Daily/sports activity recognition from movement sensor data

Tanay Sawant 19203264

26/04/2020

Abstract:

With the lifestyle getting busier day by day, people tend to lose to keep a track of things in their life. Having a healthy lifestyle, or to term it as "fitness" is the top priority of the young generation and to achieve that, they tend to do more physical activities and sports of their choice for a given set of time as per their convenience. In this report, 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. We are going to deploy a predictive model which can be employed for daily/sports activity recognition from movement sensor data.

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? (Research Question)

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 problem are we dealing with?

We are provided with a 3d data where the movements on torso arms and legs are recorded. We are using Neural Network to build a model which is used to predict the above movements.

What type of data do we have?

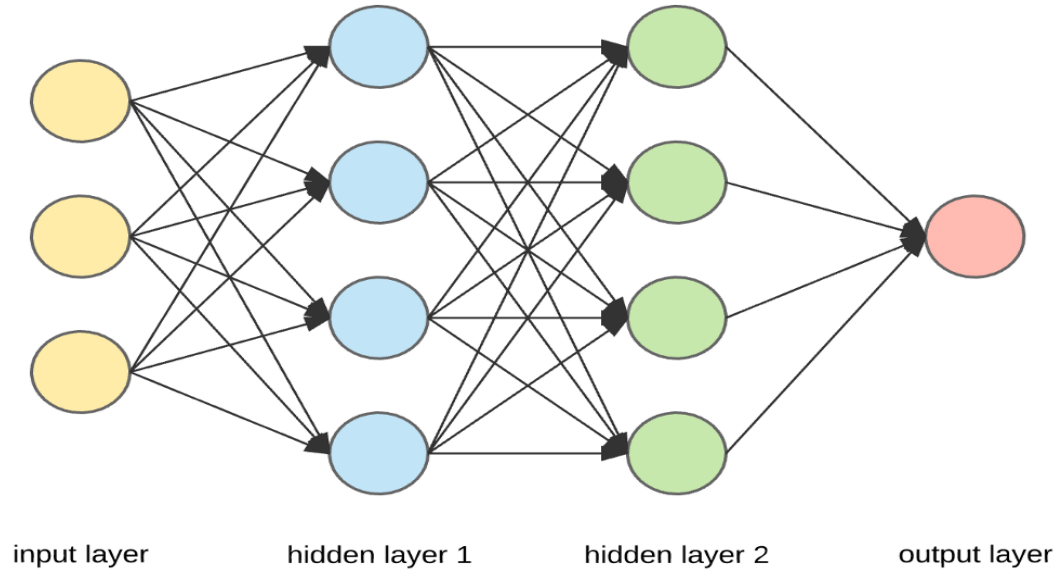
The data file data_activity_recognition.RData contains training and test data. The training data x_train includes 400 signal segments for each activity (total of 7600 signals), while the test data x_test includes 80 signal segments for each activity (totalof1520signals). The activity labels are in the 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.

Methods:

The main task for us is to try different model configurations and select the best model to be used for predictions on the given data set. So, let's first understand what is a model configuration and how it is represented.

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.



Architecture of Neural Network

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.

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 pre-processing.

Loading and pre-processing:

- We load the libraries "keras","tfruns" which are used for model building and tuning the model configurations.
- After loading the data, we observe that our data is divided into training set and test set respectively.
- Our data being 3 dimensional, we change it's dimensionality using array_reshape to a matrix of 125*45 for the x_train and x_test data.
- We normalise the input variables with the normalising function and transform the target variable to a categorical via one-hot encoding to get our dataset ready.

```
library(keras)
library(tfruns)

load("data_activity_recognition.RData")
dim(x_train) #observing the dimension of x_train.

## [1] 7600  125   45
```

```
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)))
dim(x_train) #observing the dimension of x_train after reshaping.

## [1] 7600 5625

y_train=data.matrix(data.frame(y_train))
y_test=data.matrix(data.frame(y_test))

#one-hot encoding.
y_train=to_categorical(y_train-1)
y_test=to_categorical(y_test-1)

#normalising the range:
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)
```

After the pre-processing, knowing that the dimension of our dataset is still large, we reduce it by using Principal Component Analysis (PCA).

