Analysis Of Activity Recognition Data

Nisarga G - 19203753

01/05/2020

ABSTRACT

The main task is to deploy a neural network model which can be employed for daily/sports activity recognition . The train data is used to train the model and the test data is used to evaluate its predictive performance.

INTRODUCTION

The dataset used here is activity recognition data. The data are motion sensor measurements of 19 daily and sports activities, each performed by 8 subjects (4 female, 4 male, between the ages 20 and 30) in their own style for 5 minutes. The data is in 3 dimension. We deploy multilayer neural network model as multilayer neural networks extend the idea of single hidden layer neural networks by including more than one hidden layer, hence we deploy multilayer neural network with 2 and 3 hidden layers followed by penalty based regularization of both model. Finally model tuning is done.

METHODS

Loading the data set into R:

```r
#Loading the data
data <- load("data_activity_recognition.RData")
```

Load the required libraries into R:

```r
library(keras)
library(tensorflow)
library(ggplot2)
```

DATA PREPERATION

This step is also called data cleaning , first we convert the 3D arrays into 2D with the help of array_reshape function.

```r
#Converting input data into 125 X 45 = 5625 vectors
x_train <- array_reshape(x_train, c(nrow(x_train), 125*45))
x_test <- array_reshape(x_test, c(nrow(x_test), 125*45))
```

Some data transformation methods like range normalization can be performed such that the values of x_train and y_train lies between 0 and 1.The purpose of normalization is to store each row of data only once, to avoid data anomalies. A data anomaly happens when you try to store data in two places, and one copy changes without the other copy changing in the same way.Once the normalization is done, there is no need fr scaling.Also, normalizing is the important step to perform before PCA.

```
#Range normalizing the input variables
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)
range(x_train)

## [1] 0 1

x_test <- apply(x_test, 2, range_norm)
range(x_test)

## [1] 0 1
```

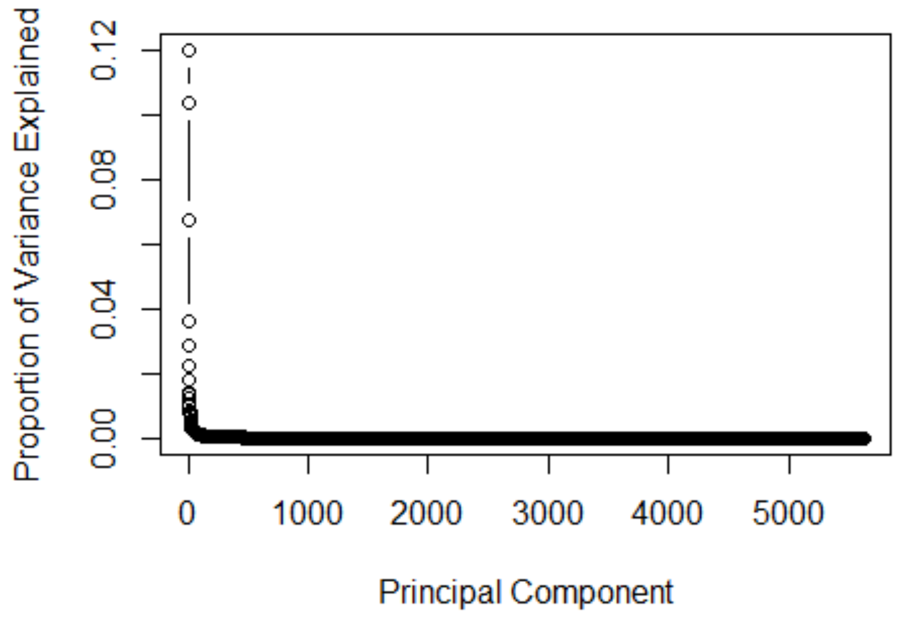
When a predictor is categorical, it is common to decompose it into a set of binary variables corresponding to their categories.This process is denoted as “one-hot encoding” and each category-related variable is a dummy variable which is a binary indicator for that level.

```
#Converting character variables to numeric variables
tmp1 <- as.factor(y_train)
y_train <- as.numeric(tmp1)
tmp2 <- as.factor(y_test)
y_test <- as.numeric(tmp2)
#Converting target variable to categories
y_train1 <- to_categorical(y_train)
y_test1<- to_categorical(y_test)
y_train1 <- y_train1[,-1]
y_test1 <- y_test1[,-1]
```

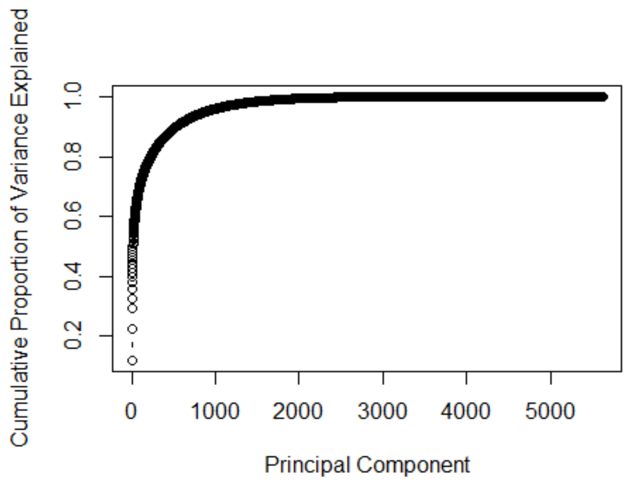
Dimension Reduction : the main goal is to extract a set of principal features via data projection or factorization. PCA is the most commonly used dimensional technique. Library factorextra and FactoMineR are helpful to peform PCA. prcomp function performs PCA on training data. proportion of variance explained is calculated and plotted.

```
library(factoextra)

## Welcome! Want to learn more? See two factoextra-related books at https://g
oo.gl/ve3WBa

library(FactoMineR)

##Dimension Reduction for x_train
pca <- prcomp(x_train, scale = TRUE)

#Computing standard deviation of each principal component
std_dev <- pca$sdev

#Computing variance
```

```r
pr_var <- std_dev^2

#Proportion of variance explained
prop_varex <- pr_var/sum(pr_var)

#Scree plot
plot(prop_varex, xlab = "Principal Component",
      ylab = "Proportion of Variance Explained",
      type = "b")
```



```r
#Cumulative scree plot
plot(cumsum(prop_varex), xlab = "Principal Component",
ylab = "Cumulative Proportion of Variance Explained",
type = "b")
```



From the Scree plot we can infer that the elbow of the curve lies somewhere around 500 , hence 500 columns are chosen for training data, test data is also predicted using the same pca function and 500 variables are selected for the final test data.

