STAT40970 – Machine Learning & A.I (Online) - Assignment 2

Arjun Vijay Anup

Table of Contents

- Type and size of the images in the data.
  - RGB type, Height = 256, Width = 256
- Depth of the convolutional layers.
  - 1st layer: 64
  - 2nd layer: 128
  - 3rd layer: 128
- Size of the filters
  - 1st layer: 6x6
  - 2nd layer: 4x4
  - 3rd layer: 2x2
- Number of batches processed in each training epoch.

$$\frac{N}{batchsize} = \frac{15725}{185} = 85$$

- Activation functions and output activation.
  - "ReLu" activation function
  - "Softmax" output activation (15 classes)
- Regularization (if any).
  - L_2 regularization (Weight Decay) on 1st Dense Layer (`kernel_regularizer = regularizer_l2(0.2)`)
  - Early Stopping on validation loss (`patience = 20`)
- The provided `activity_recognition.Rdata` file is loaded, which has three dimensional arrays of dimensions 8170 x 125 x 45 for train/validation data and 950 x 125 x 45 for test data. Each slice of the data is a 5-second sensor recording over 45 sensors.
- We sample randomly 950 segments for validation (same as test data) and leave the remaining for training data.
- For DNNs, we "flatten" our training and validation data into 2-D ($125x45 = 5625$) using `array_reshape()`
- Classes (y) are split into Training, validation and test data and one-hot label encoded.

```
library(keras)
# Load data
load("activity_recognition.Rdata")
dim(x)
```

```
[1] 8170  125   45
```

```
dim(x_test)
```

```
[1] 950 125  45
```

```
# Splitting data into training and validation
set.seed(24215155)
N <- nrow(x)
val <- sample(1:N, 950)
x.val <- x[val,,]
y.val <- y[val]
x.train <- x[-val,,]
y.train <- y[-val]



# Converting x data to 2-D (flatten)
x.train.dnn <- array_reshape(x.train, c(nrow(x.train),
                                        dim(x.train)[2]*dim(x.train)[3]))
x.val.dnn <- array_reshape(x.val, c(nrow(x.val),
                                        dim(x.val)[2]*dim(x.val)[3]))
x.test.dnn <- array_reshape(x_test, c(nrow(x_test),
                                        dim(x_test)[2]*dim(x_test)[3]))
# one hot encode y labels
y.train <- keras::to_categorical(as.numeric(factor(y.train)))
y.val <- keras::to_categorical(as.numeric(factor(y.val)))
y.test <- keras::to_categorical(as.numeric(factor(y_test)))
```

Model 1: 2-Layer Deep Neural Network with Early Stopping

- Choice of Design:

  - **2 Hidden Layers (256 units –> 128 units)** with **ReLu** activation functions to capture non-linear sensor patterns.
  - **Softmax** output layer for the 19 classes
  - **RMSProp** optimizer used for its adaptive leaerning rate
  - **Early stopping** implemented on *validation accuracy* with patience = 20 to prevent over-training.

```r
# Input shape
V <- ncol(x.train.dnn)
# Defining Model 1
model1 <- keras_model_sequential() %>%
  layer_dense(units = 256, activation = "relu", input_shape = V) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = ncol(y.train), activation = "softmax") %>%
  compile (
    loss = "categorical_crossentropy", metrics = "accuracy",
    optimizer = optimizer_rmsprop(),
  )
# Printing summary
summary(model1)
```

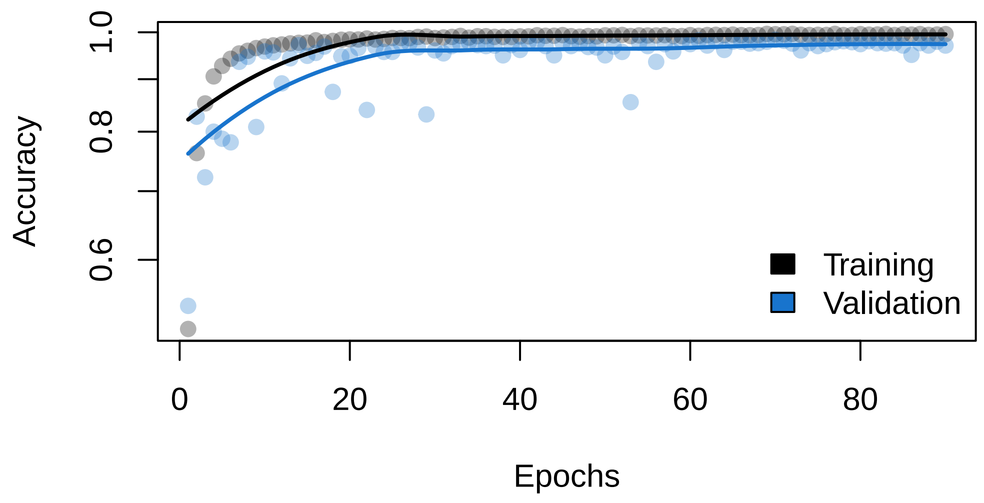
```
Model: "sequential"

________________________________________________________________________________
 Layer (type)                       Output Shape                     Param #
================================================================================
 dense_2 (Dense)                    (None, 256)                      1440256
 dense_1 (Dense)                    (None, 128)                      32896
 dense (Dense)                      (None, 20)                       2580
================================================================================
Total params: 1475732 (5.63 MB)
Trainable params: 1475732 (5.63 MB)
Non-trainable params: 0 (0.00 Byte)

________________________________________________________________________________
```

