

Package ‘kerasnip’

September 3, 2026

Type Package

Title A Bridge Between 'keras' and 'tidymodels'

Version 0.2.0

Description Provides a seamless bridge between 'keras' and the 'tidymodels' frameworks. It allows for the dynamic creation of 'parsnip' model specifications for 'keras' models.

Depends R (>= 4.1.0)

Encoding UTF-8

License MIT + file LICENSE

URL <https://davidrsch.github.io/kerasnip/>,
<https://github.com/davidrsch/kerasnip>

BugReports <https://github.com/davidrsch/kerasnip/issues>

Imports abind, generics, parsnip (>= 1.0.0), rlang, keras3, tibble,
purrr, dplyr, cli, recipes, reticulate, lobstr

Config/testthat/edition 3

Suggests testthat (>= 3.0.0), bundle, butcher, modeldata, tidymodels,
finetune, tune, dials, workflows, rsample, stacks, knitr, lme4,
rmarkdown, future, ggplot2, mgcv, probably, tailor, hardhat,
tidyr

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author David Díaz [aut, cph, cre] (ORCID:
<https://orcid.org/0000-0002-0927-9795>)

Maintainer David Díaz <davidrsch@gmail.com>

Repository CRAN

Date/Publication 2026-09-03 17:20:02 UTC

Contents

axe-kerasnip_model_fit	2
compile_keras_grid	3
create_keras_functional_spec	5
create_keras_sequential_spec	8
extract_keras_history	11
extract_keras_model	12
extract_valid_grid	13
inform_errors	14
inp_spec	16
kerasnip_add_tailor	18
kerasnip_output_view	19
kerasnip_step_truth	20
kerasnip_step_view	20
keras_evaluate	21
register_keras_loss	23
register_keras_metric	24
register_keras_optimizer	25
remove_keras_spec	26
step_collapse	27
step_lead	29
step_sequence	30
Index	33

axe-kerasnip_model_fit

Butcher axe methods for kerasnip_model_fit

Description

These methods allow `butcher::butcher()` to reduce the memory footprint of fitted `kerasnip` model objects. The Keras model itself (stored as raw bytes in `$fit$keras_bytes`) is always preserved so that `predict()` continues to work after butchering.

The main saving comes from `axe_data()`, which removes the training history object (`$fit$history`). For long training runs this can be several MB.

Usage

```
## S3 method for class 'kerasnip_model_fit'
axe_data(x, verbose = FALSE, ...)
```

```
## S3 method for class 'kerasnip_model_fit'
axe_env(x, verbose = FALSE, ...)
```

```
## S3 method for class 'kerasnip_model_fit'
axe_call(x, verbose = FALSE, ...)
```

```
## S3 method for class 'kerasnip_model_fit'
axe_ctrl(x, verbose = FALSE, ...)

## S3 method for class 'kerasnip_model_fit'
axe_fitted(x, verbose = FALSE, ...)
```

Arguments

x	A kerasnip_model_fit object.
verbose	Logical. Print information about memory released and disabled functions. Default is FALSE.
...	Not used.

Value

An axed kerasnip_model_fit object with the butcher_kerasnip_model_fit class prepended.

compile_keras_grid	<i>Compile Keras Models Over a Grid of Hyperparameters</i>
--------------------	--

Description

Pre-compiles Keras models for each hyperparameter combination in a grid.

This function is a powerful debugging tool to use before running a full `tune::tune_grid()`. It allows you to quickly validate multiple model architectures, ensuring they can be successfully built and compiled without the time-consuming process of actually fitting them. It helps catch common errors like incompatible layer shapes or invalid argument values early.

Usage

```
compile_keras_grid(spec, grid, x, y)
```

Arguments

spec	A parsnip model specification created by <code>create_keras_sequential_spec()</code> or <code>create_keras_functional_spec()</code> .
grid	A tibble or data.frame containing the grid of hyperparameters to evaluate. Each row represents a unique model architecture to be compiled. Must have at least one row. To build the model once using only the arguments already set on spec (e.g. for architecture inspection), pass <code>tibble::tibble(.rows = 1L)</code> .
x	A data frame or matrix of predictors. This is used to infer the input_shape for the Keras model.
y	A vector or factor of outcomes. This is used to infer the output shape and the default loss function for the Keras model.

Details

Compile and Validate Keras Model Architectures

The function iterates through each row of the provided `grid`. For each hyperparameter combination, it attempts to build and compile the Keras model defined by the spec. The process is wrapped in a try-catch block to gracefully handle and report any errors that occur during model instantiation or compilation.

The output is a tibble that mirrors the input `grid`, with additional columns containing the compiled model object or the error message, making it easy to inspect which architectures are valid.

Value

A tibble with the following columns:

- Columns from the input `grid`.
- `compiled_model`: A list-column containing the compiled Keras model objects. If compilation failed, the element will be `NULL`.
- `error`: A list-column containing `NA` for successes or a character string with the error message for failures.

Examples

```
if (requireNamespace("keras3", quietly = TRUE)) {
  library(keras3)
  library(parsnip)
  library(dials)

  # 1. Define layer blocks
  input_block <- function(model, input_shape) {
    keras_model_sequential(input_shape = input_shape)
  }
  hidden_block <- function(model, units = 32) {
    model |> layer_dense(units = units, activation = "relu")
  }
  output_block <- function(model, num_classes) {
    model |> layer_dense(units = num_classes, activation = "softmax")
  }

  # 2. Define a kerasnip model specification
  create_keras_sequential_spec(
    model_name = "my_mlp_grid",
    layer_blocks = list(
      input = input_block,
      hidden = hidden_block,
      output = output_block
    ),
    mode = "classification"
  )

  mlp_spec <- my_mlp_grid(
    hidden_units = tune(),
```

```

    compile_loss = "categorical_crossentropy",
    compile_optimizer = "adam"
  )

  # 3. Create a hyperparameter grid
  # Include an invalid value (-10) to demonstrate error handling
  param_grid <- tibble::tibble(
    hidden_units = c(32, 64, -10)
  )

  # 4. Prepare dummy data
  x_train <- matrix(rnorm(100 * 10), ncol = 10)
  y_train <- factor(sample(0:1, 100, replace = TRUE))

  # 5. Compile models over the grid
  compiled_grid <- compile_keras_grid(
    spec = mlp_spec,
    grid = param_grid,
    x = x_train,
    y = y_train
  )

  print(compiled_grid)
  remove_keras_spec("my_mlp_grid")

  # 6. Inspect the results
  # The row with `hidden_units = -10` will show an error.
}

