# STAT40970 Machine Learning and AI End of Term Assignment

Fanahan McSweeney - 20203868

April 27, 2021

## Question 6. Data Analysis Task

For this data analysis task, we need to build a suitable predictive model to classify the activity being carried out by a subject given a variety of sensor readings recorded while the activity is taking place. For each subject activity sample, we are given data from 125 sampling instants for 45 separate sensors. In total there are 19 different forms of activities that we need to classify.

The given training set has 1900 separate subject activity samples, so the resulting matrix has dimensions 1900 x 125 x 45. This set will be used both for training and validation. A separate test set has also been provided, which has 7220 separate subject activity samples in total, resulting in a matrix of dimensions 7220 x 125 x 45.

We will have to use the training set containing the 1900 samples for both training and validation purposes when fitting the models, and the 7220 samples in the test set will be used to evaluate the final predictive performance of the best model. I have chosen to split the training and validation data 70:30, so a randomly sampled 70% of the given training data set will be used for training, and the remaining 30% will be used for validation purposes.

```r
library("keras"); library("knitr");library("kableExtra")

# Load the data activity RData file
load("Data/data_activity_recognition.RData")

# split x any y into training and validation sets
set.seed(20203868)
# randomly select 30% of the rows from the original data set to use for validation
N <- nrow(x)
val <- sample(1:N, N*0.3)
# create x and y validation sets
x_validation <- x[val,,]
y_validation <- y[val]

# select remaining data for training x and y sets
train <- setdiff(1:N, val)
x_train <- x[train,,]
y_train <- y[train]

# set up training, validation and testing class vectors as binary class matrices
# (convert y vectors to factors, then convert to numeric vectors before one-hot encoding)
# NOTE: subtract 1 after converting factor to numeric as to_categorical expects index of first class to
y_train <- to_categorical(as.numeric(as.factor(y_train))-1)
```

```r
y_validation <- to_categorical(as.numeric(as.factor(y_validation))-1)
y_test <- to_categorical(as.numeric(as.factor(y_test))-1)
```

## (a) Deploy a predictive model which can be employed for daily/sports activity recognition from movement sensor data.

We now need to build a suitable predictive model which can be used to predict the activities carried out by the subjects. There are a number of methods that could be used for the given data set:

1. *Feed-forward Multilayer Neural Networks* - A regular feed-forward neural network could be employed, provided the given input data for each sample (which will be in the form of a 125 x 45 matrix) was converted to a vector. However, this model would assume independence between all input data (ignoring that the sampled data for each of the sensors are not all independent).
2. *Convolutional Neural Networks (CNN)* - 1D convolutional layers could also be employed in the model. These would provide an additional benefit over standard feed forward layers by giving the model the ability to detect structural patterns in the data. As each of the 45 sensors have an equal number of recorded sampling instances (125), these effectively form 45 separate time series where different samples from the same sensor will not be completely independent. Adding convolutional layers could therefore improve the model's performance by identifying structural information in these sampled time series for each sensor.
3. *Recurrent Neural Networks (RNN)* - RNNs are capable of interpreting the sequential information in the given input data, which again is very suitable for this problem as the sampling data recorded for each sensor is effectively a sequence of time-dependent data.

I decided to construct a number of different models using each of the methods above to determine which is most suitable for this problem. As there are an infinite number of network configurations that we could build for each model type, I chose to limit each model to the following restrictions:

- Adam optimiser with the default learning rate of 0.001 will be used in each model, as it is generally found to perform well in a variety of different classification tasks.
- The number of layers for any particular layer type in a model will be limited to 3, as I found that models with larger numbers of layers and complexity didn't provide any substantial improvement to the model's performance.
- Rectified Linear Unit (ReLU) activation function will be used on all feed-forward and convolutional layers.
- Models will be trained initially using 50 epochs, with a batch size equal to 1% of the number of samples in the training data.

## (b) Discuss and motivate the various decisions taken in all stages of the model building process. Try to use a range of different model configurations and evaluate and discuss their relative merits.

### Model 1

Firstly, I decided to employ a standard feed-forward neural network with multiple layers. The configuration of this model can be summarised as follows:

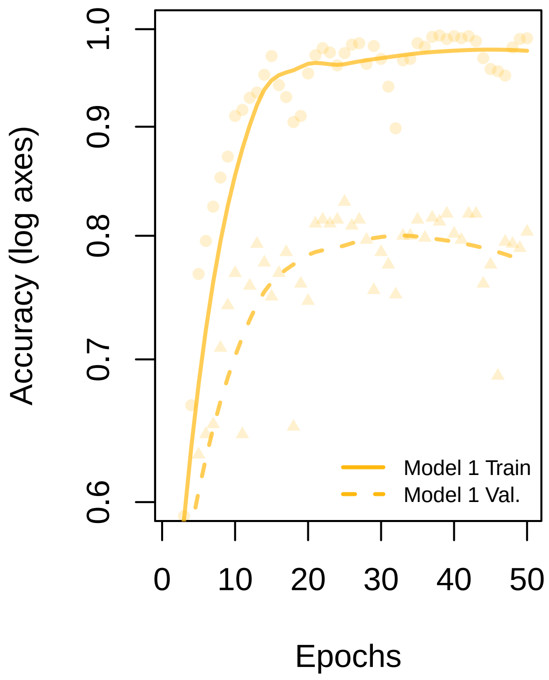
- Flattening Layer
  - Converts inputs of dimensions 125 x 45 to vectors of length 5625.
- Fully Connected Layers
  - Hidden layer, 64 units, ReLU activation
  - Hidden layer, 32 units, ReLU activation
  - Hidden layer, 16 units, ReLU activation
  - Output layer, 19 units, softmax activation

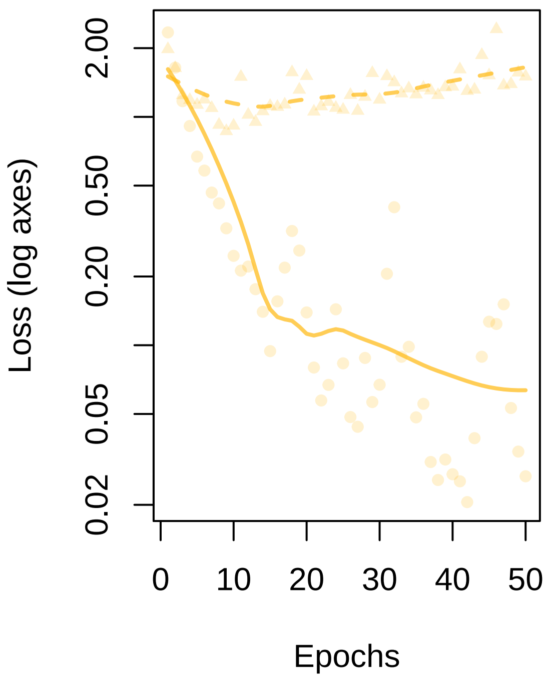
I chose to use hidden layers with decreasing numbers of units as this typically performs well, and I found that using 64, 32 and 16 units respectively in each of the 3 hidden layers provided the best performance for the given model out of a number of configurations I trialled. Below, the model is defined and is then fitted using the training data, with the validation data used to evaluate the model after each training iteration.

```r
# model 1 - Feed-forward neural network
model1 <- keras_model_sequential() %>%
  # flattening layer to flatten the input data matrices to create vectors
  layer_flatten(input_shape=c(125, 45)) %>%
  # fully connected layers
  layer_dense(units=64, activation="relu") %>%
  layer_dense(units=32, activation="relu") %>%
  layer_dense(units=16, activation="relu") %>%
  layer_dense(units=19, activation="softmax") %>%
  # compile
  compile(
    loss="categorical_crossentropy",
    metrics="accuracy",
    optimizer=optimizer_adam()
  )

# model training
fit1 <- model1 %>% fit(
  x=x_train, y=y_train,
  validation_data=list(x_validation, y_validation),
  epochs=50,
  batch_size=nrow(x_train)*0.01
)
```

