STAT40970 Machine Learning and AI Assignment 3

Fanahan McSweeney - 20203868

April 13, 2021

1. Encode the images in RGB tensors of width/height of 128 x 128.

The train, validation and test folders containing the images for this assignment are saved in the *Data* folder within the working directory. Firstly, I have created separate image data generators each set. For all sets, I have rescaled the RGB values to be within the range of 0 to 1.

For the training data image generator, I have included several data augmentation transformations. This will in effect generate "new" additional training data from the given training images by applying these transformations randomly to the images during training so that the model doesn't learn from the exact same image multiple times, which in turn can help reduce overfitting. However, the effect of each transformation on the image must be considered carefully, as the desired augmented images should look realistic and not change characteristics of the images which are fundamental to its classification. For example, it would not make sense to include vertical flip augmentation for this assignment as images of rooms will always be taken and viewed horizontally.

For this assignment, I have decided to use the following transformations:

- *zoom_range:* the amount of zoom. I selected a value of 0.2 so zoom would randomly be picked between 0.8 and 1.2.
- *shear_range:* the shear intensity in radians. I selected a value of 0.1, so that only a small amount of random shear would be introduced without distorting the image too much.
- *width_shift_range:* fraction of the total width. I selected a relatively small value of 0.1 as I didn't want to cut too much off the sides of the images, particularly as for room images the information nearer the edges can still be important when classifying the room.
- *height_shift_range:* fraction of the total height. I selected a value of 0.1 for reasons similar to *width_shift_range*.
- *brightness_range:* the range of brightness to apply. I selected a range of 0.7 to 1.3
- *rotation_range:* rotation of the image in degrees. I specified this as 20 degrees so that some rotation could be introduced but not too far from horizontal.
- *horizontal_flip:* randomly flips the image horizontally.
- *fill_mode:* specifies how points outside boundaries of the image are dealt with, for example when width shifting is applied to the image. I selected *nearest* for this so values nearest to the boundary would be repeated into the space outside the boundary.

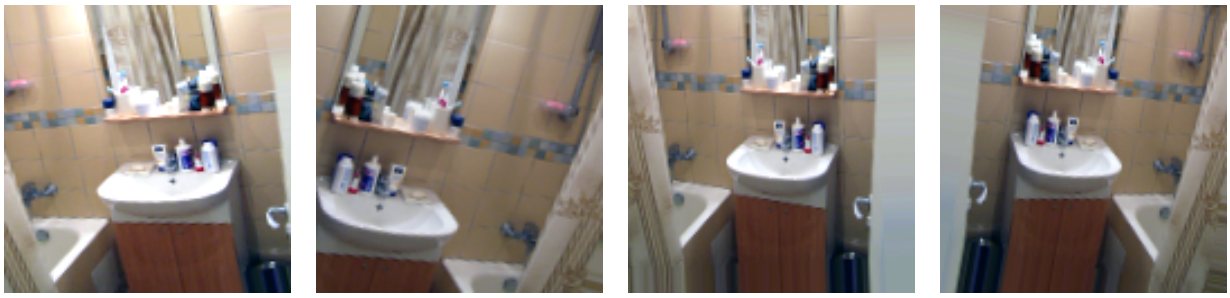
After defining the image data generators, I used the *flow_images_from_directory* function to generate batches of training, validation and test data using the image data generator objects and the images stored in the relevant directories.

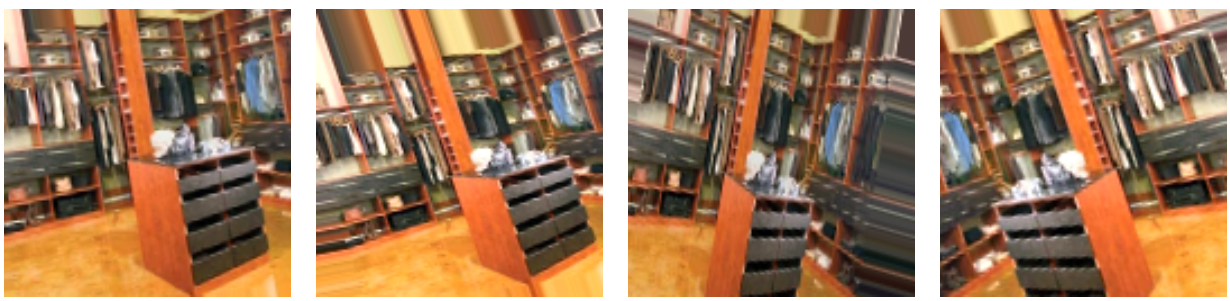
```
library("knitr")
library("keras")
library("kableExtra")
```

```
# define directories containing train/validation/test images
train_dir <- "Data/data_indoor/train"
validation_dir <- "Data/data_indoor/validation"
test_dir <- "Data/data_indoor/test"

# set data augmentation generator for training data
data_augment <- image_data_generator(
  rescale = 1/255,
  zoom_range = 0.2,
  shear_range = 0.1,
  width_shift_range = 0.1,
  height_shift_range = 0.1,
  brightness_range = c(0.7,1.3),
  rotation_range = 20,
  horizontal_flip = TRUE,
  fill_mode = "nearest"
)

# set validation and test image data generators
validation_datagen <- image_data_generator(rescale=1/255)
test_datagen <- image_data_generator(rescale=1/255)

# generate batches of train data with data augmentation
train_generator <- flow_images_from_directory(
  train_dir,
  data_augment,
  target_size=c(128, 128),
  batch_size=128,
  class_mode="categorical"
)

# generate batches of validation data
validation_generator <-flow_images_from_directory(
  validation_dir,
  validation_datagen,
  target_size=c(128, 128),
  batch_size=128,
  class_mode="categorical"
)

# generate batches of test data
test_generator <-flow_images_from_directory(
  test_dir,
  test_datagen,
  target_size=c(128, 128),
  batch_size=128,
  class_mode="categorical",
  shuffle = FALSE
)
```

