ML&AI_Assignment 3

Divya Dwivedi: 22200315

2023-04-16

The task requires deploying at least 4 different deep learning systems characterized by different configurations. The data is a collection of images of different indoor scenes from rooms and locations commonly present ina standard family home.Since we need to map image data to an output variable, I shall be using one Deep Neural network and other at least three other Convolutional Neural Networks to train my model.

```
#Loading libraries
library(keras)
library(reticulate)
library(tensorflow)
library(tfruns)
library(ggplot2)
library(gridExtra)
# Install Pillow package
#py_install("pillow")
```

We shall initially determine the batch size to train our deep learning model using a data set with 1713 samples using the ceiling function.

```
#Determine the batch size for training a deep learning model using a data set with 1713 samples
ceiling(1713*0.01)
```

```
## [1] 18
```

The function above computes the batch size as 18. This means that we will divide the dataset into mini- batches of size 18 during training. There are 1713 images in the training set and 851 images in the validationand test sets.

Hence, the batch size is set to 20 to make sure to have at least 1-2 images per class.

For a RGB image of size 64x64 pixels, the target size typically refers to the desired input shape of the imagewhen it is fed into a deep learning model.

Assuming that the image is in standard RGB format with 3 channels (Red, Green, and Blue), the target size would be (64, 64, 3), where 64 is the height and width of the image, and 3 is the number of channels

```
batch <- 20

# validation data generator - same for all models
validation_generator <- flow_images_from_directory(
directory = "data_indoor/validation",
generator = image_data_generator(rescale = 1/255),
target_size = c(64, 64),
batch_size = batch,
```

```
class_mode = "categorical"
)

# no data  augmentation
train_gen <- flow_images_from_directory(
directory = "data_indoor/train",
generator = image_data_generator(rescale = 1/255),
target_size = c(64,64),
batch_size = batch,
class_mode = "categorical"
)
```

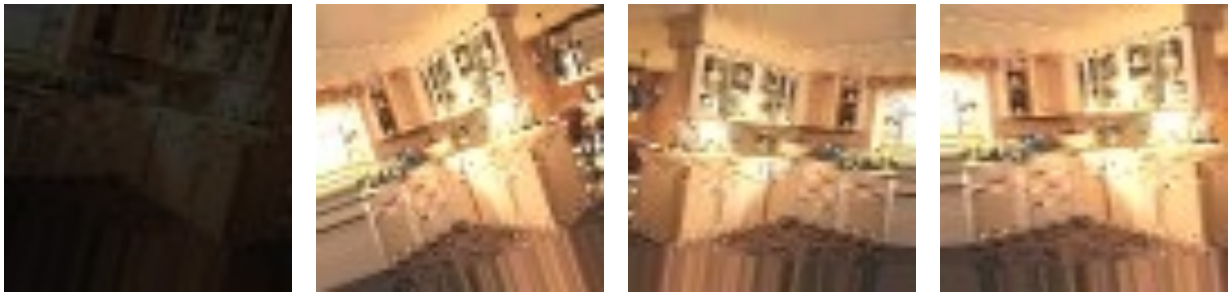
The idea behind data augmentation is the following. Given a theoretically infinite amount of data, the model would be exposed to every possible aspect of the data generating process, hence it would never over fit and will always be able to generalize well. Data augmentation generates additional training data from the available training samples, by augmenting the samples using a number of random transformations that provide realistic-looking images. The aim is that at training time, the model will never encounter the exact same image twice. This helps expose the model to more aspects of the data generating process and generalize better, thus introducing regularization

For data augmentation I have considered the below arguments:

- Apart from Range of width, shift,shear and zoom I have also considered a horizontal flip for the dataset. Since the data set consists of indoor pictures vertical flip does not fit in for this model.
- I have set the standard rotation range of 20 to capture different angle changes in the image considered.
- I have also included a brightness range argument to the data augmentation to account for different lightings in the rooms.

```
# data augmentation
data_augment <-
   image_data_generator(rescale = 1/255,
   rotation_range = 20,
   width_shift_range = 0.2,
   height_shift_range = 0.2,
   shear_range = 0.2,
   zoom_range = 0.2,
   horizontal_flip = TRUE,
   brightness_range = c(0.2, 1.5),
   fill_mode = "nearest"
)

# check effect of data augmentation - change path to see different images
img_array <- image_to_array( image_load("data_indoor/train/kitchen/cdmc1153.jpg",
   target_size = c(64,64))
   )
img_array <- array_reshape(img_array, c(1, 64, 64, 3))augmentation_generator <-
flow_images_from_data(
   img_array,
   generator = data_augment,
   batch_size = 1 )

par(mfrow = c(1, 4), mar = rep(0.5, 4))
```

```
for (i in 1:4) {
  batch1 <- generator_next(augmentation_generator)
  plot(as.raster(batch1[1,,,]))
}
```



```
#generator with data augmentation
train_gen_aug <- flow_images_from_directory(directory = "data_indoor/train",
                                            generator = data_augment,
                                            target_size = c(64, 64),
                                            batch_size = 20,
                                            class_mode = "categorical")
```

```
#Defining Keras Optimizer to resolve v2.11+ optimizer issue on M1 Mac
optimizer <- tf$keras$optimizers$legacy$Adam(learning_rate = 0.001)
```

Deep neural network

Model 1:

We have 1713 images in the training data folder and 851 images in the validation data folder.

