

neuralGAM: An R Package for fitting Generalized Additive Neural Networks

A PREPRINT

Ines Ortega-Fernandez 
GRADIANT

Marta Sestelo 
Universidade de Vigo

April 14, 2024

Abstract

Nowadays, neural networks are considered one of the most effective methods for various tasks such as anomaly detection, computer-aided disease detection, or natural language processing. However, these networks suffer from the "black-box" problem which makes it difficult to understand how they make decisions. In order to solve this issue, an R package called **neuralGAM** is introduced. This package implements a neural network topology based on Generalized Additive Models, allowing to fit an independent neural network to estimate the contribution of each feature to the output variable, yielding a highly accurate and interpretable deep learning model. The **neuralGAM** package provides a flexible framework for training Generalized Additive Neural Networks, which does not impose any restrictions on the neural network architecture. We illustrate the use of the **neuralGAM** package in both synthetic and real data examples.

Keywords: neural networks, generalized additive models, explainable artificial intelligence, deep learning.

1. Introduction

Neural networks are one of the most popular methods nowadays given their high performance on diverse tasks, such as computer vision, anomaly detection, computer-aided disease detection and diagnosis, or natural language processing. However, it is usually unclear how neural networks (NN) make decisions, and current methods that try to provide interpretability to neural networks are not robust enough. This is the typical disadvantage of neural networks, which are usually "black-box" models that are hard to interpret.

In recent years, there has been a growing trend towards increasing trust in AI systems by promoting their interpretability and explainability. This means that an AI model should

be able to justify and interpret the decisions it makes. To achieve this, post-hoc methods have emerged as a way to mimic or explain the behaviour of a trained AI model using an external, interpretable model to try to explain the decisions made by the “black-box” AI model (Došilović, Brčić, and Hlupić 2018). Some examples of these methods are LIME (Ribeiro, Singh, and Guestrin 2016a), Anchor-LIME (Ribeiro, Singh, and Guestrin 2016b), and SHAP (Lundberg and Lee 2017). The idea is to identify the rules or input characteristics that influence the underlying “black-box” model by leveraging the interpretability of the “white-box” model. However, these methods may not provide a global explanation of the model, and their explanations may depend on the set of hyperparameters chosen during training. Additionally, these methods lack robustness and may not produce consistent explanations for similar inputs.

In contrast, “ante-hoc” techniques aim to design and train inherently interpretable and explainable models that prioritise accuracy while taking explainability into account (Došilović *et al.* 2018). **neuralGAM** implements a neural network topology that achieves full interpretability and high accuracy by building an ensemble of neural networks to learn each feature’s contribution to the response variable. This approach results in a highly accurate, fully interpretable neural network-based Generalized Additive Model (GAM). We use backpropagation to estimate each feature network, and the GAM model is fitted using local scoring and backfitting algorithms to ensure that it converges and is additive (Ortega-Fernandez, Sestelo, and Villanueva 2023).

neuralGAM is, therefore, a “white-box” deep learning model which provides an exact description of how each covariate affects the response variable. The partial effects of each learned function can be visualised independently, providing information on whether the relationship between the response variable and each covariate is linear, monotonic, or complex. This is not a typical feature of “black-box” neural networks, which are hard to interpret. These characteristics make **neuralGAM** suitable for high-risk AI applications such as medical decision-making. Furthermore, by observing the partial effects, **neuralGAM** can be used to identify and mitigate bias introduced in the training data set. For example, partial functions with a positive slope can increase the probability of observing a particular class in binary classification problems. To the best of our knowledge, we are the first to implement a GAM using independent deep neural networks to estimate the smooth functions. We leverage the local scoring and backfitting algorithms to ensure that the GAM model converges and is additive (Ortega-Fernandez *et al.* 2023).

In order to streamline this task, it is imperative to design software that can effectively implement the proposed methods in a researcher-friendly and comprehensible environment. Our package fulfils this objective by offering a range of user-friendly functions. The package **neuralGAM** is the first to implement a Generalized Additive Neural Network in R, and it is freely available from the Comprehensive R Archive Network (CRAN) at <https://cran.r-project.org/package=neuralGAM>. The recent surge in the adoption of deep learning has led to a remarkable expansion of R deep learning packages in the Comprehensive R Archive Network (CRAN). As per the Machine Learning and Statistical Learning CRAN task view, a comprehensive inventory of such packages is presented below.

Single hidden layer (shallow) neural networks are implemented in R in the **nnet** (Venables and Ripley 2002), which is shipped with base R. Deep learning flavours of neural networks are implemented on the **deepnet** (Rong and Rong 2014), **RcppDL** (Kou and Sugomori 2014), **neuralnet**, and **h2o** (Aiello *et al.* 2016) packages, among others. Regarding interfaces to

high-level APIs, which provide a user-friendly interface to build and train neural networks, the most used R packages are **keras** (Chollet *et al.* 2015) and **tensorflow** (Abadi *et al.* 2015). TensorFlow is a powerful and widely used open-source deep learning library that provides a comprehensive set of tools and resources for building neural networks. In R, TensorFlow offers a flexible and intuitive API, enabling users to construct custom deep-learning models effortlessly. It supports dynamic computation graphs, allowing for efficient training and deployment of models. In 2020 the **torch** (Falbel and Luraschi 2023) package for R was announced, which uses a C++ backend instead of Python, and R6 to provide object-oriented capabilities, which might be unfamiliar to R users. Keras is a user-friendly high-level deep learning package that can be utilized to build and train neural networks. It provides a wide range of pre-built models and supports various network architectures. Its integration with TensorFlow (and Python) as the backend ensures efficient computation and seamless interoperability.