For illustrative purposes, I have plotted examples of images below from the training set after data augmentation has been applied. Specifically, the first 4 images below are randomly augmented versions of a picture of a bathroom, and the second 4 images below are randomly augmented versions of a picture of a closet.

```
# load jpeg library
library("jpeg")
# plot 8 examples of augmented images (4 bathrooms, 4 closets)
par(mfrow=c(2,4), mar=rep(0.5,4))

# load bathroom image from train set (with dimensions 128x128)
img_array <- image_to_array(image_load("Data/data_indoor/train/bathroom/b1.jpg",
                                       target_size=c(128,128)))
img_array <- array_reshape(img_array, c(1,128,128,3))
augmentation_generator <- flow_images_from_data(
  img_array, generator=data_augment, batch_size=1
  )
# generate 4 augmented images of bathroom and plot each image
for(i in 1:4) {
  batch <- generator_next(augmentation_generator)
  plot(as.raster(batch[1,,,]))}

# load closet image from train set (with dimensions 128x128)
img_array <- image_to_array(image_load("Data/data_indoor/train/closet/1.jpg",
                                       target_size =c(128,128)))
img_array <- array_reshape(img_array,c(1,128,128,3))
augmentation_generator <-flow_images_from_data(
  img_array, generator=data_augment, batch_size=1
  )
# generate 4 augmented images of closet and plot each image
for(i in 1:4) {
  batch <- generator_next(augmentation_generator)
  plot(as.raster(batch[1,,,]))}
```





2. Deploy at least 3 different CNNs characterized by different configurations, hyperparameters, and training settings. Motivate clearly the choices made in relation to the settings, configurations, and hyperparameteres used to define the different CNNs.

There are an infinite number of different CNN configurations that we can consider for this task. Ideally, we would test a broad range of different CNN structures, using different numbers of convolution layers and numbers of filters, different numbers of fully connected layers and hidden units, kernel sizes and stride values, optimisers and learning rates, pooling sizes, regularization techniques, and the list goes on!

Unfortunately that is not possible, and due to the time constraints of this assignment it is not possible/practical to test more than a relatively small number of different configurations. Below, I have created 4 different models to be deployed as part of this assignment. To reduce the total number of configurations to consider, I limited myself to the following constraints.

1. Only models containing 3 convolution layers will be considered.
2. The rectified linear activation function (ReLU) will be used for all convolution layers and for the fully connected hidden layers.
3. 2x2 maximum pooling will be used after each convolution layer.
4. The Adam optimiser will be used for each model.

Due to the nature of the image classification problem (classifying images of rooms) I decided to use *same* padding for each convolution layer, so that information at the edges of the images would be fully covered by the specified filters, as objects or features that could be helpful when classifying the room could be located anywhere in the image including areas near the borders.

Model 1

The configuration for the first model I deployed is shown below. In summary, it consists of the following:

- Convolution Layers
  - Convolution layer, 32 filters, 3x3 kernel size, ReLU activation
  - Maximum Pooling layer, 2x2
  - Convolution layer, 64 filters, 3x3 kernel size, ReLU activation
  - Maximum Pooling layer, 2x2
  - Convolution layer, 128 filters, 3x3 kernel size, ReLU activation
  - Maximum Pooling layer, 2x2
- Fully Connected Layers
  - Hidden layer, 256 units, ReLU activation
  - Hidden layer, 128 units, ReLU activation
  - Output layer, 10 units, softmax activation

I chose to use a relatively small filter size of 3x3 for each of the convolution layers for this first model to collect as much local information as possible, as the room images typically have several different objects and features of interest across the full range of the image. 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.

```r
# Model 1 ---------------------------------
model1 <- keras_model_sequential() %>%
  #
  # convolutional layers
  layer_conv_2d(filters=32, kernel_size=c(3,3), padding = "same", activation="relu",
                input_shape=c(128,128,3)) %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_conv_2d(filters=64, kernel_size=c(3,3), padding = "same", activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_conv_2d(filters=128, kernel_size=c(3,3), padding = "same", activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  #
  # fully connected layers
  layer_flatten() %>%
  layer_dense(units=256, activation="relu") %>%
  layer_dense(units=128, activation="relu") %>%
  layer_dense(units=10, activation="softmax") %>%
  #
  # compile
  compile(
    loss="categorical_crossentropy",
    metrics="accuracy",
    optimizer=optimizer_adam()
  )
```

Model 2

The configuration for the second model I deployed is shown below. It follows closely to model 1, with some minor modifications. In summary, it consists of the following:

- Convolution Layers
  - Convolution layer, 32 filters, 3x3 kernel size, ReLU activation
  - Maximum Pooling layer, 2x2
  - Convolution layer, 64 filters, 3x3 kernel size, ReLU activation
  - Maximum Pooling layer, 2x2
  - Convolution layer, 128 filters, 9x9 kernel size with a stride of 2, ReLU activation
  - Maximum Pooling layer, 2x2
- Fully Connected Layers
  - Hidden layer, 256 units, ReLU activation
  - Hidden layer, 128 units, ReLU activation
  - Hidden layer, 64 units, ReLU activation
  - Output layer, 10 units, softmax activation

I chose to use a slightly larger filter size of 9x9 with a stride of 2 for the final convolution layer for this model to collect more information of global features in the images, in an effort to gather improve the model's ability at identifying more global features in the images. I also chose to add an additional fully connected layer to see if adding a small amount of additional depth and complexity to the fully connected layers could improve performance.

```
# Model 2 ---------------------------------
model2 <- keras_model_sequential() %>%
  #
  # convolutional layers
  layer_conv_2d(filters=32, kernel_size=c(3,3), padding = "same", activation="relu",
                input_shape=c(128,128,3)) %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_conv_2d(filters=64, kernel_size=c(3,3), padding = "same", activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_conv_2d(filters=128, kernel_size=c(9,9), strides = 2, padding = "same",
                activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  #
  # fully connected layers
  layer_flatten() %>%
  layer_dense(units=256, activation="relu") %>%
  layer_dense(units=128, activation="relu") %>%
  layer_dense(units=64, activation="relu") %>%
  layer_dense(units=10, activation="softmax") %>%
  #
  # compile
  compile(
    loss="categorical_crossentropy",
    metrics="accuracy",
    optimizer =optimizer_adam()
  )
```

Model 3

The configuration for the third model I deployed is shown below. Again, this follows on closely from the previous model, but with some additions made to introduce more regularisation. In summary, it consists of the following:

- Convolution Layers
  - Convolution layer, 32 filters, 3x3 kernel size, ReLU activation
  - Maximum Pooling layer, 2x2
  - Convolution layer, 64 filters, 3x3 kernel size, ReLU activation
  - Maximum Pooling layer, 2x2
  - Dropout, rate of 0.1
  - Convolution layer, 128 filters, 9x9 kernel size with a stride of 2, ReLU activation
  - Maximum Pooling layer, 2x2
  - Dropout, rate of 0.1
- Fully Connected Layers
  - Hidden layer, 256 units, ReLU activation
  - Dropout, rate of 0.3
  - Hidden layer, 128 units, ReLU activation
  - Dropout, rate of 0.2
  - Hidden layer, 64 units, ReLU activation
  - Output layer, 10 units, softmax activation

For this model, I added dropout regularisation at a number of layers in the model in order to reduce the effect of overfitting. I added dropout after the pooling stages after 2 of the convolution layers, with a very conservative rate of 0.1 as it is not as effective when used on convolution layers, and I also added dropout with slightly higher rates to 2 of the fully connected layers.

```r
# Model 3 ---------------------------------
model3 <- keras_model_sequential() %>%
  #
  # convolutional layers
  layer_conv_2d(filters=32, kernel_size=c(3,3), padding = "same", activation="relu",
                input_shape=c(128,128,3)) %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_conv_2d(filters=64, kernel_size=c(3,3), padding = "same", activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_dropout(0.1) %>%
  layer_conv_2d(filters=128, kernel_size=c(9,9), strides = 2, padding = "same",
                activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_dropout(0.1) %>%
  #
  # fully connected layers
  layer_flatten() %>%
  layer_dense(units=256, activation="relu") %>%
  layer_dropout(0.3) %>%
  layer_dense(units=128, activation="relu") %>%
  layer_dropout(0.2) %>%
  layer_dense(units=64, activation="relu") %>%
  layer_dense(units=10, activation="softmax") %>%
  #
  # compile
  compile(
    loss="categorical_crossentropy",
    metrics="accuracy",
    optimizer =optimizer_adam()
  )
```

Model 4

The configuration for the fourth and final model I deployed is shown below. Again, this follows on closely from the previous models, but in this case I wanted to include batch normalisation as a form of regularisation. In summary, it consists of the following:

- Convolution Layers
  - Convolution layer, 32 filters, 3x3 kernel size, ReLU activation
  - Maximum Pooling layer, 2x2
  - Batch normalisation layer
  - Convolution layer, 64 filters, 3x3 kernel size, ReLU activation
  - Maximum Pooling layer, 2x2
  - Batch normalisation layer
  - Convolution layer, 128 filters, 9x9 kernel size with a stride of 2, ReLU activation
  - Maximum Pooling layer, 2x2
- Fully Connected Layers
  - Batch normalisation layer
  - Hidden layer, 256 units, ReLU activation
  - Dropout, rate of 0.1
  - Hidden layer, 128 units, ReLU activation
  - Dropout, rate of 0.1
  - Hidden layer, 64 units, ReLU activation
  - Output layer, 10 units, softmax activation

For this model, I introduced batch normalisation after the pooling following the convolution layers, which allows the convolution layers to learn more independently. This can also have a regularisation effect to reduce the likelihood of overfitting in the model. I also added dropout with a relatively low rate of 0.1 to 2 of the fully connected layers to also reduce overfitting.

```r
# Model 4 ---------------------------------
model4 <- keras_model_sequential() %>%
  #
  # convolutional layers
  layer_conv_2d(filters=32, kernel_size=c(3,3), padding = "same", activation="relu",
                input_shape=c(128,128,3)) %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_batch_normalization() %>%
  layer_conv_2d(filters=64, kernel_size=c(3,3), padding = "same", activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_batch_normalization() %>%
  layer_conv_2d(filters=128, kernel_size=c(9,9), strides = 2, padding = "same",
                activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  #
  # fully connected layers
  layer_flatten() %>%
  layer_batch_normalization() %>%
  layer_dense(units=256, activation="relu") %>%
  layer_dropout(0.1) %>%
  layer_dense(units=128, activation="relu") %>%
  layer_dropout(0.1) %>%
  layer_dense(units=64, activation="relu") %>%
  layer_dense(units=10, activation="softmax") %>%
  #
  # compile
  compile(
    loss="categorical_crossentropy",
    metrics="accuracy",
    optimizer =optimizer_adam()
  )
```

3. Train the networks on the training data images and compare their performance on the validation set of images. Comment on their relative performance.

To train the models defined above on the training data, the *fit_generator* function was used. The function also takes the validation data as an input, and evaluates the validation performance of the model at each stage of the training. Due to the difficulty of the classification task at hand, I decided to perform the initial model training using a relatively large number of epochs of 100.