As mentioned above, we set the batch size to 20 to make sure to have at least 1-2 images per class.

Hence,to ensure that all images are covered by the batch size multiplied by the steps, we set appropriatevalues for the steps arguments.

```
# Define batch size and step sizes for all models
batch <- 20
ep <- 50
val_steps <- round(860/batch)
steps_epoch <- round(1720/batch)
```

- The model architecture used is a sequential deep neural network (DNN) that has multiple fully con- nected layers with ReLU activation function, followed by a dropout layer, and an output layer with a softmax activation function.
- The input to the model is a flattened 64x64x3 image, where 64x64 is the image size and 3 is the number of color channels (RGB).
- The model has three hidden layers with 256, 128, and 64 units, respectively. Each hidden layer is followed by a dropout layer to prevent overfitting.
- The output layer has 10 units with a softmax activation function, which produces a probability distri- bution

over 10 classes.

- Regularization is applied to the fully connected layers using L2 regularization with a specified lambdavalue of 0.01.

```
# set default flags
FLAGS <-
flags( flag_numeric("dropout", 0.5),
flag_numeric("lambda", 0.01),
flag_numeric("lr", 0.01)
)

# Define model architecture
model_dnn <- keras_model_sequential() %>%

  # Flatten input
  layer_flatten(input_shape = c(64, 64, 3)) %>%

  # Fully connected layers
  layer_dense(units = 256, activation = "relu",name = "layer_1", kernel_regularizer
                = regularizer_l2(FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$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) %>%
  # Output layer
  layer_dense(units = 10, activation = "softmax",name = "layer_out") %>%
  # Compile model
  compile(loss = "categorical_crossentropy",optimizer
    = optimizer,
    metrics = c("accuracy"))

# Print summary of model architecture
summary(model_dnn)
```

```
## Model: "sequential"
## ________________________________________________________________________________
```

```
## Layer (type)                 Output Shape              Param #
## =================================================================
## flatten (Flatten)            (None, 12288)             0
## layer_1 (Dense)              (None, 256)               3145984
## dropout_2 (Dropout)          (None, 256)               0
## layer_2 (Dense)              (None, 128)               32896
## dropout_1 (Dropout)          (None, 128)               0
## layer_3 (Dense)              (None, 64)                8256
## dropout (Dropout)            (None, 64)                0
## layer_out (Dense)            (None, 10)                650
## =================================================================
## Total params: 3,187,786
## Trainable params: 3,187,786##
Non-trainable params: 0
## _________________________________________________________________
```

```
# Train the model
DNN1 <- model_dnn %>%
  fit_generator(generator = train_gen,
  steps_per_epoch =steps_epoch,
  epochs = ep,
  validation_data =validation_generator,
  validation_steps =val_steps,
  verbose = 2)

# Evaluate the model
score_dnn <- model_dnn %>% evaluate(validation_generator,
  steps = val_steps
)

# Print accuracy and loss on validation set
cat("Validation loss:", score_dnn[[1]], "\n")
```

```
## Validation loss: 2.05512
```

```
cat("Validation accuracy:", score_dnn[[2]], "\n")
```

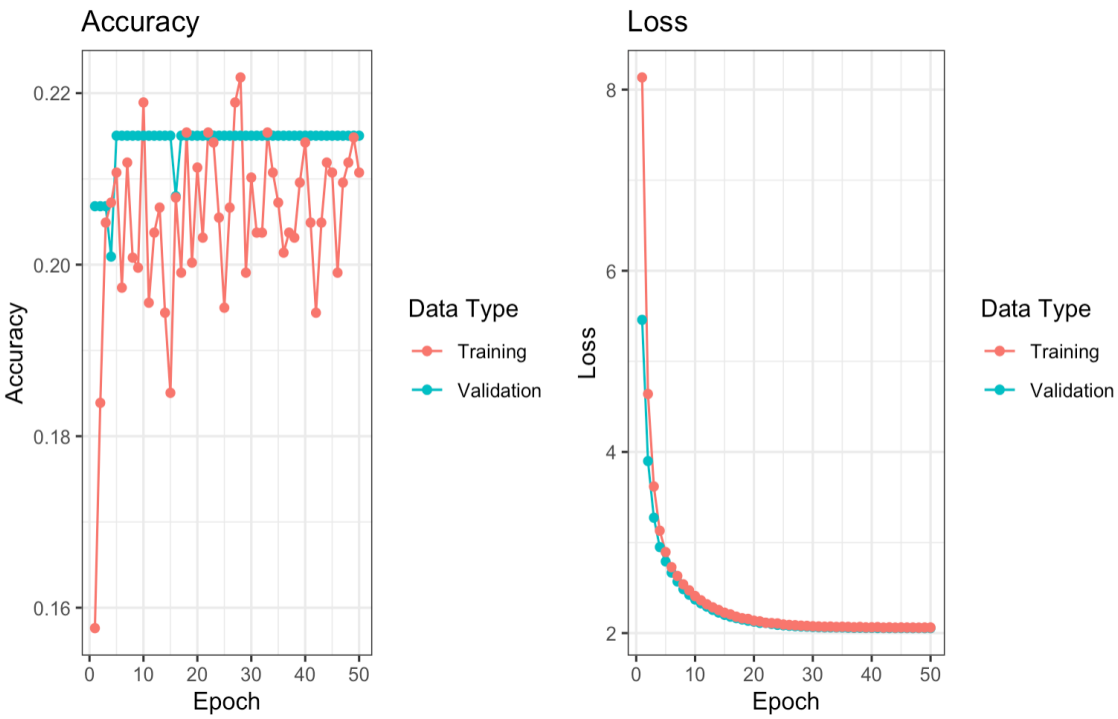
```
## Validation accuracy: 0.2150411
```

```
saveRDS(DNN1, file ="DNN.Rdata")
```