About 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. Function prcomp can be directly applied to the train data. 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 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.

```
#dimension reductionality:
pca <-prcomp(x_train)
prop <-cumsum(pca$sdev^2)/sum(pca$sdev^2)# compute cumulative proportion of v
ariance
Q <-length( prop[prop<0.99] )
x_train <-pca$x[,1:Q]
x_test <-predict(pca, x_test)[,1:Q]
V =ncol(x_train)
```

By applying PCA, our reduced number of features are 1134 from 5625 which is a good amount of decrease.
After doing PCA, we store the reduced number of columns of x_train in V which will be used as the input_shape variable in our models later.

Modelling:

We start with some of the basic models to observe how our data is training on models with different layers. First, we construct two model, first model is with 2 hidden layers and second model is with 3 hidden layers. The reason we do this, is to observe is there is any overfitting and with an additional layer,is the accuracy improving .We develop these two models without any regularization and check its performance on training data and validate it on test data.

Construction of model with 2 & 3 layers:

1) Here we have 2 hidden layers with "ReLU" activation and one output with "softmax" output activation. (3 hidden layers with "ReLU" activation)
2) In the first layer we consider 1134 units, as the input units and the input shape as V.
3) We keep the number of units of second hidden layer as 800 (500 for 3rd layers in 3 hidden layer model)
4) In the training configuration, we consider standard stochastic gradient descent for optimization, and categorical cross-entropy as error function, with accuracy as performance measure. We develop our model using the %>% operator to set everything in one go.

Model with 2 hidden layers:

```
model<-keras_model_sequential()%>%
  layer_dense(units=1134,activation="relu",input_shape=V)%>%
  layer_dense(units=800,activation="relu")%>%
  layer_dense(units=19,activation="softmax") %>%
  compile(loss="categorical_crossentropy",optimizer=optimizer_sgd(),metrics="
accuracy")

fit<-model%>%fit(x=x_train,y=y_train,
    validation_data=list(x_test,y_test),
    epochs=60,verbose=1)
```

Model Accuracy:

```
tail(fit$metrics$val_accuracy,1)

## [1] 0.9407895
```

We observe the accuracy of the model to be **94%.**

Model with 3 hidden layers:

```
model1<-keras_model_sequential()%>%
  layer_dense(units=1134,activation="relu",input_shape=V)%>%
  layer_dense(units=800,activation="relu")%>%
  layer_dense(units=500,activation="relu")%>%
  layer_dense(units=19,activation="softmax") %>%
  compile(loss="categorical_crossentropy",optimizer=optimizer_sgd(),metrics="
accuracy")

fit1<-model1%>%fit(x=x_train,y=y_train,
    validation_data=list(x_test,y_test),
    epochs=60,verbose=1)
```

Model accuracy:

```
tail(fit1$metrics$val_accuracy,1)

## [1] 0.9368421
```

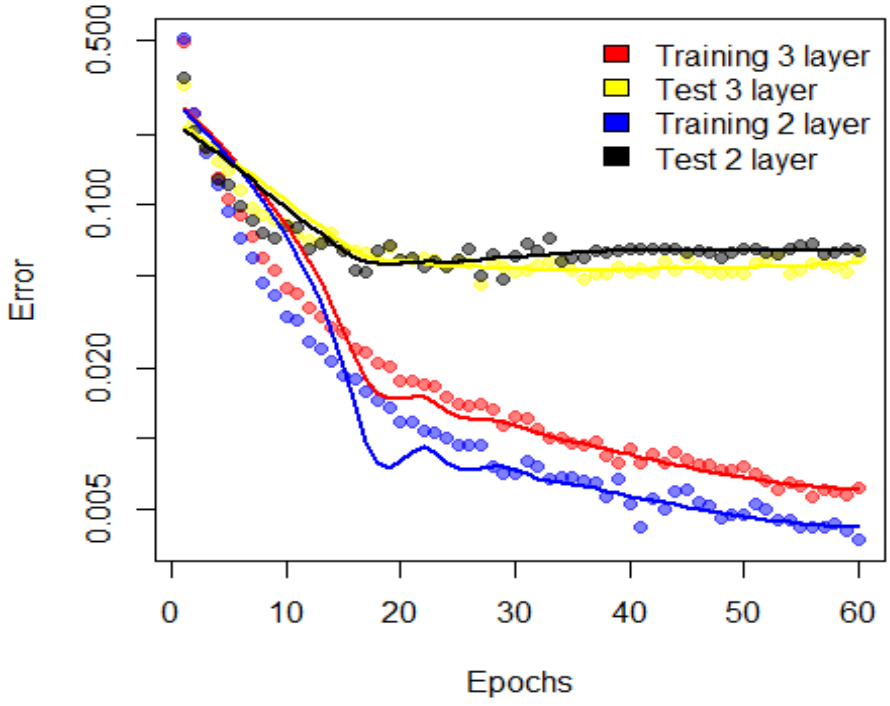
We observe the accuracy of the model to be **93.7%.**