```

```
create_keras_functional_spec
```

Create a Custom Keras Functional API Model Specification for Tidy-models

Description

This function acts as a factory to generate a new parsnip model specification based on user-defined blocks of Keras layers using the Functional API. This allows for creating complex, tunable architectures with non-linear topologies that integrate seamlessly with the tidymodels ecosystem.

Usage

```

create_keras_functional_spec(
  model_name,
  layer_blocks,
  mode = c("regression", "classification"),
  ...,
  env = parent.frame()
)

```

Arguments

model_name	A character string for the name of the new model specification function (e.g., "custom_resnet"). This should be a valid R function name.
layer_blocks	A named list of functions where each function defines a "block" (a node) in the model graph. The list names are crucial as they define the names of the nodes. The arguments of each function define how the nodes are connected. See the "Model Graph Connectivity" section for details.
mode	A character string, either "regression" or "classification".
...	Reserved for future use. Currently not used.
env	The environment in which to create the new model specification function and its associated update() method. Defaults to the calling environment (parent.frame()).

Details

This function generates all the boilerplate needed to create a custom, tunable parsnip model specification that uses the Keras Functional API. This is ideal for models with complex, non-linear topologies, such as networks with multiple inputs/outputs or residual connections.

The function inspects the arguments of your layer_blocks functions and makes them available as tunable parameters in the generated model specification, prefixed with the block's name (e.g., dense_units). Common training parameters such as fit_epochs and learn_rate are also added.

Value

Invisibly returns NULL. Its primary side effect is to create a new model specification function (e.g., custom_resnet()) in the specified environment and register the model with parsnip so it can be used within the tidymodels framework.

Model Graph Connectivity

kerasnip builds the model's directed acyclic graph by inspecting the arguments of each function in the layer_blocks list. The connection logic is as follows:

1. The **names of the elements** in the layer_blocks list define the names of the nodes in your graph (e.g., main_input, dense_path, output).
2. The **names of the arguments** in each block function specify its inputs. A block function like `my_block <- function(input_a, input_b, ...)` declares that it needs input from the nodes named input_a and input_b. kerasnip will automatically supply the output tensors from those nodes when calling my_block.

There are two special requirements:

- **Input Block:** The first block in the list is treated as the input node. Its function should not take other blocks as input, but it can have an input_shape argument, which is supplied automatically during fitting.
- **Output Block:** Exactly one block must be named "output". The tensor returned by this block is used as the final output of the Keras model.

A key feature is the automatic creation of `num_{block_name}` arguments (e.g., `num_dense_path`). This allows you to control how many times a block is repeated, making it easy to tune the depth of your network. A block can only be repeated if it has exactly one input from another block in the graph.

The new model specification function and its `update()` method are created in the environment specified by the `env` argument.

Saving and Reloading Models

To save a fitted workflow and reload it in a new R session, use `bundle::bundle()` before saving — this is required to preserve the Keras model weights:

```
library(bundle)
bundled <- bundle(fitted_workflow)
saveRDS(bundled, "model.rds")

# New session:
library(kerasnip); library(bundle)
fitted_workflow <- unbundle(readRDS("model.rds"))
predict(fitted_workflow, new_data = test_data) # works
```

Plain `saveRDS()` without `bundle()` does not preserve Keras weights, but `predict()` will still auto-register the `parsnip` model type from metadata stored on the spec.

See Also

[remove_keras_spec\(\)](#), [parsnip::new_model_spec\(\)](#), [create_keras_sequential_spec\(\)](#)

Examples

```
if (requireNamespace("keras3", quietly = TRUE)) {
  library(keras3)
  library(parsnip)

  # 1. Define block functions. These are the building blocks of our model.
  # An input block that receives the data's shape automatically.
  input_block <- function(input_shape) layer_input(shape = input_shape)

  # A dense block with a tunable `units` parameter.
  dense_block <- function(tensor, units) {
    tensor |> layer_dense(units = units, activation = "relu")
  }

  # A block that adds two tensors together (for the residual connection).
  add_block <- function(input_a, input_b) layer_add(list(input_a, input_b))

  # An output block for regression.
  output_block_reg <- function(tensor) layer_dense(tensor, units = 1)

  # 2. Create the spec. The `layer_blocks` list defines the graph.
```

```

create_keras_functional_spec(
  model_name = "my_resnet_spec",
  layer_blocks = list(
    # The names of list elements are the node names.
    main_input = input_block,

    # The argument `main_input` connects this block to the input node.
    dense_path = function(main_input, units = 32) {
      dense_block(main_input, units)
    },

    # This block's arguments connect it to the original input AND the dense
    # layer.
    add_residual = function(main_input, dense_path) {
      add_block(main_input, dense_path)
    },

    # This block must be named 'output'. It connects to the residual add
    # layer.
    output = function(add_residual) output_block_reg(add_residual)
  ),
  mode = "regression"
)

# 3. Use the newly created specification function!
# The `dense_path_units` argument was created automatically.
model_spec <- my_resnet_spec(dense_path_units = 64, fit_epochs = 10)

# You could also tune the number of dense layers since it has a single
# input:
# model_spec <- my_resnet_spec(num_dense_path = 2, dense_path_units = 32)

print(model_spec)
remove_keras_spec("my_resnet_spec")
# tune::tunable(model_spec)
}

```

```
create_keras_sequential_spec
```

Create a Custom Keras Sequential Model Specification for Tidymodels

Description

This function acts as a factory to generate a new parsnip model specification based on user-defined blocks of Keras layers using the Sequential API. This is the ideal choice for creating models that are a simple, linear stack of layers. For models with complex, non-linear topologies, see [create_keras_functional_spec\(\)](#).

