Recognising Daily and Sports Activities from movement sensor data

Akshay Ithape 19200976

30/4/2020

Abstract:

Motion sensor recognition is a rapidly growing field that has been in place nowadays. The data are motion sensor measurements of 19 daily and sports activities, each performed by 8 subjects (4 females, 4 males, between the ages 20 and 30) in their own style for 5 minutes. For each subject, five sensor units are used to record the movement on the torso, arms, and legs. Each unit is constituted by 9 sensors: x-y-z accelerometers, x-y-z gyroscopes, and x-y-z magnetometers. We have deployed a neural network with two hidden layers in order to predict the activity being performed using the motion sensor measurements. We have successfully discovered a predictive model with an accuracy of approximately 95%.

Introduction:

We have data of motion sensor measurements of 19 daily and sports activities, each performed by 8 subjects (4 females, 4 males, between the ages 20 and 30) in their own style for 5 minutes. For each subject, five sensor units are used to record the movement on the torso, arms, and legs. Each unit is constituted by 9 sensors: x-y-z accelerometers, x-y-z gyroscopes, and x-y-z magnetometers. Sensor units are calibrated to acquire data at 25 Hz sampling frequency. The 5-minute signals are divided into 5second signal segments, so that a total of 480 signal segments are obtained for each activity, thus 480 × 19 = 9120 signal segments classified into 19 classes. Each signal segment is divided into 125 sampling instants recorded using 5 × 9 = 45 sensors.

Hence, each signal segment is represented as a 125 × 45 matrix, where columns contains the 125 samples of data acquired from one of the sensors of one of the units over a period of 5 seconds, and rows contain data acquired from all of the 45 sensors at a particular sampling instant. For each signal matrix: columns 1-9 correspond to the sensors in the torso unit, columns 10-18 correspond to the sensors in right arm unit, columns 19-27 correspond to the sensors in the left arm unit, columns 28-36 correspond to the sensors in the right leg unit, and columns 37-45 correspond to the sensors in the left leg unit. For each set of 9 sensors, the first three are accelerometers, the second three are gyroscopes and the last three magnetometers.

What are we trying to predict?

Our main goal is to deploy a predictive model which can be employed for daily/sports activity recognition from movement sensor data. This model is to be selected from various model configurations and selecting the one that is giving us the best results with minimal complexity.

How is answering this question going to help us?

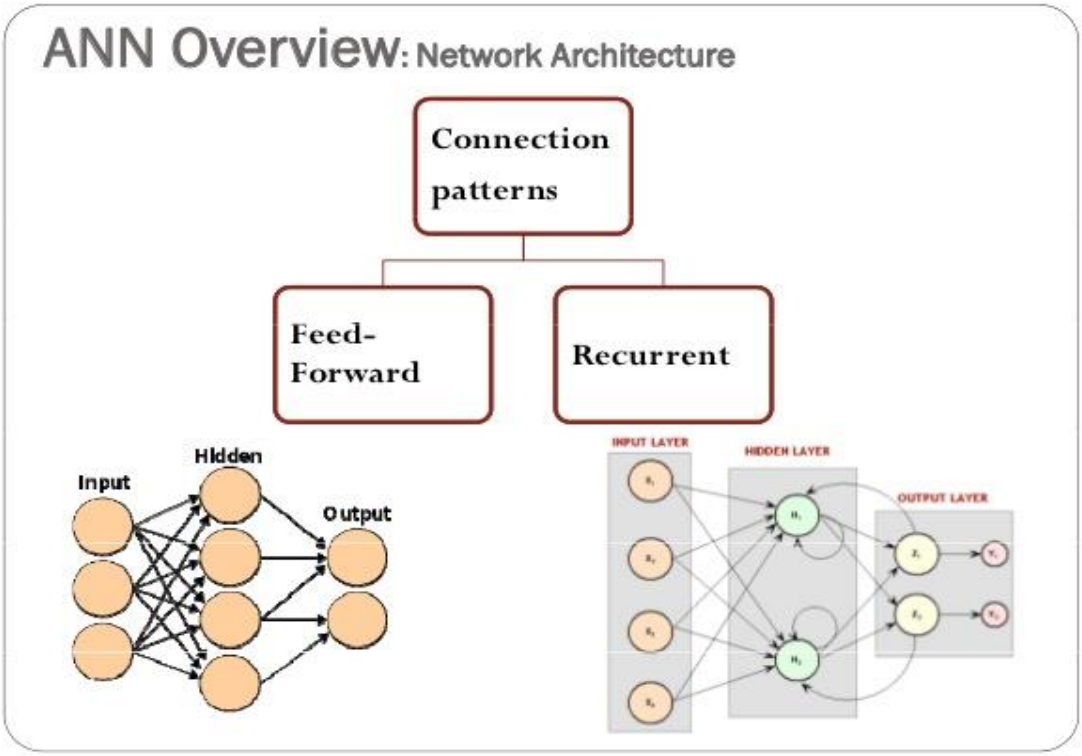
As mentioned above, fitness in today's world plays a major role in an individual's life. This model can be useful for predictions using various movements, which exercise is best suited for which body part and selective plans can be drawn out for an individual which is best suited based on the exercise that he/she wish to follow.