From the DNN Model obtained above, we get the Validation loss as 2.055 and Validation Accuracy as 0.215 for our model trained for 50 epochs. The training data accuracy is around 0.2 which tells us that the model performs poorly on the training data and the model is under fitting.

```
# Plot the accuracy and loss graphs

DNN_accuracy<- ggplot() +
  geom_line(aes(x = 1:50, y = DNN1$metrics$val_accuracy, color = "Validation")) +
  geom_line(aes(x = 1:50, y = DNN1$metrics$accuracy, color = "Training")) +
   geom_point(aes(x = 1:50, y = DNN1$metrics$val_accuracy, color = "Validation"))+
```

```
  geom_point(aes(x = 1:50, y = DNN1$metrics$accuracy, color = "Training"))+
  + labs(x = "Epoch", y = "Accuracy",color = "Data Type") +
  ggtitle("Accuracy") +
  theme_bw()


DNN_loss<- ggplot() +
  geom_line(aes(x = 1:50, y = DNN1$metrics$val_loss, color = "Validation")) +
  geom_line(aes(x = 1:50, y = DNN1$metrics$loss, color = "Training")) +
  geom_point(aes(x = 1:50, y = DNN1$metrics$val_loss, color = "Validation")) +
  geom_point(aes(x = 1:50, y = DNN1$metrics$loss, color = "Training")) +
  labs(x = "Epoch", y = "Loss",color = "Data Type") +
  ggtitle("Loss") +
  theme_bw()

# Combine the two plots side by side
grid.arrange(DNN_accuracy, DNN_loss, ncol = 2)
```



The plot confirms that the model is under fitting and is unable to capture the relationship between the image input properly to determine the correct output of the indoor setting due to the complexity of the data. I shall now be implementing CNN models for further analysis.

Convolutional Neural Network

To build a Convolutional neural network, we require configuring the layers of the model as follows:

1. Convolution Layer: A convolution layer extracts features from an image or a part of an image. We arespecifying three parameters here:

+ Filters – This is the number of filters that will be used in the convolution. E.g 64.

+ Kernel size – The length of the convolution window. E.g. (3,3) or (5,5).

+ Activation – This refers to the function of the regularizer. For example, ReLU, Leaky ReLU, Tanh, and Sigmoid.We shall stickto ReLU for our model.

2. Pooling Layer: This layer is used to reduce the size of an image. We shall use a constant pool size of (2,2).

3. Flatten Layer: This layer reduces an n-dimensional array to a single dimension.

4. Dense Layer: This layer is fully connected, which means that all of the neurons in the present layer arelinked to the next layer. For our model, we have 128 neurons in the first dense layer and 10 neurons in the second dense layer.

5. Dropout Layer: To prevent the model from overfitting, this layer ignores a set of neurons (randomly).Before we begin the task we must consider a few points:

- We shall compare different models to compare their complexity for the small training set
- The images are represented as tensors with a width and height of 64 x 64 pixels, so all sizes in the network are defined in relation to this dimension.
- As the number of layers in the convolutional neural network (CNN) increases, the number of filters in the convolutional layers also increases to create a larger number of feature maps and to capture more complex features as the network becomes deeper.
- For an input tensor of size 64 x 64 in a CNN,we shall choose the max pooling size as (2, 2), which down samples the feature maps by a factor of 2 in both dimensions. This reduces the spatial dimensions ofthe feature maps by half, while retaining the most important information.

Model 1: CNN Model with 2 convolutional layers

In the given model, the first convolutional layer has a kernel size of 5x5 and 64 filters, and the second convolutional layer has a kernel size of 3x3 and 128 filters. These values seem reasonable for a 64x64 inputimage, and the use of multiple convolutional layers with decreasing kernel sizes and increasing filter values can help capture more complex features in the image.