After training the model, we can look at its training and validation accuracy and loss curves to evaluate its performance. The accuracy curves are shown in the first plot below, and the loss curves are shown in the second plot. Note that y-axes of both plots are shown on a log scale.

```r
# function to plot accuracy and loss plots for the specified models
# NOTE: fit_list must be of the form 'cbind(fit1, fit2, ...)' where fit1,fit2 are the trained models
acc_loss_plot <- function(fit_list, acc_ylim=NULL, first_model_int=1) {
  # define vector of colours
  cols <- c("darkgoldenrod1","dodgerblue3","red", "chartreuse3")
  cols <- rep(cols[1:(length(fit_list)/2)], each=2)

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

  # create empty vectors/matrices
  out_acc <- NULL
  out_loss <- NULL
  legend_list <- NULL

  # loop through all model fit objects in the input list
  for(i in 1:(length(fit_list)/2)) {
    # add accuracy and loss values for current model to the relevant matrix
    out_acc <- cbind(out_acc, fit_list[,i]$metrics$accuracy, fit_list[,i]$metrics$val_accuracy)
    out_loss <- cbind(out_loss, fit_list[,i]$metrics$loss, fit_list[,i]$metrics$val_loss)
    # add legend labels for current model to legend label vector
    legend_list <- c(legend_list, paste0("Model ", (first_model_int+i-1), " Train"), paste0("Model ", (
  }

  # set plot window to 1 row by 2 cols
  par(mfrow=c(1,2))

  # add accuracy data points and smoothed lines for all models (train and val.)
  matplot(out_acc, pch=c(19,17), ylab="Accuracy (log axes)", xlab="Epochs",
          cex=0.7, col=adjustcolor(cols, 0.2), log="y", ylim=acc_ylim)
  matlines(apply(out_acc, 2, smooth_line), lty=c(1,2), col=adjustcolor(cols,0.7), lwd=2)

  # add legend to first plot
  legend("bottomright", legend=legend_list, col=cols, lwd=2, lty=c(1,2), bty="n", cex=0.7)

  # add loss data points and smoothed lines for all models (train and val.)
  matplot(out_loss, pch=c(19,17), ylab ="Loss (log axes)",xlab ="Epochs",
          cex=0.7, col=adjustcolor(cols, 0.2), log ="y")
  matlines(apply(out_loss, 2, smooth_line), lty=c(1,2), col=adjustcolor(cols,0.7), lwd=2)
}

# plot accuracy and loss curves for model 1
acc_loss_plot(cbind(fit1), acc_ylim = c(0.6,1))
```





The model appears to perform reasonably well, obtaining a final validation accuracy of approximately 0.8. However, there is a significant gap between the training and validation curves indicating that a level of overfitting is taking place. This can also be seen when looking at the loss curves, as the gap between validation and training loss increases with increasing numbers of epochs, and the loss curve also appears to increase with the number of epochs.

I tried to reduce the effect of overfitting in the this model by adding different forms of regularisation at the different layers such as weight decay and dropout, but I found that adding any level of regularisation to the model resulted in the reduction of the model's overall capacity during training. So while it was possible to reduce the effect of overfitting for this model, I couldn't find a reasonable trade off between reduction in overfitting and reduction in overall model capacity that lead to an improvement in the performance of the final trained model.

**Model 2**

The next model I decided to employ was a convolutional neural network. The configuration of this model can be summarised as follows:

- Convolution Layers
  - Convolution layer, 64 filters, kernel size of 5, ReLU activation
  - Maximum Pooling layer, 2x2
  - Convolution layer, 128 filters, kernel size of 25 with a stride of 5, ReLU activation
  - Maximum Pooling layer, 2x2
- Fully Connected Layers
  - Hidden layer, 64 units, ReLU activation
  - Output layer, 19 units, softmax activation

I chose to use a relatively small kernel size of 5 (i.e. the size of the convolution window) for the first convolutional layer in this model to collect information relating to sampling instances that are very close to each other, as I expected that these would be significantly more highly correlated that sampling instances further away from each other. However, I found that using a much larger kernel size of 25 in the second convolutional layer with a stride of 5 improved the model's ability to classify data from a particular class ("moving_elevator").

When looking at the classification accuracies of the validation data for each individual class while testing a number of variations of this model, I found that similar models with a relatively small kernel size on the second convolutional layer typically resulted in a relatively poor classification accuracy of the "moving_elevator" class in comparison to the remaining classes, with a somewhat significant number of these samples misclassified as "standing_elevator". Increasing the kernel size and stride appeared to lead to an improvement in the classification of this particular class, while also performing well at classifying samples from the remaining classes.

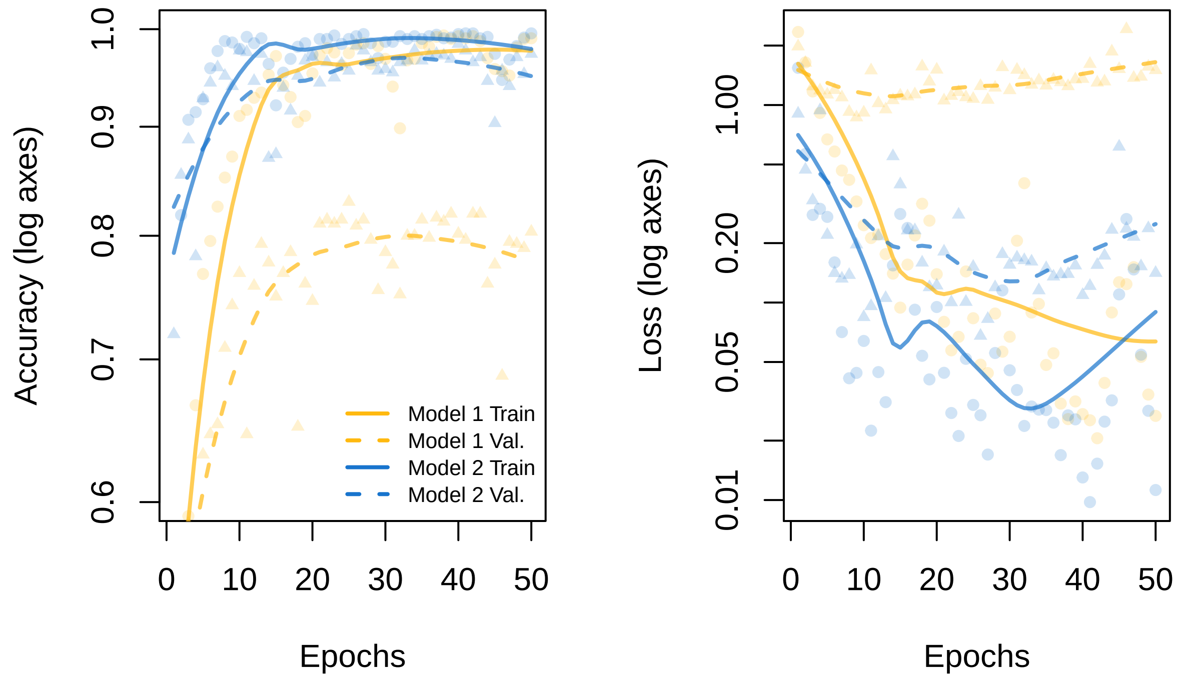
I also chose to increase the number of filters while progressing through the convolution layers to assist with learning greater levels of abstraction, and to decrease the number of hidden units while progressing through the fully connected layers. Below, the model is defined and is then fitted using the training data, with the validation data used to evaluate the model after each training iteration.

```r
# model 2 - CNN
model2 <- keras_model_sequential() %>%
  # convolutional layers
  layer_conv_1d(filters=64, kernel_size=5, activation="relu",
                input_shape=c(125, 45)) %>%
  layer_max_pooling_1d(pool_size=2) %>%
  layer_conv_1d(filters=128, kernel_size=25, strides = 5, activation="relu") %>%
  layer_max_pooling_1d(pool_size=2) %>%
  # fully connected layers
  layer_flatten() %>%
  layer_dense(units=64, activation="relu") %>%
  layer_dense(units=19, activation="softmax") %>%
  # compile
  compile(
    loss="categorical_crossentropy",
    metrics="accuracy",
    optimizer=optimizer_adam()
  )
```

```
# model training
fit2 <- model2 %>% fit(
  x=x_train, y=y_train,
  validation_data=list(x_validation, y_validation),
  epochs=50,
  batch_size=nrow(x_train)*0.01
)
```

