Human Activity Recognition with Deep Learning

DNN vs CNN

Jainel Liz Boban

Table of Contents

Load data

- The data consists of x,y,x_test and y_test objects.
- The dimension of x and x_test are given below.
- The aim is to predict one of 19 activities from 125x45 multivariate data.

```r
library(keras) # for DNN and CNN
# Load RData file and list objects in it
load("data_activity_recognition.RData")

dim(x)          # full training set input size
```

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

```r
dim(x_test)     # test set input size
```

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

- Each observation in 'x' and 'x_test' is a signal segment represented as a 2D matrix of 125 sampling instants by 45 sensor features.
- There are 8170 samples in training and 950 samples in the test set.
- Therefore, x and x_test are 3D arrays and can be used for Convolutional Neural network (CNN) using 1D convolutional layer or reshape it as 4D to use 2D convolutional layer.
- However, we can reshape it to 2D for use in Deep Nueral Networks(DNN).

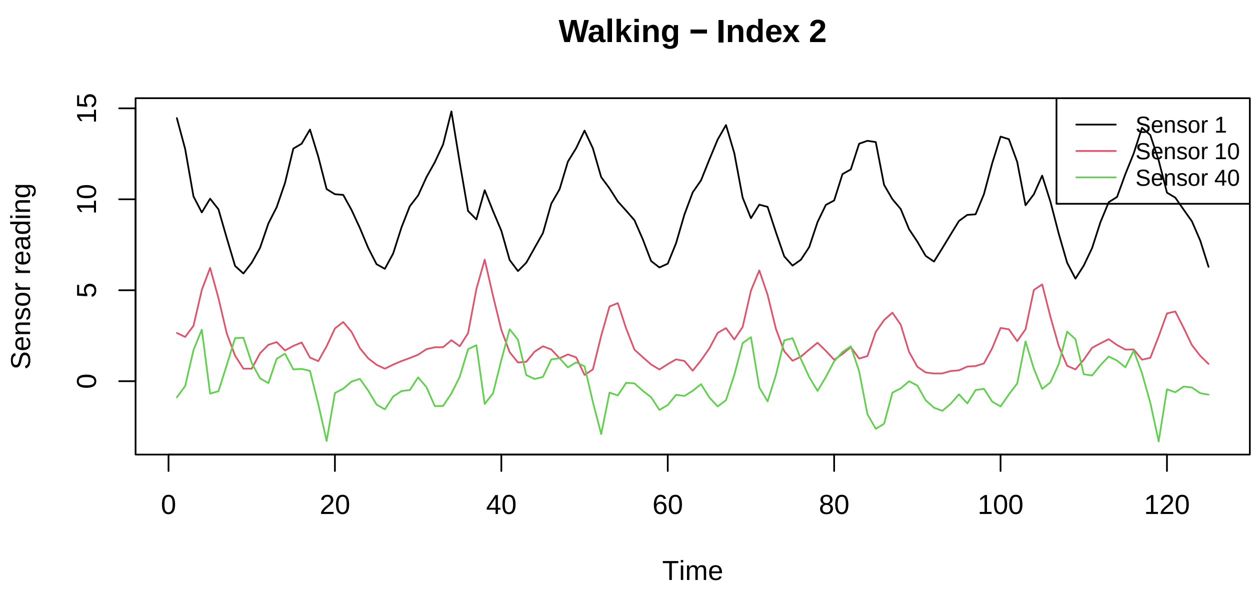
Visualize data

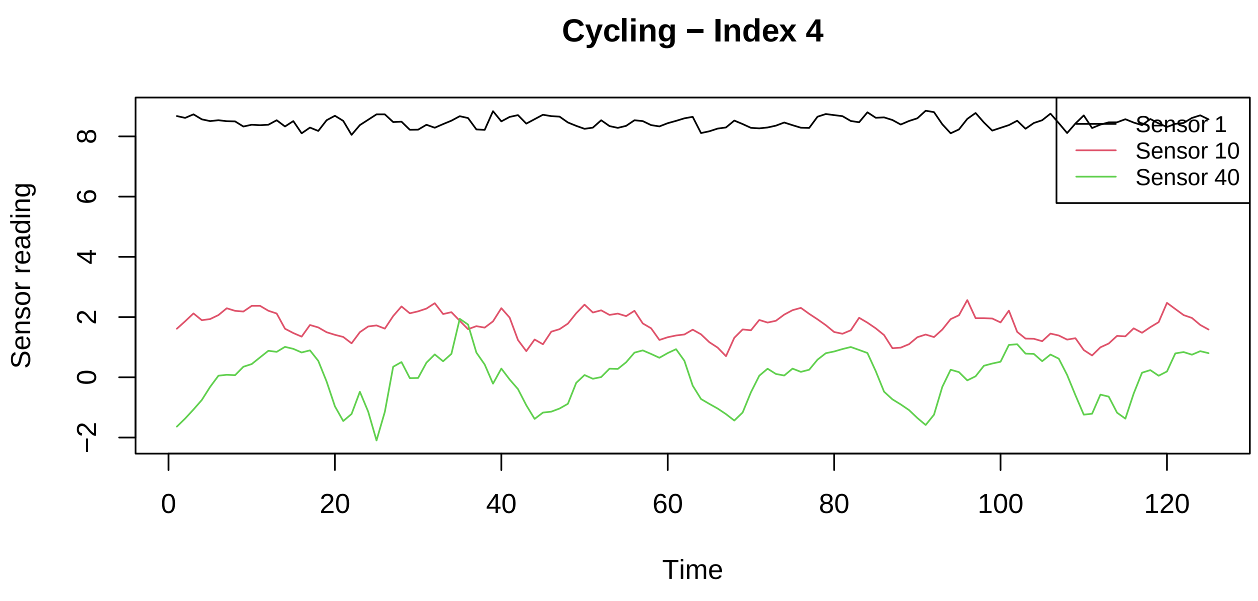
The sensor data can be viewed as a time series (sampling instants) as shown in plot:

```r
par(mfrow = c(2, 1))
# custom plot function to view data
plot_signal <- function(index, data, sensors, title = "") {
  matplot(data[index,,sensors], type = "l", lty = 1,
          ylab = "Sensor reading", xlab = "Time",
          main = title, col = 1:length(sensors))
  legend("topright", legend = paste("Sensor", sensors),
         col = 1:length(sensors), lty = 1, cex = 0.8)
}

# first walking signal
index_walking <- which(grepl("walking", y))[1]
# print 1st, 10th and 40th sensor output for walking activity
plot_signal(index_walking, x, sensors = c(1, 10, 40),
                  title = paste("Walking - Index", index_walking))

# first cycling signal
index_cycling <- which(grepl("cycling", y))[1]
plot_signal(index_cycling, x, sensors = c(1, 10, 40),
                  title = paste("Cycling - Index", index_cycling))
```





