Package 'innsight'

November 22, 2021

Type Package

Title Get the Insights of your Neural Network

Version 0.1.0

Description Interpretability methods to analyze the behavior and individual predictions of modern neural networks. Implemented methods are: '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) <arXiv:1704.02685> and gradient-based methods like 'SmoothGrad' described by Smilkov et al. (2017) <arXiv:1706.03825>, 'Gradient x Input' described by Baehrens et al. (2009) <arXiv:0912.1128> or 'Vanilla Gradient'.

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, ggplot2, R6, torch

Suggests keras, knitr, neuralnet, plotly, rmarkdown, tensorflow, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.1.2

Collate 'ConnectionWeights.R' 'Convert_keras.R' 'Convert_neuralnet.R' 'Convert_torch.R' 'Converter.R' 'DeepLift.R' 'GradienBased.R' 'InterpretingLayer.R' 'InterpretingMethod.R' 'LRP.R' 'Layer_conv1d.R' 'Layer_conv2d.R' 'Layer_dense.R' 'Layer_other.R' 'Layer_pooling.R' 'innsight.R' 'utils.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 2021-11-22 09:50:14 UTC

R topics documented:

`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' 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 reveal-cancel rule
- Gradient-based methods:
  - Vanilla Gradient, including 'Gradient x Input'

  - Smoothed gradients (SmoothGrad), including 'SmoothGrad x Input'
- ConnectionWeights

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).

Author(s)

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

Other contributors:

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

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/

ConnectionWeights *Connection Weights Method*

Description

This class implements the *Connection Weights* method investigated by Olden et al. (2004) which results in a feature relevance score for each input variable. The basic idea is to multiply up 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 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

$$W_1 * W_2 * W_3.$$

Public fields

`converter` The converter of class Converter with the stored and torch-converted model.

`channels_first` The data format of the result, i.e. channels on last dimension (`FALSE`) or on the first dimension (`TRUE`). If the data has no channels, use the default value `TRUE`.

`dtype` The type of the data and parameters (either `'float'` for torch::torch_float or `'double'` for torch::torch_double).

`result` The methods result as a torch tensor of size *(dim_in, dim_out)* and with data type `dtype`.

`output_idx` This vector determines for which outputs the method will be applied. By default (`NULL`), all outputs (but limited to the first 10) are considered.

Methods

Public methods:

- ConnectionWeights$new()
- ConnectionWeights$get_result()
- ConnectionWeights$plot()
- ConnectionWeights$clone()

Method `new()`**:**

Usage:

```
ConnectionWeights$new(
  converter,
  output_idx = NULL,
  channels_first = TRUE,
  dtype = "float"
)
```

Arguments:

`converter` The converter of class Converter with the stored and torch-converted model.

`output_idx` This vector determines for which output indices the method will be applied. By default (`NULL`), all outputs (but limited to the first 10) are considered.

`channels_first` The data format of the result, i.e. channels on last dimension (`FALSE`) or on the first dimension (`TRUE`). If the data has no channels, use the default value `TRUE`.

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

Method `get_result()`**:** This function returns the result of the *Connection Weights* method either as an array (`'array'`), a torch tensor (`'torch.tensor'` or `'torch_tensor'`) of size (dim_in, dim_out) or as a data.frame (`'data.frame'`).

Usage:

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

Arguments:

`type` 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 *ConnectionWeights* method in a ggplot2::ggplot. You can use the argument `output_idx` to select individual output nodes for the plot. The different results for the selected outputs are visualized using the method ggplot2::facet_grid. You can also use the `as_plotly` argument to generate an interactive plot based on the plot function plotly::plot_ly.

Usage:

```
ConnectionWeights$plot(
  output_idx = NULL,
  aggr_channels = "sum",
  preprocess_FUN = identity,
  as_plotly = FALSE
)
```

Arguments:

`output_idx` An integer vector containing the numbers of the output indices whose result is to be plotted, e.g. `c(1,4)` for the first and fourth model output. But this vector must be included in the vector `output_idx` from the initialization, otherwise, no results were calculated for this output node and can not be plotted. By default (`NULL`), the smallest index of all calculated output nodes is used.

`aggr_channels` 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 inputs.

`preprocess_FUN` This function is applied to the method's result before generating the plot. By default, the identity function (`identity`) is used.

`as_plotly` This boolean value (default: `FALSE`) can be used to create an interactive plot based on the library `plotly`. This function takes use of plotly::ggplotly, hence make sure that the suggested package `plotly` is installed in your R session.
Advanced: You can first output the results as a ggplot (`as_plotly = FALSE`) and then make custom changes to the plot, e.g. other theme or other fill color. Then you can manually call the function `ggplotly` to get an interactive plotly plot.

Returns: Returns either a ggplot2::ggplot (`as_plotly = FALSE`) or a plotly::plot_ly object (`as_plotly = TRUE`) with the plotted results.

