ACTL4305/5305: Week 9 Lab - Neural Network (Solution)

Week 9

Learning Objectives

- Training neural networks in R using various packages, for both classification and regression tasks.
- Enhancing neural network performance through tuning: comprehending the associated tuning parameters and employing techniques such as early stopping, dropout, or weight regularization.

1 Neural Network

This week covers two case studies: one focused on a regression problem and the other on a classification problem. We will introduce you to two packages for implementing neural networks:

- `neuralnet()`: This package is considered classical and is particularly well-suited for beginners due to its simplicity and user-friendliness. (from `neuralnet` package)
- `keras` package: Originally developed in Python, the R implementation of the `keras` package provides a user-friendly interface for designing and training neural networks, along with extensive customization options.

Additionally, there are other valuable packages for working with neural networks, including `h2o` and `caret`.

2 A Regression Problem

2.1 Data Manipulation

The data we use is a subset of `freMTPL2freq` from `CASdatasets`.

In the dataset `freMTPL2freq` risk features and claim numbers were collected for 677991 motor third-part liability policies (observed on a year). We only consider a subset of `freMTPL2freq` including 40000 observations for training and 10000 observations for testing. Our task is to predict the number of claims.

`freMTPL2freq` contains 11 columns (with `IDpol`):

- `IDpol` The policy ID (used to link with the claims dataset).
- `ClaimNb` Number of claims during the exposure period.
- `Exposure` The exposure period.
- `Area` The area code.
- `VehPower` The power of the car (ordered categorical).

- `VehAge` The vehicle age, in years.
- `DrivAge` The driver age, in years (in France, people can drive a car at 18).
- `BonusMalus` Bonus/malus, between 50 and 350: 100 means malus in France.
- `VehBrand` The car brand (unknown categories).
- `VehGas` The car gas, Diesel or regular.
- `Density` The density of inhabitants (number of inhabitants per km2) in the city the driver of the car lives in.
- `Region` The policy regions in France (based on a standard French classification)

2.1.1 Import Data

```
library(neuralnet) #neural network (slow)
library(tidyverse)
library(ROCR) #AUC plot
library(kableExtra) # Tables
```

```
load("Train-set.RData")
load("Test-set.RData")

traindata<-newtrain
testdata<-newtest
#str(traindata)
```

2.1.2 Data normalization for numeric variables

One of the most important procedures when forming a neural network is data normalization. This involves adjusting the data to a common scale so as to accurately compare predicted and actual values. Failure to normalize the data will typically result in the prediction value remaining the same across all observations, regardless of the input values.

We can do this in two ways in R:

- Scale the data frame automatically using the scale function in R.
- Transform the data using a max-min normalization technique.

We can implement both techniques, but choose to use the **max-min normalization** technique. By doing max-min normalization, all numeric variables will be in the range [0,1]. The reason we deal with the test data differently is that we can't use the information from the test set (treat it as unknown).

```
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
} #for training set, x is the variable.

normalizetest <- function(x,y) {
  return ((x - min(y)) / (max(y) - min(y)))
}
```

```
#for test set, x is the variable in test set, y is the according variable in the training set.

#Exposure
testdata$Exposure <- normalizetest(testdata$Exposure,traindata$Exposure)
traindata$Exposure <- normalize(traindata$Exposure)

#DrivAge
testdata$DrivAge <- normalizetest(testdata$DrivAge,traindata$DrivAge)
traindata$DrivAge <- normalize(traindata$DrivAge)

#BonusMalus
testdata$BonusMalus <- normalizetest(testdata$BonusMalus,traindata$BonusMalus)
traindata$BonusMalus <- normalize(traindata$BonusMalus)

#Density (take log here)
testdata$Density <- normalizetest(log(testdata$Density),log(traindata$Density))
traindata$Density <- normalize(log(traindata$Density))

#VehPower
testdata$VehPower <- normalizetest(testdata$VehPower,traindata$VehPower)
traindata$VehPower <- normalize(traindata$VehPower)

#VehAge
testdata$VehAge <- normalizetest(testdata$VehAge,traindata$VehAge)
traindata$VehAge <- normalize(traindata$VehAge)

#ClaimNb (response variable), we only scale it for neural network. So add a column.
traindata <- traindata %>% mutate(ClaimNb_nn =normalize(traindata$ClaimNb))
```