Regarding R software which provides explainability capabilities to AI models, **NeuralNetTools** (Beck 2018) can be used to interpret supervised neural network models that are created in R. The functions in the package can be used to visualise the model with the help of a neural network interpretation diagram, analyse the importance of variables by breaking down the model weights and conduct a sensitivity analysis of the response variables to changes in the input variables. It provides functions to plot neural network architectures, weight histograms, and activation functions, allowing visualisation of the connectivity and structure of neural networks, which can aid in understanding and debugging complex models. **iml** (Molnar, Casalicchio, and Bischl 2018) provides a set of tools for interpreting machine learning models, but focusing on local interpretation of individual predictions rather than global model explanations, including features such as feature importance measures with permutation-based variable importance, partial dependence plots, and accumulated local effect plots. It also includes functions to explain model predictions using Shapley values. **DALEX** (Biecek 2018) (Descriptive mACHINE Learning EXplanations) provides model-agnostic explanation and visualisation for machine learning models using explainers such as LIME, Break Down, or Shapley values. The package offers both local and global interpretability, enabling users to explore model behaviour at different levels of granularity. While **iml** and **DALEX** are model-agnostic, **NeuralNetTools** is specifically designed for neural networks.

In this paper, we explain and illustrate how neural networks can be used to fit Generalized Additive Models using the **neuralGAM** package via a simulated scenario with Gaussian response, and a real-life application related to flight delay prediction based on weather and flight condition's data.

The remainder of the paper is structured as follows: in Section 2 we briefly review the estimation procedures and explain the use of the main functions and methods of **neuralGAM**; Section 3 gives an illustration of the practical application of the package using simulated and real data; and finally, Section 4 concludes with a discussion and possible future extensions of the package.

2. Models and software

2.1. An overview of the methodology

The most common way to model the relationship between the response variable and the

covariates is by using the multiple linear regression model, where the response variable (Y) is assumed to be normally distributed, and the covariates ($X_j, j = 1, \dots, p$) are assumed to have a linear effect on the response. However, the response variable may not be normally distributed and, in such cases, Generalized Linear Models allow the use of other distribution families, e.g. binomial, Poisson, etc. Additionally, in some cases, the assumption of linearity in the effects of covariates can be too restrictive, and not supported by the available data. In this setting, nonparametric regression techniques emerge, allowing us to model this dependence between the response and the covariates without specifying in advance the function that links them. This leads to the Generalized Additive Models (Hastie and Tibshirani 1990) defined by

$$E[Y | \mathbf{X}] = m(\mathbf{X}) = h^{-1}\left(\alpha + \sum_{j=1}^p f_j(X_j)\right), \quad (1)$$

where $h(\cdot)$ is a monotonic known function (the link function) and f_1, \dots, f_p are smooth and unknown functions.

To fit the previous model in (1), from an independent random sample $\{\mathbf{X}_i, Y_i\}_{i=1}^n$, with $\mathbf{X} = X_1, \dots, X_p$, we use a combination of the local scoring algorithm (see Algorithm 1) and the backfitting algorithm (see Algorithm 2). Particularly, the estimation of the additive predictor $\eta = \alpha + \sum_{j=1}^p f_j(\cdot)$ is obtained by fitting a weighted additive model using neural networks as function estimators. For each f_j , the backfitting algorithm iteratively estimates the effect of each covariate X_1, \dots, X_p , with weights W_i , on the adjusted dependent variable Z_i , updated at each step of the local scoring algorithm.

Algorithm 1: Local scoring algorithm

Initialise $\hat{\alpha} = h(\bar{Y}_i), \hat{f}_j^0(\cdot) = 0 \forall j, l = 0$

for $l \leftarrow l + 1$ **do**

Construct an adjusted dependent variable Z_i according to Table 1 with

$$\hat{\eta}_i^{l-1} = \hat{\alpha} + \sum_{j=1}^p \hat{f}_j^{l-1}(X_{ij}) \text{ and}$$

$$\hat{\mu}_i^{l-1} = h^{-1}(\hat{\eta}_i^{l-1})$$

Form the weights W_i according to Table 1.

Fit a weighted additive model to Z_i using the backfitting algorithm (Algorithm 2)

with weights W_i to obtain the estimated functions \hat{f}_j^l , additive predictor $\hat{\eta}_i^l$ and

fitted values $\hat{\mu}_i^l$

end

Compute the convergence criterion $DEV < \delta$, with

$$DEV = \frac{\sum_{i=1}^n DEV_i(Y_i, \hat{\mu}_i^{l-1}) - DEV_i(Y_i, \hat{\mu}_i^l)}{\sum_{j=1}^n DEV_i(Y_i, \hat{\mu}_i^{l-1})} \text{ according to Table 1, where } \delta \text{ is a small threshold}$$

Until the convergence criterion is satisfied

Note that if the link function is the identity and the error distribution is Gaussian, then $Z_i = Y_i$ and the weights do not change, thus the procedure is simply an additive fit. In any other case, the dependent variable Z_i and the weights W_i are updated at each iteration l of the local scoring algorithm. This process is repeated until the convergence criterion is satisfied (see last step of Algorithm 1). Note that we use deviance because it is an appropriate

measure of discrepancy between observed and fitted values.

