULL,
  loss = NULL,
  architecture = NULL,
  flatten_input = NULL,
  early_stopping = NULL,
  device = NULL,
  verbose = NULL,
  cache_weights = NULL
)
```

Arguments

mode	A single character string for the type of model. Possible values are "unknown", "regression", or "classification".
engine	A single character string specifying what computational engine to use for fitting. Currently only "kindling" is supported.
hidden_neurons	An integer vector for the number of units in each hidden layer. Can be tuned.
activations	A character vector of activation function names for each hidden layer (e.g., "relu", "tanh", "sigmoid"). Can be tuned.
output_activation	A character string for the output activation function. Can be tuned.
bias	Logical for whether to include bias terms. Can be tuned.

epochs	An integer for the number of training iterations. Can be tuned.
batch_size	An integer for the batch size during training. Can be tuned.
penalty	A number for the regularization penalty (lambda). Default 0 (no regularization). Higher values increase regularization strength. Can be tuned.
mixture	A number between 0 and 1 for the elastic net mixing parameter. Default 0 (pure L2/Ridge regularization). • 0: Pure L2 regularization (Ridge) • 1: Pure L1 regularization (Lasso) • 0 < mixture < 1: Elastic net (combination of L1 and L2) Only relevant when penalty > 0. Can be tuned.
learn_rate	A number for the learning rate. Can be tuned.
optimizer	A character string for the optimizer type ("adam", "sgd", "rmsprop"). Can be tuned.
validation_split	A number between 0 and 1 for the proportion of data used for validation. Can be tuned.
optimizer_args	A named list of additional arguments passed to the optimizer. Cannot be tuned — pass via set_engine().
loss	A character string for the loss function ("mse", "mae", "cross_entropy", "bce"). Cannot be tuned — pass via set_engine().
architecture	An nn_arch() object for a custom architecture. Cannot be tuned — pass via set_engine().
flatten_input	Logical or NULL. Controls input flattening. Cannot be tuned — pass via set_engine().
early_stopping	An early_stop() object or NULL. Cannot be tuned — pass via set_engine().
device	A character string for the device ("cpu", "cuda", "mps"). Cannot be tuned — pass via set_engine().
verbose	Logical for whether to print training progress. Cannot be tuned — pass via set_engine().
cache_weights	Logical. If TRUE, stores trained weight matrices in the returned object. Cannot be tuned — pass via set_engine().

Details

This function creates a model specification for a feedforward neural network that can be used within tidymodels workflows. The model supports:

- Multiple hidden layers with configurable units
- Various activation functions per layer
- GPU acceleration (CUDA, MPS, or CPU)
- Hyperparameter tuning integration
- Both regression and classification tasks

Parameters that cannot be tuned (architecture, flatten_input, early_stopping, device, verbose, cache_weights, optimizer_args, loss) must be set via set_engine(), not as arguments to mlp_kindling().

Value

A model specification object with class `mlp_kindling`.

Examples

```
if (torch::torch_is_installed()) {
    box::use(
        recipes[recipe],
        workflows[workflow, add_recipe, add_model],
        tune[tune],
        parsnip[fit]
    )

    # library(recipes)
    # library(workflows)
    # library(parsnip)
    # library(tune)

    # Model specs
    mlp_spec = mlp_kindling(
        mode = "classification",
        hidden_neurons = c(128, 64, 32),
        activation = c("relu", "relu", "relu"),
        epochs = 100
    )

    # If you want to tune
    mlp_tune_spec = mlp_kindling(
        mode = "classification",
        hidden_neurons = tune(),
        activation = tune(),
        epochs = tune(),
        learn_rate = tune()
    )
     wf = workflow() |>
        add_recipe(recipe(Species ~ ., data = iris)) |>
        add_model(mlp_spec)

     fit_wf = fit(wf, data = iris)
} else {
    message("Torch not fully installed — skipping example")
}
```

`new_act_fn` *Custom Activation Function Constructor*

Description

[Experimental]

Wraps a user-supplied function into a validated custom activation, ensuring it accepts and returns a `torch_tensor`. Performs an eager dry-run probe at *definition time* so errors surface early, and wraps the function with a *call-time* type guard for safety.

Usage

```
new_act_fn(fn, probe = TRUE, .name = "<custom>")
```

Arguments

`fn`	A function taking a single tensor argument and returning a tensor. E.g. `\(x) torch::torch_tanh(x)`.
`probe`	Logical. If `TRUE` (default), runs a dry-run with a small dummy tensor at definition time to catch obvious errors early.
`.name`	A string to be stored in an attribute. Nothing special, except it is used when displaying the info of a trained neural network model. Default is `"<custom>"`.

Value

An object of class `c("custom_activation", "parameterized_activation")`, compatible with `act_funs()`.

Examples