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

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"))
)

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

# Print the result as a data.frame
cw$get_result("data.frame")

# Plot the result
plot(cw)

#----------------------- Example 2: 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 <- Converter$new(nn)

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

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

# Plot the result
plot(cw)

#----------------------- Example 3: Keras ---------------------------------
library(keras)

if (is_keras_available()) {
  # Define a model
  model <- keras_model_sequential()
  model %>%
    layer_conv_1d(
      input_shape = c(64, 3), kernel_size = 16, filters = 8,
      activation = "softplus"
    ) %>%
    layer_conv_1d(kernel_size = 16, filters = 4, activation = "tanh") %>%
    layer_conv_1d(kernel_size = 16, filters = 2, activation = "relu") %>%
    layer_flatten() %>%
    layer_dense(units = 64, activation = "relu") %>%
```

```
    layer_dense(units = 2, activation = "softmax")

  # Convert the model
  converter <- Converter$new(model)

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

  # Get the result as data.frame
  cw$get_result(type = "data.frame")

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

# ------------------------- Advanced: Plotly -------------------------------
# If you want to create an interactive plot of your results with custom
# changes, you can take use of the method plotly::ggplotly
library(ggplot2)
library(plotly)
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 <- Converter$new(nn)

# create new instance of 'LRP'
cw <- ConnectionWeights$new(converter)

library(plotly)

# Get the ggplot and add your changes
p <- plot(cw, output_idx = 1) +
  theme_bw() +
  scale_fill_gradient2(low = "green", mid = "black", high = "blue")

# Now apply the method plotly::ggplotly with argument tooltip = "text"
plotly::ggplotly(p, tooltip = "text")
```

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, dtype = "float")
```

Arguments

`modules_list`	A list of all accepted layers created by the 'Converter' class during initialization.
`dtype`	The data type for all the calculations and defined tensors. Use either `'float'` for torch::torch_float or `'double'` for torch::torch_double.

Public fields

`modules_list` A list of all accepted layers created by the 'Converter' class during initialization.

`dtype` The datatype for all the calculations and defined tensors. 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)
```

Arguments:

`x` The input torch tensor of dimensions *(batch_size, dim_in)*.

`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`.

Return:

Returns the output of the model with respect to the given inputs with dimensions *(batch_size, dim_out)*.

Method `update_ref()`

This method updates the stored intermediate values in each module from the list `modules_list` when the reference input `x_ref` has changed.

Usage:

```
self$update_ref(x_ref,channels_first = TRUE)
```

Arguments:

`x_ref` Reference input of the model of dimensions *(1, dim_in)*.

`channels_first` If the reference 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`.

Return:

Returns the output of the reference input with dimension *(1, dim_out)* after passing through the model.

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 details for more information).

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 `innsight::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_dict` 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)
- SmoothGrad including 'SmoothGrad x Input', Smilkov et al. (2017)
- Vanilla Gradient including 'Gradient x Input'
- ConnectionWeights, Olden et al. (2004)

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 with the following components:

- $input_dim
  An integer vector with the model input dimension, e.g. for a dense layer with 5 input features use `c(5)` or for a 1D-convolutional layer with signal length 50 and 4 channels use `c(4,50)`.
- $input_names (optional)
  A list with the names for each input dimension, e.g. for a dense layer with 3 input features use `list(c("X1","X2","X3"))` or for a 1D-convolutional layer with signal length 5 and 2 channels use `list(c("C1","C2"),c("L1","L2","L3","L4","L5"))`. By default (`NULL`) the names are generated.
- $output_dim (optional)
  An integer vector with the model output dimension analogous to $input_dim. This value does not need to be specified. But if it is set, the calculated value will be compared with it to avoid errors during converting.
- $output_names (optional)
  A list with the names for each output dimension analogous to $input_names. By default (`NULL`) the names are generated.
- $layers
  A list with the respective layers of the model. Each layer is represented as another list that requires the following entries depending on the type:
  - **Dense Layer:**
    - $type: `'Dense'`
    - $weight: The weight matrix of the dense layer with shape (`dim_out`, `dim_in`).
    - $bias: The bias vector of the dense layer with length `dim_out`.
    - `activation_name`: The name of the activation function for this dense layer, e.g. `'relu'`, `'tanh'` or `'softmax'`.
    - `dim_in` (optional): The input dimension of this layer. This value is not necessary, but helpful to check the format of the weight matrix.
    - `dim_out` (optional): The output dimension of this layer. This value is not necessary, but helpful to check the format of the weight matrix.
  - **Convolutional Layers:**
    - $type: `'Conv1D'` or `'Conv2D'`
    - $weight: The weight array of the convolutional layer with shape (`out_channels`, `in_channels`, `kernel_length`) for 1D or (`out_channels`, `in_channels`, `kernel_height`, `kernel_width`) for 2D.

- $bias: The bias vector of the layer with length `out_channels`.
- $activation_name: The name of the activation function for this layer, e.g. `'relu'`, `'tanh'` or `'softmax'`.
- $dim_in (optional): The input dimension of this layer according to the format (`in_channels`, `in_length`) for 1D or (`in_channels`, `in_height`, `in_width`) for 2D.
- $dim_out (optional): The output dimension of this layer according to the format (`out_channels`, `out_length`) for 1D or (`out_channels`, `out_height`, `out_width`) for 2D.
- $stride (optional): The stride of the convolution (single integer for 1D and tuple of two integers for 2D). If this value is not specified, the default values (1D: `1` and 2D: `c(1,1)`) are used.
- $padding (optional): Zero-padding added to the sides of the input before convolution. For 1D-convolution a tuple of the form (`pad_left`, `pad_right`) and for 2D-convolution (`pad_left`, `pad_right`, `pad_top`, `pad_bottom`) is required. If this value is not specified, the default values (1D: `c(0,0)` and 2D: `c(0,0,0,0)`) are used.
- $dilation (optional): Spacing between kernel elements (single integer for 1D and tuple of two integers for 2D). If this value is not specified, the default values (1D: `1` and 2D: `c(1,1)`) are used.

– **Pooling Layers:**

- $type: `'MaxPooling1D'`, `'MaxPooling2D'`, `'AveragePooling1D'` or `'AveragePooling2D'`
- $kernel_size: The size of the pooling window as an integer value for 1D-pooling and an tuple of two integers for 2D-pooling.
- $strides (optional): The stride of the pooling window (single integer for 1D and tuple of two integers for 2D). If this value is not specified (`NULL`), the value of `kernel_size` will be used.
- `dim_in` (optional): The input dimension of this layer. This value is not necessary, but helpful to check the correctness of the converted model.
- `dim_out` (optional): The output dimension of this layer. This value is not necessary, but helpful to check the correctness of the converted model.

– **Flatten Layer:**

- $type: `'Flatten'`
- $dim_in (optional): The input dimension of this layer without the batch dimension.
- $dim_out (optional): The output dimension of this layer without the batch dimension.

Note: This package works internally only with the data format 'channels first', i.e. all input dimensions and weight matrices must be adapted accordingly.

Public fields

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

`model_dict` The model stored in a named list (see details for more information). By default, the entry `model_dict$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$new()
- Converter$clone()

Method `new()`**:** Create a new Converter for a given neural network.

Usage:

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

Arguments:

`model` 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` An integer vector with the model input dimension excluding the batch dimension, e.g. for a dense layer with `5` input features use `c(5)` or for a 1D convolutional layer with signal length `50` and `4` channels use `c(4,50)`.
Note: This argument is only necessary for `torch::nn_sequential`, for all others it is automatically extracted from the passed model. In addition, the input dimension `input_dim` has to be in the format channels first.

`input_names` (Optional) A list with the names for each input dimension, e.g. for a dense layer with `3` input features use `list(c("X1","X2","X3"))` or for a 1D convolutional layer with signal length `5` and `2` channels use `list(c("C1","C2"),c("L1","L2","L3","L4","L5"))`.
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` (Optional) A list with the names for the output, e.g. for a model with `3` outputs use `list(c("Y1","Y2","Y3"))`.
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` The data type for the calculations. Use either `'float'` for torch::torch_float or `'double'` for torch::torch_double.

`save_model_as_list` This logical value specifies whether the passed model should be stored as a list (as it is described in the details also as an alternative input for a model). 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 `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
- 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

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))

# Get the converted model
converted_model <- converter$model

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


#----------------------- Example 2: 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 <- Converter$new(nn)

# Print all the layers
converter$model$modules_list
```

```
#---------------------- Example 3: Keras ---------------------------------
library(keras)

if (is_keras_available()) {
  # Define a keras model
  model <- keras_model_sequential()
  model %>%
    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 <- Converter$new(model, save_model_as_list = TRUE)

  # Print the converted model as a named list
  str(converter$model_dict)
}

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

# Define a model

model <- list()
model$input_dim <- 5
model$input_names <- list(c("Feat1", "Feat2", "Feat3", "Feat4", "Feat5"))
model$output_dim <- 2
model$output_names <- list(c("Cat", "no-Cat"))
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
  )
model$layers$Layer_2 <-
  list(
    type = "Dense",
    weight = matrix(rnorm(20 * 2), 2, 20),
    bias = rnorm(2),
```

```
    activation_name = "softmax"#,
    #dim_in = 20, # These values are optional, but
    #dim_out = 2  # useful for internal checks
  )

# Convert the model
converter <- Converter$new(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) Method*

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: *Rescale-Rule* (`'rescale'`) and *RevealCancel-Rule* (`'reveal_cancel'`).

Super class

`innsight::InterpretingMethod` -> `DeepLift`

Public fields

`x_ref` The reference input of size *(1, dim_in)* for the interpretation.

`rule_name` Name of the applied rule to calculate the contributions for the non-linear part of a neural network layer. Either `"rescale"` or `"reveal_cancel"`.

Methods

Public methods:

- DeepLift$new()
- DeepLift$plot()
- DeepLift$boxplot()
- DeepLift$clone()

Method `new()`**:** Create a new instance of the DeepLift method.

Usage:

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

Arguments:

`converter` An instance of the R6 class `Converter`.

`data` The data for which the contribution scores are to be calculated. It has to be an array or array-like format of size *(batch_size, dim_in)*.

`channels_first` The format of the given date, i.e. channels on last dimension (`FALSE`) or after the batch dimension (`TRUE`). If the data has no channels, use the default value `TRUE`.

`output_idx` This vector determines for which outputs the method will be applied. By default (`NULL`), all outputs (but limited to the first 10) are considered.

`ignore_last_act` Set this boolean value to include the last activation, or not (default: `TRUE`). In some cases, the last activation leads to a saturation problem.

`rule_name` Name of the applied rule to calculate the contributions. Use one of `'rescale'` and `'reveal_cancel'`.

`x_ref` The reference input of size *(1, dim_in)* for the interpretation. With the default value `NULL` you use an input of zeros.

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

Method `plot()`**:** This method visualizes the result of the selected method in a ggplot2::ggplot. 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 method ggplot2::facet_grid. You can also use the `as_plotly` argument to generate an interactive plot based on the plot function plotly::plot_ly.

Usage:

```
DeepLift$plot(
  data_idx = 1,
  output_idx = NULL,
  aggr_channels = "sum",
  as_plotly = FALSE
)
```

Arguments:

`data_idx` 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: `c(1)`.

`output_idx` An integer vector containing the numbers of the output indices whose result is to be plotted, e.g. `c(1,4)` for the first and fourth model output. But this vector must be included in the vector `output_idx` from the initialization, otherwise, no results were calculated for this output node and can not be plotted. By default (`NULL`), the smallest index of all calculated output nodes is used.

`aggr_channels` 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 inputs.

`as_plotly` This boolean value (default: `FALSE`) can be used to create an interactive plot based on the library `plotly`. This function takes use of plotly::ggplotly, hence make sure that the suggested package `plotly` is installed in your R session.
Advanced: You can first output the results as a ggplot (`as_plotly = FALSE`) and then make custom changes to the plot, e.g. other theme or other fill color. Then you can manually call the function `ggplotly` to get an interactive plotly plot.

Returns: Returns either a ggplot2::ggplot (`as_plotly = FALSE`) or a plotly::plot_ly (`as_plotly = TRUE`) with the plotted results.

Method `boxplot()`**:** This function visualizes the results of this method in a boxplot, where the type of visualization depends on the input dimension of the data. By default a ggplot2::ggplot is returned, but with the argument `as_plotly` an interactive plotly::plot_ly plot can be created, which however requires a successful installation of the package `plotly`.

Usage:

```
DeepLift$boxplot(
  output_idx = NULL,
  data_idx = "all",
  ref_data_idx = NULL,
  aggr_channels = "norm",
  preprocess_FUN = abs,
  as_plotly = FALSE,
  individual_data_idx = NULL,
  individual_max = 20
)
```

Arguments:

`output_idx` An integer vector containing the numbers of the output indices whose result is to be plotted, e.g. `c(1,4)` for the first and fourth model output. But this vector must

be included in the vector `output_idx` from the initialization, otherwise, no results were calculated for this output node and can not be plotted. By default (`NULL`), the smallest index of all calculated output nodes is used.

`data_idx` By default ("all"), all available data is used to calculate the boxplot information. However, this parameter can be used to select a subset of them by passing the indices. E.g. with `data_idx = 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` 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 3D inputs, this argument is used only for 1D and 2D inputs and disregarded for 3D inputs.

`aggr_channels` 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 (`'norm'`), the Euclidean norm of all channels is used.
Note: This argument is used only for 2D and 3D inputs.

`preprocess_FUN` This function is applied to the method's result before calculating the boxplots. 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 data (`identity`) to see the results' orientation, the squared data (`function(x) x^2`) to weight the outliers higher or any other function.

`as_plotly` This boolean value (default: `FALSE`) can be used to create an interactive plot based on the library `plotly` instead of `ggplot2`. Make sure that the suggested package `plotly` is installed in your R session.

`individual_data_idx` Only relevant for a `plotly` plot with input dimension `1` or `2`! This integer vector of data indices determines the available data points in a dropdown menu, which are drawn in 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` Only relevant for a `plotly` plot with input dimension `1` or `2`! 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 a ggplot2::ggplot (`as_plotly = FALSE`) or a plotly::plot_ly (`as_plotly = TRUE`) with the boxplots.

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

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 <- Converter$new(model, input_dim = c(5))

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

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

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

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

# ------------------------- Example 2: 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 <- Converter$new(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 <- DeepLift$new(converter, iris[, c(3, 4)], x_ref = x_ref)

# Get the result as a dataframe and show first 5 rows
deeplift_rescale$get_result(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 -------------------------------
library(keras)

if (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 <- Converter$new(model)

  # Apply the DeepLift method with reveal-cancel rule
  deeplift_revcancel <- DeepLift$new(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 result as boxplots for first class
  boxplot(deeplift_revcancel, output_idx = 1)

  # 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_revcancel, as_plotly = TRUE)
}

# ------------------------- Advanced: Plotly -------------------------------
# If you want to create an interactive plot of your results with custom
# changes, you can take use of the method plotly::ggplotly
library(ggplot2)
library(neuralnet)
library(plotly)
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 <- Converter$new(nn)

# create new instance of 'DeepLift'
deeplift <- DeepLift$new(converter, iris[, -5])

# Get the ggplot and add your changes
p <- plot(deeplift, output_idx = 1, data_idx = 1:2) +
  theme_bw() +
  scale_fill_gradient2(low = "green", mid = "black", high = "blue")

# Now apply the method plotly::ggplotly with argument tooltip = "text"
plotly::ggplotly(p, tooltip = "text")
```

