Neural Networks

Alfonso R. Reyes

2019-09-18

Contents

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Chapter 1

Building deep neural nets with h2o that predict arrhythmia of the heart

1.1 Introduction

27 February 2017

This week, I am showing how to build feed-forward deep neural networks or multilayer perceptrons. The models in this example are built to classify ECG data into being either from healthy hearts or from someone suffering from arrhythmia. I will show how to prepare a dataset for modeling, setting weights and other modeling parameters, and finally, how to evaluate model performance with the `h2o` package.

1.1.1 Deep learning with neural networks

Deep learning with neural networks is arguably one of the most rapidly growing applications of machine learning and AI today. They allow building complex models that consist of multiple hidden layers within artificial networks and are able to find non-linear patterns in unstructured data. Deep neural networks are usually feed-forward, which means that each layer feeds its output to subsequent layers, but recurrent or feedback neural networks can also be built. Feed-forward neural networks are also called multilayer perceptrons (MLPs).

1.1.2 H2O

The R package h2o provides a convenient interface to **H2O**, which is an open-source machine learning and deep learning platform. H2O distributes a wide range of common machine learning algorithms for classification, regression and deep learning.

1.1.3 Preparing the R session

First, we need to load the packages.

```
library(dplyr)
library(h2o)
library(ggplot2)
library(ggrepel)
library(h2o)
```

```
h2o.init()
#>
#> H2O is not running yet, starting it now...
#>
#> Note:  In case of errors look at the following log files:
#>     /tmp/RtmpxGOYLu/h2o_datascience_started_from_r.out
#>     /tmp/RtmpxGOYLu/h2o_datascience_started_from_r.err
#>
#>
#> Starting H2O JVM and connecting: . Connection successful!
#>
#> R is connected to the H2O cluster:
#>     H2O cluster uptime:         1 seconds 141 milliseconds
#>     H2O cluster timezone:       America/Chicago
#>     H2O data parsing timezone:  UTC
#>     H2O cluster version:        3.22.1.1
#>     H2O cluster version age:    8 months and 21 days !!!
#>     H2O cluster name:           H2O_started_from_R_datascience_mwl453
#>     H2O cluster total nodes:    1
#>     H2O cluster total memory:   6.96 GB
#>     H2O cluster total cores:    8
#>     H2O cluster allowed cores:  8
#>     H2O cluster healthy:        TRUE
#>     H2O Connection ip:          localhost
#>     H2O Connection port:        54321
#>     H2O Connection proxy:       NA
#>     H2O Internal Security:      FALSE
#>     H2O API Extensions:         XGBoost, Algos, AutoML, Core V3, Core V4
#>     R Version:                  R version 3.6.0 (2019-04-26)
#> Warning in h2o.clusterInfo():
#> Your H2O cluster version is too old (8 months and 21 days)!
#> Please download and install the latest version from http://h2o.ai/download/
```

```
my_theme <- function(base_size = 12, base_family = "sans"){
  theme_minimal(base_size = base_size, base_family = base_family) +
  theme(
    axis.text = element_text(size = 12),
    axis.title = element_text(size = 14),
    panel.grid.major = element_line(color = "grey"),
    panel.grid.minor = element_blank(),
    panel.background = element_rect(fill = "aliceblue"),
    strip.background = element_rect(fill = "darkgrey", color = "grey", size = 1),
    strip.text = element_text(face = "bold", size = 12, color = "white"),
    legend.position = "right",
    legend.justification = "top",
    panel.border = element_rect(color = "grey", fill = NA, size = 0.5)
  )
}
```

1.2 Arrhythmia data

The data I am using to demonstrate the building of neural nets is the arrhythmia dataset from UC Irvine's machine learning database. It contains 279 features from ECG heart rhythm diagnostics and one output column. I am not going to rename the feature columns because they are too many and the descriptions are too complex. Also, we don't need to know specifically which features we are looking at for building the models.

> For a description of each feature, see https://archive.ics.uci.edu/ml/machine-learning-databases/arrhythmia/arrhythmia.names.

The output column defines 16 classes: *class 1* samples are from healthy ECGs, the remaining classes belong to different types of arrhythmia, with *class 16* being all remaining arrhythmia cases that didn't fit into distinct classes.

```
arrhythmia <- read.table(file.path(data_raw_dir, "arrhythmia.data.txt"), sep = ",")
arrhythmia[arrhythmia == "?"] <- NA

# making sure, that all feature columns are numeric
arrhythmia[-280] <- lapply(arrhythmia[-280], as.character)
arrhythmia[-280] <- lapply(arrhythmia[-280], as.numeric)

#  renaming output column and converting to factor
colnames(arrhythmia)[280] <- "class"
arrhythmia$class <- as.factor(arrhythmia$class)
```

As usual, I want to get acquainted with the data and explore it's properties before I am building any model. So, I am first going to look at the distribution of classes and of healthy and arrhythmia samples.

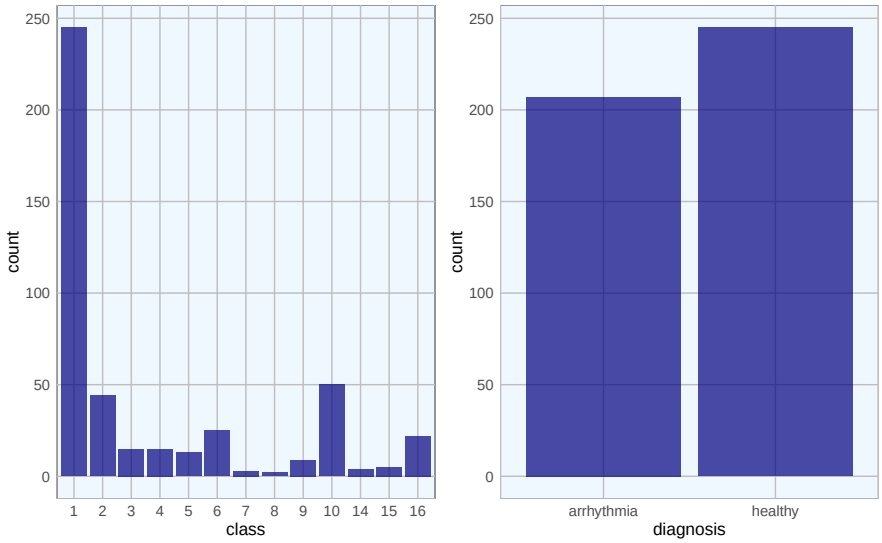
```
p1 <- ggplot(arrhythmia, aes(x = class)) +
  geom_bar(fill = "navy", alpha = 0.7) +
  my_theme()
```

Because I am interested in distinguishing healthy from arrhythmia ECGs, I am converting the output to binary format by combining all arrhythmia cases into one class.

```
#  all arrhythmia cases into one class
arrhythmia$diagnosis <- ifelse(arrhythmia$class == 1, "healthy", "arrhythmia")
arrhythmia$diagnosis <- as.factor(arrhythmia$diagnosis)
```

```
p2 <- ggplot(arrhythmia, aes(x = diagnosis)) +
  geom_bar(fill = "navy", alpha = 0.7) +
  my_theme()
```

```
library(gridExtra)
#>
#> Attaching package: 'gridExtra'
#> The following object is masked from 'package:dplyr':
#>
#>     combine
library(grid)

grid.arrange(p1, p2, ncol = 2)
```



With binary classification, we have almost the same numbers of healthy and arrhythmia cases in our dataset.

I am also interested in how much the normal and arrhythmia cases cluster in a Principal Component Analysis (PCA). I am first preparing the PCA plotting function and then run it on the feature data.

```
library(pcaGoPromoter)
#> Warning: replacing previous import 'BiocGenerics::boxplot' by
#> 'graphics::boxplot' when loading 'Biostrings'
#> Warning: replacing previous import 'BiocGenerics::image' by
#> 'graphics::image' when loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::na.exclude' by
#> 'stats::na.exclude' when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::smoothEnds' by
#> 'stats::smoothEnds' when loading 'Biostrings'
#> Warning: replacing previous import 'BiocGenerics::density' by
#> 'stats::density' when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::mad' by 'stats::mad' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::na.omit' by 'stats::na.omit'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::complete.cases' by
#> 'stats::complete.cases' when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::runmed' by 'stats::runmed'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::start' by 'stats::start' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::window<-' by 'stats::window<-'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::window' by 'stats::window'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::aggregate' by
#> 'stats::aggregate' when loading 'Biostrings'
#> Warning: replacing previous import 'BiocGenerics::weights' by
#> 'stats::weights' when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::cor' by 'stats::cor' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::cov' by 'stats::cov' when
```

```
#> loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::quantile' by 'stats::quantile'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::end' by 'stats::end' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'BiocGenerics::residuals' by
#> 'stats::residuals' when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::median' by 'stats::median'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::sd' by 'stats::sd' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::var' by 'stats::var' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::xtabs' by 'stats::xtabs'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::IQR' by 'stats::IQR' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::tail' by 'utils::tail' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::stack' by 'utils::stack' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'XVector::relist' by 'utils::relist'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::head' by 'utils::head' when
#> loading 'Biostrings'

pca_func <- function(pcaOutput2, group_name){
    centroids <- aggregate(cbind(PC1, PC2) ~ groups, pcaOutput2, mean)
    conf.rgn  <- do.call(rbind, lapply(unique(pcaOutput2$groups), function(t)
          data.frame(groups = as.character(t),
                     ellipse(cov(pcaOutput2[pcaOutput2$groups == t, 1:2]),
                           centre = as.matrix(centroids[centroids$groups == t, 2:3]),
                           level = 0.95),
                     stringsAsFactors = FALSE)))

    plot <- ggplot(data = pcaOutput2, aes(x = PC1, y = PC2, group = groups,
                                          color = groups)) +
      geom_polygon(data = conf.rgn, aes(fill = groups), alpha = 0.2) +
      geom_point(size = 2, alpha = 0.5) +
      labs(color = paste(group_name),
           fill = paste(group_name),
           x = paste0("PC1: ", round(pcaOutput$pov[1], digits = 2) * 100, "% variance"),
           y = paste0("PC2: ", round(pcaOutput$pov[2], digits = 2) * 100, "% variance")) +
      my_theme()

    return(plot)
}
```

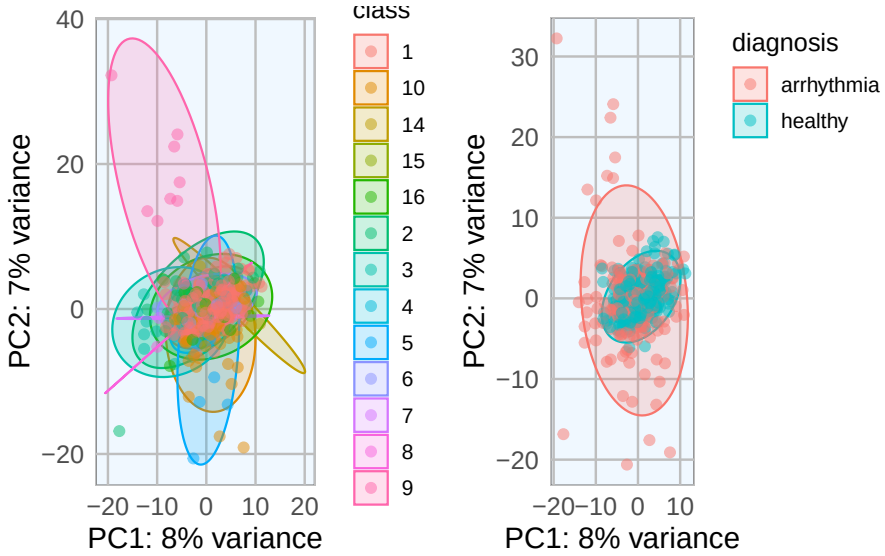
```
# Find what columns have NAs and the quantity
for (col in names(arrhythmia)) {
    n_nas <- length(which(is.na(arrhythmia[, col])))
    if (n_nas > 0) cat(col, n_nas, "\n")
}
#> V11 8
```

```
#> V12 22
#> V13 1
#> V14 376
#> V15 1
```

```
# Replace NAs with zeros
arrhythmia[is.na(arrhythmia)] <- 0
```

Find and plot the PCAs.

```
pcaOutput <- pca(t(arrhythmia[-c(280, 281)]), printDropped=FALSE,
                 scale=TRUE,
                 center = TRUE)

pcaOutput2 <- as.data.frame(pcaOutput$scores)

pcaOutput2$groups <- arrhythmia$class
p1 <- pca_func(pcaOutput2, group_name = "class")

pcaOutput2$groups <- arrhythmia$diagnosis
p2 <- pca_func(pcaOutput2, group_name = "diagnosis")

grid.arrange(p1, p2, ncol = 2)
```



The PCA shows that there is a big overlap between healthy and arrhythmia samples, i.e. there does not seem to be major global differences in all features. The class that is most distinct from all others seems to be class 9.

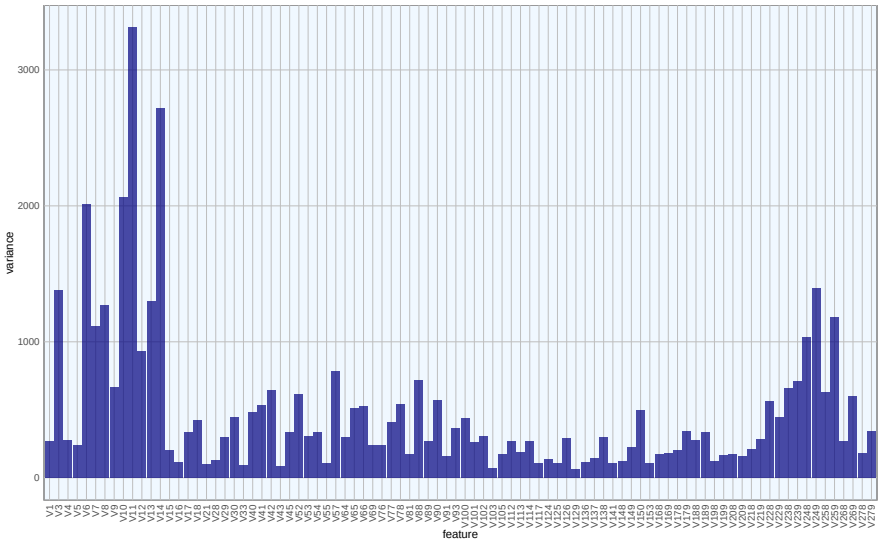
I want to give the arrhythmia cases that are very different from the rest a stronger weight in the neural network, so I define a **weight column** where every sample outside the central PCA cluster will get a "2", they will in effect be used twice in the model.

```
weights <- ifelse(pcaOutput2$PC1 < -5 & abs(pcaOutput2$PC2) > 10, 2, 1)
```

I also want to know what the variance is within features.

```
library(matrixStats)
#>
#> Attaching package: 'matrixStats'
```

```
#> The following object is masked from 'package:dplyr':
#>
#>     count

colvars <- data.frame(feature = colnames(arrhythmia[-c(280, 281)]),
                      variance = colVars(as.matrix(arrhythmia[-c(280, 281)])))

subset(colvars, variance > 50) %>%
  mutate(feature = factor(feature, levels = colnames(arrhythmia[-c(280, 281)]))) %>%
  ggplot(aes(x = feature, y = variance)) +
    geom_bar(stat = "identity", fill = "navy", alpha = 0.7) +
    my_theme() +
    theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1))
```



Features with low variance are less likely to strongly contribute to a differentiation between healthy and arrhythmia cases, so I am going to remove them. I am also concatenating the weights column:

```
arrhythmia_subset <- cbind(weights,
                           arrhythmia[, c(281, 280, which(colvars$variance > 50))])
```

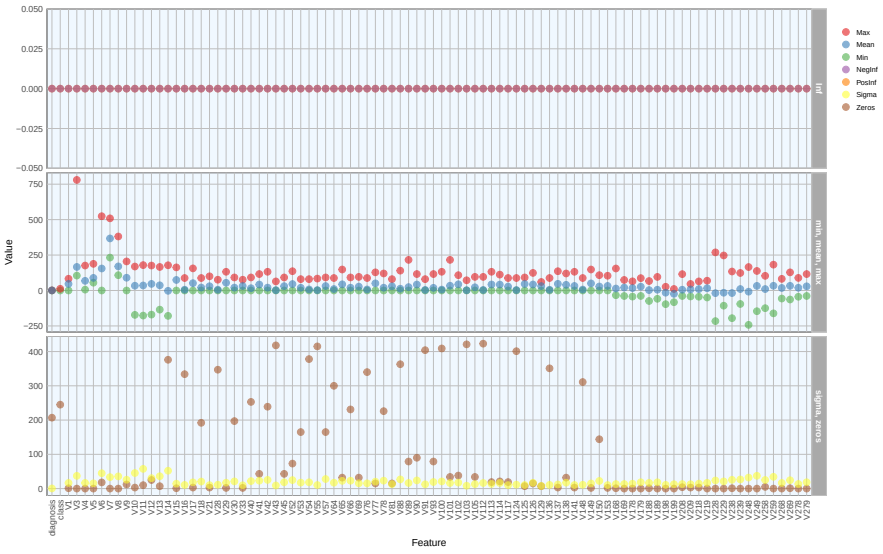
1.3 Converting the dataframe to a h2o object

Now that I have my final data frame for modeling, for working with h2o functions, the data needs to be converted from a **DataFrame** to an **H2O Frame**. This is done with the `as_h2o_frame()` function.

```
#as_h2o_frame(arrhythmia_subset)
arrhythmia_hf <- as.h2o(arrhythmia_subset, key="arrhtythmia.hex")
#>
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

We can now access all functions from the h2o package that are built to work on `h2o` Frames. A useful such function is `h2o.describe()`. It is similar to base R's `summary()` function but outputs many more descriptive measures for our data. To get a good overview about these measures, I am going to plot them.

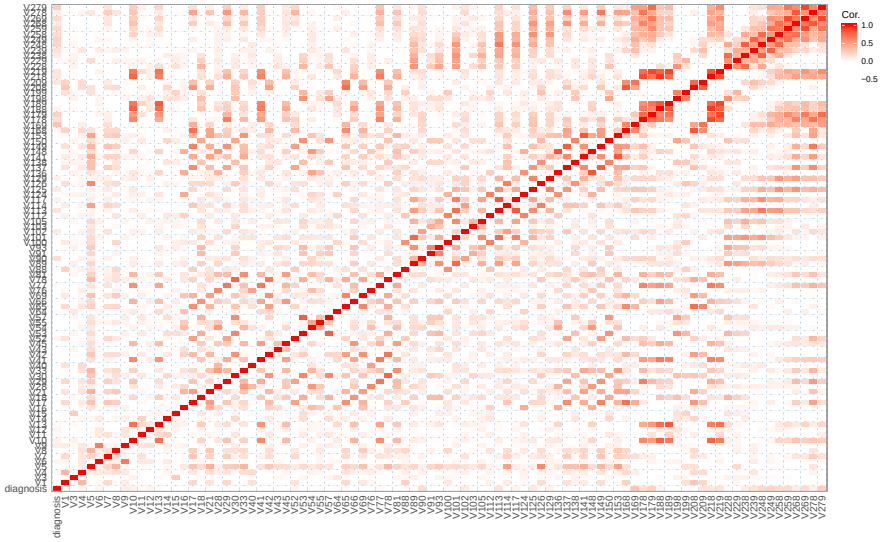
```
library(tidyr) # for gathering
#>
#> Attaching package: 'tidyr'
#> The following object is masked from 'package:S4Vectors':
#>
#>     expand
h2o.describe(arrhythmia_hf[, -1]) %>% # excluding the weights column
  gather(x, y, Zeros:Sigma) %>%
  mutate(group = ifelse(
    x %in% c("Min", "Max", "Mean"), "min, mean, max",
    ifelse(x %in% c("NegInf", "PosInf"), "Inf", "sigma, zeros"))) %>%
  # separating them into facets makes them easier to see
  mutate(Label = factor(Label, levels = colnames(arrhythmia_hf[, -1]))) %>%
  ggplot(aes(x = Label, y = as.numeric(y), color = x)) +
    geom_point(size = 4, alpha = 0.6) +
    scale_color_brewer(palette = "Set1") +
    my_theme() +
    theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1)) +
    facet_grid(group ~ ., scales = "free") +
    labs(x = "Feature",
         y = "Value",
         color = "")
#> Warning: Removed 2 rows containing missing values (geom_point).
```



I am also interested in the correlation between features and the output. We can use the `h2o.cor()` function to calculate the correlation matrix. It is again much easier to understand the data when we visualize it, so I am going to create another plot.

```
library(reshape2) # for melting
#>
#> Attaching package: 'reshape2'
#> The following object is masked from 'package:tidyr':
#>
#>     smiths
```

```
# diagnosis is now a characer column and we need to convert it again
arrhythmia_hf[, 2] <- h2o.asfactor(arrhythmia_hf[, 2])
arrhythmia_hf[, 3] <- h2o.asfactor(arrhythmia_hf[, 3]) # same for class

cor <- h2o.cor(arrhythmia_hf[, -c(1, 3)])
rownames(cor) <- colnames(cor)

melt(cor) %>%
  mutate(Var2 = rep(rownames(cor), nrow(cor))) %>%
  mutate(Var2 = factor(Var2, levels = colnames(cor))) %>%
  mutate(variable = factor(variable, levels = colnames(cor))) %>%
  ggplot(aes(x = variable, y = Var2, fill = value)) +
    geom_tile(width = 0.9, height = 0.9) +
    scale_fill_gradient2(low = "white", high = "red", name = "Cor.") +
    my_theme() +
    theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1)) +
    labs(x = "",
         y = "")
#> No id variables; using all as measure variables
```



1.4 Training, test and validation data

Now we can use the `h2o.splitFrame()` function to split the data into training, validation and test data.

Here, I am using 70% for training and 15% each for validation and testing. We could also just split the data into two sections, a training and test set but when we have sufficient samples, it is a good idea to evaluate model performance on an independent test set on top of training with a validation set. Because we can easily overfit a model, we want to get an idea about how generalizable it is - this we can only assess by looking at how well it works on previously unknown data.

I am also defining `response`, `features` and `weights` column names now.

```
splits <- h2o.splitFrame(arrhythmia_hf,
                         ratios = c(0.7, 0.15),
                         seed = 1)
```

```
train <- splits[[1]]
valid <- splits[[2]]
test <- splits[[3]]

response <- "diagnosis"
weights <- "weights"
features <- setdiff(colnames(train), c(response, weights, "class"))
```

```
summary(train$diagnosis, exact_quantiles = TRUE)
#>  diagnosis
#>  healthy   :163
#>  arrhythmia:155
```

```
summary(valid$diagnosis, exact_quantiles = TRUE)
#>  diagnosis
#>  healthy   :43
#>  arrhythmia:25
```

```
summary(test$diagnosis, exact_quantiles = TRUE)
#>  diagnosis
#>  healthy   :39
#>  arrhythmia:27
```

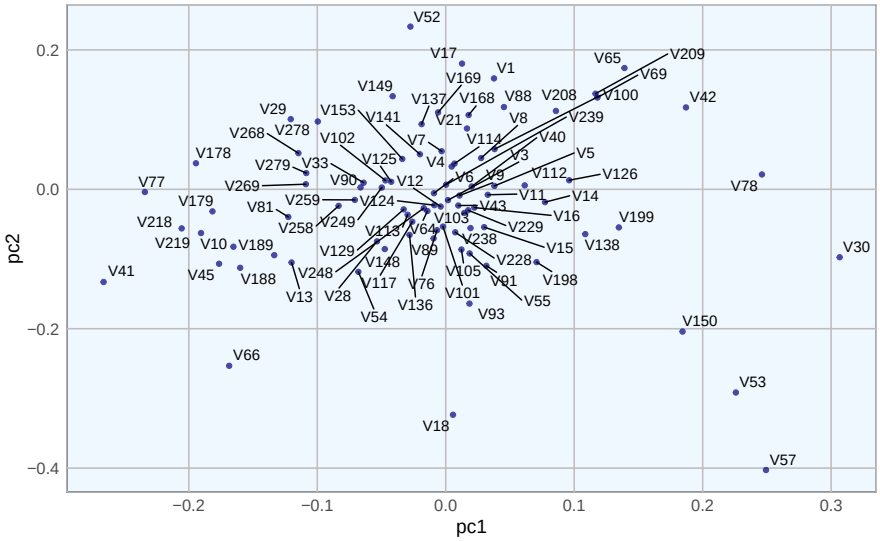
If we had more categorical features, we could use the `h2o.interaction()` function to define interaction terms, but since we only have numeric features here, we don't need this.

We can also run a PCA on the training data, using the `h2o.prcomp()` function to calculate the singular value decomposition of the Gram matrix with the power method.

```
pca <- h2o.prcomp(training_frame = train,
           x = features,
           validation_frame = valid,
           transform = "NORMALIZE",
           k = 3,
           seed = 42)
#>
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
#> Warning in doTryCatch(return(expr), name, parentenv, handler): _train:
#> Dataset used may contain fewer number of rows due to removal of rows with
#> NA/missing values. If this is not desirable, set impute_missing argument in
#> pca call to TRUE/True/true/... depending on the client language.
pca
#> Model Details:
#> ==============
#>
#> H2ODimReductionModel: pca
#> Model ID:  PCA_model_R_1568840446217_1
#> Importance of components:
#>                             pc1      pc2      pc3
#> Standard deviation     0.582620 0.507796 0.421869
#> Proportion of Variance 0.164697 0.125110 0.086351
```

```
#> Cumulative Proportion  0.164697 0.289808 0.376159
#>
#>
#> H2ODimReductionMetrics: pca
#>
#> No model metrics available for PCA
#> H2ODimReductionMetrics: pca
#>
#> No model metrics available for PCA
```

```
eigenvec <- as.data.frame(pca@model$eigenvectors)
eigenvec$label <- features

ggplot(eigenvec, aes(x = pc1, y = pc2, label = label)) +
  geom_point(color = "navy", alpha = 0.7) +
  geom_text_repel() +
  my_theme()
```



1.5 Modeling

Now, we can build a deep neural network model. We can specify quite a few parameters, like

- **Cross-validation**: Cross validation can tell us the training and validation errors for each model. The final model will be overwritten with the best model, if we don't specify otherwise.
- **Adaptive learning rate**: For deep learning with h2o, we by default use stochastic gradient descent optimization with an an adaptive learning rate. The two corresponding parameters rho and epsilon help us find global (or near enough) optima.
- **Activation function**: The activation function defines the node output relative to a given set of inputs. We want our activation function to be non-linear and continuously differentiable.
- **Hidden nodes**: Defines the number of hidden layers and the number of nodes per layer.
- **Epochs**: Increasing the number of epochs (one full training cycle on all training samples) can increase model performance, but we also run the risk of overfitting. To determine the optimal number of epochs, we need to use early stopping.

- **Early stopping**: By default, early stopping is enabled. This means that training will be stopped when we reach a certain validation error to prevent overfitting.

Of course, you need quite a bit of experience and intuition to hit on a good combination of parameters. That's why it usually makes sense to do a grid search for hyper-parameter tuning. Here, I want to focus on building and evaluating deep learning models, though. I will cover grid search in next week's post.

```
# this will take some time and all CPUs
dl_model <- h2o.deeplearning(x = features,
                             y = response,
                             weights_column = weights,
                             model_id = "dl_model",
                             training_frame = train,
                             validation_frame = valid,
                             nfolds = 15,                                  # 10x cross validation
                             keep_cross_validation_fold_assignment = TRUE,
                             fold_assignment = "Stratified",
                             activation = "RectifierWithDropout",
                             score_each_iteration = TRUE,
                             hidden = c(200, 200, 200, 200, 200),          # 5 hidden layers, each of
                             epochs = 100,
                             variable_importances = TRUE,
                             export_weights_and_biases = TRUE,
                             seed = 42)
#>
  |
  |                                                                      |   0%
  |
  |==                                                                    |   3%
  |
  |======                                                                |  10%
  |
  |=======                                                               |  11%
  |
  |========                                                              |  12%
  |
  |=========                                                             |  13%
  |
  |==========                                                            |  15%
  |
  |=============                                                         |  19%
  |
  |==============                                                        |  21%
  |
  |================                                                      |  23%
  |
  |================                                                      |  25%
  |
  |==================                                                    |  27%
  |
  |==================                                                    |  28%
  |
  |====================                                                  |  29%
  |
```

```
|==================                                                    |  31%
|
|====================                                                  |  32%
|
|====================                                                  |  33%
|
|========================                                              |  37%
|
|========================                                              |  38%
|
|=========================                                             |  39%
|
|=========================                                             |  41%
|
|=========================                                             |  41%
|
|==========================                                            |  42%
|
|==========================                                            |  44%
|
|============================                                          |  46%
|
|============================                                          |  47%
|
|============================                                          |  48%
|
|============================                                          |  49%
|
|==================================                                    |  52%
|
|==================================                                    |  53%
|
|==================================                                    |  54%
|
|==================================                                    |  55%
|
|==================================                                    |  56%
|
|====================================                                  |  57%
|
|=====================================                                 |  61%
|
|=====================================                                 |  61%
|
|======================================                                |  63%
|
|======================================                                |  64%
|
|=======================================                               |  66%
|
|=========================================                             |  68%
|
|=========================================                             |  69%
```

```
  |
  |=============================================                    |  71%
  |
  |==============================================                   |  72%
  |
  |===============================================                  |  74%
  |
  |=================================================================  |  96%
  |
  |==================================================================  |  97%
  |
  |=================================================================== |  99%
  |
  |======================================================================| 100%
```

Because training can take a while, depending on how many samples, features, nodes and hidden layers you are training on, it is a good idea to save your model.

```
# if file exists, overwrite it
h2o.saveModel(dl_model, path = file.path(data_out_dir, "dl_model"), force = TRUE)
#> [1] "/home/datascience/repos/machine-learning-rsuite/export/dl_model/dl_model"
```

We can then re-load the model again any time to check the model quality and make predictions on new data.

```
dl_model <- h2o.loadModel(file.path(data_out_dir, "dl_model/dl_model"))
```

1.6 Model performance

We now want to know how our model performed on the validation data. The summary() function will give us a detailed overview of our model. I am not showing the output here, because it is quite extensive.

```
sum_model <- summary(dl_model)
#> Model Details:
#> ==============
#>
#> H2OBinomialModel: deeplearning
#> Model Key:  dl_model
#> Status of Neuron Layers: predicting diagnosis, 2-class classification, bernoulli distribution, Cross
#>   layer units             type dropout       l1       l2 mean_rate
#> 1     1    90            Input  0.00 %       NA       NA        NA
#> 2     2   200 RectifierDropout 50.00 % 0.000000 0.000000  0.004555
#> 3     3   200 RectifierDropout 50.00 % 0.000000 0.000000  0.006484
#> 4     4   200 RectifierDropout 50.00 % 0.000000 0.000000  0.009338
#> 5     5   200 RectifierDropout 50.00 % 0.000000 0.000000  0.008749
#> 6     6   200 RectifierDropout 50.00 % 0.000000 0.000000  0.020938
#> 7     7     2          Softmax      NA 0.000000 0.000000  0.002158
#>   rate_rms momentum mean_weight weight_rms mean_bias bias_rms
#> 1       NA       NA          NA         NA        NA       NA
#> 2 0.003825 0.000000    0.002715   0.096436  0.430206 0.056454
#> 3 0.003681 0.000000   -0.008004   0.074799  0.948645 0.052865
#> 4 0.004960 0.000000   -0.007273   0.072414  0.967009 0.028673
#> 5 0.004542 0.000000   -0.005750   0.071128  0.974126 0.033538
#> 6 0.048796 0.000000   -0.010381   0.070846  0.952112 0.033043
#> 7 0.001022 0.000000   -0.041102   0.378216  0.000671 0.124920
```

```
#>
#> H2OBinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on full training frame **
#>
#> MSE:  0.0212
#> RMSE:  0.146
#> LogLoss:  0.0884
#> Mean Per-Class Error:  0.0244
#> AUC:  0.985
#> pr_auc:  0.964
#> Gini:  0.969
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>            arrhythmia healthy    Error    Rate
#> arrhythmia        156       8 0.048780  =8/164
#> healthy             0     163 0.000000  =0/163
#> Totals            156     171 0.024465  =8/327
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>                         metric threshold    value idx
#> 1                       max f1  0.422204 0.976048 170
#> 2                       max f2  0.422204 0.990279 170
#> 3                 max f0point5  0.886807 0.972046 155
#> 4                 max accuracy  0.628002 0.975535 166
#> 5                max precision  0.990334 1.000000   0
#> 6                   max recall  0.422204 1.000000 170
#> 7              max specificity  0.990334 1.000000   0
#> 8             max absolute_mcc  0.422204 0.952217 170
#> 9   max min_per_class_accuracy  0.730555 0.969512 163
#> 10 max mean_per_class_accuracy  0.422204 0.975610 170
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
#> H2OBinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on full validation frame **
#>
#> MSE:  0.185
#> RMSE:  0.431
#> LogLoss:  1
#> Mean Per-Class Error:  0.195
#> AUC:  0.87
#> pr_auc:  0.889
#> Gini:  0.74
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>            arrhythmia healthy    Error    Rate
#> arrhythmia         17       8 0.320000   =8/25
#> healthy             3      40 0.069767   =3/43
#> Totals             20      48 0.161765  =11/68
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>                         metric threshold    value idx
```

```
#> 1                       max f1  0.005650 0.879121  47
#> 2                       max f2  0.000018 0.929204  53
#> 3                 max f0point5  0.503477 0.862069  39
#> 4                 max accuracy  0.005650 0.838235  47
#> 5                max precision  0.990942 1.000000   0
#> 6                   max recall  0.000004 1.000000  59
#> 7              max specificity  0.990942 1.000000   0
#> 8             max absolute_mcc  0.005650 0.645749  47
#> 9   max min_per_class_accuracy  0.503477 0.800000  39
#> 10 max mean_per_class_accuracy  0.503477 0.806977  39
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
#> H2OBinomialMetrics: deeplearning
#> ** Reported on cross-validation data. **
#> ** 15-fold cross-validation on training data (Metrics computed for combined holdout predictions) **
#>
#> MSE:  0.176
#> RMSE:  0.419
#> LogLoss:  0.624
#> Mean Per-Class Error:  0.208
#> AUC:  0.843
#> pr_auc:  0.787
#> Gini:  0.685
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>            arrhythmia healthy    Error     Rate
#> arrhythmia        130      34 0.207317  =34/164
#> healthy            34     129 0.208589  =34/163
#> Totals            164     163 0.207951  =68/327
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>                         metric threshold    value idx
#> 1                       max f1  0.635968 0.791411 162
#> 2                       max f2  0.001835 0.881393 266
#> 3                 max f0point5  0.758633 0.797665 151
#> 4                 max accuracy  0.635968 0.792049 162
#> 5                max precision  0.988261 1.000000   0
#> 6                   max recall  0.001134 1.000000 273
#> 7              max specificity  0.988261 1.000000   0
#> 8             max absolute_mcc  0.635968 0.584094 162
#> 9   max min_per_class_accuracy  0.635968 0.791411 162
#> 10 max mean_per_class_accuracy  0.635968 0.792047 162
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
#> Cross-Validation Metrics Summary:
#>                               mean          sd  cv_1_valid  cv_2_valid
#> accuracy                 0.8611481 0.053447507  0.71428573   0.8235294
#> auc                      0.8550837  0.07367064   0.6631016  0.71666664
#> err                      0.1388519 0.053447507   0.2857143   0.1764706
#> err_count                3.3333333   1.6261748         8.0         3.0
#> f0point5                0.83738595    0.069643   0.6321839   0.6896552
#> f1                       0.8651681  0.05386143  0.73333335  0.72727275
#> f2                       0.8994148 0.046931576   0.8730159   0.7692308
```

```
#> lift_top_group           1.4293057   0.6512436         0.0         0.0
#> logloss                  0.6092432  0.18783744   0.9387797   1.1304744
#> max_per_class_error     0.23747644  0.10214973  0.47058824         0.2
#> mcc                      0.7298193  0.09693578   0.5536258  0.60385966
#> mean_per_class_accuracy  0.8558342 0.054132007   0.7647059  0.81666666
#> mean_per_class_error    0.14416581 0.054132007  0.23529412  0.18333334
#> mse                      0.1686008 0.049769267  0.29961145  0.25869125
#> precision                0.8216217   0.0810019  0.57894737   0.6666667
#> r2                      0.28901088  0.21953593 -0.25612497 -0.24602953
#> recall                   0.9271715  0.05560393         1.0         0.8
#> rmse                    0.40071017  0.06337255  0.54736775    0.508617
#> specificity             0.78449684  0.11391768   0.5294118   0.8333333
#>                        cv_3_valid cv_4_valid  cv_5_valid cv_6_valid
#> accuracy                      1.0 0.82608694   0.9444444     0.9375
#> auc                           1.0 0.84126985  0.96103895   0.984375
#> err                           0.0 0.17391305 0.055555556     0.0625
#> err_count                     0.0        4.0         1.0        1.0
#> f0point5                      1.0 0.81395346  0.98039216 0.90909094
#> f1                            1.0      0.875  0.95238096  0.9411765
#> f2                            1.0  0.9459459   0.9259259  0.9756098
#> lift_top_group          1.6666666  1.6428572   1.6363636        2.0
#> logloss                 0.1915051 0.56637216   0.4381006 0.19899572
#> max_per_class_error           0.0 0.44444445  0.09090909      0.125
#> mcc                           1.0  0.6573422   0.8918826  0.8819171
#> mean_per_class_accuracy       1.0  0.7777778  0.95454544     0.9375
#> mean_per_class_error          0.0 0.22222222 0.045454547     0.0625
#> mse                     0.05087649 0.16059141   0.1431109 0.06466938
#> precision                     1.0  0.7777778         1.0  0.8888889
#> r2                      0.78801465   0.325771   0.3978191 0.74132246
#> recall                        1.0        1.0  0.90909094        1.0
#> rmse                    0.22555818 0.40073857      0.3783 0.25430176
#> specificity                   1.0  0.5555556         1.0      0.875
#>                        cv_7_valid cv_8_valid cv_9_valid cv_10_valid
#> accuracy                0.90909094 0.90909094  0.8076923   0.8666667
#> auc                     0.91071427  0.9285714  0.7647059  0.87946427
#> err                     0.09090909 0.09090909  0.1923077  0.13333334
#> err_count                      2.0        1.0        5.0         4.0
#> f0point5                     0.875  0.8974359 0.82474226  0.88709676
#> f1                           0.875 0.93333334  0.8648649  0.84615386
#> f2                           0.875  0.9722222 0.90909094   0.8088235
#> lift_top_group                2.75  1.5714285  1.5294118    2.142857
#> logloss                  0.5307357  0.8580031  0.6874807   0.6276062
#> max_per_class_error          0.125       0.25 0.44444445  0.21428572
#> mcc                      0.8035714 0.81009257  0.5608894  0.73648536
#> mean_per_class_accuracy  0.90178573      0.875   0.748366  0.86160713
#> mean_per_class_error     0.09821428      0.125   0.251634  0.13839285
#> mse                     0.112374716 0.19855355 0.20534797   0.1689381
#> precision                    0.875      0.875        0.8   0.9166667
#> r2                       0.5143807 0.14196502 0.09271092  0.32123086
#> recall                       0.875        1.0  0.9411765  0.78571427
#> rmse                     0.33522338 0.44559348 0.45315337   0.4110208
#> specificity               0.9285714       0.75  0.5555556      0.9375
#>                        cv_11_valid  cv_12_valid cv_13_valid cv_14_valid
```

```
#> accuracy                  0.8636364    0.7619048     0.78125  0.83870965
#> auc                        0.892562    0.6818182    0.796875  0.87916666
#> err                       0.13636364   0.23809524     0.21875  0.16129032
#> err_count                        3.0          5.0         7.0         5.0
#> f0point5                  0.84745765   0.71428573  0.77380955   0.8333333
#> f1                         0.8695652          0.8   0.7878788   0.8484849
#> f2                        0.89285713   0.90909094  0.80246913  0.86419755
#> lift_top_group                   2.0          0.0         2.0         0.0
#> logloss                   0.39886966    0.9508356   0.6924645   0.5638011
#> max_per_class_error       0.18181819   0.45454547        0.25         0.2
#> mcc                       0.73029673    0.6030227  0.56360185   0.6778302
#> mean_per_class_accuracy   0.8636364   0.77272725     0.78125      0.8375
#> mean_per_class_error      0.13636364   0.22727273     0.21875      0.1625
#> mse                       0.12954943    0.2684783  0.21720445  0.14379291
#> precision                  0.8333333    0.6666667   0.7647059   0.8235294
#> r2                        0.48180228 -0.076353945  0.13118221   0.4242292
#> recall                    0.90909094          1.0      0.8125       0.875
#> rmse                      0.35992974    0.5181489  0.46605197  0.37920037
#> specificity                0.8181818   0.54545456        0.75         0.8
#>                          cv_15_valid
#> accuracy                  0.93333334
#> auc                        0.9259259
#> err                       0.06666667
#> err_count                        1.0
#> f0point5                  0.88235295
#> f1                         0.9230769
#> f2                         0.9677419
#> lift_top_group                   2.5
#> logloss                   0.36462396
#> max_per_class_error       0.11111111
#> mcc                        0.8728716
#> mean_per_class_accuracy    0.9444444
#> mean_per_class_error     0.055555556
#> mse                       0.10722163
#> precision                 0.85714287
#> r2                         0.5532432
#> recall                           1.0
#> rmse                      0.32744715
#> specificity                0.8888889
#>
#> Scoring History:
#>               timestamp          duration training_speed    epochs
#> 1  2019-09-18 16:03:00         0.000 sec             NA   0.00000
#> 2  2019-09-18 16:03:01  1 min 59.189 sec   4177 obs/sec  10.72013
#> 3  2019-09-18 16:03:02  2 min  0.244 sec   3836 obs/sec  21.44025
#> 4  2019-09-18 16:03:03  2 min  1.069 sec   4020 obs/sec  32.16038
#> 5  2019-09-18 16:03:03  2 min  1.962 sec   4023 obs/sec  42.88050
#> 6  2019-09-18 16:03:04  2 min  2.760 sec   4124 obs/sec  53.60063
#> 7  2019-09-18 16:03:05  2 min  3.686 sec   4085 obs/sec  64.32075
#> 8  2019-09-18 16:03:06  2 min  4.490 sec   4141 obs/sec  75.04088
#> 9  2019-09-18 16:03:07  2 min  5.281 sec   4202 obs/sec  85.76101
#> 10 2019-09-18 16:03:07  2 min  5.955 sec   4309 obs/sec  96.48113
#> 11 2019-09-18 16:03:08  2 min  6.661 sec   4384 obs/sec 107.20126
```

```
#>    iterations      samples training_rmse training_logloss training_r2
#> 1           0     0.000000            NA               NA          NA
#> 2           1  3409.000000       0.38306          0.53514     0.41304
#> 3           2  6818.000000       0.34810          0.38507     0.51531
#> 4           3 10227.000000       0.30740          0.33144     0.62202
#> 5           4 13636.000000       0.28218          0.27393     0.68149
#> 6           5 17045.000000       0.28487          0.27781     0.67538
#> 7           6 20454.000000       0.23736          0.20509     0.77464
#> 8           7 23863.000000       0.20890          0.16462     0.82544
#> 9           8 27272.000000       0.23711          0.18927     0.77511
#> 10          9 30681.000000       0.19374          0.13858     0.84986
#> 11         10 34090.000000       0.14574          0.08839     0.91504
#>    training_auc training_pr_auc training_lift
#> 1            NA              NA            NA
#> 2       0.88710         0.85340       2.00613
#> 3       0.92690         0.88764       2.00613
#> 4       0.93618         0.89868       2.00613
#> 5       0.94796         0.90958       2.00613
#> 6       0.95537         0.92408       2.00613
#> 7       0.96431         0.93000       2.00613
#> 8       0.97265         0.94703       2.00613
#> 9       0.97640         0.95408       2.00613
#> 10      0.98002         0.95901       2.00613
#> 11      0.98470         0.96384       2.00613
#>    training_classification_error validation_rmse validation_logloss
#> 1                             NA              NA                 NA
#> 2                        0.17737         0.40981            0.65225
#> 3                        0.13150         0.37642            0.46161
#> 4                        0.11621         0.41046            0.70123
#> 5                        0.09786         0.40806            0.60142
#> 6                        0.08257         0.41888            0.73485
#> 7                        0.06422         0.42726            0.83088
#> 8                        0.04893         0.43291            0.84339
#> 9                        0.04893         0.38827            0.67725
#> 10                       0.03976         0.43238            1.02128
#> 11                       0.02446         0.43052            1.00304
#>    validation_r2 validation_auc validation_pr_auc validation_lift
#> 1             NA             NA                NA             NA
#> 2        0.27759        0.85581           0.86142        1.58140
#> 3        0.39051        0.86233           0.87462        1.58140
#> 4        0.27531        0.86047           0.87686        1.58140
#> 5        0.28377        0.88000           0.89082        1.58140
#> 6        0.24529        0.87256           0.88604        1.58140
#> 7        0.21476        0.84651           0.87261        1.58140
#> 8        0.19386        0.83070           0.85766        1.58140
#> 9        0.35154        0.87814           0.89373        1.58140
#> 10       0.19585        0.85395           0.87313        1.58140
#> 11       0.20276        0.86977           0.88944        1.58140
#>    validation_classification_error
#> 1                               NA
#> 2                          0.16176
#> 3                          0.17647
#> 4                          0.17647
```

```
#> 5                          0.17647
#> 6                          0.19118
#> 7                          0.17647
#> 8                          0.16176
#> 9                          0.16176
#> 10                         0.17647
#> 11                         0.16176
#>
#> Variable Importances: (Extract with `h2o.varimp`)
#> =================================================
#>
#> Variable Importances:
#>   variable relative_importance scaled_importance percentage
#> 1     V169            1.000000          1.000000   0.015382
#> 2     V103            0.874618          0.874618   0.013453
#> 3     V136            0.870775          0.870775   0.013394
#> 4     V239            0.843140          0.843140   0.012969
#> 5     V112            0.836208          0.836208   0.012862
#>
#> ---
#>    variable relative_importance scaled_importance percentage
#> 85      V88            0.643007          0.643007   0.009891
#> 86     V179            0.642139          0.642139   0.009877
#> 87     V137            0.640863          0.640863   0.009858
#> 88     V168            0.638585          0.638585   0.009823
#> 89     V219            0.632628          0.632628   0.009731
#> 90      V33            0.626277          0.626277   0.009633
```

One performance metric we are usually interested in is the mean per class error for training and validation data.

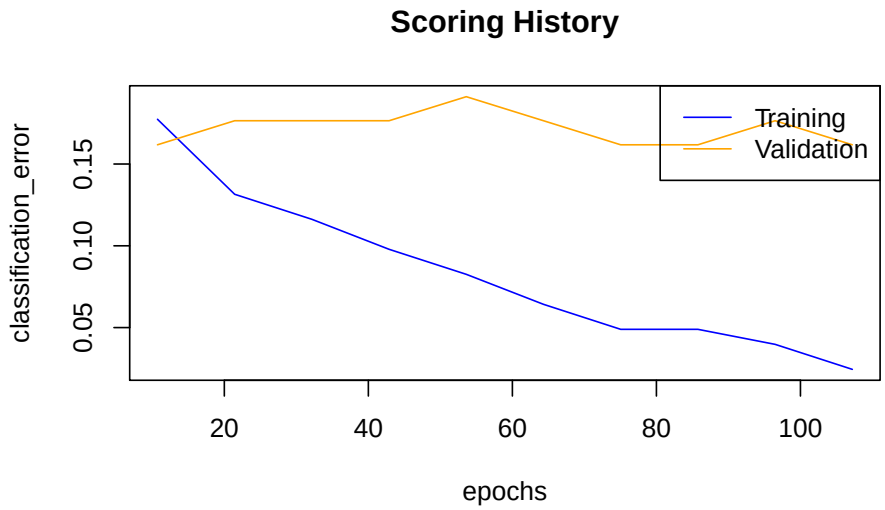
```
h2o.mean_per_class_error(dl_model, train = TRUE, valid = TRUE, xval = TRUE)
#>  train  valid   xval
#> 0.0244 0.1949 0.2080
```

The confusion matrix tells us, how many classes have been predicted correctly and how many predictions were accurate. Here, we see the errors in predictions on validation data.