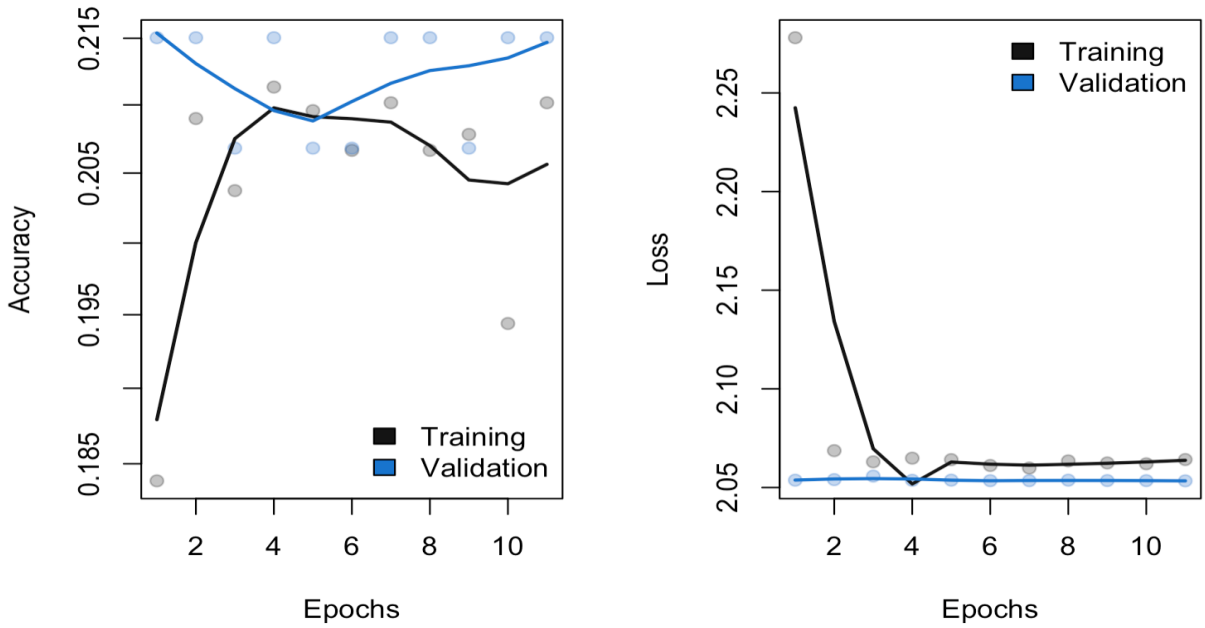
```
model_1 <- keras_model_sequential() %>%
  #  convolutional  layers
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), activation = "relu",name="layer_1",
                input_shape = c(64, 64, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu", name="layer_2") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%

  # fully  connected  layers
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%

  #Output Layer
  layer_dense(units = 10, activation = "softmax",name = "layer_out") %>%
```

```
#Compiling
compile(loss = "categorical_crossentropy",
metrics = "accuracy",
optimizer = optimizer
)
```

```
fit_1 <- model_1 %>% fit_generator( train_gen,
                                    steps_per_epoch = steps_epoch,
                                    epochs = ep,
                                    validation_data = validation_generator,
                                    validation_steps = val_steps,
                                    callbacks = list(callback_early_stopping(monitor = "val_accuracy",
                                    patience = 10,restore_best_weights = TRUE)))
```

Plotting the Accuracy and loss for model 1:

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

# check learning curves
out <- cbind(fit_1$metrics$accuracy,
             fit_1$metrics$val_accuracy,
             fit_1$metrics$loss,
             fit_1$metrics$val_loss)

cols <- c("gray8", "dodgerblue3")

par(mfrow = c(1,2))

# accuracy
matplot(out[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",col =
          adjustcolor(cols, 0.3), log = "y")

matlines(apply(out[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)

legend("bottomright", legend = c("Training", "Validation"),fill = cols, bty = "n")

# loss
matplot(out[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",col =
          adjustcolor(cols, 0.3))
matlines(apply(out[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)

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



From the CNN Model 1 with 2 convolution layers and no data augmentation we can see that there is a large gap between the error where the data overfits and intersects at one point and then converges again stating and the network might be too complex for the training data. We shall try fitting a model with more convolution layers and with data augmentation or batch normalization to check the accuracy of the model.