2.1.3 Dummy coding for categorical variables

The idea here is to transform each categorical variable to several dummy variables (0 or 1).

```
Dummy <- function(var1, short, dat2){
   names(dat2)[names(dat2) == var1]  <- "V1"
   n2 <- ncol(dat2)
   dat2$X <- as.integer(dat2$V1)
   n0 <- length(unique(dat2$X))
   for (n1 in 2:n0){dat2[, paste(short, n1, sep="")] <- as.integer(dat2$X==n1)}
   names(dat2)[names(dat2) == "V1"]  <- var1
   dat2[, c(1:n2,(n2+2):ncol(dat2))]
}

#Area
unique(traindata$Area)
```

```
## [1] A E C D B F
## Levels: A B C D E F
```

```
traindata<-Dummy("Area","ar",traindata) # ar1=A, ar2=B, ar3=C, ar4=D, ar5=E, ar6=F.
testdata<-Dummy("Area","ar",testdata)
```

```
#VehBrand
unique(traindata$VehBrand)
```

```
##  [1] B1  B2  B13 B3  B4  B12 B11 B10 B5  B14 B6
## Levels: B1 B10 B11 B12 B13 B14 B2 B3 B4 B5 B6
```

```
traindata<-Dummy("VehBrand","vb",traindata)#vb1=B1, vb2=B10, vb3=B11, vb4=B12,
#vb5=B13, vb6=B14,
testdata<-Dummy("VehBrand","vb",testdata) #vb7=B2, vb8=B3, vb9=B4, vb10=B5, vb11=B6

#VehGas
unique(traindata$VehGas)
```

```
## [1] Regular Diesel
## Levels: Diesel Regular
```

```
traindata<-Dummy("VehGas","vg",traindata) # vg1=Diesel, vg2=Regular
testdata<-Dummy("VehGas","vg",testdata)
```

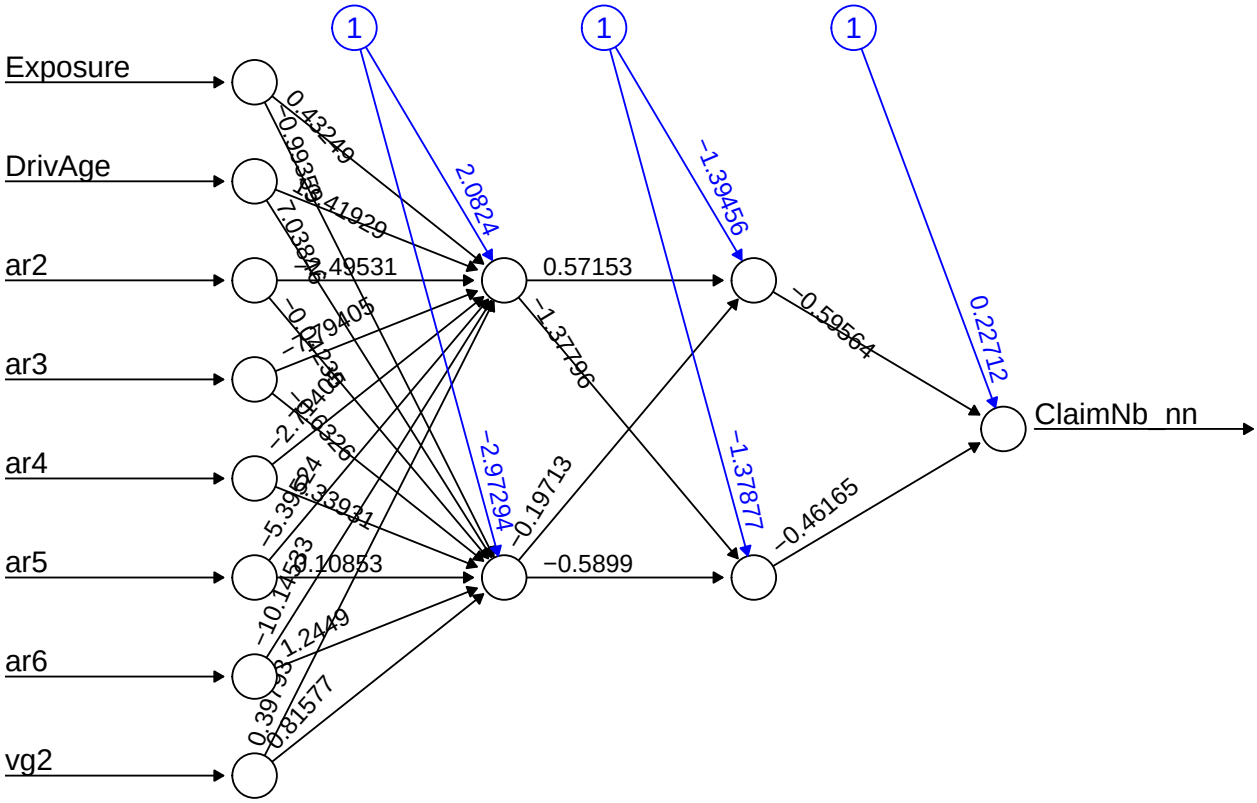
2.2 Training a Neural Network Model

To train a neural network, we use `neuralnet` function.

