Package ‘innsight’

November 26, 2024

Type Package

Title Get the Insights of Your Neural Network

Version 0.3.1

Description Interpretation methods for analyzing the behavior and individual predictions of modern neural networks in a three-step procedure: Converting the model, running the interpretation method, and visualizing the results. Implemented methods are, e.g., 'Connection Weights' described by Olden et al. (2004) <doi:10.1016/j.ecolmodel.2004.03.013>, layer-wise relevance propagation ('LRP') described by Bach et al. (2015) <doi:10.1371/journal.pone.0130140>, deep learning important features ('DeepLIFT') described by Shrikumar et al. (2017) <doi:10.48550/arXiv.1704.02685> and gradient-based methods like 'SmoothGrad' described by Smilkov et al. (2017) <doi:10.48550/arXiv.1706.03825>, 'Gradient x Input' or 'Vanilla Gradient'.
Details can be found in the accompanying scientific paper: Koenen & Wright (2024, Journal of Statistical Software, <doi:10.18637/jss.v111.i08>).

License MIT + file LICENSE

URL `https://bips-hb.github.io/innsight/`, `https://github.com/bips-hb/innsight/`

BugReports `https://github.com/bips-hb/innsight/issues/`

Depends R (>= 3.5.0)

Imports checkmate, cli, ggplot2, methods, R6, torch

Suggests covr, fastshap, GGally, grid, gridExtra, gtable, keras, knitr, lime, luz, neuralnet, palmerpenguins, plotly, rmarkdown, ranger, spelling, tensorflow, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

Language en-US

RoxygenNote 7.2.3

Collate 'AgnosticMethods.R' 'AgnosticWrapper.R' 'ConnectionWeights.R' 'Convert_keras.R' 'Convert_neuralnet.R' 'Convert_torch.R' 'ConvertedModel.R' 'Converter.R' 'DeepLift.R' 'GradienBased.R' 'InterpretingLayer.R' 'InterpretingMethod.R' 'LRP.R' 'Layer_conv1d.R' 'Layer_conv2d.R' 'Layer_dense.R' 'Layer_normalization.R' 'Layer_other.R' 'Layer_pooling.R' 'innsight.R' 'utils.R' 'utils_ggplot.R' 'utils_plotly.R' 'innsight_sugar.R' 'innsight_ggplot2.R' 'innsight_plotly.R'

NeedsCompilation no

Author Niklas Koenen [aut, cre] (<https://orcid.org/0000-0002-4623-8271>), Raphael Baudeu [ctb]

Maintainer Niklas Koenen <niklas.koenen@gmail.com>

Repository CRAN

Date/Publication 2024-11-26 09:50:02 UTC

Contents

innsight-package *Get the insight of your neural network*

Description

innsight is an R package that interprets the behavior and explains individual predictions of modern neural networks. Many methods for explaining individual predictions already exist, but hardly any of them are implemented or available in R. Most of these so-called *feature attribution* methods are only implemented in Python and, thus, difficult to access or use for the R community. In this sense, the package innsight provides a common interface for various methods for the interpretability of neural networks and can therefore be considered as an R analogue to 'iNNvestigate' or 'Captum' for Python.

Details

This package implements several model-specific interpretability (feature attribution) methods based on neural networks in R, e.g.,

- *Layer-wise relevance propagation (LRP)*
  - Including propagation rules: ϵ-rule and α-β-rule
- *Deep learning important features (DeepLift)*
  - Including propagation rules for non-linearities: *Rescale* rule and *RevealCancel* rule
- DeepSHAP
- Gradient-based methods:
  - *Vanilla* Gradient, including *Gradient*$\times$*Input*
  - Smoothed gradients *(SmoothGrad)*, including *SmoothGrad*$\times$*Input*
  - *Integrated gradients* (IntegratedGradient)
  - *Expected gradients* (ExpectedGradient)
- *ConnectionWeights*
- Model-agnostic methods:
  - *Local interpretable model-agnostic explanation (LIME)*
  - *Shapley values* (SHAP)

The package innsight aims to be as flexible as possible and independent of a specific deep learning package in which the passed network has been learned. Basically, a neural network of the libraries `torch::nn_sequential`, `keras::keras_model_sequential`, `keras::keras_model` and `neuralnet::neuralnet` can be passed to the main building block `Converter`, which converts and stores the passed model as a torch model (`ConvertedModel`) with special insights needed for interpretation. It is also possible to pass an arbitrary net in form of a named list (see details in `Converter`).

The scientific background and implementation details of innsight are described in the paper "Interpreting Deep Neural Networks with the Package innsight" by Koenen & Wright (2024), published in the *Journal of Statistical Software*. For a detailed explanation of the methods and use cases, please refer to the publication (doi: doi:10.18637/jss.v111.i08).

Author(s)

Maintainer: Niklas Koenen <niklas.koenen@gmail.com> (ORCID)

Other contributors:

- Raphael Baudeu <raphael.baudeu@gmail.com> [contributor]

References

Koenen, N., & Wright, M. N. (2024). Interpreting Deep Neural Networks with the Package innsight. Journal of Statistical Software, 111(8), 1-52. doi: doi:10.18637/jss.v111.i08

See Also

Useful links:

- https://bips-hb.github.io/innsight/
- https://github.com/bips-hb/innsight/
- Report bugs at https://github.com/bips-hb/innsight/issues/

`+,innsight_ggplot2,ANY-method`
Generic add function for `innsight_ggplot2`

Description

This generic add function allows to treat an instance of `innsight_ggplot2` as an ordinary plot object of ggplot2. For example geoms, themes and scales can be added as usual (see `ggplot2::+.gg` for more information).

Note: If `e1` represents a multiplot (i.e., `e1@mulitplot = TRUE`), `e2` is added to each individual plot. If only specific plots need to be changed, the generic assignment function should be used (see innsight_ggplot2 for details).

Usage

```
## S4 method for signature 'innsight_ggplot2,ANY'
e1 + e2
```

Arguments

`e1`	An instance of the S4 class `innsight_ggplot2`.
`e2`	An object of class `ggplot2::ggplot` or a `ggplot2::theme`.

See Also

`innsight_ggplot2`, `print.innsight_ggplot2`, `[.innsight_ggplot2`, `[[.innsight_ggplot2`, `[<-.innsight_ggplot2`, `[[<-.innsight_ggplot2`

`AgnosticWrapper` *Super class for model-agnostic interpretability methods*

Description

This is a super class for all implemented model-agnostic interpretability methods and inherits from the `InterpretingMethod` class. Instead of just an object of the `Converter` class, any model can now be passed. In contrast to the other model-specific methods in this package, only the prediction function of the model is required, and not the internal details of the model. The following model-agnostic methods are available (all are wrapped by other packages):

- *Shapley values* (`SHAP`) based on `fastshap::explain`
- *Local interpretable model-agnostic explanations* (`LIME`) based on `lime::lime`

Super class

`innsight::InterpretingMethod` -> `AgnosticWrapper`

Public fields

`data_orig` The individual instances to be explained by the method (unprocessed!).

Methods

Public methods:

- `AgnosticWrapper$new()`
- `AgnosticWrapper$clone()`

Method `new()`**:** Create a new instance of the `AgnosticWrapper` R6 class.

Usage:

```
AgnosticWrapper$new(
  model,
  data,
  data_ref,
  output_type = NULL,
  pred_fun = NULL,
  output_idx = NULL,
  output_label = NULL,
  channels_first = TRUE,
  input_dim = NULL,
  input_names = NULL,
  output_names = NULL
)
```

Arguments:

`model` (any prediction model)
A fitted model for a classification or regression task that is intended to be interpreted. A `Converter` object can also be passed. In order for the package to know how to make predictions with the given model, a prediction function must also be passed with the argument `pred_fun`. However, for models created by `nn_sequential`, `keras_model`, `neuralnet` or `Converter`, these have already been pre-implemented and do not need to be specified.

`data` (`array`, `data.frame` or `torch_tensor`)
The individual instances to be explained by the method. These must have the same format as the input data of the passed model and has to be either `matrix`, an `array`, a `data.frame` or a `torch_tensor`. If no value is specified, all instances in the dataset `data` will be explained.
Note: For the model-agnostic methods, only models with a single input and output layer is allowed!

`data_ref` (`array`, `data.frame` or `torch_tensor`)
The dataset to which the method is to be applied. These must have the same format as the input data of the passed model and has to be either `matrix`, an `array`, a `data.frame` or a `torch_tensor`.
Note: For the model-agnostic methods, only models with a single input and output layer is allowed!

`output_type` (`character(1)`)
Type of the model output, i.e., either `"classification"` or `"regression"`.

`pred_fun` (`function`)
Prediction function for the model. This argument is only needed if `model` is not a model created by `nn_sequential`, `keras_model`, `neuralnet` or `Converter`. The first argument of `pred_fun` has to be `newdata`, e.g.,

```
function(newdata, ...) model(newdata)
```

`output_idx` (`integer`, `list` or `NULL`)
These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:

- An `integer` vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., `c(1,3,4)` for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.
- A `list` of `integer` vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of indices, e.g., `list(NULL, c(1,3))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`output_label` (`character`, `factor`, `list` or `NULL`)

These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument `output_names` in the `converter` can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A `character` vector or `factor` of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed `Converter` object, e.g., `c("a", "c", "d")` for the first, third and fourth output node if the output names are `c("a", "b", "c", "d")`. If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A `list` of `charactor`/`factor` vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`channels_first` (`logical(1)`)

The channel position of the given data (argument `data`). If `TRUE`, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. If the data has no channel axis, use the default value `TRUE`.

`input_dim` (`integer`)

The model input dimension excluding the batch dimension. It can be specified as vector of integers, but has to be in the format "channels first".

`input_names` (`character`, `factor` or `list`)

The input names of the model excluding the batch dimension. For a model with a single input layer and input axis (e.g., for tabular data), the input names can be specified as a character vector or factor, e.g., for a dense layer with 3 input features use `c("X1", "X2", "X3")`. If the model input consists of multiple axes (e.g., for signal and image data), use a list of character vectors or factors for each axis in the format "channels first", e.g., use `list(c("C1", "C2"), c("L1","L2","L3","L4","L5"))` for a 1D convolutional input layer with signal length 4 and 2 channels.
Note: This argument is optional and otherwise the names are generated automatically. But if this argument is set, all found input names in the passed model will be disregarded.

`output_names` (`character`, `factor`)

A character vector with the names for the output dimensions excluding the batch dimension, e.g., for a model with 3 output nodes use `c("Y1", "Y2", "Y3")`. Instead of a character vector you can also use a factor to set an order for the plots.
Note: This argument is optional and otherwise the names are generated automatically. But if this argument is set, all found output names in the passed model will be disregarded.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
AgnosticWrapper$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`ConnectionWeights` *Connection weights method*

Description

This class implements the *Connection weights* method investigated by Olden et al. (2004), which results in a relevance score for each input variable. The basic idea is to multiply all path weights for each possible connection between an input feature and the output node and then calculate the sum over them. Besides, it is originally a global interpretation method and independent of the input data. For a neural network with 3 hidden layers with weight matrices W_1, W_2 and W_3, this method results in a simple matrix multiplication independent of the activation functions in between:

$$W_1 * W_2 * W_3.$$

In this package, we extended this method to a local method inspired by the method *Gradient*×*Input* (see `Gradient`). Hence, the local variant is simply the point-wise product of the global *Connection weights* method and the input data. You can use this variant by setting the `times_input` argument to `TRUE` and providing input data.

The R6 class can also be initialized using the `run_cw` function as a helper function so that no prior knowledge of R6 classes is required.

Super class

`innsight::InterpretingMethod` -> `ConnectionWeights`

Public fields

`times_input (logical(1))`
: This logical value indicates whether the results from the *Connection weights* method were multiplied by the provided input data or not. Thus, this value specifies whether the original global variant of the method or the local one was applied. If the value is `TRUE`, then data is provided in the field `data`.

Methods

Public methods:

- `ConnectionWeights$new()`
- `ConnectionWeights$clone()`

Method `new()`**:** Create a new instance of the class `ConnectionWeights`. When initialized, the method is applied and the results are stored in the field `result`.

Usage:

```
ConnectionWeights$new(
  converter,
  data = NULL,
  output_idx = NULL,
  output_label = NULL,
  channels_first = TRUE,
  times_input = FALSE,
  verbose = interactive(),
  dtype = "float"
)
```

Arguments:

`converter` (`Converter`)
: An instance of the `Converter` class that includes the torch-converted model and some other model-specific attributes. See `Converter` for details.

`data` (`array`, `data.frame`, `torch_tensor` or `list`)
: The data to which the method is to be applied. These must have the same format as the input data of the passed model to the converter object. This means either

- an `array`, `data.frame`, `torch_tensor` or array-like format of size *(batch_size, dim_in)*, if e.g.the model has only one input layer, or
- a `list` with the corresponding input data (according to the upper point) for each of the input layers.

This argument is only relevant if `times_input` is `TRUE`, otherwise it will be ignored because it is a locale (i.e. explanation for each data point individually) method only in this case.

`output_idx` (`integer`, `list` or `NULL`)
: These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:

- An `integer` vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., `c(1,3,4)` for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.
- A `list` of `integer` vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of indices, e.g., `list(NULL, c(1,3))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`output_label` (character, factor, list or NULL)
These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument `output_names` in the `converter` can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A `character` vector or `factor` of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed `Converter` object, e.g., `c("a", "c", "d")` for the first, third and fourth output node if the output names are `c("a", "b", "c", "d")`. If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A `list` of `charactor`/`factor` vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`channels_first (logical(1))`
The channel position of the given data (argument `data`). If `TRUE`, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. If the data has no channel axis, use the default value `TRUE`.

`times_input (logical(1))`
Multiplies the results with the input features. This variant turns the global *Connection weights* method into a local one. Default: `FALSE`.

`verbose (logical(1))`
This logical argument determines whether a progress bar is displayed for the calculation of the method or not. The default value is the output of the primitive R function `interactive()`.

`dtype (character(1))`
The data type for the calculations. Use either `'float'` for `torch_float` or `'double'` for `torch_double`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ConnectionWeights$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

- J. D. Olden et al. (2004) *An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data.* Ecological Modelling 178, p. 389–397

See Also

Other methods: DeepLift, DeepSHAP, ExpectedGradient, Gradient, IntegratedGradient, LIME, LRP, SHAP, SmoothGrad

Examples

```
#----------------------- Example 1: Torch ----------------------------------
library(torch)

# Create nn_sequential model
model <- nn_sequential(
  nn_linear(5, 12),
  nn_relu(),
  nn_linear(12, 1),
  nn_sigmoid()
)

# Create Converter with input names
converter <- Converter$new(model,
  input_dim = c(5),
  input_names = list(c("Car", "Cat", "Dog", "Plane", "Horse"))
)

# You can also use the helper function for the initialization part
converter <- convert(model,
  input_dim = c(5),
  input_names = list(c("Car", "Cat", "Dog", "Plane", "Horse"))
)

# Apply method Connection Weights
cw <- ConnectionWeights$new(converter)

# Again, you can use a helper function `run_cw()` for initializing
cw <- run_cw(converter)

# Print the head of the result as a data.frame
head(get_result(cw, "data.frame"), 5)

# Plot the result
plot(cw)

#----------------------- Example 2: Neuralnet ------------------------------
if (require("neuralnet")) {
  library(neuralnet)
  data(iris)
```

```
  # Train a Neural Network
  nn <- neuralnet((Species == "setosa") ~ Petal.Length + Petal.Width,
    iris,
    linear.output = FALSE,
    hidden = c(3, 2), act.fct = "tanh", rep = 1
  )

  # Convert the trained model
  converter <- convert(nn)

  # Apply the Connection Weights method
  cw <- run_cw(converter)

  # Get the result as a torch tensor
  get_result(cw, type = "torch.tensor")

  # Plot the result
  plot(cw)
}


# ------------------------- Example 3: Keras -------------------------------
if (require("keras") & keras::is_keras_available()) {
  library(keras)

  # Make sure keras is installed properly
  is_keras_available()

  data <- array(rnorm(10 * 32 * 32 * 3), dim = c(10, 32, 32, 3))

  model <- keras_model_sequential()
  model %>%
    layer_conv_2d(
      input_shape = c(32, 32, 3), kernel_size = 8, filters = 8,
      activation = "softplus", padding = "valid") %>%
    layer_conv_2d(
      kernel_size = 8, filters = 4, activation = "tanh",
      padding = "same") %>%
    layer_conv_2d(
      kernel_size = 4, filters = 2, activation = "relu",
      padding = "valid") %>%
    layer_flatten() %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 16, activation = "relu") %>%
    layer_dense(units = 2, activation = "softmax")

  # Convert the model
  converter <- convert(model)

  # Apply the Connection Weights method
  cw <- run_cw(converter)

  # Get the head of the result as a data.frame
```

```
  head(get_result(cw, type = "data.frame"), 5)

  # Plot the result for all classes
  plot(cw, output_idx = 1:2)
}


#------------------------ Plotly plots ------------------------------------
if (require("plotly")) {
  # You can also create an interactive plot with plotly.
  # This is a suggested package, so make sure that it is installed
  library(plotly)
  plot(cw, as_plotly = TRUE)
}
```

ConvertedModel *Converted torch-based model*

Description

This class stores all layers converted to torch in a module which can be used like the original model (but torch-based). In addition, it provides other functions that are useful for interpreting individual predictions or explaining the entire model. This model is part of the class Converter and is the core for all the necessary calculations in the methods provided in this package.