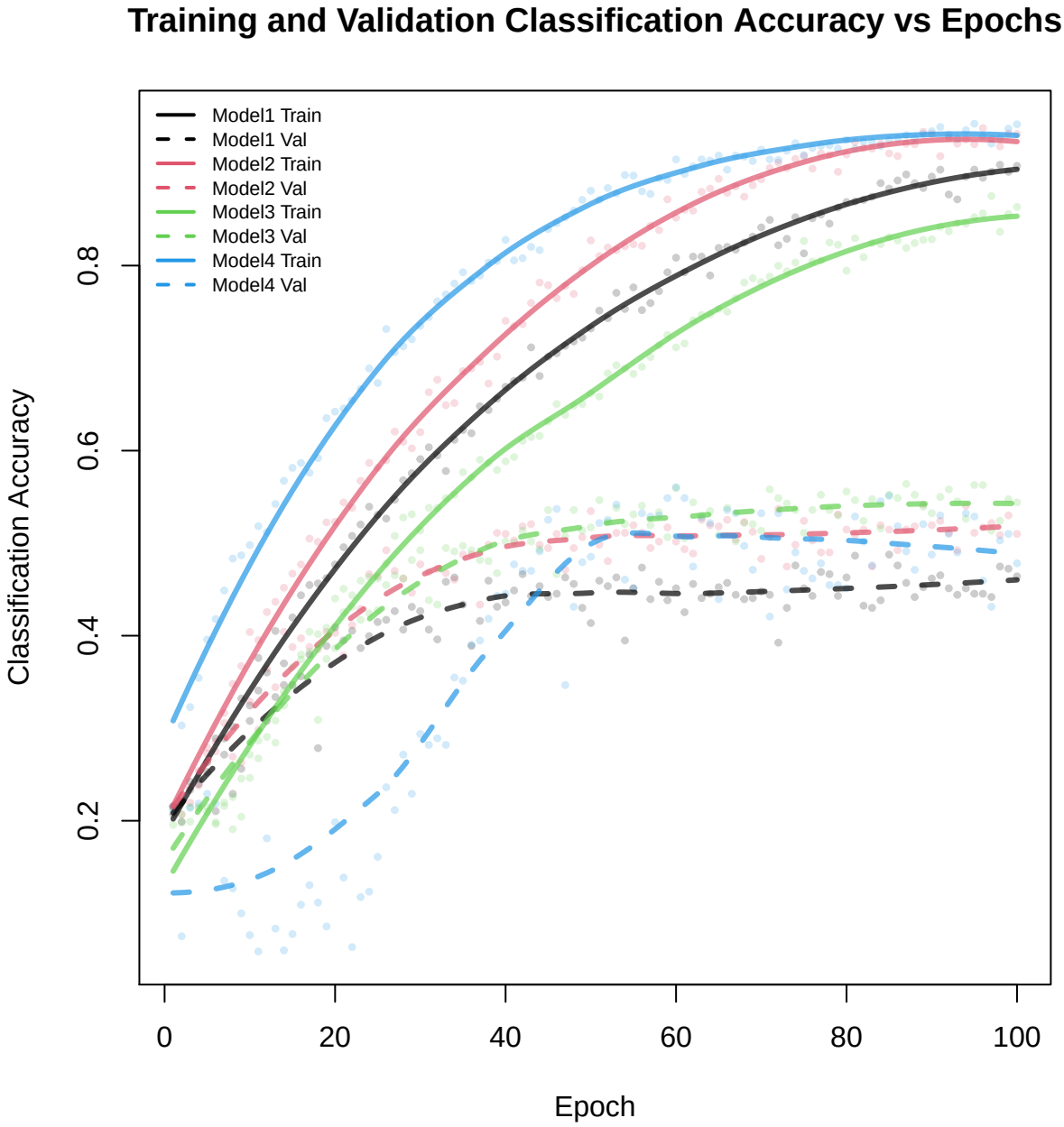
```
# Model 1 ----------------------------------
# fit the model with the training data generator, using validation data generator for validation
fit1 <- model1 %>% fit_generator(
  train_generator,
  steps_per_epoch=ceiling(train_generator$n/train_generator$batch_size),
  epochs=100,
  validation_data=validation_generator,
  validation_steps=ceiling(validation_generator$n/validation_generator$batch_size),
  verbose=0
)

# Model 2 ----------------------------------
# fit the model with the training data generator, using validation data generator for validation
fit2 <- model2 %>% fit_generator(
  train_generator,
  steps_per_epoch=ceiling(train_generator$n/train_generator$batch_size),
  epochs=100,
  validation_data=validation_generator,
  validation_steps=ceiling(validation_generator$n/validation_generator$batch_size),
  verbose=0
)

# Model 3 ----------------------------------
# fit the model with the training data generator, using validation data generator for validation
fit3 <- model3 %>% fit_generator(
  train_generator,
  steps_per_epoch=ceiling(train_generator$n/train_generator$batch_size),
  epochs=100,
  validation_data=validation_generator,
  validation_steps=ceiling(validation_generator$n/validation_generator$batch_size),
  verbose=0
)

# Model 4 ----------------------------------
# fit the model with the training data generator, using validation data generator for validation
fit4 <- model4 %>% fit_generator(
  train_generator,
  steps_per_epoch=ceiling(train_generator$n/train_generator$batch_size),
  epochs=100,
  validation_data=validation_generator,
  validation_steps=ceiling(validation_generator$n/validation_generator$batch_size),
  verbose=0
)
```

After completing the model training, we can extract the classification accuracy and loss values for the training and validation data after each epoch for each of the models and plot them to compare the performance of each model. As there can be a significant amount of variation in the accuracy and loss values between epochs, we can define a smoothing function as done below which can be used to plot a smoothed line following the general trend of the data for each model.

The first plot below shows the classification accuracy for the training and validation data for each of the four models. All models achieved a validation classification accuracy greater than 0.4, while their training classification accuracy reached over 0.8 in each case, suggesting a significant amount of overfitting is occurring. Of the four models, model 3 (where dropout was introduced for regularisation) appears to have performed the best, reaching a validation accuracy of approximately 0.5.

Model 2 performed noticeably better than model 1, reaching a validation classification accuracy that is approximately 0.5 higher than what model 1 reached. This would suggest that including a larger kernel filter on the final convolution layer, as well as an additional fully connected hidden layer, improved the performance of the model. The validation accuracy for model 4 (where batch normalisation) appeared to reach a peak after 50 epochs before it began to decrease steadily. While it is possible that this may be a local maximum and that further training of the model could once again lead to improved results, it still appears that model 3 is the best option based off the classification accuracy data.

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

# combine training/validation accuracy values of the 4 models obtained during training
accuracy_data <- cbind(fit1$metrics$accuracy, fit1$metrics$val_accuracy,
                       fit2$metrics$accuracy, fit2$metrics$val_accuracy,
                       fit3$metrics$accuracy, fit3$metrics$val_accuracy,
                       fit4$metrics$accuracy, fit4$metrics$val_accuracy)