```
h2o.confusionMatrix(dl_model, valid = TRUE)
#> Confusion Matrix (vertical: actual; across: predicted)  for max f1 @ threshold = 0.00565012071574118
#>            arrhythmia healthy    Error    Rate
#> arrhythmia         17       8 0.320000   =8/25
#> healthy             3      40 0.069767   =3/43
#> Totals             20      48 0.161765  =11/68
```

We can also plot the classification error over all epochs or samples.

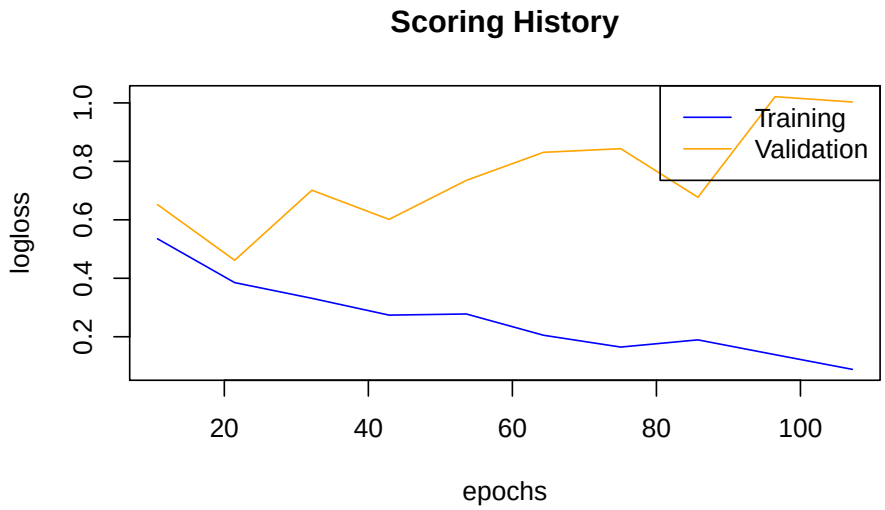
```
plot(dl_model,
     timestep = "epochs",
     metric = "classification_error")
```




```
plot(dl_model,
     timestep = "samples",
     metric = "classification_error")
```

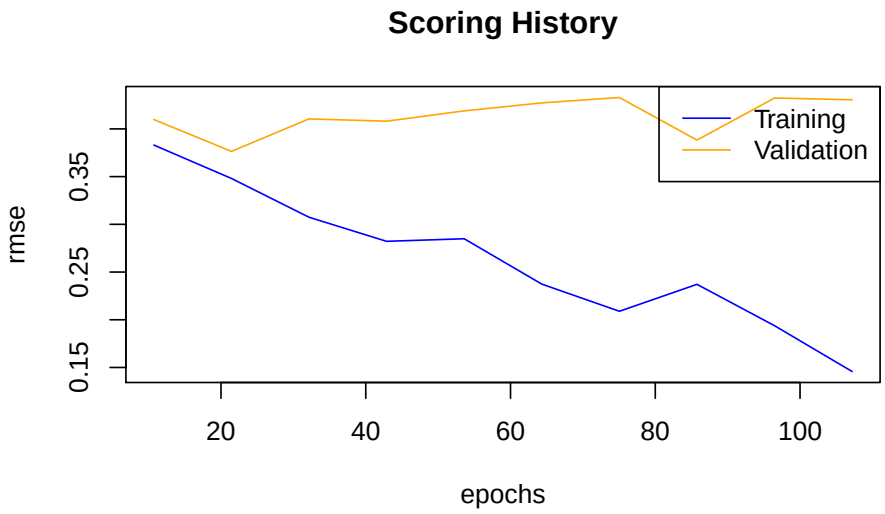
Next to the classification error, we are usually interested in the logistic loss (negative log-likelihood or log loss). It describes the sum of errors for each sample in the training or validation data or the negative logarithm of the likelihood of error for a given prediction/ classification. Simply put, the lower the loss, the better the model (if we ignore potential overfitting).

```
plot(dl_model,
     timestep = "epochs",
     metric = "logloss")
```



We can also plot the mean squared error (MSE). The **MSE** tells us the average of the prediction errors squared, i.e. the estimator's variance and bias. The closer to zero, the better a model.

```
plot(dl_model,
     timestep = "epochs",
     metric = "rmse")
```



Next, we want to know the area under the curve (AUC). **AUC** is an important metric for measuring binary classification model performances. It gives the area under the curve, i.e. the integral, of true positive vs false positive rates. The closer to 1, the better a model.

```
h2o.auc(dl_model, train = TRUE)
#> [1] 0.985
```

```
h2o.auc(dl_model, valid = TRUE)
#> [1] 0.87
```

```
h2o.auc(dl_model, xval = TRUE)
#> [1] 0.843
```

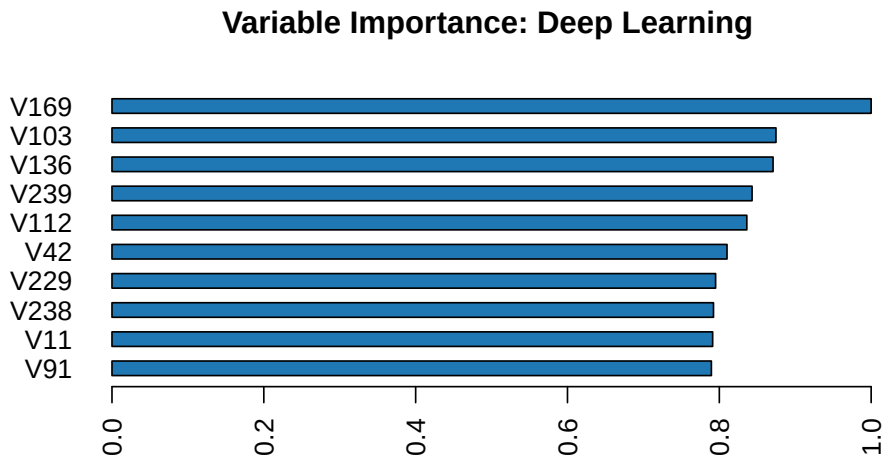
The weights for connecting two adjacent layers and per-neuron biases that we specified the model to save, can be accessed with:

```
w <- h2o.weights(dl_model, matrix_id = 1)
b <- h2o.biases(dl_model, vector_id = 1)
```

Variable importance can be extracted as well (but keep in mind, that variable importance in deep neural networks is difficult to assess and should be considered only as rough estimates).

```
h2o.varimp(dl_model)
#> Variable Importances:
#>   variable relative_importance scaled_importance percentage
#> 1     V169            1.000000          1.000000   0.015382
#> 2     V103            0.874618          0.874618   0.013453
#> 3     V136            0.870775          0.870775   0.013394
#> 4     V239            0.843140          0.843140   0.012969
#> 5     V112            0.836208          0.836208   0.012862
#>
#> ---
#>    variable relative_importance scaled_importance percentage
#> 85      V88            0.643007          0.643007   0.009891
#> 86     V179            0.642139          0.642139   0.009877
#> 87     V137            0.640863          0.640863   0.009858
#> 88     V168            0.638585          0.638585   0.009823
#> 89     V219            0.632628          0.632628   0.009731
#> 90      V33            0.626277          0.626277   0.009633
```

```
h2o.varimp_plot(dl_model)
```



1.7 Test data

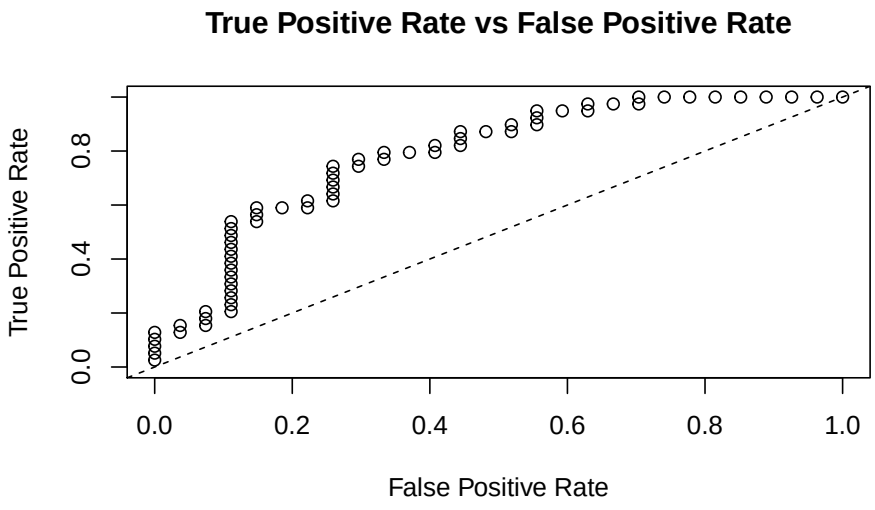
Now that we have a good idea about model performance on validation data, we want to know how it performed on unseen test data. A good model should find an optimal balance between accuracy on training and test data. A model that has 0% error on the training data but 40% error on the test data is in effect useless. It overfit on the training data and is thus not able to generalize to unknown data.

```
perf <- h2o.performance(dl_model, test)
perf
#> H2OBinomialMetrics: deeplearning
#>
#> MSE:  0.256
#> RMSE:  0.506
```

```
#> LogLoss:  1.63
#> Mean Per-Class Error:  0.303
#> AUC:  0.786
#> pr_auc:  0.788
#> Gini:  0.573
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>            arrhythmia healthy    Error    Rate
#> arrhythmia         12      15 0.555556  =15/27
#> healthy             2      37 0.051282   =2/39
#> Totals             14      52 0.257576  =17/66
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>                         metric threshold    value idx
#> 1                       max f1  0.000015 0.813187  51
#> 2                       max f2  0.000003 0.911215  57
#> 3                 max f0point5  0.376976 0.792350  35
#> 4                 max accuracy  0.376976 0.742424  35
#> 5                max precision  0.990135 1.000000   0
#> 6                   max recall  0.000003 1.000000  57
#> 7              max specificity  0.990135 1.000000   0
#> 8             max absolute_mcc  0.376976 0.478238  35
#> 9   max min_per_class_accuracy  0.376976 0.740741  35
#> 10 max mean_per_class_accuracy  0.376976 0.742165  35
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
```

Plotting the test performance's AUC plot shows us approximately how good the predictions are.

```
plot(perf)
```



We also want to know the log loss, MSE and AUC values, as well as other model metrics for the test data:

```
h2o.logloss(perf)
#> [1] 1.63
```

```
h2o.mse(perf)
#> [1] 0.256
```

```
h2o.auc(perf)
#> [1] 0.786
```

```
head(h2o.metric(perf))
#> Metrics for Thresholds: Binomial metrics as a function of classification thresholds
#>   threshold       f1       f2 f0point5 accuracy precision   recall
#> 1  0.990135 0.050000 0.031847 0.116279 0.424242  1.000000 0.025641
#> 2  0.988445 0.097561 0.063291 0.212766 0.439394  1.000000 0.051282
#> 3  0.988408 0.142857 0.094340 0.294118 0.454545  1.000000 0.076923
#> 4  0.986825 0.186047 0.125000 0.363636 0.469697  1.000000 0.102564
#> 5  0.986705 0.227273 0.155280 0.423729 0.484848  1.000000 0.128205
#> 6  0.986509 0.222222 0.154321 0.396825 0.469697  0.833333 0.128205
#>   specificity absolute_mcc min_per_class_accuracy mean_per_class_accuracy
#> 1    1.000000     0.103203               0.025641                0.512821
#> 2    1.000000     0.147087               0.051282                0.525641
#> 3    1.000000     0.181568               0.076923                0.538462
#> 4    1.000000     0.211341               0.102564                0.551282
#> 5    1.000000     0.238215               0.128205                0.564103
#> 6    0.962963     0.155921               0.128205                0.545584
#>   tns fns fps tps      tnr      fnr      fpr      tpr idx
#> 1  27  38   0   1 1.000000 0.974359 0.000000 0.025641   0
#> 2  27  37   0   2 1.000000 0.948718 0.000000 0.051282   1
#> 3  27  36   0   3 1.000000 0.923077 0.000000 0.076923   2
#> 4  27  35   0   4 1.000000 0.897436 0.000000 0.102564   3
#> 5  27  34   0   5 1.000000 0.871795 0.000000 0.128205   4
#> 6  26  34   1   5 0.962963 0.871795 0.037037 0.128205   5
```

The confusion matrix alone can be seen with the `h2o.confusionMatrix()` function, but is is also part of the performance summary.

```
h2o.confusionMatrix(dl_model, test)
#> Confusion Matrix (vertical: actual; across: predicted)  for max f1 @ threshold = 1.49810039577815e-0
#>             arrhythmia healthy    Error    Rate
#> arrhythmia         12      15 0.555556  =15/27
#> healthy             2      37 0.051282   =2/39
#> Totals             14      52 0.257576  =17/66
```

The final predictions with probabilities can be extracted with the `h2o.predict()` function. Beware though, that the number of correct and wrong classifications can be slightly different from the confusion matrix above.

Here, I combine the predictions with the actual test diagnoses and classes into a data frame. For plotting I also want to have a column, that tells me whether the predictions were correct. By default, a prediction probability above 0.5 will get scored as a prediction for the respective category. I find it often makes sense to be more stringent with this, though and set a higher threshold. Therefore, I am creating another column with stringent predictions, where I only count predictions that were made with more than 80% probability. Everything that does not fall within this range gets scored as "uncertain". For these stringent predictions, I am also creating a column that tells me whether they were accurate.

```
finalRf_predictions <- data.frame(class = as.vector(test$class),
                                  actual = as.vector(test$diagnosis),
                                  as.data.frame(h2o.predict(object = dl_model,
                                                            newdata = test)))
#>
  |
  |                                                                      |   0%
```

```
  |
  |======================================================================| 100%

finalRf_predictions$accurate <- ifelse(
  finalRf_predictions$actual == finalRf_predictions$predict, "yes", "no")

finalRf_predictions$predict_stringent <- ifelse(
  finalRf_predictions$arrhythmia > 0.8, "arrhythmia",
  ifelse(finalRf_predictions$healthy > 0.8, "healthy", "uncertain"))

finalRf_predictions$accurate_stringent <- ifelse(
  finalRf_predictions$actual == finalRf_predictions$predict_stringent, "yes",
  ifelse(finalRf_predictions$predict_stringent == "uncertain", "na", "no"))
```

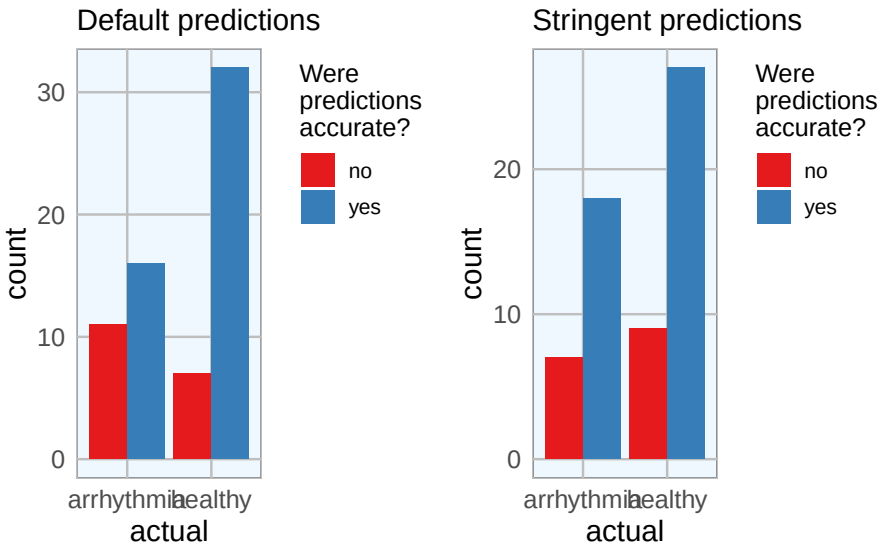
```
finalRf_predictions %>%
  group_by(actual, predict) %>%
  summarise(n = n())
#> # A tibble: 4 x 3
#> # Groups:   actual [2]
#>   actual     predict        n
#>   <fct>      <fct>      <int>
#> 1 arrhythmia arrhythmia    16
#> 2 arrhythmia healthy       11
#> 3 healthy    arrhythmia     7
#> 4 healthy    healthy       32
```

```
finalRf_predictions %>%
  group_by(actual, predict_stringent) %>%
  summarise(n = n())
#> # A tibble: 6 x 3
#> # Groups:   actual [2]
#>   actual     predict_stringent     n
#>   <fct>      <chr>             <int>
#> 1 arrhythmia arrhythmia           18
#> 2 arrhythmia healthy               7
#> 3 arrhythmia uncertain             2
#> 4 healthy    arrhythmia            9
#> 5 healthy    healthy              27
#> 6 healthy    uncertain             3
```

To get a better overview, I am going to plot the predictions (default and stringent):

```
p1 <- finalRf_predictions %>%
  ggplot(aes(x = actual, fill = accurate)) +
    geom_bar(position = "dodge") +
    scale_fill_brewer(palette = "Set1") +
    my_theme() +
    labs(fill = "Were\npredictions\naccurate?",
         title = "Default predictions")

p2 <- finalRf_predictions %>%
  subset(accurate_stringent != "na") %>%
  ggplot(aes(x = actual, fill = accurate_stringent)) +
    geom_bar(position = "dodge") +
```

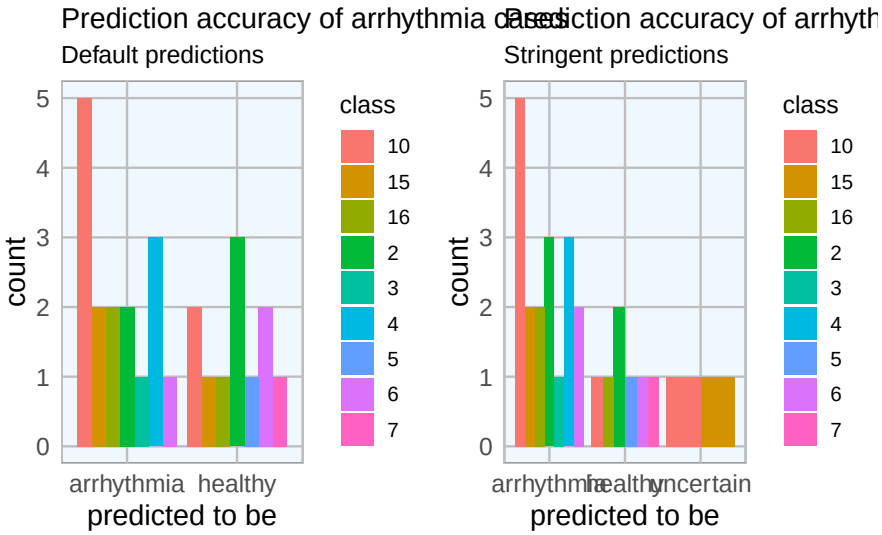
```
    scale_fill_brewer(palette = "Set1") +
    my_theme() +
    labs(fill = "Were\npredictions\naccurate?",
         title = "Stringent predictions")

grid.arrange(p1, p2, ncol = 2)
```



Being more stringent with the prediction threshold slightly reduced the number of errors but not by much.

I also want to know whether there are certain classes of arrhythmia that are especially prone to being misclassified:

```
p1 <- subset(finalRf_predictions, actual == "arrhythmia") %>%
  ggplot(aes(x = predict, fill = class)) +
    geom_bar(position = "dodge") +
    my_theme() +
    labs(title = "Prediction accuracy of arrhythmia cases",
         subtitle = "Default predictions",
         x = "predicted to be")

p2 <- subset(finalRf_predictions, actual == "arrhythmia") %>%
  ggplot(aes(x = predict_stringent, fill = class)) +
    geom_bar(position = "dodge") +
    my_theme() +
    labs(title = "Prediction accuracy of arrhythmia cases",
         subtitle = "Stringent predictions",
         x = "predicted to be")

grid.arrange(p1, p2, ncol = 2)
```



There are no obvious biases towards some classes but with the small number of samples for most classes, this is difficult to assess.

1.8 Final conclusions: How useful is the model?

Most samples were classified correctly, but the total error was not particularly good. Moreover, when evaluating the usefulness of a specific model, we need to keep in mind what we want to achieve with it and which questions we want to answer. If we wanted to deploy this model in a clinical setting, it should assist with diagnosing patients. So, we need to think about what the consequences of wrong classifications would be. Would it be better to optimize for high sensitivity, in this example as many arrhythmia cases as possible get detected - with the drawback that we probably also diagnose a few healthy people? Or do we want to maximize precision, meaning that we could be confident that a patient who got predicted to have arrhythmia does indeed have it, while accepting that a few arrhythmia cases would remain undiagnosed? When we consider stringent predictions, this model correctly classified 19 out of 27 arrhythmia cases, but 6 were misdiagnosed. This would mean that some patients who were actually sick, wouldn't have gotten the correct treatment (if decided solely based on this model). For real-life application, this is obviously not sufficient!

Next week, I'll be trying to improve the model by doing a grid search for hyper-parameter tuning.

So, stay tuned... (sorry, couldn't resist ;-))

```
sessionInfo()
#> R version 3.6.0 (2019-04-26)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Ubuntu 18.04.3 LTS
#>
#> Matrix products: default
#> BLAS/LAPACK: /usr/lib/x86_64-linux-gnu/libopenblasp-r0.2.20.so
#>
#> locale:
#>  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
#>  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
#>  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
#>  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
#>  [9] LC_ADDRESS=C               LC_TELEPHONE=C
#> [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
```

```
#>
#> attached base packages:
#>  [1] stats4    parallel  grid      stats     graphics  grDevices utils
#>  [8] datasets  methods   base
#>
#> other attached packages:
#>  [1] reshape2_1.4.3       tidyr_0.8.3          matrixStats_0.54.0
#>  [4] pcaGoPromoter_1.28.0 Biostrings_2.52.0    XVector_0.24.0
#>  [7] IRanges_2.18.0       S4Vectors_0.22.0     BiocGenerics_0.30.0
#> [10] ellipse_0.4.1        gridExtra_2.3        ggrepel_0.8.1
#> [13] ggplot2_3.1.1        h2o_3.22.1.1         dplyr_0.8.0.1
#> [16] logging_0.9-107
#>
#> loaded via a namespace (and not attached):
#>  [1] Rcpp_1.0.1           assertthat_0.2.1     zeallot_0.1.0
#>  [4] rprojroot_1.3-2      digest_0.6.18        utf8_1.1.4
#>  [7] R6_2.4.0             plyr_1.8.4           backports_1.1.4
#> [10] RSQLite_2.1.1        evaluate_0.13        pillar_1.4.0
#> [13] zlibbioc_1.30.0      rlang_0.3.4          lazyeval_0.2.2
#> [16] rstudioapi_0.10      blob_1.1.1           rmarkdown_1.12
#> [19] labeling_0.3         stringr_1.4.0        RCurl_1.95-4.12
#> [22] bit_1.1-14           munsell_0.5.0        compiler_3.6.0
#> [25] xfun_0.6             pkgconfig_2.0.2      htmltools_0.3.6
#> [28] tidyselect_0.2.5     tibble_2.1.1         bookdown_0.10
#> [31] fansi_0.4.0          crayon_1.3.4         withr_2.1.2
#> [34] bitops_1.0-6         jsonlite_1.6         gtable_0.3.0
#> [37] DBI_1.0.0            magrittr_1.5         scales_1.0.0
#> [40] cli_1.1.0            stringi_1.4.3        vctrs_0.1.0
#> [43] RColorBrewer_1.1-2   tools_3.6.0          bit64_0.9-7
#> [46] Biobase_2.44.0       glue_1.3.1           purrr_0.3.2
#> [49] yaml_2.2.0           AnnotationDbi_1.46.0 colorspace_1.4-1
#> [52] memoise_1.1.0        knitr_1.22
```

Chapter 2

Credit Scoring

2.1 Introduction

Source: https://www.r-bloggers.com/using-neural-networks-for-credit-scoring-a-simple-example/

2.2 Motivation

Credit scoring is the practice of analysing a persons background and credit application in order to assess the creditworthiness of the person. One can take numerous approaches on analysing this creditworthiness. In the end it basically comes down to first selecting the correct independent variables (e.g. income, age, gender) that lead to a given level of creditworthiness.

In other words: `creditworthiness = f(income, age, gender, …)`.

A creditscoring system can be represented by linear regression, logistic regression, machine learning or a combination of these. Neural networks are situated in the domain of machine learning. The following is an strongly simplified example. The actual procedure of building a credit scoring system is much more complex and the resulting model will most likely not consist of solely or even a neural network.

If you're unsure on what a neural network exactly is, I find this a good place to start.

For this example the R package `neuralnet` is used, for a more in-depth view on the exact workings of the package see neuralnet: `Training of Neural Networks` by F. Günther and S. Fritsch.

2.3 load the data

Dataset downloaded: https://gist.github.com/Bart6114/8675941#file-creditset-csv

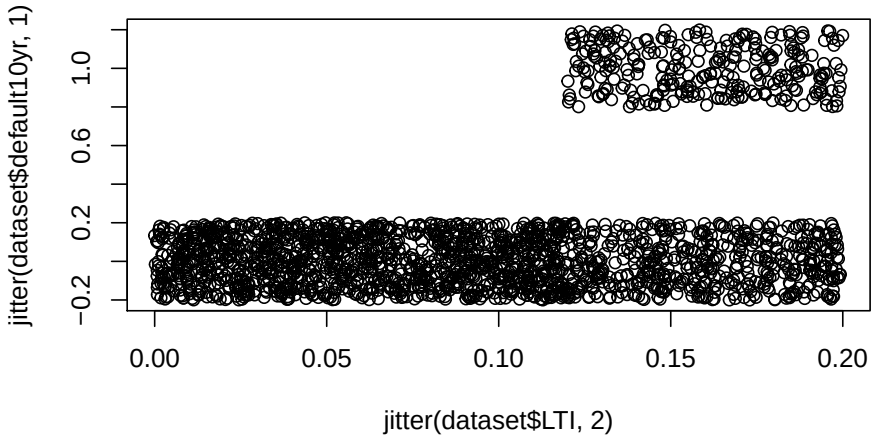
```
set.seed(1234567890)

library(neuralnet)

dataset <- read.csv(file.path(data_raw_dir, "creditset.csv"))
head(dataset)
#>   clientid income  age   loan      LTI default10yr
#> 1        1  66156 59.0 8106.5 0.122537           0
#> 2        2  34415 48.1 6564.7 0.190752           0
```

```
#> 3        3  57317 63.1 8021.0 0.139940           0
#> 4        4  42710 45.8 6103.6 0.142911           0
#> 5        5  66953 18.6 8770.1 0.130989           1
#> 6        6  24904 57.5   15.5 0.000622           0
```

```
names(dataset)
#> [1] "clientid"    "income"      "age"         "loan"        "LTI"
#> [6] "default10yr"
```

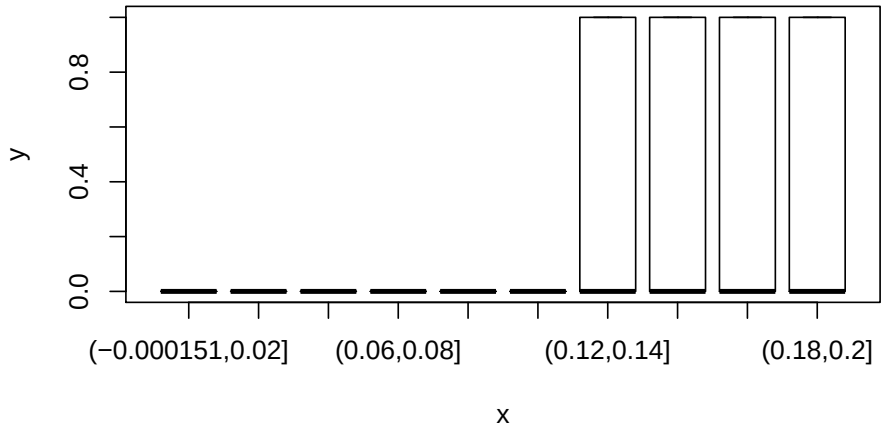
```
summary(dataset)
#>     clientid        income           age            loan
#>  Min.   :   1   Min.   :20014   Min.   :18.1   Min.   :    1
#>  1st Qu.: 501   1st Qu.:32796   1st Qu.:29.1   1st Qu.: 1940
#>  Median :1000   Median :45789   Median :41.4   Median : 3975
#>  Mean   :1000   Mean   :45332   Mean   :40.9   Mean   : 4444
#>  3rd Qu.:1500   3rd Qu.:57791   3rd Qu.:52.6   3rd Qu.: 6432
#>  Max.   :2000   Max.   :69996   Max.   :64.0   Max.   :13766
#>       LTI          default10yr
#>  Min.   :0.0000   Min.   :0.000
#>  1st Qu.:0.0479   1st Qu.:0.000
#>  Median :0.0994   Median :0.000
#>  Mean   :0.0984   Mean   :0.142
#>  3rd Qu.:0.1476   3rd Qu.:0.000
#>  Max.   :0.1999   Max.   :1.000
```

```
# distribution of defaults
table(dataset$default10yr)
#>
#>    0    1
#> 1717  283
min(dataset$LTI)
#> [1] 4.91e-05
plot(jitter(dataset$default10yr, 1) ~ jitter(dataset$LTI, 2))
```

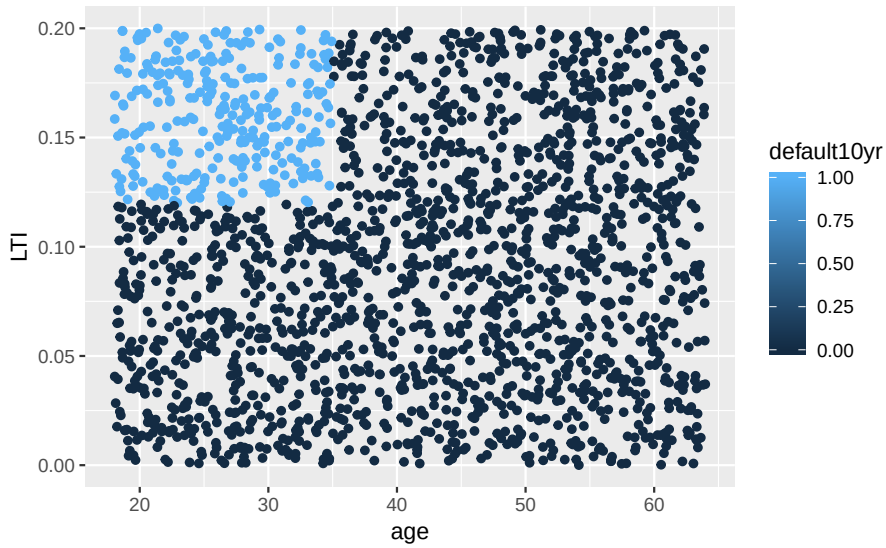



```
# convert LTI continuous variable to categorical
dataset$LTIrng <- cut(dataset$LTI, breaks = 10)
unique(dataset$LTIrng)
#>  [1] (0.12,0.14]      (0.18,0.2]       (0.14,0.16]      (-0.000151,0.02]
#>  [5] (0.1,0.12]       (0.04,0.06]      (0.06,0.08]      (0.08,0.1]
#>  [9] (0.16,0.18]      (0.02,0.04]
```

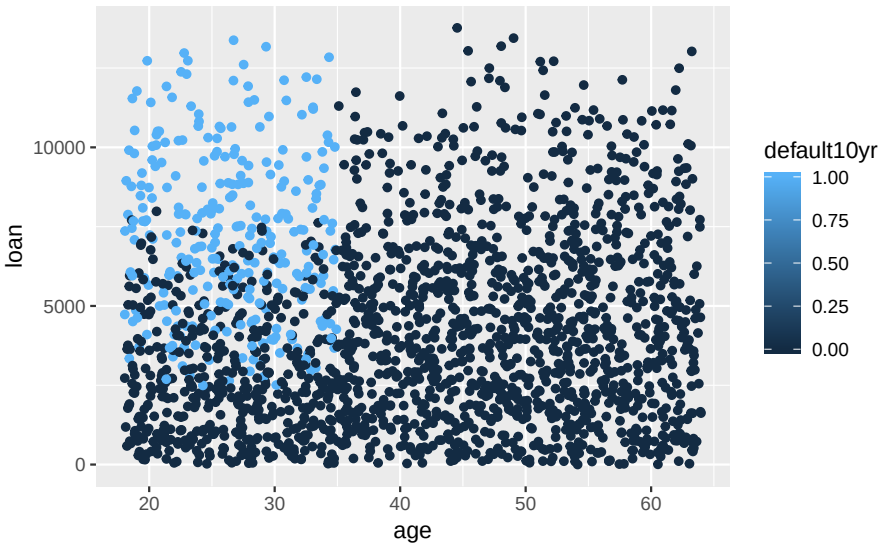
```
#> 10 Levels: (-0.000151,0.02] (0.02,0.04] (0.04,0.06] ... (0.18,0.2]
plot(dataset$LTIrng, dataset$default10yr)
```




```
# what age and LTI is more likely to default
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang

ggplot(dataset, aes(x = age, y = LTI, col = default10yr)) +
    geom_point()
```




```
# what age and loan size is more likely to default
library(ggplot2)

ggplot(dataset, aes(x = age, y = loan, col = default10yr)) +
    geom_point()
```



2.4 Objective

The dataset contains information on different clients who received a loan at least 10 years ago. The variables income (yearly), age, loan (size in euros) and LTI (the loan to yearly income ratio) are available. Our goal is to devise a model which predicts, based on the input variables LTI and age, whether or not a default will occur within 10 years.

2.5 Steps

The dataset will be split up in a subset used for training the neural network and another set used for testing. As the ordering of the dataset is completely random, we do not have to extract random rows and can just take the first x rows.

```
## extract a set to train the NN
trainset <- dataset[1:800, ]

## select the test set
testset <- dataset[801:2000, ]
```

2.5.1 Build the neural network

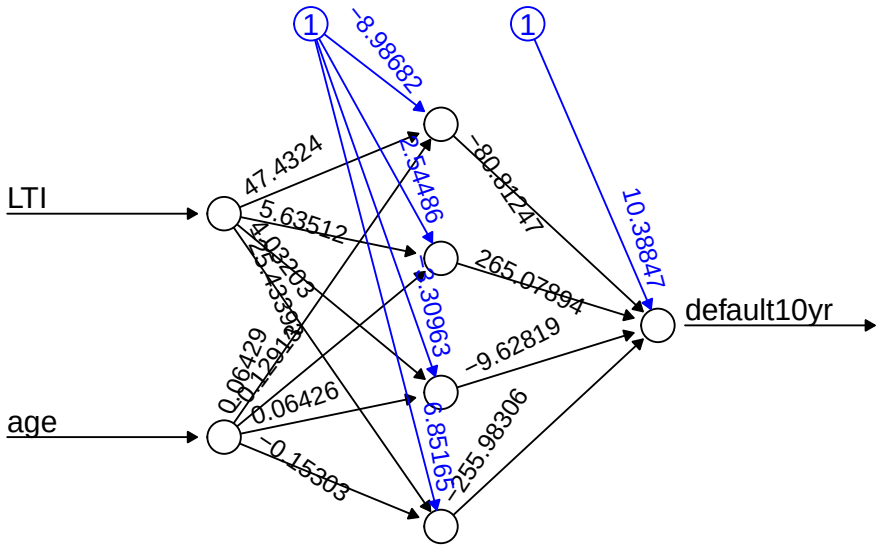
Now we'll build a neural network with 4 hidden nodes (a neural network is comprised of an input, hidden and output nodes). The number of nodes is chosen here without a clear method, however there are some rules of thumb. The `lifesign` option refers to the verbosity. The `ouput` is not linear and we will use a `threshold` value of 10%. The `neuralnet` package uses resilient backpropagation with weight backtracking as its standard algorithm.

```
## build the neural network (NN)
creditnet <- neuralnet(default10yr ~ LTI + age, trainset,
                       hidden = 4,
                       lifesign = "minimal",
                       linear.output = FALSE,
```

```
                        threshold = 0.1)
#> hidden: 4    thresh: 0.1    rep: 1/1    steps:   44487   error: 0.20554  time: 10.05 secs
```

The neuralnet package also has the possibility to visualize the generated model and show the found weights.

```
## plot the NN
plot(creditnet, rep = "best")
```



2.6 Test the neural network

Once we've trained the neural network we are ready to test it. We use the testset subset for this. The `compute` function is applied for computing the outputs based on the LTI and age inputs from the testset.

```
## test the resulting output
temp_test <- subset(testset, select = c("LTI", "age"))

creditnet.results <- compute(creditnet, temp_test)
```

The temp dataset contains only the columns LTI and age of the train set. Only these variables are used for input. The set looks as follows:

```
head(temp_test)
#>        LTI  age
#> 801 0.0231 25.9
#> 802 0.1373 40.8
#> 803 0.1046 32.5
#> 804 0.1599 53.2
#> 805 0.1116 46.5
#> 806 0.1149 47.1
```

Let's have a look at what the neural network produced:

```
results <- data.frame(actual = testset$default10yr, prediction = creditnet.results$net.result)
results[100:115, ]
#>     actual prediction
```

```
#> 900      0   7.29e-32
#> 901      0   8.17e-11
#> 902      0   4.33e-45
#> 903      1   1.00e+00
#> 904      0   8.06e-04
#> 905      0   3.54e-40
#> 906      0   1.48e-24
#> 907      1   1.00e+00
#> 908      0   1.11e-02
#> 909      0   8.05e-44
#> 910      0   6.72e-07
#> 911      1   1.00e+00
#> 912      0   9.97e-59
#> 913      1   1.00e+00
#> 914      0   3.39e-37
#> 915      0   1.18e-07
```

We can round to the nearest integer to improve readability:

```
results$prediction <- round(results$prediction)
results[100:115, ]
#>     actual prediction
#> 900      0          0
#> 901      0          0
#> 902      0          0
#> 903      1          1
#> 904      0          0
#> 905      0          0
#> 906      0          0
#> 907      1          1
#> 908      0          0
#> 909      0          0
#> 910      0          0
#> 911      1          1
#> 912      0          0
#> 913      1          1
#> 914      0          0
#> 915      0          0
```

As you can see it is pretty close! As already stated, this is a strongly simplified example. But it might serve as a basis for you to play around with your first neural network.

```
# how many predictions were wrong
indices <- which(results$actual != results$prediction)
indices
#> [1]  330 1008
```

```
# what are the predictions that failed
results[indices,]
#>      actual prediction
#> 1130      0          1
#> 1808      1          0
```

Chapter 3

Wine with neuralnet

Source: https://www.r-bloggers.com/multilabel-classification-with-neuralnet-package/

The neuralnet package is perhaps not the best option in R for using neural networks. If you ask why, for starters it does not recognize the typical formula y~., it does not support factors, it does not provide a lot of models other than a standard MLP, and it has great competitors in the nnet package that seems to be better integrated in R and can be used with the caret package, and in the MXnet package that is a high level deep learning library which provides a wide variety of neural networks.

But still, I think there is some value in the ease of use of the neuralnet package, especially for a beginner, therefore I'll be using it.

I'm going to be using both the neuralnet and, curiously enough, the nnet package. Let's load them:

```
# load libs
require(neuralnet)
#> Loading required package: neuralnet
require(nnet)
#> Loading required package: nnet
require(ggplot2)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
set.seed(10)
```

3.1 The dataset

I looked in the UCI Machine Learning Repository1 and found the wine dataset.

This dataset contains the results of a chemical analysis on 3 different kind of wines. The target variable is the label of the wine which is a factor with 3 (unordered) levels. The predictors are all continuous and represent 13 variables obtained as a result of chemical measurements.

```
# get the data file from the package location
wine_dataset_path <- file.path(data_raw_dir, "wine.data")
wine_dataset_path
#> [1] "/home/datascience/repos/machine-learning-rsuite/import/wine.data"
```

```
wines <- read.csv(wine_dataset_path)
wines
#>     X1 X14.23 X1.71 X2.43 X15.6 X127 X2.8 X3.06 X.28 X2.29 X5.64 X1.04
#> 1    1   13.2  1.78  2.14  11.2  100 2.65  2.76 0.26  1.28  4.38 1.050
#> 2    1   13.2  2.36  2.67  18.6  101 2.80  3.24 0.30  2.81  5.68 1.030
#> 3    1   14.4  1.95  2.50  16.8  113 3.85  3.49 0.24  2.18  7.80 0.860
#> 4    1   13.2  2.59  2.87  21.0  118 2.80  2.69 0.39  1.82  4.32 1.040
#> 5    1   14.2  1.76  2.45  15.2  112 3.27  3.39 0.34  1.97  6.75 1.050
#> 6    1   14.4  1.87  2.45  14.6   96 2.50  2.52 0.30  1.98  5.25 1.020
#> 7    1   14.1  2.15  2.61  17.6  121 2.60  2.51 0.31  1.25  5.05 1.060
#> 8    1   14.8  1.64  2.17  14.0   97 2.80  2.98 0.29  1.98  5.20 1.080
#> 9    1   13.9  1.35  2.27  16.0   98 2.98  3.15 0.22  1.85  7.22 1.010
#> 10   1   14.1  2.16  2.30  18.0  105 2.95  3.32 0.22  2.38  5.75 1.250
#> 11   1   14.1  1.48  2.32  16.8   95 2.20  2.43 0.26  1.57  5.00 1.170
#> 12   1   13.8  1.73  2.41  16.0   89 2.60  2.76 0.29  1.81  5.60 1.150
#> 13   1   14.8  1.73  2.39  11.4   91 3.10  3.69 0.43  2.81  5.40 1.250
#> 14   1   14.4  1.87  2.38  12.0  102 3.30  3.64 0.29  2.96  7.50 1.200
#> 15   1   13.6  1.81  2.70  17.2  112 2.85  2.91 0.30  1.46  7.30 1.280
#> 16   1   14.3  1.92  2.72  20.0  120 2.80  3.14 0.33  1.97  6.20 1.070
#> 17   1   13.8  1.57  2.62  20.0  115 2.95  3.40 0.40  1.72  6.60 1.130
#> 18   1   14.2  1.59  2.48  16.5  108 3.30  3.93 0.32  1.86  8.70 1.230
#> 19   1   13.6  3.10  2.56  15.2  116 2.70  3.03 0.17  1.66  5.10 0.960
#> 20   1   14.1  1.63  2.28  16.0  126 3.00  3.17 0.24  2.10  5.65 1.090
#> 21   1   12.9  3.80  2.65  18.6  102 2.41  2.41 0.25  1.98  4.50 1.030
#> 22   1   13.7  1.86  2.36  16.6  101 2.61  2.88 0.27  1.69  3.80 1.110
#> 23   1   12.8  1.60  2.52  17.8   95 2.48  2.37 0.26  1.46  3.93 1.090
#> 24   1   13.5  1.81  2.61  20.0   96 2.53  2.61 0.28  1.66  3.52 1.120
#> 25   1   13.1  2.05  3.22  25.0  124 2.63  2.68 0.47  1.92  3.58 1.130
#> 26   1   13.4  1.77  2.62  16.1   93 2.85  2.94 0.34  1.45  4.80 0.920
#> 27   1   13.3  1.72  2.14  17.0   94 2.40  2.19 0.27  1.35  3.95 1.020
#> 28   1   13.9  1.90  2.80  19.4  107 2.95  2.97 0.37  1.76  4.50 1.250
#> 29   1   14.0  1.68  2.21  16.0   96 2.65  2.33 0.26  1.98  4.70 1.040
#> 30   1   13.7  1.50  2.70  22.5  101 3.00  3.25 0.29  2.38  5.70 1.190
#> 31   1   13.6  1.66  2.36  19.1  106 2.86  3.19 0.22  1.95  6.90 1.090
#> 32   1   13.7  1.83  2.36  17.2  104 2.42  2.69 0.42  1.97  3.84 1.230
#> 33   1   13.8  1.53  2.70  19.5  132 2.95  2.74 0.50  1.35  5.40 1.250
#> 34   1   13.5  1.80  2.65  19.0  110 2.35  2.53 0.29  1.54  4.20 1.100
#> 35   1   13.5  1.81  2.41  20.5  100 2.70  2.98 0.26  1.86  5.10 1.040
#> 36   1   13.3  1.64  2.84  15.5  110 2.60  2.68 0.34  1.36  4.60 1.090
#> 37   1   13.1  1.65  2.55  18.0   98 2.45  2.43 0.29  1.44  4.25 1.120
#> 38   1   13.1  1.50  2.10  15.5   98 2.40  2.64 0.28  1.37  3.70 1.180
#> 39   1   14.2  3.99  2.51  13.2  128 3.00  3.04 0.20  2.08  5.10 0.890
#> 40   1   13.6  1.71  2.31  16.2  117 3.15  3.29 0.34  2.34  6.13 0.950
#> 41   1   13.4  3.84  2.12  18.8   90 2.45  2.68 0.27  1.48  4.28 0.910
#> 42   1   13.9  1.89  2.59  15.0  101 3.25  3.56 0.17  1.70  5.43 0.880
#> 43   1   13.2  3.98  2.29  17.5  103 2.64  2.63 0.32  1.66  4.36 0.820
#> 44   1   13.1  1.77  2.10  17.0  107 3.00  3.00 0.28  2.03  5.04 0.880
#> 45   1   14.2  4.04  2.44  18.9  111 2.85  2.65 0.30  1.25  5.24 0.870
#> 46   1   14.4  3.59  2.28  16.0  102 3.25  3.17 0.27  2.19  4.90 1.040
#> 47   1   13.9  1.68  2.12  16.0  101 3.10  3.39 0.21  2.14  6.10 0.910
#> 48   1   14.1  2.02  2.40  18.8  103 2.75  2.92 0.32  2.38  6.20 1.070
#> 49   1   13.9  1.73  2.27  17.4  108 2.88  3.54 0.32  2.08  8.90 1.120
#> 50   1   13.1  1.73  2.04  12.4   92 2.72  3.27 0.17  2.91  7.20 1.120
```