Usage

```
ConvertedModel(modules_list, graph, input_nodes, output_nodes, dtype = "float")
```

Arguments

modules_list (list)
A list of all accepted layers created by the Converter class during initialization.

graph (list)
The graph argument gives a way to pass an input through the model, which is especially relevant for non-sequential architectures. It can be seen as a list of steps in which order the layers from modules_list must be applied. The list contains the following elements:

- $current_nodes
This list describes the current position and the number of the respective intermediate values when passing through the model. For example, list(1,3,3) means that in this step one output from the first layer and two from the third layer (the numbers correspond to the list indices from the modules_list argument) are available for the calculation of the current layer with index used_node.

- `$used_node`
  The index of the layer from the `modules_list` argument which will be applied in this step.
- `$used_idx`
  The indices of the outputs from `current_nodes`, which are used as inputs of the current layer (`used_node`).
- `$times`
  The frequency of the output value, i.e., is the output used more than once as an input for subsequent layers?

`input_nodes` (`numeric`)
A vector of layer indices describing the input layers, i.e., they are used as the starting point for the calculations.

`output_nodes` (`numeric`)
A vector of layer indices describing the indices of the output layers.

`dtype` (`character(1)`)
The data type for all the calculations and defined tensors. Use either `'float'` for `torch::torch_float` or `'double'` for `torch::torch_double`.

Method `forward()`

The forward method of the whole model, i.e., it calculates the output $y = f(x)$ of a given input x. In doing so, all intermediate values are stored in the individual torch modules from `modules_list`.

Usage:

```
self(x,
     channels_first = TRUE,
     save_input = FALSE,
     save_preactivation = FALSE,
     save_output = FAlSE,
     save_last_layer = FALSE)
```

Arguments:

`x` The input torch tensor for this model.

`channels_first` If the input tensor `x` is given in the format 'channels first', use `TRUE`. Otherwise, if the channels are last, use `FALSE` and the input will be transformed into the format 'channels first'. Default: `TRUE`.

`save_input` Logical value whether the inputs from each layer are to be saved or not. Default: `FALSE`.

`save_preactivation` Logical value whether the preactivations from each layer are to be saved or not. Default: `FALSE`.

`save_output` Logical value whether the outputs from each layer are to be saved or not. Default: `FALSE`.

save_last_layer Logical value whether the inputs, preactivations and outputs from the last layer are to be saved or not. Default: FALSE.

Return:

Returns a list of the output values of the model with respect to the given inputs.

Method update_ref()

This method updates the intermediate values in each module from the list modules_list for the reference input x_ref and returns the output from it in the same way as in the forward method.

Usage:

```
self$update_ref(x_ref,
                channels_first = TRUE,
                save_input = FALSE,
                save_preactivation = FALSE,
                save_output = FAlSE,
                save_last_layer = FALSE)
```

Arguments:

x_ref Reference input of the model.

channels_first If the tensor x_ref is given in the format 'channels first' use TRUE. Otherwise, if the channels are last, use FALSE and the input will be transformed into the format 'channels first'. Default: TRUE.

save_input Logical value whether the inputs from each layer are to be saved or not. Default: FALSE.

save_preactivation Logical value whether the preactivations from each layer are to be saved or not. Default: FALSE.

save_output Logical value whether the outputs from each layer are to be saved or not. Default: FALSE.

save_last_layer Logical value whether the inputs, preactivations and outputs from the last layer are to be saved or not. Default: FALSE.

Return:

Returns a list of the output values of the model with respect to the given reference input.

Method set_dtype()

This method changes the data type for all the layers in modules_list. Use either 'float' for torch::torch_float or 'double' for torch::torch_double.

Usage:

```
self$set_dtype(dtype)
```

Arguments:

dtype The data type for all the calculations and defined tensors.

Converter *Converter of an artificial neural network*

Description

This class analyzes a passed neural network and stores its internal structure and the individual layers by converting the entire network into an nn_module. With the help of this converter, many methods for interpreting the behavior of neural networks are provided, which give a better understanding of the whole model or individual predictions. You can use models from the following libraries:

- torch (nn_sequential)
- keras (keras_model, keras_model_sequential),
- neuralnet

Furthermore, a model can be passed as a list (see `vignette("detailed_overview", package = "innsight")` or the website).

The R6 class can also be initialized using the convert function as a helper function so that no prior knowledge of R6 classes is required.

Details

In order to better understand and analyze the prediction of a neural network, the preactivation or other information of the individual layers, which are not stored in an ordinary forward pass, are often required. For this reason, a given neural network is converted into a torch-based neural network, which provides all the necessary information for an interpretation. The converted torch model is stored in the field `model` and is an instance of ConvertedModel. However, before the torch model is created, all relevant details of the passed model are extracted into a named list. This list can be saved in complete form in the `model_as_list` field with the argument `save_model_as_list`, but this may consume a lot of memory for large networks and is not done by default. Also, this named list can again be used as a passed model for the class `Converter`, which will be described in more detail in the section 'Implemented Libraries'.

Implemented methods:

An object of the Converter class can be applied to the following methods:

- *Layerwise Relevance Propagation* (LRP), Bach et al. (2015)
- *Deep Learning Important Features* (DeepLift), Shrikumar et al. (2017)
- *DeepSHAP*, Lundberg et al. (2017)
- *SmoothGrad* including *SmoothGrad*×*Input*, Smilkov et al. (2017)
- *Vanilla Gradient* including *Gradient*×*Input*
- *Integrated gradients* (IntegratedGradient), Sundararajan et al. (2017)
- *Expected gradients* (ExpectedGradient), Erion et al. (2021)
- *ConnectionWeights*, Olden et al. (2004)
- *Local interpretable model-agnostic explanation (LIME)*, Ribeiro et al. (2016)
- *Shapley values (SHAP)*, Lundberg et al. (2017)

Implemented libraries:

The converter is implemented for models from the libraries `nn_sequential`, `neuralnet` and `keras`. But you can also write a wrapper for other libraries because a model can be passed as a named list which is described in detail in the vignette "In-depth explanation" (see `vignette("detailed_overview", package = "innsight")` or the website).

Public fields

`model` (`ConvertedModel`)
: The converted neural network based on the torch module ConvertedModel.

`input_dim` (`list`)
: A list of the input dimensions of each input layer. Since internally the "channels first" format is used for all calculations, the input shapes are already in this format. In addition, the batch dimension isn't included, e.g., for an input layer of shape `c(*,32,32,3)` with channels in the last axis you get `list(c(3,32,32))`.

`input_names` (`list`)
: A list with the names as factors for each input dimension of the shape as stored in the field `input_dim`.

`output_dim` (`list`)
: A list of the output dimensions of each output layer.

`output_names` (`list`)
: A list with the names as factors for each output dimension of shape as stored in the field `output_dim`.

`model_as_list` (`list`)
: The model stored in a named list (see details for more information). By default, the entry `model_as_list$layers` is deleted because it may require a lot of memory for large networks. However, with the argument `save_model_as_list` this can be saved anyway.

Methods

Public methods:

- `Converter$new()`
- `Converter$print()`
- `Converter$clone()`

Method `new()`**:** Create a new Converter object for a given neural network. When initialized, the model is inspected, converted as a list and then the a torch-converted model (ConvertedModel) is created and stored in the field `model`.

Usage:

```
Converter$new(
  model,
  input_dim = NULL,
  input_names = NULL,
  output_names = NULL,
  dtype = "float",
  save_model_as_list = FALSE
)
```

Arguments:

`model` (`nn_sequential`, `keras_model`, `neuralnet` or `list`)
: A trained neural network for classification or regression tasks to be interpreted. Only models from the following types or packages are allowed: `nn_sequential`, `keras_model`, `keras_model_sequential`, `neuralnet` or a named list (see details).

`input_dim` (`integer` or `list`)
: The model input dimension excluding the batch dimension. If there is only one input layer it can be specified as a vector, otherwise use a list of the shapes of the individual input layers.
Note: This argument is only necessary for `torch::nn_sequential`, for all others it is automatically extracted from the passed model and used for internal checks. In addition, the input dimension `input_dim` has to be in the format "channels first".

`input_names` (`character`, `factor` or `list`)
: The input names of the model excluding the batch dimension. For a model with a single input layer and input axis (e.g., for tabular data), the input names can be specified as a character vector or factor, e.g., for a dense layer with 3 input features use `c("X1", "X2", "X3")`. If the model input consists of multiple axes (e.g., for signal and image data), use a list of character vectors or factors for each axis in the format "channels first", e.g., use `list(c("C1", "C2"), c("L1","L2","L3","L4","L5"))` for a 1D convolutional input layer with signal length 4 and 2 channels. For models with multiple input layers, use a list of the upper ones for each layer.
Note: This argument is optional and otherwise the names are generated automatically. But if this argument is set, all found input names in the passed model will be disregarded.

`output_names` (`character`, `factor` or `list`)
: A character vector with the names for the output dimensions excluding the batch dimension, e.g., for a model with 3 output nodes use `c("Y1", "Y2", "Y3")`. Instead of a character vector you can also use a factor to set an order for the plots. If the model has multiple output layers, use a list of the upper ones.
Note: This argument is optional and otherwise the names are generated automatically. But if this argument is set, all found output names in the passed model will be disregarded.

`dtype` (`character(1)`)
: The data type for the calculations. Use either `'float'` for torch::torch_float or `'double'` for torch::torch_double.

`save_model_as_list` (`logical(1)`)
: This logical value specifies whether the passed model should be stored as a list. This list can take a lot of memory for large networks, so by default the model is not stored as a list

(FALSE).

Returns: A new instance of the R6 class `Converter`.

Method `print()`**:** Print a summary of the `Converter` object. This summary contains the individual fields and in particular the torch-converted model (ConvertedModel) with the layers.

Usage:

```
Converter$print()
```

Returns: Returns the `Converter` object invisibly via `base::invisible`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Converter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

- J. D. Olden et al. (2004) *An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data.* Ecological Modelling 178, p. 389–397
- S. Bach et al. (2015) *On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation.* PLoS ONE 10, p. 1-46
- M. T. Ribeiro et al. (2016) *"Why should I trust you?": Explaining the predictions of any classifier.* KDD 2016, p. 1135-1144
- A. Shrikumar et al. (2017) *Learning important features through propagating activation differences.* ICML 2017, p. 4844-4866
- D. Smilkov et al. (2017) *SmoothGrad: removing noise by adding noise.* CoRR, abs/1706.03825 M. Sundararajan et al. (2017) *Axiomatic attribution for deep networks.* ICML 2017, p.3319-3328
- S. Lundberg et al. (2017) *A unified approach to interpreting model predictions.* NIPS 2017, p. 4768-4777
- G. Erion et al. (2021) *Improving performance of deep learning models with axiomatic attribution priors and expected gradients.* Nature Machine Intelligence 3, p. 620-631

Examples

```
#----------------------- Example 1: Torch ----------------------------------
library(torch)

model <- nn_sequential(
  nn_linear(5, 10),
  nn_relu(),
  nn_linear(10, 2, bias = FALSE),
  nn_softmax(dim = 2)
)
```

```
data <- torch_randn(25, 5)

# Convert the model (for torch models is 'input_dim' required!)
converter <- Converter$new(model, input_dim = c(5))

# You can also use the helper function `convert()` for initializing a
# Converter object
converter <- convert(model, input_dim = c(5))

# Get the converted model stored in the field 'model'
converted_model <- converter$model

# Test it with the original model
mean(abs(converted_model(data)[[1]] - model(data)))


#----------------------- Example 2: Neuralnet ------------------------------
if (require("neuralnet")) {
  library(neuralnet)
  data(iris)

  # Train a neural network
  nn <- neuralnet((Species == "setosa") ~ Petal.Length + Petal.Width,
    iris,
    linear.output = FALSE,
    hidden = c(3, 2), act.fct = "tanh", rep = 1
  )

  # Convert the model
  converter <- convert(nn)

  # Print all the layers
  converter$model$modules_list
}


#----------------------- Example 3: Keras ----------------------------------
if (require("keras") & keras::is_keras_available()) {
  library(keras)

  # Make sure keras is installed properly
  is_keras_available()

  # Define a keras model
  model <- keras_model_sequential() %>%
    layer_conv_2d(
      input_shape = c(32, 32, 3), kernel_size = 8, filters = 8,
      activation = "relu", padding = "same") %>%
    layer_conv_2d(
      kernel_size = 8, filters = 4,
      activation = "tanh", padding = "same") %>%
    layer_conv_2d(
      kernel_size = 4, filters = 2,
```

```
      activation = "relu", padding = "same") %>%
    layer_flatten() %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1, activation = "sigmoid")

  # Convert this model and save model as list
  converter <- convert(model, save_model_as_list = TRUE)

  # Print the converted model as a named list
  str(converter$model_as_list, max.level = 1)
}


#----------------------- Example 4: List  ----------------------------------

# Define a model

model <- list()
model$input_dim <- 5
model$input_names <- list(c("Feat1", "Feat2", "Feat3", "Feat4", "Feat5"))
model$input_nodes <- c(1)
model$output_dim <- 2
model$output_names <- list(c("Cat", "no-Cat"))
model$output_nodes <- c(2)
model$layers$Layer_1 <-
  list(
    type = "Dense",
    weight = matrix(rnorm(5 * 20), 20, 5),
    bias = rnorm(20),
    activation_name = "tanh",
    dim_in = 5,
    dim_out = 20,
    input_layers = 0, # '0' means model input layer
    output_layers = 2
  )
model$layers$Layer_2 <-
  list(
    type = "Dense",
    weight = matrix(rnorm(20 * 2), 2, 20),
    bias = rnorm(2),
    activation_name = "softmax",
    input_layers = 1,
    output_layers = -1 # '-1' means model output layer
    #dim_in = 20, # These values are optional, but
    #dim_out = 2  # useful for internal checks
  )

# Convert the model
converter <- convert(model)

# Get the model as a torch::nn_module
torch_model <- converter$model
```

```
# You can use it as a normal torch model
x <- torch::torch_randn(3, 5)
torch_model(x)
```

DeepLift *Deep learning important features (DeepLift)*

Description

This is an implementation of the *deep learning important features (DeepLift)* algorithm introduced by Shrikumar et al. (2017). It's a local method for interpreting a single element x of the dataset concerning a reference value x' and returns the contribution of each input feature from the difference of the output ($y = f(x)$) and reference output ($y' = f(x')$) prediction. The basic idea of this method is to decompose the difference-from-reference prediction with respect to the input features, i.e.,

$$\Delta y = y - y' = \sum_i C(x_i).$$

Compared to *Layer-wise relevance propagation* (see LRP), the DeepLift method is an exact decomposition and not an approximation, so we get real contributions of the input features to the difference-from-reference prediction. There are two ways to handle activation functions: the *Rescale* rule (`'rescale'`) and *RevealCancel* rule (`'reveal_cancel'`).

The R6 class can also be initialized using the `run_deeplift` function as a helper function so that no prior knowledge of R6 classes is required.

Super class

`innsight::InterpretingMethod` -> `DeepLift`

Public fields

`x_ref (list)`
: The reference input for the DeepLift method. This value is stored as a list of `torch_tensors` of shape *(1, dim_in)* for each input layer.

`rule_name (character(1))`
: Name of the applied rule to calculate the contributions. Either `'rescale'` or `'reveal_cancel'`.

Methods

Public methods:

- `DeepLift$new()`
- `DeepLift$clone()`

Method `new()`**:** Create a new instance of the `DeepLift` R6 class. When initialized, the method *DeepLift* is applied to the given data and the results are stored in the field `result`.

Usage:

```
DeepLift$new(
  converter,
  data,
  channels_first = TRUE,
  output_idx = NULL,
  output_label = NULL,
  ignore_last_act = TRUE,
  rule_name = "rescale",
  x_ref = NULL,
  winner_takes_all = TRUE,
  verbose = interactive(),
  dtype = "float"
)
```

Arguments:

`converter` (`Converter`)
An instance of the `Converter` class that includes the torch-converted model and some other model-specific attributes. See `Converter` for details.

`data` (`array`, `data.frame`, `torch_tensor` or `list`)
The data to which the method is to be applied. These must have the same format as the input data of the passed model to the converter object. This means either

- an `array`, `data.frame`, `torch_tensor` or array-like format of size *(batch_size, dim_in)*, if e.g., the model has only one input layer, or
- a `list` with the corresponding input data (according to the upper point) for each of the input layers.

`channels_first` (`logical(1)`)
The channel position of the given data (argument `data`). If `TRUE`, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. If the data has no channel axis, use the default value `TRUE`.

`output_idx` (`integer`, `list` or `NULL`)
These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:

- An `integer` vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., `c(1,3,4)` for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.
- A `list` of `integer` vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of indices, e.g., `list(NULL, c(1,3))` for the first and third output node in the second output layer.

- NULL (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

output_label (character, factor, list or NULL)
These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument output_names in the converter can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A character vector or factor of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed Converter object, e.g., c("a", "c", "d") for the first, third and fourth output node if the output names are c("a", "b", "c", "d"). If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A list of charactor/factor vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry NULL instead of a vector of labels, e.g., list(NULL, c("a", "c")) for the first and third output node in the second output layer.
- NULL (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

ignore_last_act (logical(1))
Set this logical value to include the last activation functions for each output layer, or not (default: TRUE). In practice, the last activation (especially for softmax activation) is often omitted.

rule_name (character(1))
Name of the applied rule to calculate the contributions. Use either 'rescale' or 'reveal_cancel'.

x_ref (array, data.frame, torch_tensor or list)
The reference input for the DeepLift method. This value must have the same format as the input data of the passed model to the converter object. This means either

- an array, data.frame, torch_tensor or array-like format of size *(1, dim_in)*, if e.g., the model has only one input layer, or
- a list with the corresponding input data (according to the upper point) for each of the input layers.
- It is also possible to use the default value NULL to take only zeros as reference input.

winner_takes_all (logical(1))
This logical argument is only relevant for MaxPooling layers and is otherwise ignored. With this layer type, it is possible that the position of the maximum values in the pooling kernel of the normal input x and the reference input x' may not match, which leads to a violation of the summation-to-delta property. To overcome this problem, another variant is implemented, which treats a MaxPooling layer as an AveragePooling layer in the backward pass

only, leading to an uniform distribution of the upper-layer contribution to the lower layer.