Algorithm 2: Backfitting Algorithm with neural networks

Initialise $\hat{f}_j^0(\cdot) = 0 \forall j$, $m = 0$

Iterate $m \leftarrow m + 1$

for $j \leftarrow 1, p$ **do**

 Compute partial residuals $R_{ij} = Z_i - \hat{\alpha} - \sum_{k=1}^{j-1} \hat{f}_k^m(X_{ik}) - \sum_{k=j+1}^p \hat{f}_k^{m-1}(X_{ik})$

 Fit a neural network with R_{ij} and X_{ij} to obtain \hat{f}_j^m

 Replace \hat{f}_j^m with its centered version $\hat{f}_j^m(\cdot) - \frac{\sum_{i=1}^n \hat{f}_j^m(X_{ij})}{n}$

end

Until $\frac{\sum_{j=1}^p \sum_{i=1}^n (\hat{f}_j^m(X_{ij}) - \hat{f}_j^{m-1}(X_{ij}))^2}{\sum_{j=1}^p \sum_{i=1}^n (\hat{f}_j^{m-1}(X_{ij}))^2} < \epsilon$

Particularly, given the fitted mean response $\hat{\mu}_i = \hat{E}[Y_i | \mathbf{X}_i]$, the deviance is defined as $DEV = \frac{\sum_{i=1}^n DEV_i(Y_i, \hat{\mu}_i^{l-1}) - DEV_i(Y_i, \hat{\mu}_i^l)}{\sum_{i=1}^n DEV_i(Y_i, \hat{\mu}_i^{l-1})}$, with DEV_i depending on the link (see Table 1).

Distribution	Link	Z_i	W_i	$DEV_i(Y_i, \hat{\mu}_i)$
Gaussian	identity	Y_i	1	$(Y_i - \hat{\mu}_i)^2$
Binomial (s, μ)	logit	$\eta_i + (Y_i - \mu_i)s\mu_i(1 - \mu_i)$	$s\mu_i(1 - \mu_i)$	$-2(Y_i \log \hat{\mu}_i + (1 - Y_i) \log(1 - \hat{\mu}_i))$

Table 1: Adjusted dependent variable Z_i , weights W_i , and deviance $DEV_i(Y_i, \hat{\mu}_i)$, at the local scoring algorithm for Gaussian and binomial distribution (Hastie and Tibshirani 1990).

Several approaches have been described in the literature to estimate the regression model in (1), such as Bayesian approaches (Lang and Brezger 2004), local polynomial kernel smoothers (Wand and Jones 1994; Copeland 1997) or regression splines (De Boor, De Boor, Mathématicien, De Boor, and De Boor 1978). Unlike these quite common techniques, we propose to use independent neural networks (which are universal function estimators (Hornik, Stinchcombe, and White 1989)) to learn the contribution of each covariate to the dependent variable, i.e., at each iteration of the backfitting algorithm, we train a feed-forward neural network for each covariate X_j with the entire set of training data for one epoch. The fits are improved at each epoch as the learned adjusted dependent variable Z_i approaches Y_i at each iteration of the local scoring algorithm.

2.2. Package structure and functionality

The **neuralGAM** package introduces a new methodology for fitting Generalized Additive Models, with Gaussian and binary responses, based on Neural Network techniques, and it is composed of several functions that enable users to fit the models with the methods described above. **neuralGAM** is implemented using Keras (Chollet *et al.* 2015), a high-level API for the implementation of neural networks well known for its simplicity, flexibility, and ease of use.

The package is designed along lines similar to those of other R regression packages. The functions within **neuralGAM** are briefly described in Table 2. The main function of the package

Function	Description
<code>neuralGAM</code>	Main function to fit a <code>neuralGAM</code> model. The function builds one neural network to attend to each feature in <code>data</code> , using the backfitting and local scoring algorithms to fit a weighted additive model using neural networks as function approximators.
<code>summary.neuralGAM</code>	Method of the generic summary function for <code>neuralGAM</code> objects.
<code>print.neuralGAM</code>	Method of the generic print function for <code>neuralGAM</code> object, which prints out some key components.
<code>plot.neuralGAM</code>	Visualization of <code>neuralGAM</code> objects with the generic function for plotting of R objects. Provides default plot showing the smooth and linear components of a fitted <code>neuralGAM</code> .
<code>autoplot.neuralGAM</code>	Visualization of <code>neuralGAM</code> objects with with <code>ggplot2</code> (Wickham 2016) graphics. Provides default plot showing the smooth and linear components of a fitted <code>neuralGAM</code> .
<code>predict.neuralGAM</code>	Takes a <code>neuralGAM</code> object produced by <code>neuralGAM()</code> and, given a new set of values for the model covariates, produces predictions.
<code>install_neuralGAM</code>	Creates a <code>conda</code> environment (installing <code>miniconda</code> if required) and sets up the Python requirements to run <code>neuralGAM</code> (Tensorflow and Keras).

Table 2: Summary of functions in the `neuralGAM` package.

is `neuralGAM`, which fits a generalised additive model using neural networks to estimate the contribution of each smooth function to the response. The arguments of this function are shown in Table 3. Note that through the argument `formula` users can decide to fit a model with smooth ($s(x)$), linear or factor terms (x), and by means of the argument `family` it is possible to select the conditional distribution of the response variable. So far, the user can select between Gaussian and binomial. Numerical and graphical summaries of the fitted object can be obtained by using the `print`, `summary`, `plot`, and `autoplot` methods implemented for `neuralGAM` objects. Another of these methods is available for the `predict` function which takes a fitted model of the `neuralGAM` class and, given a new data set of values of the covariates by means of the argument `newdata`, produces predictions. At last, we provide a helper function `install_neuralGAM` to assist the user in installing the required Python dependencies in a custom `conda` environment.