Notes:

- We use `neuralnet` to 'regress' the dependent `ClaimNb` variable against the other independent variables. Here we should consider feature selection problem.
- Setting the number of hidden layers to (2,2) based on the `hidden=(2,2)` formula.
- The `linear.output` variable is set to `TRUE` for regression problem. For classification problem, set it to be `FALSE`. If it is `FALSE`, then you can set the **activation function** by `act.fct`. The activation function can be 'logistic' for the logistic function and 'tanh' for tangent hyperbolicus.
- `err.fct` defines a differentiable function that is used for the calculation of the error. 'sse' and 'ce' which stand for the sum of squared errors and the cross-entropy can be used.
- The `threshold` is set to 0.05, meaning that if the change in error during an iteration is less than 5%, then no further optimization will be carried out by the model (`stepmax` is another stopping criteria).
- There are several types of `algorithm` you can use. e.g. 'backprop' refers to backpropagation, 'rprop+' and 'rprop-' refer to the resilient backpropagation with and without weight backtracking,
- Deciding on the number of hidden layers in a neural network is not an exact science. In fact, there are instances where accuracy will likely be higher without any hidden layers. Therefore, trial and error plays a significant role in this process. One possibility is to compare how the accuracy of the predictions change as we modify the number of hidden layers.

```
#neural network model. Note response variable!
nn=neuralnet(ClaimNb_nn~Exposure+DrivAge+ar2+ar3+ar4+ar5+ar6+vg2, data=traindata,
             hidden=c(2,2), linear.output=TRUE, threshold=0.05, algorithm='rprop+')
#nn$result.matrix

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



Error: 172.515865 Steps: 505

```
#glm model. Note response variable!
nnglm<-glm(ClaimNb~Exposure+DrivAge+ar2+ar3+ar4+ar5+ar6+vg2, data=traindata,
           family=poisson(link = "log"))
```

2.3 Testing The Accuracy Of The Model

As already mentioned, our neural network has been created using the training data. We then compare this to the test data to gauge the prediction of the neural network. Note for neural network, we need to back-scale the predicted values to **its original scale**!

```
#Test the resulting output
Xtest <- select(testdata,c("Exposure","DrivAge","ar2","ar3","ar4","ar5","ar6","vg2"))

nn.results <- neuralnet::compute(nn, Xtest) #predicted values
original_nn<-(nn.results$net.result)*(max(traindata$ClaimNb)-min(traindata$ClaimNb))+ min(traindata$Cla
# back-scale: use max and min of response variable in training set. Think about the reason!

glm.results<-predict(nnglm,newdata = Xtest,type =  "response")

results <- data.frame(actual = testdata$ClaimNb,
                      nn.prediction = original_nn,
                      glm.prediction =glm.results)
```

```r
# test mse
(mse<-data.frame(nn.mse=sum((results[,1]-results[,2])^2)/length(results[,2]),
                 glm.mse=sum((results[,1]-results[,3])^2)/length(results[,2])))
```

```
##       nn.mse   glm.mse
## 1 0.04917203 0.0488829
```

```r
#Neural network is worse...think about reasons?

# We haven't tuned parameters!
```

3 A Classification Problem (Credit Risk Modeling)

In this section, we will revisit the credit dataset introduced in the Week 3 Lab. Our goal is to train neural networks, and for this, we'll use the `tensorflow` and `keras` packages. TensorFlow serves as a powerful open-source platform for machine learning, and Keras complements it with its flexibility and user-friendly interface, offering an extensive range of neural network architectures and customization options. We will kick off our exploration with a simple architecture.

3.1 Installation of Required Packages

1. **Install Python 3.9.10**: Both `keras` and `tensorflow` depend on Python. Ensure that you install an older version of Python to maintain compatibility with both packages and other related dependencies. Do not opt for the latest version. From my experience, Python 3.9.10 works well. You can download it either directly from this Python's official website or specifically version 3.9.10 from this direct link. Ensure you choose the version appropriate for your operating system.

    - Alternatively, you can use the `install_python()` function from the `reticulate` package. Make sure the `reticulate` package is installed to proceed:

```
#reticulate::install_python()
```

2. **Install Visual Studio 2015, 2017, 2019, and 2022 (for Windows Only)** : Download and install from this link.

3. **Installation and Setup in R**:

    - First, run the R chunk below to install and load the `reticulate` package, which acts as a bridge, enabling R to communicate with Python.
    - Next, to integrate TensorFlow and Keras into R, first install their R packages. Once done, load their respective libraries. The `install_tensorflow()` command is used to install the tensorflow python package and its direct dependencies. A similar procedure applies to the `keras` package using `install_keras()`.