`verbose (logical(1))`
This logical argument determines whether a progress bar is displayed for the calculation of the method or not. The default value is the output of the primitive R function `interactive()`.

`dtype (character(1))`
The data type for the calculations. Use either `'float'` for `torch_float` or `'double'` for `torch_double`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DeepLift$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

A. Shrikumar et al. (2017) *Learning important features through propagating activation differences.* ICML 2017, p. 4844-4866

See Also

Other methods: `ConnectionWeights`, `DeepSHAP`, `ExpectedGradient`, `Gradient`, `IntegratedGradient`, `LIME`, `LRP`, `SHAP`, `SmoothGrad`

Examples

```
#----------------------- Example 1: Torch ----------------------------------
library(torch)

# Create nn_sequential model and data
model <- nn_sequential(
  nn_linear(5, 12),
  nn_relu(),
  nn_linear(12, 2),
  nn_softmax(dim = 2)
)
data <- torch_randn(25, 5)
ref <- torch_randn(1, 5)

# Create Converter using the helper function `convert`
converter <- convert(model, input_dim = c(5))

# Apply method DeepLift
deeplift <- DeepLift$new(converter, data, x_ref = ref)

# You can also use the helper function `run_deeplift` for initializing
# an R6 DeepLift object
```

```
deeplift <- run_deeplift(converter, data, x_ref = ref)

# Print the result as a torch tensor for first two data points
get_result(deeplift, "torch.tensor")[1:2]

# Plot the result for both classes
plot(deeplift, output_idx = 1:2)

# Plot the boxplot of all datapoints and for both classes
boxplot(deeplift, output_idx = 1:2)

# ------------------------- Example 2: Neuralnet ---------------------------
if (require("neuralnet")) {
  library(neuralnet)
  data(iris)

  # Train a neural network
  nn <- neuralnet((Species == "setosa") ~ Petal.Length + Petal.Width,
    iris,
    linear.output = FALSE,
    hidden = c(3, 2), act.fct = "tanh", rep = 1
  )

  # Convert the model
  converter <- convert(nn)

  # Apply DeepLift with rescale-rule and a reference input of the feature
  # means
  x_ref <- matrix(colMeans(iris[, c(3, 4)]), nrow = 1)
  deeplift_rescale <- run_deeplift(converter, iris[, c(3, 4)], x_ref = x_ref)

  # Get the result as a dataframe and show first 5 rows
  get_result(deeplift_rescale, type = "data.frame")[1:5, ]

  # Plot the result for the first datapoint in the data
  plot(deeplift_rescale, data_idx = 1)

  # Plot the result as boxplots
  boxplot(deeplift_rescale)
}


# ------------------------- Example 3: Keras -------------------------------
if (require("keras") & keras::is_keras_available()) {
  library(keras)

  # Make sure keras is installed properly
  is_keras_available()

  data <- array(rnorm(10 * 32 * 32 * 3), dim = c(10, 32, 32, 3))

  model <- keras_model_sequential()
  model %>%
```

```
    layer_conv_2d(
      input_shape = c(32, 32, 3), kernel_size = 8, filters = 8,
      activation = "softplus", padding = "valid") %>%
    layer_conv_2d(
      kernel_size = 8, filters = 4, activation = "tanh",
      padding = "same") %>%
    layer_conv_2d(
      kernel_size = 4, filters = 2, activation = "relu",
      padding = "valid") %>%
    layer_flatten() %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 16, activation = "relu") %>%
    layer_dense(units = 2, activation = "softmax")

  # Convert the model
  converter <- convert(model)

  # Apply the DeepLift method with reveal-cancel rule
  deeplift_revcancel <- run_deeplift(converter, data,
    channels_first = FALSE,
    rule_name = "reveal_cancel"
  )

  # Plot the result for the first image and both classes
  plot(deeplift_revcancel, output_idx = 1:2)

  # Plot the pixel-wise median reelvance image
  plot_global(deeplift_revcancel, output_idx = 1)
}


#------------------------ Plotly plots ------------------------------------
if (require("plotly")) {
  # You can also create an interactive plot with plotly.
  # This is a suggested package, so make sure that it is installed
  library(plotly)
  boxplot(deeplift, as_plotly = TRUE)
}
```

DeepSHAP *Deep Shapley additive explanations (DeepSHAP)*

Description

The *DeepSHAP* method extends the DeepLift technique by not only considering a single reference value but by calculating the average from several, ideally representative reference values at each layer. The obtained feature-wise results are approximate Shapley values for the chosen output, where the conditional expectation is computed using these different reference values, i.e., the *DeepSHAP* method decompose the difference from the prediction and the mean prediction

$f(x) - E[f(\tilde{x})]$ in feature-wise effects. The reference values can be passed by the argument `data_ref`.

The R6 class can also be initialized using the `run_deepshap` function as a helper function so that no prior knowledge of R6 classes is required.

Super class

`innsight::InterpretingMethod` -> `DeepSHAP`

Public fields

`rule_name (character(1))`
: Name of the applied rule to calculate the contributions. Either `'rescale'` or `'reveal_cancel'`.

`data_ref (list)`
: The passed reference dataset for estimating the conditional expectation as a `list` of `torch_tensors` in the selected data format (field `dtype`) matching the corresponding shapes of the individual input layers. Besides, the channel axis is moved to the second position after the batch size because internally only the format *channels first* is used.

Methods

Public methods:

- `DeepSHAP$new()`
- `DeepSHAP$clone()`

Method `new()`**:** Create a new instance of the `DeepSHAP` R6 class. When initialized, the method *DeepSHAP* is applied to the given data and the results are stored in the field `result`.

Usage:

```
DeepSHAP$new(
  converter,
  data,
  channels_first = TRUE,
  output_idx = NULL,
  output_label = NULL,
  ignore_last_act = TRUE,
  rule_name = "rescale",
  data_ref = NULL,
  limit_ref = 100,
  winner_takes_all = TRUE,
  verbose = interactive(),
  dtype = "float"
)
```

Arguments:

converter (Converter)
An instance of the Converter class that includes the torch-converted model and some other model-specific attributes. See Converter for details.

data (array, data.frame, torch_tensor or list)
The data to which the method is to be applied. These must have the same format as the input data of the passed model to the converter object. This means either

- an array, data.frame, torch_tensor or array-like format of size *(batch_size, dim_in)*, if e.g., the model has only one input layer, or
- a list with the corresponding input data (according to the upper point) for each of the input layers.

channels_first (logical(1))
The channel position of the given data (argument data). If TRUE, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., c(10,3,32,32) for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (FALSE), the channel axis is at the last position, i.e., c(10,32,32,3). If the data has no channel axis, use the default value TRUE.

output_idx (integer, list or NULL)
These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:

- An integer vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., c(1,3,4) for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.
- A list of integer vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry NULL instead of a vector of indices, e.g., list(NULL, c(1,3)) for the first and third output node in the second output layer.
- NULL (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

output_label (character, factor, list or NULL)
These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument output_names in the converter can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A character vector or factor of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed Converter object, e.g., c("a", "c", "d") for the first, third and fourth output node if the output names are c("a", "b", "c", "d"). If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A list of charactor/factor vectors of labels: If the method is to be applied to output

nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.

- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`ignore_last_act (logical(1))`
Set this logical value to include the last activation functions for each output layer, or not (default: `TRUE`). In practice, the last activation (especially for softmax activation) is often omitted.

`rule_name (character(1))`
Name of the applied rule to calculate the contributions. Use either `'rescale'` or `'reveal_cancel'`.

`data_ref (array, data.frame, torch_tensor or list)`
The reference data which is used to estimate the conditional expectation. These must have the same format as the input data of the passed model to the converter object. This means either

- an `array`, `data.frame`, `torch_tensor` or array-like format of size *(batch_size, dim_in)*, if e.g., the model has only one input layer, or
- a `list` with the corresponding input data (according to the upper point) for each of the input layers.
- or `NULL` (default) to use only a zero baseline for the estimation.

`limit_ref (integer(1))`
This argument limits the number of instances taken from the reference dataset `data_ref` so that only random `limit_ref` elements and not the entire dataset are used to estimate the conditional expectation. A too-large number can significantly increase the computation time.

`winner_takes_all (logical(1))`
This logical argument is only relevant for MaxPooling layers and is otherwise ignored. With this layer type, it is possible that the position of the maximum values in the pooling kernel of the normal input x and the reference input x' may not match, which leads to a violation of the summation-to-delta property. To overcome this problem, another variant is implemented, which treats a MaxPooling layer as an AveragePooling layer in the backward pass only, leading to an uniform distribution of the upper-layer contribution to the lower layer.

`verbose (logical(1))`
This logical argument determines whether a progress bar is displayed for the calculation of the method or not. The default value is the output of the primitive R function `interactive()`.

`dtype (character(1))`
The data type for the calculations. Use either `'float'` for `torch_float` or `'double'` for

torch_double.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
DeepSHAP$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

References

S. Lundberg & S. Lee (2017) *A unified approach to interpreting model predictions.* NIPS 2017, p. 4768–4777

See Also

Other methods: ConnectionWeights, DeepLift, ExpectedGradient, Gradient, IntegratedGradient, LIME, LRP, SHAP, SmoothGrad

Examples

```
#----------------------- Example 1: Torch ----------------------------------
library(torch)

# Create nn_sequential model and data
model <- nn_sequential(
  nn_linear(5, 12),
  nn_relu(),
  nn_linear(12, 2),
  nn_softmax(dim = 2)
)
data <- torch_randn(25, 5)

# Create a reference dataset for the estimation of the conditional
# expectation
ref <- torch_randn(5, 5)

# Create Converter
converter <- convert(model, input_dim = c(5))

# Apply method DeepSHAP
deepshap <- DeepSHAP$new(converter, data, data_ref = ref)

# You can also use the helper function `run_deepshap` for initializing
# an R6 DeepSHAP object
deepshap <- run_deepshap(converter, data, data_ref = ref)

# Print the result as a torch tensor for first two data points
get_result(deepshap, "torch.tensor")[1:2]

# Plot the result for both classes
```

```
plot(deepshap, output_idx = 1:2)

# Plot the boxplot of all datapoints and for both classes
boxplot(deepshap, output_idx = 1:2)

# ------------------------- Example 2: Neuralnet ---------------------------
if (require("neuralnet")) {
  library(neuralnet)
  data(iris)

  # Train a neural network
  nn <- neuralnet((Species == "setosa") ~ Petal.Length + Petal.Width,
    iris,
    linear.output = FALSE,
    hidden = c(3, 2), act.fct = "tanh", rep = 1
  )

  # Convert the model
  converter <- convert(nn)

  # Apply DeepSHAP with rescale-rule and a 100 (default of `limit_ref`)
  # instances as the reference dataset
  deepshap <- run_deepshap(converter, iris[, c(3, 4)],
                           data_ref = iris[, c(3, 4)])

  # Get the result as a dataframe and show first 5 rows
  get_result(deepshap, type = "data.frame")[1:5, ]

  # Plot the result for the first datapoint in the data
  plot(deepshap, data_idx = 1)

  # Plot the result as boxplots
  boxplot(deepshap)
}


# ------------------------- Example 3: Keras -------------------------------
if (require("keras") & keras::is_keras_available()) {
  library(keras)

  # Make sure keras is installed properly
  is_keras_available()

  data <- array(rnorm(10 * 32 * 32 * 3), dim = c(10, 32, 32, 3))

  model <- keras_model_sequential()
  model %>%
    layer_conv_2d(
      input_shape = c(32, 32, 3), kernel_size = 8, filters = 8,
      activation = "softplus", padding = "valid") %>%
    layer_conv_2d(
      kernel_size = 8, filters = 4, activation = "tanh",
      padding = "same") %>%
```

```
    layer_conv_2d(
      kernel_size = 4, filters = 2, activation = "relu",
      padding = "valid") %>%
    layer_flatten() %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 16, activation = "relu") %>%
    layer_dense(units = 2, activation = "softmax")

  # Convert the model
  converter <- convert(model)

  # Apply the DeepSHAP method with zero baseline (wich is equivalent to
  # DeepLift with zero baseline)
  deepshap <- run_deepshap(converter, data, channels_first = FALSE)

  # Plot the result for the first image and both classes
  plot(deepshap, output_idx = 1:2)

  # Plot the pixel-wise median of the results
  plot_global(deepshap, output_idx = 1)
}


#------------------------- Plotly plots ------------------------------------
if (require("plotly")) {
  # You can also create an interactive plot with plotly.
  # This is a suggested package, so make sure that it is installed
  library(plotly)
  boxplot(deepshap, as_plotly = TRUE)
}
```

`ExpectedGradient` *Expected Gradients*

Description

The *Expected Gradients* method (Erion et al., 2021), also known as *GradSHAP*, is a local feature attribution technique which extends the `IntegratedGradient` method and provides approximate Shapley values. In contrast to IntegratedGradient, it considers not only a single reference value x' but the whole distribution of reference values $X' \sim x'$ and averages the IntegratedGradient values over this distribution. Mathematically, the method can be described as follows:

$$E_{x' \sim X', \alpha \sim U(0,1)}[(x - x') \times \frac{\partial f(x' + \alpha(x - x'))}{\partial x}]$$

The distribution of the reference values is specified with the argument `data_ref`, of which `n` samples are taken at random for each instance during the estimation.

The R6 class can also be initialized using the `run_expgrad` function as a helper function so that no prior knowledge of R6 classes is required.

Super classes

innsight::InterpretingMethod -> innsight::GradientBased -> ExpectedGradient

Public fields

`n (integer(1))`
: Number of samples from the distribution of reference values and number of samples for the approximation of the integration path along α (default: 50).

`data_ref (list)`
: The reference input for the ExpectedGradient method. This value is stored as a list of `torch_tensors` of shape *(, dim_in)* for each input layer.

Methods

Public methods:

- `ExpectedGradient$new()`
- `ExpectedGradient$clone()`

Method `new()`**:** Create a new instance of the `ExpectedGradient` R6 class. When initialized, the method *Expected Gradient* is applied to the given data and baseline values and the results are stored in the field `result`.

Usage:

```
ExpectedGradient$new(
  converter,
  data,
  data_ref = NULL,
  n = 50,
  channels_first = TRUE,
  output_idx = NULL,
  output_label = NULL,
  ignore_last_act = TRUE,
  verbose = interactive(),
  dtype = "float"
)
```

Arguments:

`converter` (`Converter`)
: An instance of the `Converter` class that includes the torch-converted model and some other model-specific attributes. See `Converter` for details.

`data` (`array`, `data.frame`, `torch_tensor` or `list`)
: The data to which the method is to be applied. These must have the same format as the input data of the passed model to the converter object. This means either

- an `array`, `data.frame`, `torch_tensor` or array-like format of size *(batch_size, dim_in)*, if e.g., the model has only one input layer, or

- a `list` with the corresponding input data (according to the upper point) for each of the input layers.

`data_ref` (`array`, `data.frame`, `torch_tensor` or `list`)
: The reference inputs for the ExpectedGradient method. This value must have the same format as the input data of the passed model to the converter object. This means either
  - an `array`, `data.frame`, `torch_tensor` or array-like format of size *(, dim_in)*, if e.g., the model has only one input layer, or
  - a `list` with the corresponding input data (according to the upper point) for each of the input layers.
  - It is also possible to use the default value `NULL` to take only zeros as reference input.

`n` (`integer(1)`)
: Number of samples from the distribution of reference values and number of samples for the approximation of the integration path along α (default: 50).

`channels_first` (`logical(1)`)
: The channel position of the given data (argument `data`). If `TRUE`, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. If the data has no channel axis, use the default value `TRUE`.

`output_idx` (`integer`, `list` or `NULL`)
: These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:
  - An `integer` vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., `c(1,3,4)` for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.
  - A `list` of `integer` vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of indices, e.g., `list(NULL, c(1,3))` for the first and third output node in the second output layer.
  - `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`output_label` (`character`, `factor`, `list` or `NULL`)
: These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument `output_names` in the `converter` can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:
  - A `character` vector or `factor` of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed `Converter` ob-

ject, e.g., `c("a", "c", "d")` for the first, third and fourth output node if the output names are `c("a", "b", "c", "d")`. If there are multiple output layers, the names of the output nodes from the first output layer are considered.

- A `list` of `charactor/factor` vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`ignore_last_act (logical(1))`
Set this logical value to include the last activation functions for each output layer, or not (default: `TRUE`). In practice, the last activation (especially for softmax activation) is often omitted.

`verbose (logical(1))`
This logical argument determines whether a progress bar is displayed for the calculation of the method or not. The default value is the output of the primitive R function `interactive()`.

`dtype (character(1))`
The data type for the calculations. Use either `'float'` for `torch_float` or `'double'` for `torch_double`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`ExpectedGradient$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

References

G. Erion et al. (2021) *Improving performance of deep learning models with * *axiomatic attribution priors and expected gradients.* Nature Machine Intelligence 3, pp. 620-631.