```
#> 51   1   13.8  1.65  2.60  17.2   94 2.45  2.99 0.22  2.29  5.60 1.240
#> 52   1   13.8  1.75  2.42  14.0  111 3.88  3.74 0.32  1.87  7.05 1.010
#> 53   1   13.8  1.90  2.68  17.1  115 3.00  2.79 0.39  1.68  6.30 1.130
#> 54   1   13.7  1.67  2.25  16.4  118 2.60  2.90 0.21  1.62  5.85 0.920
#> 55   1   13.6  1.73  2.46  20.5  116 2.96  2.78 0.20  2.45  6.25 0.980
#> 56   1   14.2  1.70  2.30  16.3  118 3.20  3.00 0.26  2.03  6.38 0.940
#> 57   1   13.3  1.97  2.68  16.8  102 3.00  3.23 0.31  1.66  6.00 1.070
#> 58   1   13.7  1.43  2.50  16.7  108 3.40  3.67 0.19  2.04  6.80 0.890
#> 59   2   12.4  0.94  1.36  10.6   88 1.98  0.57 0.28  0.42  1.95 1.050
#> 60   2   12.3  1.10  2.28  16.0  101 2.05  1.09 0.63  0.41  3.27 1.250
#> 61   2   12.6  1.36  2.02  16.8  100 2.02  1.41 0.53  0.62  5.75 0.980
#> 62   2   13.7  1.25  1.92  18.0   94 2.10  1.79 0.32  0.73  3.80 1.230
#> 63   2   12.4  1.13  2.16  19.0   87 3.50  3.10 0.19  1.87  4.45 1.220
#> 64   2   12.2  1.45  2.53  19.0  104 1.89  1.75 0.45  1.03  2.95 1.450
#> 65   2   12.4  1.21  2.56  18.1   98 2.42  2.65 0.37  2.08  4.60 1.190
#> 66   2   13.1  1.01  1.70  15.0   78 2.98  3.18 0.26  2.28  5.30 1.120
#> 67   2   12.4  1.17  1.92  19.6   78 2.11  2.00 0.27  1.04  4.68 1.120
#> 68   2   13.3  0.94  2.36  17.0  110 2.53  1.30 0.55  0.42  3.17 1.020
#> 69   2   12.2  1.19  1.75  16.8  151 1.85  1.28 0.14  2.50  2.85 1.280
#> 70   2   12.3  1.61  2.21  20.4  103 1.10  1.02 0.37  1.46  3.05 0.906
#> 71   2   13.9  1.51  2.67  25.0   86 2.95  2.86 0.21  1.87  3.38 1.360
#> 72   2   13.5  1.66  2.24  24.0   87 1.88  1.84 0.27  1.03  3.74 0.980
#> 73   2   13.0  1.67  2.60  30.0  139 3.30  2.89 0.21  1.96  3.35 1.310
#> 74   2   12.0  1.09  2.30  21.0  101 3.38  2.14 0.13  1.65  3.21 0.990
#> 75   2   11.7  1.88  1.92  16.0   97 1.61  1.57 0.34  1.15  3.80 1.230
#> 76   2   13.0  0.90  1.71  16.0   86 1.95  2.03 0.24  1.46  4.60 1.190
#> 77   2   11.8  2.89  2.23  18.0  112 1.72  1.32 0.43  0.95  2.65 0.960
#> 78   2   12.3  0.99  1.95  14.8  136 1.90  1.85 0.35  2.76  3.40 1.060
#> 79   2   12.7  3.87  2.40  23.0  101 2.83  2.55 0.43  1.95  2.57 1.190
#> 80   2   12.0  0.92  2.00  19.0   86 2.42  2.26 0.30  1.43  2.50 1.380
#> 81   2   12.7  1.81  2.20  18.8   86 2.20  2.53 0.26  1.77  3.90 1.160
#> 82   2   12.1  1.13  2.51  24.0   78 2.00  1.58 0.40  1.40  2.20 1.310
#> 83   2   13.1  3.86  2.32  22.5   85 1.65  1.59 0.61  1.62  4.80 0.840
#> 84   2   11.8  0.89  2.58  18.0   94 2.20  2.21 0.22  2.35  3.05 0.790
#> 85   2   12.7  0.98  2.24  18.0   99 2.20  1.94 0.30  1.46  2.62 1.230
#> 86   2   12.2  1.61  2.31  22.8   90 1.78  1.69 0.43  1.56  2.45 1.330
#> 87   2   11.7  1.67  2.62  26.0   88 1.92  1.61 0.40  1.34  2.60 1.360
#> 88   2   11.6  2.06  2.46  21.6   84 1.95  1.69 0.48  1.35  2.80 1.000
#> 89   2   12.1  1.33  2.30  23.6   70 2.20  1.59 0.42  1.38  1.74 1.070
#> 90   2   12.1  1.83  2.32  18.5   81 1.60  1.50 0.52  1.64  2.40 1.080
#> 91   2   12.0  1.51  2.42  22.0   86 1.45  1.25 0.50  1.63  3.60 1.050
#> 92   2   12.7  1.53  2.26  20.7   80 1.38  1.46 0.58  1.62  3.05 0.960
#> 93   2   12.3  2.83  2.22  18.0   88 2.45  2.25 0.25  1.99  2.15 1.150
#> 94   2   11.6  1.99  2.28  18.0   98 3.02  2.26 0.17  1.35  3.25 1.160
#> 95   2   12.5  1.52  2.20  19.0  162 2.50  2.27 0.32  3.28  2.60 1.160
#> 96   2   11.8  2.12  2.74  21.5  134 1.60  0.99 0.14  1.56  2.50 0.950
#> 97   2   12.3  1.41  1.98  16.0   85 2.55  2.50 0.29  1.77  2.90 1.230
#> 98   2   12.4  1.07  2.10  18.5   88 3.52  3.75 0.24  1.95  4.50 1.040
#> 99   2   12.3  3.17  2.21  18.0   88 2.85  2.99 0.45  2.81  2.30 1.420
#> 100  2   12.1  2.08  1.70  17.5   97 2.23  2.17 0.26  1.40  3.30 1.270
#> 101  2   12.6  1.34  1.90  18.5   88 1.45  1.36 0.29  1.35  2.45 1.040
#> 102  2   12.3  2.45  2.46  21.0   98 2.56  2.11 0.34  1.31  2.80 0.800
#> 103  2   11.8  1.72  1.88  19.5   86 2.50  1.64 0.37  1.42  2.06 0.940
```

```
#> 104  2   12.5  1.73  1.98  20.5   85 2.20  1.92 0.32  1.48  2.94 1.040
#> 105  2   12.4  2.55  2.27  22.0   90 1.68  1.84 0.66  1.42  2.70 0.860
#> 106  2   12.2  1.73  2.12  19.0   80 1.65  2.03 0.37  1.63  3.40 1.000
#> 107  2   12.7  1.75  2.28  22.5   84 1.38  1.76 0.48  1.63  3.30 0.880
#> 108  2   12.2  1.29  1.94  19.0   92 2.36  2.04 0.39  2.08  2.70 0.860
#> 109  2   11.6  1.35  2.70  20.0   94 2.74  2.92 0.29  2.49  2.65 0.960
#> 110  2   11.5  3.74  1.82  19.5  107 3.18  2.58 0.24  3.58  2.90 0.750
#> 111  2   12.5  2.43  2.17  21.0   88 2.55  2.27 0.26  1.22  2.00 0.900
#> 112  2   11.8  2.68  2.92  20.0  103 1.75  2.03 0.60  1.05  3.80 1.230
#> 113  2   11.4  0.74  2.50  21.0   88 2.48  2.01 0.42  1.44  3.08 1.100
#> 114  2   12.1  1.39  2.50  22.5   84 2.56  2.29 0.43  1.04  2.90 0.930
#> 115  2   11.0  1.51  2.20  21.5   85 2.46  2.17 0.52  2.01  1.90 1.710
#> 116  2   11.8  1.47  1.99  20.8   86 1.98  1.60 0.30  1.53  1.95 0.950
#> 117  2   12.4  1.61  2.19  22.5  108 2.00  2.09 0.34  1.61  2.06 1.060
#> 118  2   12.8  3.43  1.98  16.0   80 1.63  1.25 0.43  0.83  3.40 0.700
#> 119  2   12.0  3.43  2.00  19.0   87 2.00  1.64 0.37  1.87  1.28 0.930
#> 120  2   11.4  2.40  2.42  20.0   96 2.90  2.79 0.32  1.83  3.25 0.800
#> 121  2   11.6  2.05  3.23  28.5  119 3.18  5.08 0.47  1.87  6.00 0.930
#> 122  2   12.4  4.43  2.73  26.5  102 2.20  2.13 0.43  1.71  2.08 0.920
#> 123  2   13.1  5.80  2.13  21.5   86 2.62  2.65 0.30  2.01  2.60 0.730
#> 124  2   11.9  4.31  2.39  21.0   82 2.86  3.03 0.21  2.91  2.80 0.750
#> 125  2   12.1  2.16  2.17  21.0   85 2.60  2.65 0.37  1.35  2.76 0.860
#> 126  2   12.4  1.53  2.29  21.5   86 2.74  3.15 0.39  1.77  3.94 0.690
#> 127  2   11.8  2.13  2.78  28.5   92 2.13  2.24 0.58  1.76  3.00 0.970
#> 128  2   12.4  1.63  2.30  24.5   88 2.22  2.45 0.40  1.90  2.12 0.890
#> 129  2   12.0  4.30  2.38  22.0   80 2.10  1.75 0.42  1.35  2.60 0.790
#> 130  3   12.9  1.35  2.32  18.0  122 1.51  1.25 0.21  0.94  4.10 0.760
#> 131  3   12.9  2.99  2.40  20.0  104 1.30  1.22 0.24  0.83  5.40 0.740
#> 132  3   12.8  2.31  2.40  24.0   98 1.15  1.09 0.27  0.83  5.70 0.660
#> 133  3   12.7  3.55  2.36  21.5  106 1.70  1.20 0.17  0.84  5.00 0.780
#> 134  3   12.5  1.24  2.25  17.5   85 2.00  0.58 0.60  1.25  5.45 0.750
#> 135  3   12.6  2.46  2.20  18.5   94 1.62  0.66 0.63  0.94  7.10 0.730
#> 136  3   12.2  4.72  2.54  21.0   89 1.38  0.47 0.53  0.80  3.85 0.750
#> 137  3   12.5  5.51  2.64  25.0   96 1.79  0.60 0.63  1.10  5.00 0.820
#> 138  3   13.5  3.59  2.19  19.5   88 1.62  0.48 0.58  0.88  5.70 0.810
#> 139  3   12.8  2.96  2.61  24.0  101 2.32  0.60 0.53  0.81  4.92 0.890
#> 140  3   12.9  2.81  2.70  21.0   96 1.54  0.50 0.53  0.75  4.60 0.770
#> 141  3   13.4  2.56  2.35  20.0   89 1.40  0.50 0.37  0.64  5.60 0.700
#> 142  3   13.5  3.17  2.72  23.5   97 1.55  0.52 0.50  0.55  4.35 0.890
#> 143  3   13.6  4.95  2.35  20.0   92 2.00  0.80 0.47  1.02  4.40 0.910
#> 144  3   12.2  3.88  2.20  18.5  112 1.38  0.78 0.29  1.14  8.21 0.650
#> 145  3   13.2  3.57  2.15  21.0  102 1.50  0.55 0.43  1.30  4.00 0.600
#> 146  3   13.9  5.04  2.23  20.0   80 0.98  0.34 0.40  0.68  4.90 0.580
#> 147  3   12.9  4.61  2.48  21.5   86 1.70  0.65 0.47  0.86  7.65 0.540
#> 148  3   13.3  3.24  2.38  21.5   92 1.93  0.76 0.45  1.25  8.42 0.550
#> 149  3   13.1  3.90  2.36  21.5  113 1.41  1.39 0.34  1.14  9.40 0.570
#> 150  3   13.5  3.12  2.62  24.0  123 1.40  1.57 0.22  1.25  8.60 0.590
#> 151  3   12.8  2.67  2.48  22.0  112 1.48  1.36 0.24  1.26 10.80 0.480
#> 152  3   13.1  1.90  2.75  25.5  116 2.20  1.28 0.26  1.56  7.10 0.610
#> 153  3   13.2  3.30  2.28  18.5   98 1.80  0.83 0.61  1.87 10.52 0.560
#> 154  3   12.6  1.29  2.10  20.0  103 1.48  0.58 0.53  1.40  7.60 0.580
#> 155  3   13.2  5.19  2.32  22.0   93 1.74  0.63 0.61  1.55  7.90 0.600
#> 156  3   13.8  4.12  2.38  19.5   89 1.80  0.83 0.48  1.56  9.01 0.570
```

```
#> 157  3   12.4  3.03  2.64  27.0   97 1.90  0.58 0.63  1.14  7.50 0.670
#> 158  3   14.3  1.68  2.70  25.0   98 2.80  1.31 0.53  2.70 13.00 0.570
#> 159  3   13.5  1.67  2.64  22.5   89 2.60  1.10 0.52  2.29 11.75 0.570
#> 160  3   12.4  3.83  2.38  21.0   88 2.30  0.92 0.50  1.04  7.65 0.560
#> 161  3   13.7  3.26  2.54  20.0  107 1.83  0.56 0.50  0.80  5.88 0.960
#> 162  3   12.8  3.27  2.58  22.0  106 1.65  0.60 0.60  0.96  5.58 0.870
#> 163  3   13.0  3.45  2.35  18.5  106 1.39  0.70 0.40  0.94  5.28 0.680
#> 164  3   13.8  2.76  2.30  22.0   90 1.35  0.68 0.41  1.03  9.58 0.700
#> 165  3   13.7  4.36  2.26  22.5   88 1.28  0.47 0.52  1.15  6.62 0.780
#> 166  3   13.4  3.70  2.60  23.0  111 1.70  0.92 0.43  1.46 10.68 0.850
#> 167  3   12.8  3.37  2.30  19.5   88 1.48  0.66 0.40  0.97 10.26 0.720
#> 168  3   13.6  2.58  2.69  24.5  105 1.55  0.84 0.39  1.54  8.66 0.740
#> 169  3   13.4  4.60  2.86  25.0  112 1.98  0.96 0.27  1.11  8.50 0.670
#> 170  3   12.2  3.03  2.32  19.0   96 1.25  0.49 0.40  0.73  5.50 0.660
#> 171  3   12.8  2.39  2.28  19.5   86 1.39  0.51 0.48  0.64  9.90 0.570
#> 172  3   14.2  2.51  2.48  20.0   91 1.68  0.70 0.44  1.24  9.70 0.620
#> 173  3   13.7  5.65  2.45  20.5   95 1.68  0.61 0.52  1.06  7.70 0.640
#> 174  3   13.4  3.91  2.48  23.0  102 1.80  0.75 0.43  1.41  7.30 0.700
#> 175  3   13.3  4.28  2.26  20.0  120 1.59  0.69 0.43  1.35 10.20 0.590
#> 176  3   13.2  2.59  2.37  20.0  120 1.65  0.68 0.53  1.46  9.30 0.600
#> 177  3   14.1  4.10  2.74  24.5   96 2.05  0.76 0.56  1.35  9.20 0.610
#>     X3.92 X1065
#> 1    3.40  1050
#> 2    3.17  1185
#> 3    3.45  1480
#> 4    2.93   735
#> 5    2.85  1450
#> 6    3.58  1290
#> 7    3.58  1295
#> 8    2.85  1045
#> 9    3.55  1045
#> 10   3.17  1510
#> 11   2.82  1280
#> 12   2.90  1320
#> 13   2.73  1150
#> 14   3.00  1547
#> 15   2.88  1310
#> 16   2.65  1280
#> 17   2.57  1130
#> 18   2.82  1680
#> 19   3.36   845
#> 20   3.71   780
#> 21   3.52   770
#> 22   4.00  1035
#> 23   3.63  1015
#> 24   3.82   845
#> 25   3.20   830
#> 26   3.22  1195
#> 27   2.77  1285
#> 28   3.40   915
#> 29   3.59  1035
#> 30   2.71  1285
#> 31   2.88  1515
```

```
#> 32   2.87   990
#> 33   3.00  1235
#> 34   2.87  1095
#> 35   3.47   920
#> 36   2.78   880
#> 37   2.51  1105
#> 38   2.69  1020
#> 39   3.53   760
#> 40   3.38   795
#> 41   3.00  1035
#> 42   3.56  1095
#> 43   3.00   680
#> 44   3.35   885
#> 45   3.33  1080
#> 46   3.44  1065
#> 47   3.33   985
#> 48   2.75  1060
#> 49   3.10  1260
#> 50   2.91  1150
#> 51   3.37  1265
#> 52   3.26  1190
#> 53   2.93  1375
#> 54   3.20  1060
#> 55   3.03  1120
#> 56   3.31   970
#> 57   2.84  1270
#> 58   2.87  1285
#> 59   1.82   520
#> 60   1.67   680
#> 61   1.59   450
#> 62   2.46   630
#> 63   2.87   420
#> 64   2.23   355
#> 65   2.30   678
#> 66   3.18   502
#> 67   3.48   510
#> 68   1.93   750
#> 69   3.07   718
#> 70   1.82   870
#> 71   3.16   410
#> 72   2.78   472
#> 73   3.50   985
#> 74   3.13   886
#> 75   2.14   428
#> 76   2.48   392
#> 77   2.52   500
#> 78   2.31   750
#> 79   3.13   463
#> 80   3.12   278
#> 81   3.14   714
#> 82   2.72   630
#> 83   2.01   515
#> 84   3.08   520
```

```
#> 85   3.16   450
#> 86   2.26   495
#> 87   3.21   562
#> 88   2.75   680
#> 89   3.21   625
#> 90   2.27   480
#> 91   2.65   450
#> 92   2.06   495
#> 93   3.30   290
#> 94   2.96   345
#> 95   2.63   937
#> 96   2.26   625
#> 97   2.74   428
#> 98   2.77   660
#> 99   2.83   406
#> 100  2.96   710
#> 101  2.77   562
#> 102  3.38   438
#> 103  2.44   415
#> 104  3.57   672
#> 105  3.30   315
#> 106  3.17   510
#> 107  2.42   488
#> 108  3.02   312
#> 109  3.26   680
#> 110  2.81   562
#> 111  2.78   325
#> 112  2.50   607
#> 113  2.31   434
#> 114  3.19   385
#> 115  2.87   407
#> 116  3.33   495
#> 117  2.96   345
#> 118  2.12   372
#> 119  3.05   564
#> 120  3.39   625
#> 121  3.69   465
#> 122  3.12   365
#> 123  3.10   380
#> 124  3.64   380
#> 125  3.28   378
#> 126  2.84   352
#> 127  2.44   466
#> 128  2.78   342
#> 129  2.57   580
#> 130  1.29   630
#> 131  1.42   530
#> 132  1.36   560
#> 133  1.29   600
#> 134  1.51   650
#> 135  1.58   695
#> 136  1.27   720
#> 137  1.69   515
```

```
#> 138  1.82   580
#> 139  2.15   590
#> 140  2.31   600
#> 141  2.47   780
#> 142  2.06   520
#> 143  2.05   550
#> 144  2.00   855
#> 145  1.68   830
#> 146  1.33   415
#> 147  1.86   625
#> 148  1.62   650
#> 149  1.33   550
#> 150  1.30   500
#> 151  1.47   480
#> 152  1.33   425
#> 153  1.51   675
#> 154  1.55   640
#> 155  1.48   725
#> 156  1.64   480
#> 157  1.73   880
#> 158  1.96   660
#> 159  1.78   620
#> 160  1.58   520
#> 161  1.82   680
#> 162  2.11   570
#> 163  1.75   675
#> 164  1.68   615
#> 165  1.75   520
#> 166  1.56   695
#> 167  1.75   685
#> 168  1.80   750
#> 169  1.92   630
#> 170  1.83   510
#> 171  1.63   470
#> 172  1.71   660
#> 173  1.74   740
#> 174  1.56   750
#> 175  1.56   835
#> 176  1.62   840
#> 177  1.60   560
```

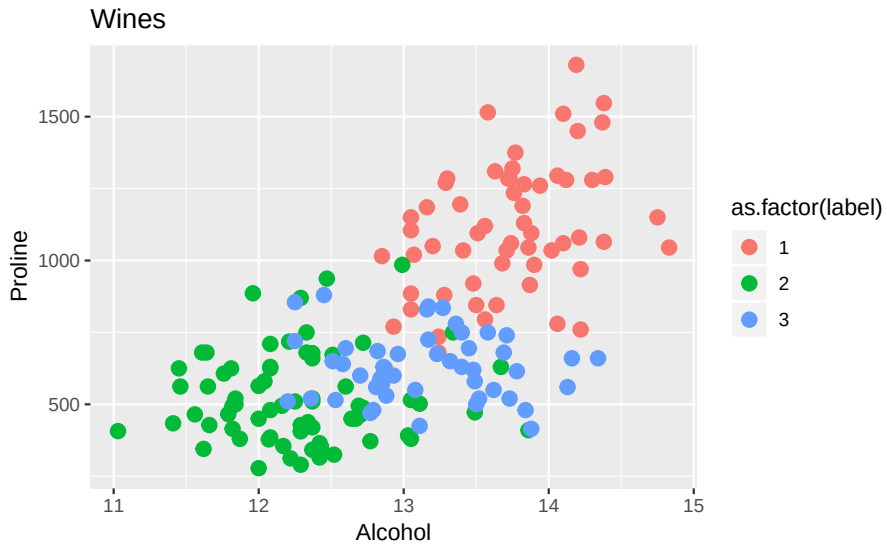
```
names(wines) <- c("label",
                  "Alcohol",
                  "Malic_acid",
                  "Ash",
                  "Alcalinity_of_ash",
                  "Magnesium",
                  "Total_phenols",
                  "Flavanoids",
                  "Nonflavanoid_phenols",
                  "Proanthocyanins",
                  "Color_intensity",
                  "Hue",
                  "OD280_OD315_of_diluted_wines",
```

```
                  "Proline")
```

```
head(wines)
#>   label Alcohol Malic_acid  Ash Alcalinity_of_ash Magnesium Total_phenols
#> 1     1    13.2       1.78 2.14              11.2       100          2.65
#> 2     1    13.2       2.36 2.67              18.6       101          2.80
#> 3     1    14.4       1.95 2.50              16.8       113          3.85
#> 4     1    13.2       2.59 2.87              21.0       118          2.80
#> 5     1    14.2       1.76 2.45              15.2       112          3.27
#> 6     1    14.4       1.87 2.45              14.6        96          2.50
#>   Flavanoids Nonflavanoid_phenols Proanthocyanins Color_intensity  Hue
#> 1       2.76                0.26            1.28            4.38 1.05
#> 2       3.24                0.30            2.81            5.68 1.03
#> 3       3.49                0.24            2.18            7.80 0.86
#> 4       2.69                0.39            1.82            4.32 1.04
#> 5       3.39                0.34            1.97            6.75 1.05
#> 6       2.52                0.30            1.98            5.25 1.02
#>   OD280_OD315_of_diluted_wines Proline
#> 1                         3.40    1050
#> 2                         3.17    1185
#> 3                         3.45    1480
#> 4                         2.93     735
#> 5                         2.85    1450
#> 6                         3.58    1290
```

```
plt1 <- ggplot(wines, aes(x = Alcohol, y = Magnesium, colour = as.factor(label))) +
    geom_point(size=3) +
    ggtitle("Wines")

plt1
```


```
plt2 <- ggplot(wines, aes(x = Alcohol, y = Proline, colour = as.factor(label))) +
    geom_point(size=3) +
    ggtitle("Wines")
plt2
```



3.2 Preprocessing

During the preprocessing phase, I have to do at least the following two things:

Encode the categorical variables. Standardize the predictors. First of all, let's encode our target variable. The encoding of the categorical variables is needed when using neuralnet since it does not like factors at all. It will shout at you if you try to feed in a factor (I am told nnet likes factors though).

In the wine dataset the variable label contains three different labels: 1,2 and 3.

The usual practice, as far as I know, is to encode categorical variables as a "one hot" vector. For instance, if I had three classes, like in this case, I'd need to replace the label variable with three variables like these:

```
#   l1,l2,l3
#   1,0,0
#   0,0,1
#   ...
```

In this case the first observation would be labelled as a 1, the second would be labelled as a 2, and so on. Ironically, the `nnet` package provides a function to perform this encoding in a painless way:

```
# Encode as a one hot vector multilabel data
train <- cbind(wines[, 2:14], class.ind(as.factor(wines$label)))

# Set labels name
names(train) <- c(names(wines)[2:14],"l1","l2","l3")
```

By the way, since the predictors are all continuous, you do not need to encode any of them, however, in case you needed to, you could apply the same strategy applied above to all the categorical predictors. Unless of course you'd like to try some other kind of custom encoding.

Now let's standardize the predictors in the [0−1]">[0−1] interval by leveraging the lapply function:

```
# Scale data
scl <- function(x) { (x - min(x))/(max(x) - min(x)) }
train[, 1:13] <- data.frame(lapply(train[, 1:13], scl))
head(train)
#>   Alcohol Malic_acid   Ash Alcalinity_of_ash Magnesium Total_phenols
#> 1   0.571      0.206 0.417            0.0309     0.326         0.576
```

```
#> 2   0.561      0.320 0.701            0.4124     0.337         0.628
#> 3   0.879      0.239 0.610            0.3196     0.467         0.990
#> 4   0.582      0.366 0.807            0.5361     0.522         0.628
#> 5   0.834      0.202 0.583            0.2371     0.457         0.790
#> 6   0.884      0.223 0.583            0.2062     0.283         0.524
#>   Flavanoids Nonflavanoid_phenols Proanthocyanins Color_intensity   Hue
#> 1      0.511                0.245           0.274           0.265 0.463
#> 2      0.612                0.321           0.757           0.375 0.447
#> 3      0.665                0.208           0.558           0.556 0.309
#> 4      0.496                0.491           0.445           0.259 0.455
#> 5      0.643                0.396           0.492           0.467 0.463
#> 6      0.460                0.321           0.495           0.339 0.439
#>   OD280_OD315_of_diluted_wines Proline l1 l2 l3
#> 1                        0.780   0.551  1  0  0
#> 2                        0.696   0.647  1  0  0
#> 3                        0.799   0.857  1  0  0
#> 4                        0.608   0.326  1  0  0
#> 5                        0.579   0.836  1  0  0
#> 6                        0.846   0.722  1  0  0
```

3.3 Fitting the model with neuralnet

Now it is finally time to fit the model.

As you might remember from the old post I wrote, `neuralnet` does not like the formula y~.. Fear not, you can build the formula to be used in a simple step:

```
# Set up formula
n <- names(train)
f <- as.formula(paste("l1 + l2 + l3 ~", paste(n[!n %in% c("l1","l2","l3")], collapse = " + ")))
f
#> l1 + l2 + l3 ~ Alcohol + Malic_acid + Ash + Alcalinity_of_ash +
#>     Magnesium + Total_phenols + Flavanoids + Nonflavanoid_phenols +
#>     Proanthocyanins + Color_intensity + Hue + OD280_OD315_of_diluted_wines +
#>     Proline
```

Note that the characters in the vector are not pasted to the right of the "~" symbol.

Just remember to check that the formula is indeed correct and then you are good to go.

Let's train the neural network with the full dataset. It should take very little time to converge. If you did not standardize the predictors it could take a lot more though.

```
nn <- neuralnet(f,
                data = train,
                hidden = c(13, 10, 3),
                act.fct = "logistic",
                linear.output = FALSE,
                lifesign = "minimal")
#> hidden: 13, 10, 3    thresh: 0.01    rep: 1/1    steps:      88  error: 0.03039  time: 0.06 secs
```

Note that I set the argument linear.output to FALSE in order to tell the model that I want to apply the activation function act.fct and that I am not doing a regression task. Then I set the activation function to logistic (which by the way is the default option) in order to apply the logistic function. The other available option is tanh but the model seems to perform a little worse with it so I opted for the default option. As

far as I know these two are the only two available options, there is no "relu" function available although it seems to be a common activation function in other packages.

As far as the number of hidden neurons, I tried some combination and the one used seems to perform slightly better than the others (around 1% of accuracy difference in cross validation score).

By using the in-built plot method you can get a visual take on what is actually happening inside the model, however the plot is not that helpful I think

```
plot(nn)
```

Let's have a look at the accuracy on the training set:

```
# Compute predictions
pr.nn <- compute(nn, train[, 1:13])

# Extract results
pr.nn_ <- pr.nn$net.result
head(pr.nn_)
#>        [,1]    [,2]     [,3]
#> [1,] 0.990 0.00317 6.99e-06
#> [2,] 0.991 0.00233 8.69e-06
#> [3,] 0.991 0.00210 8.65e-06
#> [4,] 0.986 0.00442 8.74e-06
#> [5,] 0.992 0.00212 8.32e-06
#> [6,] 0.992 0.00214 8.34e-06
```

```
# Accuracy (training set)
original_values <- max.col(train[, 14:16])
pr.nn_2 <- max.col(pr.nn_)
mean(pr.nn_2 == original_values)
#> [1] 1
```

100% not bad! But wait, this may be because our model over fitted the data, furthermore evaluating accuracy on the training set is kind of cheating since the model already "knows" (or should know) the answers. In order to assess the "true accuracy" of the model you need to perform some kind of cross validation.

3.4 Cross validating the classifier

Let's crossvalidate the model using the evergreen 10 fold cross validation with the following train and test split: 95% of the dataset will be used as training set while the remaining 5% as test set.

Just out of curiosity I decided to run a LOOCV round too. In case you'd like to run this cross validation technique, just set the proportion variable to 0.995: this will select just one observation for as test set and leave all the other observations as training set. Running LOOCV you should get similar results to the 10 fold cross validation.

```
# Set seed for reproducibility purposes
set.seed(500)
# 10 fold cross validation
k <- 10
# Results from cv
outs <- NULL
# Train test split proportions
proportion <- 0.95 # Set to 0.995 for LOOCV
```

```
# Crossvalidate, go!
for(i in 1:k)
{
    index <- sample(1:nrow(train), round(proportion*nrow(train)))
    train_cv <- train[index, ]
    test_cv <- train[-index, ]
    nn_cv <- neuralnet(f,
                       data = train_cv,
                       hidden = c(13, 10, 3),
                       act.fct = "logistic",
                       linear.output = FALSE)

    # Compute predictions
    pr.nn <- compute(nn_cv, test_cv[, 1:13])
    # Extract results
    pr.nn_ <- pr.nn$net.result
    # Accuracy (test set)
    original_values <- max.col(test_cv[, 14:16])
    pr.nn_2 <- max.col(pr.nn_)
    outs[i] <- mean(pr.nn_2 == original_values)
}

mean(outs)
#> [1] 0.978
```

98.8%, awesome! Next time when you are invited to a relaxing evening that includes a wine tasting competition I think you should definitely bring your laptop as a contestant!

Aside from that poor taste joke, (I made it again!), indeed this dataset is not the most challenging, I think with some more tweaking a better cross validation score could be achieved. Nevertheless I hope you found this tutorial useful. A gist with the entire code for this tutorial can be found here.

Thank you for reading this article, please feel free to leave a comment if you have any questions or suggestions and share the post with others if you find it useful.

Notes:

Chapter 4

Build a fully connected neural network from scratch

4.1 Introduction

http://www.parallelr.com/r-deep-neural-network-from-scratch/

```
library(neuralnet)

# Copyright 2016: www.ParallelR.com
# Parallel Blog : R For Deep Learning (I): Build Fully Connected Neural Network From Scratch
# Classification by 2-layers DNN and tested by iris dataset
# Author: Peng Zhao, patric.zhao@gmail.com

# Prediction
predict.dnn <- function(model, data = X.test) {
  # new data, transfer to matrix
  new.data <- data.matrix(data)

  # Feed Forwad
  hidden.layer <- sweep(new.data %*% model$W1 ,2, model$b1, '+')
  # neurons : Rectified Linear
  hidden.layer <- pmax(hidden.layer, 0)
  score <- sweep(hidden.layer %*% model$W2, 2, model$b2, '+')

  # Loss Function: softmax
  score.exp <- exp(score)
  probs <-sweep(score.exp, 1, rowSums(score.exp), '/')

  # select max possiblity
  labels.predicted <- max.col(probs)
  return(labels.predicted)
}

# Train: build and train a 2-layers neural network
train.dnn <- function(x, y, traindata=data, testdata=NULL,
                  model = NULL,
                  # set hidden layers and neurons
```

```
                  # currently, only support 1 hidden layer
                  hidden=c(6),
                  # max iteration steps
                  maxit=2000,
                  # delta loss
                  abstol=1e-2,
                  # learning rate
                  lr = 1e-2,
                  # regularization rate
                  reg = 1e-3,
                  # show results every 'display' step
                  display = 100,
                  random.seed = 1)
{
  # to make the case reproducible.
  set.seed(random.seed)

  # total number of training set
  N <- nrow(traindata)

  # extract the data and label
  # don't need atribute
  X <- unname(data.matrix(traindata[,x]))
  # correct categories represented by integer
  Y <- traindata[,y]
  if(is.factor(Y)) { Y <- as.integer(Y) }
  # create index for both row and col
  # create index for both row and col
  Y.len   <- length(unique(Y))
  Y.set   <- sort(unique(Y))
  Y.index <- cbind(1:N, match(Y, Y.set))

  # create model or get model from parameter
  if(is.null(model)) {
       # number of input features
       D <- ncol(X)
       # number of categories for classification
       K <- length(unique(Y))
       H <-  hidden

       # create and init weights and bias
       W1 <- 0.01*matrix(rnorm(D*H), nrow=D, ncol=H)
       b1 <- matrix(0, nrow=1, ncol=H)

       W2 <- 0.01*matrix(rnorm(H*K), nrow=H, ncol=K)
       b2 <- matrix(0, nrow=1, ncol=K)
  } else {
       D  <- model$D
       K  <- model$K
       H  <- model$H
       W1 <- model$W1
       b1 <- model$b1
       W2 <- model$W2
```

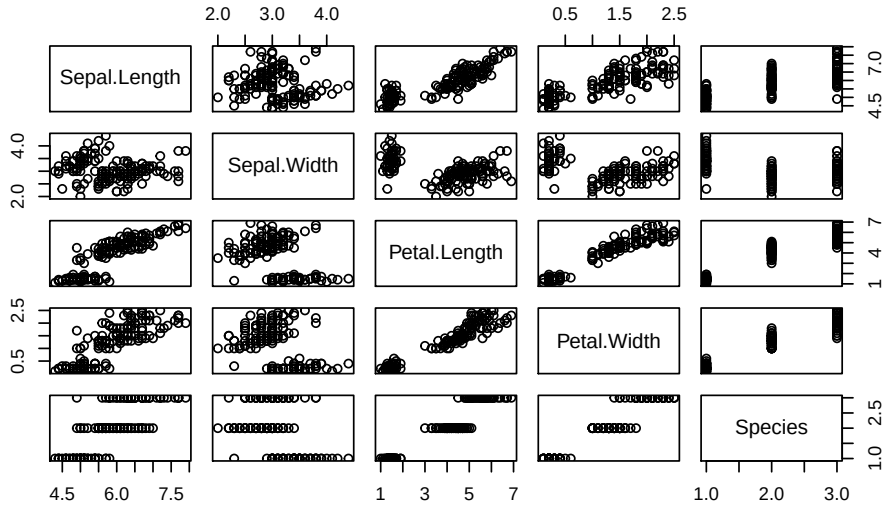
```
      b2 <- model$b2
}

# use all train data to update weights since it's a small dataset
batchsize <- N
# init loss to a very big value
loss <- 100000

# Training the network
i <- 0
while(i < maxit && loss > abstol ) {

  # iteration index
  i <- i +1

  # forward ....
  # 1 indicate row, 2 indicate col
  hidden.layer <- sweep(X %*% W1 ,2, b1, '+')
  # neurons : ReLU
  hidden.layer <- pmax(hidden.layer, 0)
  score <- sweep(hidden.layer %*% W2, 2, b2, '+')

  # softmax
  score.exp <- exp(score)
  # debug
  probs <- score.exp/rowSums(score.exp)

  # compute the loss
  corect.logprobs <- -log(probs[Y.index])
  data.loss  <- sum(corect.logprobs)/batchsize
  reg.loss   <- 0.5*reg* (sum(W1*W1) + sum(W2*W2))
  loss <- data.loss + reg.loss

  # display results and update model
  if( i %% display == 0) {
      if(!is.null(testdata)) {
          model <- list( D = D,
                         H = H,
                         K = K,
                         # weights and bias
                         W1 = W1,
                         b1 = b1,
                         W2 = W2,
                         b2 = b2)
          labs <- predict.dnn(model, testdata[,-y])
          accuracy <- mean(as.integer(testdata[,y]) == Y.set[labs])
          cat(i, loss, accuracy, "\n")
      } else {
          cat(i, loss, "\n")
      }
  }

  # backward ....
```

```
    dscores <- probs
    dscores[Y.index] <- dscores[Y.index] -1
    dscores <- dscores / batchsize


    dW2 <- t(hidden.layer) %*% dscores
    db2 <- colSums(dscores)

    dhidden <- dscores %*% t(W2)
    dhidden[hidden.layer <= 0] <- 0

    dW1 <- t(X) %*% dhidden
    db1 <- colSums(dhidden)

    # update ....
    dW2 <- dW2 + reg*W2
    dW1 <- dW1  + reg*W1

    W1 <- W1 - lr * dW1
    b1 <- b1 - lr * db1

    W2 <- W2 - lr * dW2
    b2 <- b2 - lr * db2



  }

  # final results
  # creat list to store learned parameters
  # you can add more parameters for debug and visualization
  # such as residuals, fitted.values ...
  model <- list( D = D,
                 H = H,
                 K = K,
                 # weights and bias
                 W1= W1,
                 b1= b1,
                 W2= W2,
                 b2= b2)

  return(model)
}
```

```
################################################################################
# testing
################################################################################
set.seed(1)

# 0. EDA
summary(iris)
#>   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
#>  Min.   :4.30   Min.   :2.00   Min.   :1.00   Min.   :0.1
```

```
#>  1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60   1st Qu.:0.3
#>  Median :5.80   Median :3.00   Median :4.35   Median :1.3
#>  Mean   :5.84   Mean   :3.06   Mean   :3.76   Mean   :1.2
#>  3rd Qu.:6.40   3rd Qu.:3.30   3rd Qu.:5.10   3rd Qu.:1.8
#>  Max.   :7.90   Max.   :4.40   Max.   :6.90   Max.   :2.5
#>        Species
#>  setosa    :50
#>  versicolor:50
#>  virginica :50
#>
#>
#>
plot(iris)

# 1. split data into test/train
samp <- c(sample(1:50,25), sample(51:100,25), sample(101:150,25))

# 2. train model
 ir.model <- train.dnn(x=1:4, y=5, traindata=iris[samp,], testdata=iris[-samp,], hidden=10, maxit=2000,
#> 50 1.1 0.333
#> 100 1.1 0.333
#> 150 1.09 0.333
#> 200 1.08 0.333
#> 250 1.05 0.333
#> 300 1 0.333
#> 350 0.933 0.667
#> 400 0.855 0.667
#> 450 0.775 0.667
#> 500 0.689 0.667
#> 550 0.611 0.68
#> 600 0.552 0.693
#> 650 0.507 0.747
#> 700 0.473 0.84
#> 750 0.445 0.88
#> 800 0.421 0.92
#> 850 0.399 0.947
#> 900 0.379 0.96
#> 950 0.36 0.96
#> 1000 0.341 0.973
#> 1050 0.324 0.973
#> 1100 0.307 0.973
#> 1150 0.292 0.973
#> 1200 0.277 0.973
#> 1250 0.263 0.973
#> 1300 0.25 0.973
#> 1350 0.238 0.973
#> 1400 0.227 0.973
#> 1450 0.216 0.973
#> 1500 0.207 0.973
#> 1550 0.198 0.973
#> 1600 0.19 0.973
#> 1650 0.183 0.973
#> 1700 0.176 0.973
```

```
#> 1750 0.17 0.973
#> 1800 0.164 0.973
#> 1850 0.158 0.973
#> 1900 0.153 0.973
#> 1950 0.149 0.973
#> 2000 0.144 0.973
# ir.model <- train.dnn(x=1:4, y=5, traindata=iris[samp,], hidden=6, maxit=2000, display=50)
```



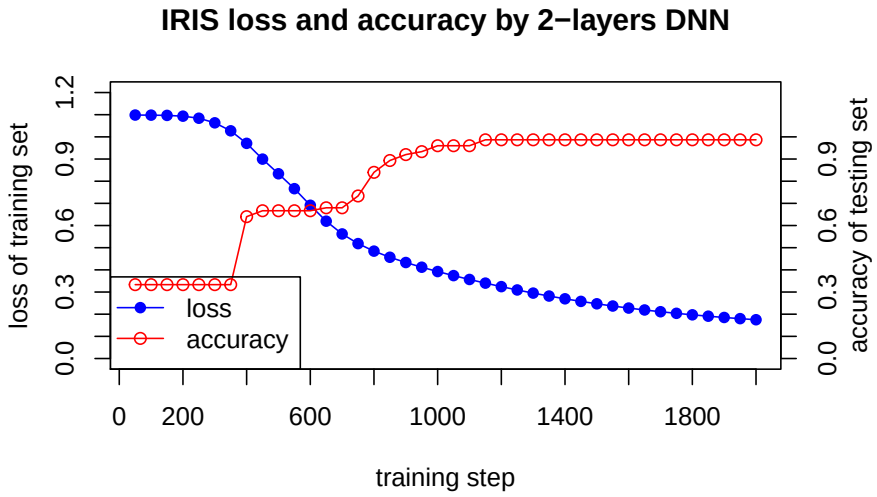

```
# 3. prediction
# NOTE: if the predict is factor, we need to transfer the number into class manually.
#       To make the code clear, I don't write this change into predict.dnn function.
labels.dnn <- predict.dnn(ir.model, iris[-samp, -5])

# 4. verify the results
table(iris[-samp,5], labels.dnn)
#>              labels.dnn
#>                1  2  3
#>   setosa      25  0  0
#>   versicolor   0 23  2
#>   virginica    0  0 25
#          labels.dnn
#            1  2  3
#setosa     25  0  0
#versicolor  0 24  1
#virginica   0  0 25
```

```
#accuracy
mean(as.integer(iris[-samp, 5]) == labels.dnn)
#> [1] 0.973
# 0.98

# 5. compare with nnet
library(nnet)
ird <- data.frame(rbind(iris3[,,1], iris3[,,2], iris3[,,3]),
                  species = factor(c(rep("s",50), rep("c", 50), rep("v", 50))))
ir.nn2 <- nnet(species ~ ., data = ird, subset = samp, size = 6, rang = 0.1,
               decay = 1e-2, maxit = 2000)
#> # weights:  51
```

```
#> initial  value 82.293110
#> iter  10 value 29.196376
#> iter  20 value 5.446284
#> iter  30 value 4.782022
#> iter  40 value 4.379729
#> iter  50 value 4.188725
#> iter  60 value 4.120587
#> iter  70 value 4.091706
#> iter  80 value 4.086017
#> iter  90 value 4.081664
#> iter 100 value 4.074111
#> iter 110 value 4.072894
#> iter 120 value 4.069011
#> iter 130 value 4.067690
#> iter 140 value 4.067633
#> final  value 4.067633
#> converged


labels.nnet <- predict(ir.nn2, ird[-samp,], type="class")
table(ird$species[-samp], labels.nnet)
#>    labels.nnet
#>      c  s  v
#>   c 23  0  2
#>   s  0 25  0
#>   v  0  0 25
#  labels.nnet
#   c  s  v
#c 22  0  3
#s  0 25  0
#v  3  0 22

# accuracy
mean(ird$species[-samp] == labels.nnet)
#> [1] 0.973
# 0.96
```

```
# Visualization
# the output from screen, copy and paste here.
data1 <- ("i loss accuracy
50 1.098421 0.3333333
100 1.098021 0.3333333
150 1.096843 0.3333333
200 1.093393 0.3333333
250 1.084069 0.3333333
300 1.063278 0.3333333
350 1.027273 0.3333333
400 0.9707605 0.64
450 0.8996356 0.6666667
500 0.8335469 0.6666667
550 0.7662386 0.6666667
600 0.6914156 0.6666667
650 0.6195753 0.68
```

```
700 0.5620381 0.68
750 0.5184008 0.7333333
800 0.4844815 0.84
850 0.4568258 0.8933333
900 0.4331083 0.92
950 0.4118948 0.9333333
1000 0.392368 0.96
1050 0.3740457 0.96
1100 0.3566594 0.96
1150 0.3400993 0.9866667
1200 0.3243276 0.9866667
1250 0.3093422 0.9866667
1300 0.2951787 0.9866667
1350 0.2818472 0.9866667
1400 0.2693641 0.9866667
1450 0.2577245 0.9866667
1500 0.2469068 0.9866667
1550 0.2368819 0.9866667
1600 0.2276124 0.9866667
1650 0.2190535 0.9866667
1700 0.2111565 0.9866667
1750 0.2038719 0.9866667
1800 0.1971507 0.9866667
1850 0.1909452 0.9866667
1900 0.1852105 0.9866667
1950 0.1799045 0.9866667
2000 0.1749881 0.9866667  ")

data.v <- read.table(text=data1, header=T)
par(mar=c(5.1, 4.1, 4.1, 4.1))
plot(x=data.v$i, y=data.v$loss, type="o", col="blue", pch=16,
     main="IRIS loss and accuracy by 2-layers DNN",
     ylim=c(0, 1.2),
     xlab="",
     ylab="",
     axe =F)
lines(x=data.v$i, y=data.v$accuracy, type="o", col="red", pch=1)
box()
axis(1, at=seq(0,2000,by=200))
axis(4, at=seq(0,1.0,by=0.1))
axis(2, at=seq(0,1.2,by=0.1))
mtext("training step", 1, line=3)
mtext("loss of training set", 2, line=2.5)
mtext("accuracy of testing set", 4, line=2)

legend("bottomleft",
       legend = c("loss", "accuracy"),
       pch = c(16,1),
       col = c("blue","red"),
       lwd=c(1,1)
)
```


IRIS loss and accuracy by 2−layers DNN
loss of training set
0.0
0.3
0.6
0.9
1.2
accuracy of testing set
0.0
0.3
0.6
0.9
loss
accuracy
0
200
600
1000
1400
1800
training step

Chapter 5

Classification and Regression with H2O Deep Learning

5.1 Introduction

Source: http://docs.h2o.ai/h2o-tutorials/latest-stable/tutorials/deeplearning/index.html

Repo: https://github.com/h2oai/h2o-tutorials

This tutorial shows how a H2O Deep Learning model can be used to do supervised classification and regression. A great tutorial about Deep Learning is given by Quoc Le here and here. This tutorial covers usage of H2O from R. A python version of this tutorial will be available as well in a separate document. This file is available in plain R, R markdown and regular markdown formats, and the plots are available as PDF files. All documents are available on Github.

If run from plain R, execute R in the directory of this script. If run from RStudio, be sure to setwd() to the location of this script.`h2o.init()` starts H2O in R's current working directory. `h2o.importFile()` looks for files from the perspective of where H2O was started.

More examples and explanations can be found in our H2O Deep Learning booklet and on our H2O Github Repository. The PDF slide deck can be found on Github.

5.2 H2O R Package

Load the H2O R package:

Source: http://docs.h2o.ai/h2o-tutorials/latest-stable/tutorials/deeplearning/index.html

```
## R installation instructions are at http://h2o.ai/download
library(h2o)
#>
#> ----------------------------------------------------------------------
#>
#> Your next step is to start H2O:
#>     > h2o.init()
#>
#> For H2O package documentation, ask for help:
#>     > ??h2o
#>
```

```
#> After starting H2O, you can use the Web UI at http://localhost:54321
#> For more information visit http://docs.h2o.ai
#>
#> ----------------------------------------------------------------------
#>
#> Attaching package: 'h2o'
#> The following objects are masked from 'package:stats':
#>
#>     cor, sd, var
#> The following objects are masked from 'package:base':
#>
#>     &&, %*%, %in%, ||, apply, as.factor, as.numeric, colnames,
#>     colnames<-, ifelse, is.character, is.factor, is.numeric, log,
#>     log10, log1p, log2, round, signif, trunc
```

5.3 Start H2O

Start up a 1-node H2O server on your local machine, and allow it to use all CPU cores and up to 2GB of memory:

```
h2o.init(nthreads=-1, max_mem_size="2G")
#>
#> H2O is not running yet, starting it now...
#>
#> Note:  In case of errors look at the following log files:
#>     /tmp/RtmpgHej5Z/h2o_datascience_started_from_r.out
#>     /tmp/RtmpgHej5Z/h2o_datascience_started_from_r.err
#>
#>
#> Starting H2O JVM and connecting: . Connection successful!
#>
#> R is connected to the H2O cluster:
#>     H2O cluster uptime:         1 seconds 298 milliseconds
#>     H2O cluster timezone:       America/Chicago
#>     H2O data parsing timezone:  UTC
#>     H2O cluster version:        3.22.1.1
#>     H2O cluster version age:    8 months and 21 days !!!
#>     H2O cluster name:           H2O_started_from_R_datascience_mwl453
#>     H2O cluster total nodes:    1
#>     H2O cluster total memory:   1.78 GB
#>     H2O cluster total cores:    8
#>     H2O cluster allowed cores:  8
#>     H2O cluster healthy:        TRUE
#>     H2O Connection ip:          localhost
#>     H2O Connection port:        54321
#>     H2O Connection proxy:       NA
#>     H2O Internal Security:      FALSE
#>     H2O API Extensions:         XGBoost, Algos, AutoML, Core V3, Core V4
#>     R Version:                  R version 3.6.0 (2019-04-26)
#> Warning in h2o.clusterInfo():
#> Your H2O cluster version is too old (8 months and 21 days)!
#> Please download and install the latest version from http://h2o.ai/download/
```