Train-validation-test splitting

- We will split the entire training data into 'train' and 'test' sets as shown.
- We will pick same number of samples for validation data-set as in the test set (950).
- For DNNs, each input sample must be 1D feature vector. Therefore, the input data should be first converted to a 2D matrix format where each row corresponds to one observation and each column corresponds to one feature.
- Since we are doing multi-class classification, we convert target sets to one-hot encoding format using *to_categorical()* from *keras* library.

```r
set.seed(24214925) # seed for reproducibility

# one-hot encoding of target variable
class_labels <- levels(factor(y))
n_classes <- length(class_labels) # number of classes to identify = 19
y <- as.numeric(factor(y)) - 1
y_test <- as.numeric(factor(y_test)) - 1
y <- keras::to_categorical(y, num_classes = n_classes)
y_test <- keras::to_categorical(y_test, num_classes = n_classes)
```

```
# convert train and test input into 125x45 vectors for DNN
x_dnn <- array_reshape(x, dim = c(nrow(x), 125*45))
x_test_dnn  <- array_reshape(x_test, dim = c(nrow(x_test), 125*45))

#val train splitting
val <- sample(1:nrow(x), 950) # there are 950 activity labels in x_test
train <- setdiff(1:nrow(x), val) # remaining indices for training
y_val <- y[val,]                # target val (DNN and CNN)
y_train <- y[-val,]             # target train (DNN and CNN)
x_val_dnn <- x_dnn[val,]        # input val (DNN)
x_train_dnn <- x_dnn[-val,]     # input train (DNN)
x_val_cnn <- x[val,,]           # input val (CNN)
x_train_cnn <- x[-val,,]        # input train (CNN)

N = nrow(x) # no. of observation in training set (DNN and CNN)
V <- ncol(x_dnn) # (DNN)
```

Model 1: DNN (2-hidden layer) + Early stoppping

- The first model is a deep neural network with mulitple layers- it has 2 hidden layers, both using "RELU"(Rectified Linear Unit) activation function.
- This activation function is chosen for the hidden layers because it avoids the vanishing gradient issue more effectively, as its derivative remains 1 for positive inputs unlike sigmoid or tanh.
- The output activation is selected as "Softmax" because the model is predicting one of many classes (19 activities) and it turns the outputs into probabilities for each class.
- This is well suited for the loss function "categorical cross-entropy" that we use here for multi-class classification.
- We use the "rmsprop" optimizer to use adaptive learning rate per parameter to adjust individual weights.
- It is chosen here as it is faster than Stochastic Gradient Descent (SGD) but simpler than Adam.
- Here, we use **early stopping** as a regularization technique to prevent over-fitting.
- This lets us to monitor model performance on the validation set during training (say "validation loss" or "validation accuracy") and stop training when performance stops improving.
- *Patience* attribute allows training to continue for a number of epochs even if no improvement is seen.
- This gives the model a chance to recover in that period.
- According to below code the training will stop early if it does not improve for 20 continuous epochs, saving training time.
- Additionally, we also reduce learning rate by 10% if loss does not improve for 10 continuous epochs, giving the optimizer a smaller step size to tune better (fine-tune).
- This is external to RMSprop as it adjusts the global learning rate and works well together with it.

```
# Model 1 configuration
model1 <- keras_model_sequential() %>%
#first hidden layer with 128 hidden units and RELU activation
layer_dense(units = 128, activation = "relu", input_shape = V) %>%
#second hidden layer with 64 hidden units and RELU activation
layer_dense(units = 64, activation = "relu") %>%
#output layer with softmax activation with multiclass
layer_dense(units = ncol(y_train), activation = "softmax") %>%
compile( # compile model
    loss = "categorical_crossentropy", # loss function for multiclass
    metrics = "accuracy", # metric to evaluate
    optimizer = optimizer_rmsprop(), # adapt optimization rmsprop
    )
# training and evaluation
fit_model1<- model1 %>% fit(
    x = x_train_dnn, y = y_train,
    validation_data = list(x_val_dnn, y_val),
    epochs = 100, # model goes through entire training set 100 times
    batch_size =  round(N * 0.01), # Batch size = 1% of training set
    verbose = 0,
    callbacks = list(
        # early stopping if validation accuracy doesn't improve for 20 epochs
        callback_early_stopping(monitor = "val_accuracy", patience = 20),
        # learning rate reduced by 0.1 if loss do not improve for 10 epochs
        callback_reduce_lr_on_plateau(monitor ="loss", patience=10, factor=0.1))
)
# we can also count parameters in a model as shown
count_params(model1)
```

