Sports Activity Identification

By Shubhang Periwal 19201104

Abstract

I will be comparing multiple neural network models, some with regularization, some without. Initially, we will perform multiple data pre-processing techniques to reduce the number of dimensions of the data such as correlation and PCA. These methods helped me reduce from 5625 columns to 656 columns while maintaining 90% variance along the data. This would be followed by implementation of several neural network architectures and using convolution neural networks, which considers the input to be an image of the size 125 by 45. Alongside with multiple configuration files to try out as many combinations as possible.

Introduction

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. 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 5- second 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.

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 (total of 1520 signals). 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.

Pre-processing

Every machine learning algorithm involves pre-processing of data, which involves reduction of dimensions, replication, reduction of noise in data, preparation of data to give as an input in the model. For this step, I am using correlation between two variables followed by PCA(Principle Component Analysis) for dimension reduction.

We also need to convert our predictor variable to one-hot encoding in order to place it as an output in our neural network.

Model Learning

I am using this pre-processed data as an input in multiple neural network architectures, such as models with 3 hidden layers, 4 hidden layers, using flags compare multiple architectures together followed by regularization, modification of hyper parameters and then using Convolution Neural Networks by converting the input data into 4 dimensions to get the optimum result.

Model Validation

In total, I created a set of over 500 neural networks, then sampled a few of them. Comparison of different models was made, and the best architecture was finally selected for a further fine tuning.

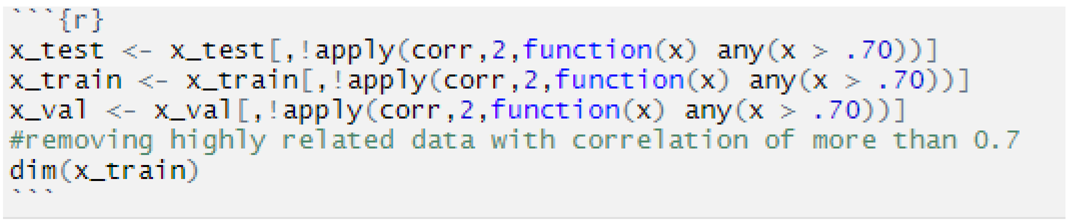
Methods

Correlation Matrix

A correlation matrix is a table showing correlation coefficients between variables. Each cell in the table shows the correlation between two variables.

In perspective of storing data in databases, storing correlated features is somehow similar to storing redundant information which it may cause wasting of storage and also it may cause inconsistent data after updating or editing tuples. If we add so much correlated features to the model we may cause the model to consider unnecessary features and we may have curse of high dimensionality problem.

I have removed all variables with more than 70% correlation with the one's that I've kept. This helps greatly in dimension reduction of the data, leading to a lesser number of parameters in the model.

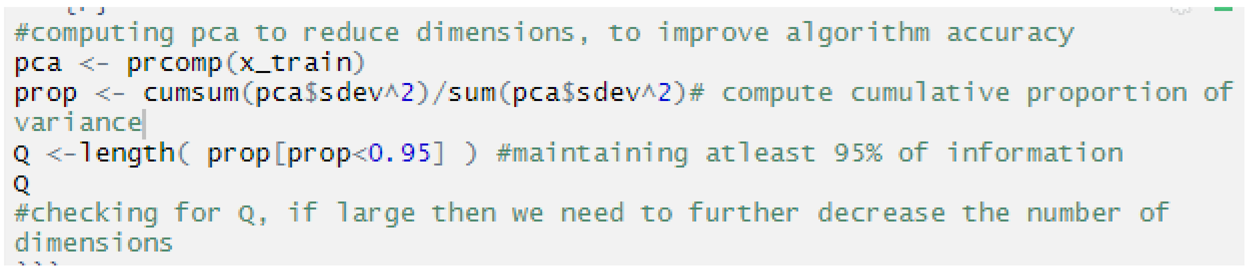


~~~
```{r}
x_test <- x_test[,!apply(corr,2,function(x) any(x > .70))]
x_train <- x_train[,!apply(corr,2,function(x) any(x > .70))]
x_val <- x_val[,!apply(corr,2,function(x) any(x > .70))]
#removing highly related data with correlation of more than 0.7
dim(x_train)
```
~~~

