Neural Networks

ACM

November 8th, 2023

```
## Registered S3 method overwritten by 'printr':
##   method                from
##   knit_print.data.frame rmarkdown
```

Our example data set is from the Wisconsin cancer study. We read in the data and remove any rows with missing data. The following code uses the package `mlbench` that contains this data set. We show the top few lines of the data.

```r
library(mlbench)
data("BreastCancer")

#Clean off rows with missing data
BreastCancer = BreastCancer[which(complete.cases(BreastCancer)==TRUE),]
dim(BreastCancer)
```

```
## [1] 683  11
```

```r
head(BreastCancer)
```

Id	Cl.thickness	Cell.size	Cell.shape	Marg.adhesion	Epith.c.size	Bare.nuclei	Bl.cromatin	Normal.nucleoli	Mitoses	Class
10000255		1	1	1	2	1	3	1	1	benign
10029455		4	4	5	7	10	3	2	1	benign
10154253		1	1	1	2	2	3	1	1	benign
10162776		8	8	1	3	4	3	7	1	benign
10170234		1	1	3	2	1	3	1	1	benign
10171228		10	10	8	7	10	9	7	1	malignant

```r
?BreastCancer
```

```
## Wisconsin Breast Cancer Database
##
## Description:
##
##      The objective is to identify each of a number of benign or
##      malignant classes. Samples arrive periodically as Dr. Wolberg
##      reports his clinical cases.  The database therefore reflects this
##      chronological grouping of the data.  This grouping information
##      appears immediately below, having been removed from the data
##      itself.  Each variable except for the first was converted into 11
##      primitive numerical attributes with values ranging from 0 through
##      10.  There are 16 missing attribute values. See cited below for
##      more details.
##
## Format:
```

```
##
##      A data frame with 699 observations on 11 variables, one being a
##      character variable, 9 being ordered or nominal, and 1 target
##      class.
##
##        [,1]   Id              Sample code number
##        [,2]   Cl.thickness    Clump Thickness
##        [,3]   Cell.size       Uniformity of Cell Size
##        [,4]   Cell.shape      Uniformity of Cell Shape
##        [,5]   Marg.adhesion   Marginal Adhesion
##        [,6]   Epith.c.size    Single Epithelial Cell Size
##        [,7]   Bare.nuclei     Bare Nuclei
##        [,8]   Bl.cromatin     Bland Chromatin
##        [,9]   Normal.nucleoli Normal Nucleoli
##        [,10]  Mitoses         Mitoses
##        [,11]  Class           Class
```

```r
str(BreastCancer)
```

```
## 'data.frame':    683 obs. of  11 variables:
##  $ Id             : chr  "1000025" "1002945" "1015425" "1016277" ...
##  $ Cl.thickness   : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 5 5 3 6 4 8 1 2 2 4 ...
##  $ Cell.size      : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 4 1 8 1 10 1 1 1 2 ...
##  $ Cell.shape     : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 4 1 8 1 10 1 2 1 1 ...
##  $ Marg.adhesion  : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 5 1 1 3 8 1 1 1 1 ...
##  $ Epith.c.size   : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 2 7 2 3 2 7 2 2 2 2 ...
##  $ Bare.nuclei    : Factor w/ 10 levels "1","2","3","4",..: 1 10 2 4 1 10 10 1 1 1 ...
##  $ Bl.cromatin    : Factor w/ 10 levels "1","2","3","4",..: 3 3 3 3 3 9 3 3 1 2 ...
##  $ Normal.nucleoli: Factor w/ 10 levels "1","2","3","4",..: 1 2 1 7 1 7 1 1 1 1 ...
##  $ Mitoses        : Factor w/ 9 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 5 1 ...
##  $ Class          : Factor w/ 2 levels "benign","malignant": 1 1 1 1 1 2 1 1 1 1 ...
```

Since `neuralnet` only deals with quantitative variables, you can convert all the qualitative variables (factors) to binary ("dummy"), with the `model.matrix` function.

```r
y = as.matrix(BreastCancer[,11])
y[which(y=="benign")] = 0
y[which(y=="malignant")] = 1
y = as.numeric(y)
x = model.matrix(~0 + Cl.thickness + Cell.size + Cell.shape +
                        Marg.adhesion + Epith.c.size + Bare.nuclei + Bl.cromatin +
                        Normal.nucleoli + Mitoses, data = BreastCancer)
dim(x)
```

```
## [1] 683  81
```

```r
df <- data.frame(y,x)
```

Let's split the data into training and validation:

```r
set.seed(92739847)
ids <- sample(1:dim(df)[1],size=dim(df)[1])
df_jumbled <- df[ids,]
train <- df_jumbled[1:450,]
test <- df_jumbled[451:nrow(df_jumbled),]
```

Fitting with neuralnet

In the first example, we will create a simple neural network with minimum effort using the `neuralnet` package. The `neuralnet` package is outdated, but it is still popular among the R community.

The `neuralnet` function is simple. It doesn't provide us the freedom to create fully customizable model architecture.In our case, we are providing it with a machine-learning formula and data, just like GLM. The formula consists of target variables and features.

```r
library("neuralnet")
?neuralnet
```

```
Training of neural networks

Description:

     Train neural networks using backpropagation, resilient
     backpropagation (RPROP) with (Riedmiller, 1994) or without weight
     backtracking (Riedmiller and Braun, 1993) or the modified globally
     convergent version (GRPROP) by Anastasiadis et al. (2005). The
     function allows flexible settings through custom-choice of error
     and activation function. Furthermore, the calculation of
     generalized weights (Intrator O. and Intrator N., 1993) is
     implemented.

Usage:

     neuralnet(formula, data, hidden = 1, threshold = 0.01,
       stepmax = 1e+05, rep = 1, startweights = NULL,
       learningrate.limit = NULL, learningrate.factor = list(minus = 0.5,
       plus = 1.2), learningrate = NULL, lifesign = "none",
       lifesign.step = 1000, algorithm = "rprop+", err.fct = "sse",
       act.fct = "logistic", linear.output = TRUE, exclude = NULL,
       constant.weights = NULL, likelihood = FALSE)

Arguments:

 formula: a symbolic description of the model to be fitted.

    data: a data frame containing the variables specified in 'formula'.

  hidden: a vector of integers specifying the number of hidden neurons
          (vertices) in each layer.

threshold: a numeric value specifying the threshold for the partial
          derivatives of the error function as stopping criteria.

 stepmax: the maximum steps for the training of the neural network.
          Reaching this maximum leads to a stop of the neural network's
          training process.

     rep: the number of repetitions for the neural network's training.

startweights: a vector containing starting values for the weights.  Set
          to 'NULL' for random initialization.
```

```
learningrate.limit: a vector or a list containing the lowest and
          highest limit for the learning rate. Used only for RPROP and
          GRPROP.

learningrate.factor: a vector or a list containing the multiplication
          factors for the upper and lower learning rate. Used only for
          RPROP and GRPROP.

learningrate: a numeric value specifying the learning rate used by
          traditional backpropagation. Used only for traditional
          backpropagation.

lifesign: a string specifying how much the function will print during
          the calculation of the neural network. 'none', 'minimal' or
          'full'.

lifesign.step: an integer specifying the stepsize to print the minimal
          threshold in full lifesign mode.

algorithm: a string containing the algorithm type to calculate the
          neural network. The following types are possible: 'backprop',
          'rprop+', 'rprop-', 'sag', or 'slr'. 'backprop' refers to
          backpropagation, 'rprop+' and 'rprop-' refer to the resilient
          backpropagation with and without weight backtracking, while
          'sag' and 'slr' induce the usage of the modified globally
          convergent algorithm (grprop). See Details for more
          information.

 err.fct: a differentiable function that is used for the calculation of
          the error. Alternatively, the strings 'sse' and 'ce' which
          stand for the sum of squared errors and the cross-entropy can
          be used.

 act.fct: a differentiable function that is used for smoothing the
          result of the cross product of the covariate or neurons and
          the weights. Additionally the strings, 'logistic' and 'tanh'
          are possible for the logistic function and tangent
          hyperbolicus.

linear.output: logical. If act.fct should not be applied to the output
          neurons set linear output to TRUE, otherwise to FALSE.

 exclude: a vector or a matrix specifying the weights, that are
          excluded from the calculation. If given as a vector, the
          exact positions of the weights must be known. A matrix with
          n-rows and 3 columns will exclude n weights, where the first
          column stands for the layer, the second column for the input
          neuron and the third column for the output neuron of the
          weight.

constant.weights: a vector specifying the values of the weights that
          are excluded from the training process and treated as fix.
```

```
likelihood: logical. If the error function is equal to the negative
          log-likelihood function, the information criteria AIC and BIC
          will be calculated. Furthermore the usage of
          confidence.interval is meaningfull.

Details:

     The globally convergent algorithm is based on the resilient
     backpropagation without weight backtracking and additionally
     modifies one learning rate, either the learningrate associated
     with the smallest absolute gradient (sag) or the smallest
     learningrate (slr) itself. The learning rates in the grprop
     algorithm are limited to the boundaries defined in
     learningrate.limit.
```

Now we'll fit a neural network. How many layers?

- There's a lot of discussion on this. See http://www.faqs.org/faqs/ai-faq/neural-nets/part1/preamble.html for some.
- One thing that is agreed upon is that the situations in which performance improves with a second (or third, etc.) hidden layer are very small.
- **One hidden layer is sufficient for the large majority of problems.**

How about the size of the hidden layer(s):

- A common rule of thumb is that the optimal size of the hidden layer is usually between the size of the input and size of the output layers.
- One way to choose would be to use the mean.