- Model is fitted with **epochs = 300** to ensure proper convergence and a batch size of 1% (Generally ideal) of Training data.

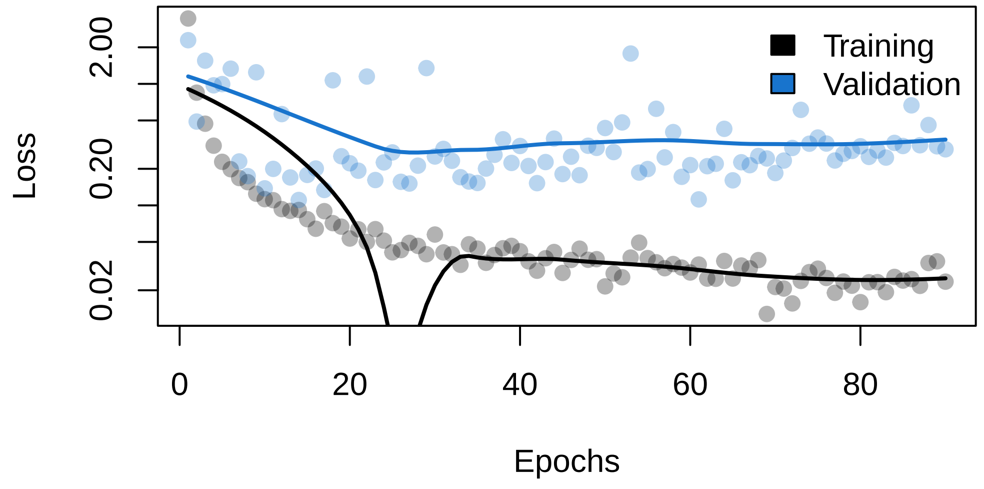
```r
set.seed(24215155)
# batch size set as 1% of training data
bs <- round(N * 0.01)
# Fit Model 1
fit1 <- model1 %>% fit(
  x = x.train.dnn, y = y.train,
  validation_data = list(x.val.dnn, y.val),
  epochs = 300, batch_size = bs, verbose = 0,
  callbacks = list(
    callback_early_stopping(monitor = "val_accuracy", patience = 20)
  )
)
```

- Accuracy and Loss (Training & Validation) by Number of Epochs were plotted to visualize the performance of the 2-layer DNN model.
- We also extract the training and validation accuracy and loss by using `evaluate()` on training and validation data.

```r
# Adding a smooth line to points
smooth_line <- function(y) {
  x <- 1:length(y)
  out <- predict( loess(y ~ x) )
  return(out)
}

# Checking performance (Plotting Accuracy and Loss vs Epochs)
cols <- c("black", "dodgerblue3")
out1 <- cbind(fit1$metrics$accuracy, fit1$metrics$val_accuracy,
              fit1$metrics$loss, fit1$metrics$val_loss)
matplot(out1[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3), log = "y")
matlines(apply(out1[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
       fill = cols, bty = "n")
```



```
matplot(out1[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",
col = adjustcolor(cols, 0.3), log = "y")
matlines(apply(out1[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training", "Validation"),
       fill = cols, bty = "n")
```



```
set.seed(24215155)
# Extracting Training Accuracy and loss (Model 1)
train.metrics.model1 <- model1 %>% evaluate(x.train.dnn, y.train, verbose = 0)
train.metrics.model1
```

```
      loss   accuracy
```

```
0.01834501 0.99529088
```

```r
# Extracting Validation Accuracy and loss (Model 1)
val.metrics.model1 <- model1 %>% evaluate(x.val.dnn, y.val, verbose = 0)
val.metrics.model1
```

```
     loss  accuracy
0.2893439 0.9705263
```

- As per the visualizations, we can see that model fit ran only upto 25 epochs, indicating no drastic improvement in validation accuracy with increase in number of epochs. The loss curves show signs of overfitting, as the gap between validation and training loss increased with number of epochs.
- The large validation accuracy observed at convergence (0.971) is ideal, but the difference in training and validation metrics denote signs of overfitting even with early stopping.