`Gradient` *Vanilla Gradient Method*

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 x Input'), i.e.

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

Super classes

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

Methods

Public methods:

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

Method `new()`**:** Create a new instance of the Vanilla Gradient method.

Usage:

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

Arguments:

`converter` An instance of the R6 class `Converter`.

`data` The data for which the gradients are to be calculated. It has to be an array or array-like format of size *(batch_size, dim_in)*.

`channels_first` The format of the given data, i.e. channels on last dimension (`FALSE`) or after the batch dimension (`TRUE`). If the data has no channels, use the default value `TRUE`.

`output_idx` This vector determines for which outputs the method will be applied. By default (`NULL`), all outputs (but limited to the first 10) are considered.

`ignore_last_act` A boolean value to include the last activation into all the calculations, or not (default: `TRUE`). In some cases, the last activation leads to a saturation problem.

`times_input` Multiplies the gradients with the input features. This method is called 'Gradient x Input'. Default: `FALSE`.

`dtype` The data type for the calculations. Use either `'float'` for torch::torch_float or `'double'` for torch::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.

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 <- Converter$new(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)

# Print the result as a data.frame for first 5 rows
grad$get_result("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 ---------------------------
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 <- Converter$new(nn)

# Calculate the gradients
gradient <- Gradient$new(converter, iris[, -5], times_input = TRUE)

# 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 <- Gradient$new(converter, iris[, -5], ignore_last_act = FALSE)

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

# ------------------------- Example 3: Keras -------------------------------
```

```
library(keras)

if (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 <- Converter$new(model)

  # Apply the Gradient method
  gradient <- Gradient$new(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)

  # 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)
}

# ------------------------ Advanced: Plotly -------------------------------
# If you want to create an interactive plot of your results with custom
# changes, you can take use of the method plotly::ggplotly
library(ggplot2)
library(plotly)
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 <- Converter$new(nn)

# create new instance of 'Gradient'
gradient <- Gradient$new(converter, iris[, -5])

library(plotly)

# Get the ggplot and add your changes
p <- plot(gradient, output_idx = 1, data_idx = 1:2) +
  theme_bw() +
  scale_fill_gradient2(low = "green", mid = "black", high = "blue")

# Now apply the method plotly::ggplotly with argument tooltip = "text"
plotly::ggplotly(p, tooltip = "text")
```

`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 x Input' (Gradient)
- 'SmoothGrad' and 'SmoothGrad x Input' (SmoothGrad)

Super class

`innsight::InterpretingMethod` -> `GradientBased`

Public fields

`times_input` Multiplies the gradients with the input features. This method is called 'Gradient x Input'.

Methods

Public methods:

- GradientBased$new()
- GradientBased$plot()
- GradientBased$boxplot()
- GradientBased$clone()

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

Usage:

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

Arguments:

`converter` The converter of class Converter with the stored and torch-converted model.

`data` The given data in an array-like format to be interpreted with the selected gradient-based method.

`channels_first` The format of the given date, i.e. channels on last dimension (`FALSE`) or after the batch dimension (`TRUE`). If the data has no channels, use the default value `TRUE`.

`output_idx` This vector determines for which outputs the method will be applied. By default (`NULL`), all outputs (but limited to the first 10) are considered.

`ignore_last_act` Set this boolean value to include the last activation, or not (default: `TRUE`). In some cases, the last activation leads to a saturation problem.

`times_input` Multiplies the gradients with the input features. This method is called 'Gradient x Input'.

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

Method `plot()`**:** This method visualizes the result of the selected method in a ggplot2::ggplot. 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 method ggplot2::facet_grid. You can also use the `as_plotly` argument to generate an interactive plot based on the plot function plotly::plot_ly.

Usage:

```
GradientBased$plot(
  data_idx = 1,
  output_idx = NULL,
  aggr_channels = "sum",
  as_plotly = FALSE
)
```

Arguments:

`data_idx` 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: `c(1)`.

`output_idx` An integer vector containing the numbers of the output indices whose result is to be plotted, e.g. `c(1,4)` for the first and fourth model output. But this vector must be included in the vector `output_idx` from the initialization, otherwise, no results were calculated for this output node and can not be plotted. By default (`NULL`), the smallest index of all calculated output nodes is used.

`aggr_channels` 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 inputs.

`as_plotly` This boolean value (default: `FALSE`) can be used to create an interactive plot based on the library `plotly`. This function takes use of plotly::ggplotly, hence make sure that the suggested package `plotly` is installed in your R session.
Advanced: You can first output the results as a ggplot (`as_plotly = FALSE`) and then make custom changes to the plot, e.g. other theme or other fill color. Then you can manually call the function `ggplotly` to get an interactive plotly plot.

Returns: Returns either a ggplot2::ggplot (`as_plotly = FALSE`) or a plotly::plot_ly (`as_plotly = TRUE`) with the plotted results.

Method `boxplot()`**:** This function visualizes the results of this method in a boxplot, where the type of visualization depends on the input dimension of the data. By default a ggplot2::ggplot is returned, but with the argument `as_plotly` an interactive plotly::plot_ly plot can be created, which however requires a successful installation of the package `plotly`.

Usage:

```
GradientBased$boxplot(
  output_idx = NULL,
  data_idx = "all",
  ref_data_idx = NULL,
  aggr_channels = "norm",
  preprocess_FUN = abs,
  as_plotly = FALSE,
  individual_data_idx = NULL,
  individual_max = 20
)
```

Arguments:

`output_idx` An integer vector containing the numbers of the output indices whose result is to be plotted, e.g. `c(1,4)` for the first and fourth model output. But this vector must be included in the vector `output_idx` from the initialization, otherwise, no results were calculated for this output node and can not be plotted. By default (`NULL`), the smallest index of all calculated output nodes is used.

`data_idx` By default ("all"), all available data is used to calculate the boxplot information. However, this parameter can be used to select a subset of them by passing the indices. E.g. with `data_idx = 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` 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 3D inputs, this argument is used only for 1D and 2D inputs and disregarded for 3D inputs.

`aggr_channels` 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 (`'norm'`), the Euclidean norm of all channels is used.
Note: This argument is used only for 2D and 3D inputs.

`preprocess_FUN` This function is applied to the method's result before calculating the boxplots. 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 data (`identity`) to see the results' orientation, the squared data (`function(x) x^2`) to weight the outliers higher or any other function.

`as_plotly` This boolean value (default: `FALSE`) can be used to create an interactive plot based on the library `plotly` instead of `ggplot2`. Make sure that the suggested package `plotly` is installed in your R session.

`individual_data_idx` Only relevant for a `plotly` plot with input dimension `1` or `2`! This integer vector of data indices determines the available data points in a dropdown menu, which are drawn in 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` Only relevant for a `plotly` plot with input dimension `1` or `2`! 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 a ggplot2::ggplot (`as_plotly = FALSE`) or a plotly::plot_ly (`as_plotly = TRUE`) with the boxplots.

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.

`InterpretingMethod` *Super class for Interpreting Methods*

Description

This is a super class for all data-based interpreting methods. Implemented are the following methods:

- Deep Learning Important Features (DeepLift)
- Layer-wise Relevance Propagation (LRP)
- Gradient-based methods:
  - Vanilla gradients including 'Gradients x Input' (Gradient)
  - Smoothed gradients including 'SmoothGrad x Input' (SmoothGrad)

Public fields

`data` The passed data as a torch tensor in the given data type (`dtype`) to be interpreted with the selected method.

`converter` An instance of the R6 class `Converter`.

`dtype` The data type for the calculations. Either `'float'` for torch::torch_float or `'double'` for torch::torch_double.

`channels_first` The format of the given date, i.e. channels on last dimension (`FALSE`) or after the batch dimension (`TRUE`). If the data has no channels, the default value `TRUE` is used.

`ignore_last_act` A boolean value to include the last activation into all the calculations, or not (default: `TRUE`). In some cases, the last activation leads to a saturation problem.

`result` The methods result of the given data as a torch tensor of size *(batch_size, dim_in, dim_out)* in the given data type (`dtype`).

`output_idx` This vector determines for which outputs the method will be applied. By default (`NULL`), all outputs (but limited to the first 10) are considered.

Methods

Public methods:

- `InterpretingMethod$new()`
- `InterpretingMethod$get_result()`
- `InterpretingMethod$clone()`

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

Usage:

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

Arguments:

converter An instance of the R6 class Converter.

data The data for which this method is to be applied. It has to be an array or array-like format of size *(batch_size, dim_in)*.

channels_first The format of the given data, i.e. channels on last dimension (FALSE) or after the batch dimension (TRUE). If the data has no channels, use the default value TRUE.

output_idx This vector determines for which output indices the method will be applied. By default (NULL), all outputs (but limited to the first 10) are considered.

ignore_last_act A boolean value to include the last activation into all the calculations, or not (default: TRUE). In some cases, the last activation leads to a saturation problem.

dtype dtype The data type for the calculations. Use either 'float' for torch::torch_float or 'double' for torch::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').

Usage:

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

Arguments:

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

LRP *Layer-wise Relevance Propagation (LRP) Method*

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"), epsilon rule ("epsilon") and alpha-beta rule ("alpha_beta").

Super class

innsight::InterpretingMethod -> LRP

Public fields

rule_name The name of the rule with which the relevance scores are calculated. Implemented are "simple", "epsilon", "alpha_beta" (default: "simple").

rule_param The parameter of the selected rule.

Methods

Public methods:

- LRP$new()
- LRP$plot()
- LRP$boxplot()
- LRP$clone()

Method new(): Create a new instance of the LRP-Method.

Usage:

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

Arguments:

converter An instance of the R6 class Converter.

data The data for which the relevance scores are to be calculated. It has to be an array or array-like format of size *(batch_size, dim_in)*.

channels_first The format of the given date, i.e. channels on last dimension (FALSE) or after the batch dimension (TRUE). If the data has no channels, use the default value TRUE.

output_idx This vector determines for which outputs the method will be applied. By default (NULL), all outputs (but limited to the first 10) are considered.

ignore_last_act Set this boolean value to include the last activation, or not (default: TRUE). In some cases, the last activation leads to a saturation problem.

rule_name The name of the rule, with which the relevance scores are calculated. Implemented are "simple", "epsilon", "alpha_beta" (default: "simple").

rule_param 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).

dtype The data type for the calculations. Use either 'float' for torch::torch_float or 'double' for torch::torch_double.

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

Method `plot()`**:** This method visualizes the result of the selected method in a ggplot2::ggplot. 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 method ggplot2::facet_grid. You can also use the `as_plotly` argument to generate an interactive plot based on the plot function plotly::plot_ly.

Usage:

```
LRP$plot(
  data_idx = 1,
  output_idx = NULL,
  aggr_channels = "sum",
  as_plotly = FALSE
)
```

Arguments:

`data_idx` 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: `c(1)`.

`output_idx` An integer vector containing the numbers of the output indices whose result is to be plotted, e.g. `c(1,4)` for the first and fourth model output. But this vector must be included in the vector `output_idx` from the initialization, otherwise, no results were calculated for this output node and can not be plotted. By default (`NULL`), the smallest index of all calculated output nodes is used.

`aggr_channels` 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 inputs.

`as_plotly` This boolean value (default: `FALSE`) can be used to create an interactive plot based on the library `plotly`. This function takes use of plotly::ggplotly, hence make sure that the suggested package `plotly` is installed in your R session.
Advanced: You can first output the results as a ggplot (`as_plotly = FALSE`) and then make custom changes to the plot, e.g. other theme or other fill color. Then you can manually call the function `ggplotly` to get an interactive plotly plot.

Returns: Returns either a ggplot2::ggplot (`as_plotly = FALSE`) or a plotly::plot_ly (`as_plotly = TRUE`) with the plotted results.

Method `boxplot()`**:** This function visualizes the results of this method in a boxplot, where the type of visualization depends on the input dimension of the data. By default a ggplot2::ggplot is returned, but with the argument `as_plotly` an interactive plotly::plot_ly plot can be created, which however requires a successful installation of the package `plotly`.

Usage:

```
LRP$boxplot(
  output_idx = NULL,
  data_idx = "all",
  ref_data_idx = NULL,
  aggr_channels = "norm",
```