In this example we'll use 81 input layers and 1 output layer, so we'll choose 40.

```r
Breast_NN <- neuralnet(y ~ ., data = train,
                       hidden = 40, lifesign = "minimal", algorithm="backprop",
                       linear.output = FALSE, err.fct = 'ce',
                       learningrate=0.1)
```

```
## hidden: 40    thresh: 0.01    rep: 1/1    steps:    2282 error: 0.04201  time: 9.63 secs
```

```r
# Look at the error and activation function
Breast_NN$err.fct
```

```
## function (x, y)
## {
##     -(y * log(x) + (1 - y) * log(1 - x))
## }
## <bytecode: 0x111558fc0>
## <environment: 0x11155c060>
## attr(,"type")
## [1] "ce"
```

```r
Breast_NN$act.fct
```

```
## function (x)
## {
##     1/(1 + exp(-x))
## }
## <bytecode: 0x11154e4b0>
## <environment: 0x111551ac8>
## attr(,"type")
## [1] "logistic"
```

```
# Here are the parameters:
dim(Breast_NN$result.matrix)
```

```
## [1] 3324    1
```

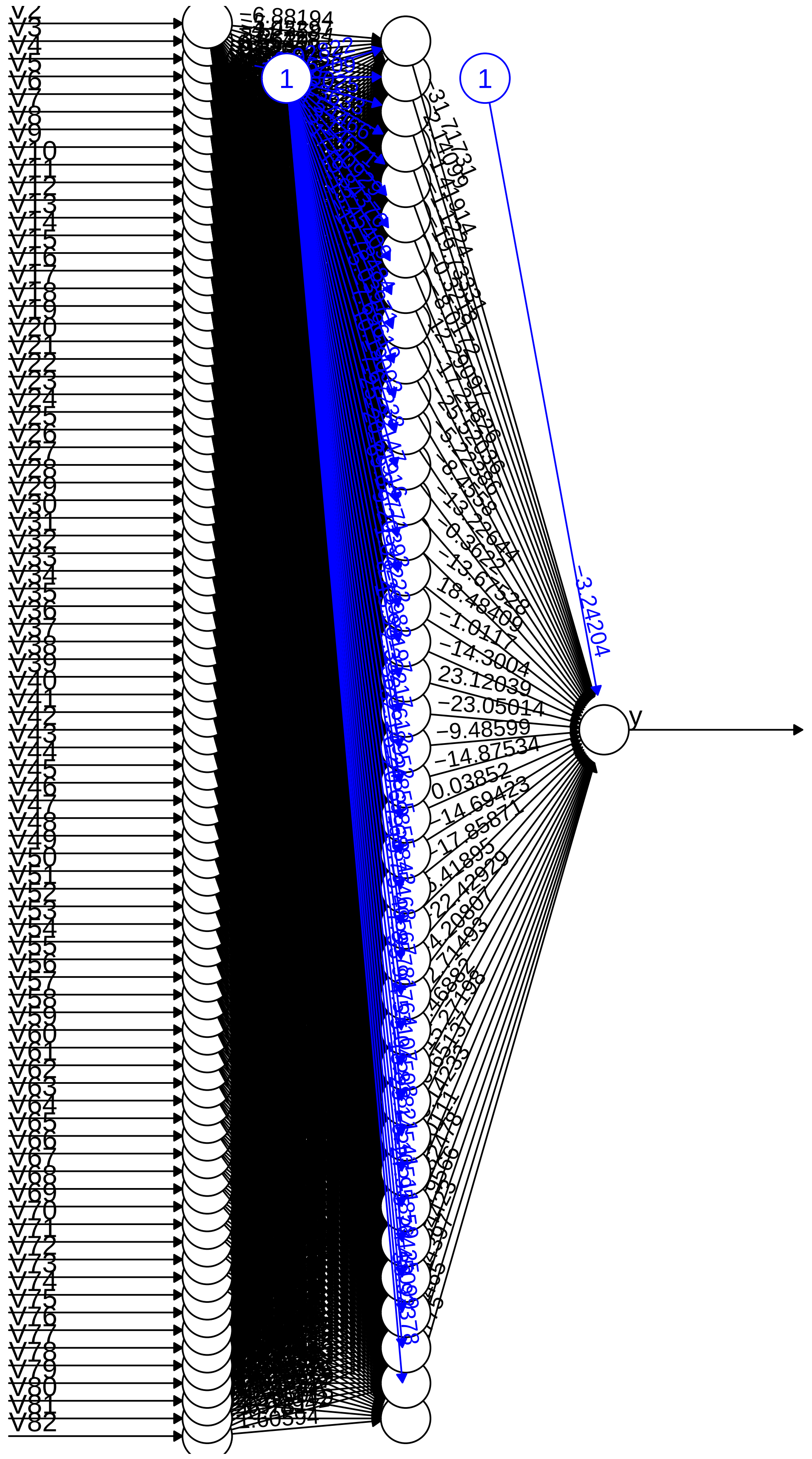
```
3321/81
```

```
## [1] 41
```

```
head(data.frame(Breast_NN$result.matrix))
```

	Breast_NN.result.matrix
error	0.0420063
reached.threshold	0.0099975
steps	2282.0000000
Intercept.to.1layhid1	-7.3351334
Cl.thickness1.to.1layhid1	1.9221154
Cl.thickness2.to.1layhid1	1.6221521

```
#plot(Breast_NN, rep="best")
```


V2
V3
V4
V5
V6
V7
V8
V9
V10
V11
V12
V13
V14
V15
V16
V17
V18
V19
V20
V21
V22
V23
V24
V25
V26
V27
V28
V29
V30
V31
V32
V33
V34
V35
V36
V37
V38
V39
V40
V41
V42
V43
V44
V45
V46
V47
V48
V49
V50
V51
V52
V53
V54
V55
V56
V57
V58
V59
V60
V61
V62
V63
V64
V65
V66
V67
V68
V69
V70
V71
V72
V73
V74
V75
V76
V77
V78
V79
V80
V81
V82
−6.88194
1.60594
1
1
−3.24204
−0.3623
18.48409
−1.0117
−14.3004
23.12039
−23.05014
−9.48599
−14.87534
0.03852
−14.69423
−17.85871
y

```
valid_pred <- predict(Breast_NN, newdata = test)
table(valid_pred>0.5, test$y)
```

/	0	1
FALSE	152	9
TRUE	1	71

```
# Accuracy
sum(diag(table(valid_pred>0.5, test$y)))/length(test$y)
```

```
## [1] 0.9570815
```

Fitting with `TensorFlow`

`TensorFlow` is an open source deep neural net framework, based on a graphical model. It is more than just a neural net platform, and supports numerical computing based on data flow graphs. Data may be represented in n-dimensional structures like vectors and matrices, or higher-dimensional tensors. Because these mathematical objects are folded into a data flow graph for computation, the moniker for this software library is an obvious one.

The computations for deep learning nets involve tensor computations, which are known to be implemented more efficiently on GPUs than CPUs. Therefore like other deep learning libraries, `TensorFlow` may be implemented on CPUs and GPUs. The generality and speed of the `TensorFlow` software, ease of installation, its documentation and examples, and runnability on multiple platforms has made `TensorFlow` the most popular deep learning toolkit today. (Opinions on this may, of course, differ.) There is a wealth of tutorial material on TF of very high quality that you may refer to here: https://www.tensorflow.org/tutorials/.

Rather than run `TensorFlow` natively, it is often easier to use it through an easy to use interface program. One of the most popular high-level APIs is `Keras` See: https://keras.io/. The site contains several examples, which make it easy to get up and running. Though originally written in Python, `Keras` has been extended to `R` via the `keras` package.

`Keras` is a high-level neural networks API developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research. Keras has the following key features:

- Allows the same code to run on CPU or on GPU, seamlessly.
- User-friendly API which makes it easy to quickly prototype deep learning models.
- Built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both.
- Supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, etc. This means that Keras is appropriate for building essentially any deep learning model, from a memory network to a neural Turing machine.

```
library(tensorflow)
library(keras)
library(reticulate)
```

To use these functions were going to convert the outcome (benign, malignant) in the data set to "one-hot encoding" using the to_categorial function. What is one-hot encoding replaces a single vector of 1s (malignant) and 0s (benign) to a bivariate dependent variable as a two-column matrix, where the first column contains a 1 if the cell is benign and the second column a 1 if the cell is malignant, for each row of the data set. This is because TensorFlow/Keras requires this format as input, to facilitate tensor calculations.

```r
y_train = as.matrix(train[,1])
y_test = as.matrix(test[,1])
x_train = as.matrix(train[,-1])
x_test = as.matrix(test[,-1])

Y_train = to_categorical(y_train,2)
Y_test = to_categorical(y_test,2)
```

`TensorFlow` is structured in a manner where one describes the neural net first, layer by layer. There is not a single function that is called, which generates the deep learning net, and runs the model. Here, we first describe for each layer in the neural net, the number of nodes, the type of activation function, and any other hyperparameters needed in the model fitting stage, such as the extent of dropout for example.

In the output layer we also state the nature of the activation function, such as sigmoid or softmax. And then, we specify a compile function which describes the loss function that will be minimized, along with the minimization algorithm. At this point in the program specification, the model is not actually run. See below for the code block that builds up the deep learning network.

The core data structure of Keras is a model, a way to organize layers. The simplest type of model is the Sequential model, a linear stack of layers.