# Plot classification accuracy data points
matplot(accuracy_data, pch=20, col = adjustcolor(rep(1:4,each=2), 0.2),
        main="Training and Validation Classification Accuracy vs Epochs",
        ylab="Classification Accuracy", xlab="Epoch", cex=0.7)
# Add smoothed lines to plot
matlines(apply(accuracy_data, 2, smooth_line), lty=c(1,2),
         col=adjustcolor(rep(1:4,each=2), 0.7), lwd=3)
# add legend
legend("topleft", c("Model1 Train", "Model1 Val", "Model2 Train", "Model2 Val",
                    "Model3 Train", "Model3 Val", "Model4 Train", "Model4 Val"),
       lty = c(1,2), col=rep(1:4,each=2), cex=0.7, bty="n", lwd=2)
```

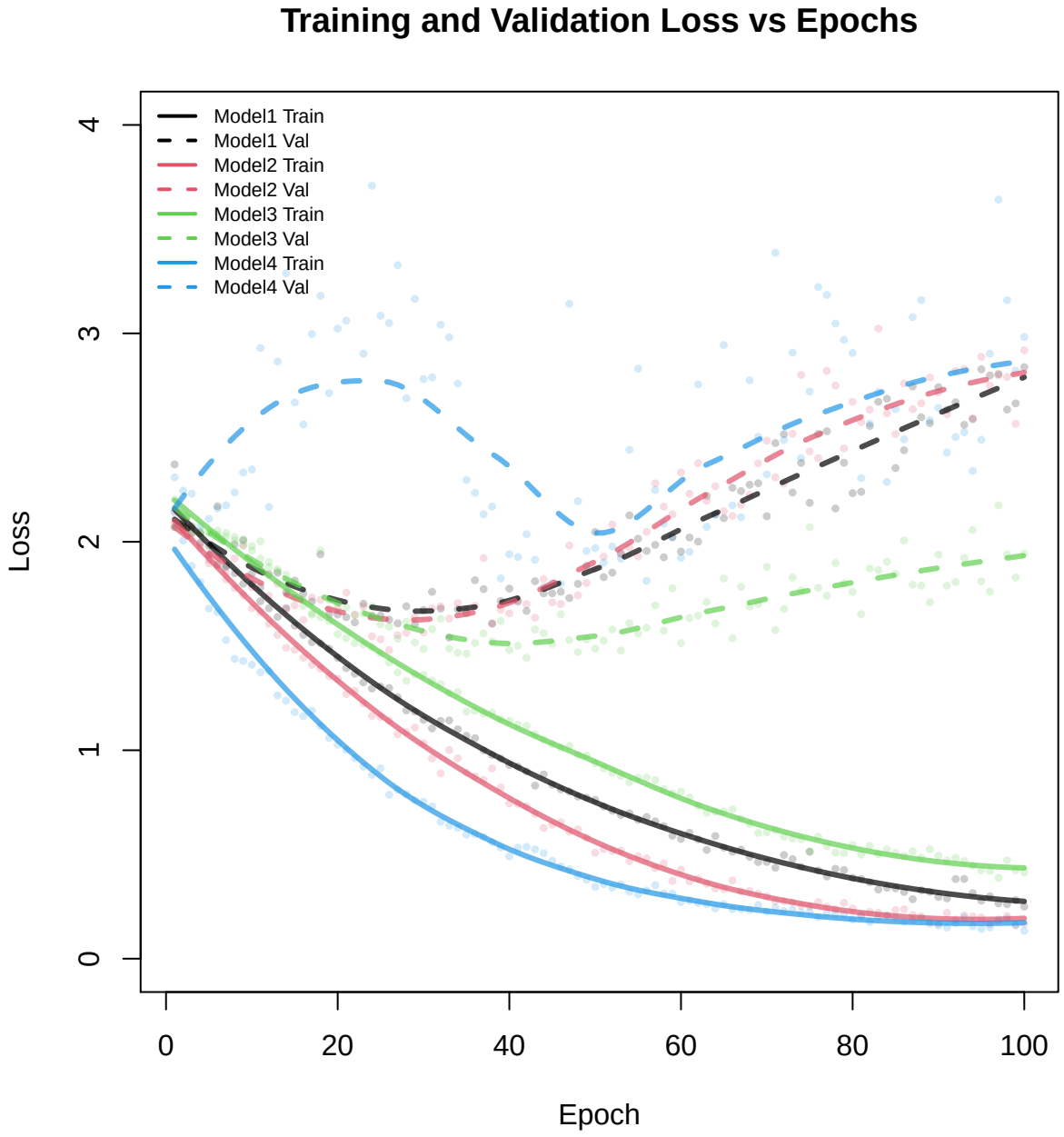


Similarly, I have plotted the training and validation loss curves for each of the models below. In each case the validation loss began to increase after approximately 40 epochs, and hence the gap between training and validation loss curves began to increase after this point, again suggesting a significant level of overfitting. Once again, model 3 appears to have performed the best based on these curves, with a much more gradual increase in the validation loss after approximately 40 epochs when compared to the remaining models, and achieving a significantly lower validation loss (approximately 2) than the remaining models (all greater than 2.5).

Based off the results obtained, it is clear that model 3 is the best performing model for this room classification task.

```
# combine training/validation loss values of the 4 models obtained during training
loss_data <- cbind(fit1$metrics$loss, fit1$metrics$val_loss,
                   fit2$metrics$loss, fit2$metrics$val_loss,
                   fit3$metrics$loss, fit3$metrics$val_loss,
```

```
                    fit4$metrics$loss, fit4$metrics$val_loss)