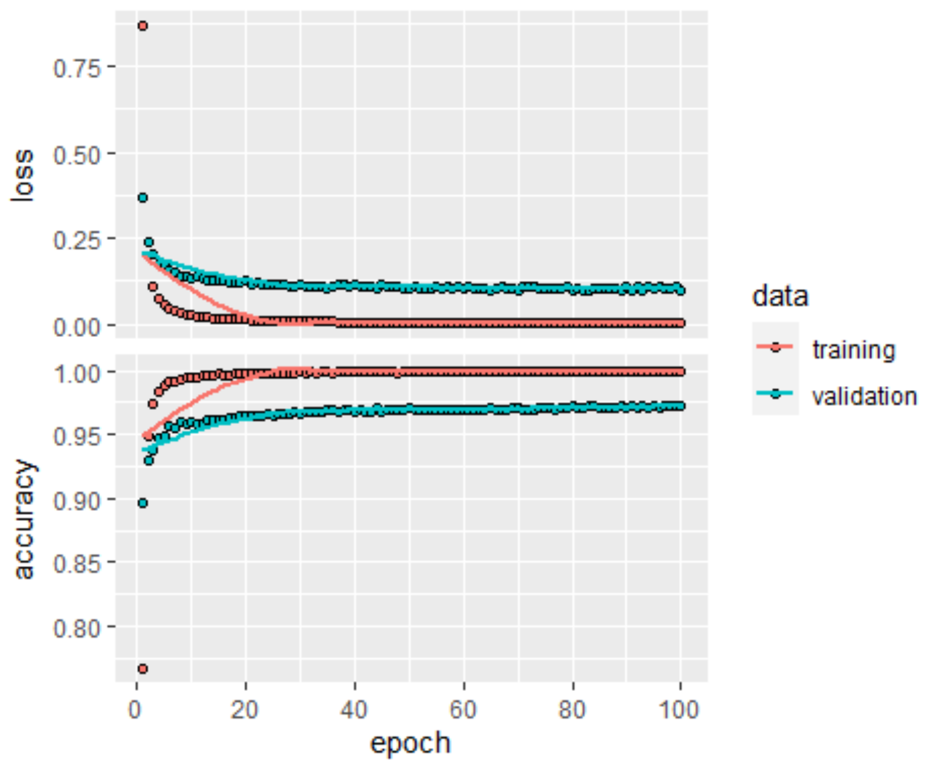
```
x_train <- data.frame(pca$x)
set.seed(19203753)
x_train <- x_train[,1:500]
#Transforming test into PCA
test.data <- predict(pca, newdata = x_test)
test.data <- as.data.frame(test.data)
#Selecting the first 500 components
set.seed(19203753)
x_test <- test.data[,1:500]
#Converting data frames to matrices
x_train <- as.matrix(x_train)
x_test <- as.matrix(x_test)
```

MULTILAYER NEURAL NETWORK

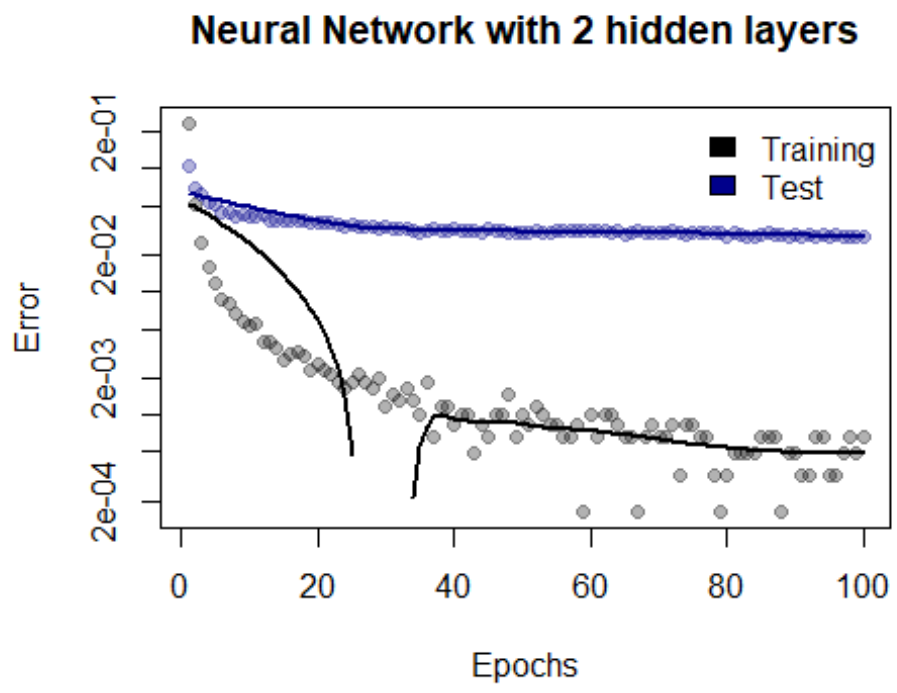
Using the functionalities of keras we deploy a neural network with 2 and 3 hidden layers with ReLU activation and one output layer with softmax activation.In the first layer we consider 256 units, as the input units, then in the next layer we set the number of neurons by halving the number of units of the previous level i.e 128(total 2 layers), 64, and so on..To compare predictive performance improvement obtained by adding one extra hidden layer to the network architecture, we create another model with 3 hidden layers.