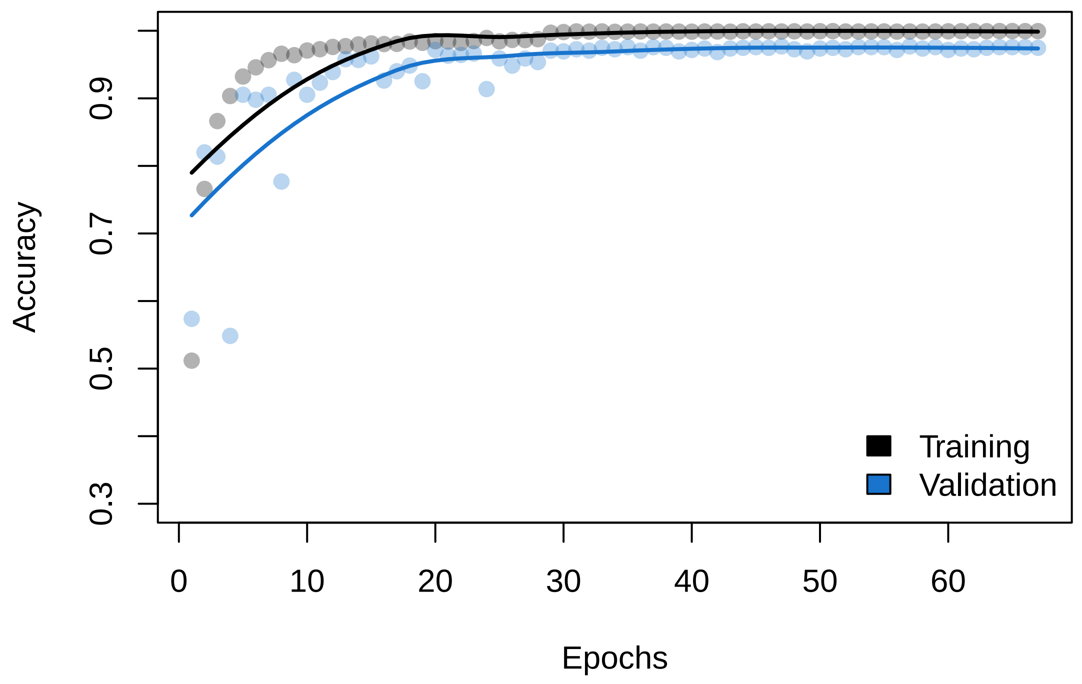
```
[1] 729619
```

Plot for performance on training and validation data

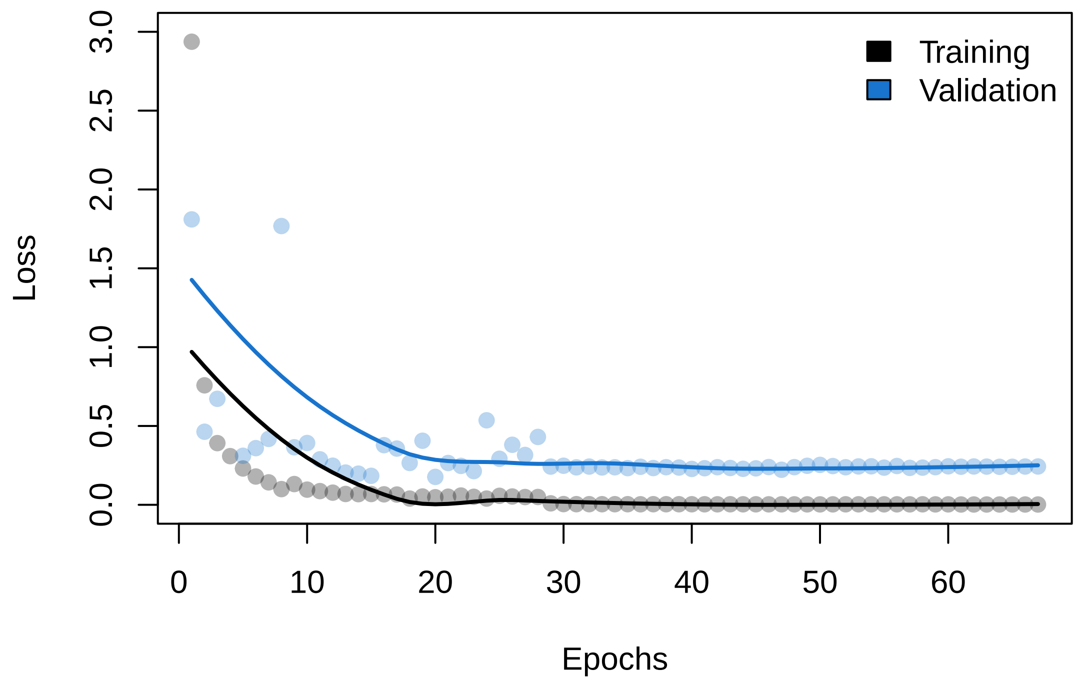
```
# function to add a smooth line to points in plot
smooth_line <- function(y , span=0.75) {
    x <- 1:length(y)
    out <- predict(loess(y ~ x), span=span)
    return(out)
}
cols <- c("black", "dodgerblue3")

# learning curves
out_model1 <-cbind(fit_model1$metrics$loss,fit_model1$metrics$val_loss,
                   fit_model1$metrics$accuracy,fit_model1$metrics$val_accuracy)

# accuracy plot
matplot(out_model1[,3:4], pch = 19, ylab = "Accuracy", xlab = "Epochs",
        col = adjustcolor(cols, 0.3), ylim = c(0.3, 1))
matlines(apply(out_model1[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend =c("Training","Validation"),fill = cols, bty = "n")
```



```
# loss plot
matplot(out_model1[,1:2],pch= 19, ylab= "Loss",xlab= "Epochs",
        col= adjustcolor(cols, 0.3),ylim= c(0, 3))
matlines(apply(out_model1[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training","Validation"),fill = cols, bty = "n")
```



Observations:

- There seems to be good generalization with no significant over-fitting as the training and validation curves are close (training accuracy almost saturates close to 1).
- We can also that the training stopped when a certain number of epochs were reached (did not reach 100 as specified) before significant over-fitting.
- There is a gap between training and validation loss shows a slight over-fitting. However, it is not that severe and we can say that the model has learned the patterns without just simply learning the training data.

Performance evaluation on training and validation data

- We can assess the performance of the model on the training and validation sets at convergence using *evaluate()* as shown:

```
#performance on training data
train_metrics_model1 <- model1 %>% evaluate(x_train_dnn, y_train, verbose = 0)
train_metrics_model1
```

```
       loss    accuracy
0.001832718 0.999445975
```

```
# evaluate performance on validation data
val_metrics_model1 <- model1 %>% evaluate(x_val_dnn, y_val, verbose = 0)
val_metrics_model1
```

```
     loss  accuracy
0.2430505 0.9747369
```

Observations:

- The final training accuracy is 99.94% with a very low training loss of 0.0018.
- Therefore the model fits the training data really well.
- The validation accuracy is 97.47% which is also high.
- The slightly high validation loss of 0.2431 suggests a small amount of over-fitting, but it is not as severe.
- Overall, the model generalizes well on unseen data (validation data) and the early stopping and regularization method seems to be effective.

Model 2: DNN (3-hidden layer) + Dropout + L2 Regularization with tuning for dropout rate, λ, learning rate η and no. of nodes in a layer.