# Plot loss data points
matplot(loss_data, pch=20, col = adjustcolor(rep(1:4,each=2), 0.2),
        main="Training and Validation Loss vs Epochs",
        ylab="Loss", xlab="Epoch", cex=0.7, ylim=c(0,4))
# Add smoothed lines to plot
matlines(apply(loss_data, 2, smooth_line), lty=c(1,2),
         col=adjustcolor(rep(1:4,each=2), 0.7), lwd=3)
# Add legend
legend("topleft", c("Model1 Train", "Model1 Val", "Model2 Train", "Model2 Val",
                    "Model3 Train", "Model3 Val", "Model4 Train", "Model4 Val"),
       lty = c(1,2), col=rep(1:4,each=2), cex=0.7, bty="n", lwd=2)
```



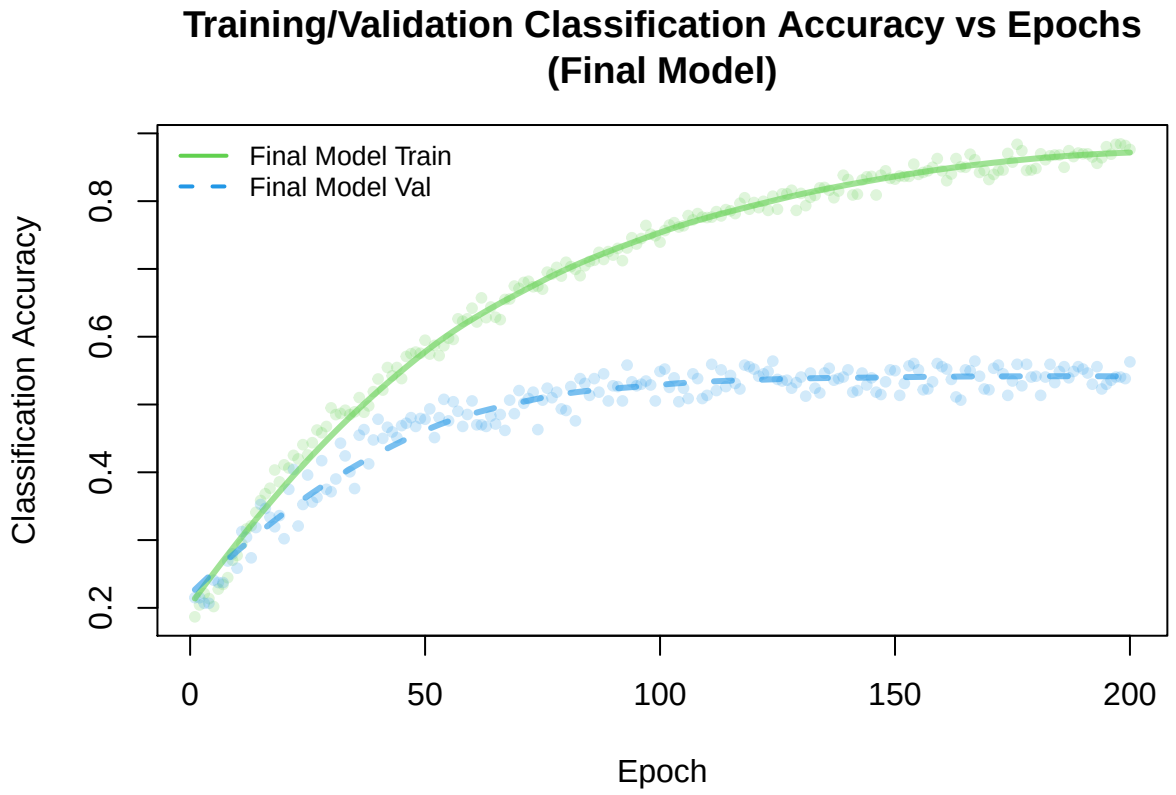
4. Evaluate the predictive performance of the best CNN on the test data images.

Having decided that model 3 performed the best of the proposed models, I decided to retrain the model again but for a larger number of epochs, as it appeared that the classification accuracy for the validation images was still increasing slowly after the model was fitted over 100 epochs. Below, I have defined the model again, but this time I chose to train it over 200 epochs.

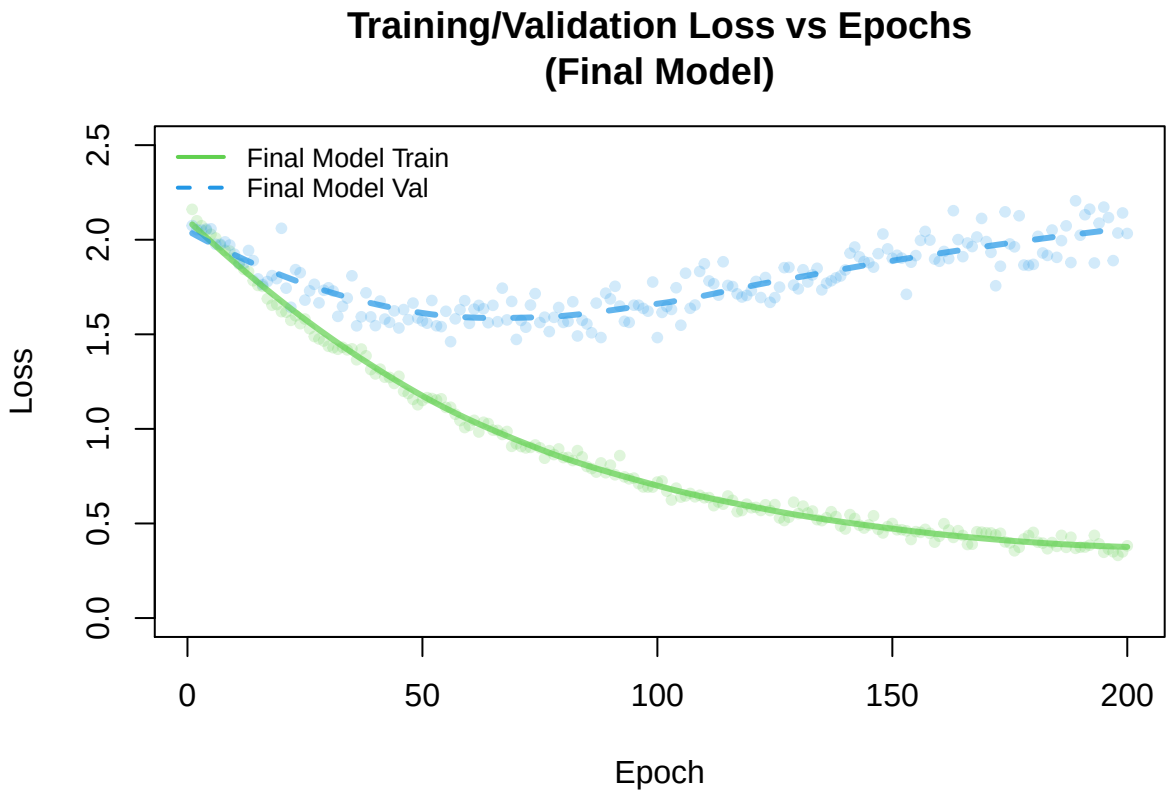
```
# Model Final --------------------------------
model_final <- keras_model_sequential() %>%
  #
  # convolutional layers
  layer_conv_2d(filters=32, kernel_size=c(3,3), padding = "same", activation="relu",
                input_shape=c(128,128,3)) %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_dropout(0.1) %>%
  layer_conv_2d(filters=64, kernel_size=c(3,3), padding = "same", activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_dropout(0.1) %>%
  layer_conv_2d(filters=128, kernel_size=c(9,9), strides = 2, padding = "same",
                activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2)) %>%
  layer_dropout(0.1) %>%
  #
  # fully connected layers
  layer_flatten() %>%
  layer_dense(units=256, activation="relu") %>%
  layer_dropout(0.4) %>%
  layer_dense(units=128, activation="relu") %>%
  layer_dropout(0.3) %>%
  layer_dense(units=64, activation="relu") %>%
  layer_dropout(0.2) %>%
  layer_dense(units=10, activation="softmax") %>%
  #
  # compile
  compile(
    loss="categorical_crossentropy",
    metrics="accuracy",
    optimizer =optimizer_adam()
  )