We begin by creating a sequential model and then adding layers using the pipe (`%>%`) operator:

```r
?keras_model_sequential
```

```
Keras Model composed of a linear stack of layers

Description:

     Keras Model composed of a linear stack of layers

Usage:

     keras_model_sequential(layers = NULL, name = NULL, ...)

Arguments:

  layers: List of layers to add to the model

    name: Name of model

     ...: Arguments passed on to 'sequential_model_input_layer'

          'input_shape' an integer vector of dimensions (not including
              the batch axis), or a 'tf$TensorShape' instance (also not
              including the batch axis).

          'batch_size' Optional input batch size (integer or NULL).

          'dtype' Optional datatype of the input. When not provided,
              the Keras default float type will be used.

          'input_tensor' Optional tensor to use as layer input. If set,
              the layer will use the 'tf$TypeSpec' of this tensor
              rather than creating a new placeholder tensor.
          'sparse' Boolean, whether the placeholder created is meant to
              be sparse. Default to 'FALSE'.

          'ragged' Boolean, whether the placeholder created is meant to
              be ragged. In this case, values of 'NULL' in the 'shape'
              argument represent ragged dimensions. For more
              information about 'RaggedTensors', see this guide.
              Default to 'FALSE'.

          'type_spec' A 'tf$TypeSpec' object to create Input from. This
              'tf$TypeSpec' represents the entire batch. When provided,
              all other args except name must be 'NULL'.

          'input_layer_name,name' Optional name of the input layer
              (string).
```

```r
model <- keras_model_sequential()
n <- nrow(x_train)
model %>%
  layer_dense(units = n/2, activation = 'relu', input_shape = c(ncol(x_train))) %>%
  # set rate: the fraction of the input units to drop.
  layer_dropout(rate = 0.25) %>%
  layer_dense(units = n/10, activation = 'relu') %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 2, activation = 'softmax')
```

The `input_shape` argument to the first layer specifies the shape of the input data (a length `ncol(x_train)` numeric vector). The final layer outputs a length 2 numeric vector (probabilities for each class) using a softmax activation function.

```r
summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  dense_2 (Dense)                    (None, 225)                     18450
##  dropout_1 (Dropout)                (None, 225)                     0
##  dense_1 (Dense)                    (None, 45)                      10170
##  dropout (Dropout)                  (None, 45)                      0
##  dense (Dense)                      (None, 2)                       92
## ================================================================================
## Total params: 28712 (112.16 KB)
## Trainable params: 28712 (112.16 KB)
## Non-trainable params: 0 (0.00 Byte)
## ________________________________________________________________________________
```

Next, compile the model with appropriate loss function, optimizer, and metrics:

```r
?compile.keras.engine.training.Model
```

```
Configure a Keras model for training

Description:

     Configure a Keras model for training
```

```
Usage:

     ## S3 method for class 'keras.engine.training.Model'
     compile(
       object,
       optimizer = NULL,
       loss = NULL,
       metrics = NULL,
       loss_weights = NULL,
       weighted_metrics = NULL,
       run_eagerly = NULL,
       steps_per_execution = NULL,
       ...,
       target_tensors = NULL,
       sample_weight_mode = NULL
     )

Arguments:

  object: Model object to compile.

optimizer: String (name of optimizer) or optimizer instance. For most
          models, this defaults to '"rmsprop"'

    loss: String (name of objective function), objective function or a
          'keras$losses$Loss' subclass instance. An objective function
          is any callable with the signature 'loss = fn(y_true,
          y_pred)', where y_true = ground truth values with shape =
          [batch_size, d0, .. dN], except sparse loss functions such as
          sparse categorical crossentropy where shape = [batch_size,
          d0, .. dN-1]. y_pred = predicted values with shape =
          [batch_size, d0, .. dN]. It returns a weighted loss float
          tensor. If a custom 'Loss' instance is used and reduction is
          set to 'NULL', return value has the shape [batch_size, d0, ..
          dN-1] i.e. per-sample or per-timestep loss values; otherwise,
          it is a scalar. If the model has multiple outputs, you can
          use a different loss on each output by passing a dictionary
          or a list of losses. The loss value that will be minimized by
          the model will then be the sum of all individual losses,
          unless 'loss_weights' is specified.

 metrics: List of metrics to be evaluated by the model during training
          and testing. Each of this can be a string (name of a built-in
          function), function or a 'keras$metrics$Metric' class
          instance. See '?tf$keras$metrics'. Typically you will use
          'metrics=list('accuracy')'. A function is any callable with
          the signature 'result = fn(y_true, y_pred)'. To specify
          different metrics for different outputs of a multi-output
          model, you could also pass a dictionary, such as
          'metrics=list(output_a = 'accuracy', output_b = c('accuracy',
          'mse'))'. You can also pass a list to specify a metric or a
          list of metrics for each output, such as
          'metrics=list(list('accuracy'), list('accuracy', 'mse'))' or
          'metrics=list('accuracy', c('accuracy', 'mse'))'. When you
          pass the strings ''accuracy'' or ''acc'', this is converted
          to one of 'tf.keras.metrics.BinaryAccuracy',
          'tf.keras.metrics.CategoricalAccuracy',
          'tf.keras.metrics.SparseCategoricalAccuracy' based on the
          loss function used and the model output shape. A similar
          conversion is done for the strings ''crossentropy'' and
          ''ce''.

loss_weights: Optional list, dictionary, or named vector specifying
          scalar numeric coefficients to weight the loss contributions
          of different model outputs. The loss value that will be
          minimized by the model will then be the _weighted sum_ of all
          individual losses, weighted by the 'loss_weights'
          coefficients. If a list, it is expected to have a 1:1 mapping
          to the model's outputs. If a dict, it is expected to map
          output names (strings) to scalar coefficients.

weighted_metrics: List of metrics to be evaluated and weighted by
          'sample_weight' or 'class_weight' during training and
          testing.

run_eagerly: Bool. Defaults to 'FALSE'. If 'TRUE', this Model's logic
          will not be wrapped in a 'tf.function'. Recommended to leave
          this as 'NULL' unless your Model cannot be run inside a
          'tf.function'. 'run_eagerly=True' is not supported when using
          'tf.distribute.experimental.ParameterServerStrategy'. If the
          model's logic uses tensors in R control flow expressions like
          'if' and 'for', the model is still traceable with
          'tf.function', but you will have to enter a
          'tfautograph::autograph({})' directly.

steps_per_execution: Int. Defaults to 1. The number of batches to run
          during each 'tf.function' call. Running multiple batches
          inside a single 'tf.function' call can greatly improve
          performance on TPUs or small models with a large Python/R
          overhead. At most, one full epoch will be run each execution.
          If a number larger than the size of the epoch is passed, the
          execution will be truncated to the size of the epoch. Note
          that if 'steps_per_execution' is set to 'N',
          'Callback.on_batch_begin' and 'Callback.on_batch_end' methods
          will only be called every 'N' batches (i.e. before/after each
          'tf.function' execution).

     ...: Arguments supported for backwards compatibility only.

target_tensors: By default, Keras will create a placeholder for the
          model's target, which will be fed with the target data during
          training. If instead you would like to use your own target
          tensor (in turn, Keras will not expect external data for
          these targets at training time), you can specify them via the
          'target_tensors' argument. It should be a single tensor (for
          a single-output sequential model).

sample_weight_mode: If you need to do timestep-wise sample weighting
```

```
          (2D weights), set this to "temporal". 'NULL' defaults to
          sample-wise weights (1D). If the model has multiple outputs,
          you can use a different 'sample_weight_mode' on each output
          by passing a list of modes.
```

```r
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)
```

From this I got the following message:

WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.RMSprop` runs slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at `tf.keras.optimizers.legacy.RMSprop`.

This error apparently happens when trying to use ADAM (on python or R). The following does work:

```r
model %>% compile(
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)
```