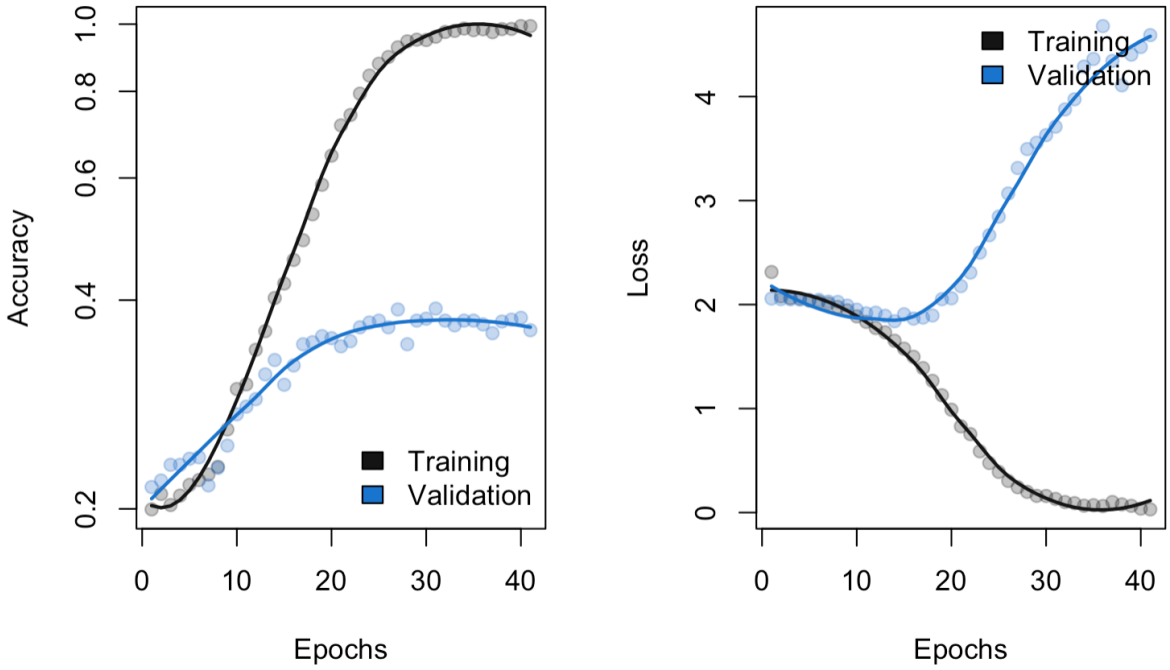
Model 2: CNN Model with 3 convolution layers

```
model_2 <- keras_model_sequential() %>%
  # convolutional layers
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), activation = "relu",name = "layer_1",
                input_shape = c(64, 64, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu",name = "layer_2")
                %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 256, kernel_size = c(3, 3), activation = "relu",name = "layer_3",)
                %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%

  # fully connected layers
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = "softmax", name = "layer_out") %>%
```

```
  # compiling the  model
  compile(loss = "categorical_crossentropy",
              metrics = "accuracy",
             optimizer = optimizer)
```

```
fit_2 <- model_2 %>% fit_generator( train_gen,
                                steps_per_epoch = steps_epoch,
                                epochs = ep,
                                validation_data = validation_generator
                                ,validation_steps = val_steps,
                                callbacks = list(callback_early_stopping(monitor = "val_accuracy",
                                patience = 10,restore_best_weights = TRUE)))
```

```
# Plot the accuracy and loss graphs
out <- cbind(fit_2$metrics$accuracy,
                    fit_2$metrics$val_accuracy,
                    fit_2$metrics$loss,
                    fit_2$metrics$val_loss)

cols <- c("gray8", "dodgerblue3")

par(mfrow = c(1,2))

# accuracy
matplot(out[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",col =
             adjustcolor(cols, 0.3), log = "y")

matlines(apply(out[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)

legend("bottomright", legend = c("Training", "Validation"),
            fill = cols, bty = "n")

# loss
matplot(out[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",col =
              adjustcolor(cols, 0.3))
matlines(apply(out[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)

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



For the CNN Model with 3 convolutional layers and no data augmentation, the plot confirms that even know the model training accuracy goes upto 0.95 there is an overfitting of the training data as our Model 1.

Model 3: CNN Model with 3 convolution layers with data augmentation

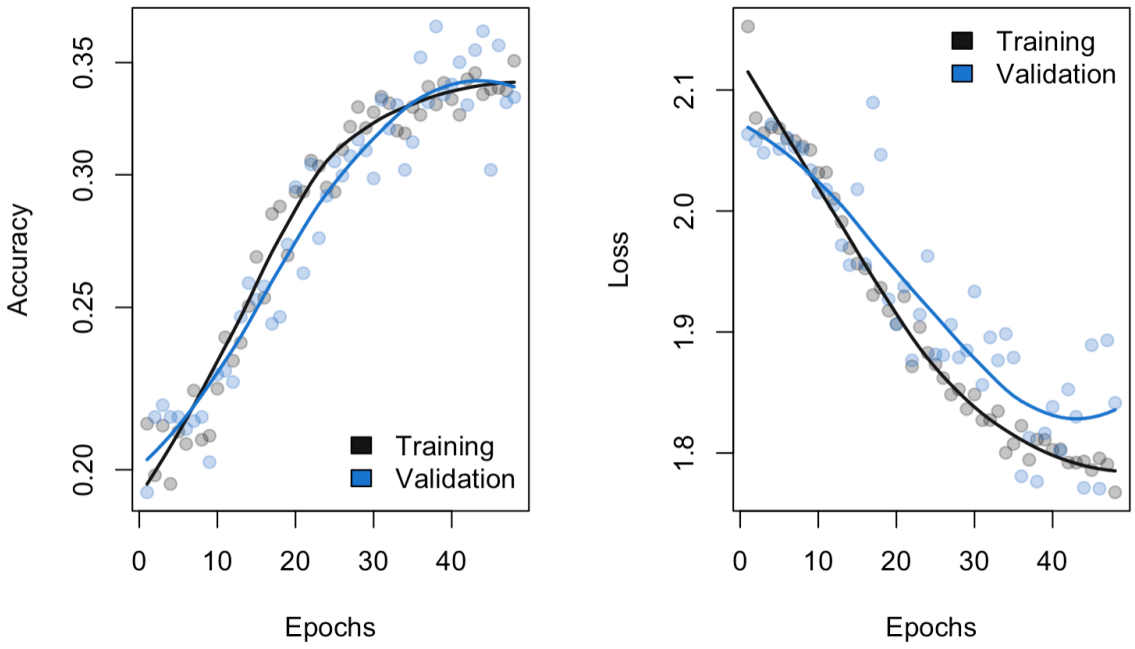
The data considered here have a relatively small number of training samples (1,713). Because of this, the model overfits as seen from the previous 2 CNN Models, since there are too few samples to learn from and hence it won't be able to generalize well to new data. To solve the problem, we use data augmentation, a specific computer vision technique widely used when processing images with deep learning models.
We shall use the 'train_gen_aug' dataset sample for this model.

