Package: nn2poly (via r-universe)

Title Neural Network Weights Transformation into Polynomial
Coefficients

Version 0.1.1

Description Implements a method that builds the coefficients of a
polynomial model that performs almost equivalently as a given
neural network (densely connected). This is achieved using
Taylor expansion at the activation functions. The obtained
polynomial coefficients can be used to explain features (and
their interactions) importance in the neural network, therefore
working as a tool for interpretability or eXplainable
Artificial Intelligence (XAI). See Morala et al. 2021
<doi:10.1016/j.neunet.2021.04.036>, and 2023
<doi:10.1109/TNNLS.2023.3330328>.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 3.5.0)

Imports Rcpp, generics, matrixStats, pracma

Suggests keras, tensorflow, reticulate, luz, torch, cowplot, ggplot2,
patchwork, testthat (>= 3.0.0), vdiffr, knitr, rmarkdown

LinkingTo Rcpp, RcppArmadillo

VignetteBuilder knitr

RoxygenNote 7.2.3

Config/testthat/edition 3

URL https://ibidat.github.io/nn2poly/

NeedsCompilation yes

Author Pablo Morala [aut, cre]
(<https://orcid.org/0000-0002-4109-2330>), Iñaki Ucar [aut]
(<https://orcid.org/0000-0001-6403-5550>), Jose Ignacio Diez
[ctr]

Maintainer Pablo Morala <moralapablo@gmail.com>

Repository CRAN

Date/Publication 2024-01-30 09:10:02 UTC

Contents

---

add_constraints   *Add constraints to a neural network*

Description

This function sets up a neural network object with the constraints required by the [nn2poly](#) algorithm. Currently supported neural network frameworks are `keras/tensorflow` and `luz/torch`.

Usage

```
add_constraints(object, type = c("l1_norm", "l2_norm"), ...)
```

Arguments

object	A neural network object in sequential form from one of the supported frameworks.
type	Constraint type. Currently, `l1_norm` and `l2_norm` are supported.
...	Additional arguments (unused).

Details

Constraints are added to the model object using callbacks in their specific framework. These callbacks are used during training when calling fit on the model. Specifically we are using callbacks that are applied at the end of each train batch.

Models in `luz/torch` need to use the [luz_model_sequential](#) helper in order to have a sequential model in the appropriate form.

Value

A `nn2poly` neural network object.

See Also

[luz_model_sequential()](#)

Examples

```
## Not run:
if (requireNamespace("keras", quietly=TRUE)) {
  # ---- Example with a keras/tensorflow network ----
  # Build a small nn:
  nn <- keras::keras_model_sequential()
  nn <- keras::layer_dense(nn, units = 10, activation = "tanh", input_shape = 2)
  nn <- keras::layer_dense(nn, units = 1, activation = "linear")

  # Add constraints
  nn_constrained <- add_constraints(nn, constraint_type = "l1_norm")

  # Check that class of the constrained nn is "nn2poly"
  class(nn_constrained)[1]
}

if (requireNamespace("luz", quietly=TRUE)) {
  # ---- Example with a luz/torch network ----

  # Build a small nn
  nn <- luz_model_sequential(
    torch::nn_linear(2,10),
    torch::nn_tanh(),
    torch::nn_linear(10,1)
  )

  # With luz/torch we need to setup the nn before adding the constraints
  nn <- luz::setup(module = nn,
    loss = torch::nn_mse_loss(),
    optimizer = torch::optim_adam,
  )

  # Add constraints
  nn <- add_constraints(nn)

  # Check that class of the constrained nn is "nn2poly"
  class(nn)[1]
}

## End(Not run)
```

---

eval_poly	*Polynomial evaluation*

Description

Evaluates one or several polynomials on the given data.

Usage

```
eval_poly(poly, newdata)
```

Arguments

poly   List containing 2 items: `labels` and `values`.

                             • `labels`: List of integer vectors with same length (or number of cols) as
                               `values`, where each integer vector denotes the combination of variables as-
                               sociated to the coefficient value stored at the same position in `values`. That
                               is, the monomials in the polynomial. Note that the variables are numbered
                               from 1 to p, with the intercept is represented by 0.