Training and Evaluation

Use the `fit()` function to train the model for 30 epochs using batches of 90 subjects:

```r
?fit.keras.engine.training.Model
```

```
Train a Keras model

Description:

     Trains the model for a fixed number of epochs (iterations on a
     dataset).

Usage:

     ## S3 method for class 'keras.engine.training.Model'
     fit(
       object,
       x = NULL,
       y = NULL,
       batch_size = NULL,
       epochs = 10,
       verbose = getOption("keras.fit_verbose", default = "auto"),
       callbacks = NULL,
       view_metrics = getOption("keras.view_metrics", default = "auto"),
       validation_split = 0,
       validation_data = NULL,
       shuffle = TRUE,
       class_weight = NULL,
       sample_weight = NULL,
       initial_epoch = 0,
       steps_per_epoch = NULL,
       validation_steps = NULL,
       ...
     )
```

```
Arguments:

  object: Model to train.

       x: Vector, matrix, or array of training data (or list if the
          model has multiple inputs). If all inputs in the model are
          named, you can also pass a list mapping input names to data.
          'x' can be 'NULL' (default) if feeding from framework-native
          tensors (e.g. TensorFlow data tensors). You can also pass a
          'tfdataset' or a generator returning a list with (inputs,
          targets) or (inputs, targets, sample_weights).

       y: Vector, matrix, or array of target (label) data (or list if
          the model has multiple outputs). If all outputs in the model
          are named, you can also pass a list mapping output names to
          data. 'y' can be 'NULL' (default) if feeding from
          framework-native tensors (e.g. TensorFlow data tensors).

batch_size: Integer or 'NULL'. Number of samples per gradient update.
          If unspecified, 'batch_size' will default to 32.

  epochs: Number of epochs to train the model. Note that in conjunction
          with 'initial_epoch', 'epochs' is to be understood as "final
          epoch". The model is not trained for a number of iterations
          given by 'epochs', but merely until the epoch of index
          'epochs' is reached.

 verbose: Verbosity mode (0 = silent, 1 = progress bar, 2 = one line
          per epoch). Defaults to 1 in most contexts, 2 if in knitr
          render or running on a distributed training server.

callbacks: List of callbacks to be called during training.

view_metrics: View realtime plot of training metrics (by epoch). The
          default ('"auto"') will display the plot when running within
          RStudio, 'metrics' were specified during model 'compile()',
          'epochs > 1' and 'verbose > 0'. Use the global
          'keras.view_metrics' option to establish a different default.

validation_split: Float between 0 and 1. Fraction of the training data
          to be used as validation data. The model will set apart this
          fraction of the training data, will not train on it, and will
          evaluate the loss and any model metrics on this data at the
          end of each epoch. The validation data is selected from the
          last samples in the 'x' and 'y' data provided, before
          shuffling.

validation_data: Data on which to evaluate the loss and any model
          metrics at the end of each epoch. The model will not be
          trained on this data. This could be a list (x_val, y_val) or
          a list (x_val, y_val, val_sample_weights). 'validation_data'
          will override 'validation_split'.
 shuffle: shuffle: Logical (whether to shuffle the training data before
          each epoch) or string (for "batch"). "batch" is a special
          option for dealing with the limitations of HDF5 data; it
          shuffles in batch-sized chunks. Has no effect when
          'steps_per_epoch' is not 'NULL'.

class_weight: Optional named list mapping indices (integers) to a
          weight (float) value, used for weighting the loss function
          (during training only). This can be useful to tell the model
          to "pay more attention" to samples from an under-represented
          class.

sample_weight: Optional array of the same length as x, containing
          weights to apply to the model's loss for each sample. In the
          case of temporal data, you can pass a 2D array with shape
          (samples, sequence_length), to apply a different weight to
          every timestep of every sample. In this case you should make
          sure to specify 'sample_weight_mode="temporal"' in
          'compile()'.

initial_epoch: Integer, Epoch at which to start training (useful for
          resuming a previous training run).

steps_per_epoch: Total number of steps (batches of samples) before
          declaring one epoch finished and starting the next epoch.
          When training with input tensors such as TensorFlow data
          tensors, the default 'NULL' is equal to the number of samples
          in your dataset divided by the batch size, or 1 if that
          cannot be determined.

validation_steps: Only relevant if 'steps_per_epoch' is specified.
          Total number of steps (batches of samples) to validate before
          stopping.

     ...: Unused
```

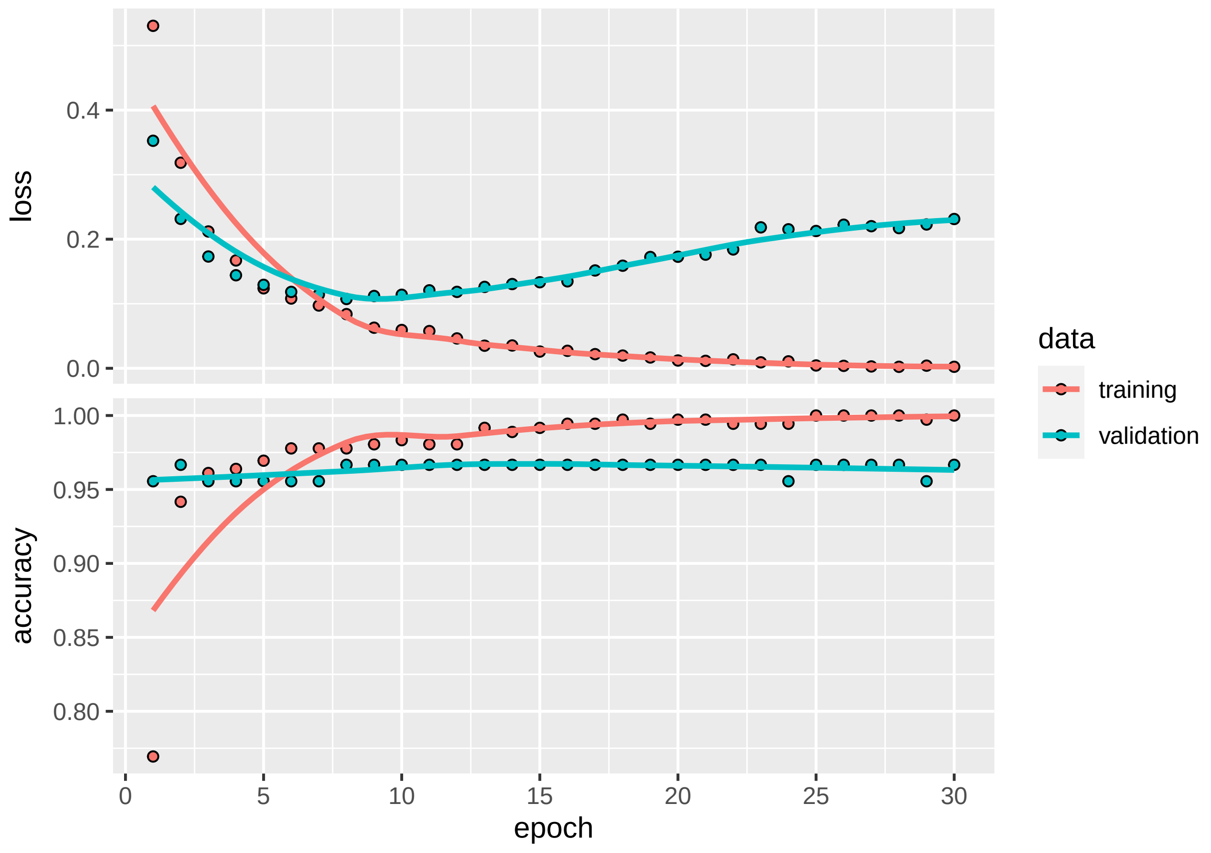
```r
history <- model %>% fit(
  x_train, Y_train,
  epochs = 30, batch_size = 90,
  validation_split = 0.2
)
```

```
Epoch 1/30
4/4 - 1s - loss: 0.5306 - accuracy: 0.7694 - val_loss: 0.3525 - val_accuracy: 0.9556 - 715ms/epoch - 17
Epoch 2/30
4/4 - 0s - loss: 0.3185 - accuracy: 0.9417 - val_loss: 0.2315 - val_accuracy: 0.9667 - 71ms/epoch - 18m
Epoch 3/30
4/4 - 0s - loss: 0.2120 - accuracy: 0.9611 - val_loss: 0.1731 - val_accuracy: 0.9556 - 40ms/epoch - 10m
Epoch 4/30
4/4 - 0s - loss: 0.1671 - accuracy: 0.9639 - val_loss: 0.1442 - val_accuracy: 0.9556 - 45ms/epoch - 11m
Epoch 5/30
4/4 - 0s - loss: 0.1239 - accuracy: 0.9694 - val_loss: 0.1293 - val_accuracy: 0.9556 - 60ms/epoch - 15m
Epoch 6/30
4/4 - 0s - loss: 0.1081 - accuracy: 0.9778 - val_loss: 0.1183 - val_accuracy: 0.9556 - 48ms/epoch - 12m
Epoch 7/30
```

```
4/4 - 0s - loss: 0.0973 - accuracy: 0.9778 - val_loss: 0.1143 - val_accuracy: 0.9556 - 45ms/epoch - 11ms
Epoch 8/30
4/4 - 0s - loss: 0.0839 - accuracy: 0.9778 - val_loss: 0.1073 - val_accuracy: 0.9667 - 48ms/epoch - 12ms
Epoch 9/30
4/4 - 0s - loss: 0.0629 - accuracy: 0.9806 - val_loss: 0.1120 - val_accuracy: 0.9667 - 40ms/epoch - 10ms
Epoch 10/30
4/4 - 0s - loss: 0.0593 - accuracy: 0.9833 - val_loss: 0.1139 - val_accuracy: 0.9667 - 40ms/epoch - 10ms
Epoch 11/30
4/4 - 0s - loss: 0.0576 - accuracy: 0.9806 - val_loss: 0.1208 - val_accuracy: 0.9667 - 32ms/epoch - 8ms/
Epoch 12/30
4/4 - 0s - loss: 0.0463 - accuracy: 0.9806 - val_loss: 0.1184 - val_accuracy: 0.9667 - 36ms/epoch - 9ms/
Epoch 13/30
4/4 - 0s - loss: 0.0350 - accuracy: 0.9917 - val_loss: 0.1261 - val_accuracy: 0.9667 - 40ms/epoch - 10ms
Epoch 14/30
4/4 - 0s - loss: 0.0353 - accuracy: 0.9889 - val_loss: 0.1305 - val_accuracy: 0.9667 - 34ms/epoch - 8ms/
Epoch 15/30
4/4 - 0s - loss: 0.0259 - accuracy: 0.9917 - val_loss: 0.1333 - val_accuracy: 0.9667 - 25ms/epoch - 6ms/
Epoch 16/30
4/4 - 0s - loss: 0.0271 - accuracy: 0.9944 - val_loss: 0.1347 - val_accuracy: 0.9667 - 39ms/epoch - 10ms
Epoch 17/30
4/4 - 0s - loss: 0.0219 - accuracy: 0.9944 - val_loss: 0.1516 - val_accuracy: 0.9667 - 39ms/epoch - 10ms
Epoch 18/30
4/4 - 0s - loss: 0.0197 - accuracy: 0.9972 - val_loss: 0.1589 - val_accuracy: 0.9667 - 30ms/epoch - 7ms/
Epoch 19/30
4/4 - 0s - loss: 0.0168 - accuracy: 0.9944 - val_loss: 0.1724 - val_accuracy: 0.9667 - 31ms/epoch - 8ms/
Epoch 20/30
4/4 - 0s - loss: 0.0120 - accuracy: 0.9972 - val_loss: 0.1728 - val_accuracy: 0.9667 - 31ms/epoch - 8ms/
Epoch 21/30
4/4 - 0s - loss: 0.0115 - accuracy: 0.9972 - val_loss: 0.1761 - val_accuracy: 0.9667 - 64ms/epoch - 16ms
Epoch 22/30
4/4 - 0s - loss: 0.0138 - accuracy: 0.9944 - val_loss: 0.1841 - val_accuracy: 0.9667 - 59ms/epoch - 15ms
Epoch 23/30
4/4 - 0s - loss: 0.0092 - accuracy: 0.9944 - val_loss: 0.2183 - val_accuracy: 0.9667 - 43ms/epoch - 11ms
Epoch 24/30
4/4 - 0s - loss: 0.0107 - accuracy: 0.9944 - val_loss: 0.2153 - val_accuracy: 0.9556 - 41ms/epoch - 10ms
Epoch 25/30
4/4 - 0s - loss: 0.0044 - accuracy: 1.0000 - val_loss: 0.2127 - val_accuracy: 0.9667 - 37ms/epoch - 9ms/
Epoch 26/30
4/4 - 0s - loss: 0.0038 - accuracy: 1.0000 - val_loss: 0.2225 - val_accuracy: 0.9667 - 44ms/epoch - 11ms
Epoch 27/30
4/4 - 0s - loss: 0.0029 - accuracy: 1.0000 - val_loss: 0.2202 - val_accuracy: 0.9667 - 35ms/epoch - 9ms/
Epoch 28/30
4/4 - 0s - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.2172 - val_accuracy: 0.9667 - 87ms/epoch - 22ms
Epoch 29/30
4/4 - 0s - loss: 0.0041 - accuracy: 0.9972 - val_loss: 0.2228 - val_accuracy: 0.9556 - 33ms/epoch - 8ms/
Epoch 30/30
4/4 - 0s - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.2313 - val_accuracy: 0.9667 - 25ms/epoch - 6ms/
```