An example of the use of `neuralGAM`, illustrating all the features provided is given in the following example of code. In the following example, `neuralGAM` is configured to fit a deep neural network with 64 units on each of its three deep layers (`num_units`).

```
R> neuralGAM(y ~ x + s(z) + s(w), data = data, num_units = c(64, 64, 64),
+   family = "gaussian", learning_rate = 0.001, activation = "relu",
+   kernel_initializer = "glorot_normal", loss = "mse",
+   kernel_regularizer = NULL, bias_regularizer = NULL,
+   bias_initializer = 'zeros', activity_regularizer = NULL,
+   bf_threshold = 0.001, ls_threshold = 0.1, max_iter_backfitting = 10,
+   max_iter_ls = 10, seed = NULL, verbose = 0, ...)
```

Note that a working Python environment with `keras` and `tensorflow` packages is required

	neuralGAM() arguments
formula	An object of class "formula": a description of the model to be fitted. You can add smooth terms using <code>s()</code> .
data	A data frame containing the model response variable and covariates required by the formula. Additional terms not present in the formula will be ignored.
num_units	Defines the architecture of each neural network. If a scalar value is provided, a single hidden layer neural network with that number of units is used. If a list of values is provided, a multi-layer neural network with each element of the list defining the number of hidden units on each hidden layer is used.
family	This is a family object specifying the distribution and link to use for fitting. By default, it is "gaussian" but also works to "binomial" for logistic regression.
learning_rate	Learning rate for the neural network optimiser.
activation	Activation function to use on every layer of the neural network. Defaults to <code>relu</code> . See <code>layer_dense</code> documentation from the Keras library.
loss	Loss function to use during neural network training. Defaults to the mean squared error (<code>mse</code>).
kernel_initializer	Kernel initializer for the Dense layers. Defaults to Xavier initializer <code>glorot_normal</code> (Glorot and Bengio 2010).
kernel_regularizer	Optional regularizer function applied to the kernel weights matrix.
bias_regularizer	Optional regularizer function applied to the bias vector.
bias_initializer	Optional initializer for the bias vector.
activity_regularizer	Optional regularizer function applied to the output of the layer.
w_train	Optional sample weights.
bf_threshold	Convergence criterion of the backfitting algorithm. By default it is 0.001.
ls_threshold	Convergence criterion of the local scoring algorithm. Defaults to 0.1.
max_iter_backfitting	An integer with the maximum number of iterations of the backfitting algorithm. Defaults to 10.
max_iter_ls	An integer with the maximum number of iterations of the local scoring Algorithm. Defaults to 10.
seed	Specifies the random number generator seed for algorithms dependent on randomization.
verbose	Verbosity mode (0 = silent, 1 = print messages). Defaults to 1.
...	Other arguments to pass on to the Adam optimizer (Kingma and Ba 2014).

Table 3: Arguments of **neuralGAM** function.

to run **neuralGAM**. We provide the helper function `install_neuralGAM()` to install the required dependencies in a custom `conda` environment. Once the dependencies are installed using `install_neuralGAM()`, you must reload the library again using `library(neuralGAM)` for the changes to take effect.

The linear predictor is composed of a linear term in x and smooth terms of z and w . The response is assumed to follow a Gaussian distribution (**family**).

Other arguments of **neuralGAM** will be familiar to **keras** users: **activation** defines the activation function for the Dense layers of each fitted Neural Network (defaults to **relu**). **neuralGAM** uses Adam (Kingma and Ba 2014) as an optimizer for stochastic gradient descent, whose behaviour can be customised using the **loss**, **learning_rate**, **kernel_initializer**, **kernel_regularizer**, **bias_regularizer**, **bias_initializer**, and **activity_regularizer** parameters. The backfitting and local scoring algorithms used to fit the GAM model can be configured using the **bf_threshold**, **ls_threshold** which adjust the convergence criterion of both algorithms, while **max_iter_backfitting** and **max_iter_ls** adjust the maximum number of iterations of each algorithm.

Finally, **seed** specifies an optional random number generator seed for algorithms dependent on randomization, **verbose** sets the verbosity of the printed outputs, while the **...** argument allows setting other arguments available for the Adam implementation in **keras** such as the exponential decay rate for the 1st and 2nd-moment estimates.

Creating efficient and appropriate data visualizations becomes increasingly important as the complexity of the data increases. **neuralGAM** provides two different methods for plotting data: one based on R's standard plotting function (**plot.default**) in which the **plot** function inherits all parameters from **graphics** package and can be set as usual, and one based on **ggplot2** (Wickham 2016). With this latter plot method, the **autoplot** function creates a **ggplot** object that can be modified later by the user just using the functionality provided in the **ggplot2** package.

The following excerpt of code shows analogous plots using the two methods implemented:

```
R> plot(ngam, main = "My plot")
R> library(ggplot2)
R> autoplot(ngam, select = "x1") + ggtitle("My plot")
```

3. Illustrative examples

In this section, we illustrate the use of **neuralGAM** package using some simulated and real data. We consider examples for each of the conditional distributions of the response variable, i.e., both Gaussian and binary.

3.1. Application to simulated data

This subsection reports the capabilities of the **neuralGAM** package in a simulated scenario. We used a sample size of $n = 30625$ which was split into 80% for training the model and 20% for testing. We consider the following predictor

$$\eta = \alpha + \sum_{j=1}^3 f_j(X_j),$$

with

$$f_j(X_j) = \begin{cases} X_j^2 & \text{if } j = 1, \\ 2X_j & \text{if } j = 2, \\ \sin X_j & \text{if } j = 3, \end{cases}$$

$\alpha = 2$, and covariates X_j drawn from an uniform distribution $U[-2.5, 2.5]$. The response follows a Gaussian distribution with $Y = \eta + \epsilon$, where ϵ is the error distributed in accordance to a $N(0, \sigma(x))$ in a homoscedastic situation with $\sigma(x) = 0.25$.