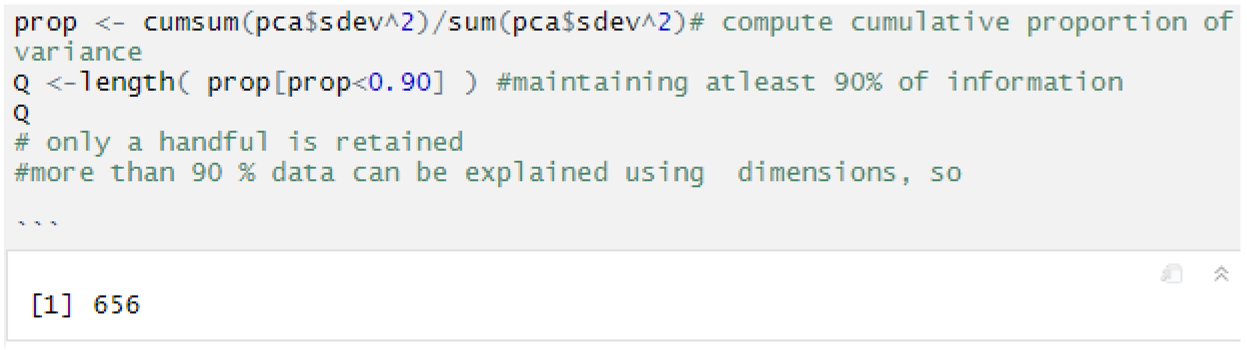
```
[1] 7000 1729
```

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.



```r
#computing pca to reduce dimensions, to improve algorithm accuracy
pca <- prcomp(x_train)
prop <- cumsum(pca$sdev^2)/sum(pca$sdev^2)# compute cumulative proportion of variance
Q <-length( prop[prop<0.95] ) #maintaining atleast 95% of information
Q
#checking for Q, if large then we need to further decrease the number of dimensions
```



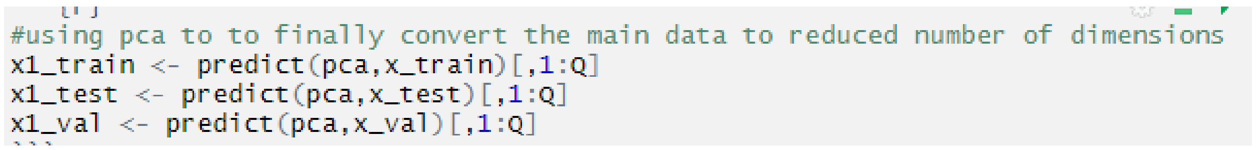
~~~r
prop <- cumsum(pca$sdev^2)/sum(pca$sdev^2)# compute cumulative proportion of variance
Q <-length( prop[prop<0.90] ) #maintaining atleast 90% of information
Q
# only a handful is retained
#more than 90 % data can be explained using  dimensions, so

```
~~~

```
[1] 656
```

The first output for 95% was over 900 dimensions, so I reduced the variation maintained by the model to 90% coming down to 656 dimensions.

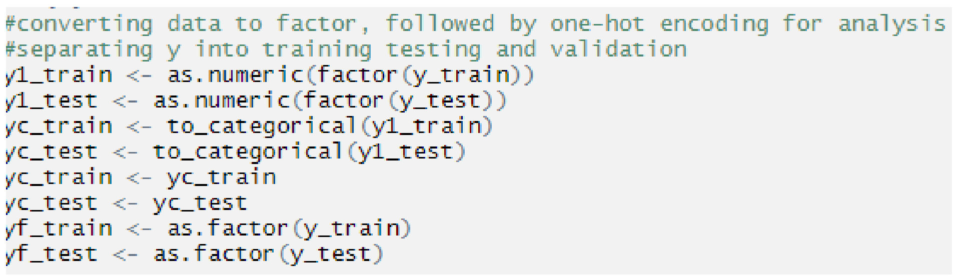
The next step is to convert data into lower dimension format for further training.



```r
#using pca to to finally convert the main data to reduced number of dimensions
x1_train <- predict(pca,x_train)[,1:Q]
x1_test <- predict(pca,x_test)[,1:Q]
x1_val <- predict(pca,x_val)[,1:Q]
```

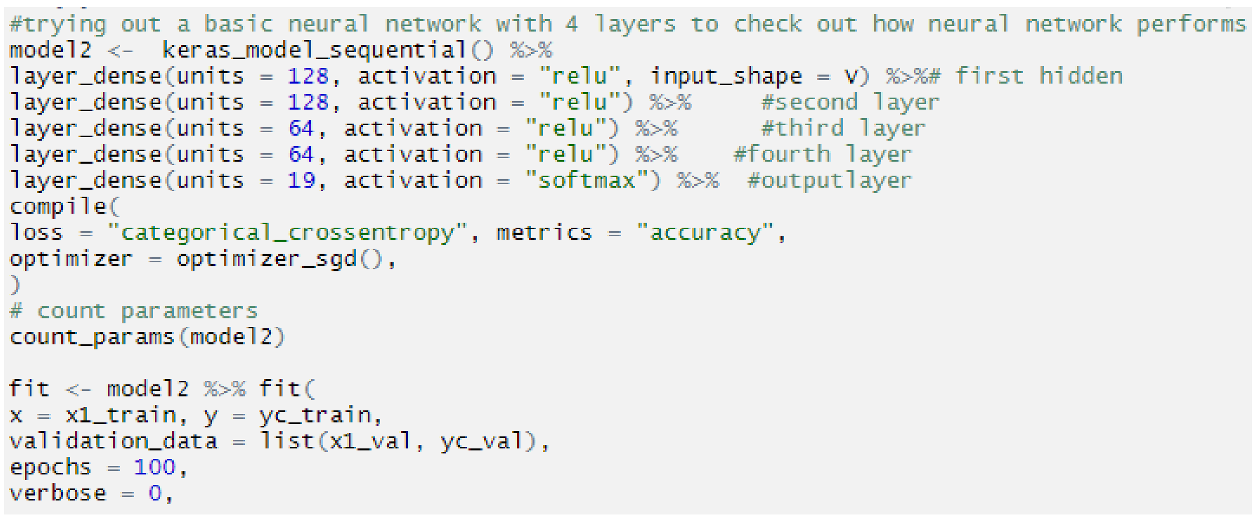
Pre-processing Predictor Variable

We next need to convert the predictor variable to categorical, so that we can use it to train our neural network model.