Usage

```
create_keras_sequential_spec(
  model_name,
  layer_blocks,
  mode = c("regression", "classification"),
  ...,
  env = parent.frame()
)
```

Arguments

model_name	A character string for the name of the new model specification function (e.g., "custom_cnn"). This should be a valid R function name.
layer_blocks	A named, ordered list of functions. Each function defines a "block" of Keras layers. The function must take a Keras model object as its first argument and return the modified model. Other arguments to the function will become tunable parameters in the final model specification.
mode	A character string, either "regression" or "classification".
...	Reserved for future use. Currently not used.
env	The environment in which to create the new model specification function and its associated update() method. Defaults to the calling environment (parent.frame()).

Details

This function generates all the boilerplate needed to create a custom, tunable parsnip model specification that uses the Keras Sequential API.

The function inspects the arguments of your layer_blocks functions (ignoring special arguments like input_shape and num_classes) and makes them available as arguments in the generated model specification, prefixed with the block's name (e.g., dense_units).

The new model specification function and its update() method are created in the environment specified by the env argument.

Value

Invisibly returns NULL. Its primary side effect is to create a new model specification function (e.g., my_mlp()) in the specified environment and register the model with parsnip so it can be used within the tidymodels framework.

Model Architecture (Sequential API)

kerasnip builds the model by applying the functions in layer_blocks in the order they are provided. Each function receives the Keras model built by the previous function and returns a modified version.

1. The **first block** must initialize the model (e.g., with keras_model_sequential()). It can accept an input_shape argument, which kerasnip will provide automatically during fitting.
2. **Subsequent blocks** add layers to the model.

3. The **final block** should add the output layer. For classification, it can accept a `num_classes` argument, which is provided automatically.

A key feature of this function is the automatic creation of `num_{block_name}` arguments (e.g., `num_hidden`). This allows you to control how many times each block is repeated, making it easy to tune the depth of your network.

Saving and Reloading Models

To save a fitted workflow and reload it in a new R session, use `bundle::bundle()` before saving — this is required to preserve the Keras model weights:

```
library(bundle)
bundled <- bundle(fitted_workflow)
saveRDS(bundled, "model.rds")

# New session:
library(kerasnip); library(bundle)
fitted_workflow <- unbundle(readRDS("model.rds"))
predict(fitted_workflow, new_data = test_data) # works
```

Plain `saveRDS()` without `bundle()` does not preserve Keras weights, but `predict()` will still auto-register the `parsnip` model type from metadata stored on the spec.

See Also

[remove_keras_spec\(\)](#), [parsnip::new_model_spec\(\)](#), [create_keras_functional_spec\(\)](#)

Examples

```
if (requireNamespace("keras3", quietly = TRUE)) {
  library(keras3)
  library(parsnip)
  library(dials)

  # 1. Define layer blocks for a complete model.
  # The first block must initialize the model. `input_shape` is passed
  # automatically.
  input_block <- function(model, input_shape) {
    keras_model_sequential(input_shape = input_shape)
  }
  # A block for hidden layers. `units` will become a tunable parameter.
  hidden_block <- function(model, units = 32) {
    model |> layer_dense(units = units, activation = "relu")
  }

  # The output block. `num_classes` is passed automatically for classification.
  output_block <- function(model, num_classes) {
    model |> layer_dense(units = num_classes, activation = "softmax")
  }
```

```
# 2. Create the spec, providing blocks in the correct order.
create_keras_sequential_spec(
  model_name = "my_mlp_seq_spec",
  layer_blocks = list(
    input = input_block,
    hidden = hidden_block,
    output = output_block
  ),
  mode = "classification"
)

# 3. Use the newly created specification function!
# Note the new arguments `num_hidden` and `hidden_units`.
model_spec <- my_mlp_seq_spec(
  num_hidden = 2,
  hidden_units = 64,
  fit_epochs = 10,
  learn_rate = 0.01
)

print(model_spec)
remove_keras_spec("my_mlp_seq_spec")
}
```

extract_keras_history *Extract Keras Training History*

Description

Extracts and returns the training history from a `parsnip model_fit` object created by `kerasnip`.

Usage

```
extract_keras_history(object)
```

Arguments

`object` A `model_fit` object produced by a `kerasnip` specification.

Details

Extract Keras Training History

The history object contains the metrics recorded during model training, such as loss and accuracy, for each epoch. This is highly useful for visualizing the training process and diagnosing issues like overfitting. The returned object can be plotted directly.

Value

A `keras_training_history` object. You can call `plot()` on this object to visualize the learning curves.

See Also

`keras_evaluate`, `extract_keras_model`

<code>extract_keras_model</code>	<i>Extract Keras Model from a Fitted Kerasnip Object</i>
----------------------------------	--

Description

Extracts and returns the underlying Keras model object from a `parsnip model_fit` object created by `kerasnip`.

Usage

```
extract_keras_model(object)
```

Arguments

<code>object</code>	A <code>model_fit</code> object produced by a <code>kerasnip</code> specification.
---------------------	--

Details

Extract the Raw Keras Model from a Kerasnip Fit

This is useful when you need to work directly with the Keras model object for tasks like inspecting layer weights, creating custom plots, or passing it to other Keras-specific functions.

Value

The raw Keras model object (`keras_model`).

See Also

`keras_evaluate`, `extract_keras_history`

extract_valid_grid	<i>Extract Valid Grid from Compilation Results</i>
--------------------	--

Description

This helper function filters the results from `compile_keras_grid()` to return a new hyperparameter grid containing only the combinations that compiled successfully.

Usage

```
extract_valid_grid(compiled_grid)
```

Arguments

`compiled_grid` A tibble, the result of a call to `compile_keras_grid()`.