What type of data do we have?

The data file data_activity_recognition.RData contains training and test data. The training data x_train includes 400 signal segments for each activity (total of 7600 signals), while the test data x_test includes 80 signal segments for each activity (totalof1520signals). The activity labels are in the objects y_train and y_test, respectively. The input data are organized in the form of 3-dimensional arrays, in which the first dimension denote the signal, the second the sampling instants and the third the sensors, so each signal is described by a vector of 125 × 45 = 5625 features.

Model Representation:

Artificial Neural Network is computing system inspired by biological neural network that constitute animal brain. Such systems "learn" to perform tasks by considering examples, generally without being programmed with any task-specific rules.

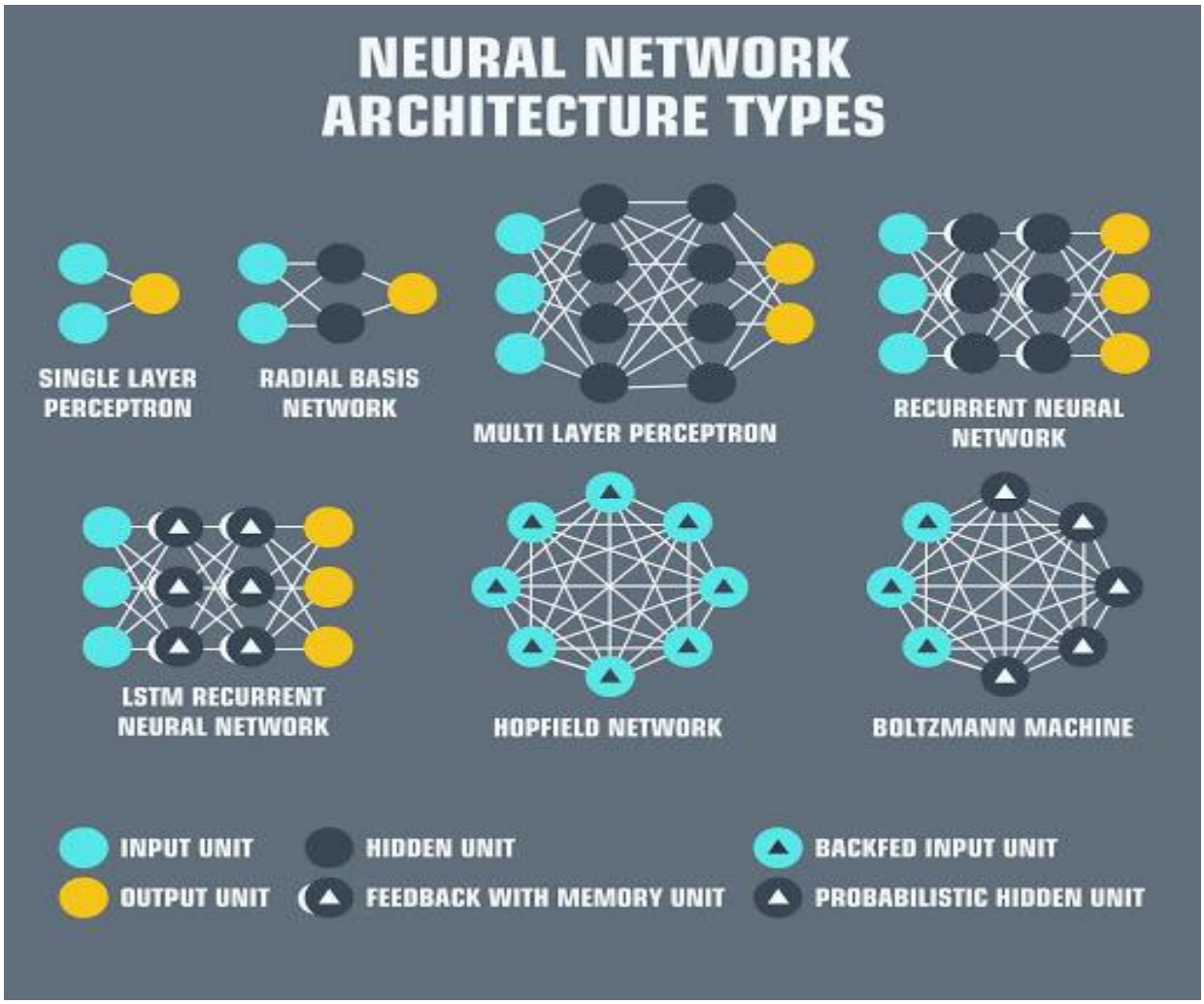


The Neural Network is constructed from 3 type of layers:

1. Input layer — initial data for the neural network.

2. Hidden layers — intermediate layer between input and output layer and place where all the computation is done.
3. Output layer — produce the result for given inputs.