Model 2: Tuned 4-Layer Deep Neural Network

- Choice of Design & Tuning:
  - **4 Hidden Layers (512 units -> 256 units -> 128 units -> 64 units)** used in order to model complex sensor patterns and interactions
  - **ReLU** activation function on all hidden layers, standard **softmax** output activation and cross-entropy loss with Adam optimizer.
  - L_2 regularization hyperparameter (Weight Decay) applied on every layer to reduce overfitting.
  - **Dropout** also included as a regularization parameter for boosting robustness and genralization
  - Optimizer hyperparameters **Learning rate** (η) and **Batch size** also considered for tuning.
  - From the set of given hyperparameter values, tuning is done via **`tfruns`** over a 20% random grid selection from all possible hyperparameter combinations.
  - Hyperparameters from the model run with the best validation accuracy is chosen, then retrained with early stopping (*patience = 20*).
- Hyperparameter value sets are given in the below code chunk:

```r
# For hyperparameter tuning
library(tfruns)
# Hyperparameter values sets
dropout.set <- c(0, 0.2, 0.4)
lambda.set <- c(0, 0.001, 0.005, 0.01)
lr.set <- c(0.001, 0.005, 0.01)
bs.set <- c(0.01, 0.015, 0.02)*N
```

- From the above given hyperparameter values sets, we obtain a grid of $3 \times 4 \times 3 \times 3 = 108$ hyperparameter combinations.

```r
# Code in model_conf.R file
# Default flag values
FLAGS <- flags(
  flag_numeric("dropout", 0.4),
  flag_numeric("lambda", 0.01),
  flag_numeric("lr", 0.01),
  flag_numeric("bs", 100)
)

# model configuration
model2 <- keras_model_sequential() %>%
  layer_dense(units = 512, input_shape = V, activation = "relu",
              name = "layer.1",
              kernel_regularizer = regularizer_l2(FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = 256, input_shape = V, activation = "relu",
              name = "layer.2",
              kernel_regularizer = regularizer_l2(FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = 128, activation = "relu",
              name = "layer.3",
              kernel_regularizer = regularizer_l2(FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = 64, activation = "relu",
              name = "layer.4",
              kernel_regularizer = regularizer_l2(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(learning_rate = FLAGS$lr)
  )

# Seed for reproducibility
set.seed(24215155)
# training and evaluation
fit <- model2 %>% fit(
  x = x.train.dnn, y = y.train,
  validation_data = list(x.val.dnn, y.val),
  epochs = 100,
  batch_size = FLAGS$bs,
  verbose = 1,
  callbacks = callback_early_stopping(monitor = "val_accuracy", patience = 20)
)
```

- Code `model_conf.R` was stored as separate R script. The above code chunk will be called within the function `tuning_run()`, which is used to run all combinations of hyperparameters (flags).
- We only consider a hyperparameter combination sub-sample of 20% from our grid, thus running only 22 combinations.

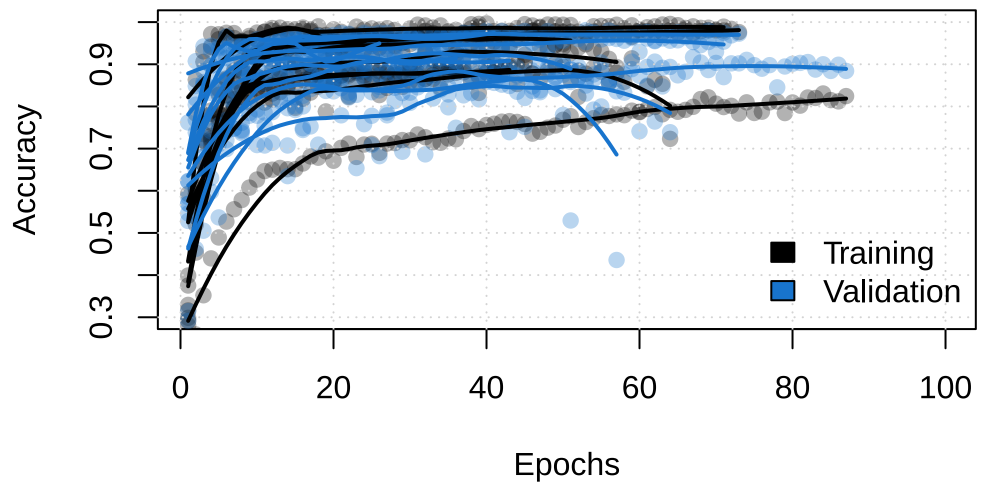
```r
set.seed(24215155)
invisible(
  capture.output(
    runs.dnn<- tuning_run("model_conf.R",
                          runs_dir = "runs_dnn",
                          flags = list(
                            dropout = dropout.set,
                            lambda = lambda.set,
                            lr = lr.set,
                            bs = bs.set
                          ),
                          sample = 0.2)
  )
)
```