```
# This code chunk won't be executed but will be displayed
install.packages("reticulate")
install.packages("tensorflow")
install.packages("keras")

library(reticulate)
library(tensorflow)

install_tensorflow()

library(keras)

install_keras()
```

After executing the above commands, TensorFlow and Keras should be integrated into your R environment, ready for use. If you encounter any issues, it is possible there were prior installations of Python or related software/packages that could be causing conflicts. Reinstalling them could potentially resolve these issues.

3.2 Data Preparation

```
library(keras)
library(tensorflow)

load("train_credit.RData") #70%
load("test_credit.RData") #30%

num_var<-c(1,5,12:23)

train_data_label <- as.numeric(xtrain0$default)-1

train_feature <- xtrain0[, -24]
test_feature <- xtest0[, -24]

# Create & standardize feature sets
mean    <- colMeans(train_feature[,num_var])
std     <- apply(train_feature[,num_var], 2, sd)
train_feature[,num_var] <- scale(train_feature[,num_var], center = mean, scale = std)
test_feature[,num_var] <- scale(test_feature[,num_var], center = mean, scale = std)
```

Instead of employing the max-min normalization technique, we opt for standardizing the numerical features using the `scale` function this time. It is important to note that we standardize our test feature sets based on the mean and standard deviation of the training features. This approach helps to minimize the potential for data leakage.

3.3 Developing the Network Architecture

```
train_feature <- as.data.frame(model.matrix( ~ 0 +. ,data = train_feature))
test_feature <- as.data.frame(model.matrix( ~ 0 +. ,data = test_feature))

# Define the model architecture
nn1 <- keras_model_sequential() %>%
  layer_dense(units = 20, activation = "relu", input_shape = ncol(train_feature)) %>%
  layer_dense(units = 10, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

In the `keras` package, we can construct our neural network using a layering approach. Here, our model is structured as a sequential neural network, and we define it using the `keras_model_sequential()` function. When designing the architecture of a neural network, two key components require your attention:

1. Layers and Nodes
2. Activation Functions

3.3.1 Layers and Nodes

This network architecture includes two hidden layers: the first layer with 20 nodes and the second layer with 10 nodes. In the first hidden layer, we explicitly define the number of nodes in the input layer using the `input_shape` parameter. It is important to note that the `input_shape` argument should match the number

of features in your dataset. However, the successive layers are able to dynamically interpret the number of expected inputs based on the previous layer. As we are dealing with a binary classification problem, the output layer comprises a single node.

3.3.2 Activation Functions

Keras offers compatibility with a range of activation functions (for a detailed list, refer to activation functions). Typically, all hidden layers within a neural network employ the same activation function. However, the choice of activation function for the output layer differs and depends on the specific prediction task.

In our case, we have employed the rectified linear unit (ReLU) function, which serves as the modern default activation function for hidden layers. For the output layer, we have implemented the sigmoid function, a suitable choice for our binary classification task.