# fit the model with the training data generator, using validation data generator for validation
# Use 200 epochs this time
fit_final <- model_final %>% fit_generator(
  train_generator,
  steps_per_epoch=ceiling(train_generator$n/train_generator$batch_size),
  epochs=200,
  validation_data=validation_generator,
  validation_steps=ceiling(validation_generator$n/validation_generator$batch_size),
  verbose=0
)
```

The resulting training and validation classification accuracy curves and loss curves are shown on the plots below. As expected, these curves appear to follow the general trends observed for model 3 in the plots seen earlier, with a very slow increase in the validation classification accuracy after 100 epochs, and a slow but steady increase in the validation loss after 100 epochs.

```
# combine training and validation accuracy values
accuracy_data_f <- cbind(fit_final$metrics$accuracy, fit_final$metrics$val_accuracy)

# Plot classification accuracy data points
matplot(accuracy_data_f, pch=20, col = adjustcolor(3:4, 0.2),
        main="Training/Validation Classification Accuracy vs Epochs\n(Final Model)",
        ylab="Classification Accuracy", xlab="Epoch")
# Add smoothed lines to plot
matlines(apply(accuracy_data_f, 2, smooth_line), lty=c(1,2),
         col=adjustcolor(3:4, 0.6), lwd=3)
# add legend
legend("topleft", c("Final Model Train", "Final Model Val"),
       lty = c(1,2), col=3:4, cex=0.8, bty="n", lwd=2)
```



```r
# combine training and validation accuracy values
loss_data_f <- cbind(fit_final$metrics$loss, fit_final$metrics$val_loss)

# Plot loss data points
matplot(loss_data_f, pch=20, col = adjustcolor(3:4, 0.2),
        main="Training/Validation Loss vs Epochs\n(Final Model)",
        ylab="Loss", xlab="Epoch", ylim=c(0,2.5))
# Add smoothed lines to plot
matlines(apply(loss_data_f, 2, smooth_line), lty=c(1,2),
         col=adjustcolor(3:4, 0.7), lwd=3)
# Add legend
legend("topleft", c("Final Model Train", "Final Model Val"),
       lty = c(1,2), col=3:4, cex=0.8, bty="n", lwd=2)
```



We can now evaluate the predictive performance of this trained model by generating predictions for the test images set, and comparing the predicted classes to the actual classes of the test data. Firstly, I will define a vector of labels corresponding to the class labels. I will then create a factor of the actual classes for each of the test images, replacing class index values with the actual class labels.

```
# vector of class labels
class_labels <-c("bathroom","bedroom","children_room",
                 "closet","corridor", "dining_room",
                 "garage","kitchen","living_room","stairs")

# generate vector of classes for the test data set
act_class <- factor(class_labels[(test_generator$classes+1)], levels = class_labels)
```

Next, I will use the *evaluate_generator* and *predict_generator* functions in order to predict the classes of the test set images. The *predict_generator* function returns a matrix with the number of rows equal to the number of images in the test set, and the number of columns equalling the number of classes in our data (10 in total). For each image, the predicted probability that the image is associcated which each class is saved in the corresponding column.

```
# Evaluate/predict classes for test data set (Final Model)
model_final %>% evaluate_generator(test_generator,
                                   steps = ceiling(test_generator$n/test_generator$batch_size))
preds_final <- model_final %>% predict_generator(test_generator,
                                                  steps = ceiling(test_generator$n/test_generator$batch_
```

To determine which class the model predicts for each test image, we can simply get the index of column which resulted in the maximum probability. The corresponding vector of indices can then be converted to a factor of predicted test values, again replacing the index values with the class labels.

Below, I have printed a confusion matrix comparing the actual classes (rows) to the predicted classes (columns) for the test images. The final column also shows the classification accuracy obtained for each individual classs. From the table, we can quickly see that the model performed worst at classifying bathrooms (0.265), while it achieved relatively high accuracy values when classifying garages (0.71), corridors (0.663) and closets (0.636).

The overall classification accuracy for the test set was 0.538 as printed below. Considering how difficult this classification task is, and the relatively large number of possible classes to choose from, I believe that the performance of this model is quite impressive given the limited CNN architecture considered for thisn assignment.

```
# find the highest probability class for each prediction
pred_class <- max.col(preds_final)
# convert to factor (added to ensure all possible class levels are included in table)
pred_class <- factor(class_labels[pred_class], levels = class_labels)
# create confusion matrix comparing predicted values with actual values
tab <- table(act_class, pred_class)