Fit the model : The function fit is used to train both model and model1.Here we consider 100 epochs. The function allows also to evaluate the performance of the model on a separate test set at each epoch, to monitor the progress during the training process.Argument verbose = 1 produces an interactive visualization of the training procedure. The output of fit is a list which contains the configuration parameters of the model (in params) and performance metrics (in metrics).The performance metrics are the accuracy and value of error function at each epoch for training and test set. We can look at the error progress over the epochs.

```
#Creating Sequential Model
model1 <- keras_model_sequential()
model1 %>%
layer_dense(units = 256, activation = "relu", input_shape = ncol(x_train)) %>
%
layer_dense(units = 128, activation = "relu") %>%
#layer_dense(units = 64, activation = "relu") %>%
layer_dense(units = 19, activation = "softmax")
#Compiling the model
model1 %>%
compile(loss = "categorical_crossentropy", metrics = "accuracy",
optimizer = optimizer_sgd(),)

fit1 <- model1 %>%
fit(x_train, y_train1,
epochs = 100,
batch_size = 32,
validation_split = 0.2)

plot(fit1)

## `geom_smooth()` using formula 'y ~ x'
```

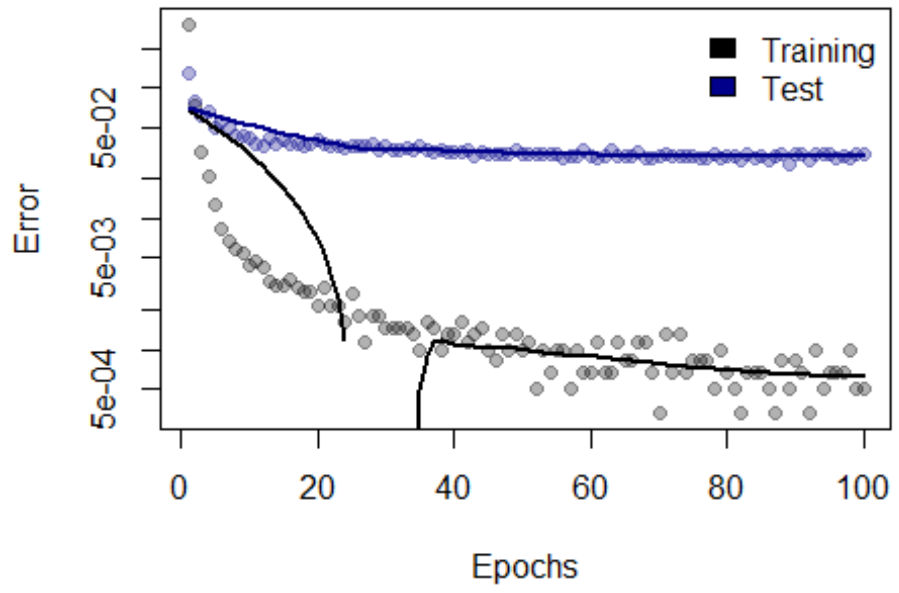


```
#Evaluating model with test data
model1_tes <- model1 %>%
evaluate(x_test, y_test1)
#Adding a smooth line to points
smooth_line <- function(y) {
x <- 1:length(y)
out <- predict(loess(y ~ x))
return(out)
}
cols <- c("black","darkblue","gray","deepskyblue")
#Checking performance
out <- 1-cbind(fit1$metrics$accuracy, fit1$metrics$val_accuracy)
matplot(out, pch = 19, ylab = "Error", xlab = "Epochs",
col = adjustcolor(cols[1:2], 0.3), log = "y",main = "Neural Network with 2 hi
dden layers" )
matlines(apply(out, 2, smooth_line), lty = 1, col = cols[1:2], lwd = 2)
legend("topright", legend = c("Training", "Test"),
fill = cols[1:2], bty = "n")
```



```
#3-layered neural network
model2 <- keras_model_sequential()
model2 %>%
layer_dense(units = 256, activation = "relu", input_shape = ncol(x_train)) %>
%
layer_dense(units = 128, activation = "relu") %>%
layer_dense(units = 64, activation = "relu") %>%
layer_dense(units = 19, activation = "softmax")
```

```r
#Compile to configure the learning process
model2 %>%
compile(loss = "categorical_crossentropy", metrics = "accuracy",
optimizer = optimizer_sgd(),)

#Fit the model
#Batch_size - number of samples used per gradient
fit2 <- model2 %>%
fit(x_train, y_train1,
epoch = 100,
batch_size = 32,
validation_split = 0.2)

#Evaluate model with test data
model2_tes <- model2 %>%
evaluate(x_test, y_test1)
cols <- c("black","darkblue","gray","deepskyblue")

#Check performance ---> error
out <- 1-cbind(fit2$metrics$accuracy, fit2$metrics$val_accuracy)
matplot(out, pch = 19, ylab = "Error", xlab = "Epochs",
        col = adjustcolor(cols[1:2], 0.3), log = "y")

matlines(apply(out, 2, smooth_line), lty = 1, col = cols[1:2], lwd = 2)
         legend("topright", legend = c("Training", "Test"),
        fill = cols[1:2], bty = "n")
```