After training this model, we can look at its training and validation accuracy and loss curves to evaluate its performance. I have recreated the accuracy and loss plots for model 1 below, but I have now added the curves from model 2 to the plots so we can compare the models.

```
# plot accuracy and loss curves for model 1 and 2
acc_loss_plot(cbind(fit1, fit2), acc_ylim = c(0.6,1))
```



The model appears to perform very well, and significantly better than model 1, obtaining a final validation accuracy of approximately 0.95. There is a small gap between the training and validation curves indicating that some level of overfitting is taking place, but it is far less significant than what we observed with model 1. A gap is also visible between validation and training loss, but the validation loss appears to still be decreasing after 50 epochs of training so it is likely that training the model for longer would improve it further.

**Model 3**

The next model I decided to employ was the same convolutional neural network as model 2 but with some regularisation added, to try and reduce the relatively small level of overfitting occurring. The configuration of this model can be summarised as follows:

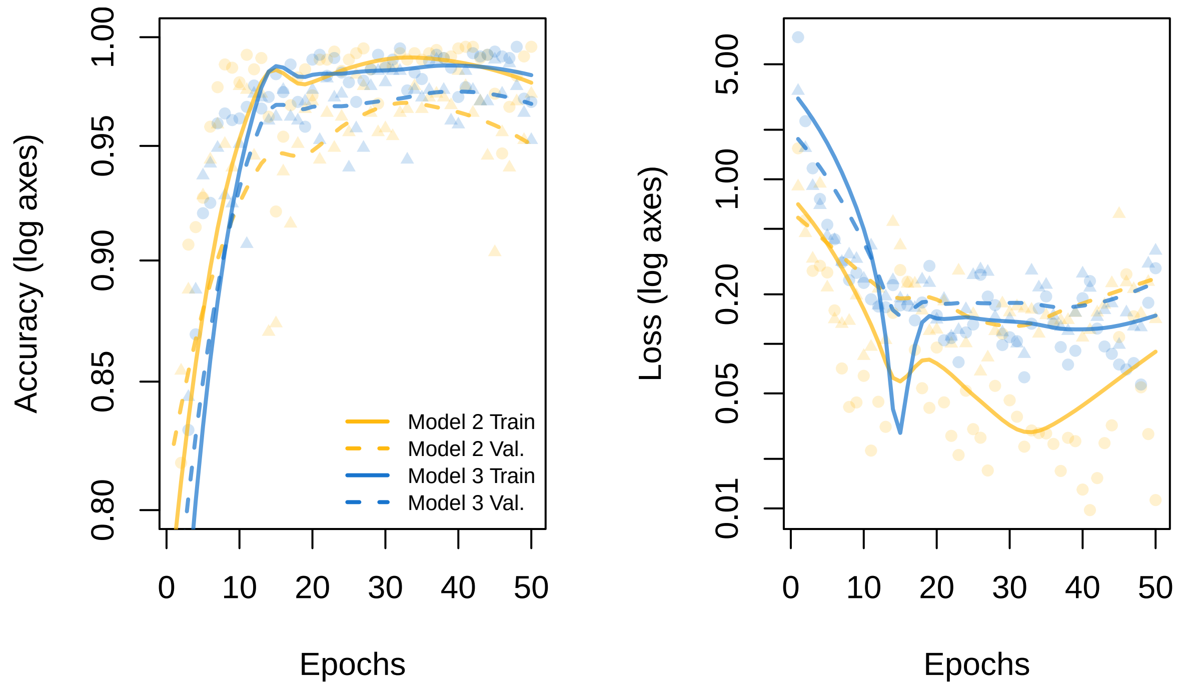
- Convolution Layers
  - Convolution layer, 64 filters, kernel size of 5, ReLU activation
  - Maximum Pooling layer, 2x2
  - Convolution layer, 128 filters, kernel size of 25 with a stride of 5, ReLU activation
  - Maximum Pooling layer, 2x2
- Fully Connected Layers
  - Hidden layer, 64 units, ReLU activation, L2 regularisation with factor of 0.1, dropout with rate of 0.1
  - Output layer, 19 units, softmax activation

Aside from the model architecture decisions made for model 2 (which I justified earlier) I decided to add a very small level or regularization to the model's fully connected layer to see if it could improve its performance by reducing overfitting without significantly reducing the model's overall capacity. This included the addition of L2 "weight decay" regularisation to this layer with a decay factor of 0.1, and the addition of dropout to the layer with a dropout rate of 0.1.

```r
# model 3 - CNN with regularization
model3 <- keras_model_sequential() %>%
  #
  # convolutional layers
  layer_conv_1d(filters=64, kernel_size=5, activation="relu",
                input_shape=c(125, 45)) %>%
  layer_max_pooling_1d(pool_size=2) %>%
  layer_conv_1d(filters=128, kernel_size=25, strides = 5, activation="relu") %>%
  layer_max_pooling_1d(pool_size=2) %>%
  #
  # fully connected layers
  layer_flatten() %>%
  layer_dense(units=64, activation="relu", kernel_regularizer=regularizer_l2(0.1)) %>%
  layer_dropout(0.1) %>%
  layer_dense(units=19, activation="softmax") %>%
  #
  # compile
  compile(
    loss="categorical_crossentropy",
    metrics="accuracy",
    optimizer=optimizer_adam()
  )

# model training
fit3 <- model3 %>% fit(
  x=x_train, y=y_train,
  validation_data=list(x_validation, y_validation),
  epochs=50,
  batch_size=nrow(x_train)*0.01
)
```

After training this model, we can look at its training and validation accuracy and loss curves to evaluate its performance. I have included the accuracy and loss curves for model 2 and model 3 on the plots below, but I have omitted model 1 and is has already been shown to be inferior to model 2.

```r
# plot accuracy and loss curves for model 2 and 3
acc_loss_plot(cbind(fit2, fit3), acc_ylim = c(0.8, 1), first_model_int = 2)
```



Model 3 appears to be an improvement on model 2, obtaining a final validation accuracy of approximately 0.97. Regularization has reduced the gap between the training and validation curves indicating a reduction in overfitting, and the gap between validation and training loss curves has also been reduced.