# print table of actual vs predicted classes and class accuracy for each class
add_header_above(
  kable_styling(
    kable(cbind(tab, class.accuracy=round(diag(tab)/rowSums(tab),3)),
          align = 'c'),
    font_size = 5),
  c("Actual"=1, "Predicted"=10, " "=1))
```

Actual	Predicted										
	bathroom	bedroom	children_room	closet	corridor	dining_room	garage	kitchen	living_room	stairs	class.accuracy
bathroom	13	9	0	1	2	1	0	15	4	4	0.265
bedroom	0	94	5	0	3	7	3	17	34	2	0.570
children_room	0	0	11	4	0	3	0	6	4	0	0.393
closet	0	2	1	21	0	0	2	5	2	0	0.636
corridor	2	3	0	3	57	1	3	8	3	6	0.663
dining_room	0	5	3	0	1	32	0	8	18	1	0.471
garage	0	1	1	2	3	1	11	2	3	1	0.440
kitchen	1	4	1	1	4	11	4	130	24	3	0.710
living_room	1	27	2	4	1	25	6	29	76	5	0.432
stairs	1	3	4	1	5	4	0	2	5	13	0.342

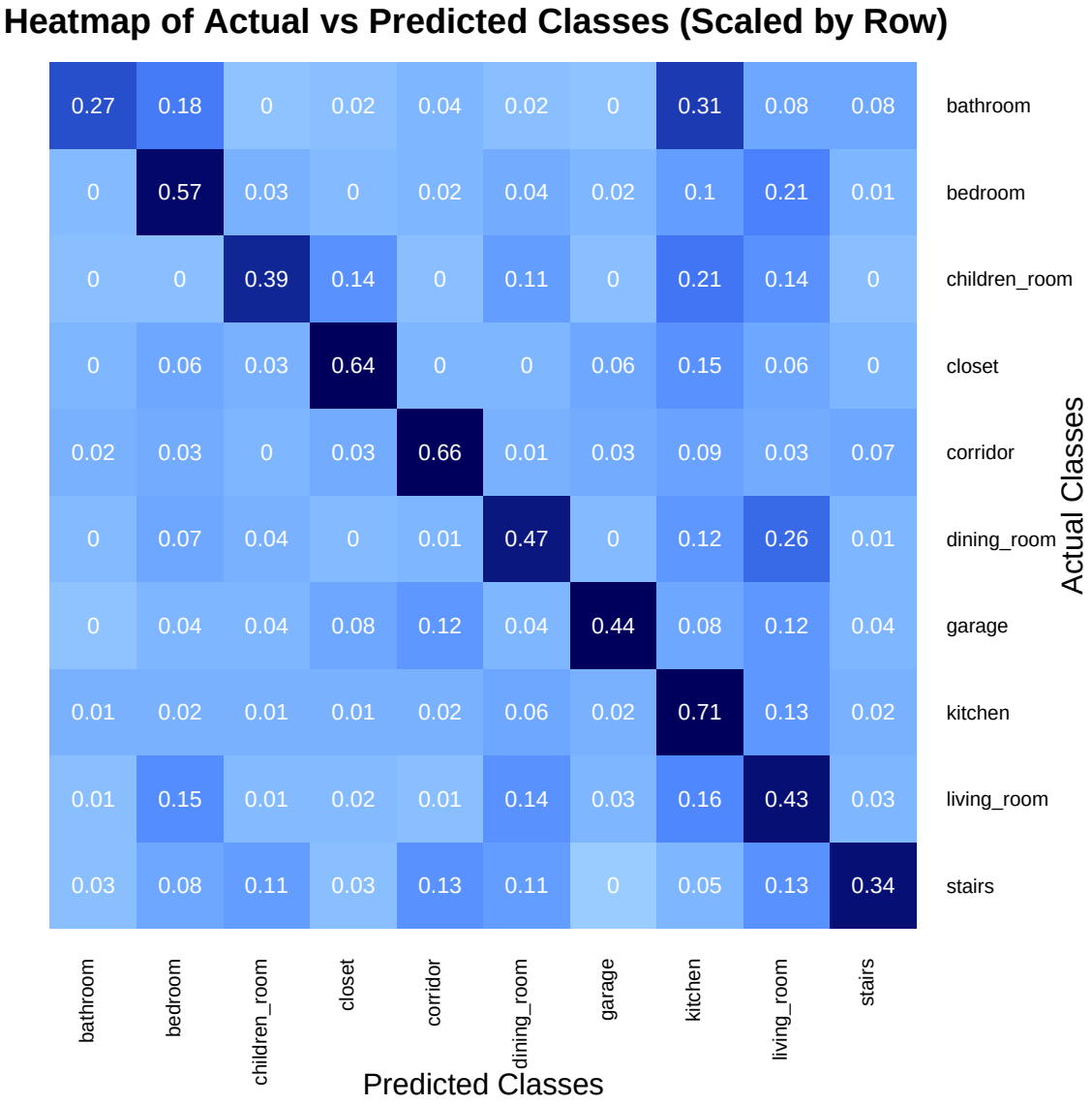
```
# print classification accuracy
cat("\nOverall Test Classification Accuracy: ", sum(diag(tab))/sum(tab), "\n\n")
```

```
##
## Overall Test Classification Accuracy:  0.5381904
```

We can also use heatmaps to aid 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 images 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 almost all of the classes, the most commonly predicted class corresponded with the actual class.

The most noticeable exception to this can be seen on the top row of the heatmap corresponding with the actual classes for the bathroom images. Earlier, this was identified as the class where the lowest test classification accuracy was achieved. We can see that 27% of the bathroom images were correctly classified as a bathroom, but we can also quickly identify that another square on that row has a darker blue colour, with a value of 31%. This corresponds to bathroom images that were incorrectly classified as kitchens, so interestingly the model classified more images of the test images of bathrooms as kitchens than bathrooms themselves!

```
# 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)
```


Heatmap of Actual vs Predicted Classes (Scaled by Row)
0.27 0.18 0 0.02 0.04 0.02 0 0.31 0.08 0.08 bathroom
0 0.57 0.03 0 0.02 0.04 0.02 0.1 0.21 0.01 bedroom
0 0 0.39 0.14 0 0.11 0 0.21 0.14 0 children_room
0 0.06 0.03 0.64 0 0 0.06 0.15 0.06 0 closet
0.02 0.03 0 0.03 0.66 0.01 0.03 0.09 0.03 0.07 corridor
0 0.07 0.04 0 0.01 0.47 0 0.12 0.26 0.01 dining_room
0 0.04 0.04 0.08 0.12 0.04 0.44 0.08 0.12 0.04 garage
0.01 0.02 0.01 0.01 0.02 0.06 0.02 0.71 0.13 0.02 kitchen
0.01 0.15 0.01 0.02 0.01 0.14 0.03 0.16 0.43 0.03 living_room
0.03 0.08 0.11 0.03 0.13 0.11 0 0.05 0.13 0.34 stairs
Actual Classes
bathroom
bedroom
children_room
closet
corridor
dining_room
garage
kitchen
living_room
stairs
Predicted Classes