Details

Filter a Grid to Only Valid Hyperparameter Sets

After running `compile_keras_grid()`, you can use this function to remove problematic hyperparameter combinations before proceeding to the full `tune::tune_grid()`.

Value

A tibble containing the subset of the original grid that resulted in a successful model compilation. The `compiled_model` and `error` columns are removed, leaving a clean grid ready for tuning.

Examples

```
if (requireNamespace("keras3", quietly = TRUE)) {
  library(keras3)
  library(parsnip)
  library(dials)

  # 1. Define layer blocks
  input_block <- function(model, input_shape) {
    keras_model_sequential(input_shape = input_shape)
  }
  hidden_block <- function(model, units = 32) {
    model |> layer_dense(units = units, activation = "relu")
  }
  output_block <- function(model, num_classes) {
    model |> layer_dense(units = num_classes, activation = "softmax")
  }

  # 2. Define a kerasnip model specification
  create_keras_sequential_spec(
    model_name = "my_mlp_grid_2",
    layer_blocks = list(
```

```

        input = input_block,
        hidden = hidden_block,
        output = output_block
      ),
      mode = "classification"
    )

    mlp_spec <- my_mlp_grid_2(
      hidden_units = tune(),
      compile_loss = "categorical_crossentropy",
      compile_optimizer = "adam"
    )

    # 3. Create a hyperparameter grid
    param_grid <- tibble::tibble(
      hidden_units = c(32, 64, -10)
    )

    # 4. Prepare dummy data
    x_train <- matrix(rnorm(100 * 10), ncol = 10)
    y_train <- factor(sample(0:1, 100, replace = TRUE))

    # 5. Compile models over the grid
    compiled_grid <- compile_keras_grid(
      spec = mlp_spec,
      grid = param_grid,
      x = x_train,
      y = y_train
    )

    # 6. Extract the valid grid
    valid_grid <- extract_valid_grid(compiled_grid)
    print(valid_grid)
    remove_keras_spec("my_mlp_grid_2")
  }

```

inform_errors

Inform About Compilation Errors

Description

This helper function inspects the results from `compile_keras_grid()` and prints a formatted, easy-to-read summary of any compilation errors that occurred.

Usage

```
inform_errors(compiled_grid, n = 10)
```

Arguments

`compiled_grid` A tibble, the result of a call to `compile_keras_grid()`.

`n` A single integer for the maximum number of distinct errors to display in detail.

Details

Display a Summary of Compilation Errors

This is most useful for interactive debugging of complex tuning grids where some hyperparameter combinations may lead to invalid Keras models.

Value

Invisibly returns the input `compiled_grid`. Called for its side effect of printing a summary to the console.

Examples

```
if (requireNamespace("keras3", quietly = TRUE)) {
  library(keras3)
  library(parsnip)
  library(dials)

  # 1. Define layer blocks
  input_block <- function(model, input_shape) {
    keras_model_sequential(input_shape = input_shape)
  }
  hidden_block <- function(model, units = 32) {
    model |> layer_dense(units = units, activation = "relu")
  }
  output_block <- function(model, num_classes) {
    model |> layer_dense(units = num_classes, activation = "softmax")
  }

  # 2. Define a kerasnip model specification
  create_keras_sequential_spec(
    model_name = "my_mlp_grid_3",
    layer_blocks = list(
      input = input_block,
      hidden = hidden_block,
      output = output_block
    ),
    mode = "classification"
  )

  mlp_spec <- my_mlp_grid_3(
    hidden_units = tune(),
    compile_loss = "categorical_crossentropy",
    compile_optimizer = "adam"
  )

  # 3. Create a hyperparameter grid
```

```

param_grid <- tibble::tibble(
  hidden_units = c(32, 64, -10)
)

# 4. Prepare dummy data
x_train <- matrix(rnorm(100 * 10), ncol = 10)
y_train <- factor(sample(0:1, 100, replace = TRUE))

# 5. Compile models over the grid
compiled_grid <- compile_keras_grid(
  spec = mlp_spec,
  grid = param_grid,
  x = x_train,
  y = y_train
)

# 6. Inform about errors
inform_errors(compiled_grid)
remove_keras_spec("my_mlp_grid_3")
}

```

inp_spec

*Remap Layer Block Arguments for Model Specification***Description**

Creates a wrapper function around a Keras layer block to rename its arguments. This is a powerful helper for defining the layer_blocks in `create_keras_functional_spec()` and `create_keras_sequential_spec()`, allowing you to connect reusable blocks into a model graph without writing verbose anonymous functions.

Usage

```
inp_spec(block, input_map)
```

Arguments

block	A function that defines a Keras layer or a set of layers. The first arguments should be the input tensor(s).
input_map	A single character string or a named character vector that specifies how to rename/remap the arguments of block.

Details

`inp_spec()` makes your model definitions cleaner and more readable. It handles the metaprogramming required to create a new function with the correct argument names, while preserving the original block's hyperparameters and their default values.

The function supports two modes of operation based on `input_map`:

1. **Single Input Renaming:** If `input_map` is a single character string, the wrapper function renames the *first* argument of the block function to the provided string. This is the common case for blocks that take a single tensor input.
2. **Multiple Input Mapping:** If `input_map` is a named character vector, the **names must match the argument names of** block and each value must be the name of an upstream layer block whose output should be fed into that argument. This orientation matches the syntax (e.g., `c(numeric = "processed_numerical")`). This is used for blocks with multiple inputs, like a concatenation layer.

Note: Prior releases accepted the opposite orientation (`c(processed_numerical = "numeric")`). Existing code written in that style must flip the names/values when upgrading to this version.

Value

A new function (a closure) that wraps the block function with renamed arguments, ready to be used in a `layer_blocks` list.