```
## Not run:
\donttest{
act_funs(relu, elu, new_act_fn(\(x) torch::torch_tanh(x)))
act_funs(new_act_fn(\(x) torch::nnf_silu(x)))
}

## End(Not run)
```

---

nn_arch *Architecture specification for train_nn()*

Description

`nn_arch()` defines an architecture specification object consumed by `train_nn()` via `architecture` (or legacy `arch`).

Conceptual workflow:

1. Define architecture with `nn_arch()`.
2. Train with `train_nn(..., architecture = <nn_arch>)`.
3. Predict with `predict()`.

Architecture fields mirror `nn_module_generator()` and are passed through without additional branching logic.

Usage

```
nn_arch(
  nn_name = "nnModule",
  nn_layer = NULL,
  out_nn_layer = NULL,
  nn_layer_args = list(),
  layer_arg_fn = NULL,
  forward_extract = NULL,
  before_output_transform = NULL,
  after_output_transform = NULL,
  last_layer_args = list(),
  input_transform = NULL
)
```

Arguments

nn_name	Character. Name of the generated module class. Default "nnModule".
nn_layer	Layer type. See nn_module_generator(). Default NULL (nn_linear).
out_nn_layer	Optional. Layer type forced on the last layer. Default NULL.
nn_layer_args	Named list. Additional arguments passed to every layer constructor. Default list().
layer_arg_fn	Formula or function. Generates per-layer constructor arguments. Default NULL (FFNN-style: list(in_dim, out_dim, bias = bias)).
forward_extract	Formula or function. Processes layer output in the forward pass. Default NULL.
before_output_transform	Formula or function. Transforms input before the output layer. Default NULL.
after_output_transform	Formula or function. Transforms output after the output layer. Default NULL.
last_layer_args	Named list or formula. Extra arguments for the output layer only. Default list().
input_transform	Formula or function. Transforms the entire input tensor before training begins. Applied once to the full dataset tensor, not per-batch. Transforms must therefore be independent of batch size. Safe examples: ~ .$unsqueeze(2) (RNN sequence dim), ~ .$unsqueeze(1) (CNN channel dim). Avoid transforms that reshape based on .$size(1) as this will reflect the full dataset size, not the mini-batch size.

Value

An object of class c("nn_arch", "kindling_arch"), implemented as a named list of nn_module_generator() arguments with an "env" attribute capturing the calling environment for custom layer resolution.

Examples

```
if (torch::torch_is_installed()) {
    # Standard architecture object for train_nn()
    std_arch = nn_arch(nn_name = "mlp_model")

    # GRU architecture spec
    gru_arch = nn_arch(
        nn_name = "GRU",
        nn_layer = "torch::nn_gru",
        layer_arg_fn = ~ if (.is_output) {
            list(.in, .out)
        } else {
            list(input_size = .in, hidden_size = .out, batch_first = TRUE)
        },
        out_nn_layer = "torch::nn_linear",
        forward_extract = ~ .[[1]],
        before_output_transform = ~ .[, .$size(2), ],
        input_transform = ~ .$unsqueeze(2)
    )

    # Custom layer architecture (resolved from calling environment)
    custom_linear = torch::nn_module(
        "CustomLinear",
        initialize = function(in_features, out_features, bias = TRUE) {
            self$layer = torch::nn_linear(in_features, out_features, bias = bias)
        },
        forward = function(x) self$layer(x)
    )
    custom_arch = nn_arch(
        nn_name = "CustomMLP",
        nn_layer = ~ custom_linear
    )

    model = train_nn(
        Sepal.Length ~ .,
        data = iris[, 1:4],
        hidden_neurons = c(64, 32),
        activations = "relu",
        epochs = 50,
        architecture = gru_arch
    )
}
```

---

nn_gens *Functions to generate* nn_module *(language) expression*

Description

Functions to generate nn_module (language) expression

Usage

```
ffnn_generator(
  nn_name = "DeepFFN",
  hd_neurons,
  no_x,
  no_y,
  activations = NULL,
  output_activation = NULL,
  bias = TRUE
)