See Also

Other methods: `ConnectionWeights`, `DeepLift`, `DeepSHAP`, `Gradient`, `IntegratedGradient`, `LIME`, `LRP`, `SHAP`, `SmoothGrad`

Examples

```
#----------------------- Example 1: Torch ----------------------------------
library(torch)

# Create nn_sequential model and data
```

```
model <- nn_sequential(
  nn_linear(5, 12),
  nn_relu(),
  nn_linear(12, 2),
  nn_softmax(dim = 2)
)
data <- torch_randn(25, 5)
ref <- torch_randn(1, 5)

# Create Converter
converter <- convert(model, input_dim = c(5))

# Apply method IntegratedGradient
int_grad <- IntegratedGradient$new(converter, data, x_ref = ref)

# You can also use the helper function `run_intgrad` for initializing
# an R6 IntegratedGradient object
int_grad <- run_intgrad(converter, data, x_ref = ref)

# Print the result as a torch tensor for first two data points
get_result(int_grad, "torch.tensor")[1:2]

# Plot the result for both classes
plot(int_grad, output_idx = 1:2)

# Plot the boxplot of all datapoints and for both classes
boxplot(int_grad, output_idx = 1:2)

# ------------------------- Example 2: Neuralnet ---------------------------
if (require("neuralnet")) {
  library(neuralnet)
  data(iris)

  # Train a neural network
  nn <- neuralnet((Species == "setosa") ~ Petal.Length + Petal.Width,
    iris,
    linear.output = FALSE,
    hidden = c(3, 2), act.fct = "tanh", rep = 1
  )

  # Convert the model
  converter <- convert(nn)

  # Apply IntegratedGradient with a reference input of the feature means
  x_ref <- matrix(colMeans(iris[, c(3, 4)]), nrow = 1)
  int_grad <- run_intgrad(converter, iris[, c(3, 4)], x_ref = x_ref)

  # Get the result as a dataframe and show first 5 rows
  get_result(int_grad, type = "data.frame")[1:5, ]

  # Plot the result for the first datapoint in the data
  plot(int_grad, data_idx = 1)
```

```
  # Plot the result as boxplots
  boxplot(int_grad)
}


# ------------------------- Example 3: Keras -------------------------------
if (require("keras") & keras::is_keras_available()) {
  library(keras)

  # Make sure keras is installed properly
  is_keras_available()

  data <- array(rnorm(10 * 32 * 32 * 3), dim = c(10, 32, 32, 3))

  model <- keras_model_sequential()
  model %>%
    layer_conv_2d(
      input_shape = c(32, 32, 3), kernel_size = 8, filters = 8,
      activation = "softplus", padding = "valid") %>%
    layer_conv_2d(
      kernel_size = 8, filters = 4, activation = "tanh",
      padding = "same") %>%
    layer_conv_2d(
      kernel_size = 4, filters = 2, activation = "relu",
      padding = "valid") %>%
    layer_flatten() %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 2, activation = "softmax")

  # Convert the model
  converter <- convert(model)

  # Apply the IntegratedGradient method with a zero baseline and n = 20
  # iteration steps
  int_grad <- run_intgrad(converter, data,
    channels_first = FALSE,
    n = 20
  )

  # Plot the result for the first image and both classes
  plot(int_grad, output_idx = 1:2)

  # Plot the pixel-wise median of the results
  plot_global(int_grad, output_idx = 1)
}


#------------------------- Plotly plots ------------------------------------
if (require("plotly")) {
  # You can also create an interactive plot with plotly.
  # This is a suggested package, so make sure that it is installed
  library(plotly)
  boxplot(int_grad, as_plotly = TRUE)
```

```
}
```

`get_result` *Get the result of an interpretation method*

Description

This is a generic S3 method for the R6 method `InterpretingMethod$get_result()`. See the respective method described in `InterpretingMethod` for details.

Usage

```
get_result(x, ...)
```

Arguments

`x`	An object of the class `InterpretingMethod` including the subclasses `Gradient`, `SmoothGrad`, `LRP`, `DeepLift`, `DeepSHAP`, `IntegratedGradient`, `ExpectedGradient` and `ConnectionWeights`.
`...`	Other arguments specified in the R6 method `InterpretingMethod$get_result()`. See `InterpretingMethod` for details.

`Gradient` *Vanilla Gradient and Gradient×Input*

Description

This method computes the gradients (also known as *Vanilla Gradients*) of the outputs with respect to the input variables, i.e., for all input variable i and output class j

$$df(x)_j/dx_i.$$

If the argument `times_input` is `TRUE`, the gradients are multiplied by the respective input value (*Gradient×Input*), i.e.,

$$x_i * df(x)_j/dx_i.$$

While the vanilla gradients emphasize prediction-sensitive features, Gradient×Input is a decomposition of the output into feature-wise effects based on the first-order Taylor decomposition.

The R6 class can also be initialized using the `run_grad` function as a helper function so that no prior knowledge of R6 classes is required.

Super classes

`innsight::InterpretingMethod` -> `innsight::GradientBased` -> `Gradient`

Methods

Public methods:

- `Gradient$new()`
- `Gradient$clone()`

Method `new()`**:** Create a new instance of the `Gradient` R6 class. When initialized, the method *Gradient* or *Gradient×Input* is applied to the given data and the results are stored in the field `result`.

Usage:

```
Gradient$new(
  converter,
  data,
  channels_first = TRUE,
  output_idx = NULL,
  output_label = NULL,
  ignore_last_act = TRUE,
  times_input = FALSE,
  verbose = interactive(),
  dtype = "float"
)
```

Arguments:

`converter` (`Converter`)
: An instance of the `Converter` class that includes the torch-converted model and some other model-specific attributes. See `Converter` for details.

`data` (`array`, `data.frame`, `torch_tensor` or `list`)
: The data to which the method is to be applied. These must have the same format as the input data of the passed model to the converter object. This means either
  - an `array`, `data.frame`, `torch_tensor` or array-like format of size *(batch_size, dim_in)*, if e.g., the model has only one input layer, or
  - a `list` with the corresponding input data (according to the upper point) for each of the input layers.

`channels_first` (`logical(1)`)
: The channel position of the given data (argument `data`). If `TRUE`, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. If the data has no channel axis, use the default value `TRUE`.

`output_idx` (`integer`, `list` or `NULL`)
: These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:
  - An `integer` vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., `c(1,3,4)` for the first, third and fourth

output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.

- A `list` of `integer` vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of indices, e.g., `list(NULL, c(1,3))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`output_label` (`character`, `factor`, `list` or `NULL`)

These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument `output_names` in the `converter` can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A `character` vector or `factor` of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed `Converter` object, e.g., `c("a", "c", "d")` for the first, third and fourth output node if the output names are `c("a", "b", "c", "d")`. If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A `list` of `charactor`/`factor` vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`ignore_last_act (logical(1))`

Set this logical value to include the last activation functions for each output layer, or not (default: `TRUE`). In practice, the last activation (especially for softmax activation) is often omitted.

`times_input (logical(1))`

Multiplies the gradients with the input features. This method is called *Gradient*×*Input*.

`verbose (logical(1))`

This logical argument determines whether a progress bar is displayed for the calculation of the method or not. The default value is the output of the primitive R function `interactive()`.

`dtype (character(1))`

The data type for the calculations. Use either `'float'` for `torch_float` or `'double'` for `torch_double`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Gradient$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other methods: ConnectionWeights, DeepLift, DeepSHAP, ExpectedGradient, IntegratedGradient, LIME, LRP, SHAP, SmoothGrad

Examples

```
#----------------------- Example 1: Torch ----------------------------------
library(torch)

# Create nn_sequential model and data
model <- nn_sequential(
  nn_linear(5, 12),
  nn_relu(),
  nn_linear(12, 2),
  nn_softmax(dim = 2)
)
data <- torch_randn(25, 5)

# Create Converter with input and output names
converter <- convert(model,
  input_dim = c(5),
  input_names = list(c("Car", "Cat", "Dog", "Plane", "Horse")),
  output_names = list(c("Buy it!", "Don't buy it!"))
)

# Calculate the Gradients
grad <- Gradient$new(converter, data)

# You can also use the helper function `run_grad` for initializing
# an R6 Gradient object
grad <- run_grad(converter, data)

# Print the result as a data.frame for first 5 rows
get_result(grad, "data.frame")[1:5,]

# Plot the result for both classes
plot(grad, output_idx = 1:2)

# Plot the boxplot of all datapoints
boxplot(grad, output_idx = 1:2)

# ------------------------- Example 2: Neuralnet ---------------------------
if (require("neuralnet")) {
  library(neuralnet)
  data(iris)
```

```
  # Train a neural network
  nn <- neuralnet(Species ~ ., iris,
    linear.output = FALSE,
    hidden = c(10, 5),
    act.fct = "logistic",
    rep = 1
  )

  # Convert the trained model
  converter <- convert(nn)

  # Calculate the gradients
  gradient <- run_grad(converter, iris[, -5])

  # Plot the result for the first and 60th data point and all classes
  plot(gradient, data_idx = c(1, 60), output_idx = 1:3)

  # Calculate Gradients x Input and do not ignore the last activation
  gradient <- run_grad(converter, iris[, -5],
                       ignore_last_act = FALSE,
                       times_input = TRUE)

  # Plot the result again
  plot(gradient, data_idx = c(1, 60), output_idx = 1:3)
}


# ------------------------- Example 3: Keras -------------------------------
if (require("keras") & keras::is_keras_available()) {
  library(keras)

  # Make sure keras is installed properly
  is_keras_available()

  data <- array(rnorm(64 * 60 * 3), dim = c(64, 60, 3))

  model <- keras_model_sequential()
  model %>%
    layer_conv_1d(
      input_shape = c(60, 3), kernel_size = 8, filters = 8,
      activation = "softplus", padding = "valid") %>%
    layer_conv_1d(
      kernel_size = 8, filters = 4, activation = "tanh",
      padding = "same") %>%
    layer_conv_1d(
      kernel_size = 4, filters = 2, activation = "relu",
      padding = "valid") %>%
    layer_flatten() %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 16, activation = "relu") %>%
    layer_dense(units = 3, activation = "softmax")

  # Convert the model
```

```
  converter <- convert(model)

  # Apply the Gradient method
  gradient <- run_grad(converter, data, channels_first = FALSE)

  # Plot the result for the first datapoint and all classes
  plot(gradient, output_idx = 1:3)

  # Plot the result as boxplots for first two classes
  boxplot(gradient, output_idx = 1:2)
}


#------------------------- Plotly plots ------------------------------------
if (require("plotly")) {
  # You can also create an interactive plot with plotly.
  # This is a suggested package, so make sure that it is installed
  library(plotly)

  # Result as boxplots
  boxplot(gradient, as_plotly = TRUE)

  # Result of the second data point
  plot(gradient, data_idx = 2, as_plotly = TRUE)
}
```

GradientBased *Super class for gradient-based interpretation methods*

Description

Super class for gradient-based interpretation methods. This class inherits from `InterpretingMethod`. It summarizes all implemented gradient-based methods and provides a private function to calculate the gradients w.r.t. to the input for given data. Implemented are:

- *Vanilla Gradients* and *Gradient*×*Input* (`Gradient`)
- *Integrated Gradients* (`IntegratedGradient`)
- *SmoothGrad* and *SmoothGrad*×*Input* (`SmoothGrad`)
- *ExpectedGradients* (`ExpectedGradient`)

Super class

`innsight::InterpretingMethod` -> `GradientBased`

Public fields

`times_input` (`logical(1)`)
: This logical value indicates whether the results were multiplied by the provided input data or not.

Methods

Public methods:

- `GradientBased$new()`
- `GradientBased$clone()`

Method `new()`**:** Create a new instance of this class. When initialized, the method is applied to the given data and the results are stored in the field `result`.

Usage:

```
GradientBased$new(
  converter,
  data,
  channels_first = TRUE,
  output_idx = NULL,
  output_label = NULL,
  ignore_last_act = TRUE,
  times_input = TRUE,
  verbose = interactive(),
  dtype = "float"
)
```

Arguments:

`converter` (`Converter`)
An instance of the `Converter` class that includes the torch-converted model and some other model-specific attributes. See `Converter` for details.

`data` (`array`, `data.frame`, `torch_tensor` or `list`)
The data to which the method is to be applied. These must have the same format as the input data of the passed model to the converter object. This means either

- an `array`, `data.frame`, `torch_tensor` or array-like format of size *(batch_size, dim_in)*, if e.g., the model has only one input layer, or
- a `list` with the corresponding input data (according to the upper point) for each of the input layers.

`channels_first` (`logical(1)`)
The channel position of the given data (argument `data`). If `TRUE`, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. If the data has no channel axis, use the default value `TRUE`.

`output_idx` (`integer`, `list` or `NULL`)
These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:

- An `integer` vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., `c(1,3,4)` for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.

- A `list` of `integer` vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of indices, e.g., `list(NULL, c(1,3))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`output_label` (`character`, `factor`, `list` or `NULL`)
These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument `output_names` in the `converter` can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A `character` vector or `factor` of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed `Converter` object, e.g., `c("a", "c", "d")` for the first, third and fourth output node if the output names are `c("a", "b", "c", "d")`. If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A `list` of `charactor`/`factor` vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`ignore_last_act` (`logical(1)`)
Set this logical value to include the last activation functions for each output layer, or not (default: `TRUE`). In practice, the last activation (especially for softmax activation) is often omitted.

`times_input` (`logical(1)`
Multiplies the gradients with the input features. This method is called *Gradient*×*Input*.

`verbose` (`logical(1)`)
This logical argument determines whether a progress bar is displayed for the calculation of the method or not. The default value is the output of the primitive R function `interactive()`.

`dtype` (`character(1)`)
The data type for the calculations. Use either `'float'` for `torch_float` or `'double'` for `torch_double`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GradientBased$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

`innsight_ggplot2` *S4 class for ggplot2-based plots*

Description

The S4 class `innsight_ggplot2` visualizes the results of the methods provided from the package `innsight` using ggplot2. In addition, it allows easier analysis of the results and modification of the visualization by basic generic functions. The individual slots are for internal use only and should not be modified.

Details

This S4 class is a simple extension of a ggplot2 object that enables a more detailed analysis of the results and a way to visualize the results of models with multiple input layers (e.g., images and tabular data). The distinction between one and multiple input layers decides the behavior of this class, and this information is stored in the slot `multiplot`.

One input layer (`multiplot = FALSE`):
If the model passed to a method from the innsight package has only one input layer, the S4 class `innsight_ggplot2` is just a wrapper of a single ggplot2 object. This object is stored as a 1x1 matrix in the slot `grobs` and the slots `output_strips` and `col_dims` contain only empty lists because no second line of stripes describing the input layer is needed. Although it is an object of the class `innsight_ggplot2`, the generic function +.innsight_ggplot2 provides a ggplot2-typical usage to modify the representation. The graphical objects are simply forwarded to the ggplot2 object in `grobs` and added using ggplot2::+.gg. In addition, some generic functions are implemented to visualize or examine individual aspects of the overall plot in more detail. All available generic functions are listed below:

- `+`
- `plot`, `print` and `show` (all behave the same)
- `[`
- `[[`

Note: In this case, the generic function `[<-` is not implemented because there is only one ggplot2 object and not multiple ones.

Multiple input layers (`multiplot = TRUE`):
If the passed model has multiple input layers, a ggplot2 object is created for each data point, input layer and output node and then stored as a matrix in the slot `grobs`. During visualization, these are combined using the function `gridExtra::arrangeGrob` and corresponding strips for the output layer/node names are added at the top. The labels, column indices and theme for the extra row of strips are stored in the slots `output_strips` and `col_dims`. The strips for the input layer and the data points (if not boxplot) are created using ggplot2::facet_grid in the individual ggplot2 objects of the grob matrix. An example structure is shown below:

```
|      Output 1: Node 1       |      Output 1: Node 3       |
|   Input 1    |   Input 2    |   Input 1    |   Input 2    |
|---------------------------------------------------------|-------------
|              |              |              |              |
| grobs[1,1]   |  grobs[1,2]  | grobs[1,3]   | grobs[1,4]   | data point 1
|              |              |              |              |
|---------------------------------------------------------|-------------
|              |              |              |              |
| grobs[2,1]   |  grobs[2,2]  | grobs[2,3]   | grobs[2,4]   | data point 2
|              |              |              |              |
```

Similar to the other case, generic functions are implemented to add graphical objects from ggplot2, create the whole plot or select only specific rows/columns. The difference, however, is that each entry in each row and column is a separate ggplot2 object and can be modified individually. For example, adds + `ggplot2::xlab("X")` the x-axis label "X" to all objects and not only to those in the last row. The generic function `[<-` allows you to replace a selection of objects in `grobs` and thus, for example, to change the x-axis title only in the bottom row. All available generic functions are listed below:

- `+`
- `plot`, `print` and `show` (all behave the same)
- `[`
- `[[`
- `[<-`
- `[[<-`

Note: Since this is not a standard visualization, the suggested packages `'grid'`, `'gridExtra'` and `'gtable'` must be installed.

Slots

`grobs` The individual ggplot2 objects arranged as a matrix (see details for more information)

`multiplot` A logical value indicating whether there are multiple input layers and therefore correspondingly individual ggplot2 objects instead of one single object.

`output_strips` A list containing the labels and themes of the strips for the output nodes. This slot is only relevant if `multiplot` is `TRUE`.

`col_dims` A list of the length of `output_strips` assigning to each strip the column index of `grobs` of the associated strip.

`boxplot` A logical value indicating whether the result of individual data points or a boxplot over multiple instances is displayed.