Examples

```
# --- Example Blocks ---
# A standard dense block with one input tensor and one hyperparameter.
dense_block <- function(tensor, units = 16) {
  tensor |> keras3::layer_dense(units = units, activation = "relu")
}

# A block that takes two tensors as input.
concat_block <- function(input_a, input_b) {
  keras3::layer_concatenate(list(input_a, input_b))
}

# An output block with one input.
output_block <- function(tensor) {
  tensor |> keras3::layer_dense(units = 1)
}

# --- Usage ---
layer_blocks <- list(
  main_input = keras3::layer_input,
  path_a = inp_spec(dense_block, "main_input"),
  path_b = inp_spec(dense_block, "main_input"),
  concatenated = inp_spec(
    concat_block,
    c(input_a = "path_a", input_b = "path_b")
  ),
  output = inp_spec(output_block, "concatenated")
)
```

kerasnip_add_tailor	<i>Attach a tailor Post-Processor to One Output or Step of a Multi-Output or Multistep Workflow</i>
---------------------	---

Description

`workflows::add_tailor()` cannot be used on a kerasnip multi-output or multistep workflow: `tailor::fit()` selects outcome/estimate via `[[`, which requires exactly one, flat, numeric column, and both a multi-output recipe (`output_1 + output_2 ~ .`) and a multistep model's nested `.pred` list-column violate that (see `vignette("multi_output_postprocessing")`). `kerasnip_add_tailor()` is a kerasnip-owned analogue that attaches a tailor post-processor to a single named output or forecast step, using `kerasnip_output_view()` or `kerasnip_step_view()` internally.

Usage

```
kerasnip_add_tailor(x, tailor, output = NULL, step = NULL, var = NULL)
```

Arguments

<code>x</code>	An unfitted workflow whose model has more than one outcome (multi-output) or is a multistep forecasting model.
<code>tailor</code>	A <code>tailor::tailor()</code> specification.
<code>output</code>	A string, the name of the outcome column to post-process (multi-output models).
<code>step</code>	An integer, the forecast step to post-process (multistep models).
<code>var</code>	A string, the forecasted variable to post-process; only needed with <code>step</code> if the model forecasts more than one variable.

Details

At `fit()` time, the underlying model is trained as usual; the relevant view is then used to fit the tailor against that output's/step's predictions (on `data_calibration` if supplied, otherwise on `data`, mirroring `workflows::add_tailor()`'s data-usage convention). At `predict()` time, the full prediction is generated, the target output's/step's value(s) are replaced with the tailor-adjusted values, and everything else (other outputs; other steps in the same nested tibble) is left untouched.

Exactly one of `output` or `step` must be supplied: `output` for a multi-output model, `step` (and `var`, if more than one variable is forecast) for a multistep model.

Value

A `kerasnip_tailored_workflow`, to be trained with `fit()`.

Examples

```
## Not run:
tlr <- tailor::tailor() |> tailor::adjust_numeric_calibration()

# multi-output
tailored_wf <- kerasnip_add_tailor(wf, tlr, output = "output_1")

# multistep
tailored_wf <- kerasnip_add_tailor(wf, tlr, step = 2)

fit_obj <- fit(tailored_wf, data = train_data, data_calibration = cal_data)
predict(fit_obj, new_data = test_data)

## End(Not run)
```

kerasnip_output_view *View a Single Output of a Multi-Output kerasnip Fit*

Description

`tailor` and `probably` are built around models with a single outcome column and a single `.pred/.pred_class` prediction column. A `kerasnip` multi-output model (e.g. a recipe with `output_1 + output_2 ~ .`) instead produces `.pred_output_1`, `.pred_output_2`, ... columns from multiple truth columns in one fit — the standard `parsnip::maybe_multivariate()` shape, but one `tailor::fit()/workflows::add_tailor()` call cannot consume it (it selects outcome/estimate via `[[`, which requires exactly one column).

`kerasnip_output_view()` wraps a fitted multi-output workflow together with one output name, presenting it as if it were an ordinary single-output fit: `predict()` returns standard `.pred/.pred_class` / `.pred_<level>` columns for that output alone, letting you calibrate or post-process each output separately with the usual `tailor/probably` calls (see `vignette("multi_output_postprocessing")`).

Usage

```
kerasnip_output_view(x, output)
```

Arguments

<code>x</code>	A fitted (trained) workflow whose model has more than one outcome column.
<code>output</code>	A string, the name of the outcome column to view.

Value

A `kerasnip_output_view` object.

Examples

```
## Not run:
fit_obj <- fit(wf, data = train_data) # wf predicts output_1 and output_2
view_1 <- kerasnip_output_view(fit_obj, "output_1")
predict(view_1, new_data = test_data) # -> a single `.pred` column

## End(Not run)
```

kerasnip_step_truth	<i>Recover Truth Values for a Multistep Forecast Step</i>
---------------------	---

Description

A multistep model's per-step outcome columns (e.g. `lead_2_value`) are engineered by `step_lead()` from a single raw column, so they are not present in a user's raw data the way genuine multi-output columns are. This re-bakes the fitted recipe on `new_data` to recover the actual future value at `kerasnip_step_view()`'s step, for calibration/interval use. Rows too close to the end of `new_data` for the lead to be computed return NA (dropped automatically by calibration routines that call `sort()/stats::complete.cases()` on the result).

Usage

```
kerasnip_step_truth(view, new_data)
```

Arguments

<code>view</code>	A <code>kerasnip_step_view</code> .
<code>new_data</code>	A data frame of raw predictors (and the original outcome column <code>step_lead()</code> was applied to).

Value

A numeric vector, one truth value per row of `new_data`.

kerasnip_step_view	<i>View a Single Forecast Step of a Multistep kerasnip Fit</i>
--------------------	--

Description

A `kerasnip` multistep (vector-valued) regression model returns a nested `.pred` list-column: one inner tibble per row, with a `.step` column plus one prediction column per forecasted variable. `tailor/probably` expect a single flat numeric `.pred` column instead — `tailor::check_variable_type()` requires `is.numeric()` on the outcome/estimate columns, which a list-column fails outright.

`kerasnip_step_view()` wraps a fitted multistep workflow together with one forecast step (and, if more than one variable is forecast, which variable), presenting it as an ordinary single-output fit: `predict()` returns a flat `.pred` column for that step alone.