```
  preprocess_FUN = abs,
  as_plotly = FALSE,
  individual_data_idx = NULL,
  individual_max = 20
)
```

Arguments:

`output_idx` An integer vector containing the numbers of the output indices whose result is to be plotted, e.g. `c(1,4)` for the first and fourth model output. But this vector must be included in the vector `output_idx` from the initialization, otherwise, no results were calculated for this output node and can not be plotted. By default (`NULL`), the smallest index of all calculated output nodes is used.

`data_idx` By default ("all"), all available data is used to calculate the boxplot information. However, this parameter can be used to select a subset of them by passing the indices. E.g. with `data_idx = 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` 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 3D inputs, this argument is used only for 1D and 2D inputs and disregarded for 3D inputs.

`aggr_channels` 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 (`'norm'`), the Euclidean norm of all channels is used.
Note: This argument is used only for 2D and 3D inputs.

`preprocess_FUN` This function is applied to the method's result before calculating the boxplots. 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 data (`identity`) to see the results' orientation, the squared data (`function(x) x^2`) to weight the outliers higher or any other function.

`as_plotly` This boolean value (default: `FALSE`) can be used to create an interactive plot based on the library `plotly` instead of `ggplot2`. Make sure that the suggested package `plotly` is installed in your R session.

`individual_data_idx` Only relevant for a `plotly` plot with input dimension `1` or `2`! This integer vector of data indices determines the available data points in a dropdown menu, which are drawn in 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` Only relevant for a `plotly` plot with input dimension `1` or `2`! 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 a ggplot2::ggplot (`as_plotly = FALSE`) or a plotly::plot_ly (`as_plotly = TRUE`) with the boxplots.

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

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 <- Converter$new(model, input_dim = c(5))

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

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

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

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

# ------------------------- Example 2: 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 <- Converter$new(nn)
```

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

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

# get the result as a torch tensor for data point one and two
lrp$get_result(type = "torch.tensor")[1:2]

# use the alpha-beta rule with alpha = 2
lrp <- LRP$new(converter, iris[, -5],
  rule_name = "alpha_beta",
  rule_param = 2
)

# include the last activation into the calculation
lrp <- LRP$new(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)

# Plot the Boxplot for the first class
boxplot(lrp)

# 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)

# ------------------------- Example 3: Keras -------------------------------
library(keras)

if (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 <- Converter$new(model)

# Apply the LRP method with the epsilon rule and eps = 0.1
lrp_eps <- LRP$new(converter, data,
  channels_first = FALSE,
  rule_name = "epsilon",
  rule_param = 0.1
)

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

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

# 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_eps, as_plotly = TRUE)

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

# ------------------------- Advanced: Plotly -------------------------------
# If you want to create an interactive plot of your results with custom
# changes, you can take use of the method plotly::ggplotly
library(ggplot2)
library(plotly)
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 <- Converter$new(nn)
```

```
# create new instance of 'LRP'
lrp <- LRP$new(converter, iris[, -5])

library(plotly)

# Get the ggplot and add your changes
p <- plot(lrp, output_idx = 1, data_idx = 1:2) +
  theme_bw() +
  scale_fill_gradient2(low = "green", mid = "black", high = "blue")

# Now apply the method plotly::ggplotly with argument tooltip = "text"
plotly::ggplotly(p, tooltip = "text")
```

SmoothGrad *SmoothGrad Method*

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 $\epsilon \sim N(0, \sigma)$

$$1/n \sum_{n} df(x+\epsilon)_j/dx_j.$$

Super classes

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

Public fields

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

`noise_level` 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 method.

Usage:

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

Arguments:

`converter` An instance of the R6 class `Converter`.

`data` The data for which the smoothed gradients are to be calculated. It has to be an array or array-like format of size *(batch_size, dim_in)*.

`channels_first` The format of the given data, i.e. channels on last dimension (`FALSE`) or after the batch dimension (`TRUE`). If the data has no channels, use the default value `TRUE`.

`output_idx` This vector determines for which outputs the method will be applied. By default (`NULL`), all outputs (but limited to the first 10) are considered.

`ignore_last_act` A boolean value to include the last activation into all the calculations, or not (default: `TRUE`). In some cases, the last activation leads to a saturation problem.

`times_input` Multiplies the gradients with the input features. This method is called 'SmoothGrad x Input'. Default: `FALSE`.

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

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

`dtype` The data type for the calculations. Use either `'float'` for torch::torch_float or `'double'` for torch::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

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 <- Converter$new(model, input_dim = c(5))

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

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

# 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 ---------------------------
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 <- Converter$new(nn)

# Calculate the smoothed gradients
smoothgrad <- SmoothGrad$new(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 <- SmoothGrad$new(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 -------------------------------
library(keras)

if (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 <- Converter$new(model)

  # Apply the SmoothGrad method
  smoothgrad <- SmoothGrad$new(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)

  # 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)
}

# ------------------------ Advanced: Plotly -------------------------------
# If you want to create an interactive plot of your results with custom
# changes, you can take use of the method plotly::ggplotly
library(ggplot2)
library(plotly)
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 <- Converter$new(nn)

# create new instance of 'SmoothGrad'
smoothgrad <- SmoothGrad$new(converter, iris[, -5])

library(plotly)

# Get the ggplot and add your changes
p <- plot(smoothgrad, output_idx = 1, data_idx = 1:2) +
  theme_bw() +
  scale_fill_gradient2(low = "green", mid = "black", high = "blue")

# Now apply the method plotly::ggplotly with argument tooltip = "text"
plotly::ggplotly(p, tooltip = "text")
```

Index