`innsight_plotly` *S4 class for plotly-based plots*

Description

The S4 class `innsight_plotly` visualizes the results of the methods provided from the package `innsight` using plotly. In addition, it allows easier analysis of the results and modification of the visualization by basic generic functions. The individual slots are for internal use only and should not be modified.

Details

This S4 class is a simple extension of a plotly object that enables a more detailed analysis of the results and a way to visualize the results of models with multiple input layers (e.g., images and tabular data).

The overall plot is created in the following order:

1. The corresponding shapes and annotations of the slots `annotations` and `shapes` are added to each plot in `plots`. This also adds the strips at the top for the output node (or input layer) and, if necessary, on the right side for the data point.
2. Subsequently, all individual plots are combined into one plot with the help of the function plotly::subplot.
3. Lastly, the global elements from the `layout` slot are added and if there are multiple input layers (`multiplot = TRUE`), another output strip is added for the columns.

An example structure of the plot with multiple input layers is shown below:

```
|      Output 1: Node 1       |      Output 1: Node 3       |
|   Input 1    |   Input 2    |   Input 1    |   Input 2    |
|---------------------------------------------------------|-------------
|              |              |              |              |
| plots[1,1]   |  plots[1,2]  | plots[1,3]   | plots[1,4]   | data point 1
|              |              |              |              |
|---------------------------------------------------------|-------------
|              |              |              |              |
| plots[2,1]   |  plots[2,2]  | plots[2,3]   | plots[2,4]   | data point 2
|              |              |              |              |
```

Additionally, some generic functions are implemented to visualize individual aspects of the overall plot or to examine them in more detail. All available generic functions are listed below:

- `plot`, `print` and `show` (all behave the same)
- `[`
- `[[`

Slots

`plots` The individual plotly objects arranged as a matrix (see details for more information).

`shapes` A list of two lists with the names `shapes_strips` and `shapes_other`. The list `shapes_strips` contains the shapes for the strips and may not be manipulated. The other list `shapes_other` contains a matrix of the same size as `plots` and each entry contains the shapes of the corresponding plot.

annotations A list of two lists with the names annotations_strips and annotations_other. The list annotations_strips contains the annotations for the strips and may not be manipulated. The other list annotations_other contains a matrix of the same size as plots and each entry contains the annotations of the corresponding plot.

multiplot A logical value indicating whether there are multiple input layers and therefore correspondingly individual ggplot2 objects instead of one single object.

layout This list contains all global layout options, e.g. update buttons, sliders, margins etc. (see plotly::layout for more details).

col_dims A list to assign a label to the columns for the output strips.

---

innsight_sugar *Syntactic sugar for object construction*

Description

Since all methods and the preceding conversion step in the innsight package were implemented using R6 classes and these always require a call to classname$new() for initialization, the following functions are defined to shorten the construction of the corresponding R6 objects:

- convert() for Converter
- run_grad() for Gradient
- run_smoothgrad() for SmoothGrad
- run_intgrad() for IntegratedGradient
- run_expgrad() for ExpectedGradient
- run_lrp() for LRP
- run_deeplift() for DeepLift
- run_deepshap for DeepSHAP
- run_cw for ConnectionWeights
- run_lime for LIME
- run_shap for SHAP

Usage

```
# Create a new `Converter` object of the given `model`
convert(model, ...)

# Apply the `Gradient` method to the passed `data` to be explained
run_grad(converter, data, ...)

# Apply the `SmoothGrad` method to the passed `data` to be explained
run_smoothgrad(converter, data, ...)

# Apply the `IntegratedGradient` method to the passed `data` to be explained
```

```
run_intgrad(converter, data, ...)

# Apply the `ExpectedGradient` method to the passed `data` to be explained
run_expgrad(converter, data, ...)

# Apply the `LRP` method to the passed `data` to be explained
run_lrp(converter, data, ...)

# Apply the `DeepLift` method to the passed `data` to be explained
run_deeplift(converter, data, ...)

# Apply the `DeepSHAP` method to the passed `data` to be explained
run_deepshap(converter, data, ...)

# Apply the `ConnectionWeights` method (argument `data` is not always required)
run_cw(converter, ...)

# Apply the `LIME` method to explain `data` by using the dataset `data_ref`
run_lime(model, data, data_ref, ...)

# Apply the `SHAP` method to explain `data` by using the dataset `data_ref`
run_shap(model, data, data_ref, ...)
```

Arguments

`model` (`nn_sequential`, `keras_model`, `neuralnet` or `list`)
A trained neural network for classification or regression tasks to be interpreted. Only models from the following types or packages are allowed: `nn_sequential`, `keras_model`, `keras_model_sequential`, `neuralnet` or a named list (see details).
Note: For the model-agnostic methods, an arbitrary fitted model for a classification or regression task can be passed. A `Converter` object can also be passed. In order for the package to know how to make predictions with the given model, a prediction function must also be passed with the argument `pred_fun`. However, for models created by `nn_sequential`, `keras_model`, `neuralnet` or `Converter`, these have already been pre-implemented and do not need to be specified.

`...` Other arguments passed to the individual constructor functions of the methods R6 classes.

`converter` (`Converter`)
An instance of the `Converter` class that includes the torch-converted model and some other model-specific attributes. See `Converter` for details.

`data` (`array`, `data.frame`, `torch_tensor` or `list`)
The data to which the method is to be applied. These must have the same format as the input data of the passed model to the converter object. This means either

- an `array`, `data.frame`, `torch_tensor` or array-like format of size *(batch_size, dim_in)*, if e.g., the model has only one input layer, or
- a `list` with the corresponding input data (according to the upper point) for each of the input layers.

Note: For the model-agnostic methods, only models with a single input and output layer is allowed!

`data_ref` (`array`, `data.frame` or `torch_tensor`)
The dataset to which the method is to be applied. These must have the same format as the input data of the passed model and has to be either `matrix`, an `array`, a `data.frame` or a `torch_tensor`.
Note: For the model-agnostic methods, only models with a single input and output layer is allowed!

Value

R6::R6Class object of the respective type.

`IntegratedGradient` *Integrated Gradients*

Description

The `IntegratedGradient` class implements the method Integrated Gradients (Sundararajan et al., 2017), which incorporates a reference value x' (also known as baseline value) analogous to the `DeepLift` method. Integrated Gradients helps to uncover the relative importance of input features in the predictions $y = f(x)$ made by a model compared to the prediction of the reference value $y' = f(x')$. This is achieved through the following formula:

$$(x - x') \times \int_{\alpha=0}^{1} \frac{\partial f(x' + \alpha(x - x'))}{\partial x} d\alpha$$

In simpler terms, it calculates how much each feature contributes to a model's output by tracing a path from a baseline input x' to the actual input x and measuring the average gradients along that path.

Similar to the other gradient-based methods, by default the integrated gradient is multiplied by the input to get an approximate decomposition of $y - y'$. However, with the parameter `times_input` only the gradient describing the output sensitivity can be returned.

The R6 class can also be initialized using the `run_intgrad` function as a helper function so that no prior knowledge of R6 classes is required.

Super classes

`innsight::InterpretingMethod` -> `innsight::GradientBased` -> `IntegratedGradient`

Public fields

`n` (`integer(1)`)
: Number of steps for the approximation of the integration path along α (default: 50).

`x_ref` (`list`)
: The reference input for the IntegratedGradient method. This value is stored as a list of `torch_tensor`s of shape *(1, dim_in)* for each input layer.

Methods

Public methods:

- `IntegratedGradient$new()`
- `IntegratedGradient$clone()`

Method `new()`**:** Create a new instance of the `IntegratedGradient` R6 class. When initialized, the method *Integrated Gradient* is applied to the given data and baseline value and the results are stored in the field `result`.

Usage:

```
IntegratedGradient$new(
  converter,
  data,
  x_ref = NULL,
  n = 50,
  times_input = TRUE,
  channels_first = TRUE,
  output_idx = NULL,
  output_label = NULL,
  ignore_last_act = TRUE,
  verbose = interactive(),
  dtype = "float"
)
```

Arguments:

`converter` (`Converter`)
: An instance of the `Converter` class that includes the torch-converted model and some other model-specific attributes. See `Converter` for details.

`data` (`array`, `data.frame`, `torch_tensor` or `list`)
: The data to which the method is to be applied. These must have the same format as the input data of the passed model to the converter object. This means either
  - an `array`, `data.frame`, `torch_tensor` or array-like format of size *(batch_size, dim_in)*, if e.g., the model has only one input layer, or
  - a `list` with the corresponding input data (according to the upper point) for each of the input layers.

`x_ref` (`array`, `data.frame`, `torch_tensor` or `list`)
: The reference input for the IntegratedGradient method. This value must have the same format as the input data of the passed model to the converter object. This means either

- an `array`, `data.frame`, `torch_tensor` or array-like format of size *(1, dim_in)*, if e.g., the model has only one input layer, or
- a `list` with the corresponding input data (according to the upper point) for each of the input layers.
- It is also possible to use the default value `NULL` to take only zeros as reference input.

`n` (`integer(1)`)
: Number of steps for the approximation of the integration path along α (default: 50).

`times_input` (`logical(1)`)
: Multiplies the integrated gradients with the difference of the input features and the baseline values. By default, the original definition of IntegratedGradient is applied. However, by setting `times_input = FALSE` only an approximation of the integral is calculated, which describes the sensitivity of the features to the output.

`channels_first` (`logical(1)`)
: The channel position of the given data (argument `data`). If `TRUE`, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. If the data has no channel axis, use the default value `TRUE`.

`output_idx` (`integer`, `list` or `NULL`)
: These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:

- An `integer` vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., `c(1,3,4)` for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.
- A `list` of `integer` vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of indices, e.g., `list(NULL, c(1,3))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`output_label` (`character`, `factor`, `list` or `NULL`)
: These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument `output_names` in the `converter` can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A `character` vector or `factor` of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed `Converter` object, e.g., `c("a", "c", "d")` for the first, third and fourth output node if the output names are `c("a", "b", "c", "d")`. If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A `list` of `charactor`/`factor` vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`ignore_last_act (logical(1))`
Set this logical value to include the last activation functions for each output layer, or not (default: `TRUE`). In practice, the last activation (especially for softmax activation) is often omitted.

`verbose (logical(1))`
This logical argument determines whether a progress bar is displayed for the calculation of the method or not. The default value is the output of the primitive R function `interactive()`.

`dtype (character(1))`
The data type for the calculations. Use either `'float'` for `torch_float` or `'double'` for `torch_double`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
IntegratedGradient$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

M. Sundararajan et al. (2017) *Axiomatic attribution for deep networks.* ICML 2017, PMLR 70, pp. 3319-3328.

See Also

Other methods: `ConnectionWeights`, `DeepLift`, `DeepSHAP`, `ExpectedGradient`, `Gradient`, `LIME`, `LRP`, `SHAP`, `SmoothGrad`

Examples

```
#---------------------- Example 1: Torch ----------------------------------
library(torch)
```

```
# Create nn_sequential model and data
model <- nn_sequential(
  nn_linear(5, 12),
  nn_relu(),
  nn_linear(12, 2),
  nn_softmax(dim = 2)
)
data <- torch_randn(25, 5)
ref <- torch_randn(1, 5)

# Create Converter
converter <- convert(model, input_dim = c(5))

# Apply method IntegratedGradient
int_grad <- IntegratedGradient$new(converter, data, x_ref = ref)

# You can also use the helper function `run_intgrad` for initializing
# an R6 IntegratedGradient object
int_grad <- run_intgrad(converter, data, x_ref = ref)

# Print the result as a torch tensor for first two data points
get_result(int_grad, "torch.tensor")[1:2]

# Plot the result for both classes
plot(int_grad, output_idx = 1:2)

# Plot the boxplot of all datapoints and for both classes
boxplot(int_grad, output_idx = 1:2)

# ------------------------- Example 2: Neuralnet ---------------------------
if (require("neuralnet")) {
  library(neuralnet)
  data(iris)

  # Train a neural network
  nn <- neuralnet((Species == "setosa") ~ Petal.Length + Petal.Width,
    iris,
    linear.output = FALSE,
    hidden = c(3, 2), act.fct = "tanh", rep = 1
  )

  # Convert the model
  converter <- convert(nn)

  # Apply IntegratedGradient with a reference input of the feature means
  x_ref <- matrix(colMeans(iris[, c(3, 4)]), nrow = 1)
  int_grad <- run_intgrad(converter, iris[, c(3, 4)], x_ref = x_ref)

  # Get the result as a dataframe and show first 5 rows
  get_result(int_grad, type = "data.frame")[1:5, ]

  # Plot the result for the first datapoint in the data
```

```
  plot(int_grad, data_idx = 1)

  # Plot the result as boxplots
  boxplot(int_grad)
}


# ------------------------- Example 3: Keras -------------------------------
if (require("keras") & keras::is_keras_available()) {
  library(keras)

  # Make sure keras is installed properly
  is_keras_available()

  data <- array(rnorm(10 * 32 * 32 * 3), dim = c(10, 32, 32, 3))

  model <- keras_model_sequential()
  model %>%
    layer_conv_2d(
      input_shape = c(32, 32, 3), kernel_size = 8, filters = 8,
      activation = "softplus", padding = "valid") %>%
    layer_conv_2d(
      kernel_size = 8, filters = 4, activation = "tanh",
      padding = "same") %>%
    layer_conv_2d(
      kernel_size = 4, filters = 2, activation = "relu",
      padding = "valid") %>%
    layer_flatten() %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 2, activation = "softmax")

  # Convert the model
  converter <- convert(model)

  # Apply the IntegratedGradient method with a zero baseline and n = 20
  # iteration steps
  int_grad <- run_intgrad(converter, data,
    channels_first = FALSE,
    n = 20
  )

  # Plot the result for the first image and both classes
  plot(int_grad, output_idx = 1:2)

  # Plot the pixel-wise median of the results
  plot_global(int_grad, output_idx = 1)
}


#------------------------- Plotly plots ------------------------------------
if (require("plotly")) {
  # You can also create an interactive plot with plotly.
  # This is a suggested package, so make sure that it is installed
```

```
  library(plotly)
  boxplot(int_grad, as_plotly = TRUE)
}
```

`InterpretingMethod` *Super class for interpreting methods*

Description

This is a super class for all interpreting methods in the `innsight` package. Implemented are the following methods:

- *Deep Learning Important Features* (`DeepLift`)
- *Deep Shapley additive explanations* (`DeepSHAP`)
- *Layer-wise Relevance Propagation* (`LRP`)
- Gradient-based methods:
  - *Vanilla gradients* including *Gradient*×*Input* (`Gradient`)
  - Smoothed gradients including *SmoothGrad*×*Input* (`SmoothGrad`)
  - *Integrated gradients* (`IntegratedGradient`)
  - *Expected gradients* (`ExpectedGradient`)
- *Connection Weights* (global and local) (`ConnectionWeights`)
- Also some model-agnostic approaches:
  - *Local interpretable model-agnostic explanations* (`LIME`)
  - *Shapley values* (`SHAP`)

Public fields

`data (list)`
: The passed data as a `list` of `torch_tensors` in the selected data format (field `dtype`) matching the corresponding shapes of the individual input layers. Besides, the channel axis is moved to the second position after the batch size because internally only the format *channels first* is used.

`converter (Converter)`
: An instance of the `Converter` class that includes the torch-converted model and some other model-specific attributes. See `Converter` for details.

`channels_first (logical(1))`
: The channel position of the given data. If `TRUE`, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. This is especially important for layers like flatten, where the order is crucial and therefore the channels have to be moved from the internal format "channels first" back to the original format before the layer is calculated.

`dtype` (`character(1)`)
The data type for the calculations. Either `'float'` for `torch_float` or `'double'` for `torch_double`.

`ignore_last_act` (`logical(1)`)
A logical value to include the last activation functions into all the calculations, or not.

`result` (`list`)
The results of the method on the passed data. A unified list structure is used regardless of the complexity of the model: The outer list contains the individual output layers and the inner list the input layers. The results for the respective output and input layer are then stored there as torch tensors in the given data format (field `dtype`). In addition, the channel axis is moved to its original place and the last axis contains the selected output nodes for the individual output layers (see `output_idx`).
For example, the structure of the result for two output layers (output node 1 for the first and 2 and 4 for the second) and two input layers with `channels_first = FALSE` looks like this:

```
List of 2 # both output layers
  $ :List of 2 # both input layers
    ..$ : torch_tensor [batch_size, dim_in_1, channel_axis, 1]
    ..$ : torch_tensor [batch_size, dim_in_2, channel_axis, 1]
 $ :List of 2 # both input layers
    ..$ : torch_tensor [batch_size, dim_in_1, channel_axis, 2]
    ..$ : torch_tensor [batch_size, dim_in_2, channel_axis, 2]
```

`output_idx` (`list`)
This list of indices specifies the output nodes to which the method is to be applied. In the order of the output layers, the list contains the respective output nodes indices and unwanted output layers have the entry `NULL` instead of a vector of indices, e.g., `list(NULL, c(1,3))` for the first and third output node in the second output layer.

`output_label` (`list`)
This list of `factors` specifies the output nodes to which the method is to be applied. In the order of the output layers, the list contains the respective output nodes labels and unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.

`verbose` (`logical(1)`)
This logical value determines whether a progress bar is displayed for the calculation of the method or not. The default value is the output of the primitive R function `interactive()`.

`winner_takes_all` (`logical(1)`)
This logical value is only relevant for models with a MaxPooling layer. Since many zeros are produced during the backward pass due to the selection of the maximum value in the pooling kernel, another variant is implemented, which treats a MaxPooling as an AveragePooling layer in the backward pass to overcome the problem of too many zero relevances. With the default value `TRUE`, the whole upper-layer relevance is passed to the maximum value in each pooling window. Otherwise, if `FALSE`, the relevance is distributed equally among all nodes in a pooling window.