### Model 4

The next model I decided to employ was a recurrent neural network using LSTM (Long Short-Term Memory) layers. The configuration of this model can be summarised as follows:

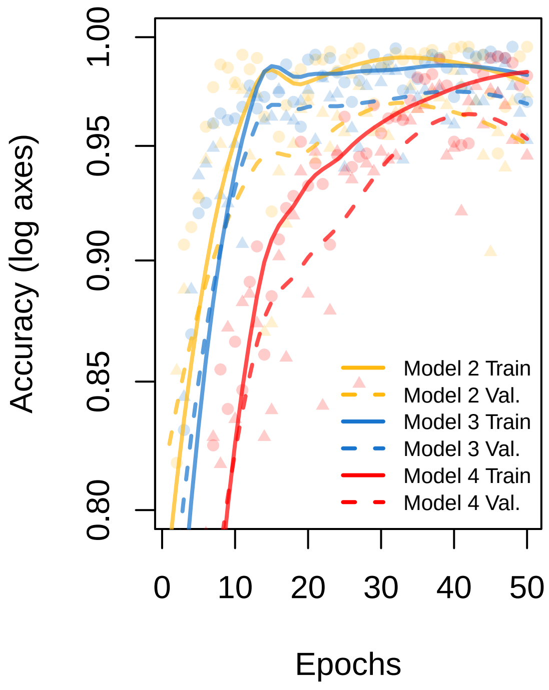
- Recurrent Layers
  - LSTM layer, *return_sequences* parameter is True (as this is not the final LTSM layer), 32 hidden units
  - LSTM layer, 32 hidden units
- Fully Connected Layers
  - Hidden layer, 64 units, ReLU activation
  - Output layer, 19 units, softmax activation

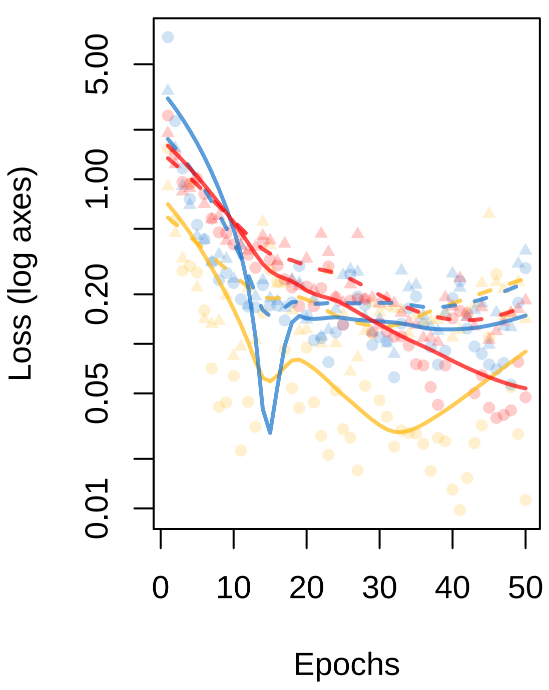
I chose to use 2 LSTM layers as I found that it resulted in the best trade-off between model complexity and performance while testing multiple similar RNN model configurations, and I found that using 32 units in both layers resulted in a satisfactory model performance. I also kept the same configuration for the fully connected layers as the previous models.

```r
# model 4 - RNN
model4 <- keras_model_sequential() %>%
  # recurrent layers
  layer_lstm(units=32, input_shape = c(125, 45), return_sequences = TRUE) %>%
  layer_lstm(units=32) %>%
  # fully connected layers
  layer_dense(units=64, activation = "relu") %>%
  layer_dense(units=19, activation ="softmax") %>%
  # compile
  compile(
    optimizer=optimizer_adam(),
    loss ="categorical_crossentropy",
    metrics ="accuracy"
  )

# model training
fit4 <- model4 %>% fit(
  x_train, y_train,
  epochs=50,
  batch_size=nrow(x_train)*0.01,
  validation_data=list(x_validation, y_validation)
)
```

After training this model, we can look at its training and validation accuracy and loss curves to evaluate its performance. I have included the accuracy and loss curves for models 2, 3 and 4 on the plots below for comparison purposes.

```r
# plot accuracy and loss curves for model 2 to 4
acc_loss_plot(cbind(fit2, fit3, fit4), acc_ylim = c(0.8, 1), first_model_int = 2)
```





Overall, the RNN model performs well achieving a relatively high validation accuracy, but model 3 (CNN with regularization) still achieved the highest final validation accuracy and appears to outperform it.

**Model 5**

The final model I decided to employ was a variant to the recurrent neural network created above but with some level of regularization added. The configuration of this model can be summarised as follows:

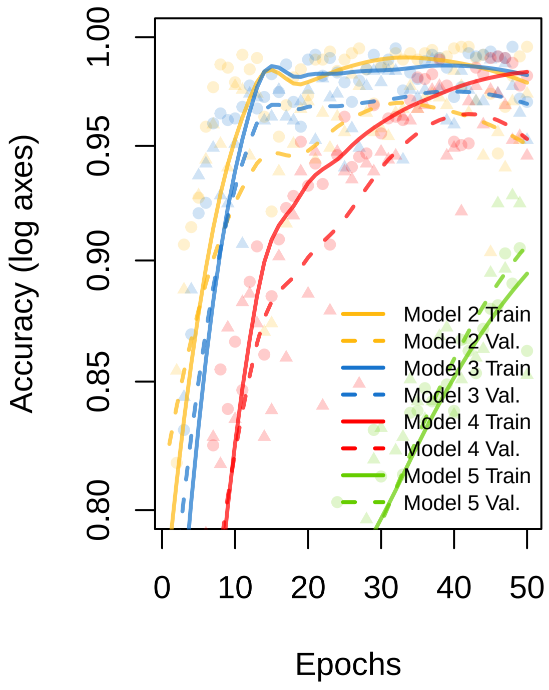
- Recurrent Layers
  - LSTM layer, *return_sequences* parameter is True (as this is not the final LTSM layer), 32 hidden units, recurrent dropout with rate of 0.1
  - LSTM layer, 32 hidden units, recurrent dropout with rate of 0.1
- Fully Connected Layers
  - Hidden layer, 64 units, ReLU activation, L2 regularisation with factor of 0.1, dropout with rate of 0.1
  - Output layer, 19 units, softmax activation

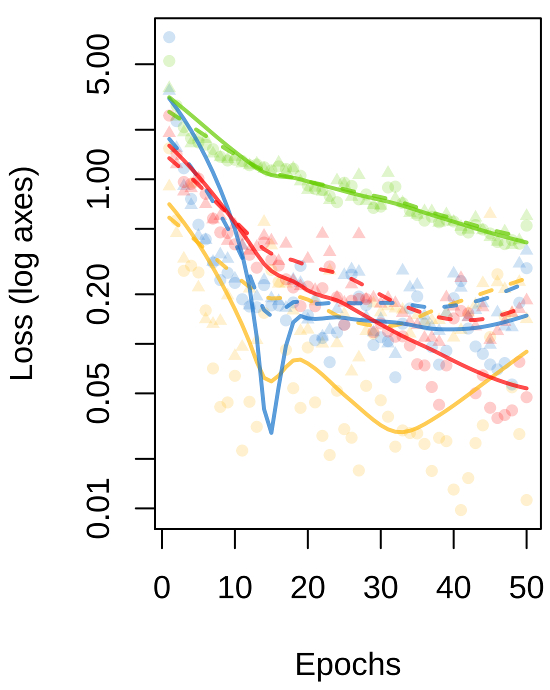
I added the same (relatively weak) regularization to the fully connected layers as I used in model 3 as there only appears to be a small level of overfitting occurring based on the relatively small gap between the accuracy and loss curves for the training and validation data. I also included recurrent dropout regularization in both of the LSTM layers due to the time dependent nature of these layers, as it applies the same dropout mask at every time point.

```r
# model 5 - RNN with regularization
# deploy model 4 and compile
model5 <- keras_model_sequential() %>%
  # recurrent layers
  layer_lstm(units=32, input_shape=c(125, 45), return_sequences=TRUE, recurrent_dropout=0.1) %>%
  layer_lstm(units=32, recurrent_dropout=0.1) %>%
  # fully connected layers
  layer_dense(units=64, activation="relu", kernel_regularizer=regularizer_l2(0.1)) %>%
  layer_dropout(0.1) %>%
  layer_dense(units=19, activation="softmax") %>%
  # compile
  compile(
    optimizer=optimizer_adam(),
    loss="categorical_crossentropy",
    metrics="accuracy"
  )

# model training
fit5 <- model5 %>% fit(
  x_train, y_train,
  epochs=50,
  batch_size=nrow(x_train)*0.01,
  validation_data=list(x_validation, y_validation)
)
```