- Below is a training code stored in an external R file. This script will be passed to *tuning_run()* using *tfruns* as shown in the next code.
- *tuning_run()* re-executes the model code for each combination of hyperparameters.
- In this deep neural network model, we will use 3 hidden layers with "RELU" activation function (same activation function as before for hidden layers).
- Output activation function is selected again as "Softmax".
- We use FLAGS here to define tunable hyperparameters such as: dropout rate, regularization parameter (L2 regularization), learning rate, batch size and first hidden layer size.
- We use the "Adam" optimizer here as it is a fast and widely used adaptive learning rate method.
- **Dropout** technique use used here for regularization to prevent overfitting.
- This will randomly drop some units during training time, so that the model will not rely on specific units (neurons).
- As dropout rate increases, there will be more regularization. However, we have to also prevent under-fitting. Hence, we can tune this parameter to get good performance.
- In addition to dropout, we can also use **L2 regularization** which adds a penalty term to the loss function to reduce complexity as shown:

$$E_Q(\mathbf{w}, \lambda) = E(\mathbf{w}) + \lambda \frac{1}{2} \mathbf{w}^\top \mathbf{w})$$

- Tuning penalty parameter λ will allow us to balance between under and over-fitting.
- Weight update equation with L2 regularization becomes:

$$\mathbf{w} \leftarrow \mathbf{w}(1 - \eta\lambda) - \eta \nabla E(\mathbf{w})$$

- As we can see **Learning rate** η controls how fast the model learns and also affects the optimizer.
- Therefore, by well-tuning η, we can get faster convergence.
- Tuning helps to balance between overshooting (too high η) and slow learning (too low η).
- The **batch-size** is also made tunable in this model.
- There should be a balance for batch size as too small batch size could lead to slow training and too big batch size may overfit the training data.
- The **number of neurons/units** in the layers of the network can also be tuned as it decides the complexity of the functions it can learn.
- Too many neurons may lead to overfitting and too few may underfit.
- Due to computational limitation, this has been demonstrated only for hidden layer 1 in the code:

```r
# content of file 'Model2_tfruns.R'

# default flags for the code
FLAGS <- flags(
  flag_numeric("dropout", 0.3),
  flag_numeric("lambda", 0.01),
  flag_numeric("lr", 0.01),
  flag_numeric("bs", 100),
  flag_integer("size_1", 256)
  )
# model configuration
model_tune <- keras_model_sequential() %>%
  # first hidden layer units tunable
  layer_dense(units = FLAGS$size_1, input_shape = V, activation = "relu",
              name = "layer_1",
              # L2 Regularization done with tunable parameter lambda
              kernel_regularizer = regularizer_l2(FLAGS$lambda)) %>%
  # dropout rate tunable
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = 128, activation = "relu", name = "layer_2",
              kernel_regularizer = regularizer_l2(FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = 64, activation = "relu", name = "layer_3",
              kernel_regularizer = regularizer_l2(FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = ncol(y_train), activation = "softmax",
              name = "layer_out") %>%
  compile(loss = "categorical_crossentropy", metrics = "accuracy",
          # adapative optimizer Adam with tunable learning rate
          optimizer = optimizer_adam(learning_rate = FLAGS$lr),
  )
# training and evaluation
fit_model_tune <- model_tune %>% fit(
    x = x_train_dnn, y = y_train,
    validation_data = list(x_val_dnn, y_val),
    epochs = 100,
    batch_size = FLAGS$bs, # batch size tunable
    verbose = 0,
    callbacks = callback_early_stopping(monitor = "val_accuracy",
                                        patience = 20))
```

- Using the above file (saved as *Model2_tfruns.R* in the current working directory) can be used along package *tfruns* to easily tune and experiment with different settings (hyperparameter combinations) for the model.
- We specify a grid of values for the hyperparameters of interest.
- We will be using random search because in case of grid search, it uses all combinations (3x3x3x2x2=108 combinations in total as per below grid) for tuning and is computationally expense.
- It is also better than manually tuning hyperparameters as it takes more time to manually tune a couple of parameters and may miss best combinations by only testing fewer options.
- In random search, we randomly sample a proportion of the total combinations (20% as per below code, reducing the count to much lesser 22 combinations).
- This will help us find acceptable results with much fewer runs.

```
set.seed(24214925)

library(tfruns)
# grid with different combinations for tuning
dropout_set <- c(0, 0.3, 0.4) # dropout rate
lambda_set <- c(0, 0.005, 0.01) # regularization parameter
lr_set <- c(0.001, 0.005, 0.01) # learning rate
bs_set <- c(0.01, 0.05)*N # batch size in percentages of no. of samples
size1_set <- c(512, 256) # tuning no. of nodes first hidden layer

# Use random search to pick random combinations from 108 combinations above
invisible(runs <- tuning_run("Model2_tfruns.R",
runs_dir = "runs_model2", # save results in folder
flags = list(
    dropout = dropout_set,
    lambda = lambda_set,
    lr = lr_set,
    bs = bs_set,
    size_1 = size1_set),
sample = 0.2)) # sample 20% = 22 combinations
```