Architecture Types of Neural Networks



Now that we know the construction of a neural network, we construct such neural network models. Before working on the dataset, we load the required libraries and do some data preprocessing.

Methods:

In order to deploy a predictive model, we're going to make use of neural networks. But before that we will have look at our data and if there is any need to preprocess it. The data file data_activity_recognition.RData contains training and test data. The training data x_train includes 400 signal segments for each activity (total of 7600 signals), while the test data x_test includes 80 signal segments for each activity (totalof1520signals). The activity labels are in the objects y_train and y_test, respectively. The input data are organized in the form of 3-dimensional arrays, in which the first dimension denote the signal, the second the sampling instants and the third the sensors, so each signal is described by a vector of 125 × 45 = 5625 features

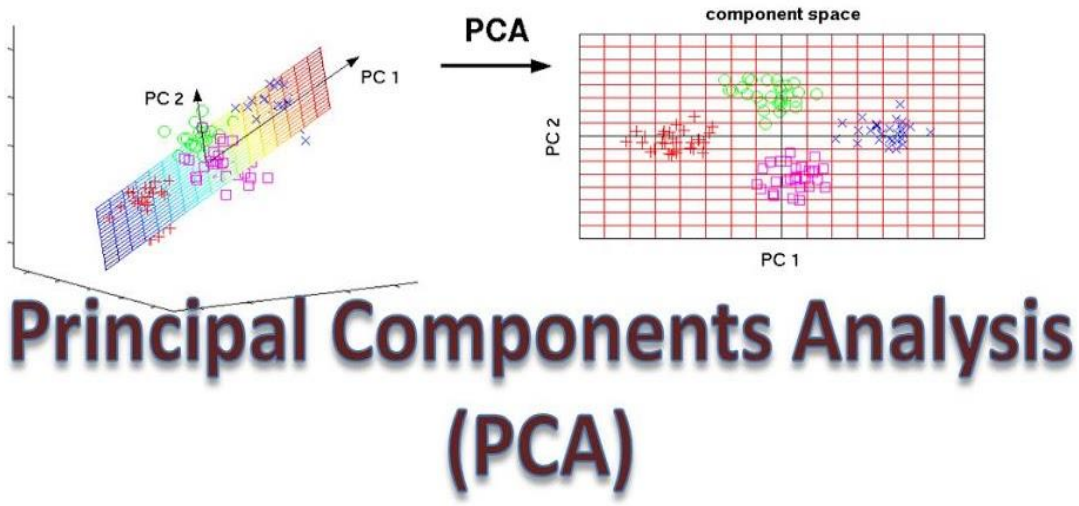
```
library(keras)
library(tfruns)

load("data_activity_recognition.RData")#Loading the data

dim(x_train)#checking the dimensions of the features

## [1] 7600  125   45

#reshaping the arrays from 3D to 2D
x_train=data.matrix(array_reshape(x_train,c(nrow(x_train),125*45)))
x_test=data.matrix(array_reshape(x_test,c(nrow(x_test),125*45)))
#checking the reshaped data
dim(x_train)

## [1] 7600 5625
```

The data file data_activity_recognition.RData contains training and test data. The training data x_train includes 400 signal segments for each activity (total of 7600 signals), while the test data x_test includes 80 signal segments for each activity (totalof1520signals). The activity labels are in the objects y_train and y_test, respectively. The input data are organized in the form of 3-dimensional arrays, in which the first dimension denote the signal, the second the sampling instants and the third the sensors, so each signal is described by a vector of 125 × 45 = 5625 features. First, we reshape the training and test feature from a 3D array to 2D array. We also perform one-hot encoding on the target training and test variable. Thereafter, we perform range normalization on our features in both the training and test sets. All these steps are taken in order to neatly fit the predictive model.

```
#performing one-hot coding on the target
y_train=data.matrix(data.frame(y_train))
y_test=data.matrix(data.frame(y_test))
y_train=to_categorical(y_train-1)
y_test=to_categorical(y_test-1)

# normalizing the range of the features
range_norm = function(x,a =0,b =1) {
  ( (x-min(x))/(max(x)-min(x)) )*(b-a)+a
  }
x_train=apply(x_train,2, range_norm)
x_test=apply(x_test,2, range_norm)

# reducing the dimensions using PCA
pca=prcomp(x_train)
prop=cumsum(pca$sdev^2)/sum(pca$sdev^2)
Q=length(prop[prop<0.99])
Q   # checking the new dimensions of the features

## [1] 1134

xz_train=pca$x[,1:Q]
xz_test <-predict(pca, x_test)[,1:Q]
```

PRINCIPAL COMPONENT ANALYSIS

As we can see we have a high-dimensional data with 5625 features and hence we go ahead with performing the Principal Component Analysis (PCA). Principal component analysis (PCA) is one of the most popular dimension reduction technique.