After training this model, again we can look at its training and validation accuracy and loss curves to evaluate its performance. I have included the accuracy and loss curves for models 2 to 5 on the plots below for comparison purposes.

```r
# plot accuracy and loss curves for model 2 to 5
acc_loss_plot(cbind(fit2, fit3, fit4, fit5), acc_ylim = c(0.8, 1), first_model_int = 2)
```





While the added regularization has reduced the gap between the training and validation accuracy and loss curves in comparison to the previous RNN model (model 4), it doesn't result in an improvement to the model performance. The validation accuracy for model 5 still appears to be increasing significantly after 50 epochs, and likewise the loss appears to be decreasing steadily, so it's possible that this model could achieve a much better performance if trained over a longer period.

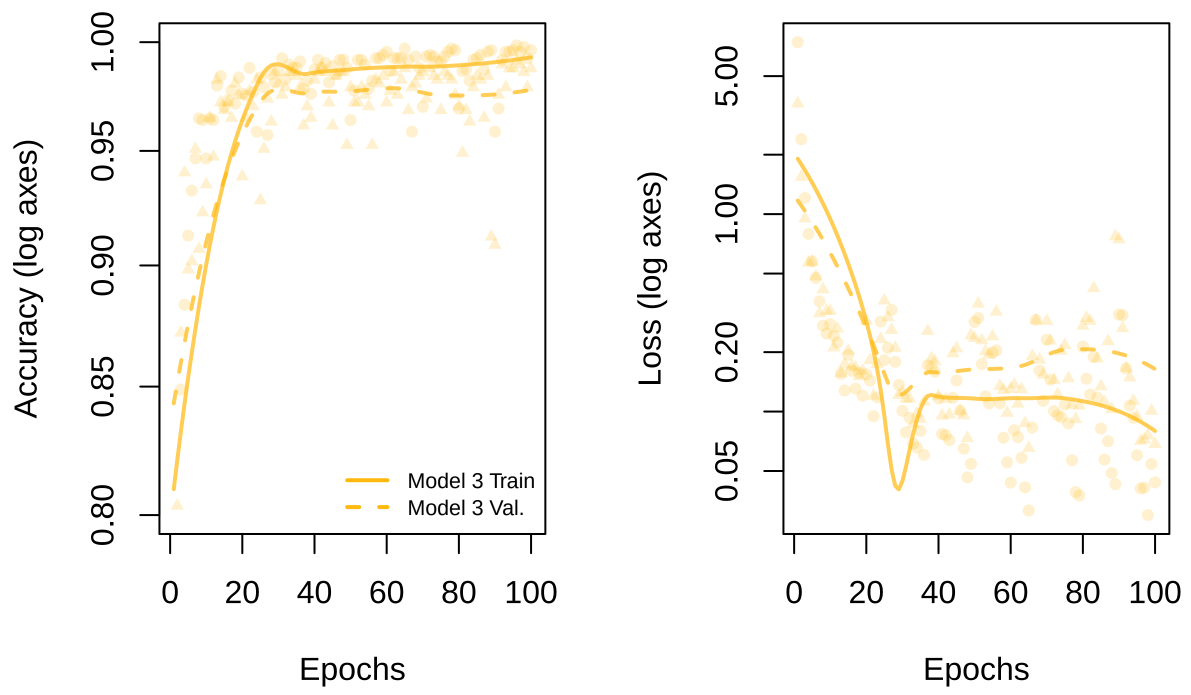
Overall I believe that model 3 - a convolutional neural network with some regularization introduced - is the most promising model of the model configurations that I trialled, based off the evidence above. While RNN models also appear to be very effective for this classification task, the models employed here also take significantly longer to train than the CNN models tested, which further adds to my conclusion that model 3 is in fact the best model.

## (c) Use the test data to evaluate the predictive performance of the best model. Comment on the ability of the model at recognizing the different activities.

We can now evaluate the predictive performance of the best model using the unseen test data set.

Having chosen the model 3 configuration as the best of the trialled configurations, I have decided firstly to retrain the model but for a longer period (100 epochs) as I believe it hadn't yet reached it's full predictive capacity after 50 epochs. After retraining the model, I have again plotted its accuracy and loss curves.

```r
# model 3 - CNN with regularization
model3 <- keras_model_sequential() %>%
  #
  # convolutional layers
  layer_conv_1d(filters=64, kernel_size=5, activation="relu",
                input_shape=c(125, 45)) %>%
  layer_max_pooling_1d(pool_size=2) %>%
  layer_conv_1d(filters=128, kernel_size=25, strides = 5, activation="relu") %>%
  layer_max_pooling_1d(pool_size=2) %>%
  #
  # fully connected layers
  layer_flatten() %>%
  layer_dense(units=64, activation="relu", kernel_regularizer=regularizer_l2(0.1)) %>%
  layer_dropout(0.1) %>%
  layer_dense(units=19, activation="softmax") %>%
  #
  # compile
  compile(
    loss="categorical_crossentropy",
    metrics="accuracy",
    optimizer=optimizer_adam()
  )

# model training
fit3 <- model3 %>% fit(
  x=x_train, y=y_train,
  validation_data=list(x_validation, y_validation),
  epochs=100,
  batch_size=nrow(x_train)*0.01
)

# plot accuracy and loss curves for the final model
acc_loss_plot(cbind(fit3), acc_ylim = c(0.8, 1), first_model_int = 3)
```



We can now evaluate the performance of the fully trained model on the test data. The overall loss and accuracy values achieved by the model on the test set are shown below.

```
# get loss and accuracy for test data
model3 %>% evaluate(x_test, y_test, verbose=0)
```

```
##       loss   accuracy
## 0.07785394 0.98310250
```

To assess the models performance at predicting the test data classes for each individual activity, we can create a confusion matrix table comparing the number the actual classes of each test sample to the class predicted by the model. Due to the relatively large number of classes in this data set, I have split the table into 2 separate tables so all rows and columns are visible.