Plot of predictive performance top 5 Tuned Model on training and validation data

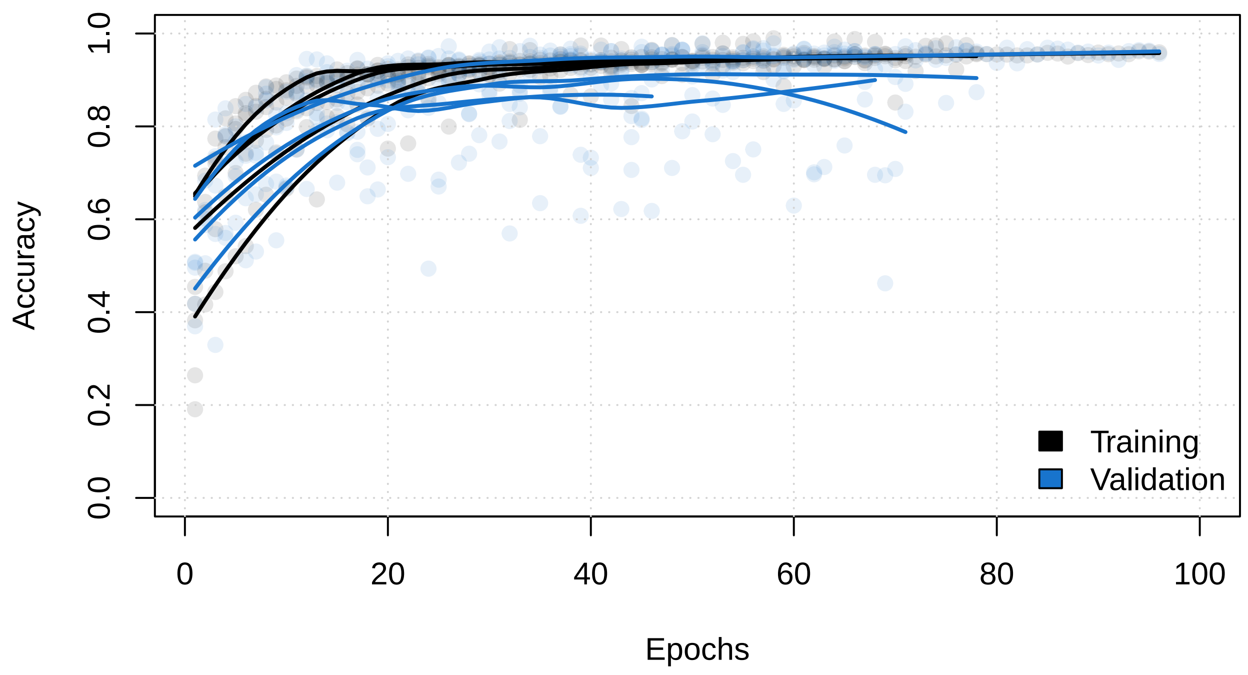
```
#custom function to read training matrix
read_metrics <- function(path, files = NULL){
  path <- paste0(path, "/")
  if (is.null(files)) files <- list.files(path)
  n <- length(files) # no. of runs (folders)
  out <- vector("list", n) # store metrics for each run
  for (i in 1:n){ # loop over each folder
      dir <- paste0(path, files[i], "/tfruns.d/") # path to directory
      tryCatch({
           #training metrics
          out[[i]] <- jsonlite::fromJSON(paste0(dir, "metrics.json"))
```

```r
          # hyperparameter values
          out[[i]]$flags <- jsonlite::fromJSON(paste0(dir, "flags.json"))
          #evaluation metrics
          out[[i]]$evaluation <- jsonlite::fromJSON(paste0(dir, "evaluation.json"))
      }, error = function(e){
          # skip corrupt files
          message("Corrupt file ", files[i], ": ", e$message)
          out[[i]] <- NULL
      })
  }
  out <- Filter(Negate(is.null),out) # remove failed runs
  return(out)
}
# results from folders
out <- read_metrics("runs_model2")
#  training and validation across runs
train_acc <- sapply(out, "[[","accuracy")
val_acc <- sapply(out, "[[","val_accuracy")
train_loss <- sapply(out, "[[","loss")
val_loss <- sapply(out, "[[","val_loss")
```

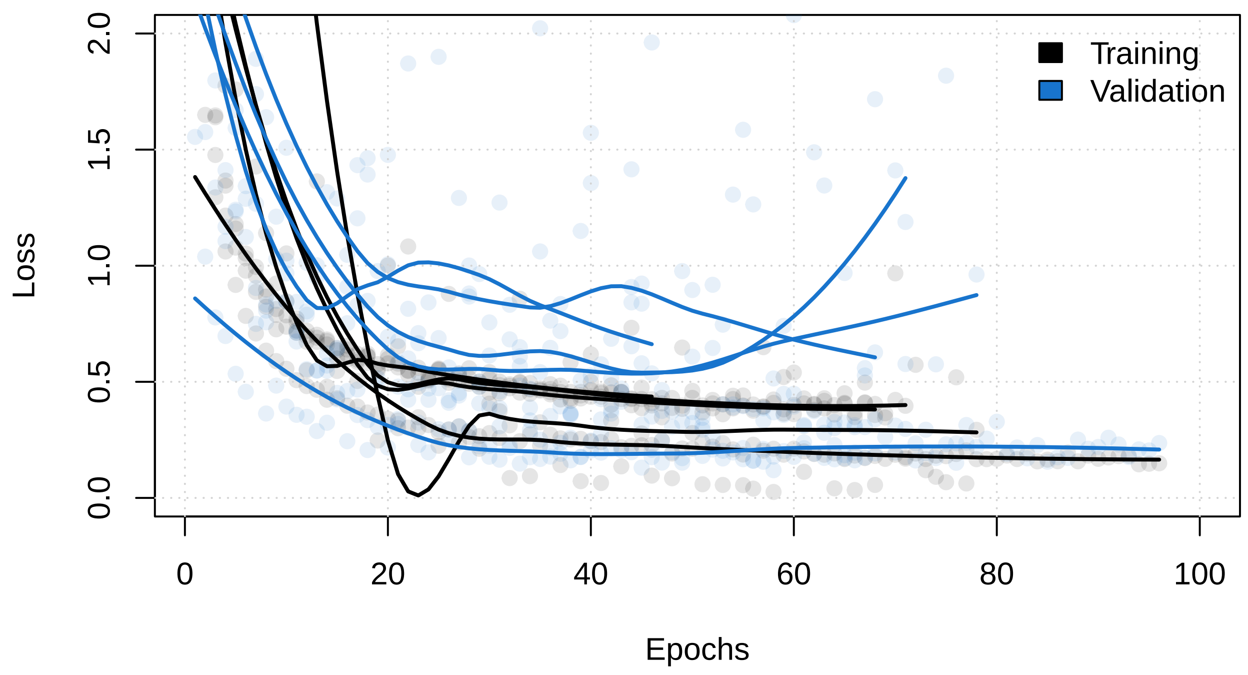
Accuracy and loss curves are given below:

```r
# select top 5 runs by validation accuracy for plotting
sel <- 5
top <- order(apply(val_acc, 2, max, na.rm = TRUE), decreasing = TRUE)[1:sel]
cols_m2  <- rep(c("black", "dodgerblue3"), each = sel)

# plot accuracy curves
out_acc <- cbind(train_acc[, top], val_acc[, top])
matplot(out_acc, pch = 19, ylab = "Accuracy", xlab = "Epochs",
  col = adjustcolor(cols_m2, 0.1), ylim = c(0, 1))
grid()
tmp <- apply(out_acc, 2, smooth_line, span=0.8) # smoothed lines over curves
tmp <- sapply(tmp, "length<-", 100) # 100 epochs by default
matlines(tmp, lty = 1, col = cols_m2, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"), # identify train/test
  fill = unique(cols_m2), bty = "n")
```



```
# plot loss curves
out_loss  <- cbind(train_loss[, top], val_loss[, top])
matplot(out_loss, pch = 19, ylab = "Loss", xlab = "Epochs",
  col = adjustcolor(cols_m2, 0.1), ylim = c(0, 2))
grid()
tmp <- apply(out_loss, 2, smooth_line, span = 0.8)
tmp <- sapply(tmp, "length<-", 100)
matlines(tmp, lty = 1, col = cols_m2, lwd = 2)
legend("topright", legend = c("Training", "Validation"),
  fill = unique(cols_m2), bty = "n")
```



Observations:

- The accuracy training accuracy is high while while validation accuracy is lower and has more variablility.
- This could be due to overfitting in some runs.
- However, we can also see runs that reach high validation accuracy above 90% showing that tuning helped find effective hyperparameter combinations from the available sample.
- It can be also observed that the loss curves are highly fluctuating (even after smoothing).
- The gap between training and validation loss curves suggest the model is more uncertain when predicting on unseen data.
- From the plots, Model 1 seems like a better fit compared to Model 2.