The idea behind PCA is that the data can be expressed in a lower dimensional subspace, characterized by independent coordinate vectors which can explain most of the variability present in the original data. The number of such vectors, and ultimately the dimension of the subspace, is usually set by keeping the first Q vectors which can explain a pre-specified proportion of the total variability in the data (usually in the range 0.70 - 0.99).

Sometimes, the size Q of the subspace can also be specified arbitrarily in advance. In R, PCA can be implemented using the prcomp() function. The function provides in output a measure of the variation explained by each coordinate vector of the subspace. This information is contained in the slot sdev, which measures the standard deviation, taking the square we obtain the explained variance. Vectors in output are ordered according to this variance, from the largest to the smallest. We keep the first Q coordinate vectors such that 99% of the variability can be explained in the subspace.

The dimensionality of the predictors has been reduced enormously, and these vectors identify directions of the subspace which can account for 99% of variability. The original input features are mapped into this low dimensional subspace, obtaining a representation of the data points through a new set of Q features, the principal components.

This representation of the data points in the new features is readily available in the slot x of the output. This will be our dimension-reduced training set. PCA learns a mapping, so we can generalize this mapping from the original data-space to the low dimensional subspace also to the test data. To do so we map the test data onto the subspace, using the function predict.

Model with 2 Hidden Layers

```
N =nrow(xz_train)
V =ncol(xz_train)


#model with 2 hidden layers only

# model definition
model1 =keras_model_sequential()%>%
  layer_dense(units =1134,activation ="relu",input_shape =V)%>%
  layer_dense(units =800,activation ="relu")%>%
  layer_dense(units =ncol(y_train),activation ="softmax")%>%
  compile(loss ="categorical_crossentropy",metrics ="accuracy",optimizer =optimizer_sgd()
)
```

Training the Model

The process of modelling means training a machine learning algorithm to predict the labels from the features, tuning it for the business need, and validating it on holdout data. The output from modelling is a trained model that can be used for *inference*, making predictions on new data points.

```
# model training
fit1=model1%>%
  fit(x =xz_train,y =y_train,
      validation_data =list(xz_test, y_test),
      epochs =100,
      verbose =1,
      )
```

Graphical Representation of the errors

```
# graphical representation of the errors
smooth_line =function(y) {
  x =1:length(y)
  out =predict(loess(y~x) )
  return(out)
  }
cols =c("deepskyblue2","red")
out =1-cbind(fit1$metrics$accuracy,fit1$metrics$val_accuracy)
```

```
matplot(out,pch =19,ylab ="Error",xlab ="No. of Epochs",col =adjustcolor(cols[1:2],0.3),l
og ="y")
matlines(apply(out,2, smooth_line),lty =1,col =cols[1:2],lwd =2)
legend("bottomleft",legend =c("Training Error","Test Error"),fill =cols[1:2],bty ="n")

# predictive accuracy of the model

tail(fit1$metrics$val_accuracy,1)

## [1] 0.925



#model with 2 hidden layers and L2 regularization with early stopping

# model definition
model2 =keras_model_sequential()%>%
  layer_dense(units =1134,activation ="relu",input_shape =V,kernel_regularizer = regulari
zer_l2(l=0.01))%>%
  layer_dense(units =800,activation ="relu",kernel_regularizer = regularizer_l2(l=0.01))%
>%
  layer_dense(units =ncol(y_train),activation ="softmax")%>%
  compile(loss ="categorical_crossentropy",metrics ="accuracy",optimizer =optimizer_sgd()
)

# model training
fit2=model2%>%
  fit(x =xz_train,y =y_train,
      validation_data =list(xz_test, y_test),
      epochs =100,
      verbose =1,
      callbacks = callback_early_stopping(monitor = "val_accuracy", patience =10)
      )

# graphical representation of the errors
out =1-cbind(fit2$metrics$accuracy,fit2$metrics$val_accuracy)
matplot(out,pch =19,ylab ="Error",xlab ="No. of Epochs",col =adjustcolor(cols[1:2],0.3),l
og ="y")
matlines(apply(out,2, smooth_line),lty =1,col =cols[1:2],lwd =2)
legend("bottomleft",legend =c("Training Error","Test Error"),fill =cols[1:2],bty ="n")

# predictive accuracy of the model

tail(fit2$metrics$val_accuracy,1)

## [1] 0.9447368
```

Predictive accuracy can also be based on the differences between the predicted values for, and the observed values of, new samples (e.g., validation samples). This is the **predictive accuracy** we refer to in this study.