Based off the table below, the model shows a very good predictive performance when classifying the unseen test data, with a classification accuracy of greater than 90% for each individual activity except for the "moving_elevator" activity. The model still achieves a reasonably high classification accuracy for this activity considering the large number of possible classes in the model, and the most common activity that it is misclassified as is "standing_elevator" which is of course an extremely similar activity, so we would expect the sensor data from both of these activities to be very similar which would increase the difficulty in classifying them correctly.

```
# create vector of the class labels
class_labels <- levels(as.factor(y))
# get actual classes for the test data
class_y <- class_labels[max.col(y_test)]
# get predicted classes for the test data
class_hat <- class_labels[(model3 %>% predict_classes(x_test))+1]
# create table comparing actual to predicted classes
tab <- table(class_y, class_hat)

# add column to table showing class accuracy for each individual class
```

```r
tab_classacc <- cbind(tab, class.accuracy=round(diag(tab)/rowSums(tab),3))

# print table of actual vs predicted classes and class accuracy for each class (first 10 columns)
add_header_above(
  kable_styling(
    kable(tab_classacc[,1:10],
          align = 'c'),
    font_size = 4.5),
  c("Actual"=1, "Predicted"=10))
```

| Actual | Predicted | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | asc_stairs | basketball | cross_trainer | cycling_horiz | cycling_vert | desc_stairs | jumping | lying_back | lying_side | moving_elevator |
| asc_stairs | 375 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 |
| basketball | 0 | 360 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 14 |
| cross_trainer | 0 | 0 | 379 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| cycling_horiz | 0 | 0 | 0 | 380 | 0 | 0 | 0 | 0 | 0 | 0 |
| cycling_vert | 1 | 0 | 0 | 0 | 379 | 0 | 0 | 0 | 0 | 0 |
| desc_stairs | 0 | 0 | 0 | 0 | 0 | 377 | 0 | 0 | 0 | 3 |
| jumping | 0 | 0 | 0 | 0 | 0 | 0 | 380 | 0 | 0 | 0 |
| lying_back | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 380 | 0 | 0 |
| lying_side | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 380 | 0 |
| moving_elevator | 0 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 321 |
| rowing | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| running_treadmill | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sitting | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| stand_elevator | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 17 |
| standing | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| stepper | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| walking | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| walking_tread | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| walking_tread_incl | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

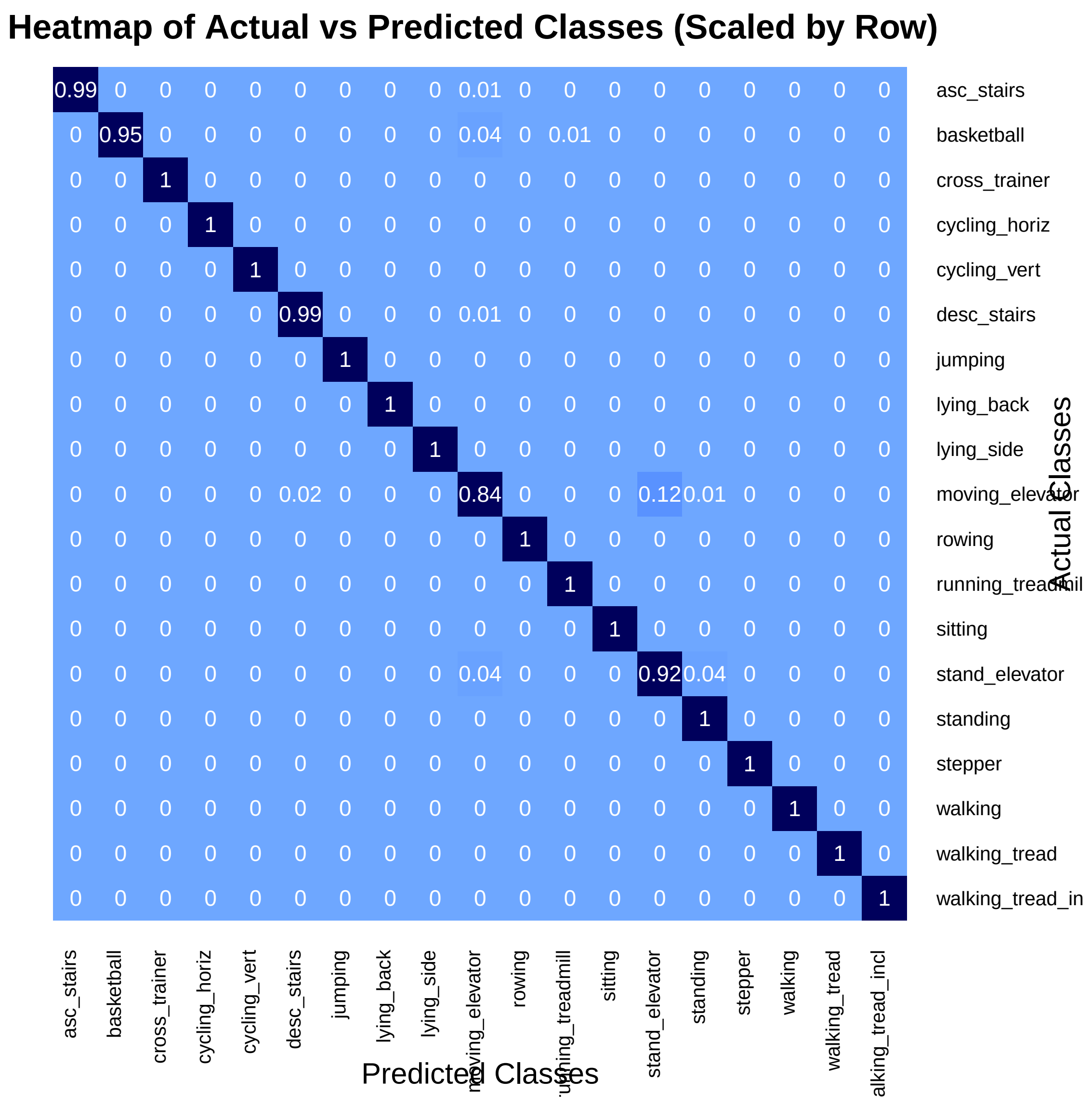
```r
# print table of actual vs predicted classes and class accuracy for each class (last 10 columns)
add_header_above(
  kable_styling(
    kable(tab_classacc[,11:20],
          align = 'c'),
    font_size = 4.5),
  c("Actual"=1, "Predicted"=9, " "=1))
```

| Actual | Predicted | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | rowing | running_treadmill | sitting | stand_elevator | standing | stepper | walking | walking_tread | walking_tread_incl | class.accuracy |
| asc_stairs | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.987 |
| basketball | 0 | 3 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0.947 |
| cross_trainer | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.997 |
| cycling_horiz | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.000 |
| cycling_vert | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.997 |
| desc_stairs | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.992 |
| jumping | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.000 |
| lying_back | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.000 |
| lying_side | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.000 |
| moving_elevator | 0 | 0 | 0 | 47 | 5 | 0 | 1 | 0 | 0 | 0.845 |
| rowing | 380 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.000 |
| running_treadmill | 0 | 380 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.000 |
| sitting | 0 | 0 | 380 | 0 | 0 | 0 | 0 | 0 | 0 | 1.000 |
| stand_elevator | 0 | 0 | 0 | 349 | 14 | 0 | 0 | 0 | 0 | 0.918 |
| standing | 0 | 0 | 0 | 0 | 380 | 0 | 0 | 0 | 0 | 1.000 |
| stepper | 0 | 0 | 0 | 0 | 0 | 379 | 0 | 0 | 0 | 0.997 |
| walking | 0 | 0 | 0 | 0 | 0 | 0 | 379 | 0 | 0 | 0.997 |
| walking_tread | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 380 | 0 | 1.000 |
| walking_tread_incl | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 380 | 1.000 |

We can also use a heatmap plot to further aid our visualisation of the information in the confusion matrix table above. The values used in the heatmap below correspond to the values in the confusion matrix divided by the sum of the values in their row, hence showing the proportion of correctly and incorrectly predicted activities for each individual class. Darker blue squares correspond with a larger proportion of predictions, and lighter blue values correspond with a smaller proportion of predictions. The predominantly dark blue squares shown along the diagonal of the heatmap are a quick indicator that for all of the classes, the most commonly predicted class corresponded with the actual class.