Usage

```
kerasnip_step_view(x, step, var = NULL)
```

Arguments

x	A fitted (trained) workflow whose model is a multistep regression model (see <code>create_keras_sequential_spec()/create_keras_functional_spec()</code> with a vector-valued output).
step	An integer, the forecast step to view.
var	A string, the forecasted variable to view. Required only if the model forecasts more than one variable; inferred otherwise.

Details

Unlike `kerasnip_output_view()`, a multistep model's per-step outcome columns (e.g. `lead_1_value`) are *recipe-engineered* from a single raw column via `step_lead()` — they are not present in a user's raw data the way genuine multi-output columns are. `kerasnip_step_truth()` recovers the true future value at a given step by re-baking the fitted recipe on raw data, which is what `int_conformal_split()` uses internally for this class.

probably: `int_conformal_full()` is also supported (see `int_conformal_full.kerasnip_step_view()`), with a materially different design from `kerasnip_output_view()`'s: refitting for a candidate value at this step means substituting it into the single *raw* column `step_lead()` derives every step's truth from, which shifts every nearby row's target too. It is only supported when `step_lead()` and `step_sequence()` share a single source column, matching all of kerasnip's own multistep examples.

Value

A `kerasnip_step_view` object.

Examples

```
## Not run:
fit_obj <- fit(wf, data = train_data) # a multistep forecasting workflow
step_2 <- kerasnip_step_view(fit_obj, step = 2)
predict(step_2, new_data = test_data) # -> a single `.pred` column

## End(Not run)
```

keras_evaluate

Evaluate a Kerasnip Model

Description

This function provides an `keras_evaluate()` method for `model_fit` objects created by kerasnip. It preprocesses the new data into the format expected by Keras and then calls `keras3::evaluate()` on the underlying model to compute the loss and any other metrics.

Usage

```
keras_evaluate(object, x, y = NULL, ...)
```

Arguments

<code>object</code>	A <code>model_fit</code> object produced by a kerasnip specification.
<code>x</code>	A data frame or matrix of new predictor data.
<code>y</code>	A vector or data frame of new outcome data corresponding to <code>x</code> .
<code>...</code>	Additional arguments passed on to <code>keras3::evaluate()</code> (e.g., <code>batch_size</code>).

Details

Evaluate a Fitted Kerasnip Model on New Data

Value

A named list containing the evaluation results (e.g., loss, accuracy). The names are determined by the metrics the model was compiled with.

Examples

```
if (requireNamespace("keras3", quietly = TRUE)) {
  library(keras3)
  library(parsnip)

  # 1. Define layer blocks
  input_block <- function(model, input_shape) {
    keras_model_sequential(input_shape = input_shape)
  }
  hidden_block <- function(model, units = 32) {
    model |> layer_dense(units = units, activation = "relu")
  }
  output_block <- function(model, num_classes) {
    model |> layer_dense(units = num_classes, activation = "softmax")
  }

  # 2. Define and fit a model ----
  create_keras_sequential_spec(
    model_name = "my_mlp_tools",
    layer_blocks = list(
      input = input_block,
      hidden = hidden_block,
      output = output_block
    ),
    mode = "classification"
  )

  mlp_spec <- my_mlp_tools(
    hidden_units = 32,
    compile_loss = "categorical_crossentropy",
```

```

    compile_optimizer = "adam",
    compile_metrics = "accuracy",
    fit_epochs = 5
) |> set_engine("keras")

x_train <- matrix(rnorm(100 * 10), ncol = 10)
y_train <- factor(sample(0:1, 100, replace = TRUE))
train_df <- data.frame(x = I(x_train), y = y_train)

fitted_mlp <- fit(mlp_spec, y ~ x, data = train_df)

# 3. Evaluate the model on new data ----
x_test <- matrix(rnorm(50 * 10), ncol = 10)
y_test <- factor(sample(0:1, 50, replace = TRUE))

eval_metrics <- keras_evaluate(fitted_mlp, x_test, y_test)
print(eval_metrics)

# 4. Extract the Keras model object ----
keras_model <- extract_keras_model(fitted_mlp)
summary(keras_model)

# 5. Extract the training history ----
history <- extract_keras_history(fitted_mlp)
plot(history)
remove_keras_spec("my_mlp_tools")
}

```

register_keras_loss	<i>Register a Custom Keras Loss</i>
---------------------	-------------------------------------

Description

Allows users to register a custom loss function so it can be used by name within kerasnip model specifications and tuned with dials.

Usage

```
register_keras_loss(name, loss_fn)
```

Arguments

name	The name to register the loss under (character).
loss_fn	The loss function.

Details

Registered losses are stored in an internal environment. When a model is compiled, kerasnip will first check this internal registry for a loss matching the provided name before checking the keras3 package.

Value

No return value, called for side effects.

See Also

[register_keras_optimizer\(\)](#), [register_keras_metric\(\)](#)

`register_keras_metric` *Register a Custom Keras Metric*

Description

Allows users to register a custom metric function so it can be used by name within kerasnip model specifications.

Usage

```
register_keras_metric(name, metric_fn)
```

Arguments

<code>name</code>	The name to register the metric under (character).
<code>metric_fn</code>	The metric function.

Details

Registered metrics are stored in an internal environment. When a model is compiled, kerasnip will first check this internal registry for a metric matching the provided name before checking the keras3 package.

Value

No return value, called for side effects.

See Also

[register_keras_optimizer\(\)](#), [register_keras_loss\(\)](#)

`register_keras_optimizer`*Register a Custom Keras Optimizer*

Description

Allows users to register a custom optimizer function so it can be used by name within kerasnip model specifications and tuned with dials.

Usage

```
register_keras_optimizer(name, optimizer_fn)
```

Arguments

<code>name</code>	The name to register the optimizer under (character).
<code>optimizer_fn</code>	The optimizer function. It should return a Keras optimizer object.