                             • `values`: Matrix (can also be a vector if single polynomial), where each
                               column represents a polynomial, with same number of rows as the length of
                               `labels`, containing at each row the value of the coefficient of the monomial
                               given by the equivalent label in that same position.

                   Example: If `labels` contains the integer vector $c(1,1,3)$ at position 5, then
                   the value stored in `values` at row 5 is the coefficient associated with the term
                   x_1^2*x_3.

newdata   Input data as matrix, vector or dataframe. Number of columns (or elements
                   in vector) should be the number of variables in the polynomial (dimension p).
                   Response variable to be predicted should not be included.

Details

Note that this function is unstable and subject to change. Therefore it is not exported but this docu-
mentations is left available so users can use it if needed to simulate data by using nn2poly:::eval_poly()

Value

Returns a matrix containing the evaluation of the polynomials. Each column corresponds to each
polynomial used and each row to each observation, meaning that each column vector corresponds
to the results of evaluating all the given data for each polynomial.

See Also

eval_poly() is also used in predict.nn2poly().

---

  luz_model_sequential   *Build a* luz *model composed of a linear stack of layers*

Description

Helper function to build luz models as a sequential model, by feeding it a stack of luz layers.

Usage

```
luz_model_sequential(...)
```

Arguments

...   Sequence of modules to be added.

Details

This step is needed so we can get the activation functions and layers and neurons architecture easily with nn2poly:::get_parameters(). Furthermore, this step is also needed to be able to impose the needed constraints when using the luz/torch framework.

Value

A nn_sequential module.

See Also

add_constraints()

Examples

```
## Not run:
if (requireNamespace("luz", quietly=TRUE)) {
# Create a NN using luz/torch as a sequential model
# with 3 fully connected linear layers,
# the first one with input = 5 variables,
# 100 neurons and tanh activation function, the second
# one with 50 neurons and softplus activation function
# and the last one with 1 linear output.
nn <- luz_model_sequential(
  torch::nn_linear(5,100),
  torch::nn_tanh(),
  torch::nn_linear(100,50),
  torch::nn_softplus(),
  torch::nn_linear(50,1)
)

nn

# Check that the nn is of class nn_squential
class(nn)
}

## End(Not run)
```

---

nn2poly   *Obtain polynomial representation*

Description

Implements the main NN2Poly algorithm to obtain a polynomial representation of a trained neural network using its weights and Taylor expansion of its activation functions.

Usage

```
nn2poly(
  object,
  max_order = 2,
  keep_layers = FALSE,
  taylor_orders = 8,
  ...,
  all_partitions = NULL
)
```

Arguments

object
:   An object for which the computation of the NN2Poly algorithm is desired. Currently supports models from the following deep learning frameworks:

    * tensorflow/keras models built as a sequential model.
    * torch/luz models built as a sequential model.

    It also supports a named list as input which allows to introduce by hand a model from any other source. This list should be of length L (number of hidden layers + 1) containing the weights matrix for each layer. Each element of the list should be named as the activation function used at each layer. Currently supported activation functions are "tanh", "softplus", "sigmoid" and "linear".

    At any layer l, the expected shape of such matrices is of the form $(h_{(l-1)} + 1) * (h_l)$, that is, the number of rows is the number of neurons in the previous layer plus the bias vector, and the number of columns is the number of neurons in the current layer L. Therefore, each column corresponds to the weight vector affecting each neuron in that layer. The bias vector should be in the first row.

max_order
:   integer that determines the maximum order that will be forced in the final polynomial, discarding terms of higher order that would naturally arise when considering all Taylor expansions allowed by taylor_orders.

keep_layers
:   Boolean that determines if all polynomials computed in the internal layers have to be stored and given in the output (TRUE), or if only the polynomials from the last layer are needed (FALSE). Default set to FALSE.

taylor_orders
:   integer or vector of length L that sets the degree at which Taylor expansion is truncated at each layer. If a single value is used, that value is set for each non linear layer and 1 for linear at each layer activation function. Default set to 8.

...	Ignored.
all_partitions	Optional argument containing the needed multipartitions as list of lists of lists. If set to NULL, nn2poly will compute said multipartitions. This step can be computationally expensive when the chosen polynomial order or the dimension are too high. In such cases, it is encouraged that the multipartitions are stored and reused when possible. Default set to NULL.