- The `read_metrics()` is used to extract values from stored runs and plots from the file name mentioned in `runs_dir`.

```r
read_metrics <- function(path, files = NULL)
# 'path' is where the runs are --> e.g. "path/to/runs"
{
  path <- paste0("runs_dnn", "/")
  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)
  }
```

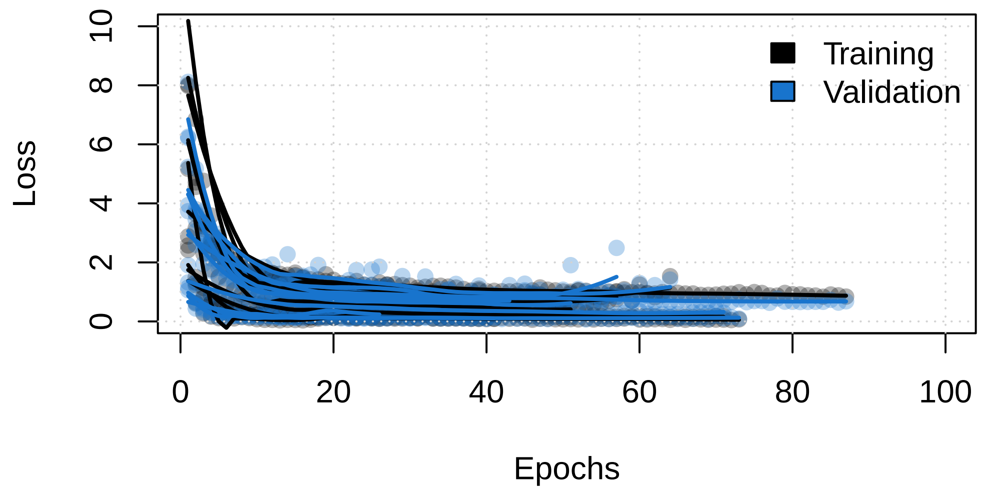
- Once we extract the values, we plot the training and validation accuracy and loss by number of epochs for the top 10 runs ordered by decreasing validation accuracy.

```
# Applying smooth lines to the points
smooth_line <- function(y, span = 0.3) {
  x <- 1:length(y)
  out <- predict(loess(y ~ x, span = span))
  return(out)
}

# Extracting results from results folder
out2 <- read_metrics("runs_dnn")

# Extracting training and validation accuracy and loss
train.acc.model2 <- sapply(out2, "[[", "accuracy")
val.acc.model2 <- sapply(out2, "[[", "val_accuracy")
train.loss.model2 <- sapply(out2, "[[", "loss")
val.loss.model2 <- sapply(out2, "[[", "val_loss")

# Selecting top 10 runs ordered by decreasing Val accuracy
top10 <- order(apply(val.acc.model2, 2, max, na.rm = TRUE),
               decreasing = TRUE)[1:10]

# Plotting accuracy curves to inspect performance and underfitting/overfitting
cols <- rep(c("black", "dodgerblue3"), each = 10)
out2.acc <- cbind(train.acc.model2[,top10], val.acc.model2[,top10])
matplot(out2.acc, pch = 19, ylab = "Accuracy", xlab = "Epochs",
        col = adjustcolor(cols, 0.3), ylim = c(0.3, 1))
grid()
tmp <- apply(out2.acc, 2, smooth_line, span = 0.5)
tmp <- sapply( tmp, "length<-", 100 ) # set default length of 100 epochs
matlines(tmp, lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = unique(cols), bty = "n")
```



```
# Plotting loss curves to inspect performance and underfitting/overfitting
out2.loss <- cbind(train.loss.model2[,top10], val.loss.model2[,top10])
matplot(out2.loss, pch = 19, ylab = "Loss", xlab = "Epochs",
        col = adjustcolor(cols, 0.3), ylim = c(0, 10))
grid()
tmp <- apply(out2.loss, 2, smooth_line, span = 0.5)
tmp <- sapply( tmp, "length<-", 100 ) # set default length of 100 epochs
matlines(tmp, lty = 1, col = cols, lwd = 2)
```

```
legend("topright", legend = c("Training", "Validation"),
fill = unique(cols), bty = "n")
```