What is overfitting?

Overfitting is a phenomenon which occurs when a model learns the detail and noise in the training data to an extent that it negatively impacts the performance of the model on new data.

So the overfitting is a major problem as it negatively impacts the performance.

Regularization technique to the rescue.

Generally, a good model does not give more weight to a particular feature. The weights are evenly distributed. This can be achieved by doing regularization.

Regularization

Regularization in machine learning is the process of regularizing the parameters that constrain, regularizes, or shrinks the coefficient estimates towards zero. In other words, this technique discourages learning a more complex or flexible model, avoiding the risk of **Overfitting**.

There are two types of regularization as follows:

- L1 Regularization or Lasso Regularization
- L2 Regularization or Ridge Regularization

What is l2 regularization?

L2 Regularization or Ridge Regularization
In L2 regularization, regularization term is the sum of square of all feature weights as shown above in the equation. L2 regularization forces the weights to be small but does not make them zero and does non sparse solution.

Is early stopping good?

The model at the time that training is stopped is then used and is known to have good generalization performance. This procedure is called “early stopping” and is perhaps one of the oldest and most widely used forms of neural network regularization. This strategy is known as early stopping.

```
#model with 3 hidden layers and L2 regularization with early stopping

# model definition
model3 =keras_model_sequential()%>%
  layer_dense(units =1134,activation ="relu",input_shape =V,kernel_regularizer = regulari
zer_l2(l=0.01))%>%
  layer_dense(units =800,activation ="relu",kernel_regularizer = regularizer_l2(l=0.01))%
>%
```

```
  layer_dense(units =500,activation ="relu",kernel_regularizer = regularizer_l2(l=0.01))%
>%
  layer_dense(units =ncol(y_train),activation ="softmax")%>%
  compile(loss ="categorical_crossentropy",metrics ="accuracy",optimizer =optimizer_sgd()
)

# model training
fit3 =model3%>%
  fit(x =xz_train,y =y_train,
      validation_data =list(xz_test, y_test),
      epochs =100,
      verbose =1,
      callbacks = callback_early_stopping(monitor = "val_accuracy", patience =10)
      )

# graphical representation of the errors
out =1-cbind(fit3$metrics$accuracy,fit3$metrics$val_accuracy)
matplot(out,pch =19,ylab ="Error",xlab ="No. of Epochs",col =adjustcolor(cols[1:2],0.3),l
og ="y")
matlines(apply(out,2, smooth_line),lty =1,col =cols[1:2],lwd =2)
legend("bottomleft",legend =c("Training Error","Test Error"),fill =cols[1:2],bty ="n")

#predictive accuracy of the model
tail(fit3$metrics$val_accuracy,1)

## [1] 0.9381579
```

Now that we have made our data ready for analysis, let's deploy various neural networks with and without regularization.

OVERFITTING AND REGULARISATION:

Due to their large flexibility, deep neural networks (and more in general deep learning methods) are highly prone to overfitting. Overfitting is when the model has a high performance in the training data, but it has poor predictive performance on new unseen data. This happens because the model also learns random aspects of the training data that do not generalize well to the test data.

Signs of overfitting: – Predictions for the same unseen data point obtained by models trained on different subsets of the data have large variability. – Large gap between training error and validation/test error. High complexity leads to perfect in training performance, but it produces poor generalization performance. The figure below depicts overfitting.

Regularization is the process of adding external information to the machine learning algorithm in order to reduce/prevent overfitting and reduce the generalization error. For this project we have used 2 methods of regularization. (1) Weight decay or L2 regularization and (2) Early stopping.

L2 regularization:

Consider the case of updating a single weight w using a gradient descent method. Weight decay induces the update:

$$w \leftarrow w(1 - \eta\lambda) - \eta\nabla E(w)$$

In the update, a weight first is decayed by the factor (1 − ηλ). Assuming an initial value of w = 0, weight decay can be viewed as a forgetting mechanism, which "softly" brings the weights closer to their initial values. Only weights updated repeatedly compensate the decay and will result to be large in magnitude compared to the others, which will be shrunk close to zero.

Early Stopping

Neural networks are trained using gradient-descent methods, which are based on multiple scans of the data. These methods are executed to convergence, that is when numerical minimization of the objective function is attained. This optimizes the error on the training data, but not necessarily the error corresponding the out-ofsample test data. The training error decreases steadily over number of epochs.