```r
#converting data to factor, followed by one-hot encoding for analysis
#separating y into training testing and validation
y1_train <- as.numeric(factor(y_train))
y1_test <- as.numeric(factor(y_test))
yc_train <- to_categorical(y1_train)
yc_test <- to_categorical(y1_test)
yc_train <- yc_train
yc_test <- yc_test
yf_train <- as.factor(y_train)
yf_test <- as.factor(y_test)
```

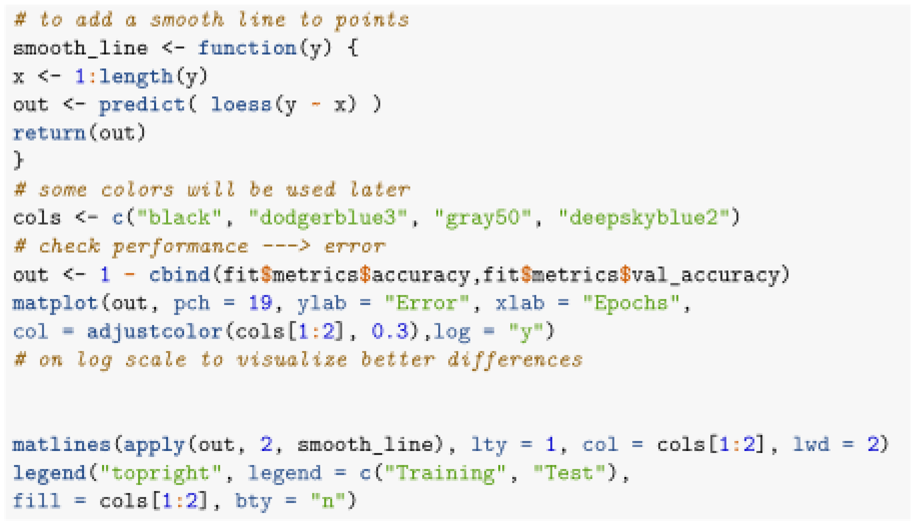
Creating the First Neural Network Architecture



```
#trying out a basic neural network with 4 layers to check out how neural network performs
model2 <-  keras_model_sequential() %>%
layer_dense(units = 128, activation = "relu", input_shape = v) %>%# first hidden
layer_dense(units = 128, activation = "relu") %>%      #second layer
layer_dense(units = 64, activation = "relu") %>%       #third layer
layer_dense(units = 64, activation = "relu") %>%     #fourth layer
layer_dense(units = 19, activation = "softmax") %>%  #outputlayer
compile(
loss = "categorical_crossentropy", metrics = "accuracy",
optimizer = optimizer_sgd(),
)
# count parameters
count_params(model2)

fit <- model2 %>% fit(
x = x1_train, y = yc_train,
validation_data = list(x1_val, yc_val),
epochs = 100,
verbose = 0,
```

The next step is to plot a curve to compare and compute the accuracy of the model. This is also necessary to check whether the model is underfitting or over fitting.



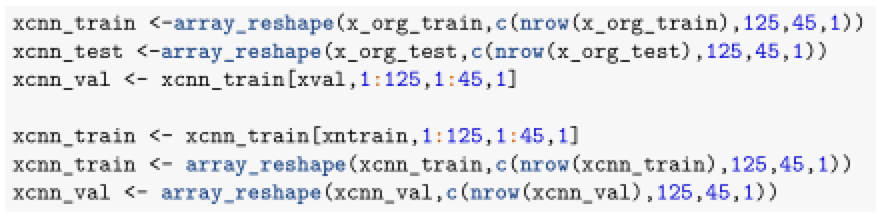
```
# to add a smooth line to points
smooth_line <- function(y) {
x <- 1:length(y)
out <- predict( loess(y ~ x) )
return(out)
}
# some colors will be used later
cols <- c("black", "dodgerblue3", "gray50", "deepskyblue2")
# check performance ---> error
out <- 1 - cbind(fit$metrics$accuracy,fit$metrics$val_accuracy)
matplot(out, pch = 19, ylab = "Error", xlab = "Epochs",
col = adjustcolor(cols[1:2], 0.3),log = "y")
# on log scale to visualize better differences


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")
```

Followed by this I have also created a model with same architecture but regularized, in order to show how my model performs and differences between regularized and non-regularized model.

<u>CNN (Convolution Neural Network)</u>

For creating this network, I have used the original un processed data, and converted the input matrix into a 4 dimension matrix in order to comply with the dimension requirement of the model and also fed data in the form of a 2-d matrix (similar to how images are stored) so that this network best identifies the class.