```
h2o.removeAll() ## clean slate - just in case the cluster was already running
#> [1] 0
```

The h2o.deeplearning function fits H2O's Deep Learning models from within R. We can run the example from the man page using the example function, or run a longer demonstration from the h2o package using the demo function::

```
args(h2o.deeplearning)
#> function (x, y, training_frame, model_id = NULL, validation_frame = NULL,
#>     nfolds = 0, keep_cross_validation_models = TRUE, keep_cross_validation_predictions = FALSE,
#>     keep_cross_validation_fold_assignment = FALSE, fold_assignment = c("AUTO",
#>         "Random", "Modulo", "Stratified"), fold_column = NULL,
#>     ignore_const_cols = TRUE, score_each_iteration = FALSE, weights_column = NULL,
#>     offset_column = NULL, balance_classes = FALSE, class_sampling_factors = NULL,
#>     max_after_balance_size = 5, max_hit_ratio_k = 0, checkpoint = NULL,
#>     pretrained_autoencoder = NULL, overwrite_with_best_model = TRUE,
#>     use_all_factor_levels = TRUE, standardize = TRUE, activation = c("Tanh",
#>         "TanhWithDropout", "Rectifier", "RectifierWithDropout",
#>         "Maxout", "MaxoutWithDropout"), hidden = c(200, 200),
#>     epochs = 10, train_samples_per_iteration = -2, target_ratio_comm_to_comp = 0.05,
#>     seed = -1, adaptive_rate = TRUE, rho = 0.99, epsilon = 1e-08,
#>     rate = 0.005, rate_annealing = 1e-06, rate_decay = 1, momentum_start = 0,
#>     momentum_ramp = 1e+06, momentum_stable = 0, nesterov_accelerated_gradient = TRUE,
#>     input_dropout_ratio = 0, hidden_dropout_ratios = NULL, l1 = 0,
#>     l2 = 0, max_w2 = 3.4028235e+38, initial_weight_distribution = c("UniformAdaptive",
#>         "Uniform", "Normal"), initial_weight_scale = 1, initial_weights = NULL,
#>     initial_biases = NULL, loss = c("Automatic", "CrossEntropy",
#>         "Quadratic", "Huber", "Absolute", "Quantile"), distribution = c("AUTO",
#>         "bernoulli", "multinomial", "gaussian", "poisson", "gamma",
#>         "tweedie", "laplace", "quantile", "huber"), quantile_alpha = 0.5,
#>     tweedie_power = 1.5, huber_alpha = 0.9, score_interval = 5,
#>     score_training_samples = 10000, score_validation_samples = 0,
#>     score_duty_cycle = 0.1, classification_stop = 0, regression_stop = 1e-06,
#>     stopping_rounds = 5, stopping_metric = c("AUTO", "deviance",
#>         "logloss", "MSE", "RMSE", "MAE", "RMSLE", "AUC", "lift_top_group",
#>         "misclassification", "mean_per_class_error", "custom",
#>         "custom_increasing"), stopping_tolerance = 0, max_runtime_secs = 0,
#>     score_validation_sampling = c("Uniform", "Stratified"), diagnostics = TRUE,
#>     fast_mode = TRUE, force_load_balance = TRUE, variable_importances = TRUE,
#>     replicate_training_data = TRUE, single_node_mode = FALSE,
#>     shuffle_training_data = FALSE, missing_values_handling = c("MeanImputation",
#>         "Skip"), quiet_mode = FALSE, autoencoder = FALSE, sparse = FALSE,
#>     col_major = FALSE, average_activation = 0, sparsity_beta = 0,
#>     max_categorical_features = 2147483647, reproducible = FALSE,
#>     export_weights_and_biases = FALSE, mini_batch_size = 1, categorical_encoding = c("AUTO",
#>         "Enum", "OneHotInternal", "OneHotExplicit", "Binary",
#>         "Eigen", "LabelEncoder", "SortByResponse", "EnumLimited"),
#>     elastic_averaging = FALSE, elastic_averaging_moving_rate = 0.9,
#>     elastic_averaging_regularization = 0.001, export_checkpoints_dir = NULL,
#>     verbose = FALSE)
#> NULL
if (interactive()) help(h2o.deeplearning)
example(h2o.deeplearning)
#>
```

```
#> h2.dpl> ## No test:
#> h2.dpl> ##D library(h2o)
#> h2.dpl> ##D h2o.init()
#> h2.dpl> ##D iris_hf <- as.h2o(iris)
#> h2.dpl> ##D iris_dl <- h2o.deeplearning(x = 1:4, y = 5, training_frame = iris_hf, seed=123456)
#> h2.dpl> ##D
#> h2.dpl> ##D # now make a prediction
#> h2.dpl> ##D predictions <- h2o.predict(iris_dl, iris_hf)
#> h2.dpl> ## End(No test)
#> h2.dpl>
#> h2.dpl>
#> h2.dpl>
if (interactive()) demo(h2o.deeplearning)  #requires user interaction
```

While **H2O Deep Learning** has many parameters, it was designed to be just as easy to use as the other supervised training methods in H2O. Early stopping, automatic data standardization and handling of categorical variables and missing values and adaptive learning rates (per weight) reduce the amount of parameters the user has to specify. Often, it's just the number and sizes of hidden layers, the number of epochs and the activation function and maybe some regularization techniques.

5.4 Let's have some fun first: Decision Boundaries

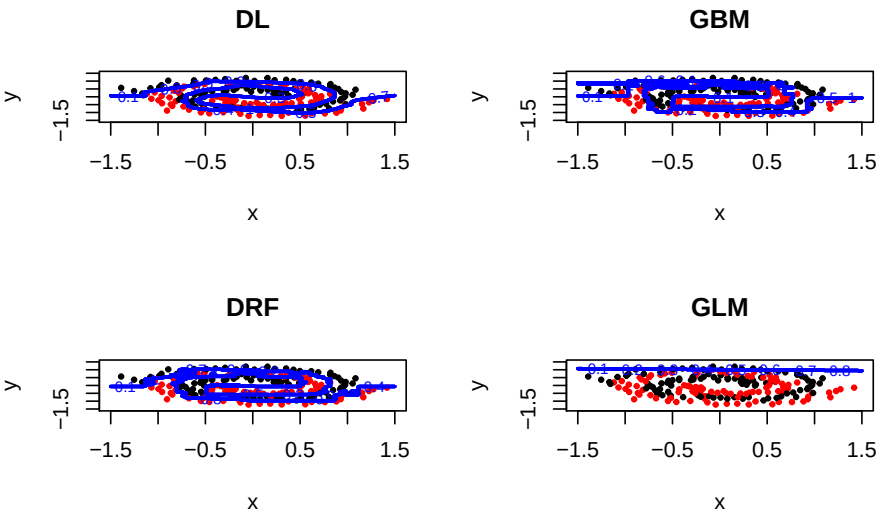
We start with a small dataset representing red and black dots on a plane, arranged in the shape of two nested spirals. Then we task H2O's machine learning methods to separate the red and black dots, i.e., recognize each spiral as such by assigning each point in the plane to one of the two spirals.

We visualize the nature of H2O Deep Learning (DL), H2O's tree methods (GBM/DRF) and H2O's generalized linear modeling (GLM) by plotting the decision boundary between the red and black spirals:

```
# setwd("~/h2o-tutorials/tutorials/deeplearning") ##For RStudio
spiral <- h2o.importFile(path = normalizePath(file.path(data_raw_dir, "spiral.csv")))
#>
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
grid   <- h2o.importFile(path = normalizePath(file.path(data_raw_dir, "grid.csv")))
#>
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%

# Define helper to plot contours
plotC <- function(name, model, data=spiral, g=grid) {
  data <- as.data.frame(data) #get data from into R
  pred <- as.data.frame(h2o.predict(model, g))
  n=0.5*(sqrt(nrow(g))-1); d <- 1.5; h <- d*(-n:n)/n
  plot(data[,-3],pch=19,col=data[,3],cex=0.5,
       xlim=c(-d,d),ylim=c(-d,d),main=name)
  contour(h,h,z=array(ifelse(pred[,1]=="Red",0,1),
          dim=c(2*n+1,2*n+1)),col="blue",lwd=2,add=T)
}
```

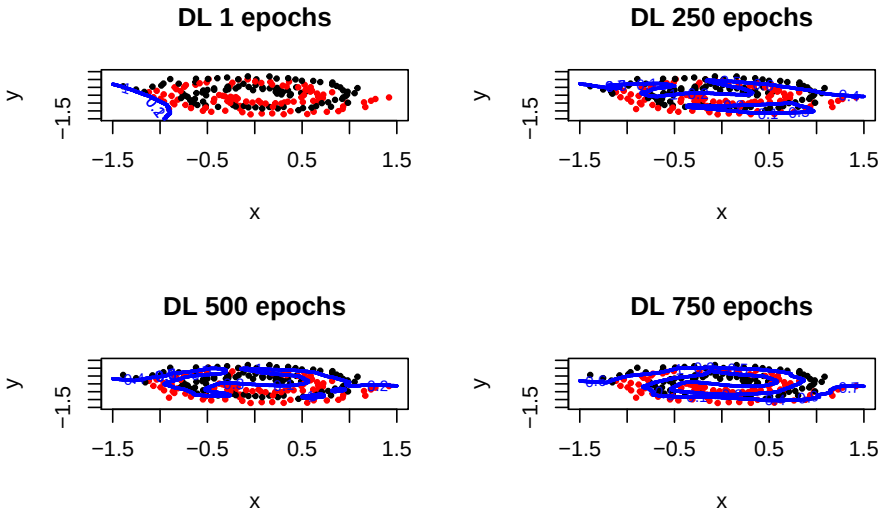
We build a few different models:

```
#dev.new(noRStudioGD=FALSE) #direct plotting output to a new window
par(mfrow=c(2,2)) #set up the canvas for 2x2 plots
plotC( "DL", h2o.deeplearning(1:2,3,spiral,epochs=1e3))
plotC("GBM", h2o.gbm         (1:2,3,spiral))
plotC("DRF", h2o.randomForest(1:2,3,spiral))
plotC("GLM", h2o.glm         (1:2,3,spiral,family="binomial"))
```



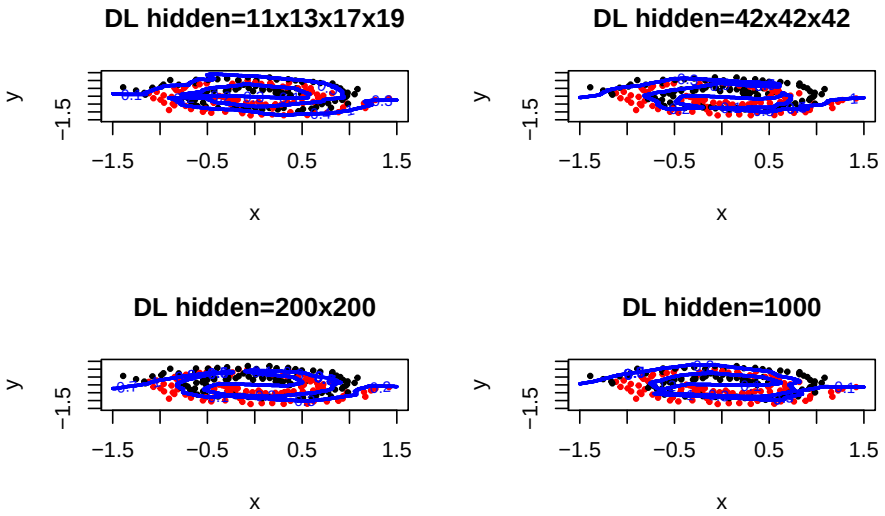
Let's investigate some more Deep Learning models. First, we explore the evolution over training time (number of passes over the data), and we use checkpointing to continue training the same model:

```
#dev.new(noRStudioGD=FALSE) #direct plotting output to a new window
par(mfrow=c(2,2)) #set up the canvas for 2x2 plots
ep <- c(1,250,500,750)
plotC(paste0("DL ",ep[1]," epochs"),
      h2o.deeplearning(1:2,3,spiral,epochs=ep[1],
                               model_id="dl_1"))
plotC(paste0("DL ",ep[2]," epochs"),
      h2o.deeplearning(1:2,3,spiral,epochs=ep[2],
            checkpoint="dl_1",model_id="dl_2"))
plotC(paste0("DL ",ep[3]," epochs"),
      h2o.deeplearning(1:2,3,spiral,epochs=ep[3],
            checkpoint="dl_2",model_id="dl_3"))
plotC(paste0("DL ",ep[4]," epochs"),
      h2o.deeplearning(1:2,3,spiral,epochs=ep[4],
            checkpoint="dl_3",model_id="dl_4"))
```



You can see how the network learns the structure of the spirals with enough training time. We explore different network architectures next:

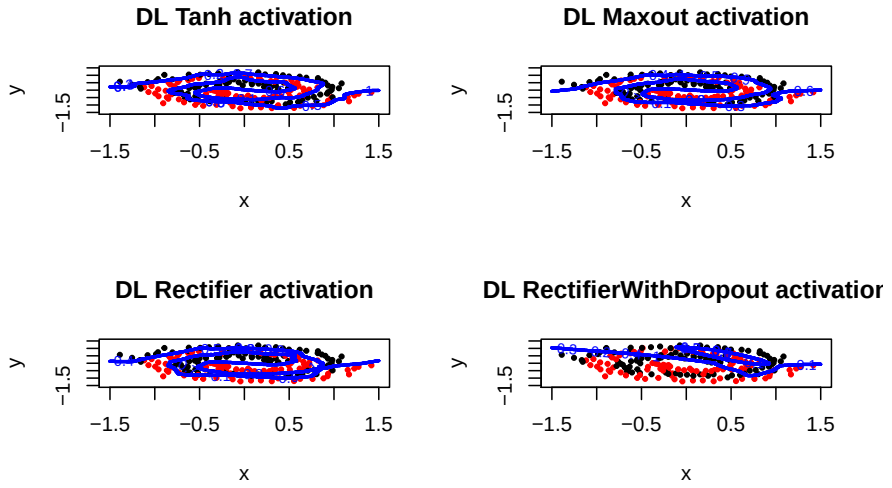
```
#dev.new(noRStudioGD=FALSE) #direct plotting output to a new window
par(mfrow=c(2,2)) #set up the canvas for 2x2 plots
for (hidden in list(c(11,13,17,19),c(42,42,42),c(200,200),c(1000))) {
  plotC(paste0("DL hidden=",paste0(hidden, collapse="x")),
        h2o.deeplearning(1:2,3 ,spiral, hidden=hidden, epochs=500))
}
```



It is clear that different configurations can achieve similar performance, and that tuning will be required for optimal performance. Next, we compare between different activation functions, including one with 50% dropout regularization in the hidden layers:

```
#dev.new(noRStudioGD=FALSE) #direct plotting output to a new window
par(mfrow=c(2,2)) #set up the canvas for 2x2 plots

for (act in c("Tanh", "Maxout", "Rectifier", "RectifierWithDropout")) {
  plotC(paste0("DL ",act," activation"),
        h2o.deeplearning(1:2,3, spiral,
              activation = act,
              hidden = c(100,100),
```

```
              epochs = 1000))
}
```



Clearly, the dropout rate was too high or the number of epochs was too low for the last configuration, which often ends up performing the best on larger datasets where generalization is important.

More information about the parameters can be found in the H2O Deep Learning booklet.

5.5 Cover Type Dataset

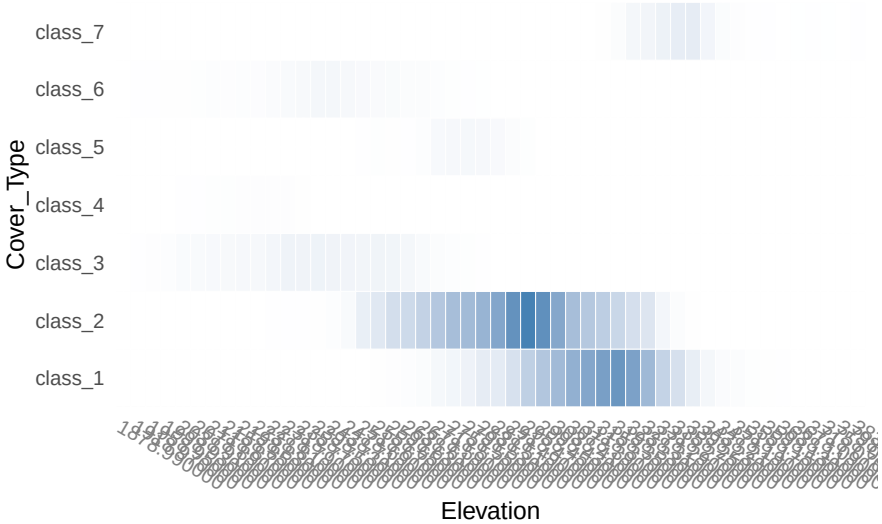
We important the full cover type dataset (581k rows, 13 columns, 10 numerical, 3 categorical). We also split the data 3 ways: 60% for training, 20% for validation (hyper parameter tuning) and 20% for final testing.

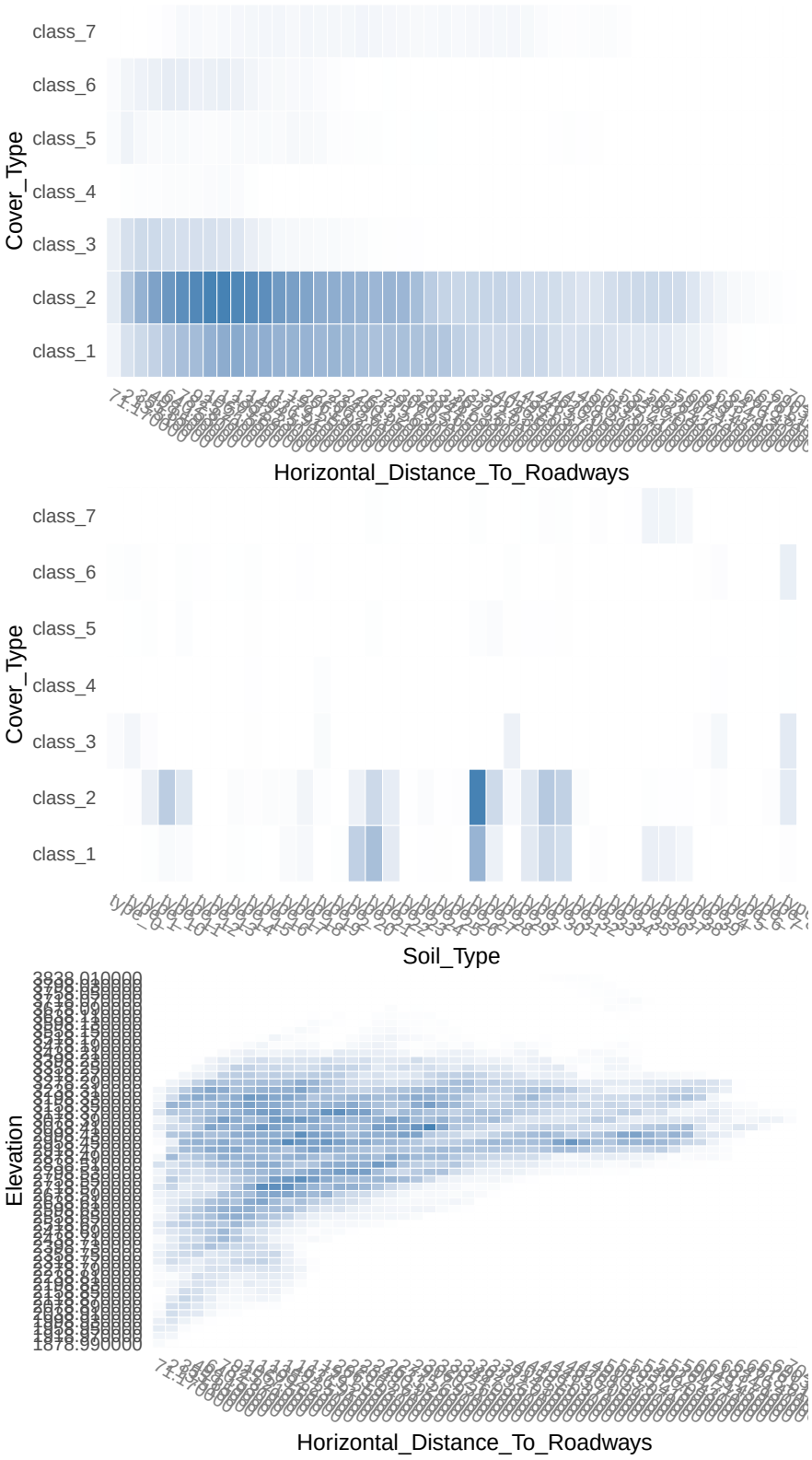
```
df <- h2o.importFile(path = normalizePath(file.path(data_raw_dir, "covtype.full.csv")))
#>
  |
  |                                                                      |   0%
  |
  |====================                                                  |  31%
  |
  |======================================================================| 100%
dim(df)
#> [1] 581012     13
df
#>   Elevation Aspect Slope Horizontal_Distance_To_Hydrology
#> 1      3066    124     5                                0
#> 2      3136     32    20                              450
#> 3      2655     28    14                               42
#> 4      3191     45    19                              323
#> 5      3217     80    13                               30
#> 6      3119    293    13                               30
#>   Vertical_Distance_To_Hydrology Horizontal_Distance_To_Roadways
#> 1                              0                            1533
#> 2                            -38                            1290
#> 3                              8                            1890
#> 4                             88                            3932
#> 5                              1                            3901
#> 6                             10                            4810
```

```
#>   Hillshade_9am Hillshade_Noon Hillshade_3pm
#> 1           229            236           141
#> 2           211            193           111
#> 3           214            209           128
#> 4           221            195           100
#> 5           237            217           109
#> 6           182            237           194
#>   Horizontal_Distance_To_Fire_Points Wilderness_Area Soil_Type Cover_Type
#> 1                                459          area_0   type_22    class_1
#> 2                               1112          area_0   type_28    class_1
#> 3                               1001          area_2    type_9    class_2
#> 4                               2919          area_0   type_39    class_2
#> 5                               2859          area_0   type_22    class_7
#> 6                               1200          area_0   type_21    class_1
#>
#> [581012 rows x 13 columns]
splits <- h2o.splitFrame(df, c(0.6, 0.2), seed=1234)
train  <- h2o.assign(splits[[1]], "train.hex") # 60%
valid  <- h2o.assign(splits[[2]], "valid.hex") # 20%
test   <- h2o.assign(splits[[3]], "test.hex")  # 20%
```

Here's a scalable way to do scatter plots via binning (works for categorical and numeric columns) to get more familiar with the dataset.

```
#dev.new(noRStudioGD=FALSE) #direct plotting output to a new window
par(mfrow=c(1,1)) # reset canvas
plot(h2o.tabulate(df, "Elevation",                       "Cover_Type"))
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
plot(h2o.tabulate(df, "Horizontal_Distance_To_Roadways", "Cover_Type"))
plot(h2o.tabulate(df, "Soil_Type",                       "Cover_Type"))
plot(h2o.tabulate(df, "Horizontal_Distance_To_Roadways", "Elevation" ))
```




class_7
class_6
class_5
class_4
class_3
class_2
class_1
Cover_Type
Horizontal_Distance_To_Roadways
Soil_Type
Elevation
Horizontal_Distance_To_Roadways

5.5.1 First Run of H2O Deep Learning

Let's run our first Deep Learning model on the covtype dataset. We want to predict the `Cover_Type` column, a categorical feature with 7 levels, and the Deep Learning model will be tasked to perform (multi-class) classification. It uses the other 12 predictors of the dataset, of which 10 are numerical, and 2 are categorical with a total of 44 levels. We can expect the Deep Learning model to have 56 input neurons (after automatic one-hot encoding).

```
response <- "Cover_Type"
predictors <- setdiff(names(df), response)
predictors
#>  [1] "Elevation"
#>  [2] "Aspect"
#>  [3] "Slope"
#>  [4] "Horizontal_Distance_To_Hydrology"
#>  [5] "Vertical_Distance_To_Hydrology"
#>  [6] "Horizontal_Distance_To_Roadways"
#>  [7] "Hillshade_9am"
#>  [8] "Hillshade_Noon"
#>  [9] "Hillshade_3pm"
#> [10] "Horizontal_Distance_To_Fire_Points"
#> [11] "Wilderness_Area"
#> [12] "Soil_Type"
```

```
train_df <- as.data.frame(train)
str(train_df)
#> 'data.frame':    349015 obs. of  13 variables:
#>  $ Elevation                         : int  3136 3217 3119 2679 3261 2885 3227 2843 2853 2883 ...
#>  $ Aspect                            : int  32 80 293 48 322 26 32 12 124 177 ...
#>  $ Slope                             : int  20 13 13 7 13 9 6 18 12 9 ...
#>  $ Horizontal_Distance_To_Hydrology  : int  450 30 30 150 30 192 108 335 30 426 ...
#>  $ Vertical_Distance_To_Hydrology    : int  -38 1 10 24 5 38 13 50 -5 126 ...
#>  $ Horizontal_Distance_To_Roadways   : int  1290 3901 4810 1588 5701 3271 5542 2642 1485 2139 ...
#>  $ Hillshade_9am                     : int  211 237 182 223 186 216 219 199 240 225 ...
#>  $ Hillshade_Noon                    : int  193 217 237 224 226 220 227 201 231 246 ...
#>  $ Hillshade_3pm                     : int  111 109 194 136 180 140 145 135 119 153 ...
#>  $ Horizontal_Distance_To_Fire_Points: int  1112 2859 1200 6265 769 2643 765 1719 2497 713 ...
#>  $ Wilderness_Area                   : Factor w/ 4 levels "area_0","area_1",..: 1 1 1 1 1 1 1 3 3 3
#>  $ Soil_Type                         : Factor w/ 40 levels "type_0","type_1",..: 22 16 15 4 15 22 15
#>  $ Cover_Type                        : Factor w/ 7 levels "class_1","class_2",..: 1 7 1 2 1 2 1 2 1
```

```
valid_df <- as.data.frame(valid)
str(valid_df)
#> 'data.frame':    116018 obs. of  13 variables:
#>  $ Elevation                         : int  3066 2655 2902 2994 2697 2990 3237 2884 2972 2696 ...
#>  $ Aspect                            : int  124 28 304 61 93 59 135 71 100 169 ...
#>  $ Slope                             : int  5 14 22 9 9 12 14 9 4 10 ...
#>  $ Horizontal_Distance_To_Hydrology  : int  0 42 511 391 306 108 240 459 175 323 ...
#>  $ Vertical_Distance_To_Hydrology    : int  0 8 18 57 -2 10 -11 141 13 149 ...
#>  $ Horizontal_Distance_To_Roadways   : int  1533 1890 1273 4286 553 2190 1189 1214 5031 2452 ...
#>  $ Hillshade_9am                     : int  229 214 155 227 234 229 241 231 227 228 ...
#>  $ Hillshade_Noon                    : int  236 209 223 222 227 215 233 222 234 244 ...
#>  $ Hillshade_3pm                     : int  141 128 206 128 125 117 118 124 142 148 ...
#>  $ Horizontal_Distance_To_Fire_Points: int  459 1001 1347 1928 1716 1048 2748 1355 6198 1044 ...
#>  $ Wilderness_Area                   : Factor w/ 4 levels "area_0","area_1",..: 1 3 3 1 1 3 1 3 1 3
```

```
#>  $ Soil_Type                          : Factor w/ 39 levels "type_0","type_1",..: 15 39 25 4 4 25 14 
#>  $ Cover_Type                         : Factor w/ 7 levels "class_1","class_2",..: 1 2 2 2 2 2 1 2 1 
```

To keep it fast, we only run for one epoch (one pass over the training data).

```
m1 <- h2o.deeplearning(
  model_id="dl_model_first",
  training_frame = train,
  validation_frame = valid,   ## validation dataset: used for scoring and early stopping
  x = predictors,
  y = response,
  #activation="Rectifier",  ## default
  #hidden=c(200,200),       ## default: 2 hidden layers with 200 neurons each
  epochs = 1,
  variable_importances=T    ## not enabled by default
)
#>
  |
  |                                                                      |   0%
  |
  |======                                                                |  10%
  |
  |=============                                                         |  20%
  |
  |===================                                                   |  30%
  |
  |==========================                                            |  40%
  |
  |================================                                      |  50%
  |
  |=======================================                               |  60%
  |
  |==============================================                        |  70%
  |
  |=====================================================                 |  80%
  |
  |============================================================          |  90%
  |
  |======================================================================| 100%
summary(m1)
#> Model Details:
#> ==============
#>
#> H2OMultinomialModel: deeplearning
#> Model Key:  dl_model_first
#> Status of Neuron Layers: predicting Cover_Type, 7-class classification, multinomial distribution, Cr
#>   layer units      type dropout       l1       l2 mean_rate rate_rms
#> 1     1    56     Input  0.00 %       NA       NA        NA       NA
#> 2     2   200 Rectifier  0.00 % 0.000000 0.000000  0.053201 0.216052
#> 3     3   200 Rectifier  0.00 % 0.000000 0.000000  0.010359 0.008217
#> 4     4     7   Softmax      NA 0.000000 0.000000  0.098231 0.258950
#>   momentum mean_weight weight_rms mean_bias bias_rms
#> 1       NA          NA         NA        NA       NA
#> 2 0.000000   -0.011749   0.117278 -0.011060 0.115161
```

```
#> 3 0.000000   -0.024653   0.119464  0.614590 0.471928
#> 4 0.000000   -0.409800   0.505029 -0.575337 0.206118
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on temporary training frame with 9864 samples **
#>
#> Training Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.126
#> RMSE: (Extract with `h2o.rmse`) 0.355
#> Logloss: (Extract with `h2o.logloss`) 0.411
#> Mean Per-Class Error: 0.307
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`
#> ========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    2806     739       1       0      10       1      17 0.2149
#> class_2     338    4417      27       0      20      47       0 0.0891
#> class_3       0      23     487       5       3      78       0 0.1829
#> class_4       0       0      14      26       0       4       0 0.4091
#> class_5       4      92       2       0      69       0       0 0.5868
#> class_6       0      35      62       4       0     179       0 0.3607
#> class_7      94      15       0       0       0       0     245 0.3079
#> Totals     3242    5321     593      35     102     309     262 0.1658
#>                    Rate
#> class_1 =   768 / 3,574
#> class_2 =   432 / 4,849
#> class_3 =     109 / 596
#> class_4 =       18 / 44
#> class_5 =      98 / 167
#> class_6 =     101 / 280
#> class_7 =     109 / 354
#> Totals  = 1,635 / 9,864
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.834246
#> 2 2  0.983779
#> 3 3  0.998479
#> 4 4  0.999595
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
#>
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on full validation frame **
#>
```

```
#> Validation Set Metrics:
#> =====================
#>
#> Extract validation frame with `h2o.getFrame("valid.hex")`
#> MSE: (Extract with `h2o.mse`) 0.131
#> RMSE: (Extract with `h2o.rmse`) 0.362
#> Logloss: (Extract with `h2o.logloss`) 0.425
#> Mean Per-Class Error: 0.33
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid = TRUE)`
#> =========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1   32950    9250       1       0     102      18     179 0.2247
#> class_2    4033   51142     345       4     317     530       9 0.0929
#> class_3       0     393    5642      65      15    1028       0 0.2101
#> class_4       0       0     202     302       0      58       0 0.4626
#> class_5      48    1043      53       0     715      11       0 0.6176
#> class_6      10     480     742      26       5    2201       0 0.3646
#> class_7    1236     138       0       0       0       0    2725 0.3352
#> Totals    38277   62446    6985     397    1154    3846    2913 0.1753
#>                     Rate
#> class_1 =   9,550 / 42,500
#> class_2 =   5,238 / 56,380
#> class_3 =    1,501 / 7,143
#> class_4 =        260 / 562
#> class_5 =    1,155 / 1,870
#> class_6 =    1,263 / 3,464
#> class_7 =    1,374 / 4,099
#> Totals  = 20,341 / 116,018
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.824674
#> 2 2  0.982830
#> 3 3  0.998043
#> 4 4  0.999647
#> 5 5  0.999991
#> 6 6  0.999991
#> 7 7  1.000000
#>
#>
#>
#>
#> Scoring History:
#>             timestamp   duration training_speed  epochs iterations
#> 1 2019-09-18 16:05:54  0.000 sec             NA 0.00000          0
#> 2 2019-09-18 16:06:00  7.612 sec   6138 obs/sec 0.10136          1
#> 3 2019-09-18 16:06:19 26.612 sec   9090 obs/sec 0.60059          6
#> 4 2019-09-18 16:06:35 42.982 sec  10173 obs/sec 1.09888         11
#>         samples training_rmse training_logloss training_r2
#> 1      0.000000            NA               NA          NA
```

```
#> 2  35375.000000       0.43680          0.59573     0.90174
#> 3 209616.000000       0.39819          0.49481     0.91834
#> 4 383527.000000       0.35493          0.41107     0.93512
#>   training_classification_error validation_rmse validation_logloss
#> 1                            NA              NA                NA
#> 2                       0.25466         0.44181           0.60760
#> 3                       0.21665         0.40487           0.50994
#> 4                       0.16575         0.36237           0.42453
#>   validation_r2 validation_classification_error
#> 1            NA                             NA
#> 2       0.89994                        0.26078
#> 3       0.91598                        0.22440
#> 4       0.93269                        0.17533
#>
#> Variable Importances: (Extract with `h2o.varimp`)
#> =================================================
#>
#> Variable Importances:
#>                            variable relative_importance scaled_importance
#> 1             Wilderness_Area.area_0            1.000000          1.000000
#> 2                          Elevation            0.910638          0.910638
#> 3    Horizontal_Distance_To_Roadways            0.902916          0.902916
#> 4 Horizontal_Distance_To_Fire_Points            0.850998          0.850998
#> 5             Wilderness_Area.area_2            0.832705          0.832705
#>   percentage
#> 1   0.033119
#> 2   0.030160
#> 3   0.029904
#> 4   0.028184
#> 5   0.027579
#>
#> ---
#>                        variable relative_importance scaled_importance
#> 51            Soil_Type.type_5            0.433003          0.433003
#> 52                       Slope            0.376897          0.376897
#> 53               Hillshade_3pm            0.361976          0.361976
#> 54                      Aspect            0.269958          0.269958
#> 55       Soil_Type.missing(NA)            0.000000          0.000000
#> 56 Wilderness_Area.missing(NA)            0.000000          0.000000
#>    percentage
#> 51   0.014341
#> 52   0.012483
#> 53   0.011988
#> 54   0.008941
#> 55   0.000000
#> 56   0.000000
```

Inspect the model in Flow for more information about model building etc. by issuing a cell with the content getModel “dl_model_first”, and pressing Ctrl-Enter.

5.5.2 Variable Importances

Variable importances for Neural Network models are notoriously difficult to compute, and there are many pitfalls. H2O Deep Learning has implemented the method of Gedeon, and returns relative variable importances in descending order of importance.

```
head(as.data.frame(h2o.varimp(m1)))
#>                              variable relative_importance scaled_importance
#> 1             Wilderness_Area.area_0               1.000             1.000
#> 2                          Elevation               0.911             0.911
#> 3    Horizontal_Distance_To_Roadways               0.903             0.903
#> 4 Horizontal_Distance_To_Fire_Points               0.851             0.851
#> 5             Wilderness_Area.area_2               0.833             0.833
#> 6             Wilderness_Area.area_1               0.737             0.737
#>   percentage
#> 1     0.0331
#> 2     0.0302
#> 3     0.0299
#> 4     0.0282
#> 5     0.0276
#> 6     0.0244
```

5.5.3 Early Stopping

Now we run another, smaller network, and we let it stop automatically once the misclassification rate converges (specifically, if the moving average of length 2 does not improve by at least 1% for 2 consecutive scoring events). We also sample the validation set to 10,000 rows for faster scoring.

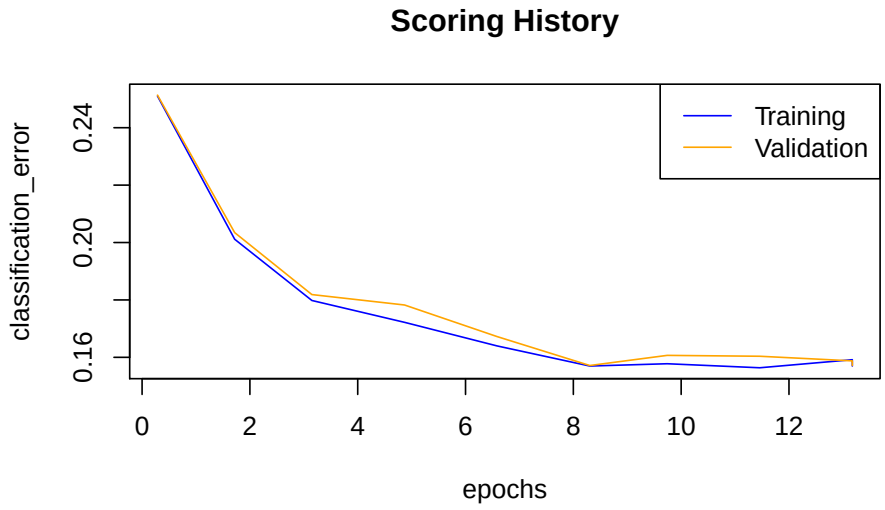
```
m2 <- h2o.deeplearning(
  model_id="dl_model_faster",
  training_frame=train,
  validation_frame=valid,
  x=predictors,
  y=response,
  hidden=c(32,32,32),                  ## small network, runs faster
  epochs=1000000,                      ## hopefully converges earlier...
  score_validation_samples=10000,      ## sample the validation dataset (faster)
  stopping_rounds=2,
  stopping_metric="misclassification", ## could be "MSE","logloss","r2"
  stopping_tolerance=0.01
)
#>
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
summary(m2)
#> Model Details:
#> ==============
#>
#> H2OMultinomialModel: deeplearning
#> Model Key:  dl_model_faster
#> Status of Neuron Layers: predicting Cover_Type, 7-class classification, multinomial distribution, Cr
#>   layer units      type dropout       l1       l2 mean_rate rate_rms
```

```
#> 1     1    56     Input  0.00 %       NA       NA        NA       NA
#> 2     2    32 Rectifier  0.00 % 0.000000 0.000000  0.043019 0.202104
#> 3     3    32 Rectifier  0.00 % 0.000000 0.000000  0.000429 0.000214
#> 4     4    32 Rectifier  0.00 % 0.000000 0.000000  0.000959 0.002080
#> 5     5     7   Softmax      NA 0.000000 0.000000  0.109826 0.286061
#>   momentum mean_weight weight_rms mean_bias bias_rms
#> 1       NA          NA         NA        NA       NA
#> 2 0.000000   -0.002224   0.263971  0.362323 0.224006
#> 3 0.000000   -0.055808   0.339650  0.712228 0.500067
#> 4 0.000000   -0.019305   0.416608  0.565933 0.755517
#> 5 0.000000   -2.929996   2.806343 -2.028375 0.757954
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on temporary training frame with 9983 samples **
#>
#> Training Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.117
#> RMSE: (Extract with `h2o.rmse`) 0.342
#> Logloss: (Extract with `h2o.logloss`) 0.385
#> Mean Per-Class Error: 0.318
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
#> ========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    3012     606       2       0       1       1      27 0.1746
#> class_2     445    4402      25       0      13      12       0 0.1011
#> class_3       1      42     485       2       1      44       0 0.1565
#> class_4       0       0      16      32       0       9       0 0.4386
#> class_5       8      95       5       0      49       4       0 0.6957
#> class_6       1      48      91       0       1     163       0 0.4638
#> class_7      61       6       0       0       0       0     273 0.1971
#> Totals     3528    5199     624      34      65     233     300 0.1570
#>                    Rate
#> class_1 =   637 / 3,649
#> class_2 =   495 / 4,897
#> class_3 =      90 / 575
#> class_4 =       25 / 57
#> class_5 =     112 / 161
#> class_6 =     141 / 304
#> class_7 =      67 / 340
#> Totals  = 1,567 / 9,983
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.843033
#> 2 2  0.985275
#> 3 3  0.997496
#> 4 4  0.999699
```

```
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
#>
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 9914 samples **
#>
#> Validation Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.119
#> RMSE: (Extract with `h2o.rmse`) 0.345
#> Logloss: (Extract with `h2o.logloss`) 0.398
#> Mean Per-Class Error: 0.321
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid = TRUE)`)
#> ========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    3055     632       0       0       1       2      32 0.1792
#> class_2     427    4287      30       0      10       9       0 0.0999
#> class_3       3      38     475       2       0      53       0 0.1681
#> class_4       0       0      26      24       0       3       0 0.5472
#> class_5      10      85       2       0      52       2       0 0.6556
#> class_6       3      53      75       0       0     177       0 0.4253
#> class_7      55       5       0       0       0       0     286 0.1734
#> Totals     3553    5100     608      26      63     246     318 0.1572
#>                    Rate
#> class_1 =   667 / 3,722
#> class_2 =   476 / 4,763
#> class_3 =      96 / 571
#> class_4 =       29 / 53
#> class_5 =      99 / 151
#> class_6 =     131 / 308
#> class_7 =      60 / 346
#> Totals  = 1,558 / 9,914
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.842849
#> 2 2  0.984567
#> 3 3  0.997882
#> 4 4  0.999597
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
#>
#>
#>
#>
```

```
#> Scoring History:
#>              timestamp   duration training_speed   epochs iterations
#> 1  2019-09-18 16:06:37  0.000 sec             NA  0.00000          0
#> 2  2019-09-18 16:06:39  1.448 sec  75582 obs/sec  0.28586          1
#> 3  2019-09-18 16:06:44  6.663 sec  92486 obs/sec  1.71768          6
#> 4  2019-09-18 16:06:49 11.922 sec  93953 obs/sec  3.15012         11
#> 5  2019-09-18 16:06:55 17.361 sec  99338 obs/sec  4.86878         17
#> 6  2019-09-18 16:07:00 23.259 sec 100074 obs/sec  6.58486         23
#> 7  2019-09-18 16:07:06 28.996 sec 101143 obs/sec  8.30562         29
#> 8  2019-09-18 16:07:11 34.122 sec 100745 obs/sec  9.74073         34
#> 9  2019-09-18 16:07:17 39.718 sec 101739 obs/sec 11.45787         40
#> 10 2019-09-18 16:07:22 44.953 sec 103308 obs/sec 13.17470         46
#> 11 2019-09-18 16:07:22 45.006 sec 103301 obs/sec 13.17470         46
#>           samples training_rmse training_logloss training_r2
#> 1        0.000000            NA               NA          NA
#> 2    99769.000000       0.43523          0.59986     0.90174
#> 3   599495.000000       0.38618          0.47742     0.92264
#> 4  1099438.000000       0.36709          0.43553     0.93010
#> 5  1699276.000000       0.35992          0.42033     0.93281
#> 6  2298214.000000       0.34863          0.40001     0.93695
#> 7  2898786.000000       0.34210          0.38546     0.93929
#> 8  3399660.000000       0.34143          0.38734     0.93953
#> 9  3998967.000000       0.34014          0.38340     0.93999
#> 10 4598167.000000       0.34099          0.38570     0.93969
#> 11 4598167.000000       0.34210          0.38546     0.93929
#>    training_classification_error validation_rmse validation_logloss
#> 1                             NA             NA                NA
#> 2                        0.25103        0.43601           0.60595
#> 3                        0.20114        0.38781           0.48037
#> 4                        0.17981        0.36861           0.43709
#> 5                        0.17219        0.36435           0.42946
#> 6                        0.16398        0.35273           0.40957
#> 7                        0.15697        0.34521           0.39827
#> 8                        0.15777        0.34679           0.40252
#> 9                        0.15637        0.34528           0.39537
#> 10                       0.15917        0.34271           0.39400
#> 11                       0.15697        0.34521           0.39827
#>    validation_r2 validation_classification_error
#> 1             NA                             NA
#> 2        0.90299                        0.25136
#> 3        0.92326                        0.20345
#> 4        0.93067                        0.18186
#> 5        0.93226                        0.17823
#> 6        0.93651                        0.16724
#> 7        0.93919                        0.15715
#> 8        0.93863                        0.16068
#> 9        0.93916                        0.16038
#> 10       0.94006                        0.15877
#> 11       0.93919                        0.15715
#>
#> Variable Importances: (Extract with `h2o.varimp`)
#> =================================================
#>
```

```
#> Variable Importances:
#>                              variable relative_importance scaled_importance
#> 1                           Elevation            1.000000          1.000000
#> 2    Horizontal_Distance_To_Roadways            0.921223          0.921223
#> 3 Horizontal_Distance_To_Fire_Points            0.888336          0.888336
#> 4             Wilderness_Area.area_3            0.876854          0.876854
#> 5                  Soil_Type.type_32            0.852413          0.852413
#>   percentage
#> 1   0.033561
#> 2   0.030917
#> 3   0.029814
#> 4   0.029428
#> 5   0.028608
#>
#> ---
#>                              variable relative_importance scaled_importance
#> 51   Vertical_Distance_To_Hydrology            0.353224          0.353224
#> 52 Horizontal_Distance_To_Hydrology            0.313158          0.313158
#> 53                           Slope            0.218501          0.218501
#> 54                          Aspect            0.076046          0.076046
#> 55           Soil_Type.missing(NA)            0.000000          0.000000
#> 56      Wilderness_Area.missing(NA)            0.000000          0.000000
#>    percentage
#> 51   0.011855
#> 52   0.010510
#> 53   0.007333
#> 54   0.002552
#> 55   0.000000
#> 56   0.000000
plot(m2)
```



5.5.4 Adaptive Learning Rate

By default, H2O Deep Learning uses an adaptive learning rate (ADADELTA) for its stochastic gradient descent optimization. There are only two tuning parameters for this method: rho and epsilon, which balance the global and local search efficiencies. rho is the similarity to prior weight updates (similar to momentum),

and epsilon is a parameter that prevents the optimization to get stuck in local optima. Defaults are rho=0.99 and epsilon=1e-8. For cases where convergence speed is very important, it might make sense to perform a few runs to optimize these two parameters (e.g., with rho in c(0.9,0.95,0.99,0.999) and epsilon in c(1e-10,1e-8,1e-6,1e-4)). Of course, as always with grid searches, caution has to be applied when extrapolating grid search results to a different parameter regime (e.g., for more epochs or different layer topologies or activation functions, etc.).

If adaptive_rate is disabled, several manual learning rate parameters become important: rate, rate_annealing, rate_decay, momentum_start, momentum_ramp, momentum_stable and `nesterov_accelerated_gradien` the discussion of which we leave to H2O Deep Learning booklet.

5.5.5 Tuning

With some tuning, it is possible to obtain less than 10% test set error rate in about one minute. Error rates of below 5% are possible with larger models. Note that deep tree methods can be more effective for this dataset than Deep Learning, as they directly partition the space into sectors, which seems to be needed here.