rnn_generator(
  nn_name = "DeepRNN",
  hd_neurons,
  no_x,
  no_y,
  rnn_type = "lstm",
  bias = TRUE,
  activations = NULL,
  output_activation = NULL,
  bidirectional = TRUE,
  dropout = 0,
  ...
)
```

Arguments

nn_name	Character. Name of the generated RNN module class. Default is "DeepRNN".
hd_neurons	Integer vector. Number of neurons in each hidden RNN layer.
no_x	Integer. Number of input features.
no_y	Integer. Number of output features.
activations	Activation function specifications for each hidden layer. Can be: • NULL: No activation functions. • Character vector: e.g., c("relu", "sigmoid"). • List: e.g., act_funs(relu, elu, softshrink = args(lambd = 0.5)). • activation_spec object from act_funs(). If the length of activations is 1L, this will be the activation throughout the architecture.
output_activation	Optional. Activation function for the output layer. Same format as activations but should be a single activation.
bias	Logical. Whether to use bias weights. Default is TRUE
rnn_type	Character. Type of RNN to use. Must be one of "rnn", "lstm", or "gru". Default is "lstm".
bidirectional	Logical. Whether to use bidirectional RNN layers. Default is TRUE.

dropout	Numeric. Dropout rate between RNN layers. Default is 0.
...	Additional arguments (currently unused).

Details

The generated FFNN module will have the specified number of hidden layers, with each layer containing the specified number of neurons. Activation functions can be applied after each hidden layer as specified. This can be used for both classification and regression tasks.

The generated module properly namespaces all torch functions to avoid polluting the global namespace.

The generated RNN module will have the specified number of recurrent layers, with each layer containing the specified number of hidden units. Activation functions can be applied after each RNN layer as specified. The final output is taken from the last time step and passed through a linear layer.

The generated module properly namespaces all torch functions to avoid polluting the global namespace.

Value

A torch module expression representing the FFNN.

A torch module expression representing the RNN.

Feed-Forward Neural Network Module Generator

The ffnn_generator() function generates a feed-forward neural network (FFNN) module expression from the torch package in R. It allows customization of the FFNN architecture, including the number of hidden layers, neurons, and activation functions.

Recurrent Neural Network Module Generator

The rnn_generator() function generates a recurrent neural network (RNN) module expression from the torch package in R. It allows customization of the RNN architecture, including the number of hidden layers, neurons, RNN type, activation functions, and other parameters.

Examples

```
# FFNN
if (torch::torch_is_installed()) {
    # Generate an MLP module with 3 hidden layers
    ffnn_mod = ffnn_generator(
        nn_name = "MyFFNN",
        hd_neurons = c(64, 32, 16),
        no_x = 10,
        no_y = 1,
        activations = 'relu'
    )

    # Evaluate and instantiate
    model = eval(ffnn_mod)()
```

```
    # More complex: With different activations
    ffnn_mod2 = ffnn_generator(
        nn_name = "MyFFNN2",
        hd_neurons = c(128, 64, 32),
        no_x = 20,
        no_y = 5,
        activations = act_funs(
            relu,
            selu,
            sigmoid
        )
    )

    # Even more complex: Different activations and customized argument
    # for the specific activation function
    ffnn_mod2 = ffnn_generator(
        nn_name = "MyFFNN2",
        hd_neurons = c(128, 64, 32),
        no_x = 20,
        no_y = 5,
        activations = act_funs(
            relu,
            selu,
            softshrink = args(lambd = 0.5)
        )
    )

    # Customize output activation (softmax is useful for classification tasks)
    ffnn_mod3 = ffnn_generator(
        hd_neurons = c(64, 32),
        no_x = 10,
        no_y = 3,
        activations = 'relu',
        output_activation = act_funs(softmax = args(dim = 2L))
    )
} else {
    message("Torch not fully installed — skipping example")
}



## RNN
if (torch::torch_is_installed()) {
    # Basic LSTM with 2 layers
    rnn_mod = rnn_generator(
        nn_name = "MyLSTM",
        hd_neurons = c(64, 32),
        no_x = 10,
        no_y = 1,
        rnn_type = "lstm",
        activations = 'relu'
    )
```

```
    # Evaluate and instantiate
    model = eval(rnn_mod)()

    # GRU with different activations
    rnn_mod2 = rnn_generator(
        nn_name = "MyGRU",
        hd_neurons = c(128, 64, 32),
        no_x = 20,
        no_y = 5,
        rnn_type = "gru",
        activations = act_funs(relu, elu, relu),
        bidirectional = FALSE
    )

} else {
    message("Torch not fully installed — skipping example")
}


## Not run:
## Parameterized activation and dropout
# (Will throw an error due to `nnf_tanh()` not being available in `{torch}`)
# rnn_mod3 = rnn_generator(
#     hd_neurons = c(100, 50, 25),
#     no_x = 15,
#     no_y = 3,
#     rnn_type = "lstm",
#     activations = act_funs(
#         relu,
#         leaky_relu = args(negative_slope = 0.01),
#         tanh
#     ),
#     bidirectional = TRUE,
#     dropout = 0.3
# )

## End(Not run)
```

---

nn_module_generator *Generalized Neural Network Module Expression Generator*

Description

[Experimental]

nn_module_generator() is a generalized function that generates neural network module expressions for various architectures. It provides a flexible framework for creating custom neural network modules by parameterizing layer types, construction arguments, and forward pass behavior.

While designed primarily for {torch} modules, it can work with custom layer implementations from the current environment, including user-defined layers like RBF networks, custom attention mechanisms, or other novel architectures.

This function serves as the foundation for specialized generators like ffnn_generator() and rnn_generator(), but can be used directly to create custom architectures.

Usage