Extract runs with validation accuracy>95% and print

The below code can be used to view the top runs (hyperparameter combinations) that achieved the maximum validation accuracy during training.

```
res <- ls_runs(metric_val_accuracy > 0.95,runs_dir = "runs_model2",
               order = metric_val_accuracy)
colu <- c("metric_val_accuracy", grep("flag", colnames(res), value = TRUE),
          "epochs_completed")
res[1:5,colu]
```

```
Data frame: 5 x 7
   metric_val_accuracy flag_dropout flag_lambda flag_lr flag_bs flag_size_1
1               0.9726          0.0       0.005   0.001    81.7         512
2               0.9558          0.3       0.000   0.001    81.7         256
3               0.9547          0.0       0.010   0.001   408.5         512
4               0.9526          0.0       0.010   0.001   408.5         512
NA                  NA           NA          NA      NA      NA          NA
   epochs_completed
1                71
2                96
3               100
4               100
NA               NA
```

Performance evaluation on training and validation data with model using optimal hyperparameters

- The first row of 'res' has the highest metric_val_accuracy of %.
- It also ran for the full 100 epochs, implying that early stopping was not triggered.
- This suggests that the model continued improving, hence we will use the same parameter values in this first combination to fit Model 2 using a slightly higher number of epochs.
- It can be noticed that in this best combination, the dropout rate is 0.

```
# final deployment using optimal hyperparameters at res[1,]
model2_optimal <- keras_model_sequential() %>%
  layer_dense(units = res$flag_size_1[1], input_shape = V, activation = "relu",
              name = "layer_1",
              kernel_regularizer = regularizer_l2(res$flag_lambda[1])) %>%
  layer_dropout(rate = res$flag_dropout[1]) %>%
  layer_dense(units = 128, activation = "relu", name = "layer_2",
              kernel_regularizer = regularizer_l2(res$flag_lambda[1])) %>%
  layer_dropout(rate = res$flag_dropout[1]) %>%
  layer_dense(units = 64, activation = "relu", name = "layer_3",
              kernel_regularizer = regularizer_l2(res$flag_lambda[1])) %>%
  layer_dropout(rate = res$flag_dropout[1]) %>%
  layer_dense(units = ncol(y_train), activation = "softmax", name = "layer_out")%>%
  compile(loss = "categorical_crossentropy", metrics = "accuracy",
          optimizer = optimizer_rmsprop(learning_rate = res$flag_lr[1]),
  )
# training and evaluation
fit_model2_optimal <- model2_optimal %>% fit(
    x = x_train_dnn, y = y_train,
    validation_data = list(x_val_dnn, y_val),
    epochs = 150,
    batch_size = res$flag_bs[1],
    verbose = 0,
    callbacks = callback_early_stopping(monitor = "val_accuracy",
                                        patience = 20))
#count parameters
count_params(model2_optimal)
```

```
[1] 2955667
```

```
#performance on training data
train_metrics_model2 <- model2_optimal%>%evaluate(x_train_dnn,y_train,verbose = 0)
train_metrics_model2
```

```
     loss  accuracy
0.3274313 0.9635734
```

```
# evaluate performance on validation data
val_metrics_model2 <-model2_optimal %>% evaluate(x_val_dnn, y_val, verbose = 0)
val_metrics_model2
```

```
     loss  accuracy
0.4531380 0.9284211
```

Observations:

- The training accuracy reached 96.36%.
- However, the validation accuracy is 92.84% which is lower.
- Validation loss is around 0.4531 compared to training loss of 0.3274.
- There seems to be some over-fitting as the model works well on training but not as much (or close) for validation.
- The total parameter count is quite high (we used more layers) which could contribute to this overfitting. - Overall, the performance is strong, but there is still some room for improvement with better tuning parameters or using other regularization techniques.

Model 3: CNN

- We use a 1D Convolutional Neural Network (CNN) since our input is a 3D tensor, where each sample consists of a 2D feature matrix over time (we do not have other spatial dimensions like an image).
- Hence the kernel size and pooling size will also be 1D.
- The network consist of 2 1D-convolution layers interleaved by 2 1D-max-pooling layers.
- A kernal size of 5 (say like first convolutional layer) means each filter views 5 features at a time (default stride is 1).
- A max pooling with window size 2 has a default stride of 2.
- A convolutional layer helps learn local signal curve features using small filters.
- Pooling prevents overfitting by by reducing spatial size keeping important information.
- Flattening before dense layers converts the final 3D tensor output into a 1D vector to be def into fully connected layers.
- Dropout regularizations (30-40%) have been applied between convolutional layers as well as fully connected layers to prevent overfitting and improve generalization.
- Adam optimizer has been used for this model.

```
# model 3 configuration
model3 <- keras_model_sequential() %>%
  # 1D convolutional layer with 5*1 kernel_size, strides=1 by default
  layer_conv_1d(filters = 128, kernel_size = 5, activation = "relu",
                input_shape = c(125, 45)) %>% # input shape 3D
  # max pooling strides = pool_size = 2
  layer_max_pooling_1d(pool_size = 2) %>%
  # 40% dropout after this layer
  layer_dropout(0.4) %>%
  layer_conv_1d(filters = 128, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_1d(pool_size = 2) %>%
  layer_dropout(0.3) %>%
  layer_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(0.3) %>%
  layer_dense(units = n_classes, activation = "softmax") %>%
  compile(loss = "categorical_crossentropy",metrics = "accuracy",
      optimizer= optimizer_adam()) # adam optimizer

 fit_model3 <-model3 %>%fit(
 x = x_train_cnn,y= y_train,
 validation_data= list(x_val_cnn,y_val),
 epochs = 100,
 batch_size = 72,
 verbose=0
 )
#performance on training data
model3 %>% evaluate(x_train_cnn, y_train, verbose = 0)
```

```
      loss   accuracy
0.01156339 0.99626040
```

```
# evaluate performance on validation data
model3 %>% evaluate(x_val_cnn, y_val, verbose = 0)
```