```
m3 <- h2o.deeplearning(
  model_id="dl_model_tuned",
  training_frame=train,
  validation_frame=valid,
  x=predictors,
  y=response,
  overwrite_with_best_model=F,    ## Return final model after 10 epochs, even if not the best
  hidden=c(128,128,128),          ## more hidden layers -> more complex interactions
  epochs=10,                      ## to keep it short enough
  score_validation_samples=10000, ## downsample validation set for faster scoring
  score_duty_cycle=0.025,         ## don't score more than 2.5% of the wall time
  adaptive_rate=F,                ## manually tuned learning rate
  rate=0.01,
  rate_annealing=2e-6,
  momentum_start=0.2,             ## manually tuned momentum
  momentum_stable=0.4,
  momentum_ramp=1e7,
  l1=1e-5,                        ## add some L1/L2 regularization
  l2=1e-5,
  max_w2=10                       ## helps stability for Rectifier
)
#>
  |
  |                                                                      |   0%
  |
  |==                                                                    |   3%
  |
  |====                                                                  |   6%
  |
  |======                                                                |   9%
  |
  |=======                                                               |  11%
  |
  |=========                                                             |  14%
  |
  |===========                                                           |  17%
  |
```

```
|=============                                                    |  20%
|
|===============                                                  |  23%
|
|=================                                                |  26%
|
|====================                                             |  29%
|
|====================                                             |  32%
|
|======================                                           |  34%
|
|========================                                         |  37%
|
|==========================                                       |  40%
|
|============================                                     |  43%
|
|==============================                                   |  46%
|
|================================                                 |  49%
|
|================================                                 |  52%
|
|==================================                               |  54%
|
|====================================                             |  57%
|
|======================================                           |  60%
|
|========================================                         |  63%
|
|==========================================                       |  66%
|
|============================================                     |  69%
|
|==============================================                   |  72%
|
|================================================                 |  74%
|
|==================================================               |  77%
|
|====================================================             |  80%
|
|======================================================           |  83%
|
|========================================================         |  86%
|
|==========================================================       |  89%
|
|============================================================     |  92%
|
|==============================================================   |  95%
```

```
  |
  |=============================================================  |  97%
  |
  |=============================================================| 100%
summary(m3)
#> Model Details:
#> ==============
#>
#> H2OMultinomialModel: deeplearning
#> Model Key:  dl_model_tuned
#> Status of Neuron Layers: predicting Cover_Type, 7-class classification, multinomial distribution, Cr
#>   layer units      type dropout       l1       l2 mean_rate rate_rms
#> 1     1    56     Input  0.00 %       NA       NA        NA       NA
#> 2     2   128 Rectifier  0.00 % 0.000010 0.000010  0.001249 0.000000
#> 3     3   128 Rectifier  0.00 % 0.000010 0.000010  0.001249 0.000000
#> 4     4   128 Rectifier  0.00 % 0.000010 0.000010  0.001249 0.000000
#> 5     5     7   Softmax      NA 0.000010 0.000010  0.001249 0.000000
#>   momentum mean_weight weight_rms mean_bias bias_rms
#> 1       NA          NA         NA        NA       NA
#> 2 0.270042   -0.010916   0.315808  0.010861 0.308801
#> 3 0.270042   -0.055477   0.221490  0.867268 0.348294
#> 4 0.270042   -0.063210   0.215936  0.804884 0.191174
#> 5 0.270042   -0.021527   0.269892  0.004505 0.794439
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on temporary training frame with 9995 samples **
#>
#> Training Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.0574
#> RMSE: (Extract with `h2o.rmse`) 0.24
#> Logloss: (Extract with `h2o.logloss`) 0.187
#> Mean Per-Class Error: 0.128
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
#> ========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>          class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    3231     367       0       0       2       2      13 0.1062
#> class_2     192    4632      10       0      10      13       3 0.0469
#> class_3       0      13     571       8       2      25       0 0.0775
#> class_4       0       0       4      35       0       4       0 0.1860
#> class_5       2      31       3       0     127       0       0 0.2209
#> class_6       1      15      36       1       0     296       0 0.1519
#> class_7      30       7       0       0       0       0     309 0.1069
#> Totals     3456    5065     624      44     141     340     325 0.0794
#>                  Rate
#> class_1 = 384 / 3,615
#> class_2 = 228 / 4,860
#> class_3 =    48 / 619
#> class_4 =      8 / 43
#> class_5 =    36 / 163
```

```
#> class_6 =    53 / 349
#> class_7 =    37 / 346
#> Totals  = 794 / 9,995
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.920560
#> 2 2  0.996498
#> 3 3  0.999700
#> 4 4  1.000000
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
#>
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 10023 samples **
#>
#> Validation Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.0616
#> RMSE: (Extract with `h2o.rmse`) 0.248
#> Logloss: (Extract with `h2o.logloss`) 0.208
#> Mean Per-Class Error: 0.161
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid = TRUE)`
#> ========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    3300     400       0       0       4       0      25 0.1150
#> class_2     168    4628       8       0      13      16       3 0.0430
#> class_3       0      19     561       6       1      26       0 0.0848
#> class_4       0       0      10      30       0       2       0 0.2857
#> class_5       3      45       4       0     109       0       0 0.3230
#> class_6       0      18      39       2       0     237       0 0.1993
#> class_7      22       5       0       0       0       0     319 0.0780
#> Totals     3493    5115     622      38     127     281     347 0.0837
#>                   Rate
#> class_1 =  429 / 3,729
#> class_2 =  208 / 4,836
#> class_3 =     52 / 613
#> class_4 =      12 / 42
#> class_5 =     52 / 161
#> class_6 =     59 / 296
#> class_7 =     27 / 346
#> Totals  = 839 / 10,023
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
```

```
#>    k hit_ratio
#> 1 1  0.916293
#> 2 2  0.995909
#> 3 3  0.999601
#> 4 4  1.000000
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
#>
#>
#>
#>
#> Scoring History:
#>               timestamp          duration training_speed   epochs
#> 1  2019-09-18 16:07:23         0.000 sec             NA  0.00000
#> 2  2019-09-18 16:07:29         6.416 sec  16447 obs/sec  0.28623
#> 3  2019-09-18 16:07:42        19.613 sec  20981 obs/sec  1.14684
#> 4  2019-09-18 16:07:51        27.872 sec  22163 obs/sec  1.72020
#> 5  2019-09-18 16:08:02        38.887 sec  21091 obs/sec  2.29155
#> 6  2019-09-18 16:08:10        46.794 sec  21912 obs/sec  2.86555
#> 7  2019-09-18 16:08:20        57.297 sec  23276 obs/sec  3.72887
#> 8  2019-09-18 16:08:32  1 min  9.478 sec  23582 obs/sec  4.58847
#> 9  2019-09-18 16:08:40  1 min 17.269 sec  23866 obs/sec  5.16196
#> 10 2019-09-18 16:08:52  1 min 28.655 sec  24252 obs/sec  6.02399
#> 11 2019-09-18 16:08:59  1 min 36.499 sec  24402 obs/sec  6.59606
#> 12 2019-09-18 16:09:07  1 min 44.628 sec  24473 obs/sec  7.16889
#> 13 2019-09-18 16:09:19  1 min 55.714 sec  24767 obs/sec  8.02787
#> 14 2019-09-18 16:09:30  2 min  6.976 sec  24977 obs/sec  8.88889
#> 15 2019-09-18 16:09:40  2 min 17.375 sec  25311 obs/sec  9.74901
#> 16 2019-09-18 16:09:44  2 min 20.928 sec  25412 obs/sec 10.03421
#>    iterations        samples training_rmse training_logloss training_r2
#> 1           0       0.000000            NA               NA          NA
#> 2           1   99900.000000       0.42461          0.55871     0.91016
#> 3           4  400265.000000       0.36386          0.41433     0.93402
#> 4           6  600375.000000       0.33460          0.35497     0.94421
#> 5           8  799786.000000       0.31826          0.32369     0.94953
#> 6          10 1000119.000000       0.30627          0.30096     0.95326
#> 7          13 1301430.000000       0.29274          0.27498     0.95729
#> 8          16 1601446.000000       0.28144          0.25803     0.96053
#> 9          18 1801600.000000       0.27123          0.24053     0.96334
#> 10         21 2102464.000000       0.26230          0.22390     0.96572
#> 11         23 2302125.000000       0.25662          0.21526     0.96718
#> 12         25 2502050.000000       0.25657          0.21498     0.96720
#> 13         28 2801846.000000       0.24881          0.20293     0.96915
#> 14         31 3102356.000000       0.24224          0.19232     0.97076
#> 15         34 3402549.000000       0.24020          0.18927     0.97125
#> 16         35 3502089.000000       0.23962          0.18680     0.97139
#>    training_classification_error validation_rmse validation_logloss
#> 1                             NA              NA                NA
#> 2                        0.24022         0.42163           0.55128
#> 3                        0.17659         0.37069           0.42487
#> 4                        0.15078         0.33673           0.36105
#> 5                        0.13487         0.32572           0.34029
```

```
#> 6                       0.12786         0.31685           0.32451
#> 7                       0.11556         0.30173           0.29507
#> 8                       0.10665         0.28967           0.27707
#> 9                       0.09835         0.28672           0.27253
#> 10                      0.09285         0.27291           0.24511
#> 11                      0.08754         0.26546           0.23595
#> 12                      0.08884         0.26914           0.24172
#> 13                      0.08434         0.25941           0.22542
#> 14                      0.07964         0.25119           0.21287
#> 15                      0.07764         0.25087           0.21213
#> 16                      0.07944         0.24821           0.20775
#>    validation_r2 validation_classification_error
#> 1             NA                              NA
#> 2        0.90783                         0.23755
#> 3        0.92876                         0.18557
#> 4        0.94121                         0.15175
#> 5        0.94499                         0.14128
#> 6        0.94795                         0.13738
#> 7        0.95280                         0.12132
#> 8        0.95649                         0.11254
#> 9        0.95738                         0.11055
#> 10       0.96138                         0.10157
#> 11       0.96346                         0.09339
#> 12       0.96244                         0.09768
#> 13       0.96511                         0.09099
#> 14       0.96729                         0.08321
#> 15       0.96737                         0.08560
#> 16       0.96806                         0.08371
#>
#> Variable Importances: (Extract with `h2o.varimp`)
#> =================================================
#>
#> Variable Importances:
#>                            variable relative_importance scaled_importance
#> 1                         Elevation            1.000000          1.000000
#> 2 Horizontal_Distance_To_Fire_Points            0.962107          0.962107
#> 3    Horizontal_Distance_To_Roadways            0.955372          0.955372
#> 4             Wilderness_Area.area_0            0.628632          0.628632
#> 5   Horizontal_Distance_To_Hydrology            0.589965          0.589965
#>   percentage
#> 1   0.047713
#> 2   0.045905
#> 3   0.045583
#> 4   0.029994
#> 5   0.028149
#>
#> ---
#>                      variable relative_importance scaled_importance
#> 51           Soil_Type.type_13            0.170216          0.170216
#> 52            Soil_Type.type_6            0.158149          0.158149
#> 53           Soil_Type.type_14            0.148051          0.148051
#> 54           Soil_Type.type_35            0.147837          0.147837
#> 55       Soil_Type.missing(NA)            0.000000          0.000000
```

```
#> 56 Wilderness_Area.missing(NA)            0.000000          0.000000
#>    percentage
#> 51   0.008121
#> 52   0.007546
#> 53   0.007064
#> 54   0.007054
#> 55   0.000000
#> 56   0.000000
```

Let's compare the training error with the validation and test set errors

```
h2o.performance(m3, train=T)          ## sampled training data (from model building)
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on temporary training frame with 9995 samples **
#>
#> Training Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.0574
#> RMSE: (Extract with `h2o.rmse`) 0.24
#> Logloss: (Extract with `h2o.logloss`) 0.187
#> Mean Per-Class Error: 0.128
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`
#> ========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    3231     367       0       0       2       2      13 0.1062
#> class_2     192    4632      10       0      10      13       3 0.0469
#> class_3       0      13     571       8       2      25       0 0.0775
#> class_4       0       0       4      35       0       4       0 0.1860
#> class_5       2      31       3       0     127       0       0 0.2209
#> class_6       1      15      36       1       0     296       0 0.1519
#> class_7      30       7       0       0       0       0     309 0.1069
#> Totals     3456    5065     624      44     141     340     325 0.0794
#>                  Rate
#> class_1 = 384 / 3,615
#> class_2 = 228 / 4,860
#> class_3 =    48 / 619
#> class_4 =      8 / 43
#> class_5 =    36 / 163
#> class_6 =    53 / 349
#> class_7 =    37 / 346
#> Totals  = 794 / 9,995
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.920560
#> 2 2  0.996498
#> 3 3  0.999700
#> 4 4  1.000000
#> 5 5  1.000000
```

```
#> 6 6  1.000000
#> 7 7  1.000000
h2o.performance(m3, valid=T)          ## sampled validation data (from model building)
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 10023 samples **
#>
#> Validation Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.0616
#> RMSE: (Extract with `h2o.rmse`) 0.248
#> Logloss: (Extract with `h2o.logloss`) 0.208
#> Mean Per-Class Error: 0.161
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid = TRUE)`
#> ========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    3300     400       0       0       4       0      25 0.1150
#> class_2     168    4628       8       0      13      16       3 0.0430
#> class_3       0      19     561       6       1      26       0 0.0848
#> class_4       0       0      10      30       0       2       0 0.2857
#> class_5       3      45       4       0     109       0       0 0.3230
#> class_6       0      18      39       2       0     237       0 0.1993
#> class_7      22       5       0       0       0       0     319 0.0780
#> Totals     3493    5115     622      38     127     281     347 0.0837
#>                   Rate
#> class_1 =  429 / 3,729
#> class_2 =  208 / 4,836
#> class_3 =     52 / 613
#> class_4 =      12 / 42
#> class_5 =     52 / 161
#> class_6 =     59 / 296
#> class_7 =     27 / 346
#> Totals  = 839 / 10,023
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.916293
#> 2 2  0.995909
#> 3 3  0.999601
#> 4 4  1.000000
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
h2o.performance(m3, newdata=train)    ## full training data
#> H2OMultinomialMetrics: deeplearning
#>
#> Test Set Metrics:
#> =====================
#>
```

```
#> MSE: (Extract with `h2o.mse`) 0.0558
#> RMSE: (Extract with `h2o.rmse`) 0.236
#> Logloss: (Extract with `h2o.logloss`) 0.184
#> Mean Per-Class Error: 0.13
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)`)
#> =========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1  113843   12621       1       0     100      21     534 0.1044
#> class_2    5569  163436     393       3     440     426      75 0.0405
#> class_3      11     511   19911     126      36     847       0 0.0714
#> class_4       0       0     258    1330       0      70       0 0.1978
#> class_5      59    1289      81       0    4269      22       0 0.2537
#> class_6      36     338    1245      63       7    8744       0 0.1619
#> class_7     880     131       0       0       1       0   11288 0.0823
#> Totals   120398  178326   21889    1522    4853   10130   11897 0.0751
#>                     Rate
#> class_1 = 13,277 / 127,120
#> class_2 =  6,906 / 170,342
#> class_3 =   1,531 / 21,442
#> class_4 =      328 / 1,658
#> class_5 =    1,451 / 5,720
#> class_6 =   1,689 / 10,433
#> class_7 =   1,012 / 12,300
#> Totals  = 26,194 / 349,015
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.924949
#> 2 2  0.996814
#> 3 3  0.999762
#> 4 4  0.999968
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
h2o.performance(m3, newdata=valid)    ## full validation data
#> H2OMultinomialMetrics: deeplearning
#>
#> Test Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.0614
#> RMSE: (Extract with `h2o.rmse`) 0.248
#> Logloss: (Extract with `h2o.logloss`) 0.203
#> Mean Per-Class Error: 0.146
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)`)
#> =========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1   37738    4523       0       0      33       3     203 0.1120
#> class_2    2063   53774     146       1     191     174      31 0.0462
```

```
#> class_3       2     203    6571      50      12     305       0 0.0801
#> class_4       0       0      94     436       0      32       0 0.2242
#> class_5      33     465      40       0    1327       5       0 0.2904
#> class_6       5     155     422      22       4    2856       0 0.1755
#> class_7     336      39       0       0       1       0    3723 0.0917
#> Totals    40177   59159    7273     509    1568    3375    3957 0.0827
#>                     Rate
#> class_1 =  4,762 / 42,500
#> class_2 =  2,606 / 56,380
#> class_3 =     572 / 7,143
#> class_4 =       126 / 562
#> class_5 =     543 / 1,870
#> class_6 =     608 / 3,464
#> class_7 =     376 / 4,099
#> Totals  = 9,593 / 116,018
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.917315
#> 2 2  0.995966
#> 3 3  0.999707
#> 4 4  0.999965
#> 5 5  0.999991
#> 6 6  1.000000
#> 7 7  1.000000
h2o.performance(m3, newdata=test)     ## full test data
#> H2OMultinomialMetrics: deeplearning
#>
#> Test Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.0612
#> RMSE: (Extract with `h2o.rmse`) 0.247
#> Logloss: (Extract with `h2o.logloss`) 0.202
#> Mean Per-Class Error: 0.145
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)`)
#> =========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1   37519    4459       0       0      33       6     203 0.1113
#> class_2    2136   53937     140       0     165     173      28 0.0467
#> class_3       2     209    6601      61      16     280       0 0.0792
#> class_4       0       0      99     410       0      18       0 0.2220
#> class_5      25     460      25       0    1381      12       0 0.2743
#> class_6      26     131     454      25       5    2829       0 0.1847
#> class_7     354      43       0       0       0       0    3714 0.0966
#> Totals    40062   59239    7319     496    1600    3318    3945 0.0827
#>                     Rate
#> class_1 =  4,701 / 42,220
#> class_2 =  2,642 / 56,579
#> class_3 =     568 / 7,169
```

```
#> class_4 =       117 / 527
#> class_5 =     522 / 1,903
#> class_6 =     641 / 3,470
#> class_7 =     397 / 4,111
#> Totals  = 9,588 / 115,979
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.917330
#> 2 2  0.996068
#> 3 3  0.999724
#> 4 4  0.999948
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
```

To confirm that the reported confusion matrix on the validation set (here, the test set) was correct, we make a prediction on the test set and compare the confusion matrices explicitly:

```
pred <- h2o.predict(m3, test)
#>
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
pred
#>   predict  class_1 class_2  class_3  class_4  class_5  class_6  class_7
#> 1 class_2 1.59e-01 0.84040 7.73e-04 3.68e-05 4.52e-05 8.77e-05 3.38e-05
#> 2 class_1 1.00e+00 0.00045 2.91e-07 2.38e-06 2.82e-08 2.94e-07 4.82e-07
#> 3 class_1 8.99e-01 0.10147 4.57e-07 3.85e-07 3.55e-09 7.77e-08 1.57e-07
#> 4 class_1 9.93e-01 0.00703 1.60e-06 4.74e-08 4.90e-08 1.58e-08 5.77e-06
#> 5 class_2 1.22e-02 0.98737 3.25e-05 4.04e-06 2.68e-04 8.21e-05 7.75e-07
#> 6 class_5 3.06e-05 0.22890 2.46e-09 1.96e-06 7.71e-01 1.64e-08 4.48e-08
#>
#> [115979 rows x 8 columns]
test$Accuracy <- pred$predict == test$Cover_Type
1-mean(test$Accuracy)
#> [1] 0.0827
```

5.5.6 Hyper-parameter Tuning with Grid Search

Since there are a lot of parameters that can impact model accuracy, hyper-parameter tuning is especially important for Deep Learning:

For speed, we will only train on the first 10,000 rows of the training dataset:

```
sampled_train=train[1:10000,]
```

The simplest hyperparameter search method is a brute-force scan of the full Cartesian product of all combinations specified by a grid search:

```
hyper_params <- list(
  hidden=list(c(32,32,32),c(64,64)),
```

```
  input_dropout_ratio=c(0,0.05),
  rate=c(0.01,0.02),
  rate_annealing=c(1e-8,1e-7,1e-6)
)
hyper_params
#> $hidden
#> $hidden[[1]]
#> [1] 32 32 32
#>
#> $hidden[[2]]
#> [1] 64 64
#>
#>
#> $input_dropout_ratio
#> [1] 0.00 0.05
#>
#> $rate
#> [1] 0.01 0.02
#>
#> $rate_annealing
#> [1] 1e-08 1e-07 1e-06
grid <- h2o.grid(
  algorithm="deeplearning",
  grid_id="dl_grid",
  training_frame=sampled_train,
  validation_frame=valid,
  x=predictors,
  y=response,
  epochs=10,
  stopping_metric="misclassification",
  stopping_tolerance=1e-2,        ## stop when misclassification does not improve by >=1% for 2 scoring
  stopping_rounds=2,
  score_validation_samples=10000, ## downsample validation set for faster scoring
  score_duty_cycle=0.025,         ## don't score more than 2.5% of the wall time
  adaptive_rate=F,                ## manually tuned learning rate
  momentum_start=0.5,             ## manually tuned momentum
  momentum_stable=0.9,
  momentum_ramp=1e7,
  l1=1e-5,
  l2=1e-5,
  activation=c("Rectifier"),
  max_w2=10,                      ## can help improve stability for Rectifier
  hyper_params=hyper_params
)
#>
  |
  |                                                                      |   0%
  |
  |===                                                                   |   4%
  |
  |=====                                                                 |   7%
  |
  |=======                                                               |  11%
```

```
  |
  |=========                                                      |  15%
  |
  |============                                                   |  18%
  |
  |===============                                                |  23%
  |
  |==================                                             |  28%
  |
  |======================                                         |  32%
  |
  |=======================                                        |  36%
  |
  |=========================                                      |  38%
  |
  |============================                                   |  41%
  |
  |==============================                                 |  44%
  |
  |================================                               |  47%
  |
  |==================================                             |  49%
  |
  |=====================================                          |  54%
  |
  |=======================================                        |  56%
  |
  |=========================================                      |  62%
  |
  |============================================                   |  66%
  |
  |================================================               |  70%
  |
  |==================================================             |  74%
  |
  |=====================================================          |  78%
  |
  |=======================================================        |  81%
  |
  |==========================================================     |  85%
  |
  |============================================================   |  90%
  |
  |================================================================   |  95%
  |
  |====================================================================| 99%
  |
  |======================================================================| 100%
grid
#> H2O Grid Details
#> ================
#>
#> Grid ID: dl_grid
```

```
#> Used hyper parameters:
#>   -  hidden
#>   -  input_dropout_ratio
#>   -  rate
#>   -  rate_annealing
#> Number of models: 24
#> Number of failed models: 0
#>
#> Hyper-Parameter Search Summary: ordered by increasing logloss
#>     hidden input_dropout_ratio rate rate_annealing        model_ids
#> 1 [64, 64]                 0.0 0.01         1.0E-8  dl_grid_model_2
#> 2 [64, 64]                 0.0 0.01         1.0E-7 dl_grid_model_10
#> 3 [64, 64]                0.05 0.01         1.0E-7 dl_grid_model_12
#> 4 [64, 64]                 0.0 0.01         1.0E-6 dl_grid_model_18
#> 5 [64, 64]                 0.0 0.02         1.0E-7 dl_grid_model_14
#>              logloss
#> 1  0.554375812035957
#> 2 0.5615972989064305
#> 3 0.5774132175393003
#> 4 0.5795132556056228
#> 5 0.5836529554680269
#>
#> ---
#>          hidden input_dropout_ratio rate rate_annealing        model_ids
#> 19 [32, 32, 32]                0.0 0.02         1.0E-6 dl_grid_model_21
#> 20 [32, 32, 32]               0.05 0.01         1.0E-8  dl_grid_model_3
#> 21 [32, 32, 32]               0.05 0.02         1.0E-7 dl_grid_model_15
#> 22 [32, 32, 32]               0.05 0.02         1.0E-6 dl_grid_model_23
#> 23 [32, 32, 32]                0.0 0.02         1.0E-7 dl_grid_model_13
#> 24 [32, 32, 32]               0.05 0.02         1.0E-8  dl_grid_model_7
#>               logloss
#> 19 0.6332954204607399
#> 20 0.6333300727746808
#> 21 0.6383509237900457
#> 22 0.6399458760574194
#> 23  0.642640090400582
#> 24 0.6454225103454274
```

Let's see which model had the lowest validation error:

```
grid <- h2o.getGrid("dl_grid",sort_by="err",decreasing=FALSE)
grid
#> H2O Grid Details
#> ================
#>
#> Grid ID: dl_grid
#> Used hyper parameters:
#>   -  hidden
#>   -  input_dropout_ratio
#>   -  rate
#>   -  rate_annealing
#> Number of models: 24
#> Number of failed models: 0
#>
```

```
#> Hyper-Parameter Search Summary: ordered by increasing err
#>         hidden input_dropout_ratio rate rate_annealing        model_ids
#> 1     [64, 64]                 0.0 0.01         1.0E-8  dl_grid_model_2
#> 2     [64, 64]                 0.0 0.01         1.0E-7 dl_grid_model_10
#> 3     [64, 64]                 0.0 0.01         1.0E-6 dl_grid_model_18
#> 4     [64, 64]                 0.0 0.02         1.0E-6 dl_grid_model_22
#> 5 [32, 32, 32]                0.05 0.01         1.0E-6 dl_grid_model_19
#>                    err
#> 1 0.24254592912358197
#> 2  0.2509792106056041
#> 3 0.25175175175175174
#> 4  0.2526158445440957
#> 5   0.252856433184302
#>
#> ---
#>          hidden input_dropout_ratio rate rate_annealing        model_ids
#> 19 [32, 32, 32]                 0.0 0.02         1.0E-7 dl_grid_model_13
#> 20 [32, 32, 32]                 0.0 0.02         1.0E-8  dl_grid_model_5
#> 21     [64, 64]                0.05 0.01         1.0E-6 dl_grid_model_20
#> 22 [32, 32, 32]                0.05 0.01         1.0E-7 dl_grid_model_11
#> 23     [64, 64]                0.05 0.02         1.0E-7 dl_grid_model_16
#> 24 [32, 32, 32]                0.05 0.02         1.0E-8  dl_grid_model_7
#>                    err
#> 19 0.2667524497673958
#> 20 0.2668004012036108
#> 21 0.2683467741935484
#> 22 0.2696741854636591
#> 23 0.2705015923566879
#> 24 0.2841873938667466

## To see what other "sort_by" criteria are allowed
#grid <- h2o.getGrid("dl_grid",sort_by="wrong_thing",decreasing=FALSE)

## Sort by logloss
h2o.getGrid("dl_grid",sort_by="logloss",decreasing=FALSE)
#> H2O Grid Details
#> ================
#>
#> Grid ID: dl_grid
#> Used hyper parameters:
#>   -  hidden
#>   -  input_dropout_ratio
#>   -  rate
#>   -  rate_annealing
#> Number of models: 24
#> Number of failed models: 0
#>
#> Hyper-Parameter Search Summary: ordered by increasing logloss
#>     hidden input_dropout_ratio rate rate_annealing        model_ids
#> 1 [64, 64]                 0.0 0.01         1.0E-8  dl_grid_model_2
#> 2 [64, 64]                 0.0 0.01         1.0E-7 dl_grid_model_10
#> 3 [64, 64]                0.05 0.01         1.0E-7 dl_grid_model_12
#> 4 [64, 64]                 0.0 0.01         1.0E-6 dl_grid_model_18
```

```
#> 5 [64, 64]                 0.0 0.02         1.0E-7 dl_grid_model_14
#>              logloss
#> 1  0.554375812035957
#> 2 0.5615972989064305
#> 3 0.5774132175393003
#> 4 0.5795132556056228
#> 5 0.5836529554680269
#>
#> ---
#>          hidden input_dropout_ratio rate rate_annealing        model_ids
#> 19 [32, 32, 32]                 0.0 0.02         1.0E-6 dl_grid_model_21
#> 20 [32, 32, 32]                0.05 0.01         1.0E-8  dl_grid_model_3
#> 21 [32, 32, 32]                0.05 0.02         1.0E-7 dl_grid_model_15
#> 22 [32, 32, 32]                0.05 0.02         1.0E-6 dl_grid_model_23
#> 23 [32, 32, 32]                 0.0 0.02         1.0E-7 dl_grid_model_13
#> 24 [32, 32, 32]                0.05 0.02         1.0E-8  dl_grid_model_7
#>               logloss
#> 19 0.6332954204607399
#> 20 0.6333300727746808
#> 21 0.6383509237900457
#> 22 0.6399458760574194
#> 23  0.642640090400582
#> 24 0.6454225103454274

## Find the best model and its full set of parameters
grid@summary_table[1,]
#> Hyper-Parameter Search Summary: ordered by increasing err
#>     hidden input_dropout_ratio rate rate_annealing       model_ids
#> 1 [64, 64]                 0.0 0.01         1.0E-8 dl_grid_model_2
#>                   err
#> 1 0.24254592912358197
best_model <- h2o.getModel(grid@model_ids[[1]])
best_model
#> Model Details:
#> ==============
#>
#> H2OMultinomialModel: deeplearning
#> Model ID:  dl_grid_model_2
#> Status of Neuron Layers: predicting Cover_Type, 7-class classification, multinomial distribution, Cr
#>   layer units      type dropout       l1       l2 mean_rate rate_rms
#> 1     1    56     Input  0.00 %       NA       NA        NA       NA
#> 2     2    64 Rectifier  0.00 % 0.000010 0.000010  0.009990 0.000000
#> 3     3    64 Rectifier  0.00 % 0.000010 0.000010  0.009990 0.000000
#> 4     4     7   Softmax      NA 0.000010 0.000010  0.009990 0.000000
#>   momentum mean_weight weight_rms mean_bias bias_rms
#> 1       NA          NA         NA        NA       NA
#> 2 0.504000   -0.008767   0.215177  0.132772 0.155228
#> 3 0.504000   -0.057464   0.188338  0.852400 0.168772
#> 4 0.504000    0.007717   0.391073  0.013818 0.573407
#>
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on training data. **
```

```
#> ** Metrics reported on full training frame **
#>
#> Training Set Metrics:
#> =====================
#>
#> Extract training frame with `h2o.getFrame("RTMP_sid_8ba3_9")`
#> MSE: (Extract with `h2o.mse`) 0.16
#> RMSE: (Extract with `h2o.rmse`) 0.4
#> Logloss: (Extract with `h2o.logloss`) 0.495
#> Mean Per-Class Error: 0.403
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
#> =========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    2505    1123       1       0       0       2      57 0.3208
#> class_2     377    4397      33       0       7      20       1 0.0906
#> class_3       0      51     469       7       5      98       0 0.2556
#> class_4       0       0      18      22       0       4       0 0.5000
#> class_5       7     121       1       0      27       0       0 0.8269
#> class_6       0      79      88       0       0     142       0 0.5405
#> class_7      91       5       0       0       0       0     242 0.2840
#> Totals     2980    5776     610      29      39     266     300 0.2196
#>                      Rate
#> class_1 =  1,183 / 3,688
#> class_2 =    438 / 4,835
#> class_3 =      161 / 630
#> class_4 =        22 / 44
#> class_5 =      129 / 156
#> class_6 =      167 / 309
#> class_7 =       96 / 338
#> Totals  = 2,196 / 10,000
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.780400
#> 2 2  0.982500
#> 3 3  0.997800
#> 4 4  0.999500
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
#>
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 9961 samples **
#>
#> Validation Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.178
```

```
#> RMSE: (Extract with `h2o.rmse`) 0.422
#> Logloss: (Extract with `h2o.logloss`) 0.554
#> Mean Per-Class Error: 0.43
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid = TRUE)`)
#> =========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    2375    1217       0       0       0       1      46 0.3473
#> class_2     459    4289      36       1       4      30       3 0.1105
#> class_3       0      55     426      20       4     106       0 0.3028
#> class_4       0       0      21      22       0       5       0 0.5417
#> class_5      11     124       1       0      18       1       0 0.8839
#> class_6       0      89      74       2       2     135       0 0.5530
#> class_7     103       1       0       0       0       0     280 0.2708
#> Totals     2948    5775     558      45      28     278     329 0.2425
#>                     Rate
#> class_1 = 1,264 / 3,639
#> class_2 =   533 / 4,822
#> class_3 =     185 / 611
#> class_4 =       26 / 48
#> class_5 =     137 / 155
#> class_6 =     167 / 302
#> class_7 =     104 / 384
#> Totals  = 2,416 / 9,961
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.757454
#> 2 2  0.974701
#> 3 3  0.996587
#> 4 4  0.999297
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000

print(best_model@allparameters)
#> $model_id
#> [1] "dl_grid_model_2"
#>
#> $training_frame
#> [1] "RTMP_sid_8ba3_9"
#>
#> $validation_frame
#> [1] "valid.hex"
#>
#> $nfolds
#> [1] 0
#>
#> $keep_cross_validation_models
#> [1] TRUE
#>
```

```
#> $keep_cross_validation_predictions
#> [1] FALSE
#>
#> $keep_cross_validation_fold_assignment
#> [1] FALSE
#>
#> $fold_assignment
#> [1] "AUTO"
#>
#> $ignore_const_cols
#> [1] TRUE
#>
#> $score_each_iteration
#> [1] FALSE
#>
#> $balance_classes
#> [1] FALSE
#>
#> $max_after_balance_size
#> [1] 5
#>
#> $max_confusion_matrix_size
#> [1] 20
#>
#> $max_hit_ratio_k
#> [1] 0
#>
#> $overwrite_with_best_model
#> [1] TRUE
#>
#> $use_all_factor_levels
#> [1] TRUE
#>
#> $standardize
#> [1] TRUE
#>
#> $activation
#> [1] "Rectifier"
#>
#> $hidden
#> [1] 64 64
#>
#> $epochs
#> [1] 10
#>
#> $train_samples_per_iteration
#> [1] -2
#>
#> $target_ratio_comm_to_comp
#> [1] 0.05
#>
#> $seed
#> [1] -3.09e+17
```

```
#>
#> $adaptive_rate
#> [1] FALSE
#>
#> $rho
#> [1] 0.99
#>
#> $epsilon
#> [1] 1e-08
#>
#> $rate
#> [1] 0.01
#>
#> $rate_annealing
#> [1] 1e-08
#>
#> $rate_decay
#> [1] 1
#>
#> $momentum_start
#> [1] 0.5
#>
#> $momentum_ramp
#> [1] 1e+07
#>
#> $momentum_stable
#> [1] 0.9
#>
#> $nesterov_accelerated_gradient
#> [1] TRUE
#>
#> $input_dropout_ratio
#> [1] 0
#>
#> $l1
#> [1] 1e-05
#>
#> $l2
#> [1] 1e-05
#>
#> $max_w2
#> [1] 10
#>
#> $initial_weight_distribution
#> [1] "UniformAdaptive"
#>
#> $initial_weight_scale
#> [1] 1
#>
#> $loss
#> [1] "Automatic"
#>
#> $distribution
```

```
#> [1] "AUTO"
#>
#> $quantile_alpha
#> [1] 0.5
#>
#> $tweedie_power
#> [1] 1.5
#>
#> $huber_alpha
#> [1] 0.9
#>
#> $score_interval
#> [1] 5
#>
#> $score_training_samples
#> [1] 10000
#>
#> $score_validation_samples
#> [1] 10000
#>
#> $score_duty_cycle
#> [1] 0.025
#>
#> $classification_stop
#> [1] 0
#>
#> $regression_stop
#> [1] 1e-06
#>
#> $stopping_rounds
#> [1] 2
#>
#> $stopping_metric
#> [1] "misclassification"
#>
#> $stopping_tolerance
#> [1] 0.01
#>
#> $max_runtime_secs
#> [1] 1.8e+308
#>
#> $score_validation_sampling
#> [1] "Uniform"
#>
#> $diagnostics
#> [1] TRUE
#>
#> $fast_mode
#> [1] TRUE
#>
#> $force_load_balance
#> [1] TRUE
#>
```

```
#> $variable_importances
#> [1] TRUE
#>
#> $replicate_training_data
#> [1] TRUE
#>
#> $single_node_mode
#> [1] FALSE
#>
#> $shuffle_training_data
#> [1] FALSE
#>
#> $missing_values_handling
#> [1] "MeanImputation"
#>
#> $quiet_mode
#> [1] FALSE
#>
#> $autoencoder
#> [1] FALSE
#>
#> $sparse
#> [1] FALSE
#>
#> $col_major
#> [1] FALSE
#>
#> $average_activation
#> [1] 0
#>
#> $sparsity_beta
#> [1] 0
#>
#> $max_categorical_features
#> [1] 2147483647
#>
#> $reproducible
#> [1] FALSE
#>
#> $export_weights_and_biases
#> [1] FALSE
#>
#> $mini_batch_size
#> [1] 1
#>
#> $categorical_encoding
#> [1] "AUTO"
#>
#> $elastic_averaging
#> [1] FALSE
#>
#> $elastic_averaging_moving_rate
#> [1] 0.9
```

```
#>
#> $elastic_averaging_regularization
#> [1] 0.001
#>
#> $x
#>  [1] "Soil_Type"
#>  [2] "Wilderness_Area"
#>  [3] "Elevation"
#>  [4] "Aspect"
#>  [5] "Slope"
#>  [6] "Horizontal_Distance_To_Hydrology"
#>  [7] "Vertical_Distance_To_Hydrology"
#>  [8] "Horizontal_Distance_To_Roadways"
#>  [9] "Hillshade_9am"
#> [10] "Hillshade_Noon"
#> [11] "Hillshade_3pm"
#> [12] "Horizontal_Distance_To_Fire_Points"
#>
#> $y
#> [1] "Cover_Type"
print(h2o.performance(best_model, valid=T))
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 9961 samples **
#>
#> Validation Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.178
#> RMSE: (Extract with `h2o.rmse`) 0.422
#> Logloss: (Extract with `h2o.logloss`) 0.554
#> Mean Per-Class Error: 0.43
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid = TRUE)`)
#> =========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>          class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    2375    1217       0       0       0       1      46 0.3473
#> class_2     459    4289      36       1       4      30       3 0.1105
#> class_3       0      55     426      20       4     106       0 0.3028
#> class_4       0       0      21      22       0       5       0 0.5417
#> class_5      11     124       1       0      18       1       0 0.8839
#> class_6       0      89      74       2       2     135       0 0.5530
#> class_7     103       1       0       0       0       0     280 0.2708
#> Totals     2948    5775     558      45      28     278     329 0.2425
#>                     Rate
#> class_1 = 1,264 / 3,639
#> class_2 =   533 / 4,822
#> class_3 =     185 / 611
#> class_4 =       26 / 48
#> class_5 =     137 / 155
#> class_6 =     167 / 302
#> class_7 =     104 / 384
#> Totals  = 2,416 / 9,961
```

```
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.757454
#> 2 2  0.974701
#> 3 3  0.996587
#> 4 4  0.999297
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
print(h2o.logloss(best_model, valid=T))
#> [1] 0.554
```

5.5.7 Random Hyper-Parameter Search

Often, hyper-parameter search for more than 4 parameters can be done more efficiently with random parameter search than with grid search. Basically, chances are good to find one of many good models in less time than performing an exhaustive grid search. We simply build up to max_models models with parameters drawn randomly from user-specified distributions (here, uniform). For this example, we use the adaptive learning rate and focus on tuning the network architecture and the regularization parameters. We also let the grid search stop automatically once the performance at the top of the leaderboard doesn't change much anymore, i.e., once the search has converged.

```
hyper_params <- list(
  activation=c("Rectifier","Tanh","Maxout","RectifierWithDropout","TanhWithDropout","MaxoutWithDropout"
  hidden=list(c(20,20),c(50,50),c(30,30,30),c(25,25,25,25)),
  input_dropout_ratio=c(0,0.05),
  l1=seq(0,1e-4,1e-6),
  l2=seq(0,1e-4,1e-6)
)
hyper_params

## Stop once the top 5 models are within 1% of each other (i.e., the windowed average varies less than
search_criteria = list(strategy = "RandomDiscrete", max_runtime_secs = 360, max_models = 100, seed=1234
dl_random_grid <- h2o.grid(
  algorithm="deeplearning",
  grid_id = "dl_grid_random",
  training_frame=sampled_train,
  validation_frame=valid,
  x=predictors,
  y=response,
  epochs=1,
  stopping_metric="logloss",
  stopping_tolerance=1e-2,        ## stop when logloss does not improve by >=1% for 2 scoring events
  stopping_rounds=2,
  score_validation_samples=10000, ## downsample validation set for faster scoring
  score_duty_cycle=0.025,         ## don't score more than 2.5% of the wall time
  max_w2=10,                      ## can help improve stability for Rectifier
  hyper_params = hyper_params,
  search_criteria = search_criteria
)
```

```
grid <- h2o.getGrid("dl_grid_random",sort_by="logloss",decreasing=FALSE)
grid

grid@summary_table[1,]
best_model <- h2o.getModel(grid@model_ids[[1]]) ## model with lowest logloss
best_model
```

Let's look at the model with the lowest validation misclassification rate:

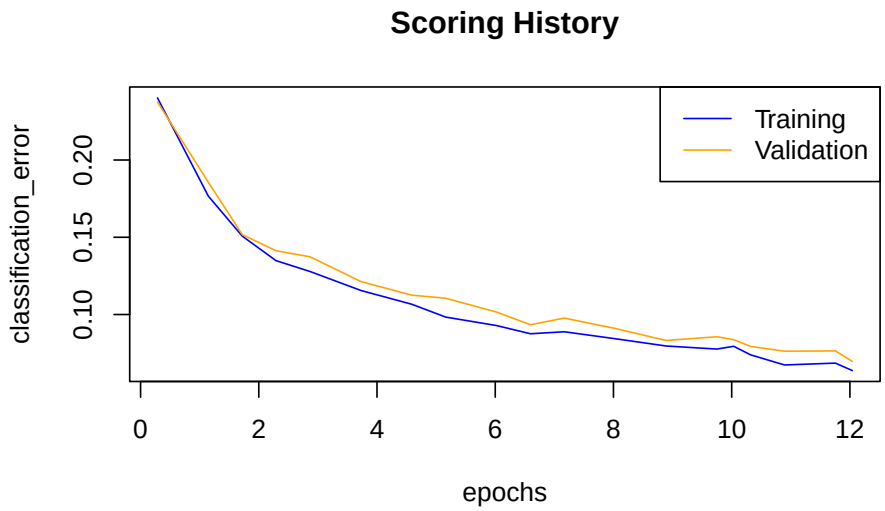
```
grid <- h2o.getGrid("dl_grid",sort_by="err",decreasing=FALSE)
best_model <- h2o.getModel(grid@model_ids[[1]]) ## model with lowest classification error (on validatio
h2o.confusionMatrix(best_model,valid=T)
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    2375    1217       0       0       0       1      46 0.3473
#> class_2     459    4289      36       1       4      30       3 0.1105
#> class_3       0      55     426      20       4     106       0 0.3028
#> class_4       0       0      21      22       0       5       0 0.5417
#> class_5      11     124       1       0      18       1       0 0.8839
#> class_6       0      89      74       2       2     135       0 0.5530
#> class_7     103       1       0       0       0       0     280 0.2708
#> Totals     2948    5775     558      45      28     278     329 0.2425
#>                    Rate
#> class_1 = 1,264 / 3,639
#> class_2 =   533 / 4,822
#> class_3 =     185 / 611
#> class_4 =       26 / 48
#> class_5 =     137 / 155
#> class_6 =     167 / 302
#> class_7 =     104 / 384
#> Totals  = 2,416 / 9,961
best_params <- best_model@allparameters
best_params$activation
#> [1] "Rectifier"
best_params$hidden
#> [1] 64 64
best_params$input_dropout_ratio
#> [1] 0
best_params$l1
#> [1] 1e-05
best_params$l2
#> [1] 1e-05
```

5.5.8 Checkpointing

Let's continue training the manually tuned model from before, for 2 more epochs. Note that since many important parameters such as epochs, l1, l2, max_w2, score_interval, train_samples_per_iteration, input_dropout_ratio, hidden_dropout_ratios, score_duty_cycle, classification_stop, regression_stop, variable_importances, force_load_balance can be modified between checkpoint restarts, it is best to specify as many parameters as possible explicitly.

```
max_epochs <- 12 ## Add two more epochs
m_cont <- h2o.deeplearning(
  model_id="dl_model_tuned_continued",
```

```
  checkpoint="dl_model_tuned",
  training_frame=train,
  validation_frame=valid,
  x=predictors,
  y=response,
  hidden=c(128,128,128),          ## more hidden layers -> more complex interactions
  epochs=max_epochs,              ## hopefully long enough to converge (otherwise restart again)
  stopping_metric="logloss",      ## logloss is directly optimized by Deep Learning
  stopping_tolerance=1e-2,        ## stop when validation logloss does not improve by >=1% for 2 scorin
  stopping_rounds=2,
  score_validation_samples=10000, ## downsample validation set for faster scoring
  score_duty_cycle=0.025,         ## don't score more than 2.5% of the wall time
  adaptive_rate=F,                ## manually tuned learning rate
  rate=0.01,
  rate_annealing=2e-6,
  momentum_start=0.2,             ## manually tuned momentum
  momentum_stable=0.4,
  momentum_ramp=1e7,
  l1=1e-5,                        ## add some L1/L2 regularization
  l2=1e-5,
  max_w2=10                       ## helps stability for Rectifier
)
summary(m_cont)
plot(m_cont)
```



Once we are satisfied with the results, we can save the model to disk (on the cluster). In this example, we store the model in a directory called `mybest_deeplearning_covtype_model`, which will be created for us since force=TRUE.