The `history` object returned by `fit()` includes loss and accuracy metrics which we can plot:

```
plot(history)
```



Evaluate the model's performance on the test data:

```
model %>% evaluate(x_test, Y_test)
```

```
## 8/8 - 0s - loss: 0.0582 - accuracy: 0.9785 - 108ms/epoch - 14ms/step
```

```
##       loss   accuracy
## 0.05818005 0.97854078
```

Generate predictions on new data:

```
model %>% predict(x_test) %>% k_argmax()
```

```
## 8/8 - 0s - 69ms/epoch - 9ms/step
```

```
## tf.Tensor(
## [1 1 0 1 1 0 1 0 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1
##  1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 1 1
##  1 1 0 0 0 1 1 0 0 0 1 0 0 1 0 1 0 0 1 1 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 1 0
##  0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0
##  1 0 0 1 0 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0
##  1 0 1 1 0 1 1 0 0 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 0 1 1 0 1 0 1 0 0
##  0 0 0 0 0 1 0 1 0 0 0], shape=(233), dtype=int64)
```

Example with image data

Let's see an example with the minst image data:

```
mnist <- dataset_mnist()
x_train <- mnist$train$x
```

```
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y
dim(x_train)
```

```
## [1] 60000    28    28
```

```
dim(x_test)
```

```
## [1] 10000    28    28
```

```
# reshape
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test  <- array_reshape(x_test, c(nrow(x_test), 784))
```

Now we normalize the values, which are pixel intensities ranging in (0,255) and create the outcome.

```
x_train <- x_train / 255
x_test <- x_test / 255
y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)
```

Now create the model in Keras.

```
model <- keras_model_sequential()
model %>%
layer_dense(units = 256, activation = 'relu', input_shape = c(784)) %>%
layer_dropout(rate = 0.4) %>%
layer_dense(units = 128, activation = 'relu') %>%
layer_dropout(rate = 0.3) %>%
layer_dense(units = 10, activation = 'softmax')
summary(model)
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                     Param #
## ================================================================================
##  dense_5 (Dense)                    (None, 256)                      200960
##  dropout_3 (Dropout)                (None, 256)                      0
##  dense_4 (Dense)                    (None, 128)                      32896
##  dropout_2 (Dropout)                (None, 128)                      0
##  dense_3 (Dense)                    (None, 10)                       1290
## ================================================================================
## Total params: 235146 (918.54 KB)
## Trainable params: 235146 (918.54 KB)
## Non-trainable params: 0 (0.00 Byte)
## ________________________________________________________________________________
```

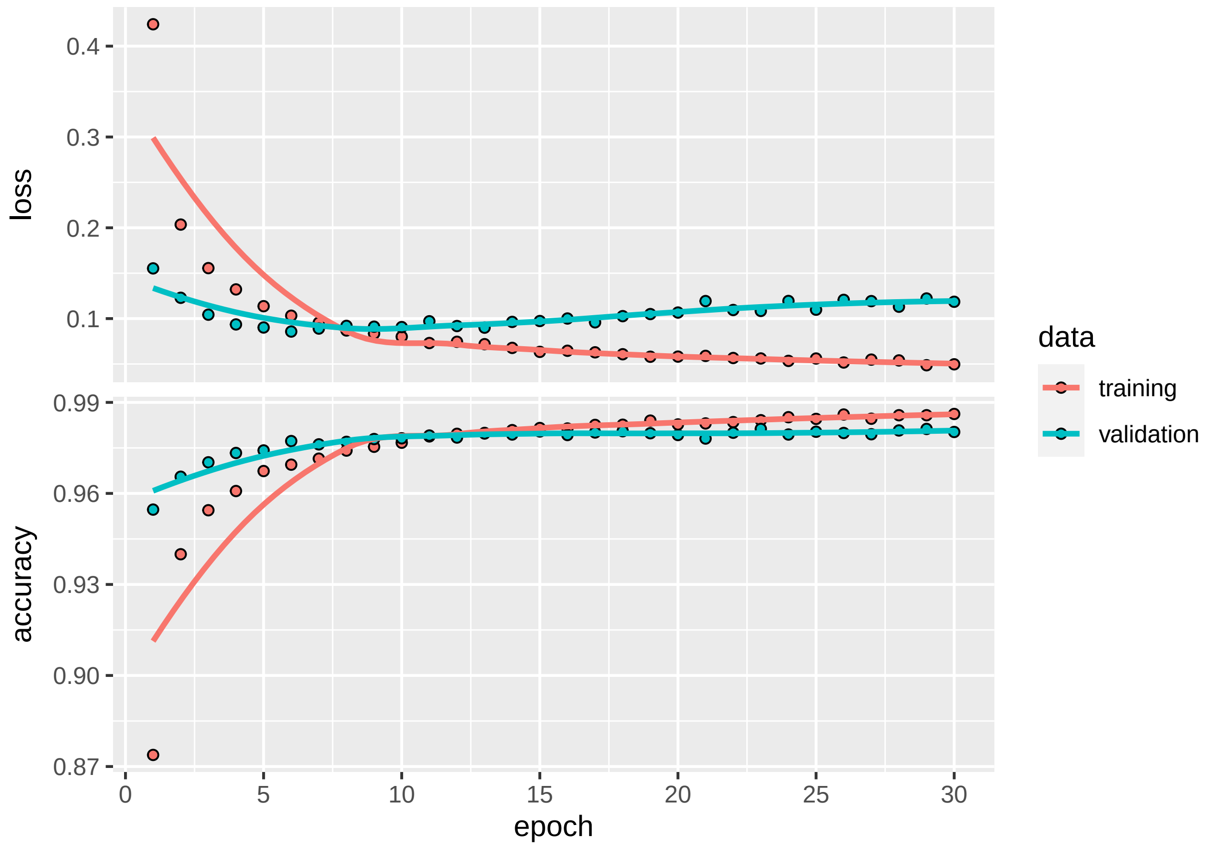
```
model %>% compile(
loss = 'categorical_crossentropy',
metrics = c('accuracy')
)
```