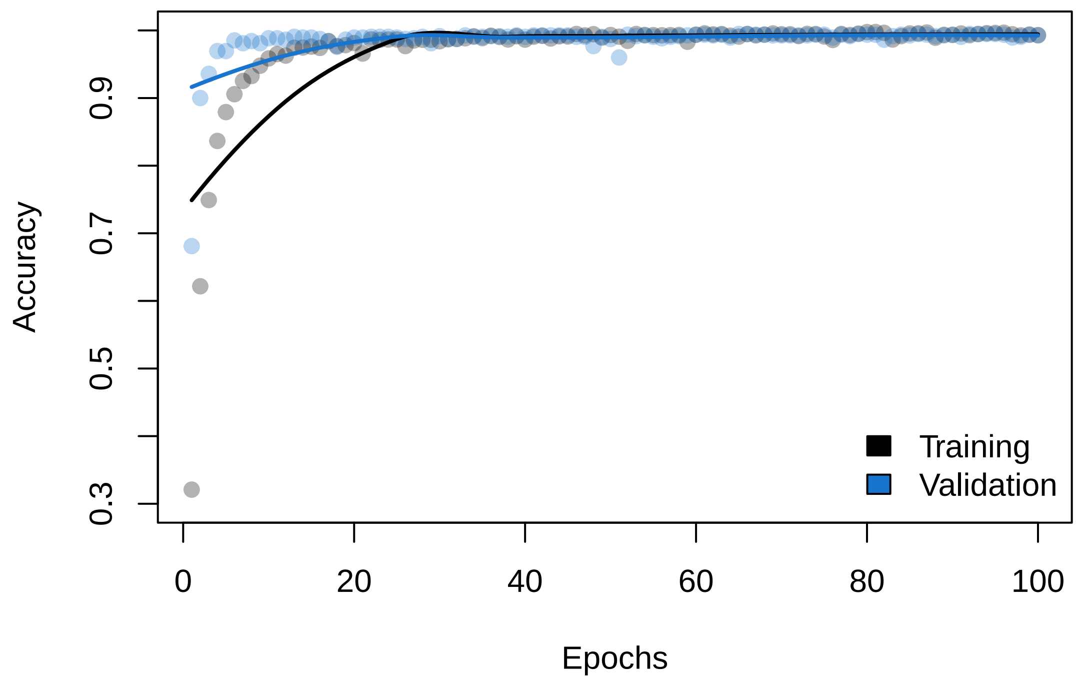
```
      loss   accuracy
0.03033537 0.99263155
```

Plot predictive performance of model on training and validaiton data

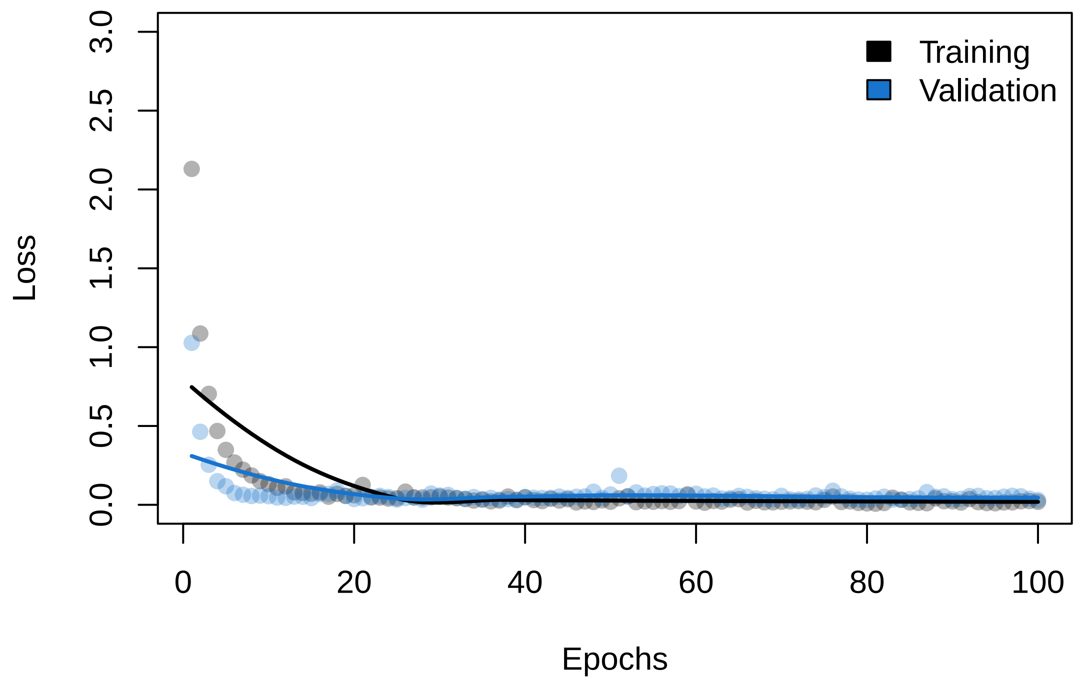
```
# learning curves
out_model3 <-cbind(fit_model3$metrics$loss,fit_model3$metrics$val_loss,
                   fit_model3$metrics$accuracy,fit_model3$metrics$val_accuracy)

# accuracy plot
matplot(out_model3[,3:4], pch = 19, ylab = "Accuracy", xlab = "Epochs",
        col = adjustcolor(cols, 0.3), ylim = c(0.3, 1))
```

```
matlines(apply(out_model3[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend =c("Training","Validation"),fill = cols, bty = "n")
```



```
# loss plot
matplot(out_model3[,1:2],pch= 19, ylab= "Loss",xlab= "Epochs",
        col= adjustcolor(cols, 0.3),ylim= c(0, 3))
matlines(apply(out_model3[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training","Validation"),fill = cols, bty = "n")
```



Observations:

- The training and validation curves almost overlap (for both training and validation).
- This shows that the model has very good generalization and has no overfitting.
- The pooling layers and dropout seem to have helped maintain generalization.
- It can be also seen that both accuracy reached a stable state obtaining high values quickly than other models.
- Both loss curves converge close to zero around 30-40 epochs and has no major fluctuations like before (model is consistent).
- Overall, the CNN model performs well both in terms of accuracy and loss (much better than the DNN models 1 & 2).