```
path <- h2o.saveModel(m_cont,
          path = file.path(data_out_dir, "mybest_deeplearning_covtype_model"), force=TRUE)
```

It can be loaded later with the following command:

```
print(path)
#> [1] "/home/datascience/repos/machine-learning-rsuite/export/mybest_deeplearning_covtype_model/dl_mod
m_loaded <- h2o.loadModel(path)
summary(m_loaded)
#> Model Details:
```

```
#> ==============
#>
#> H2OMultinomialModel: deeplearning
#> Model Key:  dl_model_tuned_continued
#> Status of Neuron Layers: predicting Cover_Type, 7-class classification, multinomial distribution, Cr
#>   layer units      type dropout       l1       l2 mean_rate rate_rms
#> 1     1    56     Input  0.00 %       NA       NA         NA       NA
#> 2     2   128 Rectifier  0.00 % 0.000010 0.000010  0.001063 0.000000
#> 3     3   128 Rectifier  0.00 % 0.000010 0.000010  0.001063 0.000000
#> 4     4   128 Rectifier  0.00 % 0.000010 0.000010  0.001063 0.000000
#> 5     5     7   Softmax      NA 0.000010 0.000010  0.001063 0.000000
#>   momentum mean_weight weight_rms mean_bias bias_rms
#> 1       NA          NA         NA        NA       NA
#> 2 0.284047   -0.010841   0.324725  0.008575 0.320140
#> 3 0.284047   -0.055705   0.226045  0.869899 0.364565
#> 4 0.284047   -0.063183   0.221199  0.804981 0.194941
#> 5 0.284047   -0.022196   0.270699  0.003139 0.790763
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on temporary training frame with 9933 samples **
#>
#> Training Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.0482
#> RMSE: (Extract with `h2o.rmse`) 0.22
#> Logloss: (Extract with `h2o.logloss`) 0.162
#> Mean Per-Class Error: 0.113
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`
#> ========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>          class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    3344     262       0       0       2       1       9 0.0757
#> class_2     182    4666       9       0      15       7       0 0.0437
#> class_3       0       4     578       2       2      18       0 0.0430
#> class_4       0       0       8      52       0       2       0 0.1613
#> class_5       6      26       5       0     137       0       0 0.2126
#> class_6       2      13      27       3       0     221       0 0.1692
#> class_7      24       4       0       0       0       0     302 0.0848
#> Totals     3558    4975     627      57     156     249     311 0.0637
#>                   Rate
#> class_1 = 274 / 3,618
#> class_2 = 213 / 4,879
#> class_3 =    26 / 604
#> class_4 =     10 / 62
#> class_5 =    37 / 174
#> class_6 =    45 / 266
#> class_7 =    28 / 330
#> Totals  = 633 / 9,933
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
#> ========================================================================
```

```
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.936273
#> 2 2  0.997684
#> 3 3  0.999698
#> 4 4  1.000000
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
#>
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 10032 samples **
#>
#> Validation Set Metrics:
#> =====================
#>
#> MSE: (Extract with `h2o.mse`) 0.053
#> RMSE: (Extract with `h2o.rmse`) 0.23
#> Logloss: (Extract with `h2o.logloss`) 0.178
#> Mean Per-Class Error: 0.15
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid = TRUE)`
#> ========================================================================
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>         class_1 class_2 class_3 class_4 class_5 class_6 class_7  Error
#> class_1    3354     269       1       0       2       0       6 0.0765
#> class_2     189    4664      15       0      17      13       3 0.0484
#> class_3       0      18     596       4       0      25       0 0.0731
#> class_4       0       0      10      32       0       4       0 0.3043
#> class_5       6      37       2       0     108       0       0 0.2941
#> class_6       0      15      38       1       0     235       0 0.1869
#> class_7      22       2       0       0       0       0     344 0.0652
#> Totals     3571    5005     662      37     127     277     353 0.0697
#>                   Rate
#> class_1 =  278 / 3,632
#> class_2 =  237 / 4,901
#> class_3 =     47 / 643
#> class_4 =      14 / 46
#> class_5 =     45 / 153
#> class_6 =     54 / 289
#> class_7 =     24 / 368
#> Totals  = 699 / 10,032
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
#> =======================================================================
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.930323
#> 2 2  0.996312
#> 3 3  0.999801
#> 4 4  1.000000
#> 5 5  1.000000
```

```
#> 6 6  1.000000
#> 7 7  1.000000
#>
#>
#>
#>
#> Scoring History:
#>              timestamp          duration training_speed   epochs
#> 1  2019-09-18 16:07:23         0.000 sec             NA  0.00000
#> 2  2019-09-18 16:07:29         6.416 sec  16447 obs/sec  0.28623
#> 3  2019-09-18 16:07:42        19.613 sec  20981 obs/sec  1.14684
#> 4  2019-09-18 16:07:51        27.872 sec  22163 obs/sec  1.72020
#> 5  2019-09-18 16:08:02        38.887 sec  21091 obs/sec  2.29155
#> 6  2019-09-18 16:08:10        46.794 sec  21912 obs/sec  2.86555
#> 7  2019-09-18 16:08:20        57.297 sec  23276 obs/sec  3.72887
#> 8  2019-09-18 16:08:32  1 min  9.478 sec  23582 obs/sec  4.58847
#> 9  2019-09-18 16:08:40  1 min 17.269 sec  23866 obs/sec  5.16196
#> 10 2019-09-18 16:08:52  1 min 28.655 sec  24252 obs/sec  6.02399
#> 11 2019-09-18 16:08:59  1 min 36.499 sec  24402 obs/sec  6.59606
#> 12 2019-09-18 16:09:07  1 min 44.628 sec  24473 obs/sec  7.16889
#> 13 2019-09-18 16:09:19  1 min 55.714 sec  24767 obs/sec  8.02787
#> 14 2019-09-18 16:09:30  2 min  6.976 sec  24977 obs/sec  8.88889
#> 15 2019-09-18 16:09:40  2 min 17.375 sec  25311 obs/sec  9.74901
#> 16 2019-09-18 16:09:44  2 min 20.928 sec  25412 obs/sec 10.03421
#> 17 2019-09-18 16:11:02  2 min 24.360 sec  25543 obs/sec 10.32054
#> 18 2019-09-18 16:11:10  2 min 32.791 sec  25483 obs/sec 10.89361
#> 19 2019-09-18 16:11:23  2 min 45.257 sec  25404 obs/sec 11.75408
#> 20 2019-09-18 16:11:27  2 min 49.472 sec  25396 obs/sec 12.04065
#>    iterations        samples training_rmse training_logloss training_r2
#> 1           0       0.000000            NA               NA          NA
#> 2           1   99900.000000       0.42461          0.55871     0.91016
#> 3           4  400265.000000       0.36386          0.41433     0.93402
#> 4           6  600375.000000       0.33460          0.35497     0.94421
#> 5           8  799786.000000       0.31826          0.32369     0.94953
#> 6          10 1000119.000000       0.30627          0.30096     0.95326
#> 7          13 1301430.000000       0.29274          0.27498     0.95729
#> 8          16 1601446.000000       0.28144          0.25803     0.96053
#> 9          18 1801600.000000       0.27123          0.24053     0.96334
#> 10         21 2102464.000000       0.26230          0.22390     0.96572
#> 11         23 2302125.000000       0.25662          0.21526     0.96718
#> 12         25 2502050.000000       0.25657          0.21498     0.96720
#> 13         28 2801846.000000       0.24881          0.20293     0.96915
#> 14         31 3102356.000000       0.24224          0.19232     0.97076
#> 15         34 3402549.000000       0.24020          0.18927     0.97125
#> 16         35 3502089.000000       0.23962          0.18680     0.97139
#> 17         36 3602023.000000       0.23663          0.18390     0.96998
#> 18         38 3802034.000000       0.22964          0.17594     0.97173
#> 19         41 4102351.000000       0.22488          0.16809     0.97289
#> 20         42 4202367.000000       0.21963          0.16190     0.97414
#>    training_classification_error validation_rmse validation_logloss
#> 1                             NA              NA                NA
#> 2                        0.24022         0.42163           0.55128
#> 3                        0.17659         0.37069           0.42487
```

```
#> 4                       0.15078         0.33673           0.36105
#> 5                       0.13487         0.32572           0.34029
#> 6                       0.12786         0.31685           0.32451
#> 7                       0.11556         0.30173           0.29507
#> 8                       0.10665         0.28967           0.27707
#> 9                       0.09835         0.28672           0.27253
#> 10                      0.09285         0.27291           0.24511
#> 11                      0.08754         0.26546           0.23595
#> 12                      0.08884         0.26914           0.24172
#> 13                      0.08434         0.25941           0.22542
#> 14                      0.07964         0.25119           0.21287
#> 15                      0.07764         0.25087           0.21213
#> 16                      0.07944         0.24821           0.20775
#> 17                      0.07390         0.24246           0.19481
#> 18                      0.06735         0.23945           0.19052
#> 19                      0.06856         0.23635           0.18518
#> 20                      0.06373         0.23021           0.17798
#>    validation_r2 validation_classification_error
#> 1             NA                              NA
#> 2        0.90783                         0.23755
#> 3        0.92876                         0.18557
#> 4        0.94121                         0.15175
#> 5        0.94499                         0.14128
#> 6        0.94795                         0.13738
#> 7        0.95280                         0.12132
#> 8        0.95649                         0.11254
#> 9        0.95738                         0.11055
#> 10       0.96138                         0.10157
#> 11       0.96346                         0.09339
#> 12       0.96244                         0.09768
#> 13       0.96511                         0.09099
#> 14       0.96729                         0.08321
#> 15       0.96737                         0.08560
#> 16       0.96806                         0.08371
#> 17       0.96996                         0.07935
#> 18       0.97070                         0.07626
#> 19       0.97145                         0.07646
#> 20       0.97291                         0.06968
#>
#> Variable Importances: (Extract with `h2o.varimp`)
#> =================================================
#>
#> Variable Importances:
#>                            variable relative_importance scaled_importance
#> 1                         Elevation            1.000000          1.000000
#> 2 Horizontal_Distance_To_Fire_Points            0.965839          0.965839
#> 3    Horizontal_Distance_To_Roadways            0.960288          0.960288
#> 4             Wilderness_Area.area_0            0.628117          0.628117
#> 5   Horizontal_Distance_To_Hydrology            0.592000          0.592000
#>   percentage
#> 1   0.047730
#> 2   0.046100
#> 3   0.045835
```

```
#> 4   0.029980
#> 5   0.028256
#>
#> ---
#>                       variable relative_importance scaled_importance
#> 51           Soil_Type.type_13            0.165256          0.165256
#> 52            Soil_Type.type_6            0.154398          0.154398
#> 53           Soil_Type.type_14            0.144444          0.144444
#> 54           Soil_Type.type_35            0.144247          0.144247
#> 55       Soil_Type.missing(NA)            0.000000          0.000000
#> 56 Wilderness_Area.missing(NA)            0.000000          0.000000
#>    percentage
#> 51   0.007888
#> 52   0.007369
#> 53   0.006894
#> 54   0.006885
#> 55   0.000000
#> 56   0.000000
```

This model is fully functional and can be inspected, restarted, or used to score a dataset, etc. Note that binary compatibility between H2O versions is currently not guaranteed.

5.5.9 Cross-Validation

For N-fold cross-validation, specify `nfolds>1` instead of (or in addition to) a validation frame, and `N+1` models will be built: 1 model on the full training data, and N models with each 1/N-th of the data held out (there are different holdout strategies). Those N models then score on the held out data, and their combined predictions on the full training data are scored to get the cross-validation metrics.

```
dlmodel <- h2o.deeplearning(
  x=predictors,
  y=response,
  training_frame=train,
  hidden=c(10,10),
  epochs=1,
  nfolds=5,
  fold_assignment="Modulo" # can be "AUTO", "Modulo", "Random" or "Stratified"
  )
dlmodel
```

N-fold cross-validation is especially useful with early stopping, as the main model will pick the ideal number of epochs from the convergence behavior of the cross-validation models.

5.6 Regression and Binary Classification

Assume we want to turn the multi-class problem above into a binary classification problem. We create a binary response as follows:

```
train$bin_response <- ifelse(train[,response] == "class_1", 0, 1)
```

Let's build a quick model and inspect the model:

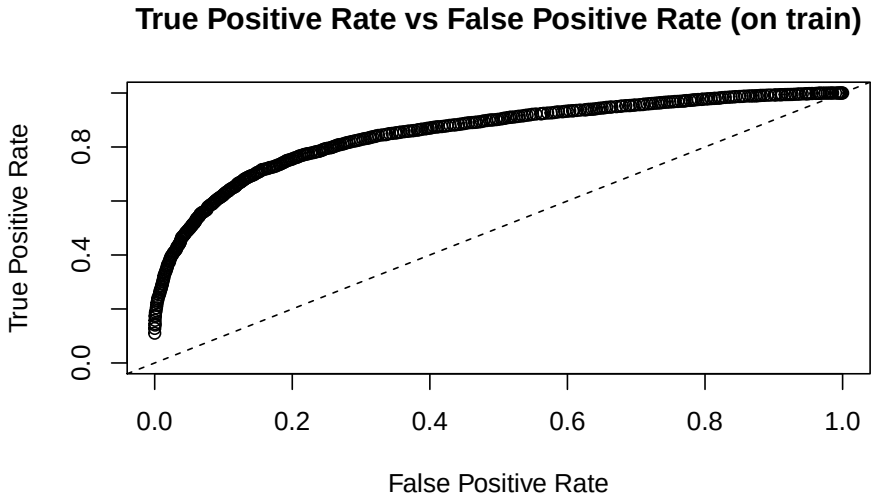
```
dlmodel <- h2o.deeplearning(
  x=predictors,
```

```
  y="bin_response",
  training_frame=train,
  hidden=c(10,10),
  epochs=0.1
)
summary(dlmodel)
```

Instead of a binary classification model, we find a regression model (H2ORegressionModel) that contains only 1 output neuron (instead of 2). The reason is that the response was a numerical feature (ordinal numbers 0 and 1), and H2O Deep Learning was run with distribution=AUTO, which defaulted to a Gaussian regression problem for a real-valued response. H2O Deep Learning supports regression for distributions other than Gaussian such as Poisson, Gamma, Tweedie, Laplace. It also supports Huber loss and per-row offsets specified via an offset_column. We refer to our H2O Deep Learning regression code examples for more information.

To perform classification, the response must first be turned into a categorical (factor) feature:

```
train$bin_response <- as.factor(train$bin_response) ##make categorical
dlmodel <- h2o.deeplearning(
  x=predictors,
  y="bin_response",
  training_frame=train,
  hidden=c(10,10),
  epochs=0.1
  #balance_classes=T    ## enable this for high class imbalance
)
summary(dlmodel) ## Now the model metrics contain AUC for binary classification
plot(h2o.performance(dlmodel)) ## display ROC curve
```



Now the model performs (binary) classification, and has multiple (2) output neurons.

5.7 Unsupervised Anomaly detection

For instructions on how to build unsupervised models with H2O Deep Learning, we refer to our previous Tutorial on Anomaly Detection with H2O Deep Learning and our MNIST Anomaly detection code example, as well as our Stacked AutoEncoder R code example and another one for Unsupervised Pretraining with an AutoEncoder R code example.

5.8 H2O Deep Learning Tips & Tricks

5.8.1 Performance Tuning

The Definitive H2O Deep Learning Performance Tuning blog post covers many of the following points that affect the computational efficiency, so it's highly recommended.

5.8.2 Activation Functions

While sigmoids have been used historically for neural networks, H2O Deep Learning implements Tanh, a scaled and shifted variant of the sigmoid which is symmetric around 0. Since its output values are bounded by -1..1, the stability of the neural network is rarely endangered. However, the derivative of the tanh function is always non-zero and back-propagation (training) of the weights is more computationally expensive than for rectified linear units, or Rectifier, which is max(0,x) and has vanishing gradient for x<=0, leading to much faster training speed for large networks and is often the fastest path to accuracy on larger problems. In case you encounter instabilities with the Rectifier (in which case model building is automatically aborted), try a limited value to re-scale the weights: max_w2=10. The Maxout activation function is computationally more expensive, but can lead to higher accuracy. It is a generalized version of the Rectifier with two non-zero channels. In practice, the Rectifier (and RectifierWithDropout, see below) is the most versatile and performant option for most problems.

5.8.3 Generalization Techniques

L1 and L2 penalties can be applied by specifying the l1 and l2 parameters. Intuition: L1 lets only strong weights survive (constant pulling force towards zero), while L2 prevents any single weight from getting too big. Dropout has recently been introduced as a powerful generalization technique, and is available as a parameter per layer, including the input layer. `input_dropout_ratio` controls the amount of input layer neurons that are randomly dropped (set to zero), while `hidden_dropout_ratios` are specified for each hidden layer. The former controls overfitting with respect to the input data (useful for high-dimensional noisy data), while the latter controls overfitting of the learned features. Note that `hidden_dropout_ratios` require the activation function to end with …WithDropout.

5.8.4 Early stopping and optimizing for lowest validation error

By default, Deep Learning training stops when the stopping_metric does not improve by at least stopping_tolerance (0.01 means 1% improvement) for stopping_rounds consecutive scoring events on the training (or validation) data. By default, `overwrite_with_best_model` is enabled and the model returned after training for the specified number of epochs (or after stopping early due to convergence) is the model that has the best training set error (according to the metric specified by stopping_metric), or, if a validation set is provided, the lowest validation set error. Note that the training or validation set errors can be based on a subset of the training or validation data, depending on the values for `score_validation_samples` or `score_training_samples`, see below. For early stopping on a predefined error rate on the training data (accuracy for classification or MSE for regression), specify `classification_stop` or `regression_stop`.

5.8.5 Training Samples per MapReduce Iteration

The parameter `train_samples_per_iteration` matters especially in multi-node operation. It controls the number of rows trained on for each `MapReduce` iteration. Depending on the value selected, one MapReduce pass can sample observations, and multiple such passes are needed to train for one epoch. All H2O compute nodes then communicate to agree on the best model coefficients (weights/biases) so far, and the

model may then be scored (controlled by other parameters below). The default value of -2 indicates auto-tuning, which attemps to keep the communication overhead at 5% of the total runtime. The parameter `target_ratio_comm_to_comp` controls this ratio. This parameter is explained in more detail in the H2O Deep Learning booklet,

5.8.6 Categorical Data

For categorical data, a feature with K factor levels is automatically one-hot encoded (horizontalized) into `K-1` input neurons. Hence, the input neuron layer can grow substantially for datasets with high factor counts. In these cases, it might make sense to reduce the number of hidden neurons in the first hidden layer, such that large numbers of factor levels can be handled. In the limit of 1 neuron in the first hidden layer, the resulting model is similar to logistic regression with stochastic gradient descent, except that for classification problems, there's still a softmax output layer, and that the activation function is not necessarily a sigmoid (Tanh). If variable importances are computed, it is recommended to turn on `use_all_factor_levels` (K input neurons for K levels). The experimental option max_categorical_features uses feature hashing to reduce the number of input neurons via the hash trick at the expense of hash collisions and reduced accuracy. Another way to reduce the dimensionality of the (categorical) features is to use `h2o.glrm()`, we refer to the GLRM tutorial for more details.

5.8.7 Sparse Data

If the input data is sparse (many zeros), then it might make sense to enable the sparse option. This will result in the input not being standardized (0 mean, 1 variance), but only de-scaled (1 variance) and 0 values remain 0, leading to more efficient back-propagation. Sparsity is also a reason why CPU implementations can be faster than GPU implementations, because they can take advantage of if/else statements more effectively.

5.8.8 Missing Values

H2O Deep Learning automatically does mean imputation for missing values during training (leaving the input layer activation at 0 after standardizing the values). For testing, missing test set values are also treated the same way by default. See the `h2o.impute` function to do your own mean imputation.

5.8.9 Loss functions, Distributions, Offsets, Observation Weights

H2O Deep Learning supports advanced statistical features such as multiple loss functions, non-Gaussian distributions, per-row offsets and observation weights. In addition to Gaussian distributions and Squared loss, H2O Deep Learning supports Poisson, Gamma, Tweedie and Laplace distributions. It also supports Absolute and Huber loss and per-row offsets specified via an `offset_column.` Observation weights are supported via a user-specified weights_column.

We refer to our H2O Deep Learning R test code examples for more information.

5.8.10 Exporting Weights and Biases

The model parameters (weights connecting two adjacent layers and per-neuron bias terms) can be stored as H2O Frames (like a dataset) by enabling `export_weights_and_biases`, and they can be accessed as follows:

```
iris_dl <- h2o.deeplearning(1:4,5,as.h2o(iris),
             export_weights_and_biases=T)
#>
```

```
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
#>
  |
  |                                                                      |   0%
  |
  |======                                                                |  10%
  |
  |======================================================================| 100%
h2o.weights(iris_dl, matrix_id=1)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width
#> 1      0.12783     0.03120       0.1501     -0.0262
#> 2     -0.16487     0.00398      -0.1796     -0.0128
#> 3     -0.00595     0.02936       0.0112      0.0463
#> 4      0.01206    -0.03346       0.1536      0.0380
#> 5     -0.04686     0.01739      -0.2189     -0.1949
#> 6      0.10783    -0.16339       0.1475      0.1661
#>
#> [200 rows x 4 columns]
h2o.weights(iris_dl, matrix_id=2)
#>        C1      C2      C3      C4      C5      C6       C7        C8
#> 1 -0.0421  0.0873  0.1032  0.0493  0.0777 -0.0860  0.01497 -0.040998
#> 2  0.1119  0.1184 -0.0523 -0.0383  0.0672 -0.0472 -0.07735 -0.085760
#> 3  0.1133 -0.0271 -0.0645  0.1013 -0.0116  0.0647  0.01483 -0.019384
#> 4 -0.0903 -0.0469  0.0915  0.0645  0.0126 -0.1322 -0.04695 -0.111394
#> 5 -0.0873 -0.0587 -0.0131  0.1224  0.0581 -0.0571 -0.00122 -0.059171
#> 6 -0.0819  0.1211 -0.0613 -0.1020  0.0667  0.0548  0.04662  0.000408
#>        C9     C10      C11     C12     C13     C14     C15     C16     C17
#> 1 -0.0440 -0.0364 -0.01144 -0.0795 -0.0798 -0.0840 -0.0385 -0.1067  0.0893
#> 2  0.0692  0.1071 -0.01064  0.0891  0.0865 -0.0776 -0.0719  0.1095  0.0445
#> 3  0.0127 -0.0461  0.00277  0.0351 -0.1386  0.0272 -0.1249 -0.0810 -0.0707
#> 4  0.0668  0.1368 -0.02408  0.0327 -0.0458  0.1283 -0.0919 -0.1094  0.1187
#> 5  0.0952 -0.0570 -0.10554 -0.0821  0.0320 -0.0738 -0.0300 -0.0941 -0.1113
#> 6 -0.0351  0.1179 -0.02703 -0.0672 -0.0164  0.1170  0.0970 -0.0470  0.1090
#>        C18      C19     C20     C21     C22     C23      C24      C25
#> 1  0.08689  0.02225  0.1206 -0.0690  0.0319 -0.0364 -0.05090 -0.05037
#> 2  0.09734 -0.08062  0.0399  0.1027 -0.0833 -0.0547  0.09267  0.05185
#> 3 -0.11916 -0.08284  0.0336  0.0633  0.0649 -0.0346  0.00571 -0.08496
#> 4  0.00632  0.09803 -0.0212 -0.0665  0.1359 -0.0759 -0.12357  0.03661
#> 5  0.00186  0.09125 -0.0818  0.0858  0.0255 -0.0576 -0.08107 -0.00722
#> 6  0.01870 -0.00543  0.0419  0.0401  0.0926  0.0635 -0.06597 -0.08127
#>       C26     C27     C28      C29      C30      C31      C32     C33
#> 1  0.0282  0.0887 -0.1214  0.09143  0.07200 -0.09157  0.07595 -0.0390
#> 2 -0.0758  0.1104 -0.0496 -0.00459 -0.08874 -0.10924  0.07171 -0.0205
#> 3  0.0528  0.1088 -0.0221 -0.01530 -0.09284  0.03791 -0.08658  0.0346
#> 4  0.0493  0.0433 -0.1015  0.05354 -0.00643  0.09072 -0.00161  0.1203
#> 5  0.0997  0.0786  0.1118 -0.12147 -0.06114 -0.00352  0.05894 -0.1160
#> 6  0.0797 -0.1283 -0.0613 -0.09978 -0.12424 -0.05458  0.01594  0.1211
#>        C34      C35      C36      C37     C38     C39      C40      C41
#> 1 -0.06196 -0.06911 -0.03342 -0.08844 -0.1164  0.1217 -0.07643 -0.10717
#> 2 -0.08745  0.03487 -0.09796  0.10160 -0.0941  0.0745 -0.00918  0.00853
```

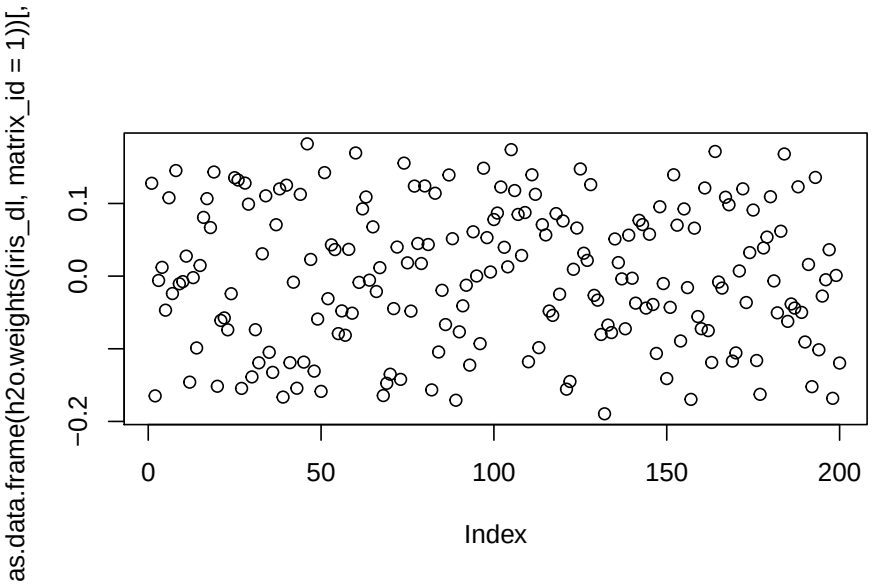
```
#> 3 -0.01081  0.10516  0.08577  0.00760 -0.0621  0.0186 -0.03089  0.03620
#> 4 -0.00596  0.00550 -0.00148 -0.00358 -0.0737  0.1133  0.00721  0.06511
#> 5 -0.08903 -0.08898  0.07231 -0.12475  0.0700 -0.0559 -0.07811  0.01823
#> 6  0.07347  0.00651 -0.07097 -0.04557 -0.0422  0.0601  0.01943  0.05346
#>       C42     C43     C44       C45      C46     C47       C48     C49
#> 1  0.1151  0.0743 -0.0653  0.094533  0.05526 -0.0310 -0.114233 -0.0790
#> 2 -0.0446  0.0355 -0.0629  0.070812 -0.01438 -0.0738 -0.000069 -0.0478
#> 3  0.1098 -0.1006 -0.1304  0.022230  0.12149 -0.1015 -0.006622  0.0831
#> 4  0.0510 -0.0849 -0.0572 -0.042663  0.04502 -0.0483 -0.012013 -0.0328
#> 5 -0.0316  0.0986 -0.0379  0.109019 -0.00642  0.0898 -0.083742  0.1087
#> 6  0.0960  0.0660 -0.0464 -0.000126 -0.07409  0.0543 -0.084661 -0.0680
#>       C50      C51     C52     C53     C54     C55     C56      C57
#> 1 -0.0330 -0.04943 -0.0151  0.0577  0.0245  0.1126  0.0388  0.00627
#> 2 -0.1076  0.00191  0.1210 -0.1144  0.0196  0.0305  0.0809  0.01624
#> 3 -0.1060  0.01020  0.0533  0.0120 -0.0686 -0.1305  0.0652  0.04340
#> 4  0.0859  0.07703  0.0797 -0.0728 -0.0238  0.1029 -0.0212  0.05030
#> 5 -0.0671 -0.06745  0.0118 -0.0486  0.1161 -0.1342  0.0166  0.04059
#> 6 -0.1244 -0.00897 -0.1195 -0.0428  0.0445  0.0880  0.0517 -0.01197
#>       C58     C59     C60       C61      C62       C63     C64      C65
#> 1  0.0771 -0.0522 -0.0470 -0.028394  0.04790  0.089048  0.1037  0.10457
#> 2  0.0209  0.0191  0.0108 -0.083047  0.02710 -0.036491  0.0331 -0.04738
#> 3 -0.0616 -0.0437 -0.0958 -0.021860  0.06576 -0.000653  0.0492  0.09300
#> 4  0.0684 -0.1045  0.0723 -0.019187  0.00277  0.103602  0.1091  0.00414
#> 5  0.0169  0.0305 -0.0518 -0.032521  0.05497 -0.035296  0.0571 -0.09255
#> 6 -0.0423 -0.0164 -0.0377 -0.000426 -0.06389 -0.020087 -0.0795  0.02603
#>       C66      C67      C68     C69     C70      C71       C72     C73
#> 1  0.04723  0.10220 -0.01884  0.0420 -0.1123  0.00192  0.000253 -0.0646
#> 2  0.06892 -0.05563  0.00178 -0.0879 -0.1032 -0.05863 -0.065380 -0.0230
#> 3 -0.10661  0.06551  0.05542 -0.0832  0.1130 -0.10356  0.040896 -0.1076
#> 4  0.00948 -0.04648 -0.08551 -0.0721 -0.0779 -0.05955  0.087991 -0.0468
#> 5  0.05514 -0.03107  0.00995 -0.1101  0.0811 -0.08852 -0.104251  0.0470
#> 6 -0.02936 -0.00475  0.09422 -0.0618 -0.1000 -0.07336  0.101635  0.0921
#>       C74       C75     C76      C77      C78     C79      C80     C81
#> 1  0.0358  0.000627 -0.0685  0.11645  0.00531 -0.0642  0.00757 -0.1068
#> 2 -0.0603  0.016079 -0.0933 -0.02089  0.04972  0.0150  0.01555 -0.1015
#> 3 -0.0793 -0.135173  0.0466 -0.11461  0.04855 -0.1068  0.06307 -0.1207
#> 4 -0.0736  0.006731  0.0666  0.00716  0.12196 -0.0983 -0.05322  0.0350
#> 5  0.0375  0.042804  0.1167  0.04143 -0.04692  0.0831 -0.05423  0.0132
#> 6  0.1221  0.060883 -0.0243 -0.05036  0.08884  0.0256  0.08862 -0.1145
#>       C82     C83     C84     C85     C86     C87     C88      C89
#> 1  0.01389  0.1201 -0.1042  0.1306 -0.0659  0.0650  0.0582 -0.03086
#> 2  0.07088  0.0582 -0.0326  0.0509 -0.0702 -0.0580 -0.1076 -0.01377
#> 3  0.00152  0.0559  0.0627  0.0205  0.0829  0.0265  0.0307 -0.06123
#> 4  0.01129 -0.0511 -0.0505 -0.0690  0.0693 -0.0191 -0.0213 -0.12633
#> 5  0.05991 -0.0795 -0.0289 -0.0987 -0.0941 -0.0835  0.0312  0.04035
#> 6 -0.10004 -0.0861 -0.0957  0.0451 -0.0505 -0.1033 -0.0481 -0.00323
#>       C90     C91     C92     C93     C94     C95      C96      C97
#> 1 -0.00502 -0.1054  0.0972  0.0430  0.1160  0.0360 -0.10116  0.085933
#> 2  0.07973 -0.0826 -0.0399  0.0633 -0.0168 -0.0929  0.04439 -0.023410
#> 3 -0.13139  0.0867  0.1259 -0.0856 -0.0247 -0.0253  0.11184 -0.005639
#> 4  0.05243  0.0805 -0.0131  0.0397 -0.0867  0.0110  0.00245 -0.000453
#> 5  0.02458  0.1052 -0.0288 -0.0848 -0.0588  0.0236  0.09611 -0.113343
#> 6  0.10122  0.0657  0.0977 -0.0403 -0.0478 -0.0938  0.01367 -0.058340
```

```
#>        C98     C99    C100    C101     C102     C103     C104    C105
#> 1  0.00461  0.0705  0.1287 -0.0550  0.05422 -0.07768  0.02050 -0.0653
#> 2 -0.11635 -0.0704  0.0242 -0.0670  0.11632 -0.10687 -0.09043  0.0360
#> 3  0.08957 -0.0241 -0.1062 -0.0217 -0.02884  0.01308  0.11537 -0.0986
#> 4 -0.06628  0.0722  0.0186  0.0667 -0.12297 -0.06988 -0.00369  0.1183
#> 5  0.11663 -0.0931 -0.0904  0.0801 -0.00345  0.08317  0.07236 -0.1240
#> 6  0.08525 -0.0530  0.0171  0.0429  0.09091 -0.00673 -0.11552  0.0672
#>       C106     C107     C108    C109    C110    C111      C112    C113
#> 1 -0.09330  0.02565 -0.00609  0.0421  0.0404 -0.0339 -0.061360 -0.0129
#> 2  0.00233 -0.10024  0.07455  0.1123  0.0782 -0.0796 -0.053981  0.1139
#> 3 -0.08723  0.06189  0.02327 -0.0539  0.1081  0.0665 -0.013018  0.0388
#> 4 -0.07975 -0.00139 -0.09287  0.1114  0.0061 -0.0122  0.000636 -0.1064
#> 5  0.05724  0.11264 -0.07843  0.0791 -0.0388 -0.0812  0.109622 -0.1046
#> 6  0.01225 -0.04051  0.09262 -0.0923 -0.0910 -0.1184  0.015054  0.0370
#>      C114    C115    C116     C117     C118    C119    C120     C121
#> 1  0.0893  0.0318  0.0375  0.00543 -0.01216  0.0135  0.0240  0.00895
#> 2  0.0414 -0.0866  0.0855  0.04991  0.00184  0.0450 -0.1195  0.00779
#> 3 -0.0986  0.0376 -0.0783 -0.04182 -0.05241 -0.0714 -0.1106  0.01534
#> 4  0.0177 -0.0981  0.1494 -0.08319 -0.08920 -0.1165  0.0156  0.04423
#> 5  0.0923 -0.0688  0.0919  0.10526  0.05222  0.0934 -0.0385  0.07644
#> 6 -0.0994 -0.0622 -0.0614 -0.05617 -0.07816  0.1023 -0.1049 -0.03828
#>      C122    C123    C124     C125    C126    C127    C128      C129
#> 1 -0.0335  0.1154  0.1009  0.03531  0.0556  0.0737 -0.0666  0.000563
#> 2  0.1008 -0.0403  0.0946 -0.02287 -0.1053 -0.1182  0.1143 -0.096493
#> 3  0.0874  0.0992 -0.0144 -0.09176  0.1292  0.0420 -0.0367 -0.090754
#> 4 -0.0635  0.0289 -0.1272 -0.03690 -0.0300 -0.0610 -0.0134 -0.039173
#> 5 -0.0932  0.1091  0.0984  0.00253  0.0197  0.0260 -0.0603 -0.017103
#> 6  0.0918  0.0160 -0.0480 -0.04069 -0.0793  0.0929  0.0262 -0.073559
#>      C130    C131    C132    C133    C134    C135     C136    C137    C138
#> 1  0.0246  0.1229  0.0864  0.1192  0.0623 -0.0356  0.11440 -0.0989 -0.0306
#> 2 -0.0604 -0.0908 -0.0645  0.0526 -0.1100 -0.0432 -0.06490 -0.0338 -0.0895
#> 3  0.0867  0.0756 -0.0393 -0.0711 -0.0171  0.0225  0.04992  0.0273 -0.0794
#> 4 -0.0863 -0.1113  0.0506  0.1127  0.0898  0.0926 -0.00305 -0.0603  0.0491
#> 5 -0.1191 -0.0296  0.1191 -0.0897 -0.1058 -0.0639 -0.04895 -0.1334  0.0492
#> 6  0.0828  0.1045  0.0292 -0.0478  0.0347 -0.0346  0.01157  0.0841  0.1166
#>       C139    C140    C141    C142     C143   C144     C145    C146
#> 1  0.09971  0.0721  0.1283  0.0655 -0.07622 0.0774 -0.00777 -0.1106
#> 2 -0.12212  0.1015  0.0482  0.0144  0.11786 0.0594  0.05931  0.0790
#> 3 -0.00736 -0.1160 -0.0587  0.0827 -0.00540 0.0567  0.04123  0.1136
#> 4 -0.09201 -0.0390  0.1229  0.0410  0.00899 0.0158  0.03078  0.0143
#> 5 -0.12324 -0.0770 -0.0071 -0.0346  0.05664 0.0436 -0.05304 -0.0263
#> 6  0.01993 -0.1186 -0.0551 -0.0554 -0.06803 0.0495  0.03765 -0.0977
#>       C147     C148    C149     C150    C151    C152    C153    C154
#> 1 -0.11650  0.01603 -0.0302 -0.08573  0.0208 -0.0340  0.0298  0.1109
#> 2  0.08605 -0.08017 -0.1151 -0.05466  0.1123  0.0945  0.0436  0.0124
#> 3 -0.06558 -0.00397 -0.0124 -0.08881  0.0701  0.0891 -0.1048  0.0175
#> 4  0.11890 -0.01553  0.1045  0.11076  0.0632  0.1212  0.0481 -0.0416
#> 5  0.00616 -0.11141  0.0861 -0.00303 -0.0825 -0.0742  0.0414  0.0187
#> 6 -0.10398  0.01735 -0.1259 -0.05250  0.0746 -0.0263  0.0047 -0.0628
#>       C155    C156     C157     C158     C159    C160    C161     C162
#> 1  0.00587  0.0538 -0.05591 -0.11223  0.07053 -0.0918 -0.0505  0.11202
#> 2  0.08482 -0.0800  0.06757 -0.08783 -0.11604 -0.0271  0.0914 -0.00459
#> 3 -0.04435  0.0557  0.02164 -0.00889  0.10257 -0.1334  0.0442 -0.15218
```

```
#> 4  0.03777 -0.0772 -0.07832 -0.01604  0.07322 -0.0335 -0.0392 -0.04962
#> 5 -0.09842 -0.0510 -0.00942 -0.07867  0.00362  0.0667 -0.0435  0.03897
#> 6 -0.11152  0.0301 -0.08371 -0.08597 -0.04411 -0.0377  0.0805 -0.08440
#>      C163    C164    C165     C166     C167     C168     C169     C170
#> 1 -0.0470 -0.1106  0.0137 -0.07625 -0.08372  0.06794 -0.05559 -0.00891
#> 2 -0.0453  0.0631 -0.0585  0.07988 -0.09018  0.07817 -0.00242  0.05795
#> 3  0.0915 -0.0425 -0.0180  0.05048 -0.10667  0.04892 -0.09012  0.01890
#> 4  0.0994  0.0545 -0.0491  0.00206 -0.00162  0.00811  0.00430 -0.07632
#> 5  0.0912  0.0103 -0.0578 -0.12141  0.10732 -0.07513  0.05090 -0.05027
#> 6 -0.0833 -0.0736  0.0423  0.05401 -0.09769 -0.09727  0.00999 -0.05769
#>      C171    C172     C173      C174     C175      C176    C177    C178
#> 1  0.0726  0.0765 -0.03345  0.000539  0.08297 -0.112874  0.0249 -0.0846
#> 2  0.0852 -0.0946  0.00369 -0.001933 -0.04844 -0.000873 -0.0222  0.1083
#> 3 -0.0489 -0.0098 -0.05220 -0.125355  0.06163  0.036121 -0.0991  0.0600
#> 4  0.1137 -0.0593 -0.07113  0.071049  0.14532 -0.039208 -0.0231  0.0804
#> 5 -0.1167  0.0036  0.11619  0.053279  0.00691  0.073048 -0.0764 -0.0716
#> 6  0.0388 -0.0853  0.08495  0.113081  0.10110 -0.003853  0.0535 -0.0728
#>       C179    C180    C181     C182    C183    C184    C185    C186
#> 1  0.04217  0.0465  0.0304  0.03261  0.0131  0.0623 -0.0636  0.0916
#> 2 -0.02873  0.0898  0.0659  0.12176  0.0276  0.0171 -0.0393 -0.0519
#> 3  0.11343 -0.0846 -0.0591 -0.02017 -0.0415  0.0336  0.0377  0.0125
#> 4 -0.05080  0.0606 -0.1005  0.03859 -0.0835  0.1018  0.1164 -0.1084
#> 5  0.10064 -0.0454 -0.0571  0.00791 -0.0175 -0.0192 -0.0703  0.1143
#> 6  0.00256  0.0593 -0.0606  0.03251 -0.0578  0.0902  0.0369  0.0586
#>      C187    C188    C189    C190    C191     C192      C193    C194
#> 1  0.0724 -0.0587  0.0473 -0.0453 -0.0438  0.00929 -0.000643 -0.1103
#> 2 -0.0252 -0.0643 -0.0810  0.1143 -0.1079  0.07810 -0.087180  0.1000
#> 3  0.0830  0.0139  0.0488 -0.0124 -0.0988  0.00301 -0.034779  0.0513
#> 4 -0.0563  0.0360  0.0426 -0.0663  0.0785 -0.13081 -0.080189 -0.0118
#> 5 -0.0826  0.1059  0.0636  0.1159  0.0531  0.08274  0.085099  0.0776
#> 6  0.0352  0.0441  0.0647  0.0866  0.0559 -0.01666  0.099376  0.0890
#>       C195     C196    C197    C198    C199     C200
#> 1 -0.10481  0.12203  0.0654  0.0294  0.1198  0.03634
#> 2  0.01139  0.08274 -0.0113  0.0285  0.0897  0.03249
#> 3  0.11617 -0.10325 -0.0720 -0.0209 -0.0356  0.00624
#> 4  0.00178  0.04353  0.0831 -0.0182  0.0367  0.01303
#> 5 -0.08615  0.00641  0.0811  0.0264 -0.0919 -0.02750
#> 6 -0.02408  0.11783 -0.1302  0.0737 -0.0638  0.01573
#>
#> [200 rows x 200 columns]
h2o.weights(iris_dl, matrix_id=3)
#>        C1      C2     C3     C4     C5    C6    C7       C8      C9   C10
#> 1 -0.2943 -0.0221 -0.216  0.667  0.357 0.208 0.287  0.00148 -0.0769 0.102
#> 2  0.0351 -0.1697 -0.185  0.621 -0.309 0.493 0.148 -0.68544 -0.1141 0.278
#> 3 -0.6050 -0.0399  0.055 -0.612  0.637 0.110 0.475 -0.65817 -0.5060 0.127
#>      C11     C12    C13   C14    C15    C16    C17    C18   C19     C20
#> 1  0.534  0.1479 -0.138 0.188  0.288  0.342 -0.401 -0.310 0.608 -0.4064
#> 2 -0.474 -0.4538  0.108 0.157 -0.556  0.174 -0.269 -0.560 0.281  0.4576
#> 3  0.510  0.0867  0.320 0.191  0.599 -0.654  0.586 -0.208 0.233 -0.0573
#>     C21     C22     C23    C24      C25    C26    C27   C28    C29    C30
#> 1 0.327  0.6139 -0.5003  0.300 -0.64122  0.684  0.551 0.572  0.228 -0.678
#> 2 0.624 -0.3150  0.1056 -0.585  0.53114 -0.638 -0.643 0.218 -0.449 -0.139
#> 3 0.139  0.0531  0.0538 -0.417 -0.00424 -0.585 -0.394 0.185  0.398  0.225
```

```
#>      C31    C32    C33     C34    C35     C36     C37     C38    C39
#> 1  0.667 -0.664 -0.586 -0.6566 -0.139 -0.0632 -0.4918 -0.0054 -0.213
#> 2 -0.594 -0.613  0.428 -0.0573  0.645 -0.5789  0.0298  0.4150 -0.318
#> 3  0.299  0.151 -0.490 -0.2106 -0.595  0.1842  0.1678 -0.1932  0.366
#>      C40   C41    C42     C43     C44    C45     C46    C47     C48    C49
#> 1  0.380 0.691  0.494  0.3176 -0.5060 -0.362 -0.0296  0.559 -0.3768  0.655
#> 2 -0.529 0.244  0.185 -0.0341  0.0761  0.010 -0.5667 -0.631  0.0877  0.327
#> 3 -0.274 0.120 -0.543  0.3044 -0.3509  0.489 -0.6235  0.154  0.4751 -0.159
#>       C50    C51    C52     C53    C54    C55    C56    C57    C58    C59
#> 1 -0.2676  0.155  0.278  0.5425 -0.127  0.394  0.383 -0.266 -0.575 -0.282
#> 2 -0.0669 -0.324  0.121  0.0785  0.501 -0.534 -0.449  0.435  0.670 -0.539
#> 3  0.0818  0.331 -0.237 -0.1026 -0.644 -0.331  0.355 -0.303 -0.428  0.355
#>      C60       C61    C62    C63    C64     C65    C66     C67    C68
#> 1  0.562 -0.27855 -0.530 -0.478 -0.629  0.0709 -0.454  0.0481  0.438
#> 2  0.316 -0.58491  0.175  0.390 -0.242  0.6663  0.483 -0.5829 -0.354
#> 3 -0.364 -0.00287 -0.489 -0.121  0.366 -0.6349  0.534 -0.4585  0.264
#>       C69    C70    C71   C72    C73    C74   C75     C76    C77     C78
#> 1 -0.0764 -0.598 -0.127 0.538  0.241 -0.558 0.456 -0.1107  0.465  0.6176
#> 2  0.5509 -0.441 -0.228 0.615 -0.087  0.387 0.423 -0.0731 -0.519 -0.0891
#> 3  0.6281 -0.417  0.229 0.539  0.494  0.211 0.613  0.3096  0.379 -0.1622
#>       C79    C80   C81    C82    C83    C84    C85    C86     C87    C88
#> 1 -0.6572  0.503 0.102 -0.661  0.398  0.137  0.300  0.412  0.0238  0.357
#> 2  0.0468 -0.377 0.179 -0.576 -0.178 -0.188  0.635 -0.417 -0.5376 -0.208
#> 3 -0.1629  0.474 0.298  0.473  0.462  0.130 -0.132  0.209  0.2179  0.372
#>      C89    C90     C91    C92     C93    C94    C95    C96    C97    C98
#> 1 -0.514 -0.167 -0.1140 -0.370  0.6681  0.417  0.686 -0.167 -0.304 -0.422
#> 2 -0.146 -0.566  0.0277  0.260 -0.0437  0.260  0.427  0.092 -0.260  0.668
#> 3 -0.509 -0.265  0.0538 -0.422 -0.2022 -0.579 -0.035 -0.637 -0.592  0.357
#>      C99   C100   C101    C102   C103   C104   C105   C106   C107   C108
#> 1 -0.216  0.623 -0.597  0.3520 -0.270  0.174 -0.319 0.1834 -0.464 -0.597
#> 2  0.286 -0.496 -0.371 -0.0504  0.500 -0.132 -0.430 0.0512 -0.464 -0.351
#> 3  0.225  0.603  0.218  0.0431  0.316  0.171 -0.469 0.3045 -0.179 -0.563
#>     C109   C110   C111   C112   C113   C114  C115   C116  C117   C118
#> 1 -0.440 -0.233  0.303 -0.455 -0.517 -0.331 0.273  0.160 0.202 -0.617
#> 2  0.335 -0.256 -0.484  0.053 -0.234 -0.482 0.143  0.357 0.309 -0.235
#> 3 -0.577 -0.194  0.672  0.439  0.516 -0.561 0.150 -0.435 0.125  0.179
#>     C119   C120    C121    C122    C123   C124   C125   C126   C127   C128
#> 1 -0.315  0.428 -0.0395 -0.5965  0.0295  0.240  0.447 -0.470  0.293  0.455
#> 2  0.158 -0.346  0.2760 -0.3515  0.3489 -0.215 -0.206 -0.531  0.602  0.604
#> 3  0.339  0.133  0.1634  0.0247 -0.2497 -0.360  0.287  0.550 -0.235 -0.230
#>     C129   C130    C131   C132   C133   C134   C135   C136    C137  C138
#> 1 -0.397 -0.164 -0.1708 -0.440  0.616  0.132  0.648 0.1048 -0.6893 0.633
#> 2  0.557 -0.198  0.4765  0.485 -0.673 -0.658 -0.151 0.0063 -0.0139 0.190
#> 3  0.115 -0.379 -0.0537  0.158 -0.597 -0.584 -0.473 0.3004  0.4636 0.582
#>     C139   C140  C141    C142    C143   C144   C145   C146    C147   C148
#> 1 -0.229 -0.124 0.585  0.1181  0.0607 -0.533  0.263  0.560  0.4851 -0.261
#> 2  0.326 -0.370 0.328  0.2604 -0.0418  0.289  0.471  0.635 -0.0526 -0.384
#> 3 -0.515  0.601 0.614 -0.0346 -0.4742 -0.285 -0.616 -0.566  0.5417 -0.152
#>      C149   C150   C151   C152   C153    C154   C155   C156    C157
#> 1 -0.5325 0.6738  0.449 -0.310 -0.605 -0.0212 -0.134 -0.161  0.1530
#> 2  0.4986 0.0228 -0.107  0.653 -0.423 -0.4857  0.623 -0.187 -0.0688
#> 3  0.0983 0.5357  0.420 -0.353  0.198  0.0816 -0.364  0.475 -0.0502
#>      C158    C159   C160   C161    C162    C163   C164   C165   C166
```