```
nn_module_generator(
  nn_name = "nnModule",
  nn_layer = NULL,
  out_nn_layer = NULL,
  nn_layer_args = list(),
  layer_arg_fn = NULL,
  forward_extract = NULL,
  before_output_transform = NULL,
  after_output_transform = NULL,
  last_layer_args = list(),
  hd_neurons,
  no_x,
  no_y,
  activations = NULL,
  output_activation = NULL,
  bias = TRUE,
  eval = FALSE,
  .env = parent.frame(),
  ...
)
```

Arguments

nn_name	Character string specifying the name of the generated neural network module class. Default is "nnModule".
nn_layer	The type of neural network layer to use. Can be specified as: • NULL (default): Uses nn_linear() from {torch} • Character string: e.g., "nn_linear", "nn_gru", "nn_lstm", "some_custom_layer" • Named function: A function object that constructs the layer • Anonymous function: e.g., \() nn_linear() or function() nn_linear() The layer constructor is first searched in the current environment, then in parent environments, and finally falls back to the {torch} namespace. This allows you to use custom layer implementations alongside standard torch layers.
out_nn_layer	Default NULL. If supplied, it forces to be the neural network layer to be used on the last layer. Can be specified as: • Character string, e.g. "nn_linear", "nn_gru", "nn_lstm", "some_custom_layer" • Named function: A function object that constructs the layer • Formula interface, e.g. ~torch::nn_linear, ~some_custom_layer

Internally, it almost works the same as `nn_layer` parameter.

`nn_layer_args` Named list of additional arguments passed to the layer constructor specified by `nn_layer`. These arguments are applied to all layers. For layer-specific arguments, use `layer_arg_fn`. Default is an empty list.

`layer_arg_fn` Optional function or formula that generates layer-specific construction arguments. Can be specified as:

- **Formula**: `~ list(input_size = .in, hidden_size = .out)` where `.in`, `.out`, `.i`, and `.is_output` are available
- **Function**: `function(i, in_dim, out_dim, is_output)` with signature as before

The formula/function should return a named list of arguments to pass to the layer constructor. Available variables in formula context:

- `.i` or `i`: Integer, the layer index (1-based)
- `.in` or `in_dim`: Integer, input dimension for this layer
- `.out` or `out_dim`: Integer, output dimension for this layer
- `.is_output` or `is_output`: Logical, whether this is the final output layer

If `NULL`, defaults to FFNN-style arguments: `list(in_dim, out_dim, bias = bias)`.

`forward_extract`

Optional formula or function that processes layer outputs in the forward pass. Useful for layers that return complex structures (e.g., RNNs return `list(output, hidden)`). Can be specified as:

- **Formula**: `~ .[[1]]` or `~ .$output` where `.` represents the layer output
- **Function**: `function(expr)` that accepts/returns a language object

Common patterns:

- Extract first element: `~ .[[1]]`
- Extract named element: `~ .$output`
- Extract with method: `~ .$get_output()`

If `NULL`, layer outputs are used directly.

`before_output_transform`

Optional formula or function that transforms input before the output layer. This is applied after the last hidden layer (and its activation) but before the output layer. Can be specified as:

- **Formula**: `~ .[, .$size(2), ]` where `.` represents the current tensor
- **Function**: `function(expr)` that accepts/returns a language object

Common patterns:

- Extract last timestep: `~ .[, .$size(2), ]`
- Flatten: `~ .$flatten(start_dim = 1)`
- Global pooling: `~ .$mean(dim = 2)`
- Extract token: `~ .[, 1, ]`

If `NULL`, no transformation is applied.

after_output_transform

Optional formula or function that transforms the output after the output layer. This is applied after `self$out(x)` (the final layer) but before returning the result. Can be specified as:

- **Formula**: `~ .$mean(dim = 2)` where `.` represents the output tensor
- **Function**: `function(expr)` that accepts/returns a language object

Common patterns:

- Global average pooling: `~ .$mean(dim = 2)`
- Squeeze dimensions: `~ .$squeeze()`
- Reshape output: `~ .$view(c(-1, 10))`
- Extract specific outputs: `~ .[, , 1:5]`

If `NULL`, no transformation is applied.

last_layer_args

Optional named list or formula specifying additional arguments for the output layer only. These arguments are appended to the output layer constructor after the arguments from `layer_arg_fn`. Can be specified as:

- **Formula**: `~ list(kernel_size = 2L, bias = FALSE)`
- **Named list**: `list(kernel_size = 2L, bias = FALSE)`

This is useful when you need to override or add specific parameters to the final layer without affecting hidden layers. For example, in CNNs you might want a different kernel size for the output layer, or in RNNs you might want to disable bias in the final linear projection. Arguments in `last_layer_args` will override any conflicting arguments from `layer_arg_fn` when `.is_output = TRUE`. Default is an empty list.

hd_neurons — Integer vector specifying the number of neurons (hidden units) in each hidden layer. The length determines the number of hidden layers in the network. Must contain at least one element.

no_x — Integer specifying the number of input features (input dimension).

no_y — Integer specifying the number of output features (output dimension).

activations — Activation function specifications for hidden layers. Can be:

- `NULL`: No activation functions applied
- Character vector: e.g., `c("relu", "sigmoid", "tanh")`
- `activation_spec` object: Created using `act_funs()`, which allows specifying custom arguments. See examples.

If a single activation is provided, it will be replicated across all hidden layers. Otherwise, the length should match the number of hidden layers.

output_activation

Optional activation function for the output layer. Same format as `activations`, but should specify only a single activation. Common choices include `"softmax"` for classification or `"sigmoid"` for binary outcomes. Default is `NULL` (no output activation).

bias — Logical indicating whether to include bias terms in layers. Default is `TRUE`. Note that this is passed to `layer_arg_fn` if provided, so custom layer argument functions should handle this parameter appropriately.

eval	Logical indicating whether to evaluate the generated expression immediately. If TRUE, returns an instantiated nn_module class that can be called directly (e.g., model()). If FALSE (default), returns the unevaluated language expression that can be inspected or evaluated later with eval(). Default is FALSE.
.env	Default is parent.frame(). The environment in which the generated expression is to be evaluated
...	Additional arguments passed to layer constructors or for future extensions.

Value

If eval = FALSE (default): A language object (unevaluated expression) representing a torch::nn_module definition. This expression can be evaluated with eval() to create the module class, which can then be instantiated with eval(result)() to create a model instance.

If eval = TRUE: An instantiated nn_module class constructor that can be called directly to create model instances (e.g., result()).

Examples