The code for the generation of this data can be found below:

```
seed <- 42
set.seed(seed)
n <- 30625
x1 <- runif(n, -2.5, 2.5)
x2 <- runif(n, -2.5, 2.5)
x3 <- runif(n, -2.5, 2.5)
f1 <- x1 ** 2
f2 <- 2 * x2
f3 <- sin(x3)
f1 <- f1 - mean(f1)
f2 <- f2 - mean(f2)
f3 <- f3 - mean(f3)
eta0 <- 2 + f1 + f2 + f3
epsilon <- rnorm(n, 0.25)
y <- eta0 + epsilon
dat <- data.frame(x1, x2, x3, y)
sample <- sample(c(TRUE, FALSE), n, replace = TRUE, prob = c(0.8, 0.2))
train <- dat[sample,]
test <- dat[!sample,]
summary(dat)
```

##	x1	x2	x3	y
## Min.	:-2.500000	Min. :-2.49950	Min. :-2.4995000	Min. :-8.2672
## 1st Qu.	:-1.272700	1st Qu.: -1.24730	1st Qu.: -1.2516000	1st Qu.: -0.5044
## Median	:-0.004200	Median : 0.01910	Median : 0.0014000	Median : 2.2692
## Mean	:-0.006529	Mean : 0.01788	Mean :-0.0004923	Mean : 2.2574
## 3rd Qu.	: 1.247700	3rd Qu.: 1.28180	3rd Qu.: 1.2469000	3rd Qu.: 4.8916
## Max.	: 2.500000	Max. : 2.49990	Max. : 2.4997000	Max. :13.5417

Once the data have been generated, we can fit the model using the **train** data set. We will use smooth terms for all the covariates to observe if **neuralGAM** can learn both linear and smooth effects:

```
library(neuralGAM)
ngam <- neuralGAM(y ~ s(x1) + s(x2) + s(x3), data = train,
                  num_units = 1024,
                  learning_rate = 0.001,
                  bf_threshold = 0.001,
                  seed = seed, verbose = 0)
ngam
```

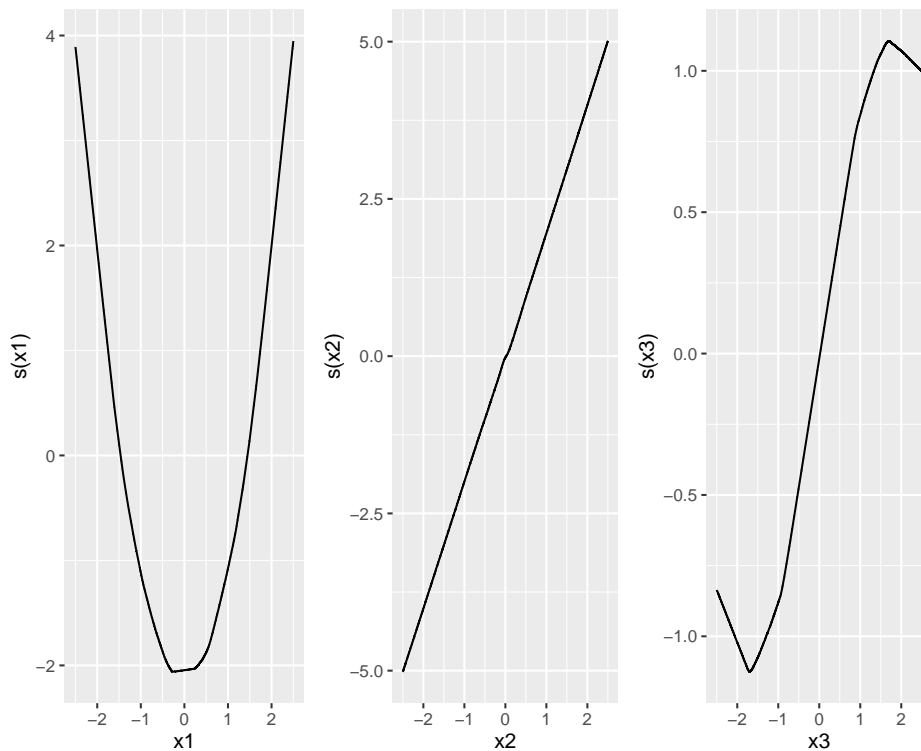
```
## Class: neuralGAM
##
## Distribution Family: gaussian
## Formula: y ~ s(x1) + s(x2) + s(x3)
## Intercept: 2.2422
```

```
## MSE: 1.0311
## Sample size: 24522
```

We can observe that the obtained mean squared error (MSE) during training was 1.0311, which reflects a good performance.

neuralGAM model can be also visualised using the `print`, `summary`, `plot` and `autoplot` functions, while the model components can be extracted using `predict`. The figure shows the estimated curve for each fitted term obtained with the `autoplot` function, as in the next example. We can observe how the model is able to properly estimate the original shape of each partial function, showcasing the good performance of the model in this scenario and the ability of **neuralGAM** to learn both linear (`x2`) and smooth functions (`x1`, `x3`) using neural networks as function estimators.

```
plots <- lapply(c("x1", "x2", "x3"), function(x) autoplot(ngam, select = x))
gridExtra::grid.arrange(grobs = plots, ncol = 3, nrow = 1)
```