`preds` (list)
: In this field, all calculated predictions are stored as a list of `torch_tensors`. Each output layer has its own list entry and contains the respective predicted values.

`decomp_goal` (list)
: In this field, the method-specific decomposition objectives are stored as a list of `torch_tensors` for each output layer. For example, GradientxInput and LRP attempt to decompose the prediction into feature-wise additive effects. DeepLift and IntegratedGradient decompose the difference between $f(x)$ and $f(x')$. On the other hand, DeepSHAP and ExpectedGradient aim to decompose $f(x)$ minus the averaged prediction across the reference values.

Methods

Public methods:

- `InterpretingMethod$new()`
- `InterpretingMethod$get_result()`
- `InterpretingMethod$plot()`
- `InterpretingMethod$plot_global()`
- `InterpretingMethod$print()`
- `InterpretingMethod$clone()`

Method `new()`**:** Create a new instance of this super class.

Usage:

```
InterpretingMethod$new(
  converter,
  data,
  channels_first = TRUE,
  output_idx = NULL,
  output_label = NULL,
  ignore_last_act = TRUE,
  winner_takes_all = TRUE,
  verbose = interactive(),
  dtype = "float"
)
```

Arguments:

`converter` (`Converter`)
: An instance of the `Converter` class that includes the torch-converted model and some other model-specific attributes. See `Converter` for details.

`data` (`array`, `data.frame`, `torch_tensor` or `list`)
: The data to which the method is to be applied. These must have the same format as the input data of the passed model to the converter object. This means either

  - an `array`, `data.frame`, `torch_tensor` or array-like format of size *(batch_size, dim_in)*, if e.g., the model has only one input layer, or

- a `list` with the corresponding input data (according to the upper point) for each of the input layers.

`channels_first (logical(1))`
The channel position of the given data (argument `data`). If `TRUE`, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. If the data has no channel axis, use the default value `TRUE`.

`output_idx (integer, list or NULL)`
These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:

- An `integer` vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g. `c(1,3,4)` for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.
- A `list` of `integer` vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of indices, e.g. `list(NULL, c(1,3))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`output_label (character, factor, list or NULL)`
These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument `output_names` in the `converter` can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A `character` vector or `factor` of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed `Converter` object, e.g., `c("a", "c", "d")` for the first, third and fourth output node if the output names are `c("a", "b", "c", "d")`. If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A `list` of `charactor/factor` vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`ignore_last_act (logical(1))`
Set this logical value to include the last activation functions for each output layer, or not

(default: `TRUE`). In practice, the last activation (especially for softmax activation) is often omitted.

`winner_takes_all (logical(1))`
This logical argument is only relevant for models with a MaxPooling layer. Since many zeros are produced during the backward pass due to the selection of the maximum value in the pooling kernel, another variant is implemented, which treats a MaxPooling as an AveragePooling layer in the backward pass to overcome the problem of too many zero relevances. With the default value `TRUE`, the whole upper-layer relevance is passed to the maximum value in each pooling window. Otherwise, if `FALSE`, the relevance is distributed equally among all nodes in a pooling window.

`verbose (logical(1))`
This logical argument determines whether a progress bar is displayed for the calculation of the method or not. The default value is the output of the primitive R function `interactive()`.

`dtype (character(1))`
The data type for the calculations. Use either `'float'` for `torch_float` or `'double'` for `torch_double`.

Method `get_result()`**:** This function returns the result of this method for the given data either as an array (`'array'`), a torch tensor (`'torch.tensor'`, or `'torch_tensor'`) of size *(batch_size, dim_in, dim_out)* or as a data.frame (`'data.frame'`). This method is also implemented as a generic S3 function `get_result`. For a detailed description, we refer to our in-depth vignette (`vignette("detailed_overview", package = "innsight")`) or our website.

Usage:

```
InterpretingMethod$get_result(type = "array")
```

Arguments:

`type (character(1))`
The data type of the result. Use one of `'array'`, `'torch.tensor'`, `'torch_tensor'` or `'data.frame'` (default: `'array'`).

Returns: The result of this method for the given data in the chosen type.

Method `plot()`**:** This method visualizes the result of the selected method and enables a visual in-depth investigation with the help of the S4 classes `innsight_ggplot2` and `innsight_plotly`. You can use the argument `data_idx` to select the data points in the given data for the plot. In addition, the individual output nodes for the plot can be selected with the argument `output_idx`. The different results for the selected data points and outputs are visualized using the ggplot2-based S4 class `innsight_ggplot2`. You can also use the `as_plotly` argument to generate an interactive plot with `innsight_plotly` based on the plot function plotly::plot_ly. For more information and the whole bunch of possibilities, see `innsight_ggplot2` and `innsight_plotly`.

Notes:

1. For the interactive plotly-based plots, the suggested package `plotly` is required.

2. The ggplot2-based plots for models with multiple input layers are a bit more complex, therefore the suggested packages `'grid'`, `'gridExtra'` and `'gtable'` must be installed in your R session.
3. If the global *Connection Weights* method was applied, the unnecessary argument `data_idx` will be ignored.
4. The predictions, the sum of relevances, and, if available, the decomposition target are displayed by default in a box within the plot. Currently, these are not generated for `plotly` plots.

Usage:

```
InterpretingMethod$plot(
  data_idx = 1,
  output_idx = NULL,
  output_label = NULL,
  aggr_channels = "sum",
  as_plotly = FALSE,
  same_scale = FALSE,
  show_preds = TRUE
)
```

Arguments:

`data_idx` (`integer`)
: An integer vector containing the numbers of the data points whose result is to be plotted, e.g., `c(1,3)` for the first and third data point in the given data. Default: `1`. This argument will be ignored for the global *Connection Weights* method.

`output_idx` (`integer`, `list` or `NULL`)
: The indices of the output nodes for which the results is to be plotted. This can be either a `integer` vector of indices or a `list` of `integer` vectors of indices but must be a subset of the indices for which the results were calculated, i.e., a subset of `output_idx` from the initialization `new()` (see argument `output_idx` in method `new()` of this R6 class for details). By default (`NULL`), the smallest index of all calculated output nodes and output layers is used.

`output_label` (`character`, `factor`, `list` or `NULL`)
: These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument `output_names` in the `converter` can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:
  - A `character` vector or `factor` of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed `Converter` object, e.g., `c("a", "c", "d")` for the first, third and fourth output node if the output names are `c("a", "b", "c", "d")`. If there are multiple output layers, the names of the output nodes from the first output layer are considered.
  - A `list` of `charactor`/`factor` vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.

- NULL (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

aggr_channels (character(1) or function)
Pass one of 'norm', 'sum', 'mean' or a custom function to aggregate the channels, e.g., the maximum (base::max) or minimum (base::min) over the channels or only individual channels with function(x) x[1]. By default ('sum'), the sum of all channels is used.
Note: This argument is used only for 2D and 3D input data.

as_plotly (logical(1))
This logical value (default: FALSE) can be used to create an interactive plot based on the library plotly (see innsight_plotly for details).
Note: Make sure that the suggested package plotly is installed in your R session.

same_scale (logical)
A logical value that specifies whether the individual plots have the same fill scale across multiple input layers or whether each is scaled individually. This argument is only used if more than one input layer results are plotted.

show_preds (logical)
This logical value indicates whether the plots display the prediction, the sum of calculated relevances, and, if available, the targeted decomposition value. For example, in the case of GradientxInput, the goal is to obtain a decomposition of the predicted value, while for DeepLift and IntegratedGradient, the goal is the difference between the prediction and the reference value, i.e., $f(x) - f(x')$.

Returns: Returns either an innsight_ggplot2 (as_plotly = FALSE) or an innsight_plotly (as_plotly = TRUE) object with the plotted individual results.

Method plot_global()**:** This method visualizes the results of the selected method summarized as boxplots/median image and enables a visual in-depth investigation of the global behavior with the help of the S4 classes innsight_ggplot2 and innsight_plotly.
You can use the argument output_idx to select the individual output nodes for the plot. For tabular and 1D data, boxplots are created in which a reference value can be selected from the data using the ref_data_idx argument. For images, only the pixel-wise median is visualized due to the complexity. The plot is generated using the ggplot2-based S4 class innsight_ggplot2. You can also use the as_plotly argument to generate an interactive plot with innsight_plotly based on the plot function plotly::plot_ly. For more information and the whole bunch of possibilities, see innsight_ggplot2 and innsight_plotly.

Notes:

1. This method can only be used for the local *Connection Weights* method, i.e., if times_input is TRUE and data is provided.
2. For the interactive plotly-based plots, the suggested package plotly is required.
3. The ggplot2-based plots for models with multiple input layers are a bit more complex, therefore the suggested packages 'grid', 'gridExtra' and 'gtable' must be installed in your R session.

Usage:

```
InterpretingMethod$plot_global(
  output_idx = NULL,
  output_label = NULL,
  data_idx = "all",
  ref_data_idx = NULL,
  aggr_channels = "sum",
  preprocess_FUN = abs,
  as_plotly = FALSE,
  individual_data_idx = NULL,
  individual_max = 20
)
```

Arguments:

output_idx (integer, list or NULL)
The indices of the output nodes for which the results is to be plotted. This can be either a vector of indices or a list of vectors of indices but must be a subset of the indices for which the results were calculated, i.e., a subset of output_idx from the initialization new() (see argument output_idx in method new() of this R6 class for details). By default (NULL), the smallest index of all calculated output nodes and output layers is used.

output_label (character, factor, list or NULL)
These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument output_names in the converter can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A character vector or factor of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed Converter object, e.g., c("a", "c", "d") for the first, third and fourth output node if the output names are c("a", "b", "c", "d"). If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A list of charactor/factor vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry NULL instead of a vector of labels, e.g., list(NULL, c("a", "c")) for the first and third output node in the second output layer.
- NULL (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

data_idx (integer)
By default, all available data points are used to calculate the boxplot information. However, this parameter can be used to select a subset of them by passing the indices. For example, with c(1:10, 25, 26) only the first 10 data points and the 25th and 26th are used to calculate the boxplots.

ref_data_idx (integer(1) or NULL)
This integer number determines the index for the reference data point. In addition to the

boxplots, it is displayed in red color and is used to compare an individual result with the summary statistics provided by the boxplot. With the default value (NULL), no individual data point is plotted. This index can be chosen with respect to all available data, even if only a subset is selected with argument data_idx.
Note: Because of the complexity of 2D inputs, this argument is used only for tabular and 1D inputs and disregarded for 2D inputs.

aggr_channels (character(1) or function)
Pass one of 'norm', 'sum', 'mean' or a custom function to aggregate the channels, e.g., the maximum (base::max) or minimum (base::min) over the channels or only individual channels with function(x) x[1]. By default ('sum'), the sum of all channels is used.
Note: This argument is used only for 2D and 3D input data.

preprocess_FUN (function)
This function is applied to the method's result before calculating the boxplots or medians. Since positive and negative values often cancel each other out, the absolute value (abs) is used by default. But you can also use the raw results (identity) to see the results' orientation, the squared data (function(x) x^2) to weight the outliers higher or any other function.

as_plotly (logical(1))
This logical value (default: FALSE) can be used to create an interactive plot based on the library plotly (see innsight_plotly for details).
Note: Make sure that the suggested package plotly is installed in your R session.

individual_data_idx (integer or NULL)
Only relevant for a plotly plot with tabular or 1D inputs! This integer vector of data indices determines the available data points in a dropdown menu, which are drawn individually analogous to ref_data_idx only for more data points. With the default value NULL, the first individual_max data points are used.
Note: If ref_data_idx is specified, this data point will be added to those from individual_data_idx in the dropdown menu.

individual_max (integer(1))
Only relevant for a plotly plot with tabular or 1D inputs! This integer determines the maximum number of individual data points in the dropdown menu without counting ref_data_idx. This means that if individual_data_idx has more than individual_max indices, only the first individual_max will be used. A too high number can significantly increase the runtime.

Returns: Returns either an innsight_ggplot2 (as_plotly = FALSE) or an innsight_plotly (as_plotly = TRUE) object with the plotted summarized results.

Method print()**:** Print a summary of the method object. This summary contains the individual fields and in particular the results of the applied method.

Usage:

```
InterpretingMethod$print()
```

Returns: Returns the method object invisibly via `base::invisible`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
InterpretingMethod$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`LIME` *Local interpretable model-agnostic explanations (LIME)*

Description

The R6 class `LIME` calculates the feature weights of a linear surrogate of the prediction model for a instance to be explained, namely the *local interpretable model-agnostic explanations (LIME)*. It is a model-agnostic method that can be applied to any predictive model. This means, in particular, that `LIME` can be applied not only to objects of the `Converter` class but also to any other model. The only requirement is the argument `pred_fun`, which generates predictions with the model for given data. However, this function is pre-implemented for models created with `nn_sequential`, `keras_model`, `neuralnet` or `Converter`. Internally, the suggested package `lime` is utilized and applied to `data.frame`.

The R6 class can also be initialized using the `run_lime` function as a helper function so that no prior knowledge of R6 classes is required.

Note: Even signal and image data are initially transformed into a `data.frame` using `as.data.frame()` and then `lime::lime` and `lime::explain` are applied. In other words, a custom `pred_fun` may need to convert the `data.frame` back into an `array` as necessary.

Super classes

`innsight::InterpretingMethod` -> `innsight::AgnosticWrapper` -> `LIME`

Methods

Public methods:

- `LIME$new()`
- `LIME$clone()`

Method `new()`**:** Create a new instance of the `LIME` R6 class. When initialized, the method *LIME* is applied to the given data and the results are stored in the field `result`.

Usage:

```
LIME$new(
  model,
  data,
  data_ref,
  output_type = NULL,
```

```
  pred_fun = NULL,
  output_idx = NULL,
  output_label = NULL,
  channels_first = TRUE,
  input_dim = NULL,
  input_names = NULL,
  output_names = NULL,
  ...
)
```

Arguments:

`model` (any prediction model)
: A fitted model for a classification or regression task that is intended to be interpreted. A `Converter` object can also be passed. In order for the package to know how to make predictions with the given model, a prediction function must also be passed with the argument `pred_fun`. However, for models created by `nn_sequential`, `keras_model`, `neuralnet` or `Converter`, these have already been pre-implemented and do not need to be specified.

`data` (`array`, `data.frame` or `torch_tensor`)
: The individual instances to be explained by the method. These must have the same format as the input data of the passed model and has to be either `matrix`, an `array`, a `data.frame` or a `torch_tensor`. If no value is specified, all instances in the dataset `data` will be explained.
Note: For the model-agnostic methods, only models with a single input and output layer is allowed!

`data_ref` (`array`, `data.frame` or `torch_tensor`)
: The dataset to which the method is to be applied. These must have the same format as the input data of the passed model and has to be either `matrix`, an `array`, a `data.frame` or a `torch_tensor`.
Note: For the model-agnostic methods, only models with a single input and output layer is allowed!

`output_type` (`character(1)`)
: Type of the model output, i.e., either `"classification"` or `"regression"`.

`pred_fun` (`function`)
: Prediction function for the model. This argument is only needed if `model` is not a model created by `nn_sequential`, `keras_model`, `neuralnet` or `Converter`. The first argument of `pred_fun` has to be `newdata`, e.g.,
`function(newdata, ...) model(newdata)`

`output_idx` (`integer`, `list` or `NULL`)
: These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:
- An `integer` vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., `c(1,3,4)` for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.

- A `list` of `integer` vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of indices, e.g., `list(NULL, c(1,3))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`output_label` (`character`, `factor`, `list` or `NULL`)
These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument `output_names` in the `converter` can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A `character` vector or `factor` of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed `Converter` object, e.g., `c("a", "c", "d")` for the first, third and fourth output node if the output names are `c("a", "b", "c", "d")`. If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A `list` of `charactor`/`factor` vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`channels_first` (`logical(1)`)
The channel position of the given data (argument `data`). If `TRUE`, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. If the data has no channel axis, use the default value `TRUE`.

`input_dim` (`integer`)
The model input dimension excluding the batch dimension. It can be specified as vector of integers, but has to be in the format "channels first".

`input_names` (`character`, `factor` or `list`)
The input names of the model excluding the batch dimension. For a model with a single input layer and input axis (e.g., for tabular data), the input names can be specified as a character vector or factor, e.g., for a dense layer with 3 input features use `c("X1", "X2", "X3")`. If the model input consists of multiple axes (e.g., for signal and image data), use a list of character vectors or factors for each axis in the format "channels first", e.g., use `list(c("C1", "C2"), c("L1","L2","L3","L4","L5"))` for a 1D convolutional input layer with signal length 4 and 2 channels.
Note: This argument is optional and otherwise the names are generated automatically. But

if this argument is set, all found input names in the passed model will be disregarded.

`output_names` (character, factor)
A character vector with the names for the output dimensions excluding the batch dimension, e.g., for a model with 3 output nodes use `c("Y1", "Y2", "Y3")`. Instead of a character vector you can also use a factor to set an order for the plots.
Note: This argument is optional and otherwise the names are generated automatically. But if this argument is set, all found output names in the passed model will be disregarded.