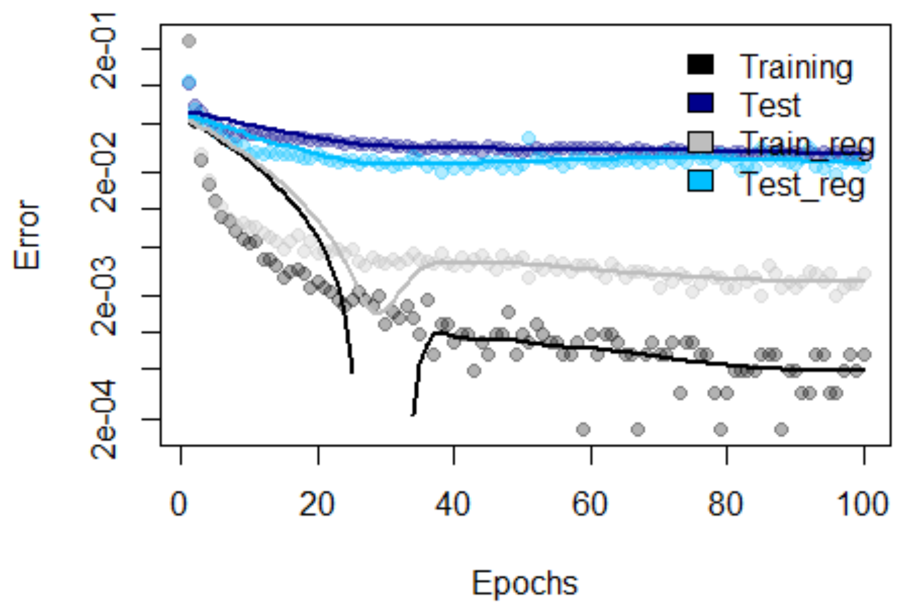
Neural Network with 3 hidden layers

PENALTY BASED REGULARIZATION

We could try to improve the generalization performance by tackling overfitting.The argument used to set a regularization term applied to the weights of each layer is kernel_regularizer.

```
#Penalty-based regularization - 2 layers
#Training configuration after regularization
model_reg1 <- keras_model_sequential() %>%
 layer_dense(units = 256, activation="relu",input_shape=ncol(x_train),kernel_
regularizer=regularizer_l2(l=0.009))%>%
 layer_dense(units=128,activation="relu",input_shape=ncol(x_train),kernel_reg
ularizer=regularizer_l2(l=0.009))%>%
 layer_dense(units = 19, activation = "softmax") %>%
 compile(loss = "categorical_crossentropy", optimizer = optimizer_sgd(), metr
ics = "accuracy")
#Train and evaluate on test data at each epoch
fit_reg1 <- model_reg1 %>%
           fit(x_train, y_train1,
           epoch = 100,
           batch_size = 32,
           validation_split = 0.2)

model3 <- model_reg1 %>%
          evaluate(x_test, y_test1)
```

```
#Loss and Accuracy
model3

## $loss
## [1] 0.6734332
##
## $accuracy
## [1] 0.8565789

#Compare regularized model VS unregularized model
out <- 1 - cbind(fit1$metrics$accuracy,fit1$metrics$val_accuracy,fit_reg1$met
rics$accuracy,fit_reg1$metrics$val_accuracy)
#Check performance
matplot(out, pch = 19, ylab = "Error", xlab = "Epochs",
        col = adjustcolor(cols, 0.3), log = "y")

matlines(apply(out, 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright",
       legend = c("Training","Test","Train_reg","Test_reg"),
       fill = cols, bty = "n")
```



Below we can compare the performance of the regularized model against the performance of the unregularized one.The test error of the regularized model is minimally lower than the test error of the unregularized one. It seems that here regularization marginally improves the generalization performance.We can inspect the weights of the two models.

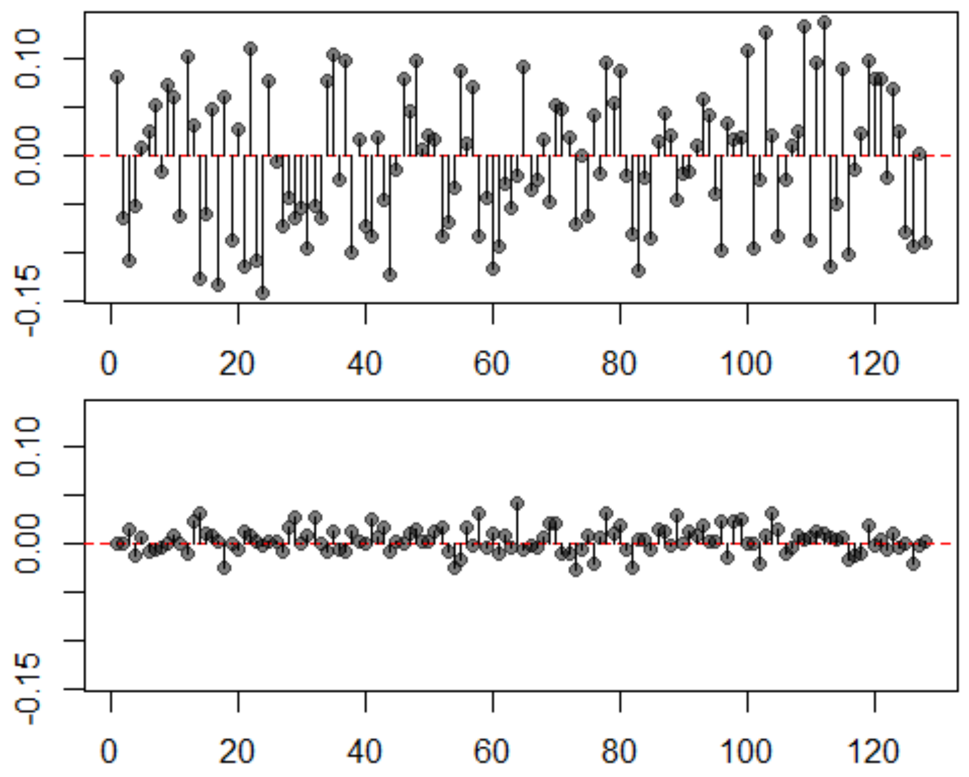
We consider one instance of the weights of the first hidden layer. The weights of the regularized model are lower in magnitude.

From the output below we can infer that the test error without regularization is 0.027 while with regularization is 0.019, hence test error has decreased after regularization for model with 2 hidden layers. While for 3 hidden layers test error has increased after regularization. Also, the weights of the regularized model are lower in magnitude.