Now run the model to get a fitted deep learning network.

```
history <- model %>% fit(
x_train, y_train,
epochs = 30, batch_size = 128,
validation_split = 0.2
)
```

```
## Epoch 1/30
## 375/375 - 3s - loss: 0.4241 - accuracy: 0.8738 - val_loss: 0.1553 - val_accuracy: 0.9547 - 3s/epoch
## Epoch 2/30
## 375/375 - 2s - loss: 0.2035 - accuracy: 0.9400 - val_loss: 0.1229 - val_accuracy: 0.9655 - 2s/epoch
## Epoch 3/30
## 375/375 - 3s - loss: 0.1556 - accuracy: 0.9545 - val_loss: 0.1043 - val_accuracy: 0.9703 - 3s/epoch
## Epoch 4/30
## 375/375 - 3s - loss: 0.1321 - accuracy: 0.9608 - val_loss: 0.0936 - val_accuracy: 0.9733 - 3s/epoch
## Epoch 5/30
## 375/375 - 2s - loss: 0.1137 - accuracy: 0.9674 - val_loss: 0.0902 - val_accuracy: 0.9742 - 2s/epoch
## Epoch 6/30
## 375/375 - 2s - loss: 0.1032 - accuracy: 0.9695 - val_loss: 0.0857 - val_accuracy: 0.9772 - 2s/epoch
## Epoch 7/30
## 375/375 - 2s - loss: 0.0955 - accuracy: 0.9715 - val_loss: 0.0890 - val_accuracy: 0.9762 - 2s/epoch
## Epoch 8/30
## 375/375 - 2s - loss: 0.0869 - accuracy: 0.9741 - val_loss: 0.0920 - val_accuracy: 0.9770 - 2s/epoch
## Epoch 9/30
## 375/375 - 2s - loss: 0.0835 - accuracy: 0.9754 - val_loss: 0.0911 - val_accuracy: 0.9779 - 2s/epoch
## Epoch 10/30
## 375/375 - 2s - loss: 0.0801 - accuracy: 0.9767 - val_loss: 0.0907 - val_accuracy: 0.9782 - 2s/epoch
## Epoch 11/30
## 375/375 - 2s - loss: 0.0731 - accuracy: 0.9788 - val_loss: 0.0971 - val_accuracy: 0.9791 - 2s/epoch
## Epoch 12/30
## 375/375 - 2s - loss: 0.0745 - accuracy: 0.9796 - val_loss: 0.0918 - val_accuracy: 0.9784 - 2s/epoch
## Epoch 13/30
## 375/375 - 2s - loss: 0.0718 - accuracy: 0.9799 - val_loss: 0.0900 - val_accuracy: 0.9797 - 2s/epoch
## Epoch 14/30
## 375/375 - 2s - loss: 0.0677 - accuracy: 0.9809 - val_loss: 0.0963 - val_accuracy: 0.9794 - 2s/epoch
## Epoch 15/30
## 375/375 - 2s - loss: 0.0634 - accuracy: 0.9816 - val_loss: 0.0973 - val_accuracy: 0.9804 - 2s/epoch
## Epoch 16/30
## 375/375 - 2s - loss: 0.0646 - accuracy: 0.9814 - val_loss: 0.1001 - val_accuracy: 0.9793 - 2s/epoch
## Epoch 17/30
## 375/375 - 2s - loss: 0.0629 - accuracy: 0.9826 - val_loss: 0.0959 - val_accuracy: 0.9801 - 2s/epoch
## Epoch 18/30
## 375/375 - 2s - loss: 0.0607 - accuracy: 0.9826 - val_loss: 0.1027 - val_accuracy: 0.9805 - 2s/epoch
## Epoch 19/30
## 375/375 - 2s - loss: 0.0580 - accuracy: 0.9840 - val_loss: 0.1050 - val_accuracy: 0.9798 - 2s/epoch
## Epoch 20/30
## 375/375 - 2s - loss: 0.0582 - accuracy: 0.9827 - val_loss: 0.1067 - val_accuracy: 0.9793 - 2s/epoch
## Epoch 21/30
## 375/375 - 2s - loss: 0.0590 - accuracy: 0.9831 - val_loss: 0.1193 - val_accuracy: 0.9781 - 2s/epoch
## Epoch 22/30
## 375/375 - 2s - loss: 0.0566 - accuracy: 0.9835 - val_loss: 0.1095 - val_accuracy: 0.9800 - 2s/epoch
## Epoch 23/30
## 375/375 - 2s - loss: 0.0561 - accuracy: 0.9841 - val_loss: 0.1085 - val_accuracy: 0.9812 - 2s/epoch
## Epoch 24/30
## 375/375 - 2s - loss: 0.0534 - accuracy: 0.9851 - val_loss: 0.1194 - val_accuracy: 0.9794 - 2s/epoch
## Epoch 25/30
## 375/375 - 2s - loss: 0.0561 - accuracy: 0.9846 - val_loss: 0.1098 - val_accuracy: 0.9803 - 2s/epoch
## Epoch 26/30
## 375/375 - 2s - loss: 0.0517 - accuracy: 0.9860 - val_loss: 0.1206 - val_accuracy: 0.9799 - 2s/epoch
## Epoch 27/30
## 375/375 - 2s - loss: 0.0549 - accuracy: 0.9846 - val_loss: 0.1192 - val_accuracy: 0.9795 - 2s/epoch
## Epoch 28/30
## 375/375 - 2s - loss: 0.0540 - accuracy: 0.9858 - val_loss: 0.1131 - val_accuracy: 0.9808 - 2s/epoch
## Epoch 29/30
## 375/375 - 2s - loss: 0.0486 - accuracy: 0.9858 - val_loss: 0.1221 - val_accuracy: 0.9812 - 2s/epoch
## Epoch 30/30
## 375/375 - 2s - loss: 0.0497 - accuracy: 0.9862 - val_loss: 0.1185 - val_accuracy: 0.9803 - 2s/epoch
```

```r
plot(history)
```



```r
model %>% evaluate(x_test, y_test)
```

```
## 313/313 - 0s - loss: 0.1102 - accuracy: 0.9816 - 300ms/epoch - 958us/step
```

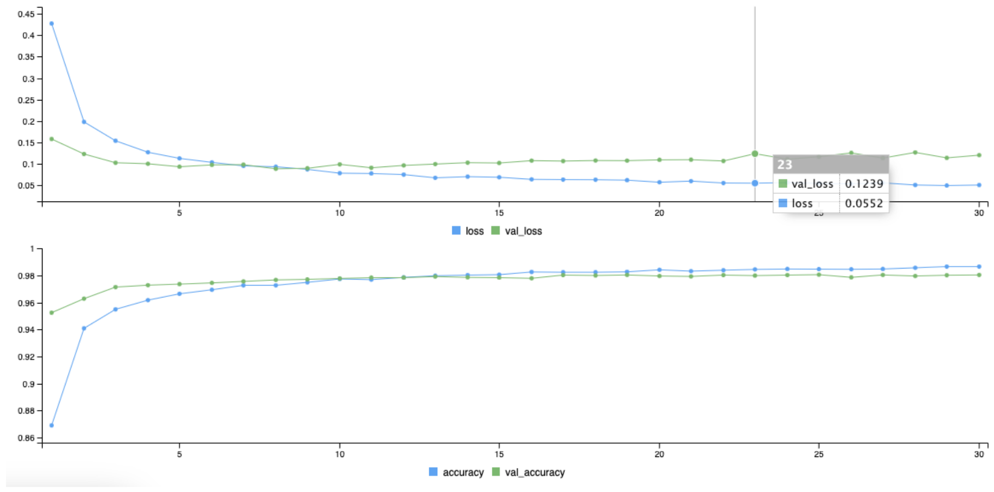
```
##      loss  accuracy
## 0.1101681 0.9816000
```

```r
model %>% predict(x_test) %>% k_argmax()
```

```
## 313/313 - 0s - 378ms/epoch - 1ms/step
```

```
## tf.Tensor([7 2 1 ... 4 5 6], shape=(10000), dtype=int64)
```

The keras package also plots the progress of the model by showing the loss function evolution by epoch, as well as accuracy, for the training and validation samples.



Forest recommended using PyTouch. There is an R package `torch` that allows you to use PyTorch-like functionality natively from R. No Python installation is required: `torch` is built directly on top of libtorch, a C++ library that provides the tensor-computation and automatic-differentiation capabilities essential to building neural networks. See https://torch.mlverse.org/ for more information.

Using `nnet` with some

Now were going to do a Neural Net on some simulated data. We're going to use the **nnet** package for the neural network:

```
library(clusterGeneration)
```

```
## Loading required package: MASS
```

```
library(nnet)
```

Now we'll simulate the data. Part of the difficulty in simulating multivariate normal data is getting a random positive definite matrix. The **clusterGeneration** package helps with this.

```
#define number of variables and observations
set.seed(2)
num.vars<-8
num.obs<-10000

#define correlation matrix for explanatory variables
#define actual parameter values
cov.mat<-genPositiveDefMat(num.vars,covMethod=c("unifcorrmat"))$Sigma
rand.vars<-mvrnorm(num.obs,rep(0,num.vars),Sigma=cov.mat)
parms1<-runif(num.vars,-10,10)
y1<-rand.vars %*% matrix(parms1) + rnorm(num.obs,sd=20)
parms2<-runif(num.vars,-10,10)
y2<-rand.vars %*% matrix(parms2) + rnorm(num.obs,sd=20)

#prep data and create neural network
rand.vars<-data.frame(rand.vars)
resp<-apply(cbind(y1,y2),2, function(y) (y-min(y))/(max(y)-min(y)))
resp<-data.frame(resp)
names(resp)<-c('Y1','Y2')
```

This model has 8 variables with 2 outcomes and a total of 10,000 observations. We'll fit this model with 8 hidden levels with the **nnet** function.

Some info about the **nnet** function:

- it is used for single-hidden layer networks only.
- The call is:

  nnet(x, y, weights, size, Wts, mask,linout = FALSE, entropy = FALSE, softmax = FALSE, censored = FALSE, skip = FALSE, rang = 0.7, decay = 0, maxit = 100, Hess = FALSE, trace = TRUE, MaxNWts = 1000, abstol = 1.0e-4, reltol = 1.0e-8, ...)

- Some of the options (from the help file):

  – formula-A formula of the form class ~ x1 + x2 + ...

  – x-matrix or data frame of x values for examples.

  – y-matrix or data frame of target values for examples.

  – weights-(case) weights for each example – if missing defaults to 1.

  – size-number of units in the hidden layer. Can be zero if there are skip-layer units.

  – Wts-initial parameter vector. If missing chosen at random.

  – linout-switch for linear output units. Default logistic output units.

  – entropy-switch for entropy (= maximum conditional likelihood) fitting. Default by least-squares.

  – softmax-switch for softmax (log-linear model) and maximum conditional likelihood fitting. linout, entropy, softmax and censored are mutually exclusive. censored A variant on softmax, in which non-zero targets mean possible classes. Thus for softmax a row of (0, 1, 1) means one example each of classes 2 and 3, but for censored it means one example whose class is only known to be 2 or 3.