```
## Not run:
\donttest{
if (torch::torch_is_installed()) {
    # Basic usage with formula interface
    nn_module_generator(
        nn_name = "MyGRU",
        nn_layer = "nn_gru",
        layer_arg_fn = ~ if (.is_output) {
            list(.in, .out)
        } else {
            list(input_size = .in, hidden_size = .out,
                 num_layers = 1L, batch_first = TRUE)
        },
        forward_extract = ~ .[[1]],
        before_output_transform = ~ .[, .$size(2), ],
        hd_neurons = c(128, 64, 32),
        no_x = 20,
        no_y = 5,
        activations = "relu"
    )

    # LSTM with cleaner syntax
    nn_module_generator(
        nn_name = "MyLSTM",
        nn_layer = "nn_lstm",
        layer_arg_fn = ~ list(
            input_size = .in,
            hidden_size = .out,
            batch_first = TRUE
        ),
        forward_extract = ~ .[[1]],
        before_output_transform = ~ .[, .$size(2), ],
        hd_neurons = c(64, 32),
```

```
    no_x = 10,
    no_y = 2
)

# CNN with global average pooling
nn_module_generator(
    nn_name = "SimpleCNN",
    nn_layer = "nn_conv1d",
    layer_arg_fn = ~ list(
        in_channels = .in,
        out_channels = .out,
        kernel_size = 3L,
        padding = 1L
    ),
    before_output_transform = ~ .$mean(dim = 2),
    hd_neurons = c(16, 32, 64),
    no_x = 1,
    no_y = 10,
    activations = "relu"
)

# CNN with after_output_transform (pooling applied AFTER output layer)
nn_module_generator(
    nn_name = "CNN1DClassifier",
    nn_layer = "nn_conv1d",
    layer_arg_fn = ~ if (.is_output) {
        list(.in, .out)
    } else {
        list(
            in_channels = .in,
            out_channels = .out,
            kernel_size = 3L,
            stride = 1L,
            padding = 1L
        )
    },
    after_output_transform = ~ .$mean(dim = 2),
    last_layer_args = list(kernel_size = 1, stride = 2),
    hd_neurons = c(16, 32, 64),
    no_x = 1,
    no_y = 10,
    activations = "relu"
)

} else {
  message("torch not installed - skipping examples")
}
}

## End(Not run)
```

`ordinal_gen` *Ordinal Suffixes Generator*

Description

This function is originally from `numform::f_ordinal()`.

Usage

```
ordinal_gen(x)
```

Arguments

`x`	Vector of numbers. Could be a string equivalent

Value

Returns a string vector with ordinal suffixes.

This is how you use it