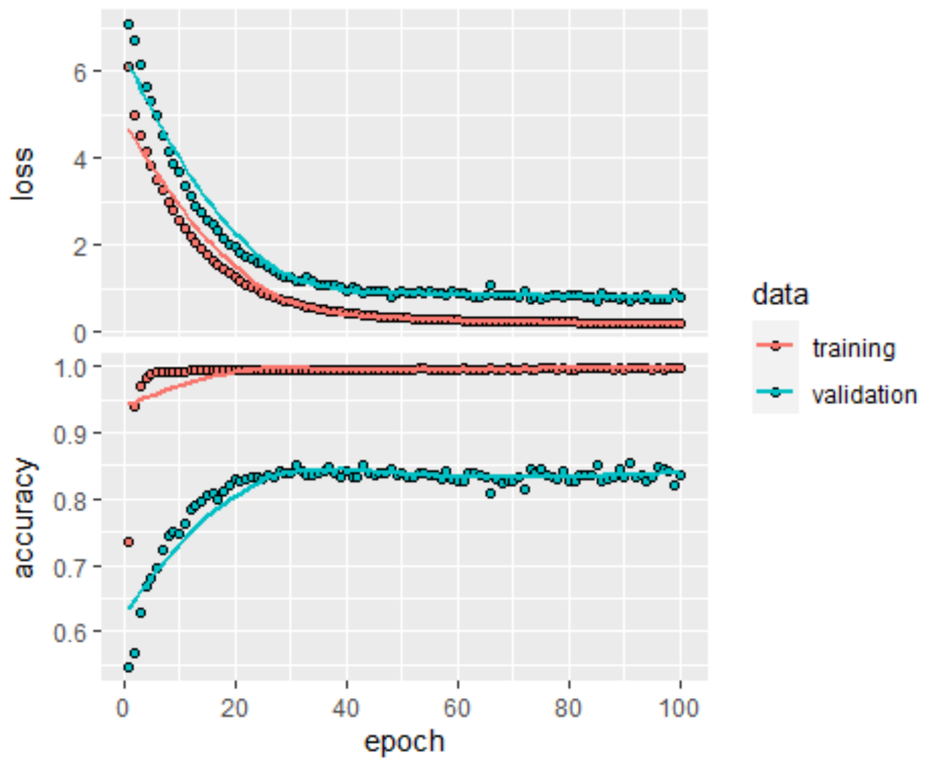
```r
apply(out, 2, min)
```

```
## [1] 0.0001644492 0.0276315808 0.0019736886 0.0190789700
```

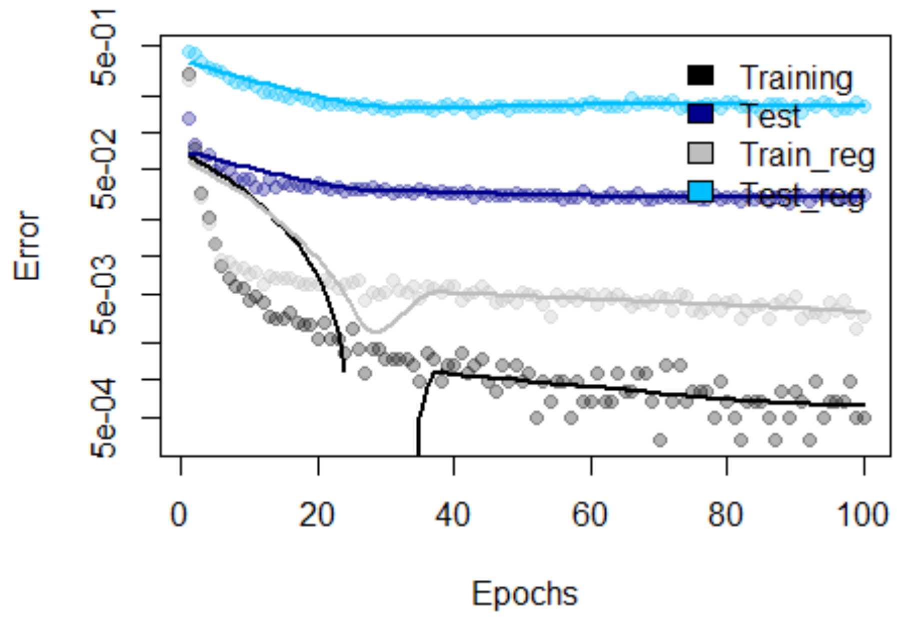
```r
#Inspecting the weights of the two models
#Get all weights
w_all <- get_weights(model1)
w_all_reg <- get_weights(model_reg1)
#Weights of first hidden layer
w <- w_all[[3]][1,]
w_reg <- w_all_reg[[3]][1,]
#Compare visually the magnitudes
par(mfrow = c(2,1), mar = c(2,2,0.5,0.5))
r <- range(w)
n <- length(w)
plot(w, ylim = r, pch = 19, col = adjustcolor(1,0.5))
abline(h = 0, lty = 2, col = "red")
segments(1:n, 0, 1:n, w)
plot(w_reg, ylim = r, pch = 19, col = adjustcolor(1,0.5))
abline(h = 0, lty = 2, col = "red")
segments(1:n, 0, 1:n, w_reg)
```