Thus, the number of epochs is a complexity parameter, which can be tuned by monitoring the error of a validation set. Early stopping consists in stopping the training procedure when the validation error start increasing (despite the training error keeps decreasing). At that stage there is no point in training further, as the neural network starts overfitting. When stopped, the training algorithm terminates and return the parameters estimated at the optimal epoch. Early stopping is simple and effective, and it can be used in conjunction to other forms of regularization.

```
# hyperparameter tuning values
size1_set=c(1134,800,500)
size2_set=c(1134,800,500)
lambda_set=c(0,0.002,0.004,0.006,0.008,0.01)

# tuning procedure
runs=tuning_run("Model configuration project.R",
                runs_dir = "runs_result",
                flags = list(size1=size1_set,
                             size2=size2_set,
                             l2=lambda_set),sample=0.5
                )

read_metrics <- function(path, files = NULL)
# 'path' is where the runs are --> e.g. "path/to/runs"
{

path <- paste0(path, "/")
if ( is.null(files) ) files <- list.files(path)
n <- length(files)
out <- vector("list", n)
for ( i in 1:n ) {
dir <- paste0(path, files[i], "/tfruns.d/")
out[[i]] <- jsonlite::fromJSON(paste0(dir, "metrics.json"))
out[[i]]$flags <- jsonlite::fromJSON(paste0(dir, "flags.json"))
}
return(out)
}
```

```
plot_learning_curve <- function(x, ylab = NULL, cols = NULL, top = 3, span = 0.4, ...)
{
# to add a smooth line to points
smooth_line <- function(y) {
x <- 1:length(y)
out <- predict( loess(y ~ x, span = span) )
return(out)
}
matplot(x, ylab = ylab, xlab = "Epochs", type = "n", ...)
grid()
matplot(x, pch = 19, col = adjustcolor(cols, 0.3), add = TRUE)
tmp <- apply(x, 2, smooth_line)
tmp <- sapply( tmp, "length<-", max(lengths(tmp)) )
set <- order(apply(tmp, 2, max, na.rm = TRUE), decreasing = TRUE)[1:top]
cl <- rep(cols, ncol(tmp))
cl[set] <- "deepskyblue2"
matlines(tmp, lty = 1, col = cl, lwd = 2)}

out = read_metrics("runs_result")
# extract validation accuracy and plot learning curve
acc = sapply(out, "[[", "val_accuracy")
plot_learning_curve(acc, col = adjustcolor("black", 0.3), ylim = c(0.85, 1),ylab = "Val a
ccuracy", top = 3)

res = ls_runs(metric_val_accuracy > 0.94,runs_dir = "runs_result", order = metric_val_acc
uracy)
res = res[,c(2,4,8:11)]
res[1:10,] # top 10 models

## Data frame: 10 x 6
##    metric_val_accuracy eval_accuracy flag_size1 flag_size2 flag_l2 samples
## 1               0.9513        0.9513       1134        800   0.008    7600
## 2               0.9507        0.9507       1134        800   0.004    7600
## 3               0.9507        0.9507        800        800   0.010    7600
## 4               0.9500        0.9500        500        800   0.004    7600
## 5               0.9493        0.9493       1134        500   0.004    7600
## 6               0.9493        0.9493        500       1134   0.006    7600
## 7               0.9487        0.9487       1134        800   0.000    7600
## 8               0.9474        0.9474       1134       1134   0.006    7600
## 9               0.9441        0.9441        800        800   0.002    7600
## 10              0.9434        0.9434        800        500   0.002    7600
```

TUNING THE SELECTED MODEL FOR HYPERPARAMETERS:

Due to the high risk of overfitting, regularization is crucial for deep neural network. Hyper parameters control the complexity of the model and need to be tuned by evaluation of the performance on a separate test/validation set. In our project, we tune the number of nodes and the penalty parameter of the L2 regularization for our selected models.

In order to tune our model, we resort to grid search. For this, we explore a grid of possible hyper parameter configurations, after plausible ranges have been set. We take into account interactions between multiple hyper parameters. To tune effectively the model configuration, we make use of package tfruns. This package provides a suite of tools for tracking, visualizing, and managing training runs and experiments, which is particularly useful to compare hyper parameters and metrics across runs to find the best performing model.

Package tfruns allows to run multiple models with different configurations and we will use it to tune the model settings and hyperparameters. The package uses “flags” to denote hyperparameters and

settings which vary from one run to the other. Flags are defined for key hyperparameters/settings, then training and evaluation is performed over the combinations of those flags to determine which combination of flags yields the best model. The function tuning_run() is called in the line of code. The first argument specifies the path to the training script containing the baseline model configuration and training/evaluation settings.