```
kindling:::ordinal_gen(1:10)
```

Note: This is not exported into public namespace. So please, refer to `numform::f_ordinal()` instead.

References

Rinker, T. W. (2021). numform: A publication style number and plot formatter version 0.7.0. https://github.com/trinker/numform

`rnn_kindling` *Recurrent Neural Network via kindling*

Description

`rnn_kindling()` defines a recurrent neural network model that can be used for classification or regression on sequential data. It integrates with the tidymodels ecosystem and uses the torch backend via kindling.

Usage

```
rnn_kindling(
  mode = "unknown",
  engine = "kindling",
  hidden_neurons = NULL,
  activations = NULL,
  output_activation = NULL,
  bias = NULL,
  bidirectional = NULL,
  dropout = NULL,
  epochs = NULL,
  batch_size = NULL,
  penalty = NULL,
  mixture = NULL,
  learn_rate = NULL,
  optimizer = NULL,
  validation_split = NULL,
  rnn_type = NULL,
  optimizer_args = NULL,
  loss = NULL,
  early_stopping = NULL,
  device = NULL,
  verbose = NULL,
  cache_weights = NULL
)
```

Arguments

mode	A single character string for the type of model. Possible values are "unknown", "regression", or "classification".
engine	A single character string specifying what computational engine to use for fitting. Currently only "kindling" is supported.
hidden_neurons	An integer vector for the number of units in each hidden layer. Can be tuned.
activations	A character vector of activation function names for each hidden layer (e.g., "relu", "tanh", "sigmoid"). Can be tuned.
output_activation	A character string for the output activation function. Can be tuned.
bias	Logical for whether to include bias terms. Can be tuned.
bidirectional	A logical indicating whether to use bidirectional RNN. Can be tuned.
dropout	A number between 0 and 1 for dropout rate between layers. Can be tuned.
epochs	An integer for the number of training iterations. Can be tuned.
batch_size	An integer for the batch size during training. Can be tuned.
penalty	A number for the regularization penalty (lambda). Default `0` (no regularization). Higher values increase regularization strength. Can be tuned.

`mixture`	A number between 0 and 1 for the elastic net mixing parameter. Default `0` (pure L2/Ridge regularization). • `0`: Pure L2 regularization (Ridge) • `1`: Pure L1 regularization (Lasso) • `0 < mixture < 1`: Elastic net (combination of L1 and L2) Only relevant when `penalty > 0`. Can be tuned.
`learn_rate`	A number for the learning rate. Can be tuned.
`optimizer`	A character string for the optimizer type ("adam", "sgd", "rmsprop"). Can be tuned.
`validation_split`	A number between 0 and 1 for the proportion of data used for validation. Can be tuned.
`rnn_type`	A character string for the type of RNN cell ("rnn", "lstm", "gru"). Cannot be tuned — pass via `set_engine()`.
`optimizer_args`	A named list of additional arguments passed to the optimizer. Cannot be tuned — pass via `set_engine()`.
`loss`	A character string for the loss function ("mse", "mae", "cross_entropy", "bce"). Cannot be tuned — pass via `set_engine()`.
`early_stopping`	An `early_stop()` object or `NULL`. Cannot be tuned — pass via `set_engine()`.
`device`	A character string for the device ("cpu", "cuda", "mps"). Cannot be tuned — pass via `set_engine()`.
`verbose`	Logical for whether to print training progress. Cannot be tuned — pass via `set_engine()`.
`cache_weights`	Logical. If `TRUE`, stores trained weight matrices in the returned object. Cannot be tuned — pass via `set_engine()`.

Details

This function creates a model specification for a recurrent neural network that can be used within tidymodels workflows. The model supports:

- Multiple RNN types: basic RNN, LSTM, and GRU
- Bidirectional processing
- Dropout regularization
- GPU acceleration (CUDA, MPS, or CPU)
- Hyperparameter tuning integration
- Both regression and classification tasks

The `device` parameter controls where computation occurs:

- `NULL` (default): Auto-detect best available device (CUDA > MPS > CPU)
- `"cuda"`: Use NVIDIA GPU
- `"mps"`: Use Apple Silicon GPU
- `"cpu"`: Use CPU only

Value

A model specification object with class `rnn_kindling`.

Examples

```
if (torch::torch_is_installed()) {
    box::use(
        recipes[recipe],
        workflows[workflow, add_recipe, add_model],
        parsnip[fit]
    )

    # Model specs
    rnn_spec = rnn_kindling(
        mode = "classification",
        hidden_neurons = c(64, 32),
        rnn_type = "lstm",
        activation = c("relu", "elu"),
        epochs = 100,
        bidirectional = TRUE
    )

    wf = workflow() |>
        add_recipe(recipe(Species ~ ., data = iris)) |>
        add_model(rnn_spec)

    fit_wf = fit(wf, data = iris)
    fit_wf
} else {
    message("Torch not fully installed - skipping example")
}
```

---

table_summary *Summarize and Display a Two-Column Data Frame as a Formatted Table*

Description

This function takes a two-column data frame and formats it into a summary-like table. The table can be optionally split into two parts, centered, and given a title. It is useful for displaying summary information in a clean, tabular format. The function also supports styling with ANSI colors and text formatting through the `{cli}` package and column alignment options.

Usage