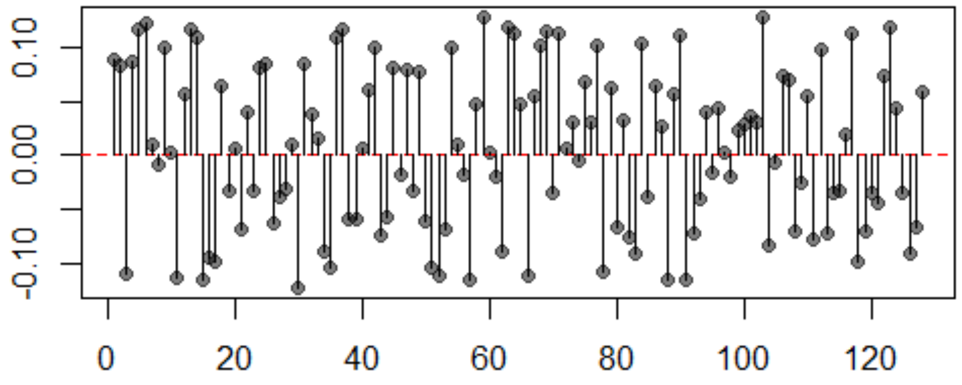
```
#Penalty-based regularization - 3 layers
#Training configuration after regularization
model_reg2 <- keras_model_sequential() %>%
  layer_dense(units=256,activation = "relu",input_shape=ncol(x_train),kernel_
regularizer=regularizer_l2(l=0.009))%>%   layer_dense(units=128,activation="r
elu",input_shape=ncol(x_train),kernel_regularizer=regularizer_l2(l=0.009))%>%
  layer_dense(units= 64,activation="relu",input_shape=ncol(x_train),kernel_re
gularizer=regularizer_l2(l=0.009))%>%
  layer_dense(units = 19, activation = "softmax") %>%
  compile(loss="categorical_crossentropy",optimizer=optimizer_sgd(), metrics
= "accuracy")

#Train and evaluate on test data at each epoch
fit_reg <- model_reg2 %>% fit(
           x_train, y_train1,
           validation_data = list(x_test, y_test1),
           epochs = 100, verbose = 1)

plot(fit_reg)

## `geom_smooth()` using formula 'y ~ x'
```

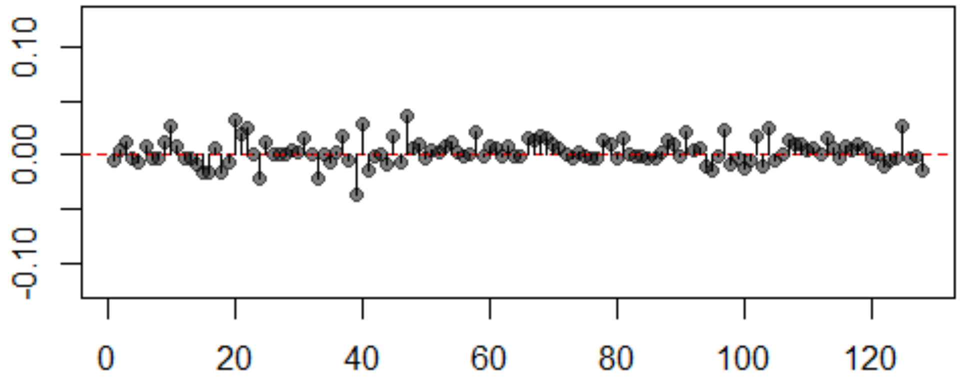


```
model4 <- model_reg2 %>%
evaluate(x_test, y_test1)
#Loss and Accuracy
model4

## $loss
## [1] 0.8105553
##
## $accuracy
## [1] 0.8355263

out <- 1 - cbind(fit2$metrics$accuracy,fit2$metrics$val_accuracy,fit_reg$metr
ics$accuracy,fit_reg$metrics$val_accuracy)

#Check performance
matplot(out, pch = 19, ylab = "Error", xlab = "Epochs",
        col = adjustcolor(cols, 0.3), log = "y")
matlines(apply(out, 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright",
        legend = c("Training","Test","Train_reg","Test_reg"),
        fill = cols, bty = "n")
```



```
apply(out, 2, min)

## [1] 0.000328958 0.026315808 0.002631605 0.146052659

#Inspecting the weights of the two models
#Get all weights
w_all <- get_weights(model2)
w_all_reg <- get_weights(model_reg2)

#Weights of first hidden layer
w <- w_all[[3]][1,]
w_reg <- w_all_reg[[3]][1,]

#Compare visually the magnitudes
par(mfrow = c(2,1), mar = c(2,2,0.5,0.5))
r <- range(w)
n <- length(w)

plot(w, ylim = r, pch = 19, col = adjustcolor(1,0.5))
abline(h = 0, lty = 2, col = "red")
segments(1:n, 0, 1:n, w)

plot(w_reg, ylim = r, pch = 19, col = adjustcolor(1,0.5))
abline(h = 0, lty = 2, col = "red")
segments(1:n, 0, 1:n, w_reg)
```





The accuracy with 2 hidden layers is 0.85 which is greater than that of 3 hidden layers which is 0.83, both after regularization, hence let us further perform model tuning on model with 2 hidden layers.