```
model_3 <- keras_model_sequential() %>%
  #  convolutional  layers
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), activation = "relu",name =
  "layer_1",input_shape = c(64, 64, 3)) %>% layer_max_pooling_2d(pool_size = c(2, 2))
  %>%
  layer_conv_2d(filters = 128, kernel_size = c(4, 4), activation = "relu",name = "layer_2")
                %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 256, kernel_size = c(3, 3), activation = "relu",name = "layer_3")
                %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%

  # fully  connected  layers
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%

  #Output  layer
  layer_dense(units = 10, activation = "softmax", name = "layer_out") %>%

  #Compiling  the  model
  compile(
  loss = "categorical_crossentropy",
  metrics = "accuracy",
  optimizer = optimizer
  )
```

```
fit_3 <- model_3 %>% fit(train_gen_aug,
                         steps_per_epoch = steps_epoch,
                         epochs = ep,
                         validation_data = validation_generator,
                         validation_steps = val_steps,
                         callbacks = list(callback_early_stopping(monitor = "val_accuracy",
                         patience = 10, restore_best_weights = TRUE)))
```

```
saveRDS(fit_3, "model3.rds")
```

```
# Plot the accuracy and loss graphs
out <- cbind(fit_3$metrics$accuracy,
                    fit_3$metrics$val_accuracy, fit_3$metrics$loss,
                    fit_3$metrics$val_loss)

cols <- c("gray8", "dodgerblue3")

par(mfrow = c(1,2))

# accuracy
matplot(out[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",col =
            adjustcolor(cols, 0.3), log = "y")

matlines(apply(out[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)

legend("bottomright", legend = c("Training", "Validation"),
           fill = cols, bty = "n")

# loss
matplot(out[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",col =
            adjustcolor(cols, 0.3))
matlines(apply(out[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)

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



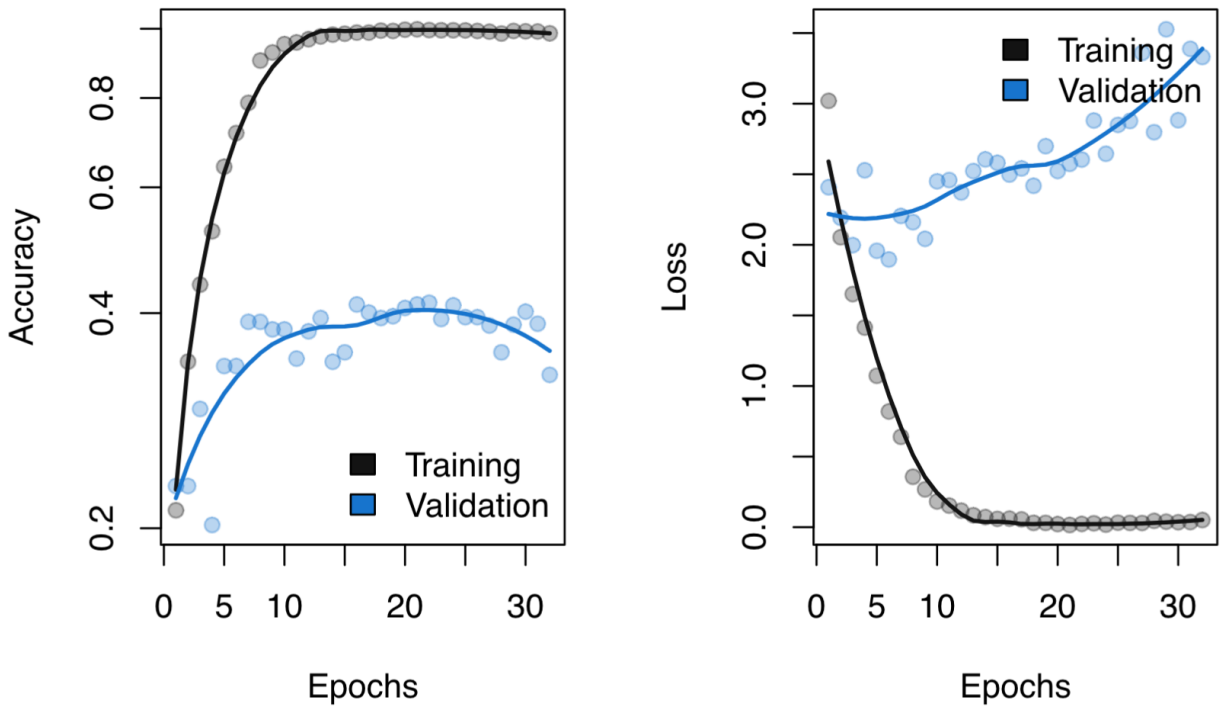
Model 4: 2 Convolution layers with Batch normalization

```
model_4 <- keras_model_sequential() %>%
  #  convolutional  layers
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), activation = "relu",input_shape = c(64, 64,
  3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_batch_normalization() %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_batch_normalization() %>%

  # fully  connected  layers
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_batch_normalization() %>% layer_dropout(rate = 0.5)
  %>%
  #Output Layer
  layer_dense(units = 10, activation = "softmax") %>%
  #compiling the  model
  compile(
  loss = "categorical_crossentropy",metrics
  = "accuracy",
  optimizer = optimizer)
```