```
#> 1 -0.2934  0.2854 0.4478  0.543  0.0992  0.0752 -0.517  0.559 -0.650
#> 2 -0.3228 -0.4128 0.3912 -0.340 -0.3029  0.6183 -0.346 -0.575  0.183
#> 3 -0.0565 -0.0511 0.0593 -0.244  0.6380 -0.4106 -0.258  0.589  0.535
#>     C167   C168   C169   C170   C171   C172    C173   C174   C175    C176
#> 1 -0.613 -0.429 -0.558  0.228 -0.528 -0.358  0.2361 -0.660 -0.495 -0.0473
#> 2  0.552 -0.584  0.602 -0.477  0.352 -0.602  0.0166  0.446  0.265  0.2689
#> 3 -0.286 -0.634 -0.264  0.350 -0.513 -0.518 -0.5546  0.335 -0.143  0.0734
#>     C177   C178   C179    C180      C181   C182   C183   C184   C185
#> 1 -0.247 -0.509  0.547 -0.4437  0.553396  0.374 -0.192 -0.518 -0.669
#> 2 -0.310  0.231 -0.317 -0.3347  0.357966 -0.352 -0.410 -0.415  0.173
#> 3 -0.396 -0.468 -0.538  0.0667 -0.000532 -0.632 -0.468  0.250  0.645
#>       C186   C187   C188   C189   C190   C191   C192  C193   C194    C195
#> 1 -0.05092 -0.140 -0.108 -0.252 -0.160 -0.577  0.123 0.390 0.3165 -0.2131
#> 2 -0.30534  0.662 -0.429 -0.571  0.670 -0.654 -0.317 0.252 0.4049 -0.0293
#> 3  0.00476  0.284 -0.181  0.333 -0.245 -0.207 -0.168 0.142 0.0543 -0.1701
#>     C196    C197   C198    C199   C200
#> 1 -0.024 -0.0569 -0.115 -0.0083 -0.649
#> 2  0.312  0.5508  0.167 -0.4435 -0.455
#> 3  0.433 -0.2022  0.631  0.0865 -0.579
#>
#> [3 rows x 200 columns]
h2o.biases(iris_dl,  vector_id=1)
#>      C1
#> 1 0.492
#> 2 0.467
#> 3 0.477
#> 4 0.474
#> 5 0.491
#> 6 0.499
#>
#> [200 rows x 1 column]
h2o.biases(iris_dl,  vector_id=2)
#>      C1
#> 1 1.006
#> 2 0.995
#> 3 0.994
#> 4 1.005
#> 5 0.995
#> 6 0.997
#>
#> [200 rows x 1 column]
h2o.biases(iris_dl,  vector_id=3)
#>          C1
#> 1  0.000386
#> 2  0.001556
#> 3 -0.002351
#>
#> [3 rows x 1 column]
#plot weights connecting `Sepal.Length` to first hidden neurons
plot(as.data.frame(h2o.weights(iris_dl,  matrix_id=1))[,1])
```



5.8.11 Reproducibility

Every run of DeepLearning results in different results since multithreading is done via Hogwild! that benefits from intentional lock-free race conditions between threads. To get reproducible results for small datasets and testing purposes, set `reproducible=T` and set `seed=1337` (pick any integer). This will not work for big data for technical reasons, and is probably also not desired because of the significant slowdown (runs on 1 core only).

5.8.12 Scoring on Training/Validation Sets During Training

The training and/or validation set errors can be based on a subset of the training or validation data, depending on the values for `score_validation_samples` (defaults to 0: all) or `score_training_samples` (defaults to 10,000 rows, since the training error is only used for early stopping and monitoring). For large datasets, Deep Learning can automatically sample the validation set to avoid spending too much time in scoring during training, especially since scoring results are not currently displayed in the model returned to R.

Note that the default value of `score_duty_cycle=0.1` limits the amount of time spent in scoring to 10%, so a large number of scoring samples won't slow down overall training progress too much, but it will always score once after the first MapReduce iteration, and once at the end of training.

Stratified sampling of the validation dataset can help with scoring on datasets with class imbalance. Note that this option also requires `balance_classes` to be enabled (used to over/under-sample the training dataset, based on the max. relative size of the resulting training dataset, max_after_balance_size):

More information can be found in the H2O Deep Learning booklet, in our H2O SlideShare Presentations, our H2O YouTube channel, as well as on our H2O Github Repository, especially in our H2O Deep Learning R tests, and H2O Deep Learning Python tests.

5.9 All done, shutdown H2O

```
h2o.shutdown(prompt=FALSE)
#> [1] TRUE
```

Chapter 6

Sensitivity analysis for neural networks

6.1 Introduction

https://beckmw.wordpress.com/tag/nnet/

I've made quite a few blog posts about neural networks and some of the diagnostic tools that can be used to 'demystify' the information contained in these models. Frankly, I'm kind of sick of writing about neural networks but I wanted to share one last tool I've implemented in R. I'm a strong believer that supervised neural networks can be used for much more than prediction, as is the common assumption by most researchers. I hope that my collection of posts, including this one, has shown the versatility of these models to develop inference into causation. To date, I've authored posts on visualizing neural networks, animating neural networks, and determining importance of model inputs. This post will describe a function for a sensitivity analysis of a neural network. Specifically, I will describe an approach to evaluate the form of the relationship of a response variable with the explanatory variables used in the model.

The general goal of a sensitivity analysis is similar to evaluating relative importance of explanatory variables, with a few important distinctions. For both analyses, we are interested in the relationships between explanatory and response variables as described by the model in the hope that the neural network has explained some real-world phenomenon. Using Garson's algorithm,1 we can get an idea of the magnitude and sign of the relationship between variables relative to each other. Conversely, the sensitivity analysis allows us to obtain information about the form of the relationship between variables rather than a categorical description, such as variable x is positively and strongly related to y. For example, how does a response variable change in relation to increasing or decreasing values of a given explanatory variable? Is it a linear response, non-linear, uni-modal, no response, etc.? Furthermore, how does the form of the response change given values of the other explanatory variables in the model? We might expect that the relationship between a response and explanatory variable might differ given the context of the other explanatory variables (i.e., an interaction may be present). The sensitivity analysis can provide this information.

As with most of my posts, I've created the sensitivity analysis function using ideas from other people that are much more clever than me. I've simply converted these ideas into a useful form in R. Ultimate credit for the sensitivity analysis goes to Sovan Lek (and colleagues), who developed the approach in the mid-1990s. The 'Lek-profile method' is described briefly in Lek et al. 19962 and in more detail in Gevrey et al. 2003.3 I'll provide a brief summary here since the method is pretty simple. In fact, the profile method can be extended to any statistical model and is not specific to neural networks, although it is one of few methods used to evaluate the latter. For any statistical model where multiple response variables are related to multiple explanatory variables, we choose one response and one explanatory variable. We obtain predictions of the response variable across the range of values for the given explanatory variable. All other explanatory variables

are held constant at a given set of respective values (e.g., minimum, 20th percentile, maximum). The final product is a set of response curves for one response variable across the range of values for one explanatory variable, while holding all other explanatory variables constant. This is implemented in R by creating a matrix of values for explanatory variables where the number of rows is the number of observations and the number of columns is the number of explanatory variables. All explanatory variables are held at their mean (or other constant value) while the variable of interest is sequenced from its minimum to maximum value across the range of observations. This matrix (actually a data frame) is then used to predict values of the response variable from a fitted model object. This is repeated for different variables.

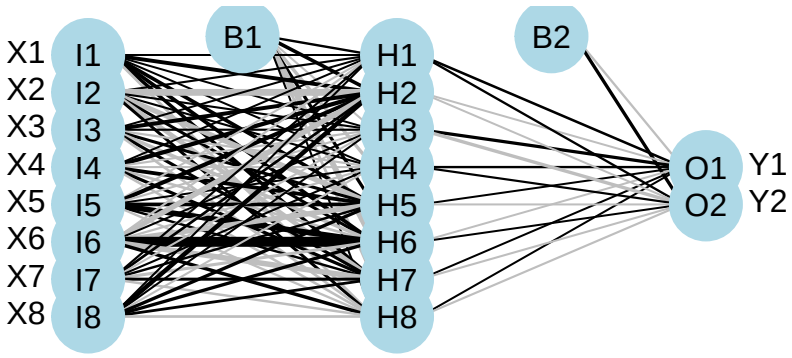
I'll illustrate the function using simulated data, as I've done in previous posts. The exception here is that I'll be using two response variables instead of one. The two response variables are linear combinations of eight explanatory variables, with random error components taken from a normal distribution. The relationships between the variables are determined by the arbitrary set of parameters (`parms1` and `parms2`). The explanatory variables are partially correlated and taken from a multivariate normal distribution.

```
require(clusterGeneration)
#> Loading required package: clusterGeneration
#> Loading required package: MASS
require(nnet)
#> Loading required package: nnet

#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')
```

```
mod1 <- nnet(rand.vars,resp,size=8,linout=T)
#> # weights:  90
#> initial  value 30121.205794
#> iter  10 value 130.537462
#> iter  20 value 57.187090
#> iter  30 value 47.285919
#> iter  40 value 42.778564
#> iter  50 value 39.837784
#> iter  60 value 36.694632
#> iter  70 value 35.140948
#> iter  80 value 34.268819
#> iter  90 value 33.772282
#> iter 100 value 33.472654
#> final  value 33.472654
```

```
#> stopped after 100 iterations
```

```
#import the function from Github
library(devtools)

# source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/466c1474d0a505ff044412703516c34f1
source("nnet_plot_update.r")

#plot each model
plot.nnet(mod1)
#> Loading required package: scales
#> Loading required package: reshape
```



6.2 The Lek profile function

We've created a neural network that hopefully describes the relationship of two response variables with eight explanatory variables. The sensitivity analysis lets us visualize these relationships. The Lek profile function can be used once we have a neural network model in our workspace. The function is imported and used as follows:

```
# source('https://gist.githubusercontent.com/fawda123/6860630/raw/b8bf4a6c88d6b392b1bfa6ef24759ae98f318
source("lek_fun.r")

lek.fun(mod1)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
```

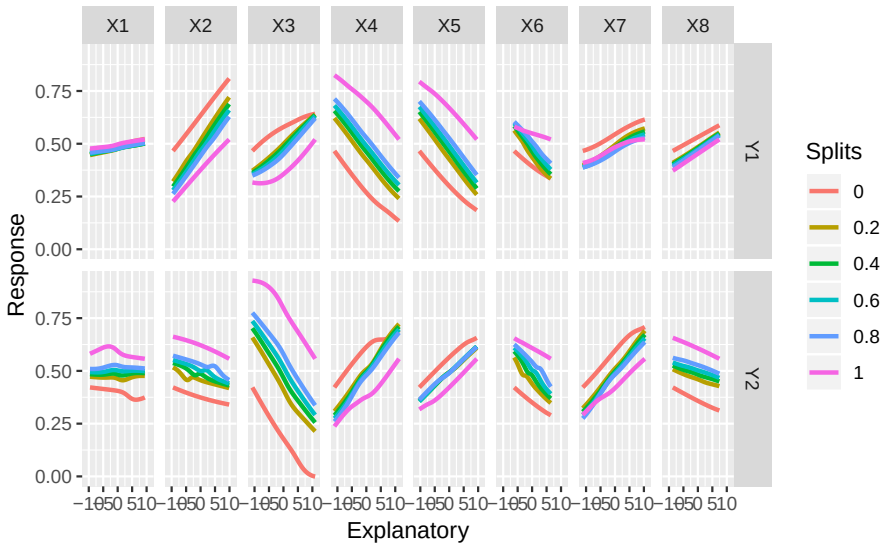


Fig: Sensitivity analysis of the two response variables in the neural network model to individual explanatory variables. Splits represent the quantile values at which the remaining explanatory variables were held constant. The function can be obtained here

By default, the function runs a sensitivity analysis for all variables. This creates a busy plot so we may want to look at specific variables of interest. Maybe we want to evaluate different quantile values as well. These options can be changed using the arguments.

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

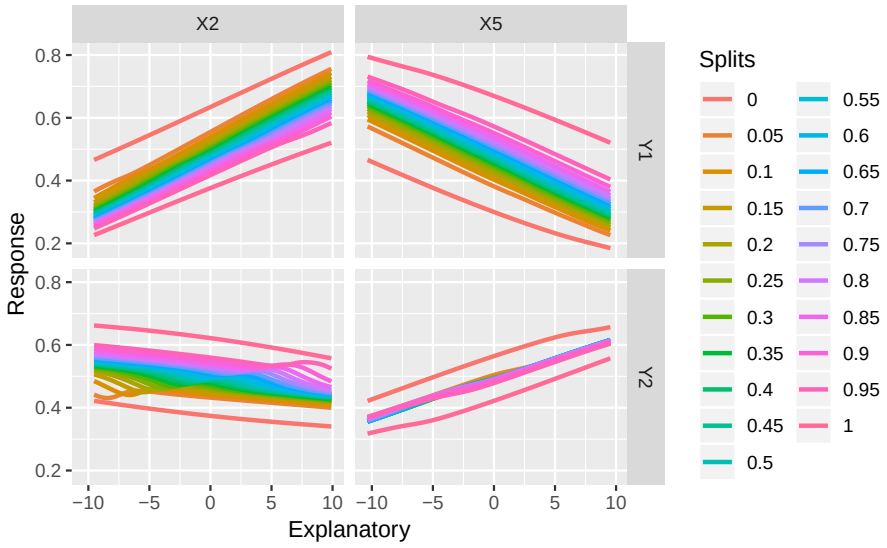
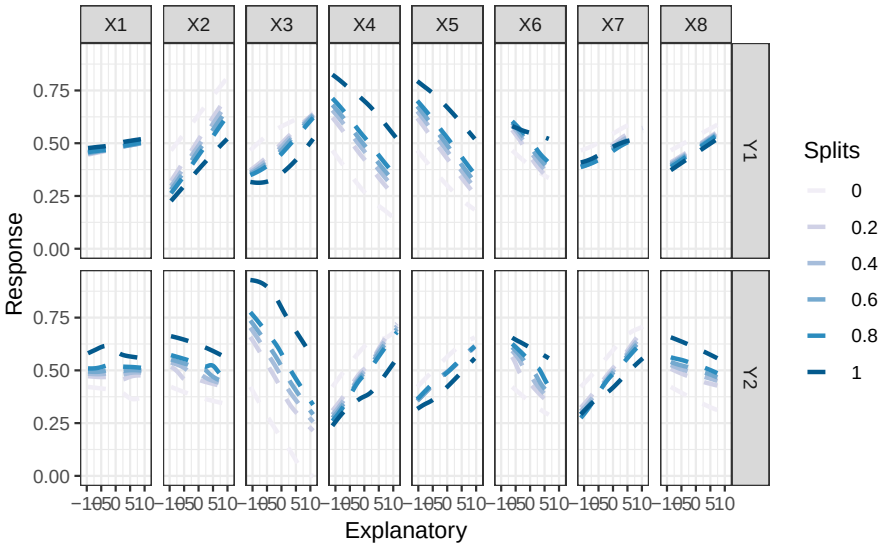


Fig: Sensitivity analysis of the two response variables in relation to explanatory variables X2 and X5 and different quantile values for the remaining variables.

The function also returns a ggplot2 object that can be further modified. You may prefer a different theme, color, or line type, for example.

```
p1<-lek.fun(mod1)
class(p1)
#> [1] "gg"     "ggplot"
# [1] "gg"     "ggplot"
```

```
p1 +
   theme_bw() +
   scale_colour_brewer(palette="PuBu") +
   scale_linetype_manual(values=rep('dashed',6)) +
   scale_size_manual(values=rep(1,6))
#> Scale for 'linetype' is already present. Adding another scale for
#> 'linetype', which will replace the existing scale.
#> Scale for 'size' is already present. Adding another scale for 'size',
#> which will replace the existing scale.
```



6.3 Getting a dataframe from `lek`

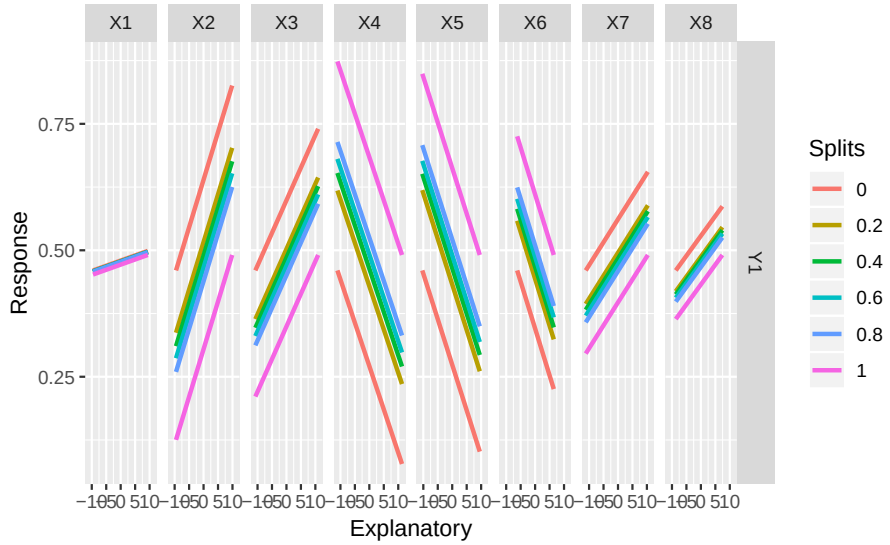
Finally, the actual values from the sensitivity analysis can be returned if you'd prefer that instead. The output is a data frame in long form that was created using melt.list from the reshape package for compatibility with ggplot2. The six columns indicate values for explanatory variables on the x-axes, names of the response variables, predicted values of the response variables, quantiles at which other explanatory variables were held constant, and names of the explanatory variables on the x-axes.

```
head(lek.fun(mod1,val.out = TRUE))
#>   Explanatory resp.name Response Splits exp.name
#> 1       -9.58        Y1    0.466      0       X1
#> 2       -9.39        Y1    0.466      0       X1
#> 3       -9.19        Y1    0.467      0       X1
#> 4       -9.00        Y1    0.467      0       X1
#> 5       -8.81        Y1    0.468      0       X1
#> 6       -8.62        Y1    0.468      0       X1
```

6.4 The `lek` function works with `lm`

I mentioned earlier that the function is not unique to neural networks and can work with other models created in R. I haven't done an extensive test of the function, but I'm fairly certain that it will work if the model object has a predict method (e.g., predict.lm). Here's an example using the function to evaluate a multiple linear regression for one of the response variables.

```
mod2 <-lm(Y1 ~ ., data = cbind(resp['Y1', drop = F], rand.vars))
lek.fun(mod2)
```



This function has little relevance for conventional models like linear regression since a wealth of `diagnostic` tools are already available (e.g., effects plots, add/drop procedures, outlier tests, etc.). The application of the function to neural networks provides insight into the relationships described by the models, insights that to my knowledge, cannot be obtained using current tools in R. This post concludes my contribution of diagnostic tools for neural networks in R and I hope that they have been useful to some of you. I have spent the last year or so working with neural networks and my opinion of their utility is mixed. I see advantages in the use of highly flexible computer-based algorithms, although in most cases similar conclusions can be made using more conventional analyses. I suggest that neural networks only be used *if there is an extremely high sample size* and other methods have proven inconclusive. Feel free to voice your opinions or suggestions in the comments.

6.5 `lek` function works with `RSNNS`

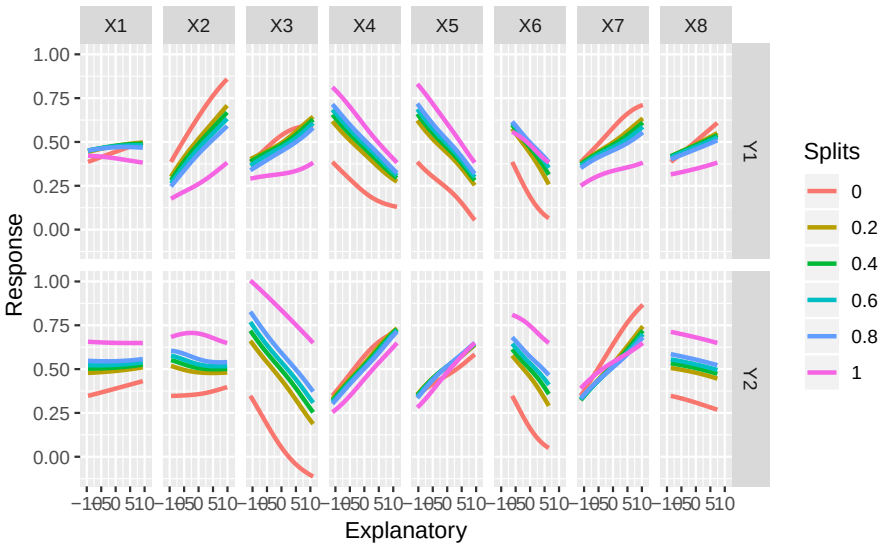
```
require(clusterGeneration)
require(RSNNS)
#> Loading required package: RSNNS
#> Loading required package: Rcpp
require(devtools)

#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')
```

```
tibble::as_tibble(rand.vars)
#> # A tibble: 10,000 x 8
#>        X1     X2     X3     X4     X5    X6    X7     X8
#>     <dbl>  <dbl>  <dbl>  <dbl>  <dbl> <dbl> <dbl>  <dbl>
#> 1  1.61    2.13   2.13   3.97  -1.34   2.00  3.11 -2.55
#> 2 -1.25    3.07  -0.325  1.61  -0.484  2.28  2.98 -1.71
#> 3 -3.17   -1.29  -1.77  -1.66  -0.549 -3.19  1.07  1.81
#> 4 -2.39    3.28  -3.42  -0.160 -1.52   2.67  7.05 -1.14
#> 5 -1.55   -0.181 -1.14   2.27  -1.68  -1.67  3.08  0.334
#> 6  0.0690 -1.54  -2.98   2.84   1.42   1.31  1.82  2.07
#> # ... with 9,994 more rows
```

```
tibble::as_tibble(resp)
#> # A tibble: 10,000 x 2
#>       Y1    Y2
#>    <dbl> <dbl>
#> 1 0.461 0.500
#> 2 0.416 0.509
#> 3 0.534 0.675
#> 4 0.548 0.619
#> 5 0.519 0.659
#> 6 0.389 0.622
#> # ... with 9,994 more rows
```

```
# create neural network model
mod2 <- mlp(rand.vars, resp, size = 8, linOut = T)

#import sensitivity analysis function
source_url('https://gist.githubusercontent.com/fawda123/6860630/raw/b8bf4a6c88d6b392b1bfa6ef24759ae98f3
#> SHA-1 hash of file is 4a2d33b94a08f46a94518207a4ae7cc412845222

#sensitivity analsyis, note 'exp.in' argument
lek.fun(mod2, exp.in = rand.vars)
```


X1
X2
X3
X4
X5
X6
X7
X8
Y1
Y2
Response
1.00
0.75
0.50
0.25
0.00
−10 −5 0 5 10
Explanatory
Splits
0
0.2
0.4
0.6
0.8
1

Chapter 7

References

1 Garson GD. 1991. Interpreting neural network connection weights. Artificial Intelligence Expert. 6:46–51. 2 Lek S, Delacoste M, Baran P, Dimopoulos I, Lauga J, Aulagnier S. 1996. Application of neural networks to modelling nonlinear relationships in Ecology. Ecological Modelling. 90:39-52. 3 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.

Chapter 8

Regression with ANN - Yacht Hydrodynamics

8.1 Introduction

Regression ANNs predict an output variable as a function of the inputs. The input features (independent variables) can be categorical or numeric types, however, for regression ANNs, we require a numeric dependent variable. If the output variable is a categorical variable (or binary) the ANN will function as a classifier (see next tutorial).

Source: http://uc-r.github.io/ann_regression

In this tutorial we introduce a neural network used for numeric predictions and cover:

- Replication requirements: What you'll need to reproduce the analysis in this tutorial.
- Data Preparation: Preparing our data.
- 1st Regression ANN: Constructing a 1-hidden layer ANN with 1 neuron.
- Regression Hyperparameters: Tuning the model.
- Wrapping Up: Final comments and some exercises to test your skills.

8.2 Replication Requirements

We require the following packages for the analysis.

```
library(tidyverse)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#> Registered S3 method overwritten by 'rvest':
#>   method            from
#>   read_xml.response xml2
#> -- Attaching packages ---------------------------------------- tidyverse 1.2.1 --
#> v ggplot2 3.1.1       v purrr   0.3.2
#> v tibble  2.1.1       v dplyr   0.8.0.1
#> v tidyr   0.8.3       v stringr 1.4.0
#> v readr   1.3.1       v forcats 0.4.0
```

```
#> -- Conflicts ------------------------------------------ tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
library(neuralnet)
#>
#> Attaching package: 'neuralnet'
#> The following object is masked from 'package:dplyr':
#>
#>     compute
library(GGally)
#> Registered S3 method overwritten by 'GGally':
#>   method from
#>   +.gg   ggplot2
#>
#> Attaching package: 'GGally'
#> The following object is masked from 'package:dplyr':
#>
#>     nasa
```

8.3 Data Preparation

Our regression ANN will use the **Yacht Hydrodynamics** data set from UCI's Machine Learning Repository. The yacht data was provided by Dr. Roberto Lopez email. This data set contains data contains results from 308 full-scale experiments performed at the Delft Ship Hydromechanics Laboratory where they test 22 different hull forms. Their experiment tested the effect of variations in the hull geometry and the ship's Froude number on the craft's residuary resistance per unit weight of displacement.

To begin we download the data from UCI.

```
url <- 'http://archive.ics.uci.edu/ml/machine-learning-databases/00243/yacht_hydrodynamics.data'

Yacht_Data <- read_table(file = url,
                         col_names = c('LongPos_COB', 'Prismatic_Coeff',
                                       'Len_Disp_Ratio', 'Beam_Draut_Ratio',
                                       'Length_Beam_Ratio','Froude_Num',
                                       'Residuary_Resist')) %>%
  na.omit()
#> Parsed with column specification:
#> cols(
#>   LongPos_COB = col_double(),
#>   Prismatic_Coeff = col_double(),
#>   Len_Disp_Ratio = col_double(),
#>   Beam_Draut_Ratio = col_double(),
#>   Length_Beam_Ratio = col_double(),
#>   Froude_Num = col_double(),
#>   Residuary_Resist = col_double()
#> )
```

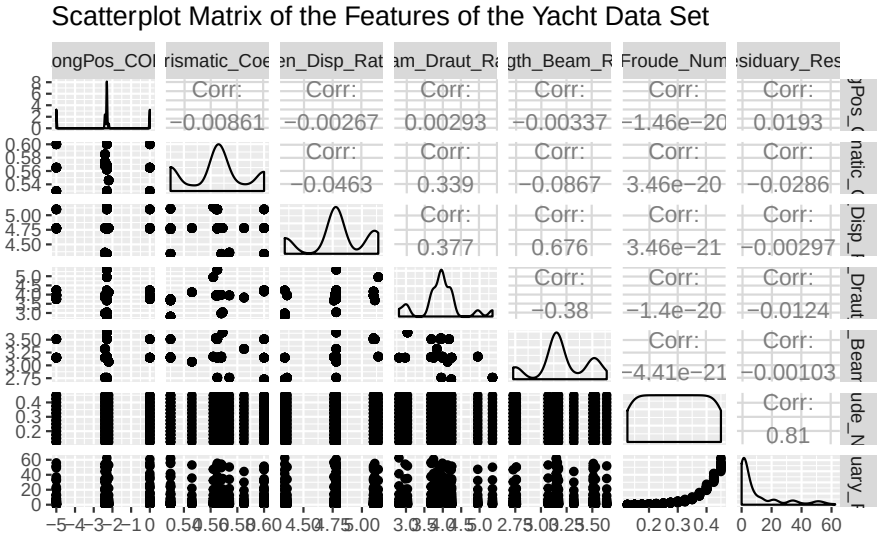
```
dplyr::glimpse(Yacht_Data)
#> Observations: 308
#> Variables: 7
#> $ LongPos_COB       <dbl> -2.3, -2.3, -2.3, -2.3, -2.3, -2.3, -2.3, -2...
#> $ Prismatic_Coeff   <dbl> 0.568, 0.568, 0.568, 0.568, 0.568, 0.568, 0....
```

```
#> $ Len_Disp_Ratio    <dbl> 4.78, 4.78, 4.78, 4.78, 4.78, 4.78, 4.78, 4....
#> $ Beam_Draut_Ratio  <dbl> 3.99, 3.99, 3.99, 3.99, 3.99, 3.99, 3.99, 3....
#> $ Length_Beam_Ratio <dbl> 3.17, 3.17, 3.17, 3.17, 3.17, 3.17, 3.17, 3....
#> $ Froude_Num        <dbl> 0.125, 0.150, 0.175, 0.200, 0.225, 0.250, 0....
#> $ Residuary_Resist  <dbl> 0.11, 0.27, 0.47, 0.78, 1.18, 1.82, 2.61, 3....
```

```
# save the dataset locally
write.csv(Yacht_Data, file = file.path(data_raw_dir, "yach_data.csv"))
```

Prior to any data analysis lets take a look at the data set.

```
ggpairs(Yacht_Data, title = "Scatterplot Matrix of the Features of the Yacht Data Set")
```



Here we see an excellent summary of the variation of each feature in our data set. Draw your attention to the bottom-most strip of scatter-plots. This shows the residuary resistance as a function of the other data set features (independent experimental values). The greatest variation appears with the Froude Number feature. It will be interesting to see how this pattern appears in the subsequent regression ANNs.

Prior to regression ANN construction we first must split the Yacht data set into test and training data sets. Before we split, first scale each feature to fall in the `[0,1]` interval.

```
# Scale the Data
scale01 <- function(x){
  (x - min(x)) / (max(x) - min(x))
}

Yacht_Data <- Yacht_Data %>%
  mutate_all(scale01)

# Split into test and train sets
set.seed(12345)
Yacht_Data_Train <- sample_frac(tbl = Yacht_Data, replace = FALSE, size = 0.80)
Yacht_Data_Test <- anti_join(Yacht_Data, Yacht_Data_Train)
#> Joining, by = c("LongPos_COB", "Prismatic_Coeff", "Len_Disp_Ratio", "Beam_Draut_Ratio", "Length_Beam
```

The `scale01()` function maps each data observation onto the `[0,1]` interval as called in the dplyr `mutate_all()` function. We then provided a seed for reproducible results and randomly extracted (without replacement) 80% of the observations to build the `Yacht_Data_Train` data set. Using dplyr's `anti_join()` function we extracted all the observations not within the `Yacht_Data_Train` data set as our test data set

in `Yacht_Data_Test`.

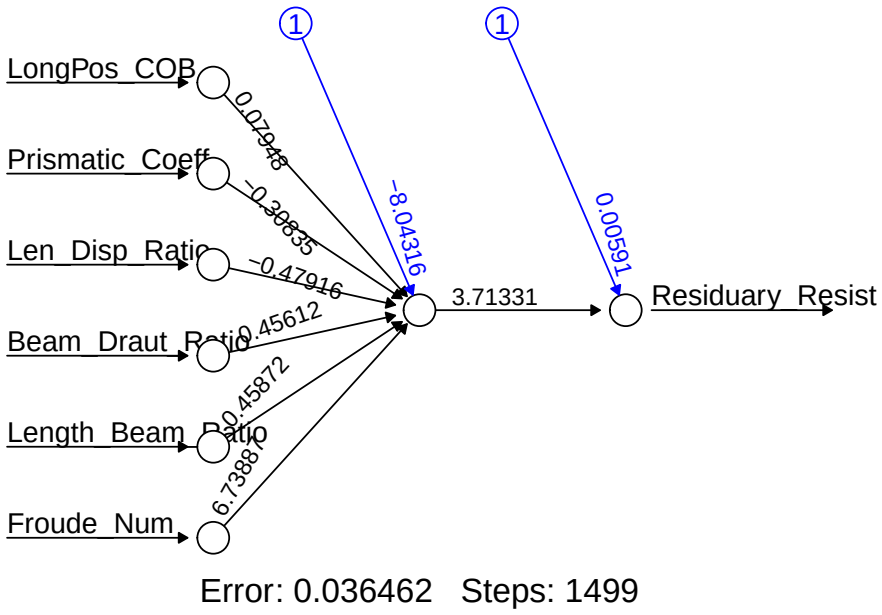
8.4 1st Regression ANN

To begin we construct a 1-hidden layer ANN with 1 neuron, the simplest of all neural networks.

```
set.seed(12321)
Yacht_NN1 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff +
                         Len_Disp_Ratio + Beam_Draut_Ratio + Length_Beam_Ratio +
                         Froude_Num, data = Yacht_Data_Train)
```

The `Yacht_NN1` is a list containing all parameters of the regression ANN as well as the results of the neural network on the test data set. To view a diagram of the `Yacht_NN1` use the `plot()` function.

```
plot(Yacht_NN1, rep = 'best')
```



This plot shows the weights learned by the `Yacht_NN1` neural network, and displays the number of iterations before convergence, as well as the SSE of the training data set. To manually compute the SSE you can use the following:

```
NN1_Train_SSE <- sum((Yacht_NN1$net.result - Yacht_Data_Train[, 7])^2)/2
paste("SSE: ", round(NN1_Train_SSE, 4))
#> [1] "SSE:  0.0365"
## [1] "SSE:  0.0361"
```

This SSE is the error associated with the training data set. A superior metric for estimating the generalization capability of the ANN would be the SSE of the test data set. Recall, the test data set contains observations not used to train the `Yacht_NN1 ANN`. To calculate the test error, we first must run our test observations through the `Yacht_NN1` ANN. This is accomplished with the neuralnet package `compute()` function, which takes as its first input the desired neural network object created by the `neuralnet()` function, and the second argument the test data set feature (independent variable(s)) values.

```
Test_NN1_Output <- compute(Yacht_NN1, Yacht_Data_Test[, 1:6])$net.result
NN1_Test_SSE <- sum((Test_NN1_Output - Yacht_Data_Test[, 7])^2)/2
NN1_Test_SSE
```

```
#> [1] 0.0139
## [1] 0.008417631461
```

The `compute()` function outputs the response variable, in our case the `Residuary_Resist`, as estimated by the neural network. Once we have the ANN estimated response we can compute the test SSE. Comparing the test error of 0.0084 to the training error of 0.0361 we see that in our case our test error is smaller than our training error.

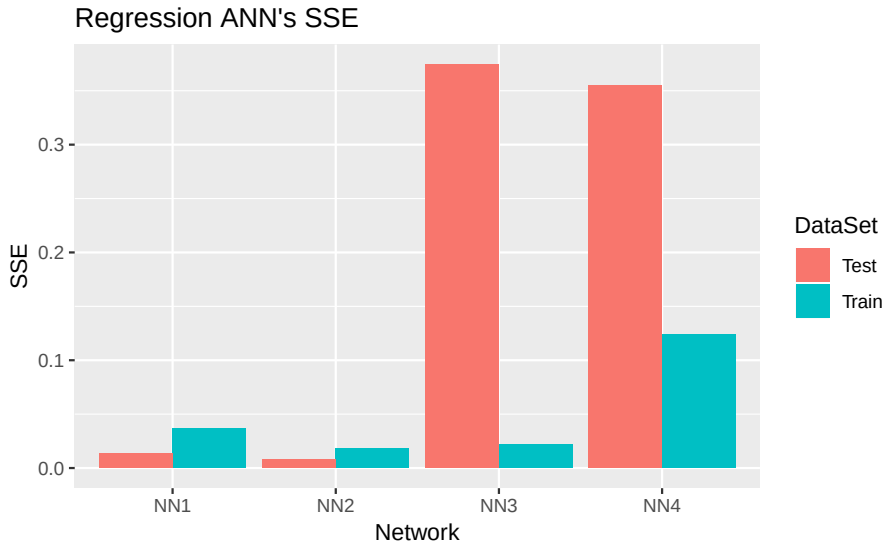
8.5 Regression Hyperparameters

We have constructed the most basic of regression ANNs without modifying any of the default hyperparameters associated with the `neuralnet()` function. We should try and improve the network by modifying its basic structure and hyperparameter modification. To begin we will add depth to the hidden layer of the network, then we will change the activation function from the logistic to the tangent hyperbolicus (tanh) to determine if these modifications can improve the test data set SSE. When using the tanh activation function, we first must rescale the data from $[0, 1]$ to $[-1, 1]$ using the `rescale` package. For the purposes of this exercise we will use the same random seed for reproducible results, generally this is not a best practice.

```
# 2-Hidden Layers, Layer-1 4-neurons, Layer-2, 1-neuron, logistic activation
# function
set.seed(12321)
Yacht_NN2 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Disp_Ratio + Beam_Draut_Ra
                       data = Yacht_Data_Train,
                       hidden = c(4, 1),
                       act.fct = "logistic")

## Training Error
NN2_Train_SSE <- sum((Yacht_NN2$net.result - Yacht_Data_Train[, 7])^2)/2

## Test Error
Test_NN2_Output <- compute(Yacht_NN2, Yacht_Data_Test[, 1:6])$net.result
NN2_Test_SSE <- sum((Test_NN2_Output - Yacht_Data_Test[, 7])^2)/2

# Rescale for tanh activation function
scale11 <- function(x) {
    (2 * ((x - min(x))/(max(x) - min(x)))) - 1
}
Yacht_Data_Train <- Yacht_Data_Train %>% mutate_all(scale11)
Yacht_Data_Test <- Yacht_Data_Test %>% mutate_all(scale11)

# 2-Hidden Layers, Layer-1 4-neurons, Layer-2, 1-neuron, tanh activation
# function
set.seed(12321)
Yacht_NN3 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Disp_Ratio + Beam_Draut_Ra
                       data = Yacht_Data_Train,
                       hidden = c(4, 1),
                       act.fct = "tanh")

## Training Error
NN3_Train_SSE <- sum((Yacht_NN3$net.result - Yacht_Data_Train[, 7])^2)/2

## Test Error
Test_NN3_Output <- compute(Yacht_NN3, Yacht_Data_Test[, 1:6])$net.result
```

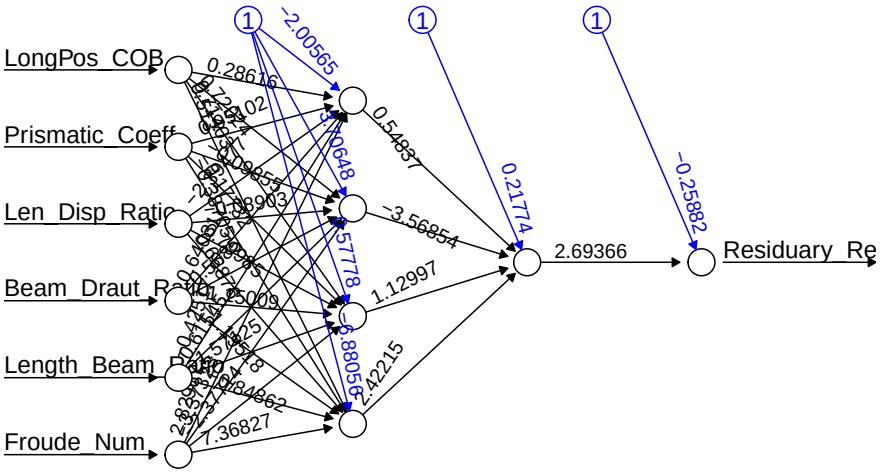
```
NN3_Test_SSE <- sum((Test_NN3_Output - Yacht_Data_Test[, 7])^2)/2

# 1-Hidden Layer, 1-neuron, tanh activation function
set.seed(12321)
Yacht_NN4 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Disp_Ratio + Beam_Draut_Ra
                       data = Yacht_Data_Train,
                       act.fct = "tanh")

## Training Error
NN4_Train_SSE <- sum((Yacht_NN4$net.result - Yacht_Data_Train[, 7])^2)/2

## Test Error
Test_NN4_Output <- compute(Yacht_NN4, Yacht_Data_Test[, 1:6])$net.result
NN4_Test_SSE <- sum((Test_NN4_Output - Yacht_Data_Test[, 7])^2)/2
```

```
# Bar plot of results
Regression_NN_Errors <- tibble(Network = rep(c("NN1", "NN2", "NN3", "NN4"), each = 2),
                               DataSet = rep(c("Train", "Test"), time = 4),
                               SSE = c(NN1_Train_SSE, NN1_Test_SSE,
                                       NN2_Train_SSE, NN2_Test_SSE,
                                       NN3_Train_SSE, NN3_Test_SSE,
                                       NN4_Train_SSE, NN4_Test_SSE))

Regression_NN_Errors %>%
  ggplot(aes(Network, SSE, fill = DataSet)) +
  geom_col(position = "dodge") +
  ggtitle("Regression ANN's SSE")
```



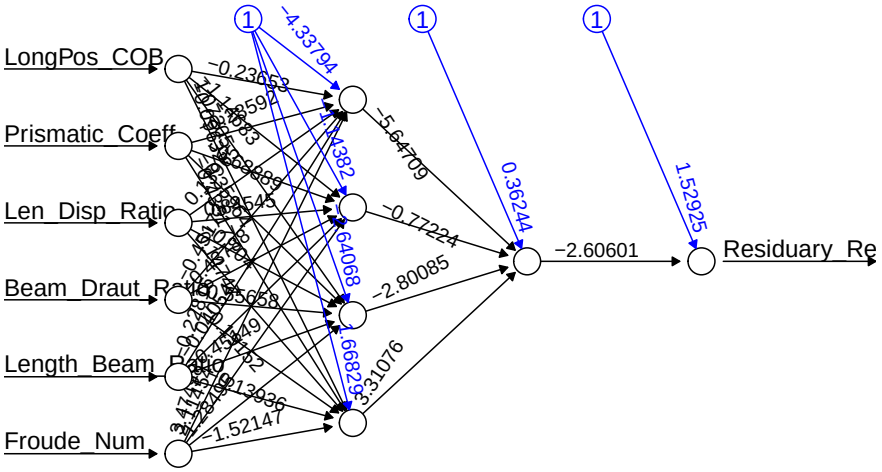
As evident from the plot, we see that the best regression ANN we found was `Yacht_NN2` with a training and test SSE of 0.0188 and 0.0057. We make this determination by the value of the training and test SSEs only. `Yacht_NN2`'s structure is presented here:

```
plot(Yacht_NN2, rep = "best")
```



Error: 0.017855 Steps: 1013

```
set.seed(12321)
Yacht_NN2 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Disp_Ratio + Beam_Draut_Ra
                       data = Yacht_Data_Train,
                       hidden = c(4, 1),
                       act.fct = "logistic",
                       rep = 10)

plot(Yacht_NN2, rep = "best")
```



Error: 0.030513 Steps: 3706

By setting the same seed, prior to running the 10 repetitions of ANNs, we force the software to reproduce the exact same `Yacht_NN2` ANN for the first replication. The subsequent 9 generated ANNs, use a different random set of starting weights. Comparing the 'best' of the 10 repetitions, to the Yacht_NN2, we observe a decrease in training set error indicating we have a superior set of weights.

8.6 Wrapping Up

We have briefly covered regression ANNs in this tutorial. In the next tutorial we will cover classification ANNs. The neuralnet package used in this tutorial is one of many tools available for ANN implementation in R. Others include:

- nnet
- autoencoder
- caret
- RSNNS
- h2o

Before you move on to the next tutorial, test your new knowledge on the exercises that follow.

1. Why do we split the yacht data into a training and test data sets?
2. Re-load the Yacht Data from the UCI Machine learning repository yacht data without scaling. Run any regression ANN. What happens? Why do you think this happens?
3. After completing exercise question 1, re-scale the yacht data. Perform a simple linear regression fitting `Residuary_Resist` as a function of all other features. Now run a regression neural network (see 1st Regression ANN section). Plot the regression ANN and compare the weights on the features in the ANN to the p-values for the regressors.
4. Build your own regression ANN using the scaled yacht data modifying one hyperparameter. Use `?neuralnet` to see the function options. Plot your ANN.

Chapter 9

Regression - cereals dataset

9.1 Introduction

Source: https://www.analyticsvidhya.com/blog/2017/09/creating-visualizing-neural-network-in-r/

Neural network is an information-processing machine and can be viewed as analogous to human nervous system. Just like human nervous system, which is made up of interconnected neurons, a neural network is made up of interconnected information processing units. The information processing units do not work in a linear manner. In fact, neural network draws its strength from parallel processing of information, which allows it to deal with non-linearity. Neural network becomes handy to infer meaning and detect patterns from complex data sets.

Neural network is considered as one of the most useful technique in the world of data analytics. However, it is complex and is often regarded as a black box, i.e. users view the input and output of a neural network but remain clueless about the knowledge generating process. We hope that the article will help readers learn about the internal mechanism of a neural network and get hands-on experience to implement it in R.

9.2 The Basics of Neural Networks

A neural network is a model characterized by an activation function, which is used by interconnected information processing units to transform input into output. A neural network has always been compared to human nervous system. Information in passed through interconnected units analogous to information passage through neurons in humans. The first layer of the neural network receives the raw input, processes it and passes the processed information to the hidden layers. The hidden layer passes the information to the last layer, which produces the output. The advantage of neural network is that it is adaptive in nature. It learns from the information provided, i.e. trains itself from the data, which has a known outcome and optimizes its weights for a better prediction in situations with unknown outcome.

A perceptron, viz. single layer neural network, is the most basic form of a neural network. A perceptron receives multidimensional input and processes it using a weighted summation and an activation function. It is trained using a labeled data and learning algorithm that optimize the weights in the summation processor. A major limitation of perceptron model is its inability to deal with non-linearity. A multilayered neural network overcomes this limitation and helps solve non-linear problems. The input layer connects with hidden layer, which in turn connects to the output layer. The connections are weighted and weights are optimized using a learning rule.

There are many learning rules that are used with neural network:

a) least mean square;

b) gradient descent;
c) newton's rule;
d) conjugate gradient etc.

The learning rules can be used in conjunction with backpropgation error method. The learning rule is used to calculate the error at the output unit. This error is backpropagated to all the units such that the error at each unit is proportional to the contribution of that unit towards total error at the output unit. The errors at each unit are then used to optimize the weight at each connection. Figure 1 displays the structure of a simple neural network model for better understanding.

9.3 Fitting a Neural Network in R

Now we will fit a neural network model in R. In this article, we use a subset of cereal dataset shared by Carnegie Mellon University (CMU). The details of the dataset are on the following link: http://lib.stat.cmu.edu/DASL/Datafiles/Cereals.html. The objective is to predict rating of the cereals variables such as calories, proteins, fat etc. The R script is provided side by side and is commented for better understanding of the user. . The data is in .csv format and can be downloaded by clicking: cereals.

Please set working directory in R using setwd( ) function, and keep cereal.csv in the working directory. We use rating as the dependent variable and calories, proteins, fat, sodium and fiber as the independent variables. We divide the data into training and test set. Training set is used to find the relationship between dependent and independent variables while the test set assesses the performance of the model. We use 60% of the dataset as training set. The assignment of the data to training and test set is done using random sampling. We perform random sampling on R using sample ( ) function. We have used set.seed( ) to generate same random sample everytime and maintain consistency. We will use the index variable while fitting neural network to create training and test data sets. The R script is as follows:

```
## Creating index variable

# Read the Data
data = read.csv(file.path(data_raw_dir, "cereals.csv"), header=T)

# Random sampling
samplesize = 0.60 * nrow(data)
set.seed(80)
index = sample( seq_len ( nrow ( data ) ), size = samplesize )

# Create training and test set
datatrain = data[ index, ]
datatest = data[ -index, ]
```

```
dplyr::glimpse(data)
#> Observations: 75
#> Variables: 6
#> $ calories <int> 70, 120, 70, 50, 110, 110, 130, 90, 90, 120, 110, 120...
#> $ protein  <int> 4, 3, 4, 4, 2, 2, 3, 2, 3, 1, 6, 1, 3, 1, 2, 2, 1, 1,...
#> $ fat      <int> 1, 5, 1, 0, 2, 0, 2, 1, 0, 2, 2, 3, 2, 1, 0, 0, 0, 1,...
#> $ sodium   <int> 130, 15, 260, 140, 180, 125, 210, 200, 210, 220, 290,...
#> $ fiber    <dbl> 10.0, 2.0, 9.0, 14.0, 1.5, 1.0, 2.0, 4.0, 5.0, 0.0, 2...
#> $ rating   <dbl> 68.4, 34.0, 59.4, 93.7, 29.5, 33.2, 37.0, 49.1, 53.3,...
```

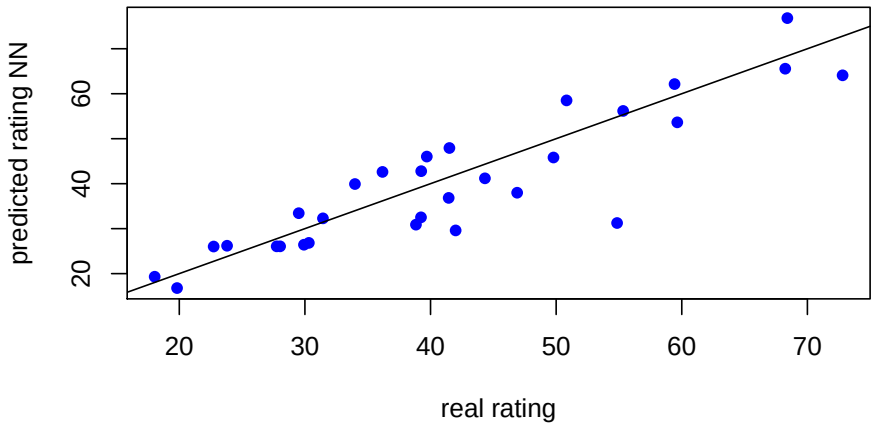
Now we fit a neural network on our data. We use neuralnet library for the analysis. The first step is to scale the cereal dataset. The scaling of data is essential because otherwise a variable may have large impact on the prediction variable only because of its scale. Using unscaled may lead to meaningless results. The

common techniques to scale data are: min-max normalization, Z-score normalization, median and MAD, and tan-h estimators. The min-max normalization transforms the data into a common range, thus removing the scaling effect from all the variables. Unlike Z-score normalization and median and MAD method, the min-max method retains the original distribution of the variables. We use min-max normalization to scale the data. The R script for scaling the data is as follows.