The `summary` method returns a summary of the fit where it is possible to observe the model architecture for both the neural networks and the linear terms. For each neural network, one can observe the type of each layer, its shape, and the number of parameters:

```
summary(ngam)

## Class: neuralGAM
##
```

```

## Distribution Family: gaussian
## Formula: y ~ s(x1) + s(x2) + s(x3)
## Intercept: 2.2422
## MSE: 1.0311
## Sample size: 24522
##
## Training History:
##
##           Timestamp Model Epoch TrainLoss
## 1 2024-01-05 12:55:08   x1      1    10.5817
## 2 2024-01-05 12:55:11   x2      1     1.8857
## 3 2024-01-05 12:55:14   x3      1     1.1577
## 4 2024-01-05 12:55:17   x1      2     1.0791
## 5 2024-01-05 12:55:20   x2      2     1.0576
## 6 2024-01-05 12:55:23   x3      2     1.0387
## 7 2024-01-05 12:55:25   x1      3     1.0507
## 8 2024-01-05 12:55:28   x2      3     1.0458
## 9 2024-01-05 12:55:31   x3      3     1.0285
##
##
## Model architecture:
##
## $x1
## Model: "x1"
## -----
## Layer (type)                Output Shape                Param #
## =====
## dense (Dense)                (None, 1)                    2
## dense_1 (Dense)              (None, 1024)                 2048
## dense_2 (Dense)              (None, 1)                    1025
## =====
## Total params: 3075 (12.01 KB)
## Trainable params: 3075 (12.01 KB)
## Non-trainable params: 0 (0.00 Byte)
## -----
##
## $x2
## Model: "x2"
## -----
## Layer (type)                Output Shape                Param #
## =====
## dense_3 (Dense)              (None, 1)                    2
## dense_4 (Dense)              (None, 1024)                 2048
## dense_5 (Dense)              (None, 1)                    1025
## =====
## Total params: 3075 (12.01 KB)
## Trainable params: 3075 (12.01 KB)
## Non-trainable params: 0 (0.00 Byte)
## -----
##
## $x3
## Model: "x3"

```

```
## -----
## Layer (type)                Output Shape          Param #
## -----
## dense_6 (Dense)             (None, 1)             2
## dense_7 (Dense)             (None, 1024)           2048
## dense_8 (Dense)             (None, 1)             1025
## -----
## Total params: 3075 (12.01 KB)
## Trainable params: 3075 (12.01 KB)
## Non-trainable params: 0 (0.00 Byte)
## -----
```

The `predict` function can be used to obtain the predictions given a new set of values for the model covariates from the `test` set. We can choose to get the predicted response, the terms, or the linear predictor by adjusting the `type` argument. When using `type = "terms"`, the specific set of terms to be obtained can be selected using the `terms` argument. If no `newdata` parameter is provided, the function returns the predictions for the original training data.

```
eta <- predict(ngam, newdata = test, type = "link", verbose = 0)
head(eta)

## [1] -1.793064 -2.181689  1.671155  5.670309  2.807211  5.099676

yhat <- predict(ngam, newdata = test, type = "response", verbose = 0)
head(yhat)

## [1] -1.793064 -2.181689  1.671155  5.670309  2.807211  5.099676
```

```
terms <- predict(ngam, newdata = test, type = "terms", verbose = 0)
head(terms)

##           x1           x2           x3
## 1 -1.952659 -1.9873928 -0.09517136
## 2  1.375532 -4.8206234 -0.97875702
## 3  3.818488 -4.9609156  0.57142270
## 4 -1.719761  4.3348103  0.81309992
## 5 -1.971988  1.4300301  1.10700893
## 6  3.836700 -0.1894903 -0.78969347

terms <-
  predict(ngam,
    newdata = test,
    type = "terms",
    terms = c("x1", "x3"), verbose = 0)
head(terms)

##           x1           x3
## 1 -1.952659 -0.09517136
## 2  1.375532 -0.97875702
```

```
## 3  3.818488  0.57142270
## 4 -1.719761  0.81309992
## 5 -1.971988  1.10700893
## 6  3.836700 -0.78969347
```

At last, we can study the performance of the model in a set of test data using different metrics, such as the Mean Squared Error (`mse`):

```
mse <- mean((yhat - test$y) ^ 2)
mse

## [1] 1.063829
```

We can observe that the Mean Squared Error in the test set has similar values to the obtained values during training (`MSE_training = 1.0311`), reflecting that the model has a good capacity to generalise to a set of unseen data.

3.2. Application to real data

We decided to also show the capabilities of the **neuralGAM** package with another data set. Particularly, this subsection details an example of its application to real data taken from the NYC Flights 13 data set from the `nycflights13` package (Wickham 2022). The data set includes airline on-time data for flights departing from all the airports in New York City during 2013. It also includes useful metadata on airlines, airports, weather conditions at different airports, and plane information.

We aim to predict whether a flight will be delayed (upon arrival) given departure flight information and certain weather conditions. With this in mind, firstly, we load the required data from the `nycflights13` package. We can join the flights and weather data to obtain the weather conditions at a given airport (`origin`) and time (`time_hour`). We will focus on flights departing from Newark Liberty International Airport (`origin == "EWR"`) in the months of October, November, and December (`month %in% c(10,11,12)`).

```
suppressMessages(library(magrittr))
suppressMessages(library(dplyr))
suppressMessages(library(ggplot2))
suppressMessages(library(gridExtra))
suppressMessages(library(nycflights13))
```

```
seed <- 1234
set.seed(seed)
data(flights, package="nycflights13")
data(weather, package="nycflights13")
data(airlines, package="nycflights13")
dat <- filter(flights, origin == "EWR" & month %in% c(12, 11, 10)) %>%
  left_join(weather, by = c("origin", "time_hour")) %>%
  select(arr_delay, dep_delay, dep_time, air_time, arr_time,
         origin, carrier, visib, distance, air_time, temp, humid) %>%
  data.frame
```