Value

Returns an object of class nn2poly.

If keep_layers = FALSE (default case), it returns a list with two items:

- An item named labels that is a list of integer vectors. Those vectors represent each monomial in the polynomial, where each integer in the vector represents each time one of the original variables appears in that term. As an example, vector c(1,1,2) represents the term $x_1^2 x_2$. Note that the variables are numbered from 1 to p, with the intercept is represented by

  1.

- An item named values which contains a matrix in which each column contains the coefficients of the polynomial associated with an output neuron. That is, if the neural network has a single output unit, the matrix values will have a single column and if it has multiple output units, the matrix values will have several columns. Each row will be the coefficient associated with the label in the same position in the labels list.

If keep_layers = TRUE, it returns a list of length the number of layers (represented by layer_i), where each one is another list with input and output elements. Each of those elements contains an item as explained before. The last layer output item will be the same element as if keep_layers = FALSE.

The polynomials obtained at the hidden layers are not needed to represent the NN but can be used to explore other insights from the NN.

See Also

Predict method for nn2poly output predict.nn2poly().

Examples

```
# Build a NN estructure with random weights, with 2 (+ bias) inputs,
# 4 (+bias) neurons in the first hidden layer with "tanh" activation
# function, 4 (+bias) neurons in the second hidden layer with "softplus",
# and 1 "linear" output unit

weights_layer_1 <- matrix(rnorm(12), nrow = 3, ncol = 4)
weights_layer_2 <- matrix(rnorm(20), nrow = 5, ncol = 4)
weights_layer_3 <- matrix(rnorm(5), nrow = 5, ncol = 1)

# Set it as a list with activation functions as names
nn_object = list("tanh" = weights_layer_1,
                 "softplus" = weights_layer_2,
```

```
                        "linear" = weights_layer_3)

# Obtain the polynomial representation (order = 3) of that neural network
final_poly <- nn2poly(nn_object, max_order = 3)

# Change the last layer to have 3 outputs (as in a multiclass classification)
# problem
weights_layer_4 <- matrix(rnorm(20), nrow = 5, ncol = 4)

# Set it as a list with activation functions as names
nn_object = list("tanh" = weights_layer_1,
                 "softplus" = weights_layer_2,
                 "linear" = weights_layer_4)
# Obtain the polynomial representation of that neural network
# In this case the output is formed by several polynomials with the same
# structure but different coefficient values
final_poly <- nn2poly(nn_object, max_order = 3)

# Polynomial representation of each hidden neuron is given by
final_poly <- nn2poly(nn_object, max_order = 3, keep_layers = TRUE)
```

---

plot.nn2poly   *Plot method for* nn2poly *objects.*

Description

A function that takes a polynomial (or several ones) as given by the **nn2poly** algorithm, and then plots their absolute magnitude as barplots to be able to compare the most important coefficients.

Usage

```
## S3 method for class 'nn2poly'
plot(x, ..., n = NULL)
```

Arguments

x	A nn2poly object, as returned by the **nn2poly** algorithm.
...	Ignored.
n	An integer denoting the number of coefficients to be plotted, after ordering them by absolute magnitude.

Details

The plot method represents only the polynomials at the final layer, even if x is generated using nn2poly() with keep_layers=TRUE.

Value

A plot showing the n most important coefficients.