- When observing the loss curves, training process show signs of overfitting.
- Reasonably high validation accuracy observed, with top 5 runs having accuracy higher than 90%
- Some training accuracy curves lie lower than the validation accuracy curves. This is due to the fact that while performing dropout, fit is trained on training data with some network nodes “switched off”. However, during validation, full network is used, often resulting in higher observed validation accuracy.
- `ls_runs()` used to extract Top 5 runs having highest validation accuracy

```
# Extracting top 5 runs with the highest validation accuracy > 0.85
top10runs <- ls_runs(metric_val_accuracy > 0.90,
                    runs_dir = "runs_dnn",
                    order = metric_val_accuracy)
# Selecting Accuracy, Lambda, Dropout, Regularization Parameter, Batch size,
# Epoch
sel.col <- c("metric_val_accuracy", grep("flag", colnames(top10runs), value = TRUE),
             "epochs_completed")
top10runs[1:5,sel.col]
```

```
Data frame: 5 x 6
  metric_val_accuracy flag_dropout flag_lambda flag_lr flag_bs epochs_completed
1              0.9726          0.2       0.000   0.001   81.70               73
2              0.9705          0.2       0.000   0.001  122.55               41
3              0.9537          0.0       0.001   0.001  122.55               71
4              0.9400          0.2       0.005   0.001   81.70               43
5              0.9389          0.0       0.000   0.005  122.55               26
```

- **Best Run Hyperparameter values:**
  - Weight Decay (λ) = 0
  - Dropout = 0.2
  - Learning Rate (η) = 0.001
  - Batch Size = 81.7
- We define the best model and then retrain the model with the best model hyperparameter values on the training/validation data with a larger number of epochs (150) to ensure convergence.

```
# Deploying model using optimal hyper-parameters
model2 <- keras_model_sequential() %>%
  layer_dense(units = 512, input_shape = V, activation = "relu",
              name = "layer.1",
              kernel_regularizer =
                regularizer_l2(top10runs$flag_lambda[1])) %>%
  layer_dropout(rate = top10runs$flag_dropout[1]) %>%
  layer_dense(units = 256, input_shape = V, activation = "relu",
              name = "layer.2",
              kernel_regularizer =
                regularizer_l2(top10runs$flag_lambda[1])) %>%
  layer_dropout(rate = top10runs$flag_dropout[1]) %>%
  layer_dense(units = 128, activation = "relu",
              name = "layer.3",
              kernel_regularizer =
                regularizer_l2(top10runs$flag_lambda[1])) %>%
  layer_dropout(rate = top10runs$flag_dropout[1]) %>%
  layer_dense(units = 64, activation = "relu",
              name = "layer.4",
              kernel_regularizer =
                regularizer_l2(top10runs$flag_lambda[1])) %>%
  layer_dropout(rate = top10runs$flag_dropout[1]) %>%
  layer_dense(units = ncol(y.train), activation = "softmax",
              name = "layer.out") %>%
  compile(loss = "categorical_crossentropy", metrics = "accuracy",
          optimizer = optimizer_adam(learning_rate = top10runs$flag_lr[1])
  )
# Summary of Model 2
summary(model2)
```

```
Model: "sequential"
________________________________________________________________________________
 Layer (type)                       Output Shape                    Param #
================================================================================
 layer.1 (Dense)                    (None, 512)                     2880512
 dropout_3 (Dropout)                (None, 512)                     0
 layer.2 (Dense)                    (None, 256)                     131328
 dropout_2 (Dropout)                (None, 256)                     0
 layer.3 (Dense)                    (None, 128)                     32896
 dropout_1 (Dropout)                (None, 128)                     0
 layer.4 (Dense)                    (None, 64)                      8256
 dropout (Dropout)                  (None, 64)                      0
 layer.out (Dense)                  (None, 20)                      1300
================================================================================
Total params: 3054292 (11.65 MB)
Trainable params: 3054292 (11.65 MB)
Non-trainable params: 0 (0.00 Byte)
________________________________________________________________________________
```

```
set.seed(24215155)
# Fitting Model with best hyperparameters
fit2 <- model2 %>% fit(
  x = x.train.dnn, y = y.train,
  validation_data = list(x.val.dnn, y.val),
  epochs = 150,
  batch_size = top10runs$flag_bs[1],
  verbose = 0,
  callbacks = callback_early_stopping(monitor = "val_accuracy", patience = 20)
)
```

