Machine Learning and AI - Daily and sports activities sensor data

Premysl Velek, 16213669

Abstract: This report present a deep neural network to classify daily and sports activities from motion sensor measurements. A four stage hyperparameter tuning process was implemented to select the optimal data pre-processing method, network architecture and regularization methods. The final model selected has two hidden layers with 384 units (256-128), early stopping, weight decay ($\lambda = 0.0001$) and dropout ($rate = 0.2$) regularization, low learning rate (0.0005) and batch size of 76 (1 % of the training sample). It achieved accuracy of 0.9855 on test data. Following the tuning process, the impact of the different hyperparameters on the model performance is discussed.

Introduction

This report presents the implementation of a deep neural network model to classify daily and sports activities from motion sensor measurements. To main goal of the following analysis is to train and test a deep neural network classification model, present the model configuration and discuss how it affects the performance of the model.

The resulting model can have various uses, predominantly as part of mobile apps and wearable devices that follow and track physical activities of their users (eg. fitness trackers, life-style apps or personal analytics apps). An important use of the model is within the rapidly developing field of health data analytics. The model can give medical doctors detailed information about their patients' physical activities which can then be coupled with their clinical data and used to better understand how peoples' life-style affects their health.[1]

Data

The data used to train and test the model are motion sensor measurements of 19 daily and sports activities, each performed by 8 subjects (4 female, 4 male, between the age of 20 and 30) in their own style for 5 minutes. Each participant in the experiment was equipped with five units - each with 9 individual sensors - which recorded the movement on the torso, arms, and legs.

The 19 daily activities performed are:

- Sitting
- Standing
- Lying on back
- Lying on right side
- Ascending stairs
- Descending stairs
- Standing in an elevator still
- Moving around in an elevator
- Walking in a parking lot
- Walking on a flat treadmill with a speed of 4 km/h
- Walking on a 15 deg inclined treadmill with a speed of 4 km/h

- Running on a treadmill with a speed of 8 km/h

[1]For overview of the topic see: Loncar-Turukalo T, Zdravevski E, Machado da Silva J et al, Literature on Wearable Technology for Connected Health: Scoping Review of Research Trends, Advances, and Barriers, J Med Internet Res. 2019 Sep 5;21(9):e14017. doi: 10.2196/14017.

- Exercising on a stepper
- Exercising on a cross trainer
- Cycling on an exercise bike in horizontal position
- Cycling on an exercise bike in vertical position
- Rowing
- Jumping
- Playing basketball

The dataset (sensor data) contains the sensor measurements divided into 5-second signal segments. There is a total of 480 signal segments (60 signal segments for each of the 8 participants) for each activity. These 480 segments are classified into 19 classes corresponding to the daily and physical activities listed above (total of 9120 segments). Each signal segment is represented by a 125 $\times$ 45 matrix, where rows contain the 125 samples of data acquired from one of the sensors of one of the units over a period of 5 seconds, and columns represent the 45 individual sensors (grouped into 5 units).

The dataset was first presented by Altun, Barshan and Tunçel in 2010[2] and is available online in the UCI Machine Learning Repository[3].

The dataset is extremely wide, with 5625 (125 x 45) separate variables for each observation. To visualize the data and explore the (approximate) relations between the different classes, a principal component analysis was performed on part of the data (1520 observations). Figure 1 shows the 2d projection of the first two principal components, amounting to over 32 % of the variance.

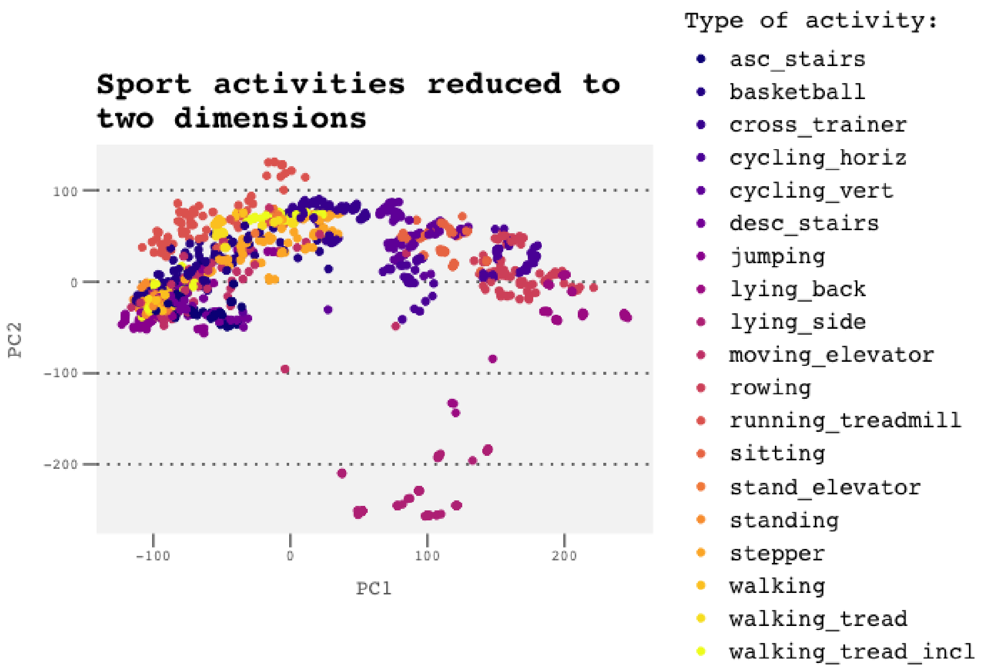

Figure 1: Visualisation of part of the sensor data: first two principal components

[2]K. Altun, B. Barshan, and O. Tunçel, Comparative study on classifying human activities with miniature inertial and magnetic sensors, Pattern Recognition, 43(10):3605-3620, October 2010.
[3]https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities

To train, validate and test the model, the sensor data was divided into train, validation and test subsets. The train dataset included 7600 observations (~ 83.3 %), the test and validation dataset contained each 760 observations (~ 8.3 %). The classes were distributed equally in the training set (each class is represented by 400 observations). The validation and test datasets combined contained 80 observations for each class; a random split was used to generate the validation and test datasets which produced a roughly equal number of classes in each set: the mean number of observations in the test and validation data for each class was 40, the standard deviation was 4.73.

Method

The following hyperparameters and configurations were considered during the training and validation step:

- Data preprocessing
- Model architecture: number of layers and number of nodes in each layer
- Dropout rate
- Weight decay rate
- Batch size
- Learning rate
- Patience rate for early stopping

The Adam optimization algorithm and the ReLU activation fictions were chosen for all models considered. The Adam algorithm is based on adaptive estimates of lower-order moments (Adaptive moment estimation) and has been shown to perform better than other stochastic optimization methods.[4] ReLU (Rectified Linear Unit) is the most successful and effectively the default activation function in deep learning models across domains and research communities.[5]

To identify the optimal model configuration, several models with different parameters were considered in four successive stages: (1) Pre-rocessing, (2)architecture, (3) Elimination tuning and (4) Final tuning.

Each model was trained on the training data and validated using the validation set. All model were trained over a maximum of 100 epochs, the performance on both train and validation set was measured after each epoch, using the accuracy metric (proportion of correctly classified classes to all classes). In Stages 3 and 4, each model considered was also evaluated using the test data.