In this modified model, batch normalization layers are added after each convolutional and fully connected layer, and a dropout layer is added after the first fully connected layer.

```
fit_4 <- model_4 %>% fit(train_gen,
                         steps_per_epoch = steps_epoch,
                         epochs = ep,
                         validation_data = validation_generator,
                         validation_steps = val_steps,
                         callbacks = list(callback_early_stopping(monitor = "val_accuracy"
                         ,patience = 10,restore_best_weights = TRUE)))
```

```
# Plot the accuracy and loss graphs
out <- cbind(fit_4$metrics$accuracy,
             fit_4$metrics$val_accuracy, fit_4$metrics$loss,
             fit_4$metrics$val_loss)

cols <- c("gray8", "dodgerblue3")

par(mfrow = c(1,2))

# accuracy
matplot(out[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",col =
          adjustcolor(cols, 0.3), log = "y")

matlines(apply(out[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)

legend("bottomright", legend = c("Training", "Validation"),
       fill = cols, bty = "n")

# loss
matplot(out[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",col =
          adjustcolor(cols, 0.3))
matlines(apply(out[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)

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



In this model the accuracy goes up to 0.34 from 0.17 and the loss reduces significantly for the Training data.
There is a huge gap between the two and it means that the data overfits.
We can see that adding batch normalization to the convolution layers does not increase the model performance much.

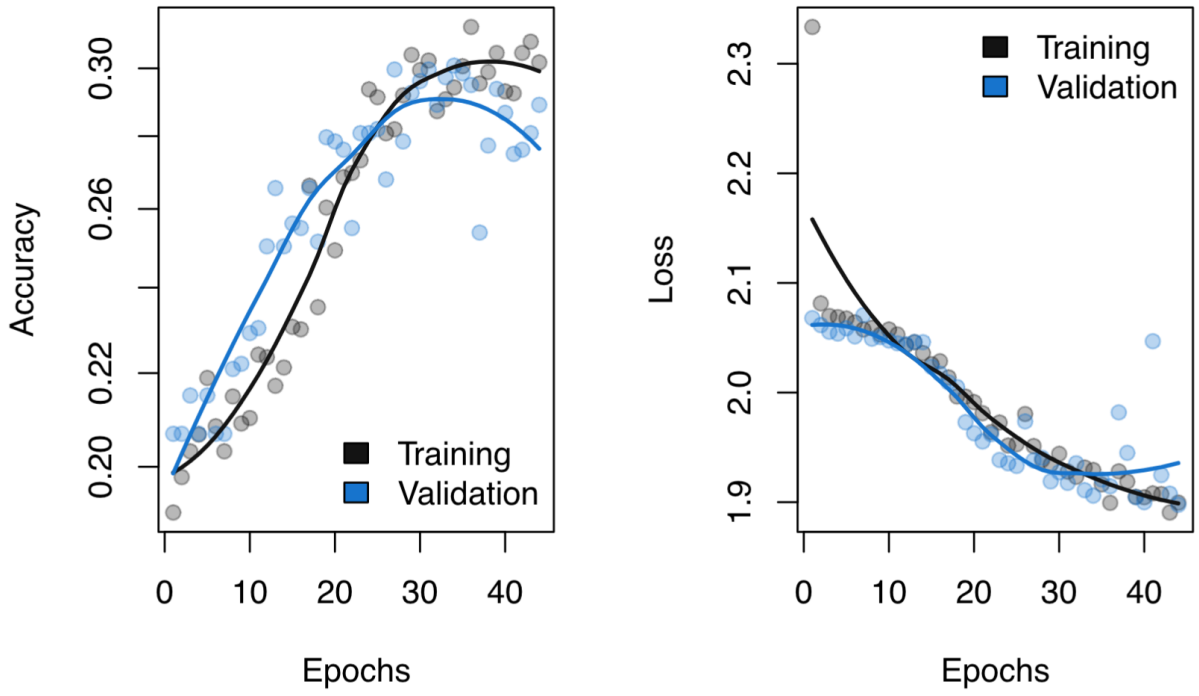
Model 5: 2 Convolution layers and 2 dense layers with data augmentation