```
xcnn_train <-array_reshape(x_org_train,c(nrow(x_org_train),125,45,1))
xcnn_test <-array_reshape(x_org_test,c(nrow(x_org_test),125,45,1))
xcnn_val <- xcnn_train[xval,1:125,1:45,1]

xcnn_train <- xcnn_train[xntrain,1:125,1:45,1]
xcnn_train <- array_reshape(xcnn_train,c(nrow(xcnn_train),125,45,1))
xcnn_val <- array_reshape(xcnn_val,c(nrow(xcnn_val),125,45,1))
```

- A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.
- The innovation of convolutional neural networks is the ability to automatically learn a large number of filters in parallel specific to a training dataset under the constraints of a specific predictive modeling problem, such as image classification. The result is highly specific features that can be detected anywhere on input images.
- The next model is a similar model but with regularization.

Other Models

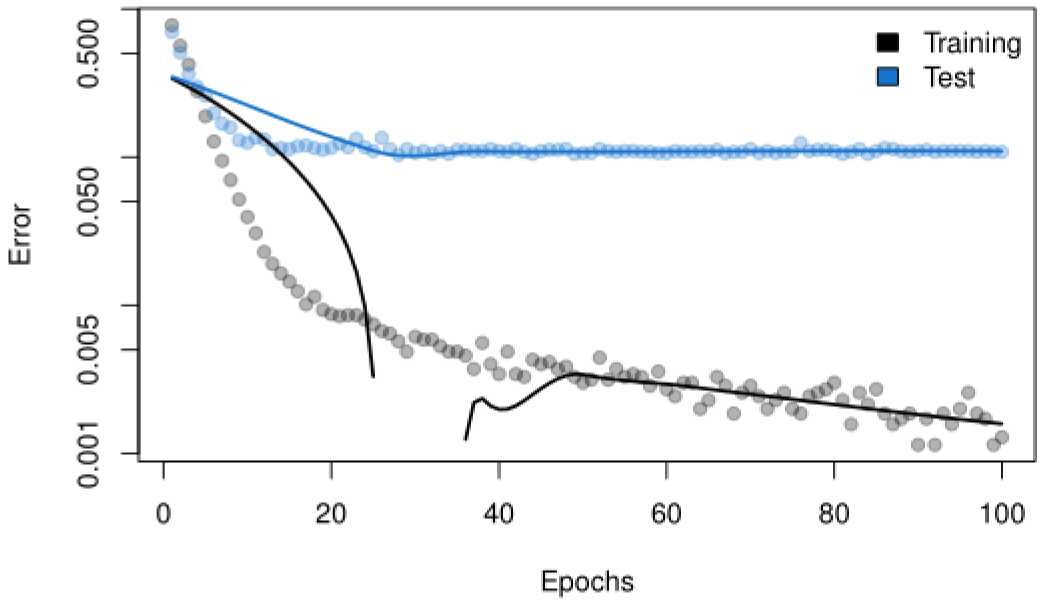
- I have tried to give different kind of shapes to my neural network, with the third layer having as low as 8 nodes.
- I have also given different combinations of dropouts with 0 being one of them, to check whether no dropout improves the model or not.
- The next model, takes the architecture of the previous model, after which I try tuning the values of lambda and other tuning parameters in order to get the best possible combination.

Results and Discussions

- After converting the entire data set into 2 dimensions, it has 5625 features, so we need to reduce this set of features using correlation matrix and PCA as discussed above.