As expected, the square with the darkest shade of blue that doesn't fall on the diagonal corresponds with "moving_elevator" activity samples that were incorrectly predicted to be "standing_elevator" activities, but in general this plot shows that predictive performance of the model on unseen test data is, in general, very good across all activity classes.

```r
# load gplots library for heatmap.2 function
library(gplots)

# add heatmap to visualise proportion proportions of correct predictions for each class
heatmap.2(tab, Rowv = NA, Colv = NA, cexRow = 0.8, cexCol = 0.8, dendrogram = "none",
          scale = "row", trace = "none", asp=1, key=FALSE,
        xlab = "Predicted Classes", ylab = "Actual Classes",
        cellnote = round(tab/rowSums(tab),2), notecex = 0.9, notecol = "white",
        margins = c(6,6), col=rev(rich.colors(50, "blues")))

# add title above heatmap
title("             Heatmap of Actual vs Predicted Classes (Scaled by Row)", line = -6)
```



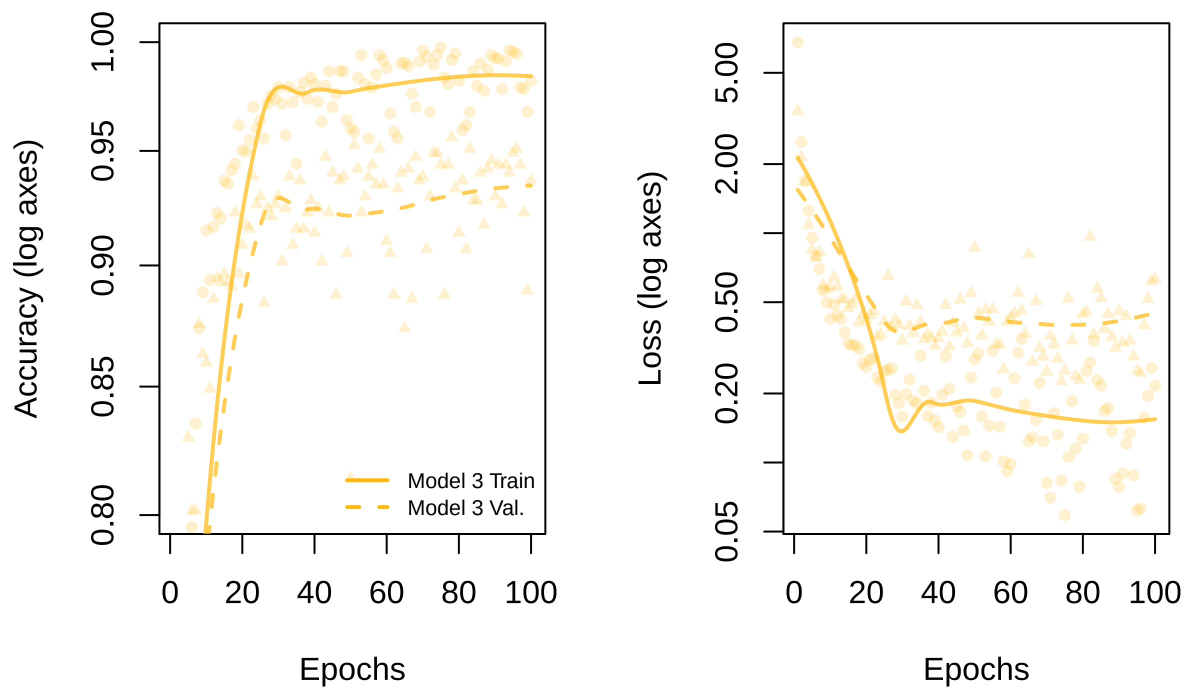
## Looking at the effect of using a single sensor unit

Out of interest, I also wanted to evaluate the predictive performance of this model configuration if data from only one of the sensor units was available. For example, it would be far more practical to gather data for this task if only a single sensor unit was required, or possibly a device such as a suitable smart phone (which the majority of people will already have access to) could be used by anyone to gather the required data and classify the activities they were carrying out during the day.

Therefore, I have decided to retrain the model and evaluate it but only using data from the first sensor unit, which corresponds to the unit connected to the torso of the subjects. This equates to the first 9 columns of the sensor data. The only adjustment that I will make to the model is to alter the input shape from 125 x 45 to 125 x 9.

The loss and accuracy curves for the newly trained model are plotted below, as well as the loss and accuracy values obtained on the test data.

```r
# model 3 - CNN with regularization
model3 <- keras_model_sequential() %>%
  #
  # convolutional layers
  layer_conv_1d(filters=64, kernel_size=5, activation="relu",
                input_shape=c(125, 9)) %>%
  layer_max_pooling_1d(pool_size=2) %>%
  layer_conv_1d(filters=128, kernel_size=25, strides = 5, activation="relu") %>%
  layer_max_pooling_1d(pool_size=2) %>%
  #
  # fully connected layers
  layer_flatten() %>%
  layer_dense(units=64, activation="relu", kernel_regularizer=regularizer_l2(0.1)) %>%
  layer_dropout(0.1) %>%
  layer_dense(units=19, activation="softmax") %>%
  #
  # compile
  compile(
    loss="categorical_crossentropy",
    metrics="accuracy",
    optimizer=optimizer_adam()
  )

# model training
fit3 <- model3 %>% fit(
  x=x_train[,,1:9], y=y_train,
  validation_data=list(x_validation[,,1:9], y_validation),
  epochs=100,
  batch_size=nrow(x_train[,,1:9])*0.01
)

# plot accuracy and loss curves for the final model
acc_loss_plot(cbind(fit3), acc_ylim = c(0.8, 1), first_model_int = 3)
```



```
# get loss and accuracy for test data
model3 %>% evaluate(x_test[,,1:9], y_test, verbose=0)
```

```
##      loss  accuracy
## 0.4242232 0.9382271
```

Again, we can reproduce the confusion matrix tables from earlier to evaluate the performance of the model when data from only the torso sensor unit is available. Interestingly, the model again achieves a very good predictive performance that isn't significantly worse that the predictive performance achieved when data from all 9 units (45 sensors) was used.

```
# get actual classes for the test data
class_y <- class_labels[max.col(y_test)]
# get predicted classes for the test data
class_hat <- class_labels[(model3 %>% predict_classes(x_test[,,1:9]))+1]
# create table comparing actual to predicted classes
tab <- table(class_y, class_hat)

# add column to table showing class accuracy for each individual class
tab_classacc <- cbind(tab, class.accuracy=round(diag(tab)/rowSums(tab),3))

# print table of actual vs predicted classes and class accuracy for each class (first 10 columns)
add_header_above(
  kable_styling(
    kable(tab_classacc[,1:10],
          align = 'c'),
    font_size = 4.5),
  c("Actual"=1, "Predicted"=10))
```