```
table_summary(
  data,
```

```
  title = NULL,
  l = NULL,
  header = FALSE,
  center_table = FALSE,
  border_char = "-",
  style = list(),
  align = NULL,
  ...
)
```

Arguments

data	A data frame with exactly two columns. The data to be summarized and displayed.
title	A character string. An optional title to be displayed above the table.
l	An integer. The number of rows to include in the left part of a split table. If NULL, the table is not split.
header	A logical value. If TRUE, the column names of data are displayed as a header.
center_table	A logical value. If TRUE, the table is centered in the terminal.
border_char	Character used for borders. Default is "\u2500".
style	A list controlling the visual styling of table elements using ANSI formatting. Can include the following components: • left_col: Styling for the left column values. • right_col: Styling for the right column values. • border_text: Styling for the border. • title: Styling for the title. • sep: Separator character between left and right column. Each style component can be either a predefined style string (e.g., "blue", "red_italic", "bold") or a function that takes a context list with/without a value element and returns the styled text.
align	Controls the alignment of column values. Can be specified in three ways: • A single string: affects only the left column (e.g., "left", "center", "right"). • A vector of two strings: affects both columns in order (e.g., c("left", "right")). • A list with named components: explicitly specifies alignment for each column
...	Additional arguments (currently unused).

Value

This function does not return a value. It prints the formatted table to the console.

Examples

```
# Create a sample data frame
df = data.frame(
    Category = c("A", "B", "C", "D", "E"),
    Value = c(10, 20, 30, 40, 50)
)

# Display the table with a title and header
table_summary(df, title = "Sample Table", header = TRUE)

# Split the table after the second row and center it
table_summary(df, l = 2, center_table = TRUE)

# Use styling and alignment
table_summary(
    df, header = TRUE,
    style = list(
        left_col = "blue_bold",
        right_col = "red",
        title = "green",
        border_text = "yellow"
    ),
    align = c("center", "right")
)

# Use custom styling with lambda functions
table_summary(
    df, header = TRUE,
    style = list(
        left_col = \(ctx) cli::col_red(ctx), # ctx$value is another option
        right_col = \(ctx) cli::col_blue(ctx)
    ),
    align = list(left_col = "left", right_col = "right")
)
```

---

train_nnsnip *Parsnip Interface of* train_nn()

Description

[Experimental]

train_nnsnip() defines a neural network model specification that can be used for classification or regression. It integrates with the tidymodels ecosystem and uses train_nn() as the fitting backend, supporting any architecture expressible via nn_arch() — feedforward, recurrent, convolutional, and beyond.

Usage