  – rang-Initial random weights on [-rang, rang]. Value about 0.5 unless the inputs are large, in which case it should be chosen so that rang * max(|x|) is about 1.

  – decay-parameter for weight decay. Default 0.

  – maxit-maximum number of iterations. Default 100.

  – abstol-Stop if the fit criterion falls below abstol, indicating an essentially perfect fit.

  – reltol-Stop if the optimizer is unable to reduce the fit criterion by a factor of at least 1 - reltol.

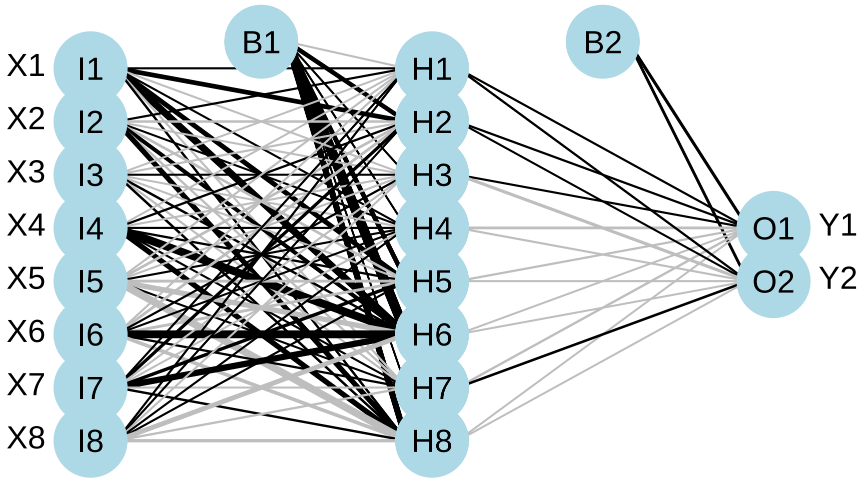
```
mod1<-nnet(rand.vars,resp,size=8,linout=T)
```

```
## # weights:  90
## initial  value 30066.949169
## iter  10 value 131.361932
## iter  20 value 51.449404
## iter  30 value 44.110444
## iter  40 value 39.959512
## iter  50 value 37.678074
## iter  60 value 35.715896
## iter  70 value 34.182676
## iter  80 value 33.162657
## iter  90 value 32.665243
## iter 100 value 32.413667
## final  value 32.413667
## stopped after 100 iterations
```

To plot the network we're going to import a function from Github

```
library(devtools)
library(reshape)
source_url('https://gist.github.com/fawda123/7471137/raw/cd6e6a0b0bdb4e065c597e52165e5ac887f5fe95/nnet_p

plot.nnet(mod1)
```



Full url is 'https://gist.github.com/fawda123/7471137/raw/cd6e6a0b0bdb4e065c597e52165e5ac887f5fe95/nnet_plot_update.r'

The shows all of the connections that fit. The black (gray) lines indicate a positive (negative) association. The thickness of the line indicates the strength of the association.

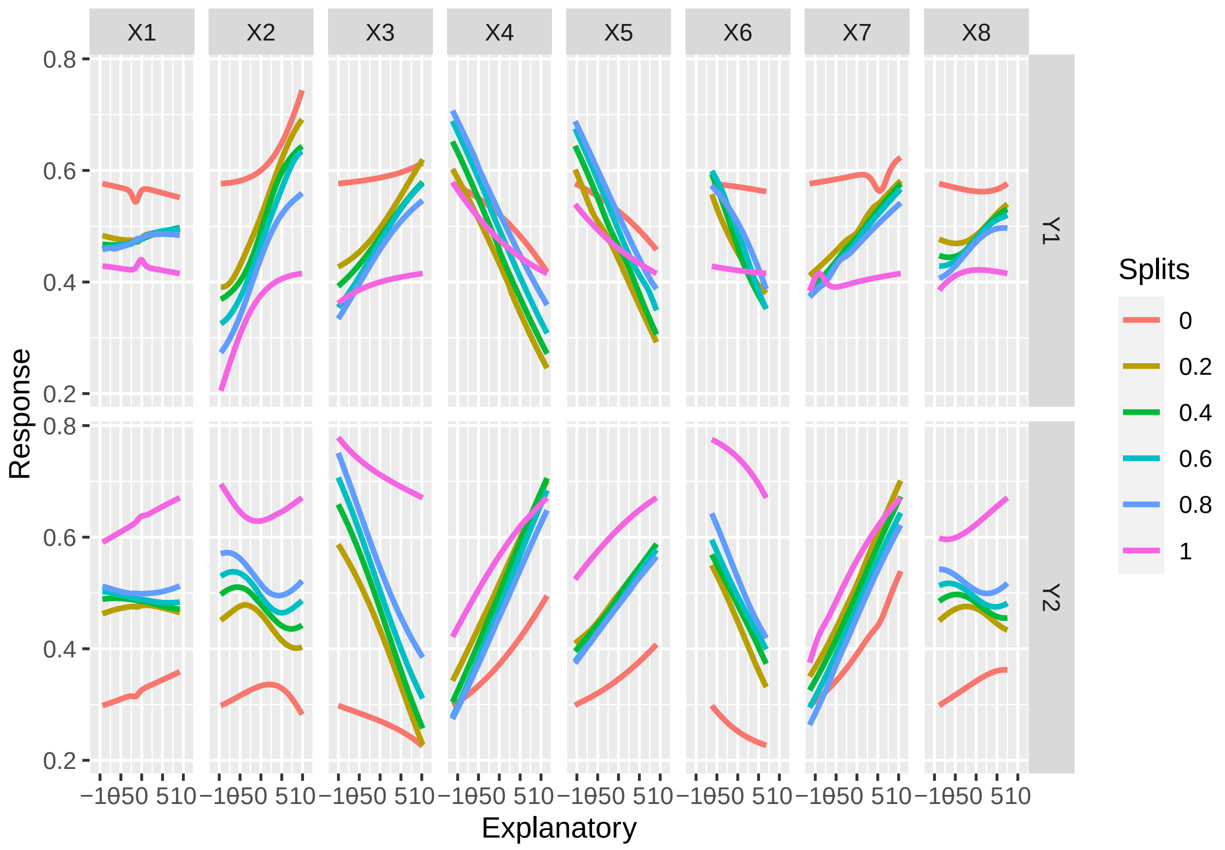
We can also visualize the effect that each variable has on the outcome using a method discribed in Gevrey et al. (2003) [1] (see below for reference). The hard thing about visualizing the effect one variable, say X_1, has on the outcome, say Y, is that all of the other variables, say $X_2, X_3, \ldots, X_r$ interact with this relationship. As a result, we visualize the effect between X_1 and Y for fixed values of $X_2, X_3, \ldots, X_r$. One way to do this is to:

- Set $X_2, X_3, \ldots, X_r$ to a fixed quantile, for example Q_1, then feed all values of X_1 into the network and calculate the expected value of Y for each value.
- Plot the expectation of Y by X_1.
- Set $X_2, X_3, \ldots, X_r$ to a different fixed quantile, repeat as necessary.

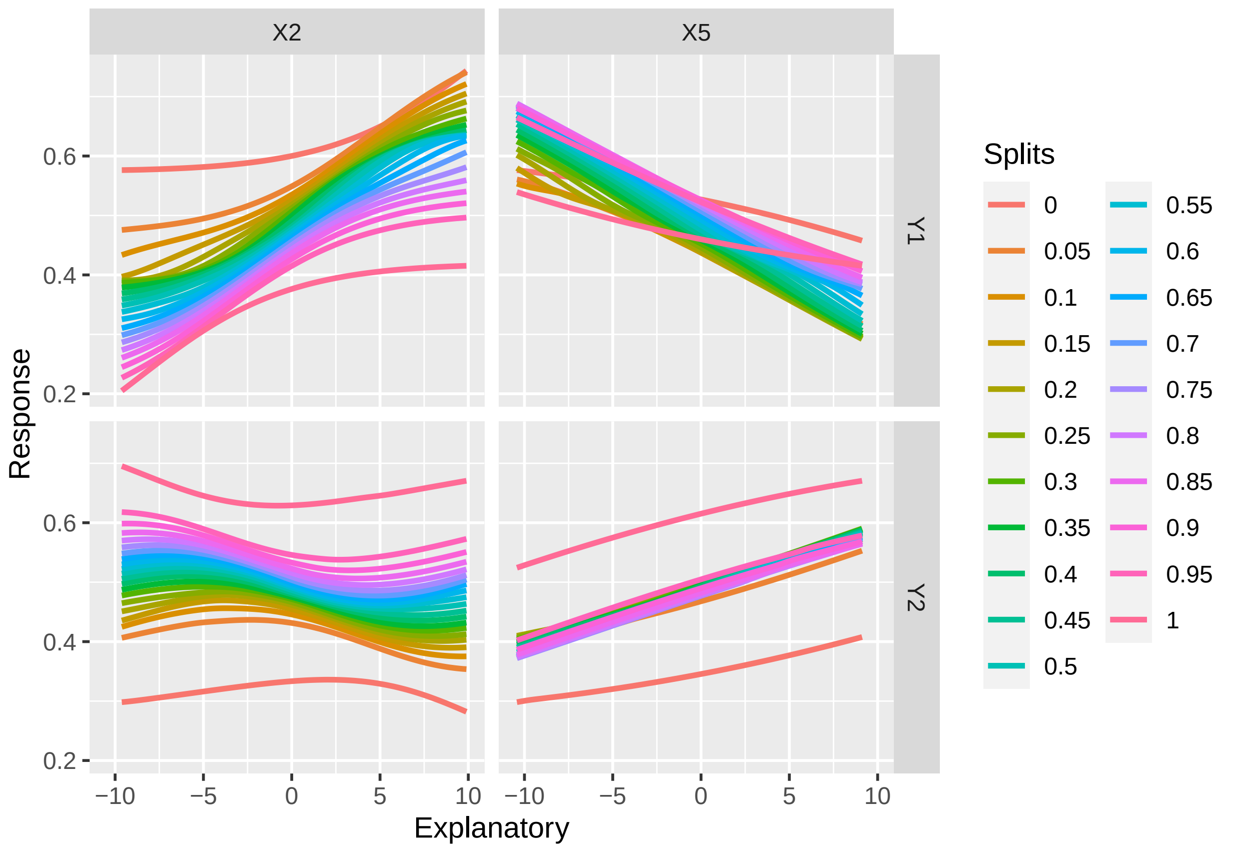
This method will give you some idea of what is impacting the outcome, but it will not give any information on the interactions between the variables.

```
library(ggplot2)
source('https://gist.githubusercontent.com/fawda123/6860630/raw/b8bf4a6c88d6b392b1bfa6ef24759ae98f31877c
lek.fun(mod1)
```