Plot and observe:

In order to compare these two models, we plot the training and testing errors of both the models in a graph to observe which model performs better on the training and test set.

```
smooth_line<-function(y){
  x<-1:length(y)
  out<-predict(loess(y~x))
  return(out)
}

cols <- c("red", "yellow","blue","black")
out <-1-cbind(fit$metrics$accuracy,fit$metrics$val_accuracy,fit1$metrics$accu
racy,fit1$metrics$val_accuracy)# check performance
matplot(out,pch =19,ylab ="Error",xlab ="Epochs",col =adjustcolor(cols,0.5),l
og ="y")
matlines(apply(out,2, smooth_line),lty =1,col =cols,lwd =2)
legend("topright",legend =c("Training 3 layer","Test 3 layer","Training 2 lay
er","Test 2 layer"),fill =cols,bty ="n")
```




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

```
## [1] 0.005657911 0.045394719 0.003684223 0.048026323
```

We observe that, our both the models have similar testing error around 0.05, and the model with 2 and 3, both tend to show overfitting on training data as the classification error is tending towards zero for both the models (0.0056 and 0.0048) when the number of epochs increase.

Result 1: 2 hidden layers or 3 hidden layers? Overfitting?

1.) We observe **overfitting.**

2.) Since there is no much difference in the model accuracies with 2 hidden layers and 3 hidden layers, we decide to do our further analysis on a model with **2 hidden layers as it has reduced the model complexity.**

How to overcome overfitting?

We could try to improve the generalization performance by tackling overfitting. To this purpose, we can employ a penalty-based regularization, and we use weight decay. Regularization can be implemented in keras at the level of model configuration when specifying the characteristics of each layer. The argument used to set a regularization term applied to the weights of each layer is kernel_regularizer. We use the function regularizer_l2 to implement weight decay, with argument l=0.009 used to specify the

magnitude of λ and check if overfitting is removed and if there is an increase in the accuracy.

Model with 2 layers with L2 regularization:

```
model_reg2 <-keras_model_sequential()%>%
  layer_dense(units =1134,activation ="relu",input_shape =V,
              kernel_regularizer =regularizer_l2(l =0.009))%>%
  layer_dense(units =800,activation ="relu",
              kernel_regularizer =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_reg2 <-model_reg2%>%
  fit(x =x_train,y =y_train,
      validation_data =list(x_test, y_test),
      epochs =60,verbose =1)
```

Model accuracy:

```
tail(fit_reg2$metrics$val_accuracy,1)

## [1] 0.9427631
```

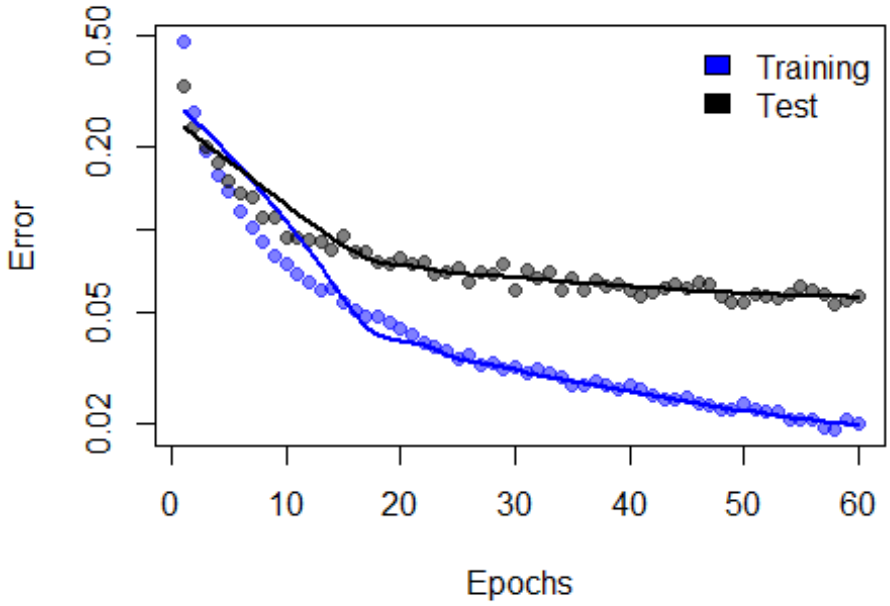
The accuracy of the model is found to be **94.3%.** This accuracy obtained is the highest till now in all the 3 models we have observed.