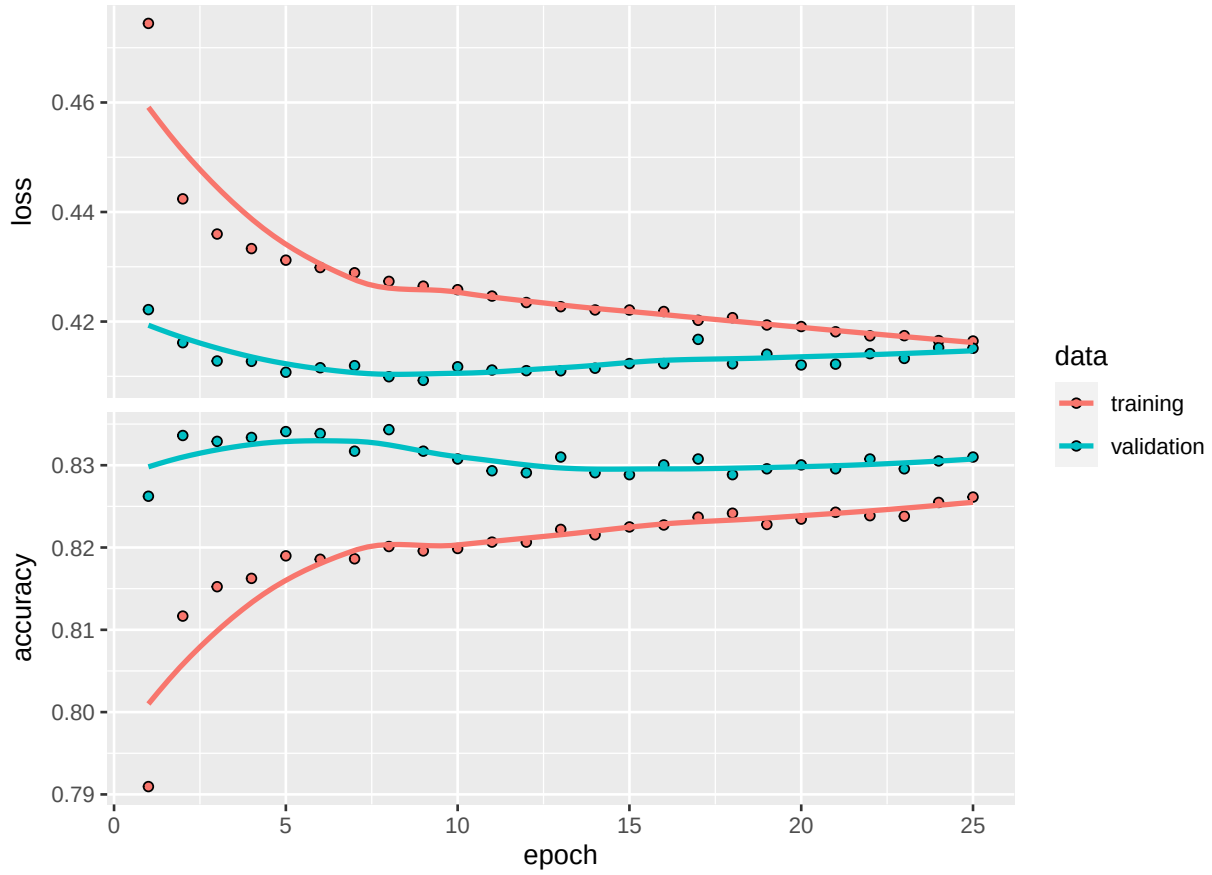
3.4 Compiling the Model with Appropriate Configuration

```
# Compile the model
nn1 %>% compile(loss = "binary_crossentropy",
                optimizer = "adam",
                metrics = c("accuracy"))
```

Before fitting the model, we compile it, specifying the loss function, optimizer, and metrics used for performance evaluation. In this case, we have selected the Adam optimizer, a popular choice in deep learning. Additionally, it is important to note that a metric is a function that is used to judge the performance of your model. Metric functions are similar to loss functions, except that the results from evaluating a metric are not used when training the model. For our metric, we use accuracy. However, you are encouraged to adjust this according to the specific requirements and context of your task.

3.5 Model Training

```
# Fit the model to the training data
train_feature_matrix <- as.matrix(train_feature)

history1 <- nn1 %>% fit(train_feature_matrix, train_data_label,
                        epochs = 25, batch_size = 32, validation_split = 0.2)

plot(history1)
```



Now that we have created our base model, it is time to train it using our data. We use `fit` to estimate network parameters. Here we establish that the feature data are `train_feature_matrix` and output data are `train_data_label`. We are using mini-batches of `batch_size` 32, which are iterated over 25 times — corresponding to the specified number of `epochs`. Additionally, we allocate 20% of the training data as a validation split by setting `validation_split = 0.2`. The training history is stored in the `history1` variable, and we plot the training process.

```
library(PRROC)
library(caret)

test_feature_matrix <- as.matrix(test_feature)

nn1_probpred <- predict(nn1, test_feature_matrix)
nn1_auc <- roc.curve(scores.class0 = nn1_probpred,
                          weights.class0 = as.numeric(xtest0$default)-1, curve = T)
nn1_auc$auc
```

```
## [1] 0.7685382
```

```
nn1_pred <- ifelse(nn1_probpred > 0.5, 1, 0)
nn1_conf <- confusionMatrix(as.factor(nn1_pred),  xtest0$default, positive="1")
```

Lastly, we proceed to evaluate the model's fit using the classification metrics introduced in Week 7. The resulting metrics are as follows for `nn1`:

- Accuracy: 0.819091
- AUC: 0.7685382
- Sensitivity: 0.3261307
- Specificity: 0.9590526
- F-score: 0.443609

It is essential to recognize that this comparison might not be equitable for determining the best-performing model. This week, we trained the model on the complete dataset, whereas in Weeks 7 and 8, we used subset samples solely for illustrative purposes. This distinction must be kept in mind while drawing conclusions regarding model performance.