The final performance on the validation and test data, and the shape of the learning curve were used to select the optimal hyperparamenters setting. The accuracy performance of the final selected model on test data was reported as the main performance metric.

The model was implemented using the R interface of the Keras API[6].

Stage 1: Data preprocessing

To select the optimal method of data preprocessing, three methods were tested: (1) Normalization, (2) Standardization, and (3) Identity (the original raw measurements).

The data obtained by the three preprocessing method were fitted to the same simple neural network model, with 2 layers and 512-128 nodes, ReLU activation function, softmax output activation function and Adam optimizer with the learning rate of 0.001. Figure 2 shows the training and validation learning curves for the three processing methods.

[4]Kingma, DP, Ba J, Adam: A Method for Stochastic Optimization, published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015, arXiv e-prints December 2015, https://arxiv.org/abs/1412.6980

[5]Ramachandran P, Zoph B, Le QV, Searching for Activation Functions, arXiv e-prints October 2017, https://arxiv.org/abs/1710.05941v2

[6]https://keras.rstudio.com/index.html

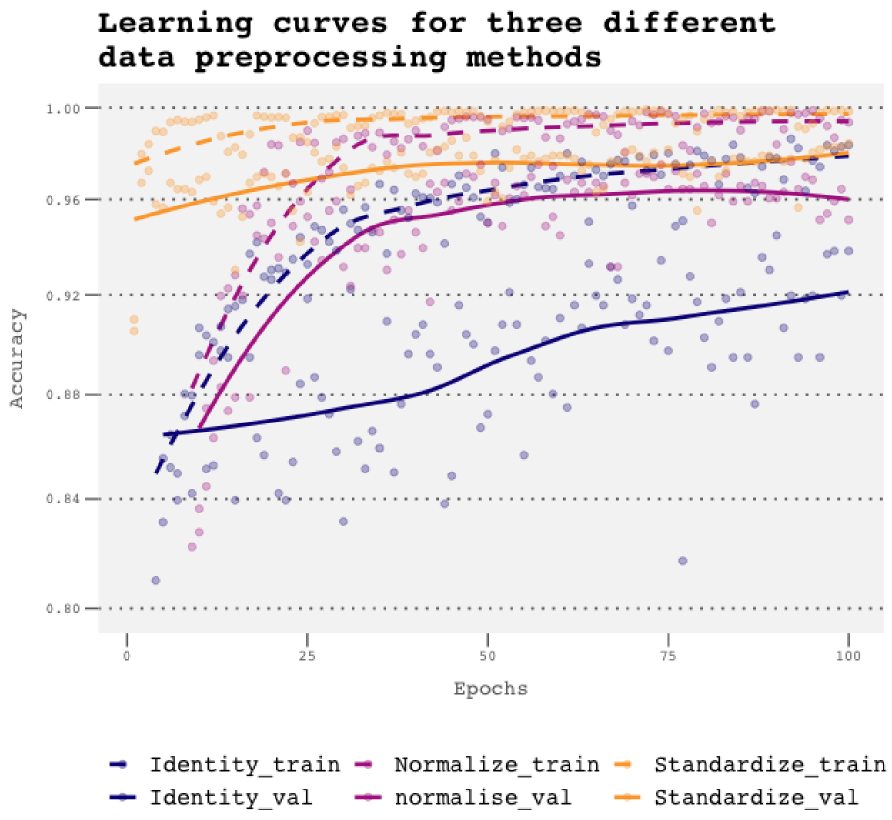

Figure 2: Training and validation performance for models fitted to normalised, standardised and raw data

Based on the result of the training and validation data, the standardization method was selected for the final model. The model with standardization data had the best training and validation performance and relatively steady learning curve as compared to the two other methods. It also better generalized on validation data as compared to the normalized dataset which had similar training accuracy but lower validation accuracy.

Stage 2: Architecture

To identify the optimal number of hidden layers, three different models with 2, 4 and 6 layers were trained and validated, using a simple model with ReLU activation function, softmax output activation function and Adam optimizer. The number of layers in the models were as follows:

- 2-layer model: 512-256
- 4-layer model: 512-256-128-64
- 6-layer model: 512-256-192-128-64-32

Figure 3 shows the training and validation learning curves for the three models. It's clear that the number of layers doesn't have a significant impact on the performance of the model, and more layers don't improve the performance of the model - the final accuracy values for all three models are within a narrow range. Even though the performance of the 6-layer model was more stable throughout the training epoch, this is not a sufficient reason to decide for more complex model, as regularization was addressed only in the stages 3 and 4.

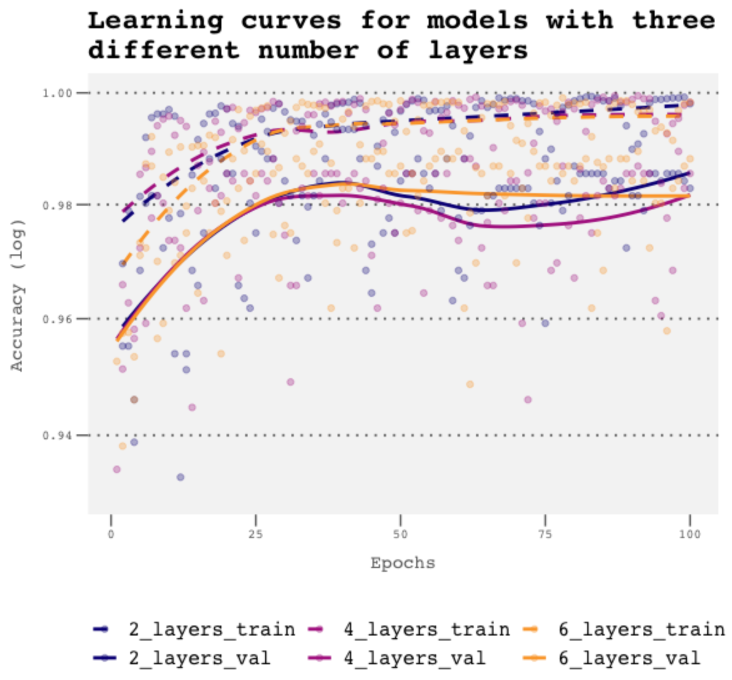

Figure 3: Training and validation performance for models with different number of layers

As simpler models should be preferred, a 2-layer model was selected at this stage.

Stage 3: Elimination tuning

Following the selection of data preprocessing method and the number of layers, all remaining hyperparameters were selected using a random grid search.[7] Grid search systematically searches through the configuration space and can fit model with every possible configurations, given the number of parameters we want to tune and the range of possible values.

In Stage 3 a wider set of values was considered with the aim to narrow down the range of possible values. Rather than identify the best hyperparameters for the model, the goal was to identify values or range of values for that don't produce good results. Those values were subsequently eliminated from the grid search implemented in Stage 4.

In the Elimination tuning, the following values were considered:

dropout	0, 0.2, 0.4
nodes_1	512, 256
nodes_2	256, 128, 64
lambda	0, 0.0001, 0.0015, 0.018
batch_size	22, 76, 152
learning_rate	0.001, 0.005, 0.01
patience	10, 20

There is 1296 possible combinations of the hyperparameters. The random grid search selected randomly 38 (3 %) of the possible combinations, and recorded the training, validation and test metrics for each of them.