Examples

```
# --- Single polynomial output ---
# Build a NN structure with random weights, with 2 (+ bias) inputs,
# 4 (+bias) neurons in the first hidden layer with "tanh" activation
# function, 4 (+bias) neurons in the second hidden layer with "softplus",
# and 2 "linear" output units

weights_layer_1 <- matrix(rnorm(12), nrow = 3, ncol = 4)
weights_layer_2 <- matrix(rnorm(20), nrow = 5, ncol = 4)
weights_layer_3 <- matrix(rnorm(5), nrow = 5, ncol = 1)

# Set it as a list with activation functions as names
nn_object = list("tanh" = weights_layer_1,
                 "softplus" = weights_layer_2,
                 "linear" = weights_layer_3)

# Obtain the polynomial representation (order = 3) of that neural network
final_poly <- nn2poly(nn_object, max_order = 3)

# Plot all the coefficients, one plot per output unit
plot(final_poly)

# Plot only the 5 most important coeffcients (by absolute magnitude)
# one plot per output unit
plot(final_poly, n = 5)

# --- Multiple output polynomials ---
# Build a NN structure with random weights, with 2 (+ bias) inputs,
# 4 (+bias) neurons in the first hidden layer with "tanh" activation
# function, 4 (+bias) neurons in the second hidden layer with "softplus",
# and 2 "linear" output units

weights_layer_1 <- matrix(rnorm(12), nrow = 3, ncol = 4)
weights_layer_2 <- matrix(rnorm(20), nrow = 5, ncol = 4)
weights_layer_3 <- matrix(rnorm(10), nrow = 5, ncol = 2)

# Set it as a list with activation functions as names
nn_object = list("tanh" = weights_layer_1,
                 "softplus" = weights_layer_2,
                 "linear" = weights_layer_3)

# Obtain the polynomial representation (order = 3) of that neural network
final_poly <- nn2poly(nn_object, max_order = 3)

# Plot all the coefficients, one plot per output unit
plot(final_poly)

# Plot only the 5 most important coeffcients (by absolute magnitude)
```

```
# one plot per output unit
plot(final_poly, n = 5)
```

---

plot_diagonal   *Plots a comparison between two sets of points.*

Description

If the points come from the predictions of an NN and a PM and the line (`plot.line = TRUE`) is
displayed, in case the method does exhibit asymptotic behavior, the points should not fall in the
line.

Usage

```
plot_diagonal(
  x_axis,
  y_axis,
  xlab = NULL,
  ylab = NULL,
  title = NULL,
  plot.line = TRUE
)
```

Arguments

x_axis	Values to plot in the x axis.
y_axis	Values to plot in the y axis.
xlab	Lab of the x axis
ylab	Lab of the y axis.
title	Title of the plot.
plot.line	If a red line with `slope = 1` and `intercept = 0` should be plotted.

Value

Plot (ggplot object).

---

plot_taylor_and_activation_potentials

Plots activation potentials and Taylor expansion.

Description

Function that allows to take a NN and the data input values and plot the distribution of data activation potentials (sum of input values * weights) at all neurons together at each layer with the Taylor expansion used in the activation functions. If any layer is 'linear' (usually will be the output), then that layer will not be an approximation as Taylor expansion is not needed.

Usage

```
plot_taylor_and_activation_potentials(
  object,
  data,
  max_order,
  taylor_orders = 8,
  constraints,
  taylor_interval = 1.5,
  ...
)
```

Arguments

object
An object for which the computation of the NN2Poly algorithm is desired. Currently supports models from the following deep learning frameworks:

- tensorflow/keras models built as a sequential model.
- torch/luz models built as a sequential model.

It also supports a named list as input which allows to introduce by hand a model from any other source. This list should be of length L (number of hidden layers + 1) containing the weights matrix for each layer. Each element of the list should be named as the activation function used at each layer. Currently supported activation functions are "tanh", "softplus", "sigmoid" and "linear".

At any layer l, the expected shape of such matrices is of the form $(h_{(l-1)} + 1) * (h_l)$, that is, the number of rows is the number of neurons in the previous layer plus the bias vector, and the number of columns is the number of neurons in the current layer L. Therefore, each column corresponds to the weight vector affecting each neuron in that layer. The bias vector should be in the first row.

data
Matrix or data frame containing the predictor variables (X) to be used as input to compute their activation potentials. The response variable column should not be included.

max_order
integer that determines the maximum order that will be forced in the final polynomial, discarding terms of higher order that would naturally arise when considering all Taylor expansions allowed by taylor_orders.

taylor_orders	integer or vector of length L that sets the degree at which Taylor expansion is truncated at each layer. If a single value is used, that value is set for each non linear layer and 1 for linear at each layer activation function. Default set to 8.
constraints	Boolean parameter determining if the NN is constrained (TRUE) or not (FALSE). This only modifies the plots title to show "constrained" or "unconstrained" respectively.
taylor_interval	
	optional parameter determining the interval in which the Taylor expansion is represented. Default is 1.5.
...	Additional parameters.