- After removing variables with correlation higher than 70%, I am left with 1729 features.
- Then I perform PCA on the remaining features in order to reduce further excessive dimensionality.
- After PCA, I am trying to maintain 90% of the original variance which leaves me with 656 features.
- I performed removal by correlation before PCA so that PCA does not get affected by highly correlated features.

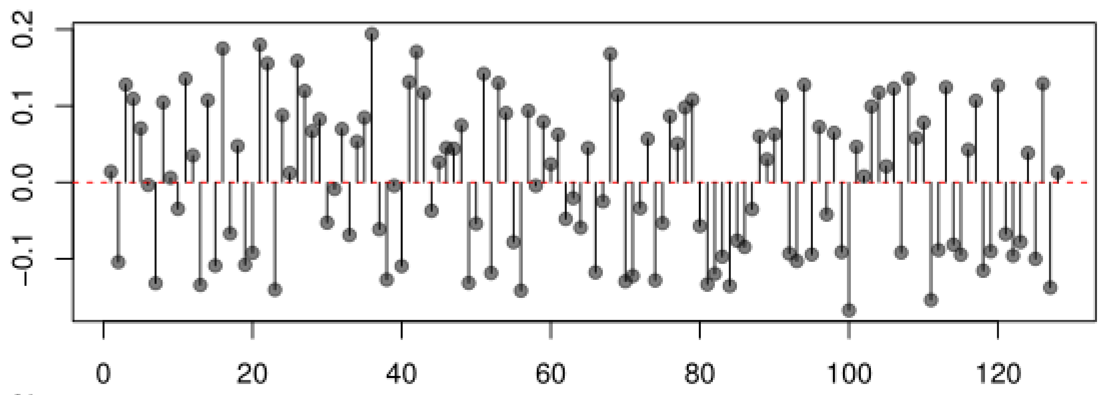
Model 1

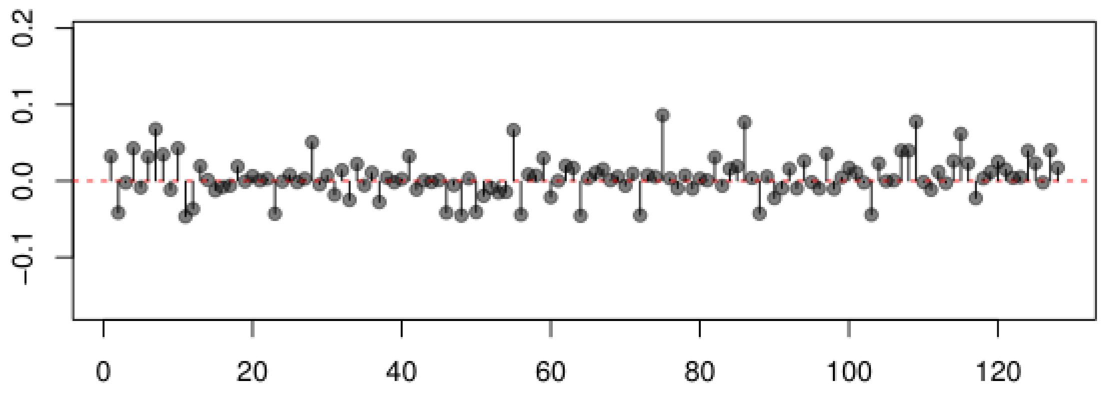
The first model gives the accuracy of 88.55%, which is really good as this classification was done among 19 classes.



<u>Model 2</u>

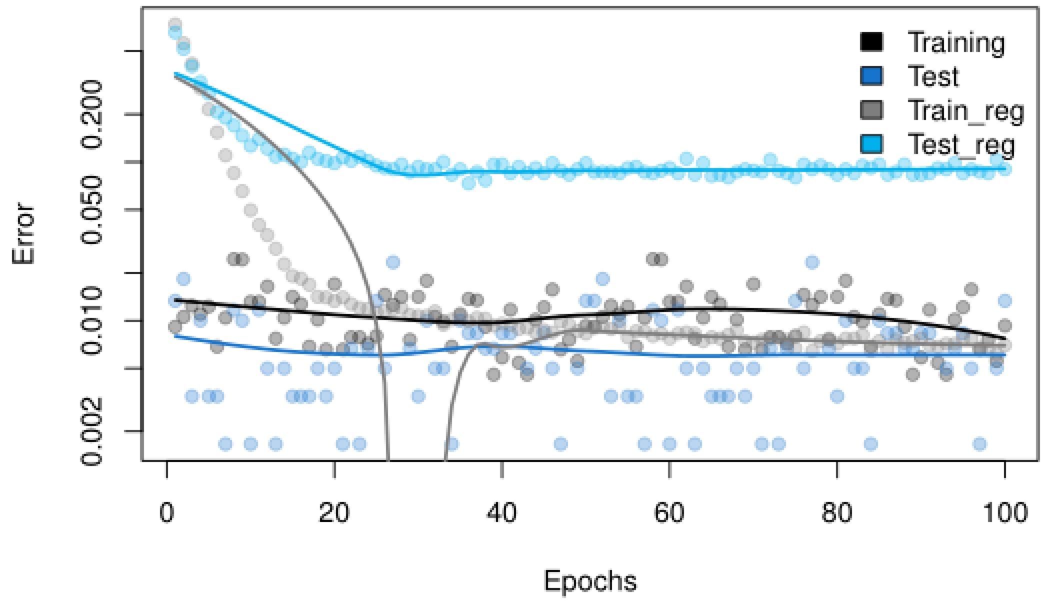
This model has same principle architecture as the first model, but with regularization to prevent overfitting. This model has a higher accuracy of 91% on the test set.





<u>CNN (Convolution Neural Network)</u>

- For this, the number of dimensions of the original data was increased by 1.
- The entire dataset of dimensions 7600 x 125 x 45 x1 was fed into the network, where the dimension of the input layer was 125 x 45 (matrix form). This data was not reduced by PCA or correlation so that we do not lose any features of this image.
- This is the best performing model with accuracy over 98.6% for the normal model as well as the regularized model.

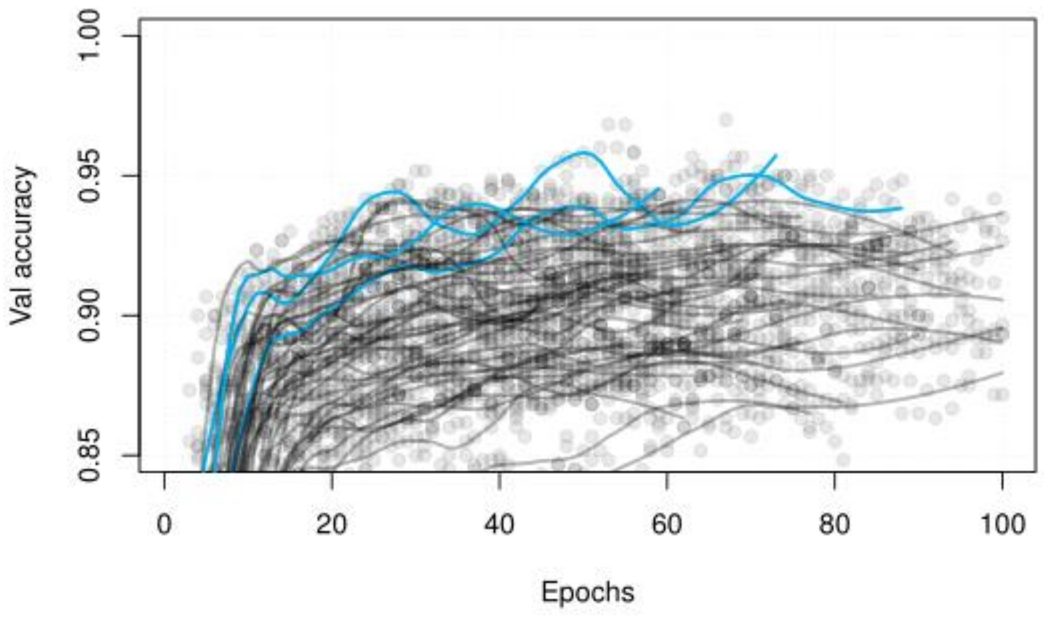


We can see its performance in the above figure.

Modifying Architecture