`...` other arguments forwarded to `lime::explain`.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
LIME$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other methods: `ConnectionWeights`, `DeepLift`, `DeepSHAP`, `ExpectedGradient`, `Gradient`, `IntegratedGradient`, `LRP`, `SHAP`, `SmoothGrad`

Examples

```
#----------------------- Example 1: Torch ----------------------------------
library(torch)

# Create nn_sequential model and data
model <- nn_sequential(
  nn_linear(5, 12),
  nn_relu(),
  nn_linear(12, 2),
  nn_softmax(dim = 2)
  )
data <- torch_randn(25, 5)

# Calculate LIME for the first 10 instances and set the
# feature and outcome names
lime <- LIME$new(model, data[1:10, ], data_ref = data,
                 input_names = c("Car", "Cat", "Dog", "Plane", "Horse"),
                 output_names = c("Buy it!", "Don't buy it!"))

# You can also use the helper function `run_lime` for initializing
# an R6 LIME object
lime <- run_lime(model, data[1:10, ], data_ref = data,
                 input_names = c("Car", "Cat", "Dog", "Plane", "Horse"),
                 output_names = c("Buy it!", "Don't buy it!"))

# Get the result as an array for the first two instances
get_result(lime)[1:2,, ]
```

```
# Plot the result for both classes
plot(lime, output_idx = c(1, 2))

# Show the boxplot over all 10 instances
boxplot(lime, output_idx = c(1, 2))

# We can also forward some arguments to lime::explain, e.g. n_permutatuins
# to get more accurate values
lime <- run_lime(model, data[1:10, ], data_ref = data,
                 input_names = c("Car", "Cat", "Dog", "Plane", "Horse"),
                 output_names = c("Buy it!", "Don't buy it!"),
                 n_perturbations = 200)

# Plot the boxplots again
boxplot(lime, output_idx = c(1, 2))

#----------------------- Example 2: Converter object --------------------------
# We can do the same with an Converter object (all feature and outcome names
# will be extracted by the LIME method!)
conv <- convert(model,
                input_dim = c(5),
                input_names = c("Car", "Cat", "Dog", "Plane", "Horse"),
                output_names = c("Buy it!", "Don't buy it!"))

# Calculate LIME for the first 10 instances
lime <- run_lime(conv, data[1:10], data_ref = data, n_perturbations = 300)

# Plot the result for both classes
plot(lime, output_idx = c(1, 2))

#----------------------- Example 3: Other model -------------------------------
if (require("neuralnet") & require("ranger")) {
  library(neuralnet)
  library(ranger)
  data(iris)

  # Fit a random forest unsing the ranger package
  model <- ranger(Species ~ ., data = iris, probability = TRUE)

  # There is no pre-implemented predict function for ranger models, i.e.,
  # we have to define it ourselves.
  pred_fun <- function(newdata, ...) {
    predict(model, newdata, ...)$predictions
  }

  # Calculate LIME for the instances of index 1 and 111 and add
  # the outcome labels (for LIME, the output_type is required!)
  lime <- run_lime(model, iris[c(1, 111), -5],
                   data_ref = iris[, -5],
                   pred_fun = pred_fun,
                   output_type = "classification",
                   output_names = levels(iris$Species),
```

```
                 n_perturbations = 300)

  # Plot the result for the first two classes and all selected instances
  plot(lime, data_idx = 1:2, output_idx = 1:2)

  # Get the result as a torch_tensor
  get_result(lime, "torch_tensor")
}
```

LRP *Layer-wise relevance propagation (LRP)*

Description

This is an implementation of the *layer-wise relevance propagation (LRP)* algorithm introduced by Bach et al. (2015). It's a local method for interpreting a single element of the dataset and calculates the relevance scores for each input feature to the model output. The basic idea of this method is to decompose the prediction score of the model with respect to the input features, i.e.,

$$f(x) = \sum_i R(x_i).$$

Because of the bias vector that absorbs some relevance, this decomposition is generally an approximation. There exist several propagation rules to determine the relevance scores. In this package are implemented: simple rule ("simple"), ε-rule ("epsilon") and α-β-rule ("alpha_beta").

The R6 class can also be initialized using the `run_lrp` function as a helper function so that no prior knowledge of R6 classes is required.

Super class

`innsight::InterpretingMethod` -> `LRP`

Public fields

`rule_name` (`character(1)` or `list`)
: The name of the rule with which the relevance scores are calculated. Implemented are `"simple"`, `"epsilon"`, `"alpha_beta"` (and `"pass"` but only for 'BatchNorm_Layer'). However, this value can also be a named list that assigns one of these three rules to each implemented layer type separately, e.g., `list(Dense_Layer = "simple", Conv2D_Layer = "alpha_beta")`. Layers not specified in this list then use the default value `"simple"`. The implemented layer types are:

- 'Dense_Layer'
- 'Conv1D_Layer'
- 'Conv2D_Layer'
- 'BatchNorm_Layer'
- 'AvgPool1D_Layer'
- 'AvgPool2D_Layer'
- 'MaxPool1D_Layer'
- 'MaxPool2D_Layer'

`rule_param` (`numeric` or `list`)
: The parameter of the selected rule. Similar to the argument `rule_name`, this can also be a named list that assigns a rule parameter to each layer type.

Methods

Public methods:

- LRP$new()
- LRP$clone()

Method `new()`**:** Create a new instance of the `LRP` R6 class. When initialized, the method *LRP* is applied to the given data and the results are stored in the field `result`.

Usage:

```
LRP$new(
  converter,
  data,
  channels_first = TRUE,
  output_idx = NULL,
  output_label = NULL,
  ignore_last_act = TRUE,
  rule_name = "simple",
  rule_param = NULL,
  winner_takes_all = TRUE,
  verbose = interactive(),
  dtype = "float"
)
```

Arguments:

`converter` (Converter)
An instance of the `Converter` class that includes the torch-converted model and some other model-specific attributes. See Converter for details.

`data` (array, data.frame, torch_tensor or `list`)
The data to which the method is to be applied. These must have the same format as the input data of the passed model to the converter object. This means either

- an `array`, `data.frame`, `torch_tensor` or array-like format of size *(batch_size, dim_in)*, if e.g., the model has only one input layer, or
- a `list` with the corresponding input data (according to the upper point) for each of the input layers.

`channels_first` (`logical(1)`)
The channel position of the given data (argument `data`). If `TRUE`, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. If the data has no channel axis, use the default value `TRUE`.

`output_idx` (`integer`, `list` or `NULL`)
These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:

- An `integer` vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., `c(1,3,4)` for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.
- A `list` of `integer` vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of indices, e.g., `list(NULL, c(1,3))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`output_label` (`character`, `factor`, `list` or `NULL`)
These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument `output_names` in the `converter` can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A `character` vector or `factor` of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed `Converter` object, e.g., `c("a", "c", "d")` for the first, third and fourth output node if the output names are `c("a", "b", "c", "d")`. If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A `list` of `charactor`/`factor` vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`ignore_last_act` (`logical(1)`)
Set this logical value to include the last activation functions for each output layer, or not (default: `TRUE`). In practice, the last activation (especially for softmax activation) is often omitted.

`rule_name` (`character(1)` or `list`)
The name of the rule with which the relevance scores are calculated. Implemented are `"simple"`, `"epsilon"`, `"alpha_beta"`. You can pass one of the above characters to apply this rule to all possible layers. However, this value can also be a named list that assigns one of these three rules to each implemented layer type separately, e.g., `list(Dense_Layer = "simple", Conv2D_Layer = "alpha_beta")`. Layers not specified in this list then use the default value `"simple"`. The implemented layer types are:

· 'Dense_Layer' · 'Conv1D_Layer' · 'Conv2D_Layer'

- 'BatchNorm_Layer'
- 'AvgPool1D_Layer'
- 'AvgPool2D_Layer'
- 'MaxPool1D_Layer'
- 'MaxPool2D_Layer'

Note: For normalization layers like 'BatchNorm_Layer', the rule `"pass"` is implemented as well, which ignores such layers in the backward pass.

`rule_param` (`numeric(1)` or `list`)
The parameter of the selected rule. Note: Only the rules `"epsilon"` and `"alpha_beta"` take use of the parameter. Use the default value `NULL` for the default parameters ("epsilon" : 0.01, "alpha_beta" : 0.5). Similar to the argument `rule_name`, this can also be a named list that assigns a rule parameter to each layer type. If the layer type is not specified in the named list, the default parameters will be used.

`winner_takes_all (logical(1))`
This logical argument is only relevant for models with a MaxPooling layer. Since many zeros are produced during the backward pass due to the selection of the maximum value in the pooling kernel, another variant is implemented, which treats a MaxPooling as an AveragePooling layer in the backward pass to overcome the problem of too many zero relevances. With the default value `TRUE`, the whole upper-layer relevance is passed to the maximum value in each pooling window. Otherwise, if `FALSE`, the relevance is distributed equally among all nodes in a pooling window.

`verbose (logical(1))`
This logical argument determines whether a progress bar is displayed for the calculation of the method or not. The default value is the output of the primitive R function `interactive()`.

`dtype (character(1))`
The data type for the calculations. Use either `'float'` for `torch_float` or `'double'` for `torch_double`.

Returns: A new instance of the R6 class `LRP`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LRP$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

S. Bach et al. (2015) *On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation.* PLoS ONE 10, p. 1-46

See Also

Other methods: `ConnectionWeights`, `DeepLift`, `DeepSHAP`, `ExpectedGradient`, `Gradient`, `IntegratedGradient`, `LIME`, `SHAP`, `SmoothGrad`

Examples

```
 #----------------------- Example 1: Torch ----------------------------------
library(torch)

# Create nn_sequential model and data
model <- nn_sequential(
  nn_linear(5, 12),
  nn_relu(),
  nn_linear(12, 2),
  nn_softmax(dim = 2)
)
data <- torch_randn(25, 5)

# Create Converter
converter <- convert(model, input_dim = c(5))

# Apply method LRP with simple rule (default)
lrp <- LRP$new(converter, data)

# You can also use the helper function `run_lrp` for initializing
# an R6 LRP object
lrp <- run_lrp(converter, data)

# Print the result as an array for data point one and two
get_result(lrp)[1:2,,]

# Plot the result for both classes
plot(lrp, output_idx = 1:2)

# Plot the boxplot of all datapoints without a preprocess function
boxplot(lrp, output_idx = 1:2, preprocess_FUN = identity)

# ------------------------- Example 2: Neuralnet ---------------------------
if (require("neuralnet")) {
  library(neuralnet)
  data(iris)
  nn <- neuralnet(Species ~ .,
    iris,
    linear.output = FALSE,
    hidden = c(10, 8), act.fct = "tanh", rep = 1, threshold = 0.5
  )

  # Create an converter for this model
  converter <- convert(nn)

  # Create new instance of 'LRP'
  lrp <- run_lrp(converter, iris[, -5], rule_name = "simple")

  # Get the result as an array for data point one and two
  get_result(lrp)[1:2,,]

  # Get the result as a torch tensor for data point one and two
```

```
  get_result(lrp, type = "torch.tensor")[1:2]

  # Use the alpha-beta rule with alpha = 2
  lrp <- run_lrp(converter, iris[, -5],
    rule_name = "alpha_beta",
    rule_param = 2
  )

  # Include the last activation into the calculation
  lrp <- run_lrp(converter, iris[, -5],
    rule_name = "alpha_beta",
    rule_param = 2,
    ignore_last_act = FALSE
  )

  # Plot the result for all classes
  plot(lrp, output_idx = 1:3)
}


# ------------------------- Example 3: Keras -------------------------------
if (require("keras") & keras::is_keras_available()) {
  library(keras)

  # Make sure keras is installed properly
  is_keras_available()

  data <- array(rnorm(10 * 60 * 3), dim = c(10, 60, 3))

  model <- keras_model_sequential()
  model %>%
    layer_conv_1d(
      input_shape = c(60, 3), kernel_size = 8, filters = 8,
      activation = "softplus", padding = "valid") %>%
    layer_conv_1d(
      kernel_size = 8, filters = 4, activation = "tanh",
      padding = "same") %>%
    layer_conv_1d(
      kernel_size = 4, filters = 2, activation = "relu",
      padding = "valid") %>%
    layer_flatten() %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 16, activation = "relu") %>%
    layer_dense(units = 3, activation = "softmax")

  # Convert the model
  converter <- convert(model)

  # Apply the LRP method with the epsilon rule for the dense layers and
  # the alpha-beta rule for the convolutional layers
  lrp_comp <- run_lrp(converter, data,
    channels_first = FALSE,
    rule_name = list(Dense_Layer = "epsilon", Conv1D_Layer = "alpha_beta"),
```

```
    rule_param = list(Dense_Layer = 0.1, Conv1D_Layer = 1)
  )

  # Plot the result for the first datapoint and all classes
  plot(lrp_comp, output_idx = 1:3)

  # Plot the result as boxplots for first two classes
  boxplot(lrp_comp, output_idx = 1:2)
}


#------------------------- Plotly plots ------------------------------------
if (require("plotly")) {
  # You can also create an interactive plot with plotly.
  # This is a suggested package, so make sure that it is installed
  library(plotly)

  # Result as boxplots
  boxplot(lrp, as_plotly = TRUE)

  # Result of the second data point
  plot(lrp, data_idx = 2, as_plotly = TRUE)
}
```

---

plot_global *Get the result of an interpretation method*

Description

This is a generic S3 method for the R6 method `InterpretingMethod$plot_global()`. See the respective method described in `InterpretingMethod` for details.

Usage

```
plot_global(x, ...)
```

Arguments

x	An object of the class `InterpretingMethod` including the subclasses `Gradient`, `SmoothGrad`, `LRP`, `DeepLift`, `DeepSHAP`, `IntegratedGradient`, `ExpectedGradient` and `ConnectionWeights`.
...	Other arguments specified in the R6 method `InterpretingMethod$plot_global()`. See `InterpretingMethod` for details.

print,innsight_ggplot2-method
Generic print, plot and show for innsight_ggplot2

Description

The class innsight_ggplot2 provides the generic visualization functions print, plot and show, which all behave the same in this case. They create the plot of the results (see innsight_ggplot2 for details) and return it invisibly.

Usage

```
## S4 method for signature 'innsight_ggplot2'
print(x, ...)

## S4 method for signature 'innsight_ggplot2'
show(object)

## S4 method for signature 'innsight_ggplot2'
plot(x, y, ...)
```

Arguments

x	An instance of the S4 class innsight_ggplot2.
...	Further arguments passed to the base function print if x@multiplot is FALSE. Otherwise, if x@multiplot is TRUE, the arguments are passed to gridExtra::arrangeGrob.
object	An instance of the S4 class innsight_ggplot2.
y	unused argument

Value

For multiple plots (x@multiplot = TRUE), a gtable::gtable and otherwise a ggplot2::ggplot object is returned invisibly.

See Also

innsight_ggplot2, +.innsight_ggplot2, [.innsight_ggplot2, [[.innsight_ggplot2, [<-.innsight_ggplot2, [[<-.innsight_ggplot2

print,innsight_plotly-method
Generic print, plot and show for innsight_plotly

Description

The class innsight_plotly provides the generic visualization functions print, plot and show, which all behave the same in this case. They create a plot of the results using plotly::subplot (see innsight_plotly for details) and return it invisibly.

Usage

```
## S4 method for signature 'innsight_plotly'
print(x, shareX = TRUE, ...)

## S4 method for signature 'innsight_plotly'
show(object)

## S4 method for signature 'innsight_plotly'
plot(x, y, ...)
```

Arguments

x	An instance of the S4 class innsight_plotly.
shareX	A logical value whether the x-axis should be shared among the subplots.
...	Further arguments passed to plotly::subplot.
object	An instance of the S4 class innsight_plotly.
y	unused argument

SHAP *Shapley values*

Description

The R6 class SHAP calculates the famous Shapley values based on game theory for an instance to be explained. It is a model-agnostic method that can be applied to any predictive model. This means, in particular, that SHAP can be applied not only to objects of the Converter class but also to any other model. The only requirement is the argument pred_fun, which generates predictions with the model for given data. However, this function is pre-implemented for models created with nn_sequential, keras_model, neuralnet or Converter. Internally, the suggested package fastshap is utilized and applied to data.frame.

The R6 class can also be initialized using the run_shap function as a helper function so that no prior knowledge of R6 classes is required.

Note: Even signal and image data are initially transformed into a data.frame using as.data.frame() and then fastshap::explain is applied. In other words, a custom pred_fun may need to convert the data.frame back into an array as necessary.

Super classes

innsight::InterpretingMethod -> innsight::AgnosticWrapper -> SHAP

Methods

Public methods:

- SHAP$new()
- SHAP$clone()

Method new()**:** Create a new instance of the SHAP R6 class. When initialized, the method *SHAP* is applied to the given data and the results are stored in the field result.