[7]The tfruns package was used to manage the training runs during the tuning process, https://cran.r-project.org/package=tfruns

Figure 4 shows the validation learning curves of those 38 models in a series of plots to visualize how the different hyperparameters impact the validation results.[8]

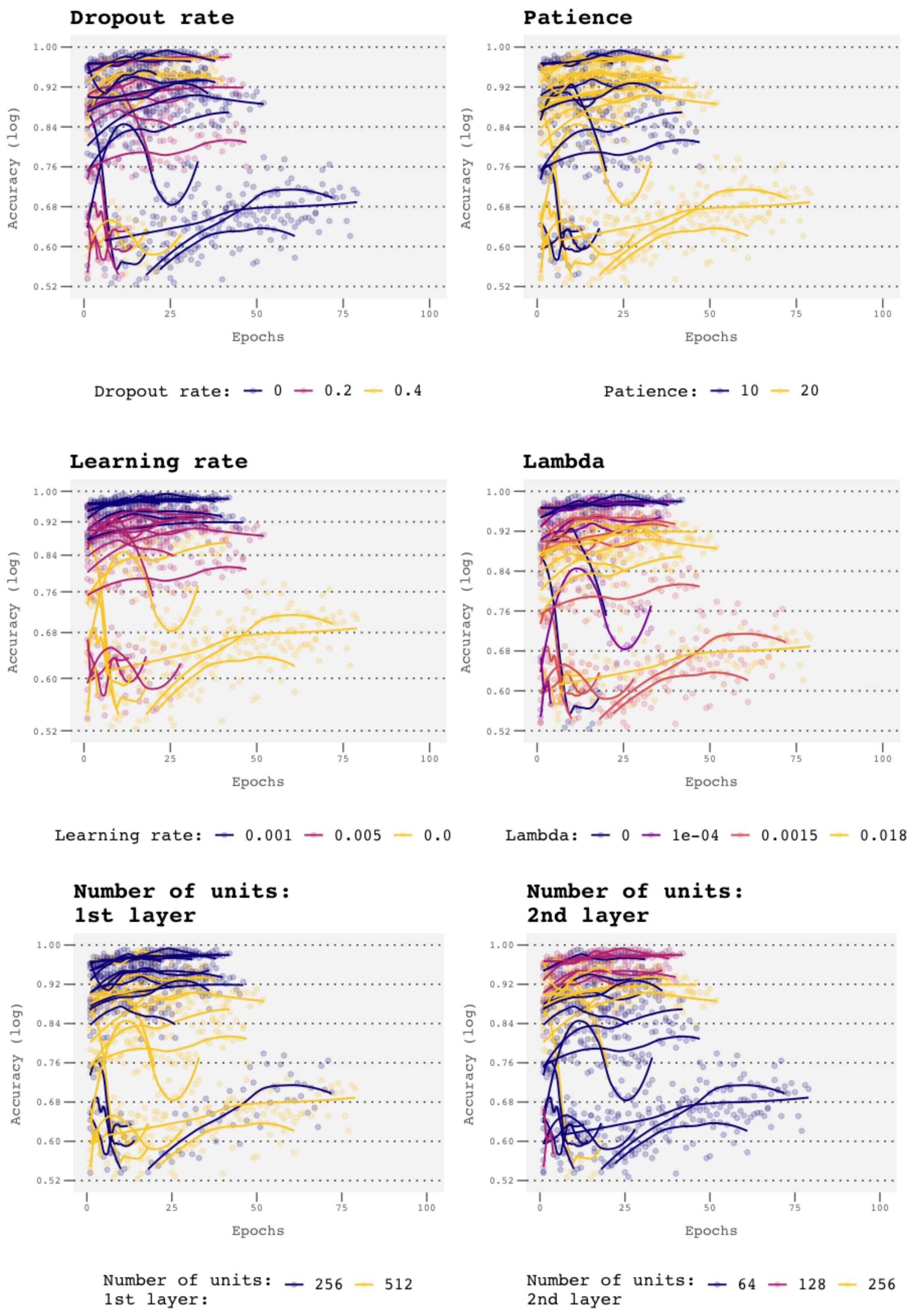

Figure 4: Figure 4: Validation performance for 38 models with different hyperparameters

[8]The effect of batch size is not included

The 10 worst performing models are listed in the table below:

	dropout	nodes_1	nodes_2	lambda	batch	lr	patience
23	0.4	512	256	0.0001	22	0.010	20
24	0.2	512	64	0.0015	22	0.010	20
16	0.2	512	128	0.0001	76	0.010	20
37	0.0	256	128	0.0001	22	0.010	20
4	0.4	256	128	0.0183	22	0.010	10
13	0.0	256	64	0.0001	22	0.010	20
3	0.2	512	128	0.0001	22	0.005	20
35	0.0	512	256	0.0000	76	0.010	20
33	0.2	256	64	0.0015	76	0.010	20
31	0.2	256	64	0.0015	76	0.010	10

Based on the results of the first grid search, the following hyperparameters negatively impacted the model performance and were eliminated from the tuning grid in the Stage 4.

1. Learning rate of 0.01 and 0.005
2. Lambda values of 0.0183 and 0.0015
3. The number of units 512 in the first hidden layer
4. The number of units 64 and 256 in the second hidden layer

As the results of the Stage 3 showed that low learning rates were much better choice for the model, an extra value of 0.0005 for learning rate was added to the grid.

On the contrary, there was no clear evidence that the different values for the dropout rate and for batch size significantly affect the model: the models with different dropout rate and batch size scored both low and high. While batch sizes of 22 and 76 produced the worst performing models (see the table above), they were also included in one of the best models. All values considered for dropout rate were equally represented among best and worst performing models. This could be because those two hyperparameters in reality don't affect significantly the model or because of the limited scope of the grid search (3 % of the possible combinations)[9]

As for the patience values, the learning curves of the high scoring models didn't show significant signs of declining performance over the epochs. This means that that we could keep the patience values higher without worrying about declining performance. In the next stage, only the patience value of 20 was considered.

All values of dropout rate and batch size were kept in the grid search in the next phase.

Stage 4: Final tuning

Following the results of the elimination tuning, the range of possible configuration settings was reduced from 1296 to 48. Using similar approach and similar model as in Stage 3, the following values were considered:

dropout	0, 0.2, 0.4
lambda	0, 1e-04
batch_size	22, 76, 152
learning_rate	0.0005, 0.001

The number of nodes and the patience value were fixed for all models.

[9]There were only eight models with dropout rate 0.4, five of them coupled with learning rate of 0.01 and 0.005, which we excluded already from the grid. So there were not enough models trained that would conclusively show whether or not dropout rate of 0.4 positively affect the performance.

At this stage a full grid search was performed, testing the complete range of possible configurations.