3.6 Model Tuning

Exercise: Model Tuning: For further exploration, the provided template code below presents a neural network with a few additional adjustments. These adjustments include implementing an early stopping callback, incorporating batch normalization, and introducing a `dropout` rate of 20%. Please feel free to modify and experiment with the code to explore various alterations to the neural network's architecture. Some potential adjustments involve:

- Modifying the model's capacity by adjusting the number of layers and nodes in each layer.
- Adjusting the number of epochs for training or incorporating an early stopping callback that halts training if the loss function fails to improve after a specified number of epochs.
- Adding and experimenting with other customization options such as batch normalization, dropout, and weight regularization. It is important to note that not all of these options need to be added, and some may serve the same purpose (preventing overfitting).
- Adjusting the learning rate to a suitable value for efficient training.

You can refer to the UC Business Analytics R Programming Guide for additional references on R implementations.

```
# This code chunk won't be executed but will be displayed
#Exercise
# Define the model architecture
nn2 <- keras_model_sequential() %>%
  layer_dense(units = 20, activation = "relu", input_shape = ncol(train_feature)) %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 10, activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 1, activation = "sigmoid")

# Compile the model
nn2 %>% compile(loss = "binary_crossentropy", optimizer = "adam", metrics = c("accuracy"))

history2 <- nn2 %>% fit(train_feature_matrix, train_data_label,
                    epochs = 50, batch_size = 32, validation_split = 0.2,
                    callbacks = list(callback_early_stopping(patience = 5)))

plot(history2)

nn2_probpred <- predict(nn2, test_feature_matrix)
```

```
nn2_auc <- roc.curve(scores.class0 = nn2_probpred,
                     weights.class0 = as.numeric(xtest0$default)-1, curve = T)
nn2_auc$auc

nn2_pred <- ifelse(nn2_probpred > 0.5, 1, 0)
nn2_conf <- confusionMatrix(as.factor(nn2_pred),  xtest0$default, positive="1")
```

4 Training with ANN Package (an alternative for credit data)

Given that `keras` and `tensorflow` are dependent on their Python counterparts, they sometimes encounter compatibility issues with other Python packages, which can be challenging to resolve. As an alternative, we present another R package, `ANN2`, designed for rapid neural network training.

4.1 Data Manipulation

Our first step is to scale the numerical variables by **max-min normalization** technique and dummy code the categorical variables.

```
library(ANN2) #neural network (fast)

load("train_credit.RData") #70%
load("test_credit.RData") #30%

num_var<-c(1,5,12:23)
train_cre<-xtrain0
test_cre<-xtest0

# now we try to scale all numerical variables by a for-loop
for  (i in seq(num_var)) {
test_cre[num_var[i]] <- normalizetest(test_cre[num_var[i]],train_cre[num_var[i]])
#scale predictors in test set. Here, remember scale test set first!!!
train_cre[num_var[i]] <- normalize(train_cre[num_var[i]])
#scale predictors in train set

}

#str(train_cre)
#response variable, should be numeric
train_cre$default<-ifelse(train_cre$default==0,0,1)
test_cre$default<-ifelse(test_cre$default==0,0,1)
```

Then we manipulate categorical variables by dummy coding.

```
#SEX
unique(train_cre$SEX)
```

```
## [1] Female Male
## Levels: Female Male
```

```
train_cre<-Dummy("SEX","sex",train_cre) # sex1= Female, sex2=Male
test_cre<-Dummy("SEX","sex",test_cre)

#EDUCATION
unique(train_cre$EDUCATION)
```

```
## [1] 2 1 3 4
## Levels: 1 2 3 4
```

```
train_cre<-Dummy("EDUCATION","edu",train_cre) #edu1=1,edu2=2,edu3=3,edu4=4
test_cre<-Dummy("EDUCATION","edu",test_cre)

#MARRIAGE
unique(train_cre$MARRIAGE)
```

```
## [1] 2 1 3
## Levels: 1 2 3
```

```
train_cre<-Dummy("MARRIAGE","mar",train_cre) #mar1=1,mar2=2,mar3=3
test_cre<-Dummy("MARRIAGE","mar",test_cre)
```

4.2 Training a Neural Network Model

4.2.1 neuralnetwork function

We don't use `neuralnet()` in this case because it is old and slow. We will use `neuralnetwork()`, which is more than 10 times faster.