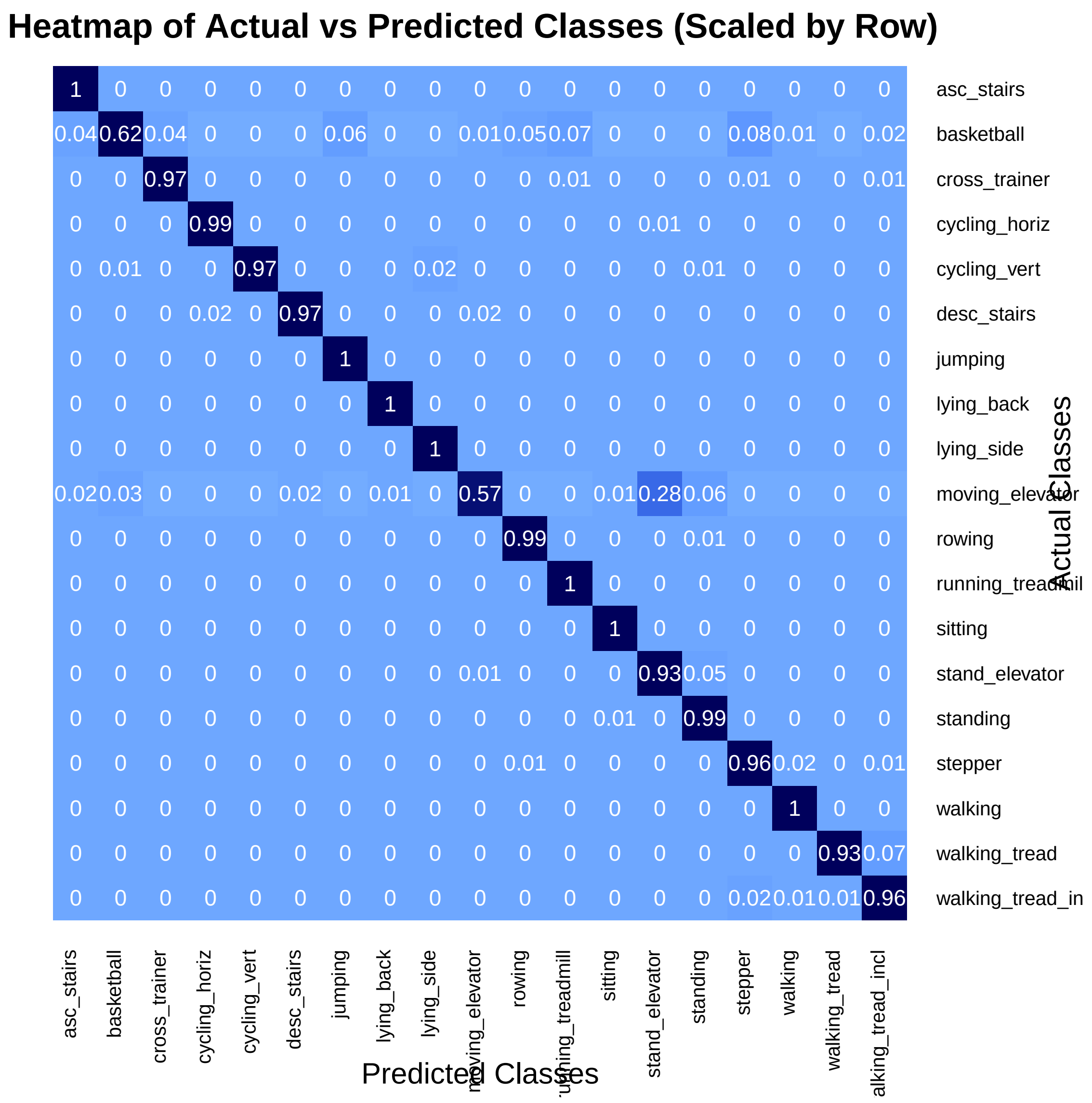
| Actual | Predicted | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | asc__stairs | basketball | cross__trainer | cycling__horiz | cycling__vert | desc__stairs | jumping | lying__back | lying__side | moving__elevator |
| asc__stairs | 379 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| basketball | 15 | 236 | 14 | 0 | 0 | 0 | 21 | 0 | 1 | 5 |
| cross__trainer | 0 | 1 | 368 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| cycling__horiz | 0 | 0 | 0 | 378 | 0 | 0 | 0 | 0 | 0 | 0 |
| cycling__vert | 1 | 3 | 0 | 0 | 367 | 0 | 0 | 0 | 7 | 0 |
| desc__stairs | 0 | 0 | 0 | 6 | 0 | 367 | 1 | 0 | 0 | 6 |
| jumping | 0 | 0 | 0 | 0 | 0 | 0 | 379 | 0 | 0 | 0 |
| lying__back | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 380 | 0 | 0 |
| lying__side | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 380 | 0 |
| moving__elevator | 6 | 12 | 1 | 1 | 0 | 6 | 0 | 3 | 0 | 217 |
| rowing | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| running__treadmill | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sitting | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| stand__elevator | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 |
| standing | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| stepper | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| walking | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| walking__tread | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| walking__tread__incl | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

```r
# print table of actual vs predicted classes and class accuracy for each class (last 10 columns)
add_header_above(
  kable_styling(
    kable(tab_classacc[,11:20],
          align = 'c'),
    font_size = 4.5),
  c("Actual"=1, "Predicted"=9, " "=1))
```

| Actual | Predicted | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | rowing | running__treadmill | sitting | stand__elevator | standing | stepper | walking | walking__tread | walking__tread__incl | class.accuracy |
| asc__stairs | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.997 |
| basketball | 18 | 25 | 1 | 0 | 0 | 32 | 3 | 0 | 9 | 0.621 |
| cross__trainer | 0 | 4 | 0 | 0 | 0 | 4 | 0 | 1 | 2 | 0.968 |
| cycling__horiz | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0.995 |
| cycling__vert | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0.966 |
| desc__stairs | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.966 |
| jumping | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.997 |
| lying__back | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.000 |
| lying__side | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.000 |
| moving__elevator | 0 | 0 | 4 | 107 | 21 | 1 | 1 | 0 | 0 | 0.571 |
| rowing | 375 | 0 | 1 | 0 | 3 | 1 | 0 | 0 | 0 | 0.987 |
| running__treadmill | 0 | 380 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.000 |
| sitting | 0 | 0 | 379 | 0 | 1 | 0 | 0 | 0 | 0 | 0.997 |
| stand__elevator | 0 | 0 | 1 | 355 | 19 | 0 | 0 | 0 | 0 | 0.934 |
| standing | 0 | 0 | 5 | 0 | 375 | 0 | 0 | 0 | 0 | 0.987 |
| stepper | 5 | 0 | 0 | 0 | 0 | 363 | 6 | 1 | 3 | 0.955 |
| walking | 0 | 0 | 0 | 0 | 0 | 0 | 379 | 0 | 0 | 0.997 |
| walking__tread | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 353 | 27 | 0.929 |
| walking__tread__incl | 0 | 0 | 0 | 0 | 0 | 8 | 3 | 5 | 364 | 0.958 |

We can also recreate the heatmap plot from earlier for the new model that only uses data from the torso sensor unit, which again highlights the strong predictive performance of the model.

```r
# load gplots library for heatmap.2 function
library(gplots)

# add heatmap to visualise proportion proportions of correct predictions for each class
heatmap.2(tab, Rowv = NA, Colv = NA, cexRow = 0.8, cexCol = 0.8, dendrogram = "none",
          scale = "row", trace = "none", asp=1, key=FALSE,
        xlab = "Predicted Classes", ylab = "Actual Classes",
        cellnote = round(tab/rowSums(tab),2), notecex = 0.9, notecol = "white",
        margins = c(6,6), col=rev(rich.colors(50, "blues")))

# add title above heatmap
title("             Heatmap of Actual vs Predicted Classes (Scaled by Row)", line = -6)
```



So while the model does achieve the best performance after using the full range of data provided (i.e. from all 5 sensor units, 45 sensors in total), we can still achieve a reasonably comparable performance using data from only a single sensor unit (i.e. 9 sensors).