Usage:

```
SHAP$new(
  model,
  data,
  data_ref,
  pred_fun = NULL,
  output_idx = NULL,
  output_label = NULL,
  channels_first = TRUE,
  input_dim = NULL,
  input_names = NULL,
  output_names = NULL,
  ...
)
```

Arguments:

model (any prediction model)
A fitted model for a classification or regression task that is intended to be interpreted. A Converter object can also be passed. In order for the package to know how to make predictions with the given model, a prediction function must also be passed with the argument pred_fun. However, for models created by nn_sequential, keras_model, neuralnet or Converter, these have already been pre-implemented and do not need to be specified.

data (array, data.frame or torch_tensor)
The individual instances to be explained by the method. These must have the same format as the input data of the passed model and has to be either matrix, an array, a data.frame or a torch_tensor. If no value is specified, all instances in the dataset data will be explained.
Note: For the model-agnostic methods, only models with a single input and output layer is allowed!

data_ref (array, data.frame or torch_tensor)
The dataset to which the method is to be applied. These must have the same format as the input data of the passed model and has to be either matrix, an array, a data.frame or a torch_tensor.
Note: For the model-agnostic methods, only models with a single input and output layer is

allowed!

pred_fun (function)
: Prediction function for the model. This argument is only needed if model is not a model created by nn_sequential, keras_model, neuralnet or Converter. The first argument of pred_fun has to be newdata, e.g.,

```
function(newdata, ...) model(newdata)
```

output_idx (integer, list or NULL)
: These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:

- An integer vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., c(1,3,4) for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.
- A list of integer vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry NULL instead of a vector of indices, e.g., list(NULL, c(1,3)) for the first and third output node in the second output layer.
- NULL (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

output_label (character, factor, list or NULL)
: These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument output_names in the converter can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A character vector or factor of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed Converter object, e.g., c("a", "c", "d") for the first, third and fourth output node if the output names are c("a", "b", "c", "d"). If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A list of charactor/factor vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry NULL instead of a vector of labels, e.g., list(NULL, c("a", "c")) for the first and third output node in the second output layer.
- NULL (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

channels_first (logical(1))
: The channel position of the given data (argument data). If TRUE, the channel axis is placed at the second position between the batch size and the rest of the input axes, e.g., c(10,3,32,32) for a batch of ten images with three channels and a height and width of 32

pixels. Otherwise (FALSE), the channel axis is at the last position, i.e., c(10,32,32,3). If the data has no channel axis, use the default value TRUE.

input_dim (integer)
The model input dimension excluding the batch dimension. It can be specified as vector of integers, but has to be in the format "channels first".

input_names (character, factor or list)
The input names of the model excluding the batch dimension. For a model with a single input layer and input axis (e.g., for tabular data), the input names can be specified as a character vector or factor, e.g., for a dense layer with 3 input features use c("X1", "X2", "X3"). If the model input consists of multiple axes (e.g., for signal and image data), use a list of character vectors or factors for each axis in the format "channels first", e.g., use list(c("C1", "C2"), c("L1","L2","L3","L4","L5")) for a 1D convolutional input layer with signal length 4 and 2 channels.
Note: This argument is optional and otherwise the names are generated automatically. But if this argument is set, all found input names in the passed model will be disregarded.

output_names (character, factor)
A character vector with the names for the output dimensions excluding the batch dimension, e.g., for a model with 3 output nodes use c("Y1", "Y2", "Y3"). Instead of a character vector you can also use a factor to set an order for the plots.
Note: This argument is optional and otherwise the names are generated automatically. But if this argument is set, all found output names in the passed model will be disregarded.

... other arguments forwarded to fastshap::explain.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
SHAP$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other methods: ConnectionWeights, DeepLift, DeepSHAP, ExpectedGradient, Gradient, IntegratedGradient, LIME, LRP, SmoothGrad

Examples

```
#----------------------- Example 1: Torch ----------------------------------
library(torch)

# Create nn_sequential model and data
model <- nn_sequential(
  nn_linear(5, 12),
  nn_relu(),
  nn_linear(12, 2),
```

```
  nn_softmax(dim = 2)
  )
data <- torch_randn(25, 5)

# Calculate Shapley values for the first 10 instances and set the
# feature and outcome names
shap <- SHAP$new(model, data[1:10, ], data_ref = data,
                 input_names = c("Car", "Cat", "Dog", "Plane", "Horse"),
                 output_names = c("Buy it!", "Don't buy it!"))

# You can also use the helper function `run_shap` for initializing
# an R6 SHAP object
shap <- run_shap(model, data[1:10, ], data_ref = data,
                 input_names = c("Car", "Cat", "Dog", "Plane", "Horse"),
                 output_names = c("Buy it!", "Don't buy it!"))

# Get the result as an array for the first two instances
get_result(shap)[1:2,, ]

# Plot the result for both classes
plot(shap, output_idx = c(1, 2))

# Show the boxplot over all 10 instances
boxplot(shap, output_idx = c(1, 2))

# We can also forward some arguments to fastshap::explain, e.g. nsim to
# get more accurate values
shap <- run_shap(model, data[1:10, ], data_ref = data,
                 input_names = c("Car", "Cat", "Dog", "Plane", "Horse"),
                 output_names = c("Buy it!", "Don't buy it!"),
                 nsim = 10)

# Plot the boxplots again
boxplot(shap, output_idx = c(1, 2))

#----------------------- Example 2: Converter object --------------------------
# We can do the same with an Converter object (all feature and outcome names
# will be extracted by the SHAP method!)
conv <- convert(model,
                input_dim = c(5),
                input_names = c("Car", "Cat", "Dog", "Plane", "Horse"),
                output_names = c("Buy it!", "Don't buy it!"))

# Calculate Shapley values for the first 10 instances
shap <- run_shap(conv, data[1:10], data_ref = data)

# Plot the result for both classes
plot(shap, output_idx = c(1, 2))

#----------------------- Example 3: Other model -------------------------------
if (require("neuralnet") & require("ranger")) {
  library(neuralnet)
  library(ranger)
```

```
  data(iris)

  # Fit a random forest unsing the ranger package
  model <- ranger(Species ~ ., data = iris, probability = TRUE)

  # There is no pre-implemented predict function for ranger models, i.e.,
  # we have to define it ourselves.
  pred_fun <- function(newdata, ...) {
    predict(model, newdata, ...)$predictions
  }

  # Calculate Shapley values for the instances of index 1 and 111 and add
  # the outcome labels
  shap <- run_shap(model, iris[c(1, 111), -5], data_ref = iris[, -5],
                   pred_fun = pred_fun,
                   output_names = levels(iris$Species),
                   nsim = 10)

  # Plot the result for the first two classes and all selected instances
  plot(shap, data_idx = 1:2, output_idx = 1:2)

  # Get the result as a torch_tensor
  get_result(shap, "torch_tensor")
}
```

`SmoothGrad` *SmoothGrad and SmoothGrad×Input*

Description

SmoothGrad was introduced by D. Smilkov et al. (2017) and is an extension to the classical *Vanilla Gradient* method. It takes the mean of the gradients for `n` perturbations of each data point, i.e., with $\epsilon \sim N(0, \sigma)$

$$1/n \sum_n df(x + \epsilon)_j / dx_j.$$

Analogous to the *Gradient×Input* method, you can also use the argument `times_input` to multiply the gradients by the inputs before taking the average (*SmoothGrad×Input*).

The R6 class can also be initialized using the `run_smoothgrad` function as a helper function so that no prior knowledge of R6 classes is required.

Super classes

`innsight::InterpretingMethod` -> `innsight::GradientBased` -> `SmoothGrad`

Public fields

`n` (`integer(1)`)
Number of perturbations of the input data (default: 50).

`noise_level` (`numeric(1)`)
The standard deviation of the Gaussian perturbation, i.e., $\sigma = (max(x) - min(x)) *$ `noise_level`.

Methods

Public methods:

- `SmoothGrad$new()`
- `SmoothGrad$clone()`

Method `new()`**:** Create a new instance of the `SmoothGrad` R6 class. When initialized, the method *SmoothGrad* or *SmoothGrad*$\times$*Input* is applied to the given data and the results are stored in the field `result`.

Usage:

```
SmoothGrad$new(
  converter,
  data,
  channels_first = TRUE,
  output_idx = NULL,
  output_label = NULL,
  ignore_last_act = TRUE,
  times_input = FALSE,
  n = 50,
  noise_level = 0.1,
  verbose = interactive(),
  dtype = "float"
)
```

Arguments:

`converter` (`Converter`)
An instance of the `Converter` class that includes the torch-converted model and some other model-specific attributes. See `Converter` for details.

`data` (`array`, `data.frame`, `torch_tensor` or `list`)
The data to which the method is to be applied. These must have the same format as the input data of the passed model to the converter object. This means either

- an `array`, `data.frame`, `torch_tensor` or array-like format of size *(batch_size, dim_in)*, if e.g., the model has only one input layer, or
- a `list` with the corresponding input data (according to the upper point) for each of the input layers.

`channels_first` (`logical(1)`)
The channel position of the given data (argument `data`). If `TRUE`, the channel axis is

placed at the second position between the batch size and the rest of the input axes, e.g., `c(10,3,32,32)` for a batch of ten images with three channels and a height and width of 32 pixels. Otherwise (`FALSE`), the channel axis is at the last position, i.e., `c(10,32,32,3)`. If the data has no channel axis, use the default value `TRUE`.

`output_idx` (`integer`, `list` or `NULL`)
: These indices specify the output nodes for which the method is to be applied. In order to allow models with multiple output layers, there are the following possibilities to select the indices of the output nodes in the individual output layers:

- An `integer` vector of indices: If the model has only one output layer, the values correspond to the indices of the output nodes, e.g., `c(1,3,4)` for the first, third and fourth output node. If there are multiple output layers, the indices of the output nodes from the first output layer are considered.
- A `list` of `integer` vectors of indices: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired indices of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of indices, e.g., `list(NULL, c(1,3))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`output_label` (`character`, `factor`, `list` or `NULL`)
: These values specify the output nodes for which the method is to be applied. Only values that were previously passed with the argument `output_names` in the `converter` can be used. In order to allow models with multiple output layers, there are the following possibilities to select the names of the output nodes in the individual output layers:

- A `character` vector or `factor` of labels: If the model has only one output layer, the values correspond to the labels of the output nodes named in the passed `Converter` object, e.g., `c("a", "c", "d")` for the first, third and fourth output node if the output names are `c("a", "b", "c", "d")`. If there are multiple output layers, the names of the output nodes from the first output layer are considered.
- A `list` of `charactor`/`factor` vectors of labels: If the method is to be applied to output nodes from different layers, a list can be passed that specifies the desired labels of the output nodes for each output layer. Unwanted output layers have the entry `NULL` instead of a vector of labels, e.g., `list(NULL, c("a", "c"))` for the first and third output node in the second output layer.
- `NULL` (default): The method is applied to all output nodes in the first output layer but is limited to the first ten as the calculations become more computationally expensive for more output nodes.

`ignore_last_act` (`logical(1)`)
: Set this logical value to include the last activation functions for each output layer, or not (default: `TRUE`). In practice, the last activation (especially for softmax activation) is often omitted.

`times_input` (`logical(1)`

Multiplies the gradients with the input features. This method is called *SmoothGrad*$\times$*Input*.

`n` (`integer(1)`)
Number of perturbations of the input data (default: 50).

`noise_level` (`numeric(1)`)
Determines the standard deviation of the Gaussian perturbation, i.e., $\sigma = (max(x) - min(x)) *$ `noise_level`.

`verbose` (`logical(1)`)
This logical argument determines whether a progress bar is displayed for the calculation of the method or not. The default value is the output of the primitive R function `interactive()`.

`dtype` (`character(1)`)
The data type for the calculations. Use either `'float'` for `torch_float` or `'double'` for `torch_double`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SmoothGrad$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

D. Smilkov et al. (2017) *SmoothGrad: removing noise by adding noise.* CoRR, abs/1706.03825

See Also

Other methods: `ConnectionWeights`, `DeepLift`, `DeepSHAP`, `ExpectedGradient`, `Gradient`, `IntegratedGradient`, `LIME`, `LRP`, `SHAP`

Examples

```
# ------------------------- Example 1: Torch -------------------------------
library(torch)

# Create nn_sequential model and data
model <- nn_sequential(
  nn_linear(5, 10),
  nn_relu(),
  nn_linear(10, 2),
  nn_sigmoid()
)
data <- torch_randn(25, 5)

# Create Converter
converter <- convert(model, input_dim = c(5))
```

```
# Calculate the smoothed Gradients
smoothgrad <- SmoothGrad$new(converter, data)

# You can also use the helper function `run_smoothgrad` for initializing
# an R6 SmoothGrad object
smoothgrad <- run_smoothgrad(converter, data)

# Print the result as a data.frame for first 5 rows
head(get_result(smoothgrad, "data.frame"), 5)

# Plot the result for both classes
plot(smoothgrad, output_idx = 1:2)

# Plot the boxplot of all datapoints
boxplot(smoothgrad, output_idx = 1:2)

# ------------------------- Example 2: Neuralnet ---------------------------
if (require("neuralnet")) {
  library(neuralnet)
  data(iris)

  # Train a neural network
  nn <- neuralnet(Species ~ ., iris,
    linear.output = FALSE,
    hidden = c(10, 5),
    act.fct = "logistic",
    rep = 1
  )

  # Convert the trained model
  converter <- convert(nn)

  # Calculate the smoothed gradients
  smoothgrad <- run_smoothgrad(converter, iris[, -5], times_input = FALSE)

  # Plot the result for the first and 60th data point and all classes
  plot(smoothgrad, data_idx = c(1, 60), output_idx = 1:3)

  # Calculate SmoothGrad x Input and do not ignore the last activation
  smoothgrad <- run_smoothgrad(converter, iris[, -5], ignore_last_act = FALSE)

  # Plot the result again
  plot(smoothgrad, data_idx = c(1, 60), output_idx = 1:3)
}


# ------------------------- Example 3: Keras -------------------------------
if (require("keras") & keras::is_keras_available()) {
  library(keras)

  # Make sure keras is installed properly
  is_keras_available()
```

```
  data <- array(rnorm(64 * 60 * 3), dim = c(64, 60, 3))

  model <- keras_model_sequential()
  model %>%
    layer_conv_1d(
      input_shape = c(60, 3), kernel_size = 8, filters = 8,
      activation = "softplus", padding = "valid") %>%
    layer_conv_1d(
      kernel_size = 8, filters = 4, activation = "tanh",
      padding = "same") %>%
    layer_conv_1d(
      kernel_size = 4, filters = 2, activation = "relu",
      padding = "valid") %>%
    layer_flatten() %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 16, activation = "relu") %>%
    layer_dense(units = 3, activation = "softmax")

  # Convert the model
  converter <- convert(model)

  # Apply the SmoothGrad method
  smoothgrad <- run_smoothgrad(converter, data, channels_first = FALSE)

  # Plot the result for the first datapoint and all classes
  plot(smoothgrad, output_idx = 1:3)

  # Plot the result as boxplots for first two classes
  boxplot(smoothgrad, output_idx = 1:2)
}


#------------------------- Plotly plots ------------------------------------
if (require("plotly")) {
  # You can also create an interactive plot with plotly.
  # This is a suggested package, so make sure that it is installed
  library(plotly)

  # Result as boxplots
  boxplot(smoothgrad, as_plotly = TRUE)

  # Result of the second data point
  plot(smoothgrad, data_idx = 2, as_plotly = TRUE)
}
```

[,innsight_ggplot2-method

Indexing plots of `innsight_ggplot2`

Description

The S4 class innsight_ggplot2 visualizes the results in the form of a matrix, with the output nodes (and also the input layers) in the columns and the selected data points in the rows. With these basic generic indexing functions, the plots of individual rows and columns can be accessed, modified and the overall plot can be adjusted accordingly.

Usage

```
## S4 method for signature 'innsight_ggplot2'
x[i, j, ..., restyle = TRUE, drop = TRUE]

## S4 method for signature 'innsight_ggplot2'
x[[i, j, ..., restyle = TRUE]]

## S4 replacement method for signature 'innsight_ggplot2'
x[i, j, ...] <- value

## S4 replacement method for signature 'innsight_ggplot2'
x[[i, j, ...]] <- value
```

Arguments

x	An instance of the S4 class innsight_ggplot2.
i	The numeric (or missing) index for the rows.
j	The numeric (or missing) index for the columns.
...	other unused arguments
restyle	This logical value determines whether the labels and facet stripes remain as they were in the original plot or are adjusted to the subplot accordingly. However, this argument is only used if the innsight_ggplot2 instance is a multiplot, i.e., x@multiplot is TRUE.
drop	unused argument
value	Another instance of the S4 class innsight_ggplot2 but of shape i x j.

Value

- [.innsight_ggplot2: Selects only the plots from the i-th rows and j-th columns and returns them as a new instance of innsight_ggplot2. If restyle = TRUE the facet stripes and axis labels of the original plot are transferred to the subplot, otherwise they are returned as they are.
- [[.innsight_ggplot2: Selects only the subplot in row i and column j and returns it as a ggplot2::ggplot object. If restyle = TRUE the facet stripes and axis labels of the original plot are transferred to the subplot, otherwise they are returned as they are.
- [<-.innsight_ggplot2: Replaces the plots in the rows i and columns j with those from value and returns the modified instance of innsight_ggplot2.
- [[<-.innsight_ggplot2: Replaces the plot from the i-th row and j-th column with the plot from value and returns the modified instance of innsight_ggplot2.

See Also

innsight_ggplot2, print.innsight_ggplot2, +.innsight_ggplot2

[,innsight_plotly-method
Indexing plots of innsight_plotly

Description

The S4 class innsight_plotly visualizes the results as a matrix of plots based on plotly::plot_ly. The output nodes (and also input layers) are displayed in the columns and the selected data points in the rows. With these basic generic indexing functions, the plots of individual rows and columns can be accessed.

Usage

```
## S4 method for signature 'innsight_plotly'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'innsight_plotly'
x[[i, j, ..., drop]]
```

Arguments

x	An instance of the S4 class innsight_plotly.
i	The numeric (or missing) index for the rows.
j	The numeric (or missing) index for the columns.
...	other unused arguments
drop	unused argument

Value

- [.innsight_plotly: Selects the plots from the i-th rows and j-th columns and returns them as a new instance of innsight_plotly.
- [[.innisght_plotly: Selects only the single plot in the i-th row and j-th column and returns it as a plotly object.

See Also

innsight_plotly, print.innsight_plotly, plot.innsight_plotly, show.innsight_plotly

Index