```
#setting a grid of values for the flags/hyperparameters of interest:
hdlayer1 <- c(128,64,32)
dropout1 <- c(0,0.1,0.3)
hdlayer2 <- c(64,16)
dropout2 <- c(0,0.2)
hdlayer3 <- c(64,32,16)
dropout3 <- c(0,0.1)
# total combinations 3 x 3 x 2 x 2 x 3 x 2 = 216
```

Out of these 216 combinations, I tried out a small sample of 21 architectures in order to retrieve the best possible combination.



The best combination was with

- Hidden Layer 1: 128 nodes
- Hidden Layer 2: 64 nodes
- Hidden Layer 3: 32 nodes
- Drop out 1: 0
- Drop out 2: 0
- Drop out 3: 0

This model had an accuracy of 95% on the testing set.

After this I created another set of flags, with nodes fixed as above, to try out different hyperparameter features.

```
# run ------------------------------------------------------------
dropout_set <- c(0, 0.3, 0.4, 0.5)
lambda_set <- c(0, exp( seq(-6, -4, length = 9) ))
lr_set <- c(0.001, 0.002, 0.005, 0.01)
bs_set <- c(0.005, 0.01, 0.02, 0.03)*nrow(x1_train)

runs <- tuning_run("mlai_2.R",
                   runs_dir = "nn2",
                   flags = list(
                     dropout = dropout_set,
                     lambda = lambda_set,
                     lr = lr_set,
                     bs = bs_set
                   ),
                   sample = 0.02)
```

The best result obtained was with lambda (weight decay) equal to 0.0052, batch size as 210, Learning rate as 0.001 and drop out rate as 0.

Conclusion:

The convolution neural network performs best on this data set, when compared to other architectures, when the entire data is fed into the network. This shows that, CNN can be used for different data sets such as this one. Sampling is a good technique to improve results in a very short amount of time.

Appendix

~~~~
---

title: "ML and AI project"

author: "Shubhang Periwal 19201104"

date: "4/22/2020"

output:

  pdf_document: default

  word_document: default

---



```{r, warning=FALSE}

data <- load("data_activity_recognition.RData")

tensorflow::tf$random$set_seed(0)

set.seed(19201104)

```



```{r, warning=FALSE}

library(keras)
~~~~

~~~~
library(tfruns)
library(reticulate)
library(jsonlite)
library(dplyr)
library(Rtsne)
```



```{r}
dim(x_test)#checking the dimensions of the variables
x_org_train <- x_train#storing the original data for CNN
x_org_test <- x_test #storing the original data for CNN
range(x_test)
x_train <-array_reshape(x_train,c(nrow(x_train),125*45))
#converting x data to 2 dimensions
x_test <-array_reshape(x_test,c(nrow(x_test),125*45))

dim(x_test)
#checking for new dimensions
```

```{r}
~~~~

~~~
x_train <- scale(x_train)#scaling data

x_test <- scale(x_test)

x_test <- as.matrix(x_test) #converting data to matrix to perform computaions such

# as PCA and correlation

x_train <- as.matrix(x_train)

xval <- sample(1:nrow(x_train),600)

#separating validation set from original set

xntrain <- setdiff(1:nrow(x_train),xval)

x_val <- x_train[xval,]

x_train <- x_train[xntrain,]


```



```{r}

corr <- cor(x_train)

#performing correlation, to remove similar variables

# this is done to reduce the dimensions and decrease the

# number of parameters of the model, which might lead to a higher accuracy

corr[upper.tri(corr)] <- 0

#making diagnal as 0, so I don't end up removing
~~~

```
# all the variables and upper triangular part as 0

diag(corr)<- 0

```
```

```
```{r}

x_test <- x_test[,!apply(corr,2,function(x) any(x > .70))]

x_train <- x_train[,!apply(corr,2,function(x) any(x > .70))]

x_val <- x_val[,!apply(corr,2,function(x) any(x > .70))]

#removing highly related data with correlation of more than 0.7

dim(x_train)