We construct a binary response variable `delay` which describes if the flight was delayed at arrival:

```
dat$delay = ifelse(dat$arr_delay > 0, 1, 0)
dat <- dat[!rowSums(is.na(dat)),]
print(dat %>% count(delay))

##   delay      n
## 1      0 16215
## 2      1 12358

head(dat)

##   arr_delay dep_delay dep_time air_time arr_time origin carrier visib distance
## 1      -34      -13     447      69      614    EWR      US     10      529
## 2      -22       5     522     174      735    EWR      UA     10     1400
## 3       -3      -9     551     117      727    EWR      UA     10      719
## 4      -13      -9     551      40     655    EWR      B6     10      200
## 5      -46      -6     554     169     757    EWR      UA     10     1400
## 6      -30      -5     555     117     810    EWR      B6     10      937
##   temp humid delay
## 1 53.06 89.31     0
## 2 53.06 89.31     0
## 3 55.04 89.40     0
## 4 55.04 89.40     0
## 5 55.04 89.40     0
## 6 55.04 89.40     0
```

We split the data set into training (80%) and test (20%) splits, and fit a **neuralGAM** model using the **binomial** family with 2 layers with 256 and 128 units on each layer. The model will try to predict whether a flight is going to be delayed considering the flight air time, the departure delay, and weather conditions (temperature and humidity at origin) as covariates:

```
sample <- sample(nrow(dat), 0.8 * nrow(dat))
train <- dat[sample, ]
test <- dat[-sample, ]
```

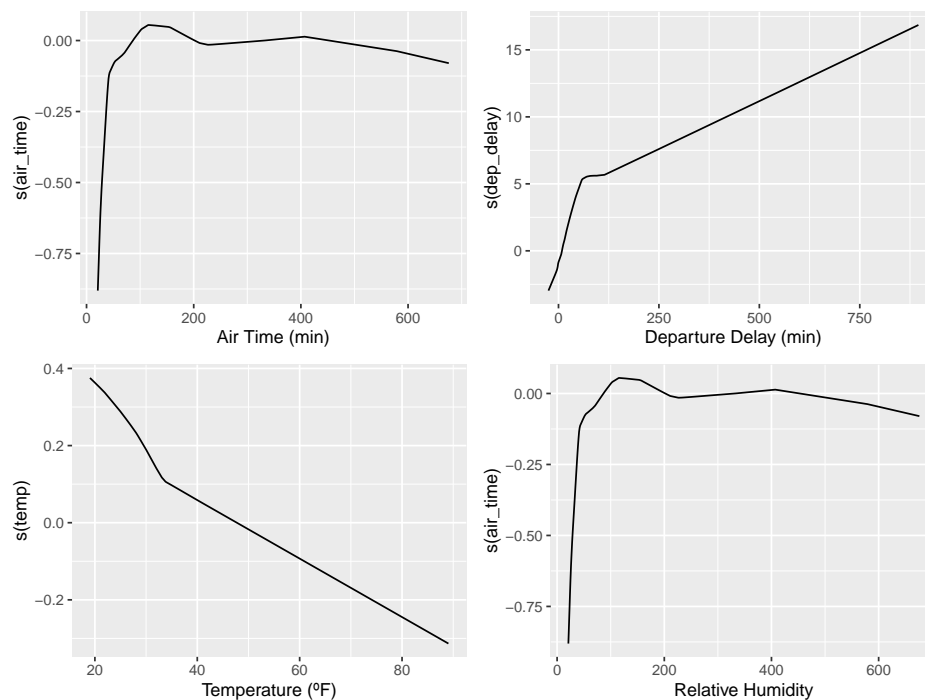
```
library(neuralGAM)
ngam <- neuralGAM(delay ~ s(air_time) + s(dep_delay) + s(temp) + s(humid),
  data = train,
  num_units = c(256, 128),
  family = "binomial",
  seed = seed,
  bf_threshold = 1e-2,
  ls_threshold = 0.01,
  loss = "mse",
  verbose = 0)

ngam
```

```
## Class: neuralGAM
##
## Distribution Family: binomial
## Formula: delay ~ s(air_time) + s(dep_delay) + s(temp) + s(humid)
## Intercept: 0.1024
## MSE: 0.1529
## Sample size: 22858
```

The graphical representation of the model can be obtained by means of the `autoplot` function, specifying with the `select` argument the covariate to be plotted.

```
p1 <- autoplot(ngam, select = "air_time", xlab = "Air Time (min)")
p2 <- autoplot(ngam, select = "dep_delay", xlab = "Departure Delay (min)")
p3 <- autoplot(ngam, select = "temp", xlab = "Temperature (°C)")
p4 <- autoplot(ngam, select = "air_time", xlab = "Relative Humidity")
gridExtra::grid.arrange(grobs = list(p1,p2,p3,p4), ncol = 2, nrow = 2)
```



The plot shows the learned partial effects for each covariate. Observing the plots we can study how the departure (`air_time`, `dep_delay`) and weather conditions (`temp`, `humid`) influence the delay at arrival. As expected, higher values of departure delay increase the probability of a flight being delayed at arrival. Regarding the influence of the `air_time` (the number of minutes the flight is on air), we can see how flights of up to 60min have a higher probability of being delayed, and the probability decreases for larger flights. This could be because larger flights have more time to recover from departure delays on the route.

At last, as expected the arrival delay is also influenced by the weather conditions: lower temperatures increase the probability of delay, especially below 32°F (0°C). Similarly, higher

relative humidity values increase the delay probability, since they represent less favourable weather conditions for flight departure.