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```



```r
lek.fun(mod1,var.sens=c('X2','X5'),split.vals=seq(0,1,by=0.05))
```



Full url is 'https://gist.githubusercontent.com/fawda123/6860630/raw/b8bf4a6c88d6b392b1bfa6ef24759ae9 8f31877c/lek_fun.r'

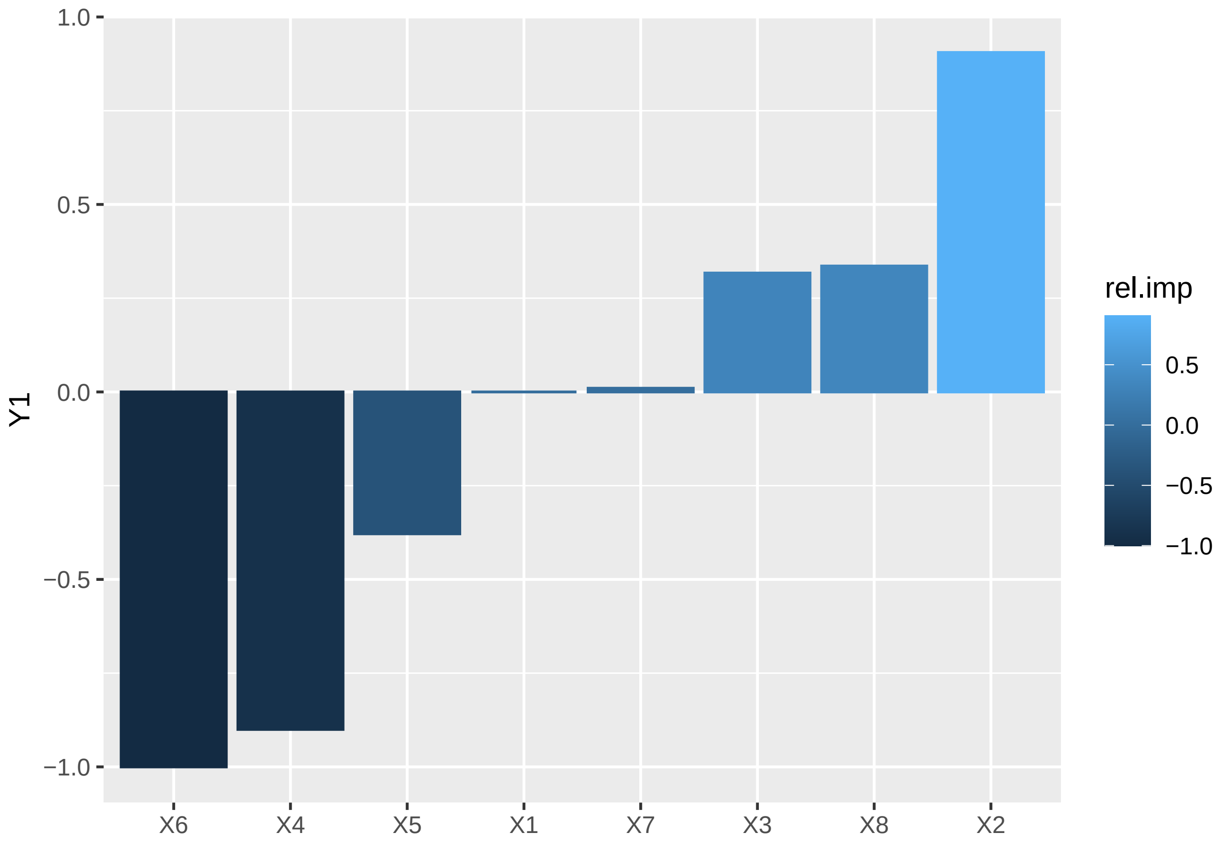
Next we're going to look at some measures of variable importance. The goal here is to measure the relative importance of explanatory variables for their relation to one or more response variables. The method we'll use here is described in Garson (1991) [2]. Here is the basic idea:

- Identify all weighted connections between a specific input node and a specific response variable.
- Do this for all other explanatory variables until we have a list of all weights for each input variable.
- The connections are "tallied"" for each input node and scaled relative to all other inputs.
- A single value is obtained for each explanatory variable that describes the relationship with response variable in the model (see the appendix in Goh 1995 [3] for a more detailed description).

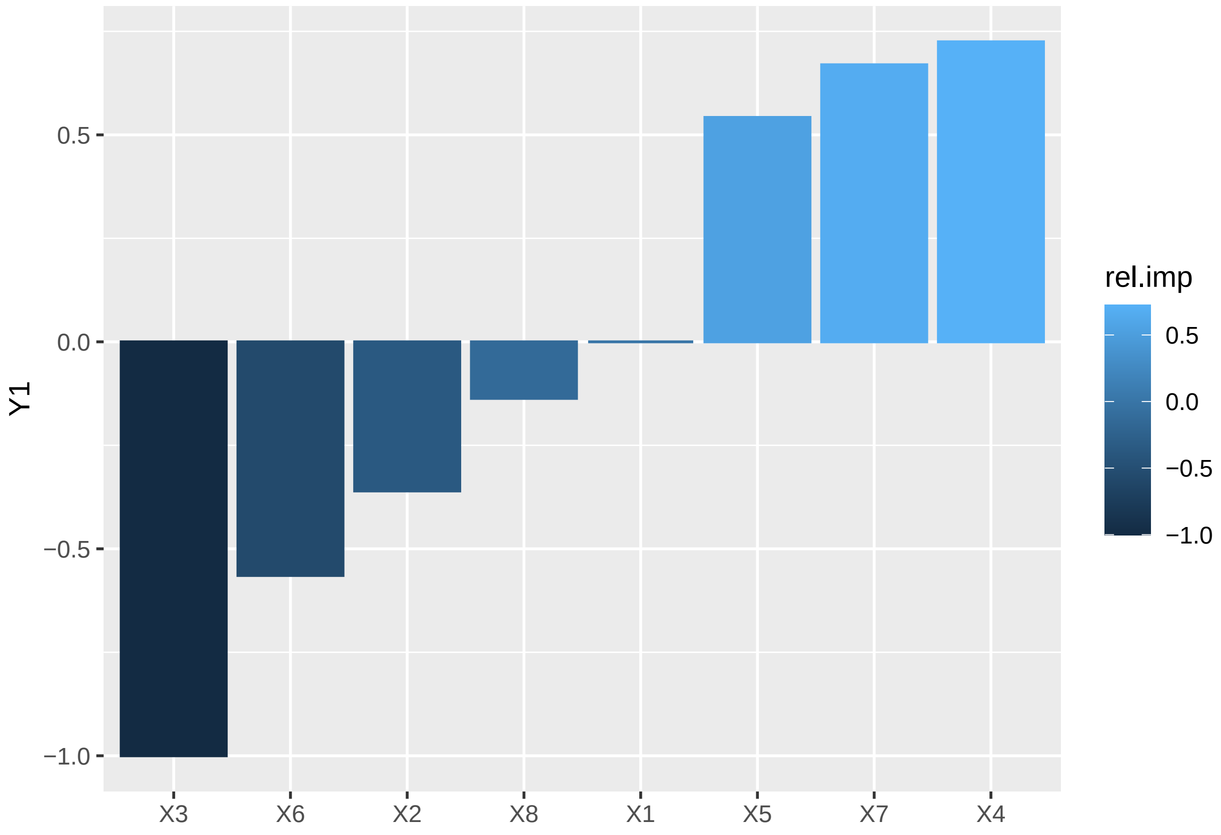
To do this we will import 'gar.fun' from Github (this is not a package just a function).

```
source_url('https://gist.githubusercontent.com/fawda123/6206737/raw/d6f365c283a8cae23fb20892dc223bc5764
#create a pretty color vector for the bar plot
cols<-colorRampPalette(c('lightgreen','lightblue'))(num.vars)

#use the function on the model created above
par(mar=c(3,4,1,1),family='serif')
gar.fun('Y1',mod1)
```



```
gar.fun('Y2',mod1)
```



[1] Gevrey M, Dimopoulos I, Lek S. 2003. Review and comparison of methods to study the contribution of variables in artificial neural network models. Ecological Modelling. 160:249-264.

[2] Garson, G.D. 1991. Interpreting neural network connection weights. Artificial Intelligence Expert. 6(4):46–51.

[3] Goh, A.T.C. 1995. Back-propagation neural networks for modeling complex systems. Artificial Intelligence in Engineering. 9(3):143–151.