Value

A list of plots.

---

predict.nn2poly   *Predict method for* nn2poly *objects.*

Description

Predicted values obtained with a nn2poly object on given data.

Usage

```
## S3 method for class 'nn2poly'
predict(object, newdata, layers = NULL, ...)
```

Arguments

object	Object of class inheriting from 'nn2poly'.
newdata	Input data as matrix, vector or dataframe. Number of columns (or elements in vector) should be the number of variables in the polynomial (dimension p). Response variable to be predicted should not be included.
layers	Vector containing the chosen layers from object to be evaluated. If set to NULL, all layers are computed. Default is set to NULL.
...	Further arguments passed to or from other methods.

Details

Internally uses eval_poly() to obtain the predictions. However, this only works with a objects of class nn2poly while eval_poly() can be used with a manually created polynomial in list form.

When object contains all the internal polynomials also, as given by nn2poly(object, keep_layers = TRUE), it is important to note that there are two polynomial items per layer (input/output). These polynomial items will also contain several polynomials of the same structure, one per neuron in the layer, stored as matrix rows in $values. Please see the NN2Poly original paper for more details.

Note also that "linear" layers will contain the same input and output results as Taylor expansion is not used and thus the polynomials are also the same. Because of this, in the situation of evaluating multiple layers we provide the final layer with "input" and "output" even if they are the same, for consistency.

Value

Returns a matrix or list of matrices with the evaluation of each polynomial at each layer as given by the provided `object` of class `nn2poly`.

If `object` contains the polynomials of the last layer, as given by `nn2poly(object, keep_layers = FALSE)`, then the output is a matrix with the evaluation of each data point on each polynomial. In this matrix, each column represents the evaluation of a polynomial and each column corresponds to each point in the new data to be evaluated.

If `object` contains all the internal polynomials also, as given by `nn2poly(object, keep_layers = TRUE)`, then the output is a list of layers (represented by `layer_i`), where each one is another list with `input` and `output` elements, where each one contains a matrix with the evaluation of the "input" or "output" polynomial at the given layer, as explained in the case without internal polynomials.

See Also

`nn2poly()`: function that obtains the nn2poly polynomial object, `eval_poly()`: function that can evaluate polynomials in general, `stats::predict()`: generic predict function.

Examples

```
# Build a NN structure with random weights, with 2 (+ bias) inputs,
# 4 (+bias) neurons in the first hidden layer with "tanh" activation
# function, 4 (+bias) neurons in the second hidden layer with "softplus",
# and 1 "linear" output unit

weights_layer_1 <- matrix(rnorm(12), nrow = 3, ncol = 4)
weights_layer_2 <- matrix(rnorm(20), nrow = 5, ncol = 4)
weights_layer_3 <- matrix(rnorm(5), nrow = 5, ncol = 1)

# Set it as a list with activation functions as names
nn_object = list("tanh" = weights_layer_1,
                 "softplus" = weights_layer_2,
                 "linear" = weights_layer_3)

# Obtain the polynomial representation (order = 3) of that neural network
final_poly <- nn2poly(nn_object, max_order = 3)

# Define some new data, it can be vector, matrix or dataframe
newdata <- matrix(rnorm(10), ncol = 2, nrow = 5)

# Predict using the obtained polynomial
predict(object = final_poly, newdata = newdata)

# Change the last layer to have 3 outputs (as in a multiclass classification)
# problem
```

```
weights_layer_4 <- matrix(rnorm(20), nrow = 5, ncol = 4)

# Set it as a list with activation functions as names
nn_object = list("tanh" = weights_layer_1,
                 "softplus" = weights_layer_2,
                 "linear" = weights_layer_4)

# Obtain the polynomial representation of that neural network
# Polynomial representation of each hidden neuron is given by
final_poly <- nn2poly(nn_object, max_order = 3, keep_layers = TRUE)

# Define some new data, it can be vector, matrix or dataframe
newdata <- matrix(rnorm(10), ncol = 2, nrow = 5)

# Predict using the obtained polynomials (for all layers)
predict(object = final_poly, newdata = newdata)

# Predict using the obtained polynomials (for chosen layers)
predict(object = final_poly, newdata = newdata, layers = c(2,3))
```

Index