We can also study the performance of the model in a set of test data using the `predict` function:

```
predictions <- predict(ngam, newdata = test, type = "response", verbose = 0)
suppressMessages(library(pROC))
roc_data <- roc(test$delay, predictions)

## Setting levels: control = 0, case = 1
## Setting direction: controls < cases

roc_data$auc

## Area under the curve: 0.8301
```

The model obtains an Area under the ROC curve value of 0.8301, showing that **neuralGAM** has a good performance on real applications.

4. Summary and discussion

This paper discussed the implementation of a new framework developed to train Generalized Additive Models with (deep) neural networks. In particular, the proposed methodology trains an independent neural network to estimate the contribution of each covariate to the response, applying the backfitting and local scoring algorithms to ensure that the Generalized Additive Model converges and is additive.

The current version of the package implements Gaussian and binomial response, making it suitable for multiple applications. Future work includes the implementation of additional response types, such as the Poisson response, and the ability to perform multinomial logistic regression. Moreover, we foresee supporting additional neural network training frameworks, APIs or R packages.

Computational details

The results in this paper were obtained using R 4.3.1 with the **neuralGAM** 1.1.0 package. R itself and all packages used are available from the Comprehensive R Archive Network (CRAN) at <https://CRAN.R-project.org/>.

Acknowledgments

This work was supported by the Ayudas Cervera para Centros Tecnológicos grant of the Spanish Centre for the Development of Industrial Technology (CDTI) under the project ÉGIDA (CER-20191012); the Xunta de Galicia (Centro singular de investigación de Galicia accreditation 2019-2022) and the European Union (European Regional Development Fund - ERDF); and the Grant PID2020-118101GB-I00(MINECO/AEI/FEDER, UE).

References

- Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015). “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.” Software available from [tensorflow.org](https://www.tensorflow.org/), URL <https://www.tensorflow.org/>.
- Aiello S, Eckstrand E, Fu A, Landry M, Aboyoun P (2016). “Machine Learning with R and H2O.” *H2O booklet*, **550**.
- Beck MW (2018). “NeuralNetTools: Visualization and analysis tools for neural networks.” *Journal of statistical software*, **85**(11), 1.
- Biecek P (2018). “DALEX: Explainers for complex predictive models in R.” *The Journal of Machine Learning Research*, **19**(1), 3245–3249.
- Chollet F, *et al.* (2015). “Keras.” <https://keras.io>.
- Copeland KA (1997). “Local polynomial modelling and its applications.”
- De Boor C, De Boor C, Mathématicien EU, De Boor C, De Boor C (1978). *A practical guide to splines*, volume 27. Springer-Verlag New York.
- Došilović FK, Brčić M, Hlupić N (2018). “Explainable artificial intelligence: A survey.” In *2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO)*, pp. 0210–0215. IEEE.
- Falbel D, Luraschi J (2023). *torch: Tensors and Neural Networks with 'GPU' Acceleration*. <https://torch.mlverse.org/docs>, <https://github.com/mlverse/torch>.
- Glorot X, Bengio Y (2010). “Understanding the difficulty of training deep feedforward neural networks.” In YW Teh, M Titterton (eds.), *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pp. 249–256. PMLR, Chia Laguna Resort, Sardinia, Italy. URL <https://proceedings.mlr.press/v9/glorot10a.html>.
- Hastie T, Tibshirani R (1990). “Generalized Additive Models.” *London: Chapman and Hall*, **1931**(11), 683–741.
- Hornik K, Stinchcombe M, White H (1989). “Multilayer feedforward networks are universal approximators.” *Neural Networks*, **2**(5), 359–366. ISSN 0893-6080. doi:[https://doi.org/10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8). URL <https://www.sciencedirect.com/science/article/pii/0893608089900208>.
- Kingma DP, Ba J (2014). “Adam: A method for stochastic optimization.” *arXiv preprint arXiv:1412.6980*.
- Kou Q, Sugomori Y (2014). “RcppDL: Deep learning methods via rcpp.”

- Lang S, Brezger A (2004). “Bayesian P-splines.” *Journal of computational and graphical statistics*, **13**(1), 183–212.
- Lundberg SM, Lee SI (2017). “A unified approach to interpreting model predictions.” *Advances in neural information processing systems*, **30**.
- Molnar C, Casalicchio G, Bischl B (2018). “iml: An R package for interpretable machine learning.” *Journal of Open Source Software*, **3**(26), 786.
- Ortega-Fernandez I, Sestelo M, Villanueva NM (2023). “Explainable generalized additive neural networks with independent neural network training.” *Statistics and Computing*, **34**(1), 6. ISSN 1573-1375. doi:10.1007/s11222-023-10320-5. URL <https://doi.org/10.1007/s11222-023-10320-5>.
- Ribeiro MT, Singh S, Guestrin C (2016a). “Model-Agnostic Interpretability of Machine Learning.” *arXiv preprint arXiv:1606.05386*. 1606.05386, URL <http://arxiv.org/abs/1606.05386>.
- Ribeiro MT, Singh S, Guestrin C (2016b). “Nothing else matters: Model-agnostic explanations by identifying prediction invariance.” *arXiv preprint arXiv:1611.05817*.
- Rong X, Rong MX (2014). “Package ‘deepnet.’”
- Venables WN, Ripley BD (2002). *Modern Applied Statistics with S*. Fourth edition. Springer, New York. ISBN 0-387-95457-0, URL <https://www.stats.ox.ac.uk/pub/MASS4/>.
- Wand MP, Jones MC (1994). *Kernel smoothing*. CRC press.
- Wickham H (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN 978-3-319-24277-4. URL <https://ggplot2.tidyverse.org>.
- Wickham H (2022). *nycflights13: Flights that Departed NYC in 2013*. R package version 1.0.2, URL <https://github.com/hadley/nycflights13>.