Details

Registered optimizers are stored in an internal environment. When a model is compiled, kerasnip will first check this internal registry for an optimizer matching the provided name before checking the keras3 package.

The `optimizer_fn` can be a simple function or a partially applied function using `purrr::partial()`. This is useful for creating versions of Keras optimizers with specific settings.

Value

No return value, called for side effects.

See Also

[register_keras_loss\(\)](#), [register_keras_metric\(\)](#)

Examples

```
if (requireNamespace("keras3", quietly = TRUE)) {  
  # Register a custom version of Adam with a different default beta_1  
  my_adam <- purrr::partial(keras3::optimizer_adam, beta_1 = 0.8)  
  register_keras_optimizer("my_adam", my_adam)  
  
  # Now "my_adam" can be used as a string in a model spec, e.g.,  
  # my_model_spec(compile_optimizer = "my_adam")  
}
```

remove_keras_spec	<i>Remove a Keras Model Specification and its Registrations</i>
-------------------	---

Description

This function completely removes a model specification that was previously created by [create_keras_sequential_spec\(\)](#) or [create_keras_functional_spec\(\)](#). It cleans up both the function in the user's environment and all associated registrations within the parsnip package.

Usage

```
remove_keras_spec(model_name, env = parent.frame())
```

Arguments

model_name	A character string giving the name of the model specification function to remove (e.g., "my_mlp").
env	The environment from which to remove the function and its update() method. Defaults to the calling environment (parent.frame()).

Details

This function is essential for cleanly unloading a dynamically created model. It performs three main actions:

1. It removes the model specification function (e.g., my_mlp()) and its corresponding update() method from the specified environment.
2. It searches parsnip's internal model environment for all objects whose names start with the model_name and removes them. This purges the fit methods, argument definitions, and other registrations.
3. It removes the model's name from parsnip's master list of models.

This function uses the un-exported get_model_env() to perform the cleanup.

Value

Invisibly returns TRUE after attempting to remove the objects.

See Also

[create_keras_sequential_spec\(\)](#), [create_keras_functional_spec\(\)](#)

Examples

```
if (requireNamespace("keras3", quietly = TRUE)) {
  # First, create a dummy spec
  input_block <- function(model, input_shape) {
    keras3::keras_model_sequential(input_shape = input_shape)
  }
  dense_block <- function(model, units = 16) {
    model |> keras3::layer_dense(units = units)
  }
  create_keras_sequential_spec(
    "my_temp_model",
    list(
      input = input_block,
      dense = dense_block
    ),
    "regression"
  )

  # Check it exists in the environment and in parsnip
  exists("my_temp_model")
  "my_temp_model" %in% parsnip::show_engines("my_temp_model")$model

  # Now remove it
  remove_keras_spec("my_temp_model")

  # Check it's gone
  !exists("my_temp_model")
  !model_exists("my_temp_model")
}
```

step_collapse

Collapse Predictors into a single list-column

Description

`step_collapse()` creates a *specification* of a recipe step that will convert a group of predictors into a single list-column. This is useful for custom models that need the predictors in a different format.

Usage

```
step_collapse(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  columns = NULL,
  new_col = "predictor_matrix",
```

```

    skip = FALSE,
    id = recipes::rand_id("collapse")
  )

```

Arguments

<code>recipe</code>	A recipe object. The step will be added to the sequence of operations for this recipe.
<code>...</code>	One or more selector functions to choose which variables are affected by the step. See <code>[selections()]</code> for more details. For the <code>tidy</code> method, these are not currently used.
<code>role</code>	For model terms created by this step, what analysis role should they be assigned?. By default, the new columns are used as predictors.
<code>trained</code>	A logical to indicate if the quantities for preprocessing have been estimated.
<code>columns</code>	A character string of the selected variable names. This is <code>NULL</code> until the step is trained by <code>[prep.recipe()]</code> .
<code>new_col</code>	A character string for the name of the new list-column. The default is "predictor_matrix".
<code>skip</code>	A logical. Should the step be skipped when the recipe is baked by <code>[bake.recipe()]</code> ? While all operations are baked when <code>prep</code> is run, skipping when <code>bake</code> is run may be other times when it is desirable to skip a processing step.
<code>id</code>	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of existing steps (if any). For the `tidy` method, a tibble with columns `terms` (the selected column names), `value` (the name of the destination list-column), and `id` (the step identifier).

Examples

```

library(recipes)

# 2 predictors
dat <- data.frame(
  x1 = 1:10,
  x2 = 11:20,
  y = 1:10
)

rec <- recipe(y ~ ., data = dat) %>%
  step_collapse(x1, x2, new_col = "pred") %>%
  prep()

bake(rec, new_data = NULL)

```

step_lead

*Create a Lead Predictor***Description**

`step_lead()` creates a *specification* of a recipe step that will create one or more new columns of data that are leading (i.e. future) values of existing columns. This is the target-side companion to `[recipes::step_lag()]` (which only supports positive lag/past values, not lead/future values) and is intended for building multi-step-ahead forecasting targets, e.g. `step_lead(y, lead = 1:6)` produces the next six values of `y` as separate columns, one per row.

Usage

```
step_lead(
  recipe,
  ...,
  lead = 1,
  prefix = "lead_",
  default = NA,
  role = "outcome",
  trained = FALSE,
  columns = NULL,
  keep_original_cols = TRUE,
  skip = FALSE,
  id = recipes::rand_id("lead")
)
```

Arguments

<code>recipe</code>	A recipe object. The step will be added to the sequence of operations for this recipe.
<code>...</code>	One or more selector functions to choose which variables are leading. See <code>[selections()]</code> for more details. For the tidy method, these are not currently used.
<code>lead</code>	A vector of nonnegative integers. Each value produces a leading column for each selected variable.
<code>prefix</code>	A prefix added to the leading columns names. The default naming convention is <code><prefix><lead value>_<original variable name></code> , e.g. <code>lead_1_value</code> .
<code>default</code>	Value to fill in the trailing rows that don't have a complete future window (analogous to default in <code>[recipes::step_lag()]</code>). Defaults to NA.
<code>role</code>	For model terms created by this step, what analysis role should they be assigned?. By default, the new columns are used as outcomes.
<code>trained</code>	A logical to indicate if the quantities for preprocessing have been estimated.
<code>columns</code>	A character string of the selected variable names. This is NULL until the step is trained by <code>[prep.recipe()]</code> .

keep_original_cols	A logical to keep the original variables in the output. Defaults to TRUE.
skip	A logical. Should the step be skipped when the recipe is baked by <code>[bake.recipe()]</code> ? While all operations are baked when prep is run, skipping when bake is run may be other times when it is desirable to skip a processing step.
id	A character string that is unique to this step to identify it.