The hyperparameters for the ten best performing models - based on accuracy on the test data - are listed in the table below:

	test acc	val acc	dropout	lambda	batch	lr	epochs
27	0.9868	0.9921	0.0	1e-04	76	1e-03	55
30	0.9868	0.9855	0.0	0e+00	76	1e-03	34
8	0.9855	0.9868	0.2	1e-04	76	5e-04	78
6	0.9842	0.9803	0.0	0e+00	152	5e-04	37
1	0.9829	0.9737	0.4	1e-04	152	5e-04	50
11	0.9829	0.9842	0.2	0e+00	76	5e-04	48
13	0.9829	0.9816	0.4	1e-04	22	5e-04	99
24	0.9829	0.9868	0.0	0e+00	152	1e-03	35
32	0.9829	0.9750	0.2	1e-04	22	1e-03	51
16	0.9816	0.9763	0.4	0e+00	22	5e-04	73

We see that these best performing models have the accuracy on the test data over 0.98. Figure 5 show the learning curves for validation accuracy (left) and validation loss (right) for all models with the 10 best performing models highlighted:

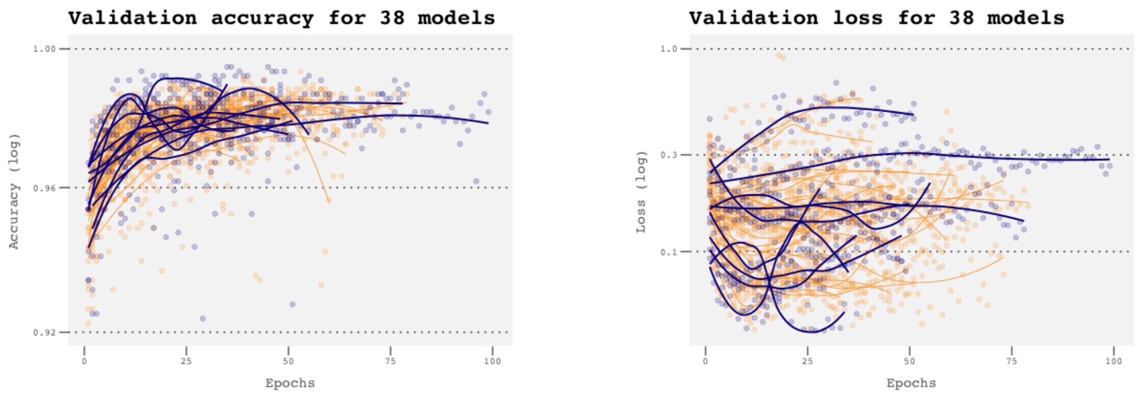

Figure 5: Validation accuracy and loss for 38 models with different hyperparameters

The learning curves suggest that many of the best performing models are over-fitted. Although their accuracy is high, the validation loss are either increasing or unstable. The final model was selected based on the test accuracy, test loss and a visual inspection of the learning curves of the best models.

Results and discussion

The final model selected as a result of four-stage training process has the following configuration and hyper-parameters:

- Two hidden layers with 256-128 nodes
- Activation function: ReLU
- Output activation function: Softmax
- Optimization method: Adam
- Weight decay regularization rate (lambda): 0.0001
- Dropout regularization rate: 0.2

- Early stopping regularization (patience): 20
- Batch size: 76
- Learning rate: 0.0005

The accuracy on the test data for this model is 0.9855 (the third best accuracy score). Overall, the hyper-parameter tuning process showed that there are several possible configurations for the model. Of the 38 models considered in the final stage, 15 reached test accuracy of over 0.98.

Figure 6 shows the accuracy training and validation curve for the final model:

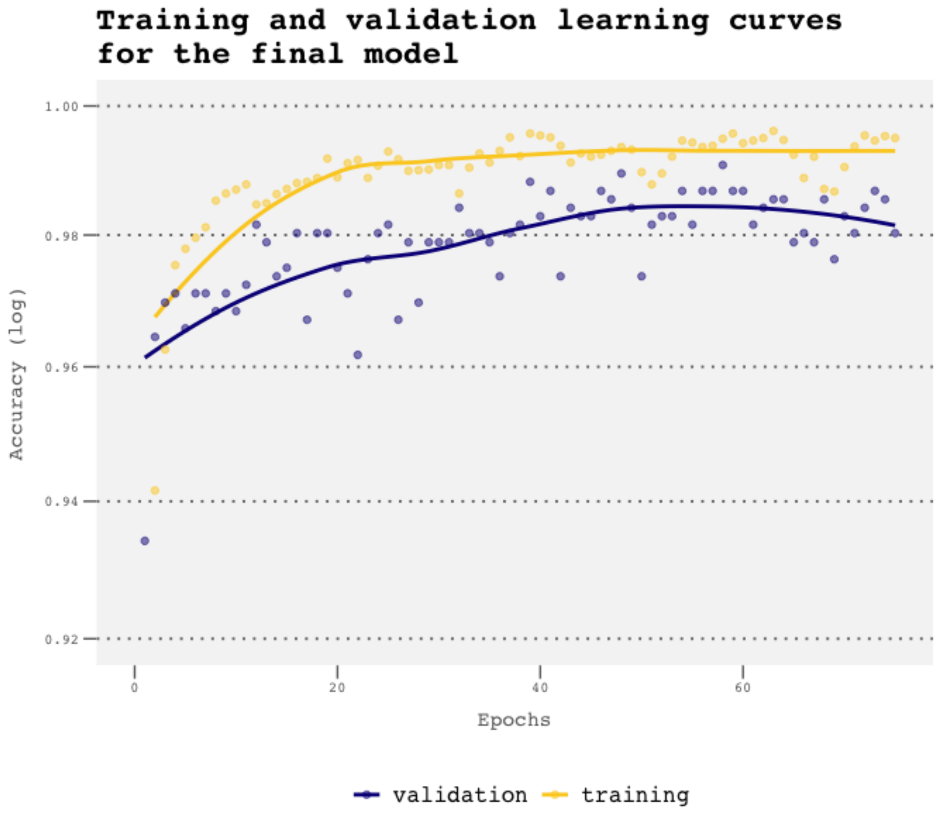

Figure 6: Training and validation accuracy learning curve for the final models

The learning rate seems to have the biggest impact on the model performance, whereby models with learning rate of 0.001 or lower clearly outperform models with higher learning rates. Similarly, higher values of lambda for weight decay regularization produced sub-optimal solutions. The best models have the value of lambda either zero or close to zero (0.0001).

Dropout regularization method doesn't have strong influence on the model, as the results show that models with high test accuracy have their dropout rate ranging from 0 to 0.4. Similarly, batch size don't significantly impact the model performance as models with all three values (22, 76, 152) produce models good model (with accuracy score of over 0.98).

The result also show that most models converge to their optimal performance very early on in training process. After around 30 epochs the increase in performance is minimal. In most cases the early stopping regularization stopped the training process within the first 50 epochs. No model completed the maximum of 100 epochs, only one model reached over 90 epochs, additional four models (including the final selected model) reached more than 70 training epochs. The optimal number of epoch is likely around 60 epochs.

After this, most model become over-fitted with corresponding decrease in accuracy and increase in the loss function.

In terms of the model architecture, the results presented indicate that higher number of hidden layers and higher number of nodes don't increase the performance of the model.

Conclusion

This report presented an implementation of deep neural network to classify daily and sporting activities based on motion sensor measurement. The final test accuracy reached over 0.98 %. Considering the intended use of the model (healthy life-style or heath care), this accuracy is sufficient for effective prediction of daily physical activities.