MODEL TUNING

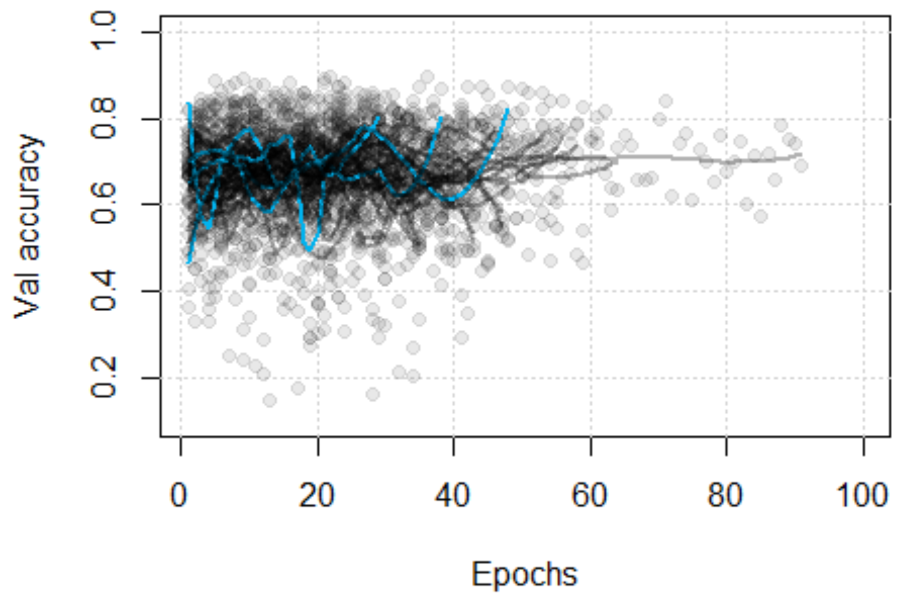
“tfruns” package is installed and loaded to effectively tune the model configuration. config_file.R:A separate R script file is used where we have the model instantiation and declaration of the flags that control the number of nodes within each layer and dropout rate. We have deployed a multilayer neural network using 2 hidden layers with ReLU (Rectified Linear Units) activation function and the output layer with softmax output activation. Grid Search: We have set a grid of values for both the hyperparameters we are interested in tuning, i.e. “number of nodes within each layer” and “dropout rate”, setting 4 values each for each hyperparameter for each layer, resulting in 4 X 4 X 4 X 4 = 256 possible model configurations to be evaluated. “tuning_run” function is used to perform an exhaustive grid search, running all combinations of the specified training flags defined in a pre-specified grid. 256 possible combinations will take hours of time to evaluate. So we have considered a random sub- sample of 20% of the total possible configurations, thus running only 51 models. We have also specified the directory name as “runs_example” where all the runs for 51 models will be stored.

```
library(tfruns)
#Splitting the test data in half for validation and model tuning
#and the other half for actual testing and evaluation of the generalized
```

```r
#predictive performance
set.seed(1234)
val <- sample(1:nrow(x_test), 760)
test <- setdiff(1:nrow(x_test), val)
x_val <- x_test[val,]
y_val <- y_test1[val,]
x_test <- x_test[test,]
y_test1 <- y_test1[test,]

library(jsonlite)
tf$reset_default_graph <- tf$compat$v1$reset_default_graph
#Setting a grid of values for the flags/hyperparameters
dense_units1_set <- c(256, 128, 64, 32)
dropout1_set <- c(0.0, 0.1, 0.2, 0.4)
dense_units2_set <- c(128, 64, 32, 16)
dropout2_set <- c(0.0, 0.1, 0.2, 0.4)
#Hyperparameter tuning
runs <- tuning_run("Configuration_File_project.R",
                    runs_dir = "runs_example",
                    flags = list(
                    dense_units1 = dense_units1_set,
                    dropout1 = dropout1_set,
                    dense_units2 = dense_units2_set,
                    dropout2 = dropout2_set),
                    sample = 0.2)

## 256 total combinations of flags (sampled to 51 combinations)

##
## > score <- model %>% evaluate(x_test, y_test1, verbose = 0)

##
## Run completed: runs_example/2020-05-01T18-28-00Z
```

Determine Optimal configuration: Now we have a folder named “runs_example” in our working directory which has all the results and information about all the 51 runs. “jsonlite” package is installed and loaded to get the scores stored in .json format in all the runs folders in “runs_example” directory. Two functions are used to explore the results - read_metrics (to extract the learning scores) and plot_learning curve (to plot the corresponding validation learning curves for all the sampled model configurations). read_metrics function:Learning scores for 51 runs are extracted using the function read_metrics. The output infers the different parameters of the learning scores, which includes loss during each epoch before convergence, accuracy, validation loss, validation accuracy, flags for each hyperparameter for each hidden layer, evaluation loss and evaluation accuracy. We have also calculated the validation accuracy for each epoch for each run having 51 combinations of tuned hyperparameters. It can be observed that the maximum validation accuracy for any run (out of 51 sampled runs) is 0.8961. Validation learning curve: A validation learning curve is a plot of model learning performance over number of iterations with different combinations of hyperparameters, i.e. it shows the curve of validation accuracy over number of epochs at convergence. In our case, we can see 4 runs colored in blue. These runs are the top 4 runs that have the highest validation accuracy at convergence and the convergence is mostly before 80 epochs. The curves show that there are many model configurations that led to a validation accuracy larger than 0.80, with few runs having even more validation accuracy.

```r
#Extracting values from the stored runs
read_metrics <- function(path, files =NULL)
{
  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]]$flags <- jsonlite::fromJSON(paste0(dir, "flags.json"))
  out[[i]]$evaluation <- jsonlite::fromJSON(paste0(dir,"evaluation.json"))
}
  return(out)
}

#Plotting the corresponding validation learning curves
plot_learning_curve <- function(x, ylab = NULL, cols = NULL, top = 3,
                                span = 0.4, ...)
{
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)
}

#Extracting results
out <- read_metrics("runs_example")
tfruns::ls_runs()

## Data frame: 51 x 28
##                                run_dir eval_loss eval_accuracy metric_loss
## 1  runs_example/2020-05-01T18-28-00Z    3.2064        0.6105      1.5630
## 2  runs_example/2020-05-01T18-27-12Z    3.9373        0.5513      0.7659
## 3  runs_example/2020-05-01T18-26-22Z    2.8230        0.7092      1.1843
## 4  runs_example/2020-05-01T18-25-33Z    4.0916        0.5868      2.5579
## 5  runs_example/2020-05-01T18-24-50Z    3.0166        0.6842      1.1172
## 6  runs_example/2020-05-01T18-24-08Z    2.3467        0.7355      1.5195
## 7  runs_example/2020-05-01T18-23-26Z    1.8018        0.7118      0.7476
## 8  runs_example/2020-05-01T18-22-44Z    2.1268        0.7263      0.9724
## 9  runs_example/2020-05-01T18-22-03Z    1.3103        0.7961      0.7002
## 10 runs_example/2020-05-01T18-21-19Z    3.1585        0.6474      0.4715
##    metric_accuracy metric_val_loss metric_val_accuracy
## 1           0.7416          3.2596              0.6289
## 2           0.9441          3.5576              0.5724
## 3           0.9096          2.7245              0.7211
## 4           0.8047          4.3262              0.5711
## 5           0.9093          2.7392              0.6921
## 6           0.8876          2.2669              0.7461
## 7           0.9446          1.8236              0.7303
## 8           0.8751          2.2140              0.7211
## 9           0.9483          1.2633              0.8026
## 10          0.9655          2.9851              0.6487
## # ... with 41 more rows
## # ... with 21 more columns:
## #   flag_dense_units1, flag_dense_units2, flag_dropout1, flag_dropout2,
## #   samples, batch_size, epochs, epochs_completed, metrics, model,
## #   loss_function, optimizer, learning_rate, script, start, end, completed
,
## #   output, source_code, context, type

#Extracting validation accuracy and plotting learning curve
acc <- sapply(out, "[[", "val_accuracy")
plot_learning_curve(acc, col = adjustcolor("black", 0.3),
                    ylim = c(0.10, 1.0), ylab = "Val accuracy", top = 4)
```



```
#Maximum validation accuracy
max_acc <- acc[which.max(acc)]
max_acc

## [1] 0.8961
```