Flags are specified using argument flags, while argument sample is used to set the proportions of configurations to tbe considered. Argument runs_dir is employed to specify the directory in which all runs will be stored. In your local working directory, you should have the folder runs_results, which includes all results and information about the runs. Now we can explore the results and determine the optimal configuration for the selected model.

The two functions read_metrics() and plot_learning_curve() will be used to extract values from the stored runs and plot the corresponding validation learning curves. With this, we will conclude our analysis.

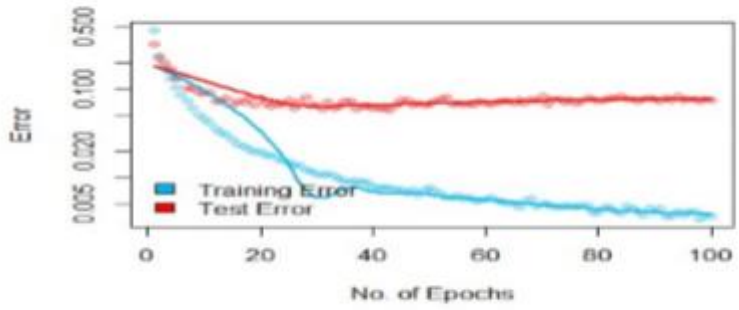
Results and Discussions:

- For all the models used for analysis in this project, our activation function for each hidden layer is the Rectified Linear Unit (Relu) and the output function in the output layer is the "softmax" function.
- The error function used for each model is the categorical_crossentropy() while the optimizer for each model is the stochastic gradient descent.
- The number of nodes in the first layer are 1134, second layer are 800 and third layer (if any) are 500.
- The regularization methods used(if at all) are L2 regularization and early stopping with lambda = 0.01 and patience level 10 respectively.

Selection of the optimal model (Less complex and more accurate):

After the preprocessing of the data, we begin the first part of our project which is the selection of the optimal model. We compare the following three models:

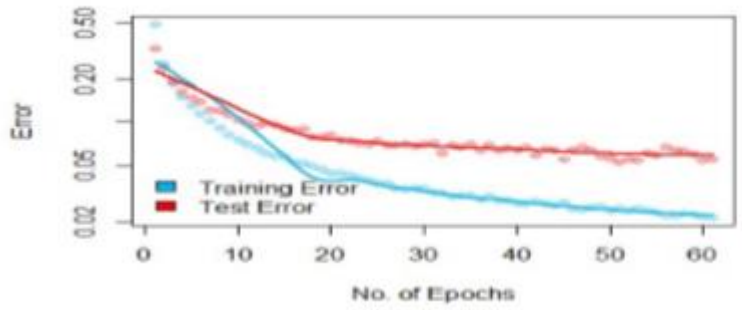
2 hidden layers without regularization: We start with a simple neural network model with 2 hidden layers without any type of regularization. We train this model on the reduced training set and check its predictive accuracy on the reduced test set. We plot the training and test errors over 100 epochs and get the following graph:



Looking at the graph we can see that there is clear evidence of overfitting for the model with 2 hidden layers and no regularization and the predictive accuracy for this model is approximately 92.5%

2 hidden layers with L2 regularization and early stopping:

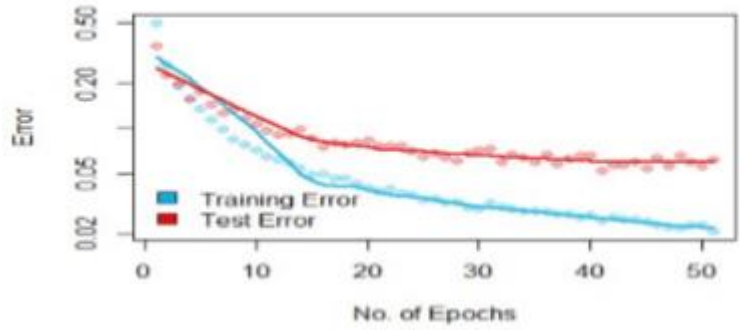
In order to get rid of the over fitting taking place in the previous model, we implement L2 regularization and early stopping in the same model. We train this new model on the reduced training set and check its predictive accuracy on the reduced test set. We plot the training and test errors over 100 epochs and get the following graph:



Looking at the graph above, we can say that we have successfully got rid of the overfitting in the model without regularization and the training of the model stopped after around 60 epochs. The predictive accuracy of the model is approximately 94%.

3 hidden layers with L2 regularization and early stopping:

To check if we can get better accuracy by adding an additional layer to the previous neural network model, we add an additional hidden layer to the regularized model. We train this new model on the reduced training set and check its predictive accuracy on the reduced test set. We plot the training and test errors over 100 epochs and get the following graph:



Looking at the graph above, we can say that by adding an additional hidden layer to the regularized model, we get a similar pattern in values of the train and test error as compared to the regularized model. Also, the predictive accuracy of this model is approximately 94%.