References

- R Core Team (2013). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/.
- R interface of the Keras API, https://keras.rstudio.com/index.html
- tfruns R package: https://cran.r-project.org/package=tfruns
- gglot2 R package: Wickham H (2016). ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. ISBN 978-3-319-24277-4, https://ggplot2.tidyverse.org.
- viridis R package, https://CRAN.R-project.org/package=viridis
- Kingma, DP, Ba J, Adam: A Method for Stochastic Optimization, published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015, arXiv e-prints December 2015, https://arxiv.org/abs/1412.6980
- Ramachandran P, Zoph B, Le QV, Searching for Activation Functions, arXiv e-prints October 2017, https://arxiv.org/abs/1710.05941v2
- K. Altun, B. Barshan, and O. Tunçel, Comparative study on classifying human activities with miniature inertial and magnetic sensors, Pattern Recognition, 43(10):3605-3620, October 2010, https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities
- Loncar-Turukalo T, Zdravevski E, Machado da Silva J et al, Literature on Wearable Technology for Connected Health: Scoping Review of Research Trends, Advances, and Barriers, J Med Internet Res. 2019 Sep 5;21(9):e14017. doi: 10.2196/14017.

Appendix 1 - R script

```r
# libraries
library(keras)
library(ggplot2)
library(reshape2)
library(viridis)
library(dplyr)
library(tfruns)
library(gridExtra)
library(jsonlite)
```

```r
# load helper functions
source('helpers.R')

# load data
load('data_activity_recognition.RData')

# check the dimensions
dim(x_test)
dim(x_train)

# reshape the matrix into vectors - each signal segment is represented by a
# vector of 125*45 measurements
x_train <- array_reshape(x_train, c(nrow(x_train), 125 * 45))
x_test <- array_reshape(x_test, c(nrow(x_test), 125 * 45))

##########################################################################################
### Visualisation of the classes ##########################################################
##########################################################################################

# reduce the dimension of the data with PCA
x_test_pca <- prcomp(x_test)

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

# proportion of variance explained by the first two and ten components
prop[2]

# plot the first two principal components (around 31 % of the variance)
ggplot(as.data.frame(x_test_pca$x[, c(1:2)]), aes(x = x_test_pca$x[, 1], y = x_test_pca$x[, 2])) +
  geom_point(aes(colour = y_test)) +
  theme_pv() +
  scale_colour_viridis_d(option = 'C', name = 'Type of activity:') +
  labs(title = 'Sport activities reduced to \ntwo dimensions', y = "PC2", x = "PC1") +
  theme(legend.position = 'right',
        legend.text = element_text(size = 12))

#####

######## Data preparation ######################################################

# see the range of the value
range(x_train)

# convert classes from character to numeric value
y_test <- as.factor(y_test)
y_test <- as.numeric(y_test)
y_test <- y_test - 1

y_train <- as.factor(y_train)
y_train <- as.numeric(y_train)
y_train <- y_train - 1
```

```r
# one-hot encoding
y_train <- to_categorical(y_train)
y_test <- to_categorical(y_test)

######## Validation-test split ######################################################

# split the test data in two halves: one for validation
# and the other for actual testing
set.seed(12)

val <- sample(1:nrow(x_test), floor(nrow(x_test) * 0.5))
test <- setdiff(1:nrow(x_test), val)

x_val <- x_test[val, ]
y_val <- y_test[val, ]

x_test <- x_test[test, ]
y_test <- y_test[test, ]

V <- ncol(x_train)
N <- nrow(x_train)

# check if the classes are distributed equaly in the test and val data
mean(colSums(y_val))
mean(colSums(y_test))

sd(colSums(y_val))
sd(colSums(y_test))

########################################################################################################
################ Selecting data preprocessing methods ##################################################
########################################################################################################

# normalisation
normalise <- function(x){
  (x - min(x)) / (max(x) - min(x))
}

x_train_n <- apply(x_train, 2, normalise)
x_val_n <- apply(x_val, 2, normalise)

# standardisation
x_train_s <- scale(x_train)
x_val_s <- scale(x_val)



# dnn to select the data preprocessing method
model <- keras_model_sequential() %>%
  layer_dense(units = 512, input_shape = V, activation = "relu", name = "layer_1") %>%
  layer_dense(units = 128, activation = "relu", name = "layer_2") %>%
  layer_dense(units = ncol(y_train), activation = "softmax", name = "layer_out") %>%
  compile(
    loss = "categorical_crossentropy",
    metrics = "accuracy",
    optimizer = optimizer_adam()
  )

# put the three data in a list
dprep_test <- list(norm = list(x_train_n, x_val_n),
                   identity = list(x_train, x_val),
                   stand = list(x_train_s, x_val_s))

# initialise list to store the learning curve data
fits <- list()


# run three models
for(i in 1:3){
  fit <- model %>% fit(
    x = dprep_test[[i]][1], y = y_train,
    validation_data = list(dprep_test[[i]][2], y_val),
    epochs = 100,
    verbose = 1)
  fits[[i]] <- fit
}


###### Plotting #############

coln <- c('Identity_train', 'Identity_val',
          'Normalize_train', 'normalise_val',
          'Standardize_train', 'Standardize_val')

# see the helpers.R file for details for data_pt
plot_learn_c <- data_pt(fits, coln = coln)

# colour palette
col <- rep(viridis(3, option = 'C', end = 0.8), each = 2)

# plot train and validation learning curves
ggplot(plot_learn_c, aes(x = id, y = value, group = variable,
                         colour = variable, linetype = variable)) +
  stat_smooth(data = plot_learn_c, method = 'loess', geom = 'line',
              se = FALSE, size = 1) + #, linetype = "dashed") +
  #stat_smooth(data = plot_learn_c_v, method = 'loess', geom = 'line',
              #se = FALSE, size = 1) +
  scale_y_log10(limits = c(0.8, 1), breaks = seq(0.80, 1, by = 0.04)) +
  theme_pv() +
  scale_colour_manual(values = col) +
  scale_linetype_manual(values = c(2, 1, 2, 1, 2, 1)) +
  geom_point(alpha = 0.3) +
  labs(title = 'Learning curves for three different \ndata preprocessing methods',
       y = "Accuracy", x = "Epochs") +
  theme(legend.position = "bottom",
```

```r
        legend.title = element_blank(),
        legend.text = element_text(size = 12))

# we go with standardising

#####################################################################
################ Selecting number of layers #########################
#####################################################################

# Model with 2 - 4 - 6 layers, each with decreasing number of units

# 3 dnn with different number of layers
model_2 <- keras_model_sequential() %>%
  layer_dense(units = 512, input_shape = V, activation = "relu", name = "layer_1") %>%
  layer_dense(units = 256, activation = "relu", name = "layer_2") %>%
  layer_dense(units = ncol(y_train), activation = "softmax", name = "layer_out") %>%

  compile(
    loss = "categorical_crossentropy",
    metrics = "accuracy",
    optimizer = optimizer_adam()
  )

model_4 <- keras_model_sequential() %>%
  layer_dense(units = 512, input_shape = V, activation = "relu", name = "layer_1") %>%
  layer_dense(units = 256, activation = "relu", name = "layer_2") %>%
  layer_dense(units = 128, activation = "relu", name = "layer_3") %>%
  layer_dense(units = 64, activation = "relu", name = "layer_4") %>%
  layer_dense(units = ncol(y_train), activation = "softmax", name = "layer_out") %>%

  compile(
    loss = "categorical_crossentropy",
    metrics = "accuracy",
    optimizer = optimizer_adam()
  )

model_6 <- keras_model_sequential() %>%
  layer_dense(units = 512, input_shape = V, activation = "relu", name = "layer_1") %>%
  layer_dense(units = 256, activation = "relu", name = "layer_2") %>%
  layer_dense(units = 192, activation = "relu", name = "layer_3") %>%
  layer_dense(units = 128, activation = "relu", name = "layer_4") %>%
  layer_dense(units = 64, activation = "relu", name = "layer_5") %>%
  layer_dense(units = 32, activation = "relu", name = "layer_6") %>%
  layer_dense(units = ncol(y_train), activation = "softmax", name = "layer_out") %>%

  compile(
    loss = "categorical_crossentropy",
    metrics = "accuracy",
    optimizer = optimizer_adam()
  )


# initialise list to store the learning curve data
```

```r
fits_a <- list()

# run three models

# 2 layrers
fit <- model_2 %>% fit(
  x = x_train_s, y = y_train,
  validation_data = list(x_val_s, y_val),
  epochs = 100,
  verbose = 1)
fits_a[[1]] <- fit

# 4 layrers
fit <- model_4 %>% fit(
  x = x_train_s, y = y_train,
  validation_data = list(x_val_s, y_val),
  epochs = 100,
  verbose = 1)
fits_a[[2]] <- fit

# 4 layers
fit <- model_6 %>% fit(
  x = x_train_s, y = y_train,
  validation_data = list(x_val_s, y_val),
  epochs = 100,
  verbose = 1)
fits_a[[3]] <- fit


###### Plotting #############

coln_a <- c('2_layers_train', '2_layers_val',
          '4_layers_train', '4_layers_val',
          '6_layers_train', '6_layers_val')

# see the helpers.R file for details for data_pt
plot_learn_c_a <- data_pt(fits_a, coln = coln_a)

# colour palette
col <- rep(viridis(3, option = 'C', end = 0.8), each = 2)

# plot train and validation learning curves
ggplot(plot_learn_c_a, aes(x = id, y = value, group = variable,
                         colour = variable, linetype = variable)) +
  stat_smooth(data = plot_learn_c_a, method = 'loess', geom = 'line',
              se = FALSE, size = 1) +
  scale_y_log10(limits = c(0.93, 1), breaks = seq(0.92, 1, by = 0.02)) +
  theme_pv() +
  scale_colour_manual(values = col) +
  scale_linetype_manual(values = c(2, 1, 2, 1, 2, 1)) +
  geom_point(alpha = 0.3) +
  labs(title = 'Learning curves for models with three \ndifferent number of layers',
       y = "Accuracy (log)", x = "Epochs") +
  theme(legend.position = "bottom",
        legend.title = element_blank(),
        legend.text = element_text(size = 12))

# we go with 2 layers

######################################################################
################ Hyperparameter tuning - rough tuning ################
######################################################################

# run #########
dropout_set <- c(0, 0.2, 0.4)
units_1_set <- c(512, 256)
units_2_set <- c(256, 128, 64)
lambda_set <- c(0, exp( seq(-9, -4, length = 3) ))
bs_set <- floor(c(0.003, 0.01, 0.02) * N)
lr_set <- c(0.001, 0.005, 0.01)
