Package ‘kindling’

February 4, 2026

Type Package

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

Version 0.2.0

Description Provides a higher-level interface to the 'torch' package for defining,
training, and fine-tuning neural networks, including its depth, powered by code generation.
This package currently supports few to several architectures, namely feedforward (multi-
layer perceptron)
and recurrent neural networks (Recurrent Neural Networks (RNN), Long Short-
Term Memory (LSTM), Gated Recurrent Unit (GRU)),
while also reduces boilerplate 'torch' code while enabling seamless integra-
tion with 'torch'. The model methods
to train neural networks from this package also bridges to titanic ML frameworks in R, namely
'tidymodels' ecosystem, which enables the '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, vip, ggplot2,
tune, dials, hardhat

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

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]

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

Repository CRAN

Date/Publication 2026-02-04 16:00:08 UTC

Contents

`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)`

Value

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

args — *Activation Function Arguments Helper*

Description

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.

ffnn — *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
    )
}
```

`ffnn_generator` *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)
```

ffnn_wrapper *Basemodels-tidymodels wrappers*

Description

Basemodels-tidymodels wrappers

Usage

```
ffnn_wrapper(formula, data, ...)

rnn_wrapper(formula, data, ...)
```

Arguments

formula	A formula specifying the model (e.g., y ~ x1 + x2)
data	A data frame containing the training data
...	Additional arguments passed to the underlying training function

Details

These wrapper functions are designed to interface with the `{tidymodels}` ecosystem, particularly for use with `tune::tune_grid()` and workflows. They handle the conversion of tuning parameters (especially list-column parameters from `grid_depth()`) into the format expected by `ffnn()` and `rnn()`.

Value

- `ffnn_wrapper()` returns an object of class `"ffnn_fit"` containing the trained feedforward neural network model and metadata. See `ffnn()` for details.
- `rnn_wrapper()` returns an object of class `"rnn_fit"` containing the trained recurrent neural network model and metadata. See `rnn()` for details.

FFNN (MLP) Wrapper for `{tidymodels}` interface

This is a function to interface into `{tidymodels}` (do not use this, use `kindling::ffnn()` instead).

RNN Wrapper for `{tidymodels}` interface

This is a function to interface into `{tidymodels}` (do not use this, use `kindling::rnn()` instead).

`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-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, ...)

## S3 method for class 'ffnn_fit'
vi_model(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

You can directly use `vip::vi()` and `vip::vi_model()` to extract the variable importance from the fitted `ffnn()` model.

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()`
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
    )

    model_mlp |>
        vip::vi(type = 'garson') |>
        vip::vip()
} else {
    message("Torch not fully installed - skipping example")
}
```

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,
  optimizer_args = NULL,
  loss = NULL,
  validation_split = NULL,
  device = NULL,
  verbose = 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.
`optimizer_args`	A named list of additional arguments passed to the optimizer. Cannot be tuned.
`loss`	A character string for the loss function ("mse", "mae", "cross_entropy", "bce"). Cannot be tuned.
`validation_split`	A number between 0 and 1 for the proportion of data used for validation. Can be tuned.
`device`	A character string for the device to use ("cpu", "cuda", "mps"). If NULL, auto-detects available GPU. Cannot be tuned.
`verbose`	Logical for whether to print training progress. Default FALSE. Cannot be tuned.

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

The `hidden_neurons` parameter 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.

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 `mlp_kindling`.

Examples

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

    # 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")
}
```

`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,
  rnn_type = 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,
  optimizer_args = NULL,
  loss = NULL,
  validation_split = NULL,
  device = NULL,
  verbose = 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.
`rnn_type`	A character string for the type of RNN cell ("rnn", "lstm", "gru"). Cannot 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.
`optimizer_args`	A named list of additional arguments passed to the optimizer. Cannot be tuned.
`loss`	A character string for the loss function ("mse", "mae", "cross_entropy", "bce"). Cannot be tuned.
`validation_split`	A number between 0 and 1 for the proportion of data used for validation. Can be tuned.
`device`	A character string for the device to use ("cpu", "cuda", "mps"). If NULL, auto-detects available GPU. Cannot be tuned.
`verbose`	Logical for whether to print training progress. Default FALSE. Cannot be tuned.

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")
)
```

Index