Thus, looking at all the facts stated above we can infer that the model which is less complex and giving a high accuracy is the neural network with 2 hidden layers along with L2 regularization and early stopping with a predictive accuracy of 94%.

Tuning model with 2 hidden layers along with L2 regularization & early stopping:

Now we will tune the values of the number of nodes in each layer and the penalty of the L2 regularization to see if we get a better prediction accuracy.

```
FLAGS = flags(
 flag_numeric("size1",1000), flag_numeric("size2",500),flag_numeric("l2",0.4)
)
# model definition
model = keras_model_sequential()%>%
layer_dense(units =FLAGS$size1,activation ="relu",input_shape =V,kernel_regularizer = regularizer_l2(l=FLAGS$l2))%>%
 layer_dense(units =FLAGS$size2,activation ="relu",kernel_regularizer = regularizer_l2(l=FLAGS$l2))%>%
 layer_dense(units =ncol(y_train),activation ="softmax")%>%
 compile(loss ="categorical_crossentropy",metrics ="accuracy",optimizer =optimizer_sgd())
# model training
fit = model%>%
 fit(x =xz_train,y =y_train,
    validation_data =list(xz_test, y_test),
    epochs =100,
    verbose =1,
    callbacks = callback_early_stopping(monitor = "val_accuracy", patience = 10)
 )
# store accuracy on test set for each run
score <- model %>% evaluate(
 xz_test, y_test,
 verbose = 0
)
```

We tune this configuration for the number of nodes in the 2 layers and lambda.

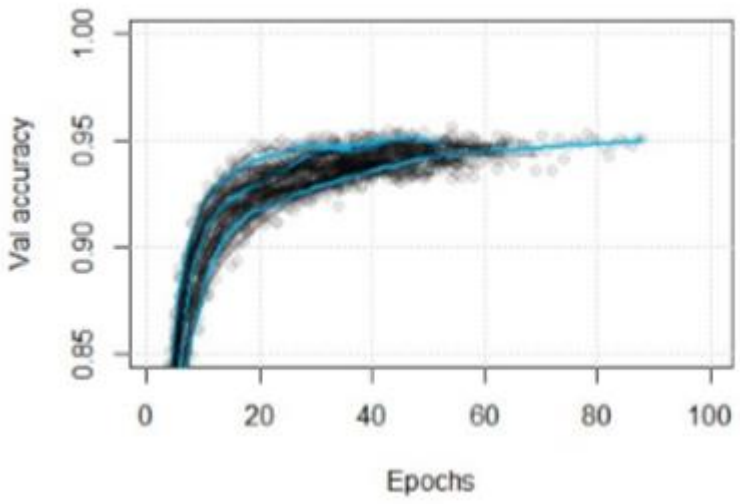
Following is the hyper parameters grid:

1st layer size: 1134 – 800 – 500

2nd layer size: 1134 – 800 – 500

Weight decay λ: 0 – 0.002 – 0.004 –0.006 –0.008 –0.01

There are 54 possible combinations of the hyper parameters. We sample out 50% of the configurations from the grid, resulting in 27 possible configurations to be evaluated. After we tune the 27 models, we get the following graphs for the accuracies of these models over 100 epochs:



Looking at the graph above, we can say that all the 27 models have the predictive accuracy between 90% and 95%. The three curves in the blue color represent the accuracies of the top 3 model configurations. These can be tabulated as:

	Accuracy	Layer 1 nodes	Layer 2 nodes	Lambda
1	0.9513	1134	800	0.008
2	0.9507	1134	800	0.004
3	0.9507	800	800	0.01

Thus, looking at the topmost row in the table, we can say that the predictive accuracy is the highest for the model with 1134 nodes in the first layer, 800 nodes in the second layer and the lambda value being 0.008. This highest accuracy is approximately 95%.

Conclusion:

Thus, in our entire analysis, we have done everything right from data preprocessing to optimal model selection and tuning the selected optimal model. We achieved all this by doing the following:

(1) Reshaping the 3D features to 2D features, performed one-hot encoding on the target variables, normalizing the range of the features and reducing the high-dimensional features by implementing PCA.
(2) Deploying a simple 2 layered neural network model without any regularization and comparing this model's performance to the performance of the 2 layered regularized neural network.
(3) Observing the regularized 2 layered model being better, we compared this with a 3 layered regularized model and concluding the 3 layered regularized model to be optimal as both had similar performance.
(4) Tuning the optimal 2 layered regularized model and finding the optimal values for the hyper parameters. In the end we successfully obtain a predictive model having an accuracy of 95%.

References:

The entire project has been referred from the lecture materials for the course STAT-40970 Machine Learning and Artificial Intelligence.