Details

Combine with `[recipes::step_naomit()]` (e.g. `step_naomit(starts_with(prefix))`) to drop the trailing rows that don't have a full future window, mirroring how `[step_sequence()]`'s padding = "drop" removes rows lacking a full past window.

Value

An updated version of recipe with the new step added to the sequence of existing steps (if any). For the tidy method, a tibble with columns `terms` (the selected column names), `value` (the name of the resulting leading column), and `id` (the step identifier).

Examples

```
library(recipes)

dat <- data.frame(y = 1:10)

rec <- recipe(y ~ ., data = dat) %>%
  step_lead(y, lead = 1:2) %>%
  prep()

bake(rec, new_data = NULL)
```

step_sequence

Build a Sliding Window of Predictors for Sequence Models

Description

`step_sequence()` creates a *specification* of a recipe step that converts one or more ordered numeric predictor columns into a single list-column of (timesteps, features) matrices, one per row. This is the shape expected by recurrent layer blocks (e.g. `keras3::layer_lstm()`, `keras3::layer_gru()`) used with [create_keras_functional_spec\(\)](#) or [create_keras_sequential_spec\(\)](#).

Usage

```
step_sequence(
  recipe,
  ...,
  timesteps,
  role = "predictor",
```

```

    trained = FALSE,
    columns = NULL,
    new_col = "sequence_matrix",
    padding = c("drop", "zero"),
    skip = FALSE,
    id = recipes::rand_id("sequence")
  )

```

Arguments

<code>recipe</code>	A recipe object. The step will be added to the sequence of operations for this recipe.
<code>...</code>	One or more selector functions to choose which (already time-ordered) numeric variables are windowed. See <code>[selections()]</code> for more details. All selected columns become "features" in the resulting window. For the tidy method, these are not currently used.
<code>timesteps</code>	A single integer. The sliding window length (number of past rows, including the current one) to include in each window.
<code>role</code>	For model terms created by this step, what analysis role should they be assigned?. By default, the new column is used as a predictor.
<code>trained</code>	A logical to indicate if the quantities for preprocessing have been estimated.
<code>columns</code>	A character string of the selected variable names. This is NULL until the step is trained by <code>[prep.recipe()]</code> .
<code>new_col</code>	A character string for the name of the new list-column. The default is "sequence_matrix".
<code>padding</code>	One of "drop" (default) or "zero". Rows without a full timesteps history need special handling: "drop" removes them from the data (as <code>[recipes::step_naomit()]</code> does), while "zero" left-pads the missing history with rows of zeros so no rows are dropped.
<code>skip</code>	A logical. Should the step be skipped when the recipe is baked by <code>[bake.recipe()]</code> ? While all operations are baked when prep is run, skipping when bake is run may be other times when it is desirable to skip a processing step.
<code>id</code>	A character string that is unique to this step to identify it.

Value

An updated version of recipe with the new step added to the sequence of existing steps (if any). For the tidy method, a tibble with columns `terms` (the selected column names), `value` (the name of the destination list-column), `timesteps`, and `id` (the step identifier).

Examples

```

library(recipes)

dat <- data.frame(x1 = 1:10, x2 = 11:20, y = 1:10)

rec <- recipe(y ~ ., data = dat) %>%

```

```
step_sequence(x1, x2, timesteps = 3, new_col = "window") %>%  
  prep()  
  
bake(rec, new_data = NULL)
```

Index

axe-kerasnip_model_fit, [2](#)
axe_call.kerasnip_model_fit
 (axe-kerasnip_model_fit), [2](#)
axe_ctrl.kerasnip_model_fit
 (axe-kerasnip_model_fit), [2](#)
axe_data.kerasnip_model_fit
 (axe-kerasnip_model_fit), [2](#)
axe_env.kerasnip_model_fit
 (axe-kerasnip_model_fit), [2](#)
axe_fitted.kerasnip_model_fit
 (axe-kerasnip_model_fit), [2](#)

compile_keras_grid, [3](#)
create_keras_functional_spec, [5](#)
create_keras_functional_spec(), [8](#), [10](#),
 [16](#), [26](#), [30](#)
create_keras_sequential_spec, [8](#)
create_keras_sequential_spec(), [7](#), [16](#),
 [26](#), [30](#)

extract_keras_history, [11](#)
extract_keras_model, [12](#)
extract_valid_grid, [13](#)

inform_errors, [14](#)
inp_spec, [16](#)
int_conformal_split(), [21](#)

keras_evaluate, [21](#)
kerasnip_add_tailor, [18](#)
kerasnip_output_view, [19](#)
kerasnip_output_view(), [18](#), [21](#)
kerasnip_step_truth, [20](#)
kerasnip_step_truth(), [21](#)
kerasnip_step_view, [20](#)
kerasnip_step_view(), [18](#), [20](#)

parsnip::new_model_spec(), [7](#), [10](#)

register_keras_loss, [23](#)
register_keras_loss(), [24](#), [25](#)

register_keras_metric, [24](#)
register_keras_metric(), [24](#), [25](#)
register_keras_optimizer, [25](#)
register_keras_optimizer(), [24](#)
remove_keras_spec, [26](#)
remove_keras_spec(), [7](#), [10](#)

step_collapse, [27](#)
step_lead, [29](#)
step_sequence, [30](#)