```
```

```
```{r}

#computing pca to reduce dimensions, to improve algorithm accuracy

pca <- prcomp(x_train)

prop <- cumsum(pca$sdev^2)/sum(pca$sdev^2)# compute cumulative proportion of variance

Q <-length( prop[prop<0.95] ) #maintaining atleast 95% of information

Q

#checking for Q, if large then we need to further decrease the number of dimensions
```

~~~
```



```{r}

prop <- cumsum(pca$sdev^2)/sum(pca$sdev^2)# compute cumulative proportion of variance

Q <-length( prop[prop<0.90] ) #maintaining atleast 90% of information

Q

# only a handful is retained

#more than 90 % data can be explained using  dimensions, so


```
~~~

Using 656 dimensions and retaining 90% information

~~~
```{r}

#using pca to to finally convert the main data to reduced number of dimensions

x1_train <- predict(pca,x_train)[,1:Q]

x1_test <- predict(pca,x_test)[,1:Q]

x1_val <- predict(pca,x_val)[,1:Q]

```
~~~

```{r}

#converting data to factor, followed by one-hot encoding for analysis

#separating y into training testing and validation

y1_train <- as.numeric(factor(y_train))

y1_test <- as.numeric(factor(y_test))

yc_train <- to_categorical(y1_train)

yc_test <- to_categorical(y1_test)

yc_train <- yc_train

yc_test <- yc_test

yf_train <- as.factor(y_train)

yf_test <- as.factor(y_test)


yc_test <- yc_test[,-1]

yc_train <- yc_train[,-1]

yc_val <- yc_train[xval,]

yc_train <- yc_train[xntrain,]

V <- ncol(x1_train)
```

```{r}

#trying out a basic neural network with 4 layers to check out how neural network performs

model2 <-  keras_model_sequential() %>%

layer_dense(units = 128, activation = "relu", input_shape = V) %>%# first hidden

layer_dense(units = 128, activation = "relu") %>%     #second layer

layer_dense(units = 64, activation = "relu") %>%      #third layer

layer_dense(units = 64, activation = "relu") %>%    #fourth layer

layer_dense(units = 19, activation = "softmax") %>%  #outputlayer

compile(

loss = "categorical_crossentropy", metrics = "accuracy",

optimizer = optimizer_sgd(),

)

# count parameters

count_params(model2)


fit <- model2 %>% fit(

x = x1_train, y = yc_train,

validation_data = list(x1_val, yc_val),

epochs = 100,

verbose = 0,


)
```

~~~~
# store accuracy on test set for each run
score <- model2 %>% evaluate(
  x1_test, yc_test,
  verbose = 0
)
score


```



```{r}
# to add a smooth line to points
smooth_line <- function(y) {
x <- 1:length(y)
out <- predict( loess(y ~ x) )
return(out)
}
# some colors will be used later
cols <- c("black", "dodgerblue3", "gray50", "deepskyblue2")
# check performance ---> error
~~~~

~~~
out <- 1 - cbind(fit$metrics$accuracy,fit$metrics$val_accuracy)

matplot(out, pch = 19, ylab = "Error", xlab = "Epochs",

col = adjustcolor(cols[1:2], 0.3),log = "y")

# on log scale to visualize better differences



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")


```


```{r}

#trying the same model as above, but with regularization to prevent overfitting

model_reg <- keras_model_sequential() %>%

layer_dense(units = 128, activation = "relu", input_shape = V,

kernel_regularizer = regularizer_l2(l = 0.009)) %>%

layer_dense(units = 128, activation = "relu",

kernel_regularizer = regularizer_l2(l = 0.009)) %>%

layer_dense(units = 64, activation = "relu",

kernel_regularizer = regularizer_l2(l = 0.009)) %>%

layer_dense(units = 32, 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_reg <- model_reg %>% fit(

x = x1_train, y = yc_train,

validation_data = list(x1_val, yc_val),

epochs = 100,

verbose = 0

)


# store accuracy on test set for each run

score <- model_reg %>% evaluate(

  x1_test, yc_test,

  verbose = 0

)

score
```

~~~
```

```{r}
out <- 1 - cbind(fit$metrics$accuracy,
fit$metrics$val_accuracy,
fit_reg$metrics$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")

```
~~~

~~~
```{r}

# get all weights

w_all <- get_weights(model2)

w_all_reg <- get_weights(model_reg)

# weights of first hidden layer

# one input --> 64 units

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)
~~~

~~~
```

```{r}
#CNN data preprocessing
# we need to convert this data to 3 dimensions, so that we can use convolution neural network
dim(x_org_train)
dim(x_org_test)
xcnn_train <-array_reshape(x_org_train,c(nrow(x_org_train),125,45,1))
xcnn_test <-array_reshape(x_org_test,c(nrow(x_org_test),125,45,1))
xcnn_val <- xcnn_train[xval,1:125,1:45,1]

xcnn_train <- xcnn_train[xntrain,1:125,1:45,1]
xcnn_train <- array_reshape(xcnn_train,c(nrow(xcnn_train),125,45,1))
xcnn_val <- array_reshape(xcnn_val,c(nrow(xcnn_val),125,45,1))

```