- We extract the training and validation accuracy and loss by using `evaluate()` on training and validation data for the Tuned DNN model (Model 2)

```
set.seed(24215155)
# Extracting Training Accuracy and loss (Model 2)
train.metrics.model2 <- model2 %>% evaluate(x.train.dnn, y.train, verbose = 0)
train.metrics.model2
```

```
    loss   accuracy
```

```
0.01538579 0.99487537
```

```
# Extracting Validation Accuracy and loss (Model 2)
val.metrics.model2 <- model2 %>% evaluate(x.val.dnn, y.val, verbose = 0)
val.metrics.model2
```

```
     loss  accuracy
0.1049439 0.9736842
```

- Compared to **Model 1 (2-Layer DNN)**, we observe a **better** generalized predictive performance in **Model 2 (Tuned 4-Layer DNN)** as the validation loss is almost half of what is seen in Model 1 while the validation accuracy remains almost the same.

Model 3: 4-Layer Convolutional Neural Network with Early stopping, Dropout and L2 regularization

- Here training data points are treated as 125x45 sensor single "images". So we add an additional dimension (or channel) to our Training, Validation and Test data.
- Choice of Design & Tuning:
  - **Three 2-D convolution layers (32 filter, 2x2 kernel -> 64 filters, 3x3 kernel -> 64 filters, 3x3 kernel)**
  - **Max pooling at each layer** (2x2)
  - Final dense layer with 128 units (**ReLu activation function,** L_2 regularization = 0.0001, dropout = 0.1)
  - Output layer with 19 units (classes) (**Softmax activation**)
  - **Adam optimizer** with early stopping (**patience = 20**)
- We train the CNN model on the training/validation data.

```
# Converting x data to 4-D for CNN as CNN (layer_conv_2D) takes in 4-D data
x.train.cnn <- array_reshape(x.train, c(nrow(x.train),
                                        dim(x.train)[2],
                                        dim(x.train)[3], 1))
x.val.cnn <- array_reshape(x.val, c(nrow(x.val),
                                    dim(x.val)[2],
                                    dim(x.val)[3], 1))
x.test.cnn <- array_reshape(x_test, c(nrow(x_test),
                                      dim(x_test)[2],
                                      dim(x_test)[3], 1))
# Defining model
model3 <- keras_model_sequential() %>%
  # Convolutional layers
  layer_conv_2d(filters = 32, kernel_size = c(2,2), activation = "relu",
                input_shape = c(dim(x.train.cnn)[2],
                                dim(x.train.cnn)[3],
                                dim(x.train.cnn)[4])) %>%
  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 = 128, activation = "relu", kernel_regularizer =
                regularizer_l2(0.001)) %>%
  layer_dropout(0.1) %>%
  layer_dense(units = ncol(y.train), activation = "softmax") %>%
  # Compiling
  compile(
    loss = "categorical_crossentropy", metrics = "accuracy",
    optimizer = optimizer_adam()
    )
```

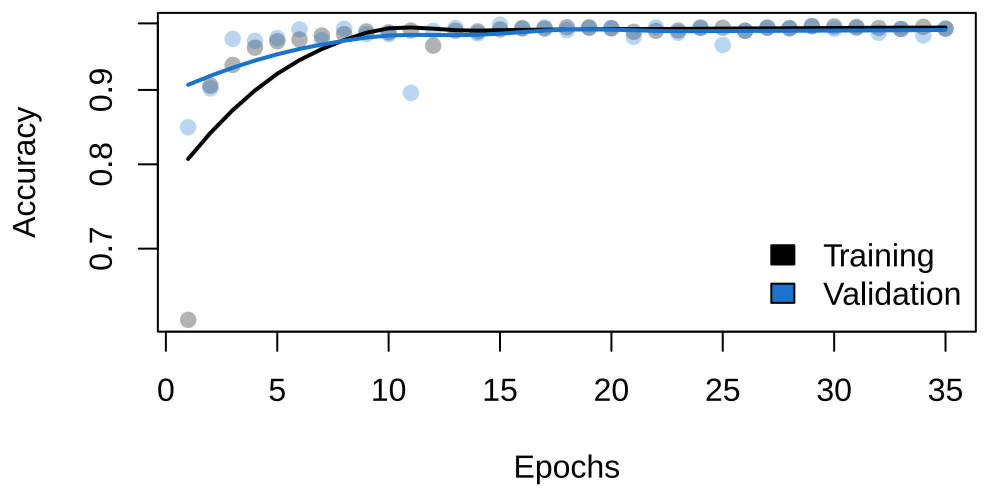
```r
# Summary of CNN Model
summary(model3)
```

```
Model: "sequential_1"

________________________________________________________________________________
 Layer (type)                       Output Shape                      Param #
================================================================================
 conv2d_2 (Conv2D)                  (None, 124, 44, 32)               160
 max_pooling2d_2 (MaxPooling2D)     (None, 62, 22, 32)                0
 conv2d_1 (Conv2D)                  (None, 60, 20, 64)                18496
 max_pooling2d_1 (MaxPooling2D)     (None, 30, 10, 64)                0
 conv2d (Conv2D)                    (None, 28, 8, 64)                 36928
 max_pooling2d (MaxPooling2D)       (None, 14, 4, 64)                 0
 flatten (Flatten)                  (None, 3584)                      0
 dense_1 (Dense)                    (None, 128)                       458880
 dropout_4 (Dropout)                (None, 128)                       0
 dense (Dense)                      (None, 20)                        2580
================================================================================
Total params: 517044 (1.97 MB)
Trainable params: 517044 (1.97 MB)
Non-trainable params: 0 (0.00 Byte)
________________________________________________________________________________
```