patience_set <- c(10, 20)

runs <- tuning_run("model_conf.R",
                   runs_dir = "runs_3",
                   flags = list(
                     dropout = dropout_set,
                     units_1 = units_1_set,
                     units_2 = units_2_set,
                     lambda = lambda_set,
                     bs = bs_set,
                     lr = lr_set,
                     patience = patience_set),
                   sample = 0.03)


# get the worst models and their parameters
worst <- ls_runs(runs_dir = "runs_3", order = metric_val_accuracy)

# select only the relevant paramenters
worst <- s[, c(2, 4, 8:14, 18)]
worst[order(worst$eval_accuracy)[1:10], ]


########### extract results ############

run_3 <- read_metrics("runs_3")

# extract validation accuracy and plot learning curve
acc_3 <- as.data.frame(sapply(run_3, "[[", "val_accuracy"))

# extract the parameters values for each run
param_3 <- as.data.frame(sapply(run_3, "[[", "flags"))
param_3 <- apply(param_3, 2, unlist)

# add id variable for easy melting
acc_3$id <- c(1:100)
```

```r
# convert to long format for easy plotting
acc_3 <- melt(acc_3, id.var = 'id')

# extact column names for each hyperparameter
dropout <- as.factor(param_3['dropout', ])
lambda <- as.factor(param_3['lambda', ])
batch <- as.factor(param_3['bs', ])
lr <- as.factor(param_3['lr', ])
patience <- as.factor(param_3['patience', ])
units_1 <- as.factor(param_3['units_1', ])
units_2 <- as.factor(param_3['units_2', ])

# convert to long format
dropout <- rep(dropout, each = 100)
lambda <- rep(lambda, each = 100)
batch <- rep(batch, each = 100)
lr <- rep(lr, each = 100)
patience <- rep(patience, each = 100)
units_1 <- rep(units_1, each = 100)
units_2 <- rep(units_2, each = 100)

# add the hyperparameter values to the metrics data frame
acc_3$dropout <- dropout
acc_3$lambda <- lambda
acc_3$batch <- batch
acc_3$lr <- lr
acc_3$patience <- patience
acc_3$units_1 <- units_1
acc_3$units_2 <- units_2

# colour palette
col <- viridis(2, option = 'C', end = 0.8)



############  plotting ##################

# plot the learning curves colour coded by hyberparameter values considered


# dropout
dr <- ggplot(acc_3, aes(x = id, y = value, group = variable, colour = dropout)) +
  geom_point(data = acc_3, alpha = 0.2) +
  stat_smooth(method = 'loess', geom = 'line', se = FALSE, size = 0.7) +
  scale_colour_viridis_d(name = 'Dropout rate:', option = 'C', end = 0.9) +
  scale_y_continuous(limits = c(0.52, 1), breaks = seq(0.52, 1, by = 0.08)) +
  theme_pv() +
  labs(title = 'Dropout rate', y = "Accuracy (log)", x = "Epochs") +
  theme(legend.position = "bottom",
        legend.text = element_text(size = 12))


# batch size
```

```r
bs <- ggplot(acc_3, aes(x = id, y = value, group = variable, colour = batch)) +
  geom_point(data = acc_3, alpha = 0.2) +
  stat_smooth(method = 'loess', geom = 'line',
              se = FALSE, size = 0.7) +
  scale_colour_viridis_d(name = 'Batch size:', option = 'C', end = 0.9) +
  scale_y_continuous(limits = c(0.52, 1), breaks = seq(0.52, 1, by = 0.08)) +
  theme_pv() +
  labs(title = 'Batch size', y = "Accuracy (log)", x = "Epochs") +
  theme(legend.position = "bottom",
        legend.text = element_text(size = 12))


# learning rate
ler <- ggplot(acc_3, aes(x = id, y = value, group = variable, colour = lr)) +
  geom_point(data = acc_3, alpha = 0.2) +
  stat_smooth(method = 'loess', geom = 'line',
              se = FALSE, size = 0.7) +
  scale_colour_viridis_d(name = 'Learning rate:', option = 'C', end = 0.9) +
  scale_y_log10(limits = c(0.52, 1), breaks = seq(0.52, 1, by = 0.08)) +
  theme_pv() +
  labs(title = 'Learning rate', y = "Accuracy (log)", x = "Epochs") +
  theme(legend.position = "bottom",
        legend.text = element_text(size = 12))


# lambda
lam <- ggplot(acc_3, aes(x = id, y = value, group = variable, colour = lambda)) +
  geom_point(data = acc_3, alpha = 0.2) +
  stat_smooth(method = 'loess', geom = 'line', se = FALSE, size = 0.7) +
  scale_colour_viridis_d(name = 'Lambda:', option = 'C', end = 0.9) +
  scale_y_continuous(limits = c(0.52, 1), breaks = seq(0.52, 1, by = 0.08)) +
  theme_pv() +
  labs(title = 'Lambda', y = "Accuracy (log)", x = "Epochs") +
  theme(legend.position = "bottom",
        legend.text = element_text(size = 12))

# patience
pat <- ggplot(acc_3, aes(x = id, y = value, group = variable, colour = patience)) +
  geom_point(data = acc_3, alpha = 0.2) +
  stat_smooth(method = 'loess', geom = 'line', se = FALSE, size = 0.7) +
  scale_colour_viridis_d(name = 'Patience:', option = 'C', end = 0.9) +
  scale_y_continuous(limits = c(0.52, 1), breaks = seq(0.52, 1, by = 0.08)) +
  theme_pv() +
  labs(title = 'Patience', y = "Accuracy (log)", x = "Epochs") +
  theme(legend.position = "bottom",
        legend.text = element_text(size = 12))

# units first layer
u_1 <- ggplot(acc_3, aes(x = id, y = value, group = variable, colour = units_1)) +
  geom_point(data = acc_3, alpha = 0.2) +
  stat_smooth(method = 'loess', geom = 'line', se = FALSE, size = 0.7) +
  scale_colour_viridis_d(name = 'Number of units:\n1st layer:', option = 'C', end = 0.9) +
  scale_y_continuous(limits = c(0.52, 1), breaks = seq(0.52, 1, by = 0.08)) +
  theme_pv() +
  labs(title = 'Number of units:\n1st layer', y = "Accuracy (log)", x = "Epochs") +
  theme(legend.position = "bottom",
        legend.text = element_text(size = 12))

# units second layer
u_2 <- ggplot(acc_3, aes(x = id, y = value, group = variable, colour = units_2)) +
  geom_point(data = acc_3, alpha = 0.2) +
  stat_smooth(method = 'loess', geom = 'line',
              se = FALSE, size = 0.7) +
  scale_colour_viridis_d(name = 'Number of units:\n2nd layer', option = 'C', end = 0.9) +
  scale_y_continuous(limits = c(0.52, 1), breaks = seq(0.52, 1, by = 0.08)) +
  theme_pv() +
  labs(title = 'Number of units:\n2nd layer', y = "Accuracy (log)", x = "Epochs") +
  theme(legend.position = "bottom",
        legend.text = element_text(size = 12))
##########

# Plotting multiple plots
grid.arrange(dr, pat, ler, lam, u_1, u_2)
grid.arrange(u_1, u_2, widths = c(1, 1),
             layout_matrix = rbind(c(1, 2), c(3, NA)))


######################################################################
################ Hyperparameter tuning - fine tuning #################
######################################################################

dropout_set <- c(0, 0.2, 0.4)
lambda_set <- c(0, 1e-04)
bs_set <- floor(c(0.003, 0.01, 0.02) * N)
lr_set <- c(0.001, 0.0005)


runs_2 <- tuning_run("model_conf_2.R",
                   runs_dir = "runs_4",
                   flags = list(
                     dropout = dropout_set,
                     lambda = lambda_set,
                     bs = bs_set,
                     lr = lr_set))

# get the worst models and their parameters
best <- ls_runs(runs_dir = "runs_4", order = eval_accuracy)

# select only the relevant paramenters
best <- best[, c(2, 7:11, 15)]


# plot all learning curves

# extract validation accuracy and loss and save as data frame
run_4 <- read_metrics("runs_4")
```

```r
acc_4 <- as.data.frame(sapply(run_4, "[[", "val_accuracy"))
loss_4 <- as.data.frame(sapply(run_4, "[[", "val_loss"))

# extract evaluation metrics
eval <- as.data.frame(sapply(run_4, "[[", "evaluation"))
eval <- apply(eval, 2, unlist)
eval <- eval['accuracy', ]

eval_ord <- order(eval, decreasing = TRUE)

acc_4_best <- acc_4[, eval_ord[1:10]]
acc_4 <- acc_4[, - eval_ord[1:10]]

loss_4_best <- loss_4[, eval_ord[1:10]]
loss_4 <- loss_4[, - eval_ord[1:10]]

# add id variable for easy melting
acc_4$id <- c(1:100)
acc_4_best$id <- c(1:100)

loss_4$id <- c(1:100)
loss_4_best$id <- c(1:100)

# convert to long format for easy plotting
acc_4 <- melt(acc_4, id.var = 'id')
loss_4 <- melt(loss_4, id.var = 'id')
acc_4_best <- melt(acc_4_best, id.var = 'id')
loss_4_best <- melt(loss_4_best, id.var = 'id')


########### plotting ############

# colour palette
col <- viridis(2, option = 'C', end = 0.8)

# accuracy
accu <- ggplot(acc_4, aes(x = id, y = value, group = variable)) +
  geom_point(data = acc_4, alpha = 0.2, colour = col[2]) +
  stat_smooth(method = 'loess', geom = 'line', se = FALSE, size = 0.3,
              colour = col[2]) +
  geom_point(data = acc_4_best, alpha = 0.2, colour = col[1]) +
  stat_smooth(data = acc_4_best, method = 'loess', geom = 'line', se = FALSE,
              size = 0.7, colour = col[1]) +
  scale_y_log10(limits = c(0.92, 1), breaks = seq(0.92, 1, by = 0.04)) +
  theme_pv() +
  labs(title = 'Validation accuracy for 38 models',
       y = "Accuracy (log)", x = "Epochs") +
  theme(legend.position = "bottom",
        legend.text = element_text(size = 12))


# accuracy
loss <- ggplot(loss_4, aes(x = id, y = value, group = variable)) +
  geom_point(data = loss_4, alpha = 0.2, colour = col[2]) +
  stat_smooth(method = 'loess', geom = 'line', se = FALSE,
              size = 0.3, colour = col[2]) +
  geom_point(data = loss_4_best, alpha = 0.2, colour = col[1]) +
  stat_smooth(data = loss_4_best, method = 'loess', geom = 'line',
              se = FALSE, size = 0.7, colour = col[1]) +
  scale_y_log10(limits = c(0.04, 1)) +
  theme_pv() +
  labs(title = 'Validation loss for 38 models', y = "Loss (log)", x = "Epochs") +
  theme(legend.position = "bottom",
        legend.text = element_text(size = 12))

# Plotting multiple plots
grid.arrange(accu, loss, nrow = 1)


# plotting the validation and training accuracy for the final model

# extract the training and validation data
final_val <- as.data.frame(sapply(run_4, "[[", "val_accuracy"))
final_train <- as.data.frame(sapply(run_4, "[[", "accuracy"))

# select the best model
final_val <- as.data.frame(final_val[, 29])
final_train <- as.data.frame(final_train[, 29])

final <- cbind(final_val, final_train)

# add id variable for easy melting
final$id <- c(1:100)

# add a column name
names(final)[1] <- 'validation'
names(final)[2] <- 'training'

final <- melt(final, id.var = 'id')

# final plotting
ggplot(final, aes(x = id, y = value, group = variable, colour = variable)) +
  stat_smooth(method = 'loess', geom = 'line', se = FALSE, size = 1) +
  geom_point(alpha = 0.5) +
  scale_colour_viridis_d(option = 'C', end = 0.9) +
  scale_y_log10(limits = c(0.90, 1), breaks = seq(0.90, 1, by = 0.02)) +
  theme_pv() +
  xlim(0, 75) +
  labs(title = 'Training and validation learning curves \nfor the final model',
       y = "Accuracy (log)", x = "Epochs") +
  theme(legend.position = "bottom",
        legend.title = element_blank(),
        legend.text = element_text(size = 12))


##########
```