Plot and observe:

We plot the training and testing error for the model in a graph and observe.

```
# plot to observe the improvement:

cols <-c("blue","black")
out <-1-cbind(fit_reg2$metrics$accuracy,fit_reg2$metrics$val_accuracy)# check
performance
matplot(out,pch =19,ylab ="Error",xlab ="Epochs",col =adjustcolor(cols,0.5),l
og ="y")
matlines(apply(out,2, smooth_line),lty =1,col =cols,lwd =2)
legend("topright",legend =c("Training","Test"),fill =cols,bty ="n")
```




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

```
## [1] 0.01894736 0.05328947
```

With the above graphs, we can see that we have removed overfitting. The minimum value for the training error is found to be 0.018 which has improved from 0.0056 and the test error is still around 0.05.

Result 2: Impact of regularization?

1.) **Our issue of overfitting is solved.** We can observe from the graph that the model is no more overfitting as the training error is not approaching 0 with the same rate as the number of epochs increase.
2.) We observe that **there is no much change in the accuracy**, a slight increase of 0.3% with the model configuration selected.

Now that we have move forward with a model of 2 hidden layers over a model of 3 hidden layers, resolved the issue of overfitting, we did this all on a random selection of the units, the hyper-parameters. We now tune the nodes and the hyper-parameter of the weight decay regularization to check **if we can get a model with a better combination and which gives us better accuracy.**

What is tuning?

The ideal way of tuning is we split the test data in validation and testing, after doing so we use the validation set to tune the model for various parameters and then test the data.

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.

Why are we tuning?

We do not split the test data because we are tuning just to find a better combination of the nodes and hyper-parameters IF ANY!

Tuning sets:

This chunk of code below is the parameter set used for tuning the weight decay regularization (lambda) and the nodes of each layer (2 hidden layers).

```
#sets for tuning:

l1_set <-c(1134,1100,1000)
l2_set<-c(1000,800,600)
lambda_set <-c(0,exp(seq(-6,-4,length =5)))
```

About tuning_run:

- This function is used to run all combinations of the specified training flags. By default, the function performs an exhaustive grid search, running all combinations of settings/hyperparameters defined in a pre-specified grid.
- 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 be considered. Argument runs_dir is employed to specify the directory in which all runs will be stored. We extract a sample of 0.5 from all the combinations of model, which comes down to 27 models from a total of 54 models.

```
runs<-tuning_run("model.R",
                 runs_dir = "run3",
                 flags = list(lambda=lambda_set,
                              l1=l1_set,
                              l2=l2_set),sample = 0.5)

## 54 total combinations of flags (sampled to 27 combinations)
```

Extracting the stored results:

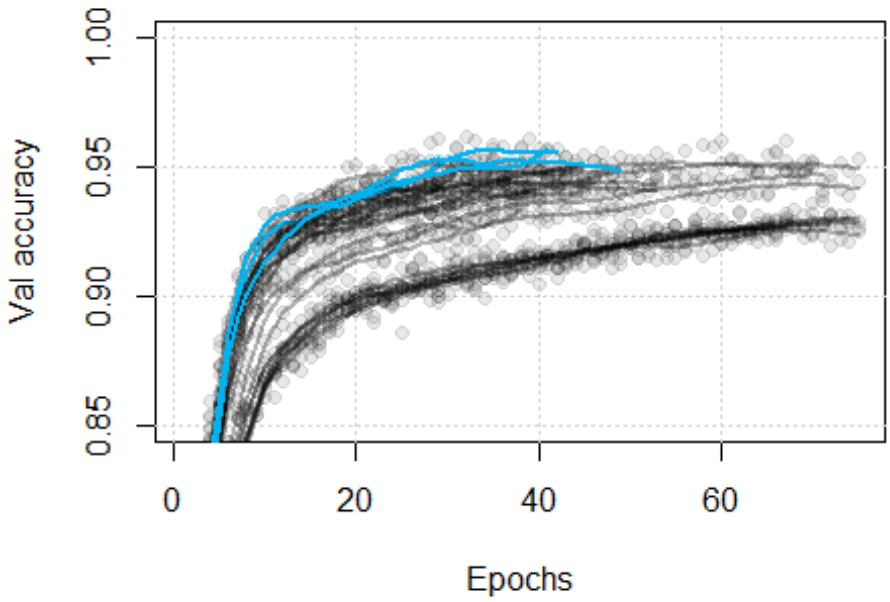
With the “run3” storing the results and information obtained for various runs, we extract the results from that folder which is stored in the local directory. The two functions below will be used to extract values from the stored runs and plot the corresponding validation learning curves. Note that package jsonlite is required as scores are stored .json format. Note that runs have different number of epochs because of early stopping.

```r
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)
}


#We now extract the learning scores and produce the validation accuracy learn
ing curve. Note that runs have different number of epochs because of early st
opping.

out <-read_metrics("run3")
# 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 accuracy",top =3)
```



In the above plot, the top 3 runs are coloured in blue as those with highest validation accuracy at convergence. We observe that most of model configurations led to a validation of accuracy ranging in 0.90-0.95

Extracting validation accuracies >0.93:

The function ls_runs can be used to list and subset runs according to different criteria. We extract runs with validation accuracy larger than 0.93.

```
#Extracting the result:
res <- ls_runs(metric_val_accuracy > 0.93,
runs_dir = "run3", order = metric_val_accuracy)
```

Observing the output:

```
#Observing the dataframe based on highest val_accuracy to check if we get any
better combination after tuning:
res <-res[,c(2,4,6:8)]
res

## Data frame: 19 x 5
##    metric_val_accuracy metric_accuracy flag_lambda flag_l1 flag_l2
## 1               0.9566          0.9842      0.0041    1100     800
## 2               0.9566          0.9855      0.0041    1000     800
## 3               0.9539          0.9854      0.0041    1000     600
## 4               0.9533          0.9866      0.0067    1100     800
```

```
## 5                0.9513          0.9907      0.0000    1134    1000
## 6                0.9500          0.9896      0.0041    1100    1000
## 7                0.9493          0.9901      0.0025    1134     600
## 8                0.9487          0.9926      0.0000    1000     600
## 9                0.9467          0.9859      0.0041    1100     600
## 10               0.9461          0.9922      0.0000    1000    1000
## # ... with 9 more rows
```

We observe that we get a data frame of 19*5 whose validation accuracy is greater than 0.93. This might suggest that a few good models have been selected in the 27 models which are tuned. The best validation accuracy is **0.9566(96% accuracy approximately) with a lambda value of 0.0041 and the units of the layers being 1100 and 800 respectively.**

Conclusion:

In order to answer to the main research question, we summarize our findings:
We have done data pre-processing, compared 2 models, regularized a model and tuned it as well.

1.) Model with 2 hidden layers is preferred than a model with 3 hidden layers as both the models give about the similar output.
2.) Overfitting was observed in both the models, and moving forward with a 2 hidden layer model, we remove it with L2 regularization and find that there is no much improvement int the accuracy.
3.) Just to find if we have any better combinations of nodes and hyper-parameters, we tune the model. We get the best model with lambda value 0.0041 and units of the layers being 1100 and 800 respectively giving 96% accuracy from the sub samples generated.

With the predictive model which can be employed for daily/sports activity recognition from movement sensor data, with an accuracy rate of 96% obtained from the best sub sampled model, we can say that which exercise for which body part is best suited for an individual.

References:

1.) Lecture notes and labs for Machine Learning & AI.
2.) https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a