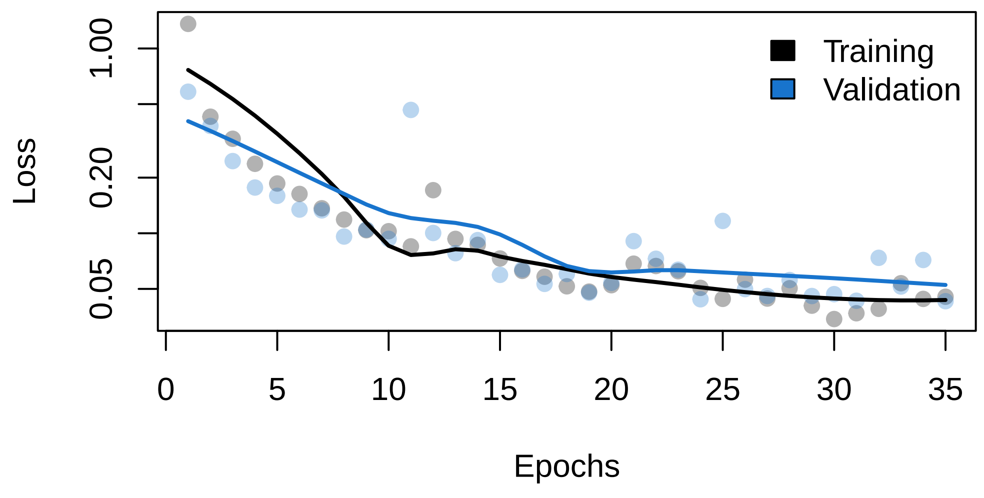
```r
set.seed(24215155)
# model training
fit3 <- model3 %>% fit(
x = x.train.cnn, y = y.train,
validation_data = list(x.val.cnn, y.val),
epochs = 150,
batch_size = bs,
callbacks = list(callback_early_stopping(monitor = "val_accuracy", patience = 20)),
verbose = 0
)
```

- We extract the performance metrics from the CNN fit and plot the training and validation accuracy and loss by number of epochs.

```r
# Adding a smooth line to points
smooth_line <- function(y) {
  x <- 1:length(y)
  out <- predict( loess(y ~ x) )
  return(out)
}
# Checking performance
cols <- c("black", "dodgerblue3")
out3 <- cbind(fit3$metrics$accuracy, fit3$metrics$val_accuracy,
             fit3$metrics$loss, fit3$metrics$val_loss)
matplot(out3[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3), log = "y")
matlines(apply(out3[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
       fill = cols, bty = "n")
```



```r
matplot(out3[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",
col = adjustcolor(cols, 0.3), log = "y")
matlines(apply(out3[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training", "Validation"),
       fill = cols, bty = "n")
```



- The CNN network attains great predictive performance on the validation data as seen from our plots.
- So, lets evaluate the CNN model on training and validation data to compare performance metrics.

```r
set.seed(24215155)
# Extracting Training Accuracy and loss (Model 3)
train.metrics.model3 <- model3 %>% evaluate(x.train.cnn, y.train, verbose = 0)
train.metrics.model3
```

```
      loss   accuracy
0.03948023 0.99376732
```

```r
# Extracting Validation Accuracy and loss (Model 3)
val.metrics.model3 <- model3 %>% evaluate(x.val.cnn, y.val, verbose = 0)
val.metrics.model3
```

```
      loss   accuracy 
0.04289421 0.99157894 
```

- **Model 3 (CNN Model)** has both high training and validation accuracy, combined with minimal loss (**Accuracy = 0.9915789, Loss = 0.0428942**) - suggesting it trains and generalizes extremely well.