Appendix 2 - Model configurations files

```r
####### model_conf.R ##############

#======== Model and settings configuration
#

# model instantiation -----------------------------------------------
# set defaul flags
FLAGS <- flags(
  flag_numeric("dropout", 0),
  flag_numeric("units_1", 512),
  flag_numeric("units_2", 256),
  flag_numeric('lambda', 0),
  flag_numeric('bs', floor(0.003 * N)),
  flag_numeric('lr', 0.001),
  flag_numeric('patience', 10)
)

# model configuration
model <- keras_model_sequential() %>%
  layer_dense(units = FLAGS$units_1, input_shape = V,
              activation = "relu", name = "layer_1",
              kernel_regularizer = regularizer_l2(l = FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = FLAGS$units_2, activation = "relu", name = "layer_2",
              kernel_regularizer = regularizer_l2(l = FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = ncol(y_train), activation = "softmax",
              name = "layer_out") %>%
  compile(loss = "categorical_crossentropy", metrics = "accuracy",
          optimizer = optimizer_adam(lr = FLAGS$lr))

fit <- model %>% fit(
  x = x_train_s, y = y_train,
  validation_data = list(x_val_s, y_val),
  epochs = 100,
  batch_size = FLAGS$bs,
  verbose = 1,
  callbacks = callback_early_stopping(monitor = "val_accuracy",
                                      patience = FLAGS$patience)
)

# store accuracy on test set for each run
score <- model %>% evaluate(
  scale(x_test), y_test,
  verbose = 0
)

####### model_conf_2.R ##############

##======== Model and settings configuration
#
```

```r
# model instantiation -----------------------------------------------
# set defaul flags
FLAGS <- flags(
  flag_numeric("dropout", 0),
  flag_numeric('lambda', 0),
  flag_numeric('bs', floor(0.003 * N)),
  flag_numeric('lr', 0.001)
)

# model configuration
model <- keras_model_sequential() %>%
  layer_dense(units = 256, input_shape = V,
              activation = "relu", name = "layer_1",
              kernel_regularizer = regularizer_l2(l = FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = 128, activation = "relu", name = "layer_2",
              kernel_regularizer = regularizer_l2(l = FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = ncol(y_train), activation = "softmax", name = "layer_out") %>%

  compile(loss = "categorical_crossentropy", metrics = "accuracy",
          optimizer = optimizer_adam(lr = FLAGS$lr))

fit <- model %>% fit(
  x = x_train_s, y = y_train,
  validation_data = list(x_val_s, y_val),
  epochs = 100,
  batch_size = FLAGS$bs,
  verbose = 1,
  callbacks = callback_early_stopping(monitor = "val_accuracy", patience = 20)
)

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

Appendix 3 - helper functions

The main script require the `helper.R` file to be places in the working directory!