Notes:

- `maxEpochs`: the maximum number of epochs (one iteration through training data).
- `batchSize`: the number of observations to use in each batch. Batch learning is computationally faster than stochastic gradient descent. However, large batches might not result in optimal learning, see Le Cun for details.
- `L1`: L1 regularization. Non-negative number. Set to zero for no regularization. `L2` :L2 regularization. Non-negative number. Set to zero for no regularization.
- `lossFunction`: which loss function should be used. Options are "log", "quadratic", "absolute", "huber" and "pseudo-huber".
- `regression`: logical indicating regression or classification. In case of `TRUE` (regression), the activation function in the last hidden layer will be the linear activation function (identity function). In case of `FALSE` (classification), the activation function in the last hidden layer will be the softmax, and the log loss function should be used.
- `standardize`: logical indicating if X and y should be standardized before training the network. Recommended to leave at TRUE for faster convergence.
- `learn.rates`: the size of the steps made in gradient descent. If set too large, optimization can become unstable. Is set too small, convergence will be slow.
- `optim.type`: type of optimizer to use for updating the parameters. Options are 'sgd', 'rmsprop' and 'adam'. SGD is implemented with momentum. If you concern the result is not stable, then you can try different type of optimizer to ensure the convergence.
- `activ.functions`: character vector of activation functions to be used in each hidden layer. Possible options are 'tanh', 'sigmoid', 'relu', 'linear', 'ramp' and 'step'. Should be either the size of the number of hidden layers or equal to one. If a single activation type is specified, this type will be broadcasted across the hidden layers.

```
#str(train_cre)
#neural network model
#nn_cre=neuralnet(default~LIMIT_BAL+AGE
 #                 +BILL_AMT1+BILL_AMT2+BILL_AMT3+BILL_AMT4+BILL_AMT5+BILL_AMT6
 #                 +PAY_AMT1+PAY_AMT2+PAY_AMT3+PAY_AMT4+PAY_AMT5+PAY_AMT6
 #                +sex2+edu2+edu3+edu4+mar2+mar3, data=train_cre,
 #               hidden=c(5,3), linear.output=FALSE, threshold=0.1,
#act.fct = "logistic",stepmax=10^6,lifesign='full',lifesign.step=1000)
#plot(nn_cre)

# classification task by neuralnetwork, the package name is ANN2.
NN <- neuralnetwork(X = train_cre[,c(1,5,12:23,25:30)],
                    y = train_cre[,24], hidden.layers = c(5, 3),optim.type = 'adam',
                    learn.rates = 0.005,val.prop = 0,regression=FALSE,
                    verbose=FALSE, random.seed = 2020)

NN_L2<-neuralnetwork(X = train_cre[,c(1,5,12:23,25:30)],
                    y = train_cre[,24], hidden.layers = c(5, 3),optim.type = 'adam',
                    learn.rates = 0.005, val.prop = 0,regression=FALSE,verbose=FALSE,
                    L2=0.3,random.seed = 2020)

#glm model
nn_creglm<-glm(default~LIMIT_BAL+AGE
                 +PAY_AMT1+BILL_AMT1
                 +sex2+edu2+edu3+edu4+mar2+mar3,
               data=train_cre,family=binomial)
```

4.3 Testing The Accuracy Of The Model

4.3.1 Prediction on default probability

```
#neuralnet
#predict_crenn <- neuralnet::compute(nn_cre,test_cre)

# neuralnetwork
y_pred <- predict(NN, newdata = test_cre[,c(1,5,12:23,25:30)])
y_predL2<-predict(NN_L2, newdata = test_cre[,c(1,5,12:23,25:30)])
#glm
predict_creglm <- predict(nn_creglm, newdata = test_cre, type = "response")

results_cre <- data.frame(actual = test_cre$default,
                          neuralnet = y_pred$probabilities[,2], glm=predict_creglm)
```

4.3.2 Classifiction

Now, we convert probabilities into binary classes (default or non-default).

```
# Converting probabilities into binary classes setting threshold level 0.3
threshold=0.3
```

```
class_cre <- data.frame(actual = test_cre$default,
                        neuralnetwork = ifelse(results_cre[,2]>threshold, 1, 0),
                        glm=ifelse(results_cre[,3]>threshold, 1, 0))

table(class_cre[,1],class_cre[,2])
```

```
##
##        0    1
##   0 5533 1476
##   1 1019  971
```

```
table(class_cre[,1],class_cre[,3])
```

```
##
##        0    1
##   0 6011  998
##   1 1483  507
```

Actually, the `threshold` is a hyperparameter which can be decided by cross validation and bootstrap. It is also decided by other external factors like how much risk a bank is will to take or profitability or even regulatory requirements.