Results

- For better viewing and ease of comparison between predictive performances of model, we tabulate the Validation Accuracy and Losses across all three models.

```r
# All performance results of Models in Dataframe
results <- data.frame(
  Model          = c("Model 1",
                     "Model 2",
                     "Model 3"),
  Type           = c("2-Layer DNN",
                     "4-Layer Tuned DNN",
                     "4-Layer CNN"),
  Val_Accuracy   = c(val.metrics.model1["accuracy"],
                     val.metrics.model2["accuracy"],
                     val.metrics.model3["accuracy"]),
  Val_Loss       = c(val.metrics.model1["loss"],
                     val.metrics.model2["loss"],
                     val.metrics.model3["loss"])
)

# Display table using kable()
knitr::kable(
  results,
  digits = 3,
  caption = "Comparison of Validation Performance across Models"
)
```

Table 1: Comparison of Validation Performance across Models

Model	Type	Val_Accuracy	Val_Loss
Model 1	2-Layer DNN	0.971	0.289
Model 2	4-Layer Tuned DNN	0.974	0.105
Model 3	4-Layer CNN	0.992	0.043

- **Model 3** (*4-Layer Convolutional Neural Network with Early stopping, Dropout and L2 regularization*) is selected as the best model as it has the best predictive performance metrics (Validation Accuracy = 0.992 and Loss = 0.043) among the 3 selected models.
- Now, we evaluate **Model 3** on the test dataset to compute its true out-of-sample predictive performance using `evaluate()`

```r
set.seed(24215155)
# Computing test accuracy and loss
eval.model3 <- model3 %>% evaluate(x.test.cnn, y.test, verbose = 0)
eval.model3
```

```
      loss   accuracy 
0.06502713 0.98736840 
```

```r
# Confusion matrix
y.raw <- max.col(y.test) - 1
class.hat.model3 <- model3 %>% predict(x.test.cnn, verbose = 0) %>% max.col()
tab.model3 <- table (True = y.raw, Pred = class.hat.model3-1)
tab.model3
```

```
    Pred
True  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
  1  50  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  2    0 50  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  3    0  0 50  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  4    0  0  0 50  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  5    0  0  0  0 50  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  6    0  0  0  0  0 50  0  0  0  0  0  0  0  0  0  0  0  0  0
  7    0  0  0  0  0  0 50  0  0  0  0  0  0  0  0  0  0  0  0
  8    0  0  0  0  0  0  0 50  0  0  0  0  0  0  0  0  0  0  0
  9    0  0  0  0  0  0  0  0 50  0  0  0  0  0  0  0  0  0  0
  10   0  0  0  0  0  0  0  0  0 38  0  0  0 10  2  0  0  0  0
  11   0  0  0  0  0  0  0  0  0  0 50  0  0  0  0  0  0  0  0
  12   0  0  0  0  0  0  0  0  0  0  0 50  0  0  0  0  0  0  0
  13   0  0  0  0  0  0  0  0  0  0  0  0 50  0  0  0  0  0  0
  14   0  0  0  0  0  0  0  0  0  0  0  0  0 50  0  0  0  0  0
  15   0  0  0  0  0  0  0  0  0  0  0  0  0  0 50  0  0  0  0
  16   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 50  0  0  0
  17   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 50  0  0
  18   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 50  0
  19   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 50
```

```r
# Class-wise accuracy, precision
acc.model3 <- diag(tab.model3)/rowSums(tab.model3)
prec.model3 <- diag(tab.model3)/colSums(tab.model3)
y.labels <- levels(factor(y))
class.metrics <- data.frame(Class = y.labels, Accuracy = acc.model3, Precision = prec.model3)
class.metrics
```

```
               Class Accuracy Precision
1          asc_stairs     1.00 1.0000000
2          basketball     1.00 1.0000000
3       cross_trainer     1.00 1.0000000
4       cycling_horiz     1.00 1.0000000
5        cycling_vert     1.00 1.0000000
6         desc_stairs     1.00 1.0000000
7             jumping     1.00 1.0000000
8          lying_back     1.00 1.0000000
9          lying_side     1.00 1.0000000
10    moving_elevator     0.76 1.0000000
11             rowing     1.00 1.0000000
12  running_treadmill     1.00 1.0000000
13            sitting     1.00 1.0000000
14     stand_elevator     1.00 0.8333333
15           standing     1.00 0.9615385
16            stepper     1.00 1.0000000
17            walking     1.00 1.0000000
18      walking_tread     1.00 1.0000000
19 walking_tread_incl     1.00 1.0000000
```

Results

- **Overall Accuracy = 0.987**, **Overall Loss = 0.065**
- Most class-wise accuracy and precision values are all at or near 1, with "moving_elevator" class showing a slightly lower accuracy of 0.76 and "stand_elevator" with a slightly lower precision 0.8333333