```
## Scale data for neural network

max = apply(data , 2 , max)
min = apply(data, 2 , min)
scaled = as.data.frame(scale(data, center = min, scale = max - min))
```

```
## Fit neural network

# install library
# install.packages("neuralnet ")

# load library
library(neuralnet)

# creating training and test set
trainNN = scaled[index , ]
testNN = scaled[-index , ]

# fit neural network
set.seed(2)
NN = neuralnet(rating ~ calories + protein + fat + sodium + fiber,
               trainNN, hidden = 3 , linear.output = T )

# plot neural network
plot(NN)
```

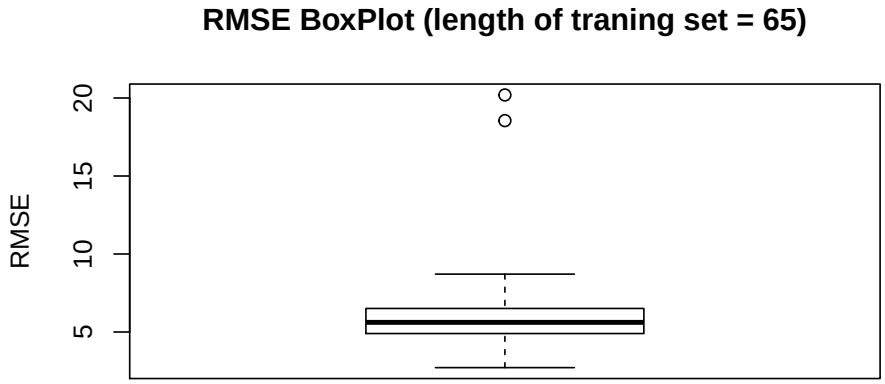
```
## Prediction using neural network

predict_testNN = compute(NN, testNN[,c(1:5)])
predict_testNN = (predict_testNN$net.result * (max(data$rating) - min(data$rating))) + min(data$rating)

plot(datatest$rating, predict_testNN, col='blue', pch=16, ylab = "predicted rating NN", xlab = "real ra

abline(0,1)

# Calculate Root Mean Square Error (RMSE)
RMSE.NN = (sum((datatest$rating - predict_testNN)^2) / nrow(datatest)) ^ 0.5
```




```
## Cross validation of neural network model

# install relevant libraries
# install.packages("boot")
# install.packages("plyr")

# Load libraries
library(boot)
library(plyr)

# Initialize variables
set.seed(50)
k = 100
RMSE.NN = NULL

List = list( )

# Fit neural network model within nested for loop
for(j in 10:65){
    for (i in 1:k) {
        index = sample(1:nrow(data),j )

        trainNN = scaled[index,]
        testNN = scaled[-index,]
        datatest = data[-index,]

        NN = neuralnet(rating ~ calories + protein + fat + sodium + fiber, trainNN, hidden = 3, linear.
        predict_testNN = compute(NN,testNN[,c(1:5)])
        predict_testNN = (predict_testNN$net.result*(max(data$rating)-min(data$rating)))+min(data$ratin

        RMSE.NN [i]<- (sum((datatest$rating - predict_testNN)^2)/nrow(datatest))^0.5
    }
    List[[j]] = RMSE.NN
}

Matrix.RMSE = do.call(cbind, List)
```

```
## Prepare boxplot
boxplot(Matrix.RMSE[,56], ylab = "RMSE", main = "RMSE BoxPlot (length of traning set = 65)")
```

Figure 9.1: Variation of RMSE




```
## Variation of median RMSE
# install.packages("matrixStats")
library(matrixStats)
#>
#> Attaching package: 'matrixStats'
#> The following object is masked from 'package:plyr':
#>
#>     count

med = colMedians(Matrix.RMSE)

X = seq(10,65)

plot (med~X, type = "l", xlab = "length of training set", ylab = "median RMSE", main = "Variation of RMS
```

Figure 9.1) shows that the median RMSE of our model decreases as the length of the training the set. This is an important result. The reader must remember that the model accuracy is dependent on the length of training set. The performance of neural network model is sensitive to training-test split.

9.4 End Notes

The article discusses the theoretical aspects of a neural network, its implementation in R and post training evaluation. Neural network is inspired from biological nervous system. Similar to nervous system the information is passed through layers of processors. The significance of variables is represented by weights of each connection. The article provides basic understanding of back propagation algorithm, which is used to assign these weights. In this article we also implement neural network on R. We use a publically available dataset shared by CMU. The aim is to predict the rating of cereals using information such as calories, fat, protein etc. After constructing the neural network we evaluate the model for accuracy and robustness. We compute RMSE and perform cross-validation analysis. In cross validation, we check the variation in model accuracy as the length of training set is changed. We consider training sets with length 10 to 65. For each length a 100 samples are random picked and median RMSE is calculated. We show that model accuracy increases when training set is large. Before using the model for prediction, it is important to check the robustness of performance through cross validation.

The article provides a quick review neural network and is a useful reference for data enthusiasts. We have provided commented R code throughout the article to help readers with hands on experience of using neural networks.

Chapter 10

Fitting a neural network

10.1 Introduction

https://www.r-bloggers.com/fitting-a-neural-network-in-r-neuralnet-package/

https://datascienceplus.com/fitting-neural-network-in-r/

Neural networks have always been one of the fascinating machine learning models in my opinion, not only because of the fancy backpropagation algorithm but also because of their complexity (think of deep learning with many hidden layers) and structure inspired by the brain.

Neural networks have not always been popular, partly because they were, and still are in some cases, computationally expensive and partly because they did not seem to yield better results when compared with simpler methods such as support vector machines (SVMs). Nevertheless, Neural Networks have, once again, raised attention and become popular.

Update: We published another post about Network analysis at DataScience+ Network analysis of Game of Thrones

In this post, we are going to fit a simple neural network using the neuralnet package and fit a linear model as a comparison.

10.2 The dataset

We are going to use the Boston dataset in the MASS package. The Boston dataset is a collection of data about housing values in the suburbs of Boston. Our goal is to predict the median value of owner-occupied homes (medv) using all the other continuous variables available.

```
set.seed(500)
library(MASS)
data <- Boston
```

```
dplyr::glimpse(data)
#> Observations: 506
#> Variables: 14
#> $ crim    <dbl> 0.00632, 0.02731, 0.02729, 0.03237, 0.06905, 0.02985, ...
#> $ zn      <dbl> 18.0, 0.0, 0.0, 0.0, 0.0, 0.0, 12.5, 12.5, 12.5, 12.5,...
#> $ indus   <dbl> 2.31, 7.07, 7.07, 2.18, 2.18, 2.18, 7.87, 7.87, 7.87, ...
#> $ chas    <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
#> $ nox     <dbl> 0.538, 0.469, 0.469, 0.458, 0.458, 0.458, 0.524, 0.524...
```

```
#> $ rm      <dbl> 6.58, 6.42, 7.18, 7.00, 7.15, 6.43, 6.01, 6.17, 5.63, ...
#> $ age     <dbl> 65.2, 78.9, 61.1, 45.8, 54.2, 58.7, 66.6, 96.1, 100.0,...
#> $ dis     <dbl> 4.09, 4.97, 4.97, 6.06, 6.06, 6.06, 5.56, 5.95, 6.08, ...
#> $ rad     <int> 1, 2, 2, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, ...
#> $ tax     <dbl> 296, 242, 242, 222, 222, 222, 311, 311, 311, 311, 311,...
#> $ ptratio <dbl> 15.3, 17.8, 17.8, 18.7, 18.7, 18.7, 15.2, 15.2, 15.2, ...
#> $ black   <dbl> 397, 397, 393, 395, 397, 394, 396, 397, 387, 387, 393,...
#> $ lstat   <dbl> 4.98, 9.14, 4.03, 2.94, 5.33, 5.21, 12.43, 19.15, 29.9...
#> $ medv    <dbl> 24.0, 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, ...
```

First we need to check that no datapoint is missing, otherwise we need to fix the dataset.

```
apply(data,2,function(x) sum(is.na(x)))
#>    crim      zn   indus    chas     nox      rm     age     dis     rad
#>       0       0       0       0       0       0       0       0       0
#>     tax ptratio   black   lstat    medv
#>       0       0       0       0       0
```

There is no missing data, good. We proceed by randomly splitting the data into a train and a test set, then we fit a linear regression model and test it on the test set. Note that I am using the gml() function instead of the lm() this will become useful later when cross validating the linear model.

```
index <- sample(1:nrow(data),round(0.75*nrow(data)))
train <- data[index,]
test <- data[-index,]
lm.fit <- glm(medv~., data=train)
summary(lm.fit)
#>
#> Call:
#> glm(formula = medv ~ ., data = train)
#>
#> Deviance Residuals:
#>     Min       1Q   Median       3Q      Max
#> -15.211   -2.559   -0.655    1.828   29.711
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  31.11170    5.45981    5.70  2.5e-08 ***
#> crim         -0.11137    0.03326   -3.35  0.00090 ***
#> zn            0.04263    0.01431    2.98  0.00308 **
#> indus         0.00148    0.06745    0.02  0.98247
#> chas          1.75684    0.98109    1.79  0.07417 .
#> nox         -18.18485    4.47157   -4.07  5.8e-05 ***
#> rm            4.76034    0.48047    9.91  < 2e-16 ***
#> age          -0.01344    0.01410   -0.95  0.34119
#> dis          -1.55375    0.21893   -7.10  6.7e-12 ***
#> rad           0.28818    0.07202    4.00  7.6e-05 ***
#> tax          -0.01374    0.00406   -3.38  0.00079 ***
#> ptratio      -0.94755    0.14012   -6.76  5.4e-11 ***
#> black         0.00950    0.00290    3.28  0.00115 **
#> lstat        -0.38890    0.05973   -6.51  2.5e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> (Dispersion parameter for gaussian family taken to be 20.2)
```

```
#>
#>     Null deviance: 32463.5  on 379  degrees of freedom
#> Residual deviance:  7407.1  on 366  degrees of freedom
#> AIC: 2237
#>
#> Number of Fisher Scoring iterations: 2
pr.lm <- predict(lm.fit,test)
MSE.lm <- sum((pr.lm - test$medv)^2)/nrow(test)
```

The sample(x,size) function simply outputs a vector of the specified size of randomly selected samples from the vector x. By default the sampling is without replacement: index is essentially a random vector of indeces. Since we are dealing with a regression problem, we are going to use the mean squared error (MSE) as a measure of how much our predictions are far away from the real data.

10.3 Preparing to fit the neural network

Before fitting a neural network, some preparation need to be done. Neural networks are not that easy to train and tune.

As a first step, we are going to address data preprocessing. It is good practice to normalize your data before training a neural network. I cannot emphasize enough how important this step is: depending on your dataset, avoiding normalization may lead to useless results or to a very difficult training process (most of the times the algorithm will not converge before the number of maximum iterations allowed). You can choose different methods to scale the data (z-normalization, min-max scale, etc...). I chose to use the min-max method and scale the data in the interval [0,1]. Usually scaling in the intervals [0,1] or [-1,1] tends to give better results. We therefore scale and split the data before moving on:

```
maxs <- apply(data, 2, max)
mins <- apply(data, 2, min)

scaled <- as.data.frame(scale(data, center = mins, scale = maxs - mins))

train_ <- scaled[index,]
test_ <- scaled[-index,]
```

Note that scale returns a matrix that needs to be coerced into a data.frame.

10.4 Parameters

As far as I know there is no fixed rule as to how many layers and neurons to use although there are several more or less accepted rules of thumb. Usually, if at all necessary, one hidden layer is enough for a vast numbers of applications. As far as the number of neurons is concerned, it should be between the input layer size and the output layer size, usually 2/3 of the input size. At least in my brief experience testing again and again is the best solution since there is no guarantee that any of these rules will fit your model best. Since this is a toy example, we are going to use 2 hidden layers with this configuration: 13:5:3:1. The input layer has 13 inputs, the two hidden layers have 5 and 3 neurons and the output layer has, of course, a single output since we are doing regression. Let's fit the net:

```
library(neuralnet)
n <- names(train_)
f <- as.formula(paste("medv ~", paste(n[!n %in% "medv"], collapse = " + ")))
nn <- neuralnet(f,data=train_,hidden=c(5,3),linear.output=T)
```

A couple of notes:

- For some reason the formula y~. is not accepted in the neuralnet() function. You need to first write the formula and then pass it as an argument in the fitting function.
- The hidden argument accepts a vector with the number of neurons for each hidden layer, while the argument linear.output is used to specify whether we want to do regression linear.output=TRUE or classification linear.output=FALSE

The `neuralnet` package provides a nice tool to plot the model:

This is the graphical representation of the model with the weights on each connection:

```
plot(nn)
```

The black lines show the connections between each layer and the weights on each connection while the blue lines show the bias term added in each step. The bias can be thought as the intercept of a linear model. The net is essentially a black box so we cannot say that much about the fitting, the weights and the model. Suffice to say that the training algorithm has converged and therefore the model is ready to be used.

10.5 Predicting medv using the neural network

Now we can try to predict the values for the test set and calculate the MSE. Remember that the net will output a normalized prediction, so we need to scale it back in order to make a meaningful comparison (or just a simple prediction).

```
pr.nn <- compute(nn,test_[,1:13])

pr.nn_ <- pr.nn$net.result*(max(data$medv)-min(data$medv))+min(data$medv)
test.r <- (test_$medv)*(max(data$medv)-min(data$medv))+min(data$medv)

MSE.nn <- sum((test.r - pr.nn_)^2)/nrow(test_)
```

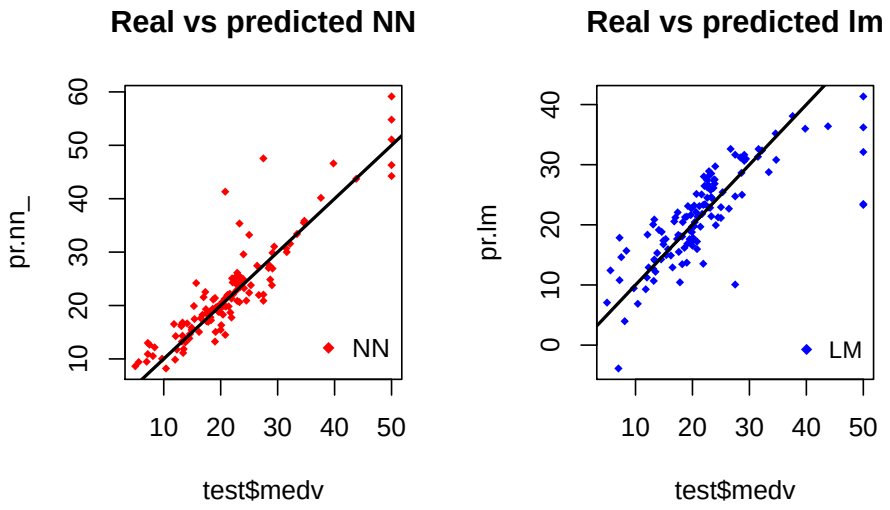
we then compare the two MSEs

```
print(paste(MSE.lm,MSE.nn))
#> [1] "31.2630222372615 16.4595537665717"
```

Apparently, the net is doing a better work than the linear model at predicting medv. Once again, be careful because this result depends on the train-test split performed above. Below, after the visual plot, we are going to perform a fast cross validation in order to be more confident about the results.

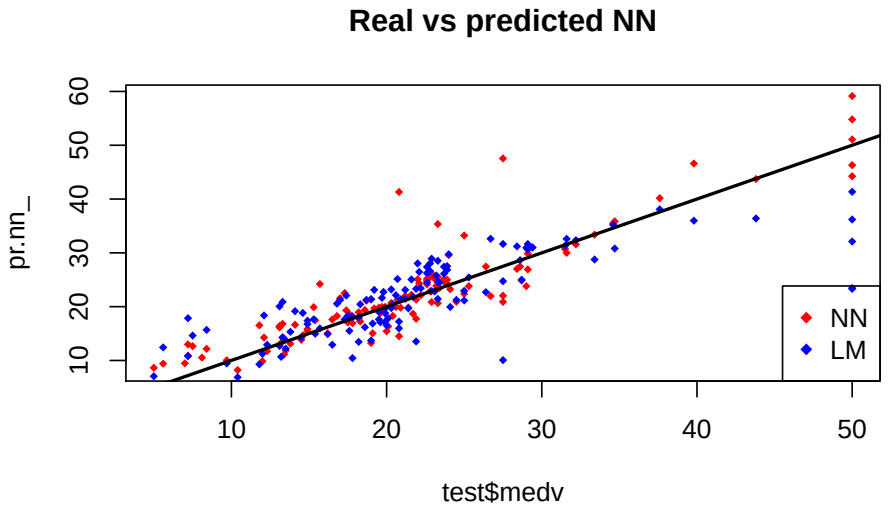
A first visual approach to the performance of the network and the linear model on the test set is plotted below

```
par(mfrow=c(1,2))

plot(test$medv,pr.nn_,col='red',main='Real vs predicted NN',pch=18,cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend='NN',pch=18,col='red', bty='n')

plot(test$medv,pr.lm,col='blue',main='Real vs predicted lm',pch=18, cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend='LM',pch=18,col='blue', bty='n', cex=.95)
```



By visually inspecting the plot we can see that the predictions made by the neural network are (in general) more concetrated around the line (a perfect alignment with the line would indicate a MSE of 0 and thus an ideal perfect prediction) than those made by the linear model.

```
plot(test$medv,pr.nn_,col='red',main='Real vs predicted NN',pch=18,cex=0.7)
points(test$medv,pr.lm,col='blue',pch=18,cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend=c('NN','LM'),pch=18,col=c('red','blue'))
```



10.6 A (fast) cross validation

Cross validation is another very important step of building predictive models. While there are different kind of cross validation methods, the basic idea is repeating the following process a number of time:

train-test split

- Do the train-test split
- Fit the model to the train set
- Test the model on the test set
- Calculate the prediction error
- Repeat the process K times

Then by calculating the average error we can get a grasp of how the model is doing.

We are going to implement a fast cross validation using a for loop for the neural network and the cv.glm() function in the boot package for the linear model. As far as I know, there is no built-in function in R to perform cross-validation on this kind of neural network, if you do know such a function, please let me know in the comments. Here is the 10 fold cross-validated MSE for the linear model:

```
library(boot)
set.seed(200)
lm.fit <- glm(medv~.,data=data)
cv.glm(data,lm.fit,K=10)$delta[1]
#> [1] 23.2
```

Now the net. Note that I am splitting the data in this way: 90% train set and 10% test set in a random way for 10 times. I am also initializing a progress bar using the plyr library because I want to keep an eye on the status of the process since the fitting of the neural network may take a while.

```
set.seed(450)
cv.error <- NULL
k <- 10

library(plyr)
pbar <- create_progress_bar('text')
pbar$init(k)
#>
  |
  |                                                                      |   0%

for(i in 1:k){
    index <- sample(1:nrow(data),round(0.9*nrow(data)))
    train.cv <- scaled[index,]
    test.cv <- scaled[-index,]

    nn <- neuralnet(f,data=train.cv,hidden=c(5,2),linear.output=T)

    pr.nn <- compute(nn,test.cv[,1:13])
    pr.nn <- pr.nn$net.result*(max(data$medv)-min(data$medv))+min(data$medv)

    test.cv.r <- (test.cv$medv)*(max(data$medv)-min(data$medv))+min(data$medv)

    cv.error[i] <- sum((test.cv.r - pr.nn)^2)/nrow(test.cv)

    pbar$step()
}
#>
  |
  |======                                                                |  10%
  |
  |=============                                                         |  20%
  |
  |====================                                                  |  30%
  |
  |==========================                                            |  40%
  |
  |================================                                      |  50%
  |
```

```
  |=====================================                                |  60%
  |
  |=============================================                        |  70%
  |
  |=====================================================                |  80%
  |
  |=================================================================    |  90%
  |
  |=====================================================================| 100%
```

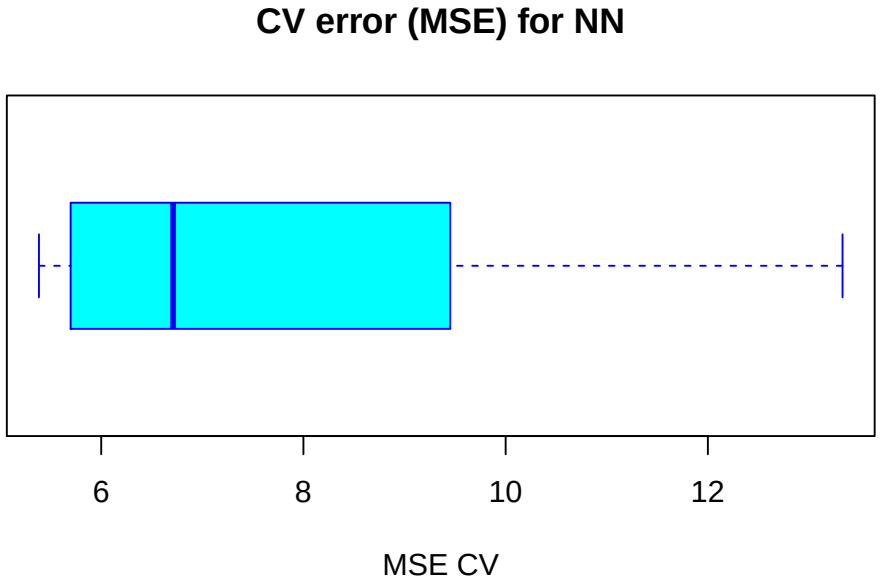
After a while, the process is done, we calculate the average MSE and plot the results as a boxplot

```
mean(cv.error)
#> [1] 7.64
```

```
cv.error
#>  [1] 13.33  7.10  6.58  5.70  6.84  5.77 10.75  5.38  9.45  5.50
```

The code for the box plot: The code above outputs the following boxplot:

```
boxplot(cv.error,xlab='MSE CV',col='cyan',
        border='blue',names='CV error (MSE)',
        main='CV error (MSE) for NN',horizontal=TRUE)
```



As you can see, the average MSE for the neural network (10.33) is lower than the one of the linear model although there seems to be a certain degree of variation in the MSEs of the cross validation. This may depend on the splitting of the data or the random initialization of the weights in the net. By running the simulation different times with different seeds you can get a more precise point estimate for the average MSE.

10.7 A final note on model interpretability

Neural networks resemble black boxes a lot: explaining their outcome is much more difficult than explaining the outcome of simpler model such as a linear model. Therefore, depending on the kind of application you need, you might want to take into account this factor too. Furthermore, as you have seen above, extra care is needed to fit a neural network and small changes can lead to different results.

A gist with the full code for this post can be found here.

Thank you for reading this post, leave a comment below if you have any question.

Chapter 11

Visualization of neural networks

https://beckmw.wordpress.com/tag/neuralnet/

In my last post I said I wasn't going to write anymore about neural networks (i.e., multilayer feedforward perceptron, supervised ANN, etc.). That was a lie. I've received several requests to update the neural network plotting function described in the original post. As previously explained, R does not provide a lot of options for visualizing neural networks. The only option I know of is a plotting method for objects from the neuralnet package. This may be my opinion, but I think this plot leaves much to be desired (see below). Also, no plotting methods exist for neural networks created in other packages, i.e., nnet and RSNNS. These packages are the only ones listed on the CRAN task view, so I've updated my original plotting function to work with all three. Additionally, I've added a new option for plotting a raw weight vector to allow use with neural networks created elsewhere. This blog describes these changes, as well as some new arguments added to the original function.

As usual, I'll simulate some data to use for creating the neural networks. The dataset contains eight input variables and two output variables. The final dataset is a data frame with all variables, as well as separate data frames for the input and output variables. I've retained separate datasets based on the syntax for each package.

```
library(clusterGeneration)
#> Loading required package: MASS
library(tictoc)

seed.val<- 12345
set.seed(seed.val)

num.vars<-8
num.obs<-1000

# input variables
cov.mat <-genPositiveDefMat(num.vars,covMethod=c("unifcorrmat"))$Sigma
rand.vars <-mvrnorm(num.obs,rep(0,num.vars),Sigma=cov.mat)

# output variables
parms <-runif(num.vars,-10,10)
y1 <- rand.vars %*% matrix(parms) + rnorm(num.obs,sd=20)
parms2 <- runif(num.vars,-10,10)
y2 <- rand.vars %*% matrix(parms2) + rnorm(num.obs,sd=20)

# final datasets
```

```
rand.vars <- data.frame(rand.vars)
resp <- data.frame(y1,y2)
names(resp) <- c('Y1','Y2')
dat.in <- data.frame(resp, rand.vars)
```

```
dplyr::glimpse(dat.in)
#> Observations: 1,000
#> Variables: 10
#> $ Y1 <dbl> 25.442, -14.578, -36.214, 15.216, -6.393, -20.849, -28.665,...
#> $ Y2 <dbl> 16.9, 38.8, 31.2, -31.2, 93.3, 11.7, 59.7, -103.5, -49.8, 5...
#> $ X1 <dbl> 3.138, -0.705, -4.373, 0.837, 0.787, 1.923, -1.419, 1.121, ...
#> $ X2 <dbl> 0.195, -0.302, 0.773, 1.311, 3.506, 1.245, 3.800, -0.165, 0...
#> $ X3 <dbl> -1.795, -2.596, 2.308, 4.081, -3.921, 1.473, -0.926, 7.101,...
#> $ X4 <dbl> -2.7216, 3.0589, 1.2455, 3.4607, 2.3775, -2.9833, 2.6669, -...
#> $ X5 <dbl> 0.0407, 0.7602, -3.0217, -4.2799, 2.0859, 1.4765, 0.0561, 2...
#> $ X6 <dbl> -1.4820, -0.5014, 0.0603, -1.8551, 2.2817, 1.7386, 1.7450, ...
#> $ X7 <dbl> -0.7169, -0.3618, -1.5283, 4.2026, -6.1548, -0.3545, -6.028...
#> $ X8 <dbl> 1.152, 1.810, -1.357, 0.598, -1.425, -1.210, -1.004, 2.494,...
```

The various neural network packages are used to create separate models for plotting.

```
# first model with nnet
#nnet function from nnet package
library(nnet)
set.seed(seed.val)
tic()
mod1 <- nnet(rand.vars, resp, data = dat.in, size = 10, linout = T)
#> # weights:  112
#> initial  value 4784162.893260
#> iter  10 value 1794537.980652
#> iter  20 value 1577753.498759
#> iter  30 value 1485254.945755
#> iter  40 value 1449238.248788
#> iter  50 value 1427720.291804
#> iter  60 value 1416977.236373
#> iter  70 value 1405167.753521
#> iter  80 value 1395046.792257
#> iter  90 value 1370522.267277
#> iter 100 value 1363709.540981
#> final  value 1363709.540981
#> stopped after 100 iterations
toc()
#> 0.201 sec elapsed
```

```
# nn <- neuralnet(form.in,
#                 data = dat.sc,
#                 # hidden = c(13, 10, 3),
#                 hidden = c(5),
#                 act.fct = "tanh",
#                 linear.output = FALSE,
#                 lifesign = "minimal")
```

```
# 2nd model with neuralnet
# neuralnet function from neuralnet package, notice use of only one response
library(neuralnet)
```

```
softplus <- function(x) log(1 + exp(x))
sigmoid  <- function(x) log(1 + exp(-x))

dat.sc <- scale(dat.in)
form.in <- as.formula('Y1 ~ X1+X2+X3+X4+X5+X6+X7+X8')
set.seed(seed.val)
tic()
mod2 <- neuralnet(form.in, data = dat.sc, hidden = 10, lifesign = "minimal",
                  linear.output = FALSE,
                  act.fct = "tanh")
#> hidden: 10    thresh: 0.01    rep: 1/1    steps:   26361 error: 160.06372    time: 59.35 secs
toc()
#> 59.356 sec elapsed
```

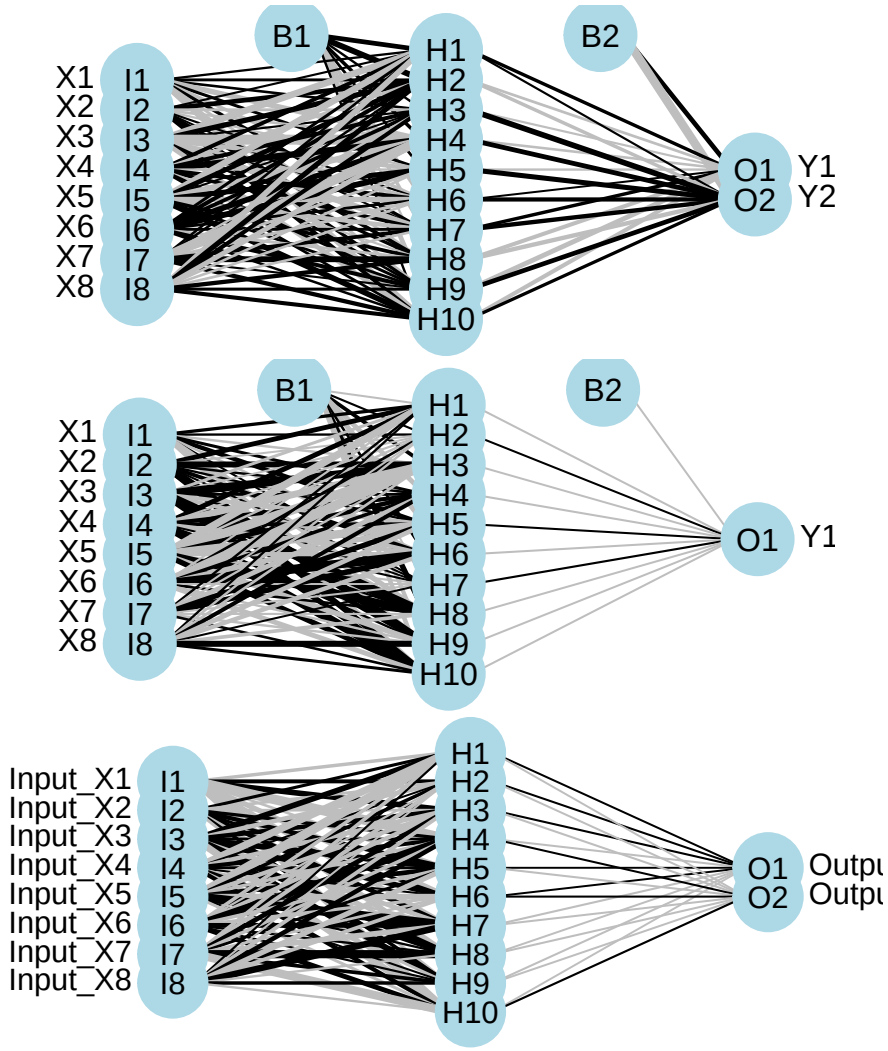
```
# third model with RSNNS
# mlp function from RSNNS package
library(RSNNS)
#> Loading required package: Rcpp
set.seed(seed.val)
tic()
mod3 <- mlp(rand.vars, resp, size = 10, linOut = T)
toc()
#> 0.339 sec elapsed
```

I've noticed some differences between the functions that could lead to some confusion. For simplicity, the above code represents my interpretation of the most direct way to create a neural network in each package. Be very aware that direct comparison of results is not advised given that the default arguments differ between the packages. A few key differences are as follows, although many others should be noted. First, the functions differ in the methods for passing the primary input variables.

The `nnet` function can take separate (or combined) x and y inputs as data frames or as a formula, the `neuralnet` function can only use a formula as input, and the mlp function can only take a data frame as combined or separate variables as input. As far as I know, the neuralnet function is not capable of modelling multiple response variables, unless the response is a categorical variable that uses one node for each outcome. Additionally, the default output for the neuralnet function is linear, whereas the opposite is true for the other two functions.

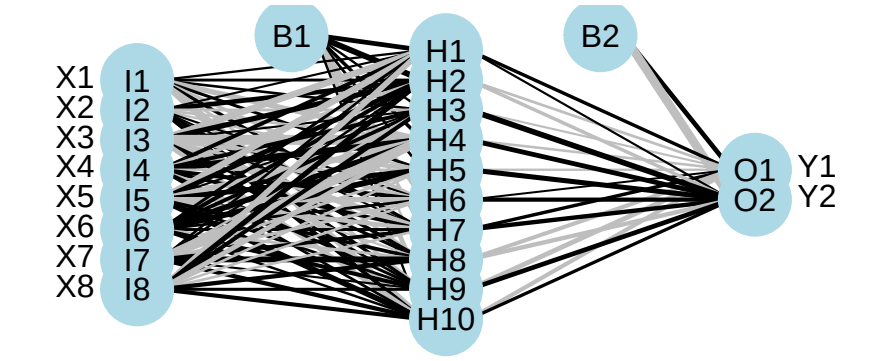
Specifics aside, here's how to use the updated plot function. Note that the same syntax is used to plot each model

```
# import the function from Github
library(devtools)
source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/466c1474d0a505ff044412703516c34f1a4
#> SHA-1 hash of file is 74c80bd5ddbc17ab3ae5ece9c0ed9beb612e87ef

# plot each model
plot.nnet(mod1)
#> Loading required package: scales
#> Loading required package: reshape
plot.nnet(mod2)
plot.nnet(mod3)
#> Warning in plot.nnet(mod3): Bias layer not applicable for rsnns object
```



The plotting function can also now be used with an arbitrary weight vector, rather than a specific model object. The struct argument must also be included if this option is used. I thought the easiest way to use the plotting function with your own weights was to have the input weights as a numeric vector, including bias layers. I've shown how this can be done using the weights directly from mod1 for simplicity.

```
wts.in <- mod1$wts
struct <- mod1$n
plot.nnet(wts.in,struct=struct)
```



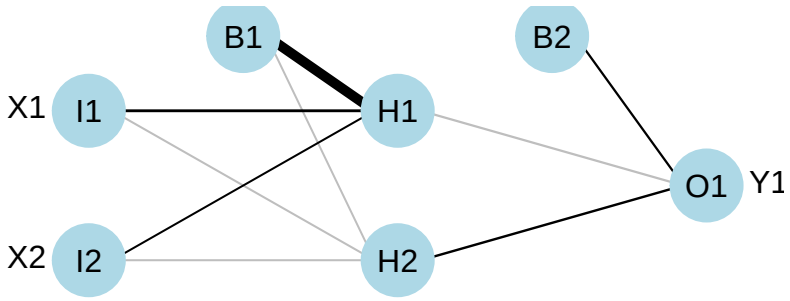
Note that `wts.in` is a numeric vector with length equal to the expected given the architecture (i.e., for 8 10

2 network, 100 connection weights plus 12 bias weights). The plot should look the same as the plot for the neural network from nnet.

The weights in the input vector need to be in a specific order for correct plotting. I realize this is not clear by looking directly at wt.in but this was the simplest approach I could think of. The weight vector shows the weights for each hidden node in sequence, starting with the bias input for each node, then the weights for each output node in sequence, starting with the bias input for each output node. Note that the bias layer has to be included even if the network was not created with biases. If this is the case, simply input a random number where the bias values should go and use the argument bias=F. I'll show the correct order of the weights using an example with plot.nn from the neuralnet package since the weights are included directly on the plot.

If we pretend that the above figure wasn't created in R, we would input the mod.in argument for the updated plotting function as follows. Also note that struct must be included if using this approach.

```
mod.in<-c(13.12,1.49,0.16,-0.11,-0.19,-0.16,0.56,-0.52,0.81)
struct<-c(2,2,1) #two inputs, two hidden, one output
plot.nnet(mod.in, struct=struct)
```



Note the comparability with the figure created using the neuralnet package. That is, larger weights have thicker lines and color indicates sign (+ black, – grey).

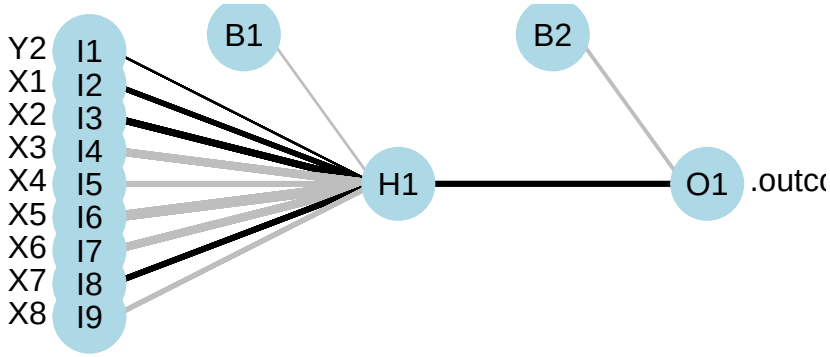
One of these days I'll actually put these functions in a package. In the meantime, please let me know if any bugs are encountered.

11.1 caret and plot NN

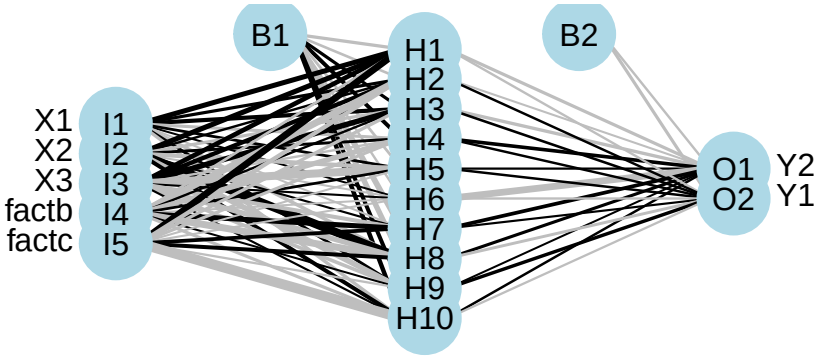
I've changed the function to work with neural networks created using the train function from the caret package. The link above is updated but you can also grab it here.

```
library(caret)
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#>
#> Attaching package: 'caret'
#> The following objects are masked from 'package:RSNNS':
#>
#>     confusionMatrix, train
mod4 <- train(Y1 ~., method='nnet', data=dat.in, linout=T)
```

```
plot.nnet(mod4,nid=T)
#> Warning in plot.nnet(mod4, nid = T): Using best nnet model from train
#> output
```




```
fact<-factor(sample(c('a','b','c'),size=num.obs,replace=T))
form.in<-formula('cbind(Y2,Y1)~X1+X2+X3+fact')
mod5<-nnet(form.in,data=cbind(dat.in,fact),size=10,linout=T)
#> # weights:  82
#> initial  value 4799569.423556
#> iter  10 value 2864553.218126
#> iter  20 value 2595828.194160
#> iter  30 value 2517965.483941
#> iter  40 value 2464882.178217
#> iter  50 value 2444238.700834
#> iter  60 value 2424302.290643
#> iter  70 value 2395226.949866
#> iter  80 value 2375558.751266
#> iter  90 value 2343011.050867
#> iter 100 value 2298860.593948
#> final  value 2298860.593948
#> stopped after 100 iterations
plot.nnet(mod5,nid=T)
```

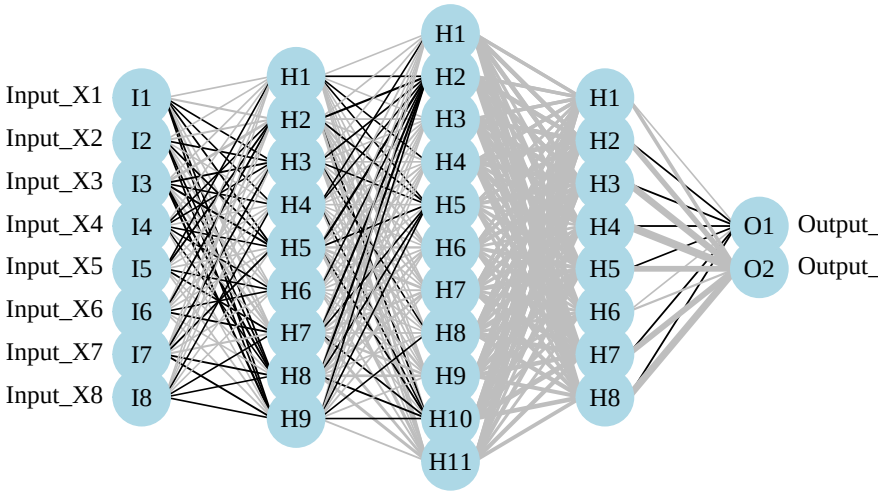


11.2 Multiple hidden layers

More updates… I've now modified the function to plot multiple hidden layers for networks created using the mlp function in the `RSNNS` package and neuralnet in the neuralnet package. As far as I know, these are the only neural network functions in R that can create multiple hidden layers. All others use a single hidden layer. I have not tested the plotting function using manual input for the weight vectors with multiple hidden layers.

My guess is it won't work but I can't be bothered to change the function unless it's specifically requested. The updated function can be grabbed here (all above links to the function have also been changed).

```
library(RSNNS)

# neural net with three hidden layers, 9, 11, and 8 nodes in each
tic()
mod <-mlp(rand.vars, resp,
          size = c(9,11,8),
          linOut = T)
toc()
#> 0.381 sec elapsed
par(mar=numeric(4),family='serif')
plot.nnet(mod)
#> Warning in plot.nnet(mod): Bias layer not applicable for rsnns object
```

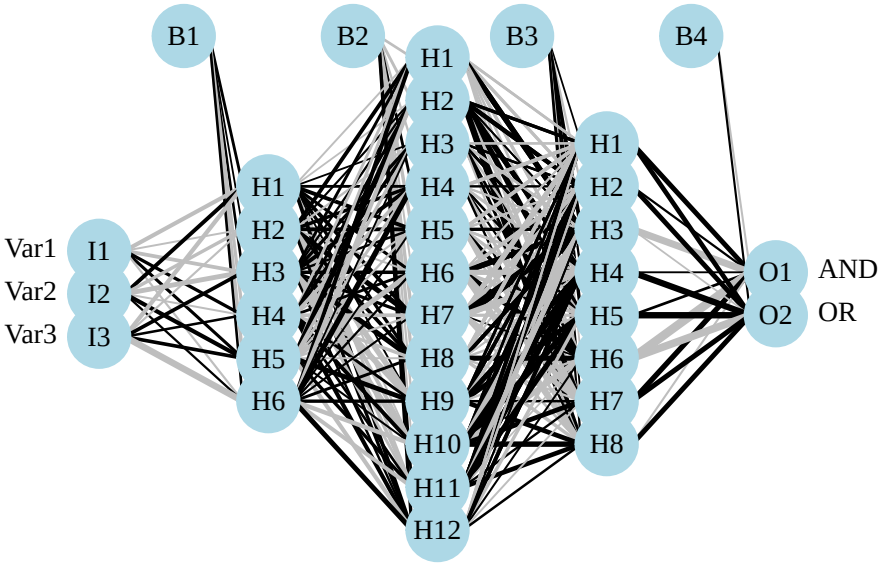


11.3 Binary predictors

Here's an example using the `neuralnet` function with binary predictors and categorical outputs (credit to Tao Ma for the model code).

```
library(neuralnet)

#response
AND<-c(rep(0,7),1)
OR<-c(0,rep(1,7))

# response with predictors
binary.data <- data.frame(expand.grid(c(0,1), c(0,1), c(0,1)), AND, OR)

#model
tic()
net <- neuralnet(AND+OR ~ Var1+Var2+Var3,
                 binary.data, hidden =c(6,12,8),
                 rep = 10,
                 err.fct="ce",
                 linear.output=FALSE)
```

```
toc()
#> 0.107 sec elapsed
#plot ouput
par(mar=numeric(4),family='serif')
plot.nnet(net)
```



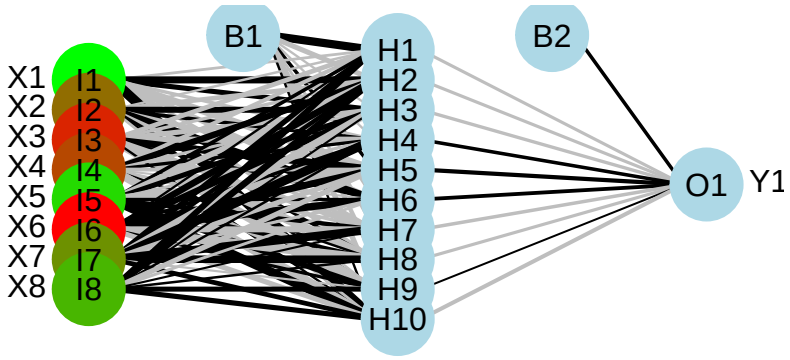
11.4 color coding the input layer

The color vector argument (circle.col) for the nodes was changed to allow a separate color vector for the input layer.

The following example shows how this can be done using relative importance of the input variables to color-code the first layer.

```
# example showing use of separate colors for input layer
# color based on relative importance using 'gar.fun'

##
#create input data
seed.val<-3
set.seed(seed.val)

num.vars<-8
num.obs<-1000

#input variables
library(clusterGeneration)
cov.mat<-genPositiveDefMat(num.vars,covMethod=c("unifcorrmat"))$Sigma
rand.vars<-mvrnorm(num.obs,rep(0,num.vars),Sigma=cov.mat)

# output variables
parms<-runif(num.vars,-10,10)
y1<-rand.vars %*% matrix(parms) + rnorm(num.obs,sd=20)
```

```
# final datasets
rand.vars<-data.frame(rand.vars)
resp<-data.frame(y1)
names(resp)<-'Y1'
dat.in <- data.frame(resp,rand.vars)

##
# create model
library(nnet)
mod1 <- nnet(rand.vars,resp,data=dat.in,size=10,linout=T)
#> # weights:  101
#> initial  value 844959.580478
#> iter  10 value 543616.101824
#> iter  20 value 479986.887846
#> iter  30 value 465607.784054
#> iter  40 value 454237.073298
#> iter  50 value 445032.412421
#> iter  60 value 433191.158624
#> iter  70 value 426321.161292
#> iter  80 value 424900.966883
#> iter  90 value 423816.437605
#> iter 100 value 422064.114812
#> final  value 422064.114812
#> stopped after 100 iterations

##
# relative importance function
library(devtools)
source_url('https://gist.github.com/fawda123/6206737/raw/2e1bc9cbc48d1a56d2a79dd1d33f414213f5f1b1/gar_fu
#> SHA-1 hash of file is 9faa58824c46956c3ff78081696290d9b32d845f

# relative importance of input variables for Y1
rel.imp <- gar.fun('Y1',mod1,bar.plot=F)$rel.imp

#color vector based on relative importance of input values
cols<-colorRampPalette(c('green','red'))(num.vars)[rank(rel.imp)]

##
#plotting function
source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/466c1474d0a505ff044412703516c34f1a4
#> SHA-1 hash of file is 74c80bd5ddbc17ab3ae5ece9c0ed9beb612e87ef

#plot model with new color vector
#separate colors for input vectors using a list for 'circle.col'
plot(mod1,circle.col=list(cols,'lightblue'))
```


B1
B2
X1
X2
X3
X4
X5
X6
X7
X8
I1
I2
I3
I4
I5
I6
I7
I8
H1
H2
H3
H4
H5
H6
H7
H8
H9
H10
O1
Y1