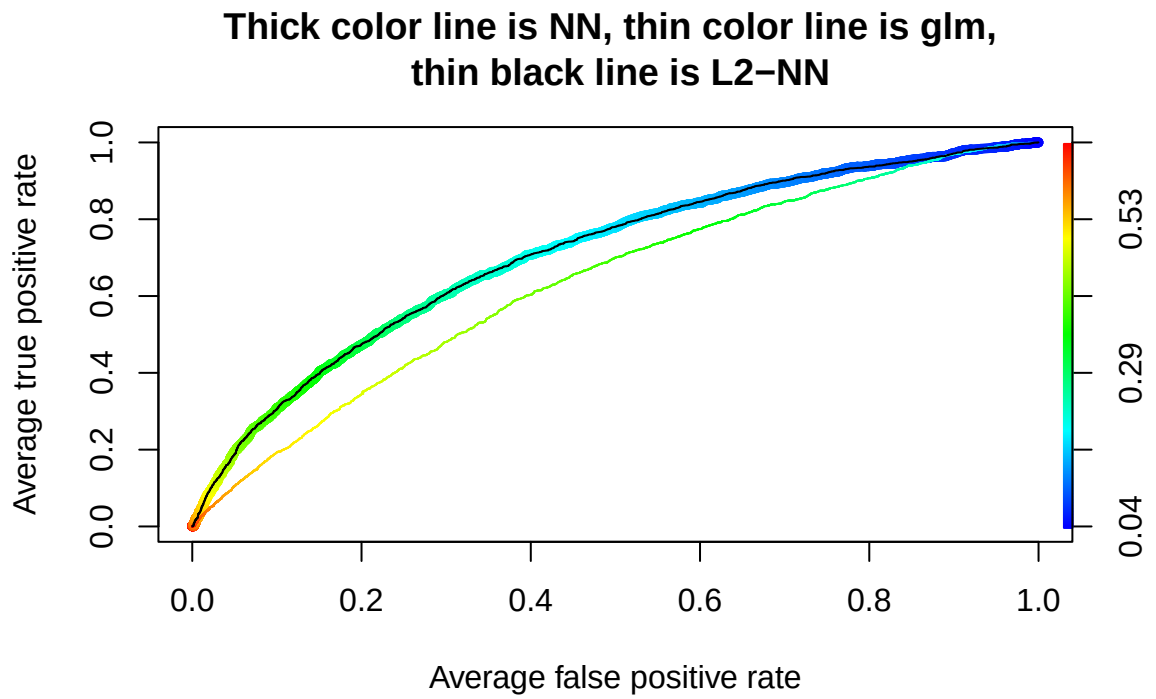
Let's see the AUC curve.

```
#ROCRpred_nn<- prediction(predict_crenn$net.result, test_cre$default)
#ROCRperf_nn <- performance(ROCRpred_nn, 'tpr', 'fpr')

ROCRpred_glm<- prediction(predict_creglm, test_cre$default)
ROCRperf_glm<- performance(ROCRpred_glm, 'tpr', 'fpr')
auc_glm <- performance(ROCRpred_glm, measure = "auc")

ROCRpred_Ann<- prediction(y_pred$probabilities[,2], test_cre$default)
ROCRperf_Ann <- performance(ROCRpred_Ann, 'tpr', 'fpr')
auc_Ann <- performance(ROCRpred_Ann, measure = "auc")


ROCRpred_AnnL2<- prediction(y_predL2$probabilities[,2], test_cre$default)
ROCRperf_AnnL2 <- performance(ROCRpred_AnnL2, 'tpr', 'fpr')
auc_AnnL2 <- performance(ROCRpred_AnnL2, measure = "auc")


plot(ROCRperf_Ann,
     avg= "threshold",
     colorize=TRUE,
     lwd= 5,
     main= "Thick color line is NN, thin color line is glm, \n thin black line is L2-NN")

plot(ROCRperf_glm,
     avg= "threshold",
     colorize=TRUE,
     lwd= 1, add=TRUE)
```

Table 1: Evaluation of different models using AUC

Methods	AUC
GLM	0.6330120
NN	0.7064756
NN-L2	0.7067783

```
plot(ROCRperf_AnnL2,
     avg= "threshold",
     lwd= 1, add=TRUE)
```




```
#plot(ROCRperf_nn, avg= "threshold",colorize=FALSE,lwd = 5,add=TRUE)
```

We can see the AUC values of three models in Table 1.

We can see that the neural net work with L2 regularization achieves better prediction.