```
train_nnsnip(
  mode = "unknown",
  engine = "kindling",
  hidden_neurons = NULL,
  activations = NULL,
  output_activation = NULL,
  bias = NULL,
  epochs = NULL,
  batch_size = NULL,
  penalty = NULL,
  mixture = NULL,
  learn_rate = NULL,
  optimizer = NULL,
  validation_split = NULL,
  optimizer_args = NULL,
  loss = NULL,
  architecture = NULL,
  flatten_input = NULL,
  early_stopping = NULL,
  device = NULL,
  verbose = NULL,
  cache_weights = NULL
)
```

Arguments

mode	A single character string for the type of model. Possible values are "unknown", "regression", or "classification".
engine	A single character string specifying what computational engine to use for fitting. Currently only "kindling" is supported.
hidden_neurons	An integer vector for the number of units in each hidden layer. Can be tuned.
activations	A character vector of activation function names for each hidden layer (e.g., "relu", "tanh", "sigmoid"). Can be tuned.
output_activation	A character string for the output activation function. Can be tuned.
bias	Logical for whether to include bias terms. Can be tuned.
epochs	An integer for the number of training iterations. Can be tuned.
batch_size	An integer for the batch size during training. Can be tuned.
penalty	A number for the regularization penalty (lambda). Default `0` (no regularization). Higher values increase regularization strength. Can be tuned.
mixture	A number between 0 and 1 for the elastic net mixing parameter. Default `0` (pure L2/Ridge regularization). • `0`: Pure L2 regularization (Ridge) • `1`: Pure L1 regularization (Lasso)

- `0 < mixture < 1`: Elastic net (combination of L1 and L2) Only relevant when `penalty > 0`. Can be tuned.

`learn_rate` A number for the learning rate. Can be tuned.

`optimizer` A character string for the optimizer type ("adam", "sgd", "rmsprop"). Can be tuned.

`validation_split` A number between 0 and 1 for the proportion of data used for validation. Can be tuned.

`optimizer_args` A named list of additional arguments passed to the optimizer. Cannot be tuned — pass via `set_engine()`.

`loss` A character string or a valid `{torch}` function for the loss function ("mse", "mae", "cross_entropy", "bce"). Cannot be tuned — pass via `set_engine()`.

`architecture` An `nn_arch()` object for a custom architecture. Cannot be tuned — pass via `set_engine()`.

`flatten_input` Logical or `NULL`. Controls input flattening. Cannot be tuned — pass via `set_engine()`.

`early_stopping` An `early_stop()` object or `NULL`. Cannot be tuned — pass via `set_engine()`.

`device` A character string for the device to use ("cpu", "cuda", "mps"). If `NULL`, auto-detects the best available device. Cannot be tuned — pass via `set_engine()`.

`verbose` Logical for whether to print training progress. Default `FALSE`. Cannot be tuned — pass via `set_engine()`.

`cache_weights` Logical. If `TRUE`, stores trained weight matrices in the returned object. Cannot be tuned — pass via `set_engine()`.

Details

This function creates a model specification for a neural network that can be used within tidymodels workflows. The underlying engine is `train_nn()`, which is architecture-agnostic: when `architecture = NULL` it falls back to a standard feed-forward network, but any architecture expressible via `nn_arch()` can be used instead. The model supports:

- Configurable hidden layers and activation functions (default MLP path)
- Custom architectures via `nn_arch()` (recurrent, convolutional, etc.)
- GPU acceleration (CUDA, MPS, or CPU)
- Hyperparameter tuning integration
- Both regression and classification tasks

When using the default MLP path (no custom `architecture`), `hidden_neurons` accepts an integer vector where each element represents the number of neurons in that hidden layer. For example, `hidden_neurons = c(128, 64, 32)` creates a network with three hidden layers. Pass an `nn_arch()` object via `set_engine()` to use a custom architecture instead.

The `device` parameter controls where computation occurs:

- `NULL` (default): Auto-detect best available device (CUDA > MPS > CPU)
- `"cuda"`: Use NVIDIA GPU

- "mps": Use Apple Silicon GPU
- "cpu": Use CPU only

When tuning, you can use special tune tokens:

- For hidden_neurons: use tune("hidden_neurons") with a custom range
- For activation: use tune("activation") with values like "relu", "tanh"

Value

A model specification object with class train_nnsnip.

Examples

```
if (torch::torch_is_installed()) {
    box::use(
        recipes[recipe],
        workflows[workflow, add_recipe, add_model],
        tune[tune],
        parsnip[fit]
    )

    # Model spec
    nn_spec = train_nnsnip(
        mode = "classification",
        hidden_neurons = c(30, 5),
        activations = c("relu", "elu"),
        epochs = 100
    )

    wf = workflow() |>
        add_recipe(recipe(Species ~ ., data = iris)) |>
        add_model(nn_spec)

    fit_wf = fit(wf, data = iris)
} else {
    message("Torch not fully installed — skipping example")
}
```

Index

∗ **datasets**
    layer_prs, 23
.i *(*layer_prs*)*, 23
.in *(*layer_prs*)*, 23
.is_output *(*layer_prs*)*, 23
.layer *(*layer_prs*)*, 23
.out *(*layer_prs*)*, 23

act_funs, 2
act_funs(), *9*, *11*
args, 3
autoplot_diagnostics, 4

early_stop, 5
early_stop(), *9*, *11*, *26*, *42*, *47*

ffnn *(*kindling-basemodels*)*, 18
ffnn(), *4*
ffnn_generator *(*nn_gens*)*, 30

garson.ffnn_fit *(*kindling-varimp*)*, 21
gen-nn-train, 6
ggplot2::ggplot(), *4*
grid_depth, 14

kindling-basemodels, 18
kindling-varimp, 21

layer_prs, 23

mlp_kindling, 25

new_act_fn, 27
nn_arch, 28
nn_arch(), *6*, *9*, *11*, *26*, *45*, *47*
nn_gens, 30
nn_module_generator, 34

olden.ffnn_fit *(*kindling-varimp*)*, 21
ordinal_gen, 40

parent.frame(), *38*
plot_diagnostics
        *(*autoplot_diagnostics*)*, 4
predict.nn_fit(), *6*, *11*
predict.nn_fit_ds(), *11*

rnn *(*kindling-basemodels*)*, 18
rnn(), *4*
rnn_generator *(*nn_gens*)*, 30
rnn_kindling, 40

table_summary, 43
train_nn *(*gen-nn-train*)*, 6
train_nn(), *4*, *28*, *45*, *47*
train_nnsnip, 45

vi_model.ffnn_fit *(*kindling-varimp*)*, 21