Performance evaluation on training and validation data

```
#performance on training data
train_metrics_model3 <- model3 %>% evaluate(x_train_cnn, y_train, verbose = 0)
train_metrics_model3
```

```
      loss   accuracy
0.01156339 0.99626040
```

```
# evaluate performance on validation data
val_metrics_model3 <- model3 %>% evaluate(x_val_cnn, y_val, verbose = 0)
val_metrics_model3
```

```
      loss   accuracy
0.03033537 0.99263155
```

Observations:

- Just as we had seen from plot, we can see the values of accuracies for both training ang validation sets are high for model 3 using CNN (Training accuracy of 99.63% and validation accuracy of 99.26%)
- The losses are also low (training loss of 0.0116 and validation loss of 0.0303)
- Therefore, model 3 generalizes well to unseen data (no overfitting).

Choosing best fit model

Metric	Model 1	Model 2	Model 3
Training Accuracy	99.94%	96.36%	99.63%
Validation Accuracy	97.47%	92.84%	99.26%
Training Loss	0.0018	0.3274	0.0116
Validation Loss	0.2431	0.4531	0.0303

Observations:

- As discussed before, clearly, **Model 3 using CNN is the best model** among the 3.
- Model 3 balances both training and validation metrics.
- Although training loss is low for Model 1, we can see that Model 3 (CNN) gives the best predictive performance on validation data in terms of both accuracy and loss.
- This could be due to CNN's ability to preserve spatial structure and learn local patterns more effectively than a DNN.
- Max-pooling helps to keep the important features and also reduce complexity.

Best Model

The best model found based on performance on validation data is **Model 3 (CNN)**.

Retrain best model using full training data (train+val)

```
model3_final <- keras_model_sequential() %>%
  layer_conv_1d(filters = 128, kernel_size = 5, activation = "relu",
                input_shape = c(125, 45)) %>%
  layer_max_pooling_1d(pool_size = 2) %>%
  layer_dropout(0.4) %>%
  layer_conv_1d(filters = 128, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_1d(pool_size = 2) %>%
  layer_dropout(0.3) %>%
  layer_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(0.3) %>%
  layer_dense(units = n_classes, activation = "softmax") %>%
  compile(loss = "categorical_crossentropy",metrics = "accuracy",
      optimizer= optimizer_adam()) # adam optimizer

 fit_model3_final <-model3_final %>%fit(
 x = x,y = y,
 epochs = 100,
 batch_size = 72,
 verbose=0
 )
```

Test performance overall

```
set.seed(24214925)
# overall test accuracy and loss
final_train_metrics <- model3_final %>%evaluate(x_test,y_test, verbose= 0)
final_train_metrics
```

```
      loss   accuracy
0.03109396 0.99368423
```

Observation:

- Very good test performance with 99.37% test accuracy and 0.0311 overall test loss.
- CNN model has good ability to distinguish between the 19 physical activities on unseen test data.
- Hence, it can generalize well.

Class-wise performance

```
# find class with the highest predicted probability
test_hat <-model3_final %>% predict(x_test, verbose=0)%>% max.col()
tab <-table(max.col(y_test),test_hat) # confusion matrix

# class-accuracy/sensitivity = TP / Total in class
cl.acc <- diag(tab) / rowSums(tab)
# Precision = TP / (TP + FP)
precision <- diag(tab) / colSums(tab)

# print confusion matrix with results
cbind(tab, Acc=round(cl.acc,2), Prec=round(precision,2))
```

```
    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19  Acc Prec
1  50  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 1.00 1.00
2   0 50  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 1.00 1.00
3   0  0 50  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 1.00 1.00
4   0  0  0 50  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 1.00 1.00
5   0  0  0  0 50  0  0  0  0  0  0  0  0  0  0  0  0  0  0 1.00 1.00
6   0  0  0  0  0 50  0  0  0  0  0  0  0  0  0  0  0  0  0 1.00 0.96
7   0  0  0  0  0  0 50  0  0  0  0  0  0  0  0  0  0  0  0 1.00 1.00
8   0  0  0  0  0  0  0 50  0  0  0  0  0  0  0  0  0  0  0 1.00 1.00
9   0  0  0  0  0  0  0  0 50  0  0  0  0  0  0  0  0  0  0 1.00 1.00
10  0  0  0  0  0  2  0  0  0 44  0  0  1  3  0  0  0  0  0 0.88 1.00
11  0  0  0  0  0  0  0  0  0  0 50  0  0  0  0  0  0  0  0 1.00 1.00
12  0  0  0  0  0  0  0  0  0  0  0 50  0  0  0  0  0  0  0 1.00 1.00
13  0  0  0  0  0  0  0  0  0  0  0  0 50  0  0  0  0  0  0 1.00 0.98
14  0  0  0  0  0  0  0  0  0  0  0  0  0 50  0  0  0  0  0 1.00 0.94
15  0  0  0  0  0  0  0  0  0  0  0  0  0  0 50  0  0  0  0 1.00 1.00
16  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 50  0  0  0 1.00 1.00
17  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 50  0  0 1.00 1.00
18  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 50  0 1.00 1.00
19  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 50 1.00 1.00
```

```
# print results using labels
final_class_metrics <- data.frame(Class_No = 1:length(class_labels),Class_Label = class_labels,
    Accuracy = round(cl.acc,2),Precision = round(precision,2))
print(final_class_metrics, row.names = FALSE)
```

```
 Class_No        Class_Label Accuracy Precision
        1         asc_stairs     1.00      1.00
        2         basketball     1.00      1.00
        3      cross_trainer     1.00      1.00
        4      cycling_horiz     1.00      1.00
        5       cycling_vert     1.00      1.00
        6        desc_stairs     1.00      0.96
        7            jumping     1.00      1.00
        8         lying_back     1.00      1.00
        9         lying_side     1.00      1.00
       10    moving_elevator     0.88      1.00
       11             rowing     1.00      1.00
       12  running_treadmill     1.00      1.00
       13            sitting     1.00      0.98
       14     stand_elevator     1.00      0.94
       15           standing     1.00      1.00
       16            stepper     1.00      1.00
       17            walking     1.00      1.00
       18       walking_tread     1.00      1.00
       19 walking_tread_incl     1.00      1.00
```

Observations:

- The final model using CNN shows excellent classification with 100% accuracy and precision for most activities.

- The high class-wise accuracy (sensitivity) shows that the model has learned to distinguish between the majority of activities effectively and is able to generalize well on unseen data.
- The high precision for most classes shows that when model predicted a certain activity it was usually correct.
- There are only 4 classes where either accuracy or precision is less than 1 with very few misclassifications as shown in confusion matrix.