```r
####### Custom theme used for all plots in the project
library(ggthemes)
library(ggplot2)

# Define the basics
theme_pv <- function(base_size = 12,
                     base_family = 'mono',
                     base_rect_size = base_size / 170){
```

```
  theme(
    # Define the text in general
    text = element_text(family = base_family),

    # Define the plot title
    plot.title = element_text(color = 'black', face = "bold", hjust = 0, size = 17),

    # Define titles for the axes
    axis.title = element_text(
      colour = rgb(105, 105, 105, maxColorValue = 255),
      size = rel(1), margin = margin(t = 20, r = 20)),
    axis.title.x = element_text(margin = margin(t = 10, b = 10)),
    axis.title.y = element_text(margin = margin(r = 10)),

    # Define the style of the caption
    plot.caption = element_text(hjust = 1,
                                colour = rgb(150, 150, 150, maxColorValue = 255)),

    # Define the style of the axes text and ticks
    axis.text = element_text(
      color = rgb(105, 105, 105, maxColorValue = 255), size = rel(0.7)),
    axis.ticks = element_line(colour = rgb(105, 105, 105, maxColorValue = 255)),
    axis.ticks.length.y = unit(.25, "cm"),
    axis.ticks.length.x = unit(.25, "cm"),

    # Style the grid lines
    panel.grid.major.y = element_line(
      rgb(105, 105, 105, maxColorValue = 255), linetype = "dotted", size = rel(1.3)),
    panel.ontop = FALSE,
    panel.grid.minor.x = element_blank(),
    panel.grid.minor.y = element_blank(),
    panel.grid.major.x = element_blank(),

    # Specify the background of the plot panel
    panel.background = element_rect(fill = rgb(245, 245, 245, maxColorValue = 255)),

    # Define the style of the legend
    legend.key = element_rect(fill = "white", colour = NA),
    legend.title = element_text(size = rel(1.2)),

    # Define the aspect ratio of the panel
    aspect.ratio = 0.7
  )
}

######### extract metrics from tfruns output
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)
```

```r
  for ( i in n:1 ) {
    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)
}


# function to plot the learning curves for preprocessing and architecture.
# It takes as input the list of  validation and training data (fits) and names
# for the legend (coln) and returns a data frame which can be  passed to ggplot
data_pt <- function(fits, coln) {
  n <- length(fits)

  # create empty dataframe to store the performance metrics
  learn <- data.frame(matrix(ncol = length(coln), nrow = 100))

  for (i in 1:n) {
    # bind the accuracy metrics together
    learn[, c((i * 2) - 1, i * 2)] <- cbind(fits[[i]]$metrics$accuracy,
                                            fits[[i]]$metrics$val_accuracy)
  }

  # add the column names
  colnames(learn) <- coln

  #id variable for position in matrix
  learn$id <- 1:nrow(learn)

  #reshape to long format
  plot_learn <- melt(learn, id.var = 'id')

  plot_learn
}
```