RESULTS AND DISCUSSIONS:

Best model: We have extracted runs with validation accuracy larger than 0.84("res" object). The result shows a data frame with the hyperparameter combinations of number of nodes and the dropout rate for both the hidden layers. It also consists of validation and test accuracy associated to each run which have validation accuracy greater than 0.84. We have also extracted the model configuration and performance of the top 10 runs so as to check what accuracy or loss they behold with what combination of number of nodes and dropout rate for both hidden layers. But only 1st row had values followed by NAs hence I have displayed only 1st row. It can be observed that the best model (within the random sub-sample of the grid) is the one which has the highest validation accuracy (0.8961), i.e. 89.6% chances are there that the type of digit is predicted correctly on the validation set and eval_accuracy 0.807, i.e. 80.7% accuracy is there that predictions on the type of digit will be correct when tested on the test data. Generalization performance is quite good with almost 81% accuracy in predictions and after tuning, the best model should have 64 nodes in the first hidden layer and 64 nodes/units in the second hidden layer with dropout rate of 0.0 in both the hidden layers.

```
#Extracting runs with validation accuraAZ Zcy larger than 0.87
res <- ls_runs(metric_val_accuracy > 0.84,
runs_dir = "runs_example", order = metric_val_accuracy)
res

##  $ run_dir             : chr "runs_example/2020-05-01T17-54-10Z"
##  $ eval_loss           : num 1.42
##  $ eval_accuracy       : num 0.807
##  $ metric_val_accuracy: num 0.855
##  $ metric_loss         : num 0.928
##  $ metric_accuracy     : num 0.931
##  $ metric_val_loss     : num 1.3
##  $ flag_dense_units1   : int 256
##  $ flag_dense_units2   : int 128
##  $ flag_dropout1       : num 0.1
##  $ flag_dropout2       : num 0
##  $ samples             : int 7600
##  $ batch_size          : int 76
##  $ epochs              : int 100
##  $ epochs_completed    : int 31
##  $ metrics             : chr "(metrics data frame)"
##  $ model               : chr "(model summary)"
##  $ loss_function       : chr "categorical_crossentropy"
##  $ optimizer           : chr "<tensorflow.python.keras.optimizer_v2.adam.Ad
am>"
##  $ learning_rate       : num 0.01
##  $ script              : chr "m1.R"
##  $ start               : POSIXct[1:1], format: "2020-05-01 17:54:27"
##  $ end                 : POSIXct[1:1], format: "2020-05-01 17:54:53"
##  $ completed           : logi TRUE
##  $ output              : chr "(script ouptut)"
##  $ source_code         : chr "(source archive)"
##  $ context             : chr "local"
##  $ type                : chr "training"

#Best hyperparameter values
res <- res[,c(2,4,8:11)]
res[1,]

## Data frame: 10 x 6
##      metric_val_accuracy eval_accuracy flag_dense_units1 flag_dense_units2
## 1                 0.8553        0.8066               256               128


##      flag_dropout1 flag_dropout2
## 1              0.1             0
#Extracting run with maximum validation accuracy
res[which.max(res$metric_val_accuracy),]

##  $ eval_accuracy       : num 0.807
##  $ metric_val_accuracy: num 0.855
##  $ flag_dense_units1   : int 256
##  $ flag_dense_units2   : int 128
##  $ flag_dropout1      : num 0.1
##  $ flag_dropout2      : num 0
```

Configuration file saved as separate file named : Cofiguration_File_project.R

```
#Setting default flags

FLAGS <- flags(flag_integer("dense_units1", 256),

            flag_integer("dense_units2", 128),

            flag_numeric("dropout1", 0.1),

            flag_numeric("dropout2", 0.1))


#Model configuration

model <- keras_model_sequential() %>%

  layer_dense(units = FLAGS$dense_units1, input_shape = ncol(x_train),

            activation = "relu", name = "layer_1",

            kernel_regularizer = regularizer_l2(0.01)) %>%

  layer_dropout(rate = FLAGS$dropout1) %>%

  layer_dense(units=FLAGS$dense_units2, activation="relu",name="layer_2",

            kernel_regularizer = regularizer_l2(0.01)) %>%

  layer_dropout(rate = FLAGS$dropout2) %>%

  layer_dense(units = ncol(y_train1), activation = "softmax",

            name = "layer_out") %>%

  compile(loss = "categorical_crossentropy", metrics = "accuracy",

         optimizer = optimizer_adam(lr = 0.01),)


#Training and evaluation

fit <- model %>% fit(
  x = x_train, y = y_train1,
  validation_data = list(x_val, y_val),
  epochs = 100,
  batch_size = 76,
  verbose = 1,
  callbacks = callback_early_stopping(
    monitor = "val_accuracy", patience = 20))


#Storing accuracy on test set for each run
score <- model %>%
evaluate(x_test,y_test1,verbose=0)
```

CONCLUSION

Multilayer neural network of 2 and 3 hidden layers is employed for the activity recognition data , after regularization of both the models , model with 2 hidden layer came out to be best with accuracy approximately equal to 0.85, further model tuning was performed on multilayer neural network with 2 hidden layers for which the accuracy increased to 0.89.

REFERENCE

No references as such.