```{r}
#cnn model
model <- keras_model_sequential() %>%
~~~

```
#

# convolutional layers


layer_conv_2d(filters = 32, kernel_size = c(2,2), activation = "relu",input_shape = c(125,45,1)) %>%


layer_max_pooling_2d(pool_size = c(2,2)) %>%


layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu") %>%


layer_max_pooling_2d(pool_size = c(2,2)) %>%


layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu") %>%


layer_max_pooling_2d(pool_size = c(2,2)) %>%

#

# fully connected layers

layer_flatten() %>%

layer_dense(units = 64, activation = "relu", kernel_regularizer = regularizer_l2(0.1)) %>%

layer_dropout(0.1) %>%

layer_dense(units = 19, activation = "softmax") %>%

#

# compile
```

```
compile(

loss = "categorical_crossentropy",

metrics = "accuracy",

optimizer = optimizer_adam()

)

# model training

# NOTE : this will require some time

fit <- model %>% fit(

x = xcnn_train, y = yc_train,

validation_data = list(xcnn_val, yc_val),

epochs = 50,

batch_size = 40, # roughly 0.5% of training observations

verbose = 0

)


score <- model %>% evaluate(

  xcnn_test, yc_test,

  verbose = 0

)

score
```

~~~
```

```{r, eval=FALSE}
# to add a smooth line to points
smooth_line <- function(y) {
x <- 1:length(y)
out <- predict( loess(y ~ x) )
return(out)
}
# check performance
cols <- c("black", "dodgerblue3")
out <- cbind(fit$metrics$accuracy,
fit$metrics$val_accuracy)
matplot(out, pch = 19, ylab = "Accuracy", 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("bottomright", legend = c("Training", "Test"),
fill = cols[1:2], bty = "n")


```
~~~

```{r}

model_reg <- keras_model_sequential() %>%

#

# convolutional layers



layer_conv_2d(filters = 32, kernel_size = c(2,2), activation = "relu",input_shape = c(125,45,1),kernel_regularizer = regularizer_l2(l = 0.009)) %>%



layer_max_pooling_2d(pool_size = c(2,2)) %>%



layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu",kernel_regularizer = regularizer_l2(l = 0.009)) %>%



layer_max_pooling_2d(pool_size = c(2,2)) %>%



layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu",kernel_regularizer = regularizer_l2(l = 0.009)) %>%



layer_max_pooling_2d(pool_size = c(2,2)) %>%

#

# fully connected layers

layer_flatten() %>%

layer_dense(units = 64, activation = "relu", kernel_regularizer = regularizer_l2(0.1)) %>%
layer_dropout(0.1) %>%

layer_dense(units = 19, activation = "softmax") %>%

#

# compile

compile(

loss = "categorical_crossentropy",

metrics = "accuracy",

optimizer = optimizer_adam()

)

# model training

# NOTE : this will require some time

fit <- model %>% fit(

x = xcnn_train, y = yc_train,

validation_data = list(xcnn_val, yc_val),

epochs = 50,

batch_size = 40, # roughly 0.5% of training observations

verbose = 0

)


score <- model %>% evaluate(

  xcnn_test, yc_test,

  verbose = 0
```

~~~
)

score


```

```{r}

out <- 1 - cbind(fit$metrics$accuracy,

fit$metrics$val_accuracy,

fit_reg$metrics$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")

```




```{r}

#setting a grid of values for the flags/hyperparameters of interest:
~~~

~~~
hdlayer1 <- c(128,64,32)

dropout1 <- c(0,0.1,0.3)

hdlayer2 <- c(64,16)

dropout2 <- c(0,0.2)

hdlayer3 <- c(64,32,16)

dropout3 <- c(0,0.1)

# total combinations 3 x 3 x 2 x 2 x 3 x 2 = 216


```




```{r}

# run --------------------------------------------------------------

runs <- tuning_run("mlai_1.R", #creating runs to simulate output

                    runs_dir = "nn",

                    flags = list(

                   hdlayer_1 = hdlayer1,

                   dropout_1 = dropout1,

                   hdlayer_2 = hdlayer2,

                   dropout_2 = dropout2,

                   hdlayer_3 = hdlayer3,

                   dropout_3 = dropout3
~~~

~~~~
                    ),
                  sample = 0.1)
#sampling 21 models

```

```{r}
#Determing the optimal configuration for the data
#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)

}
```

```{r}

# extract results

out <- read_metrics("nn")

# 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)

```

```{r}

res1<- ls_runs(metric_val_accuracy > 0.8, runs_dir = "nn", order = metric_val_accuracy)

res1

```

```{r}

res1 <- res1[,c(2,4,8:13)]

res1[1:10,]

```

```{r}
```

~~~
# run --------------------------------------------------------------

dropout_set <- c(0, 0.3, 0.4, 0.5)

lambda_set <- c(0, exp( seq(-6, -4, length = 9) ))

lr_set <- c(0.001, 0.002, 0.005, 0.01)

bs_set <- c(0.005, 0.01, 0.02, 0.03)*nrow(x1_train)


runs <- tuning_run("mlai_2.R",

          runs_dir = "nn2",

          flags = list(

            dropout = dropout_set,

            lambda = lambda_set,

            lr = lr_set,

            bs = bs_set

          ),

          sample = 0.02)



```



```{r}

# extract results

out <- read_metrics("nn2")
~~~

~~~
# 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)

```




```{r}

res2<- ls_runs(metric_val_accuracy > 0.8, runs_dir = "nn2", order = metric_val_accuracy)

res2

```



```{r}

res2 <- res2[,c(2,4,8:13)]

res2[1:10,]

```
~~~