```
model_5 <- keras_model_sequential() %>%
  #  convolutional  layers
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), activation = "relu",name="layer_1",
                input_shape = c(64, 64, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu",name="layer_2")
                %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%

  # fully  connected  layers
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(rate = 0.4) %>%
  #Output layer
  layer_dense(units = 10, activation = "softmax",
              name="layer_out") %>%
```

```
  # compiling the  model
  compile(
  loss = "categorical_crossentropy",metrics
  = "accuracy",
  optimizer = optimizer
  )
```

```
fit_5 <- model_5 %>% fit(train_gen_aug,
                                steps_per_epoch = steps_epoch,
                                epochs = ep,
                                validation_data = validation_generator,
                                validation_steps = val_steps,
                                callbacks = list(callback_early_stopping(monitor = "val_accuracy"
                                ,patience = 10, restore_best_weights = TRUE)))
```

```
# Plot the accuracy and loss graphs
out <- cbind(fit_5$metrics$accuracy,
                    fit_5$metrics$val_accuracy,
                    fit_5$metrics$loss,
                    fit_5$metrics$val_loss)

cols <- c("gray8", "dodgerblue3")

par(mfrow = c(1,2))

# accuracy
matplot(out[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",col =
          adjustcolor(cols, 0.3), log = "y")

matlines(apply(out[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)

legend("bottomright", legend = c("Training", "Validation"),
          fill = cols, bty = "n")

# loss
matplot(out[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",col =
          adjustcolor(cols, 0.3))
matlines(apply(out[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)

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

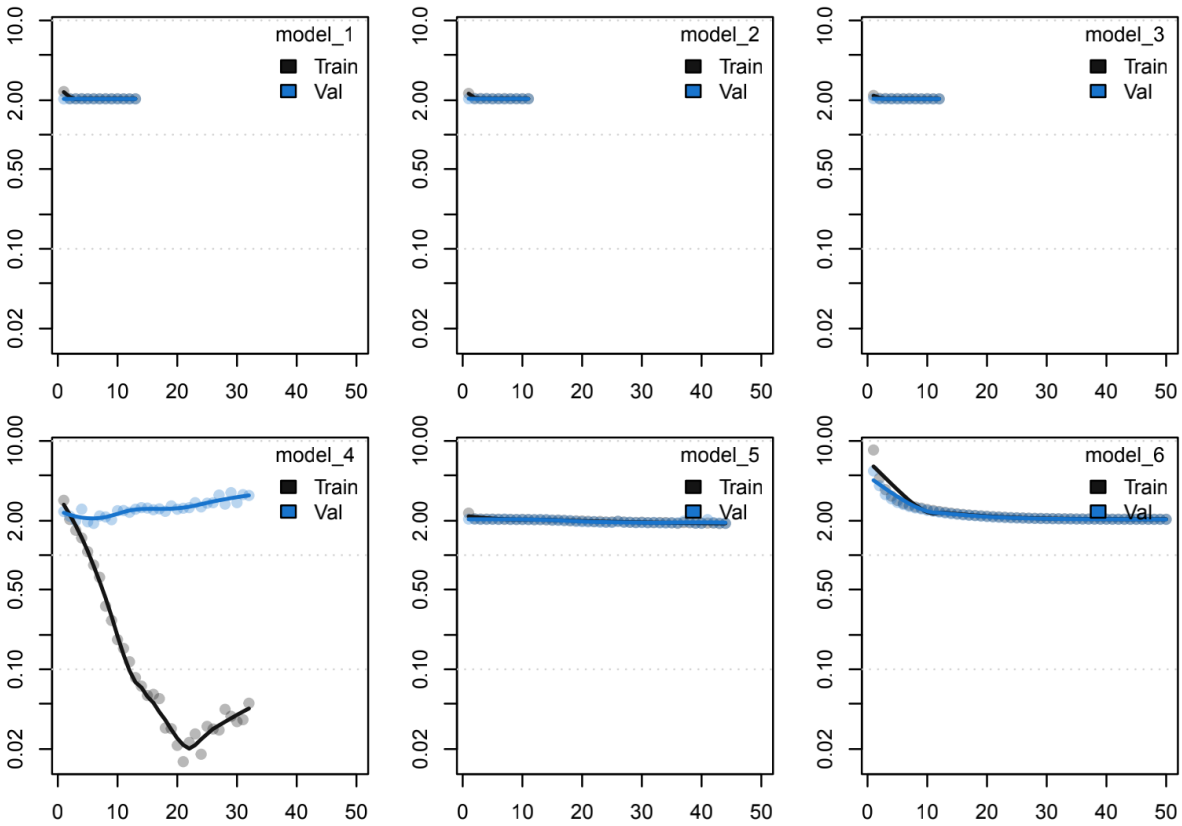


We shall now look at the accuracy and loss plots for all models to compare their performances. We have 5 CNN model and 1 DNN Model for our data.

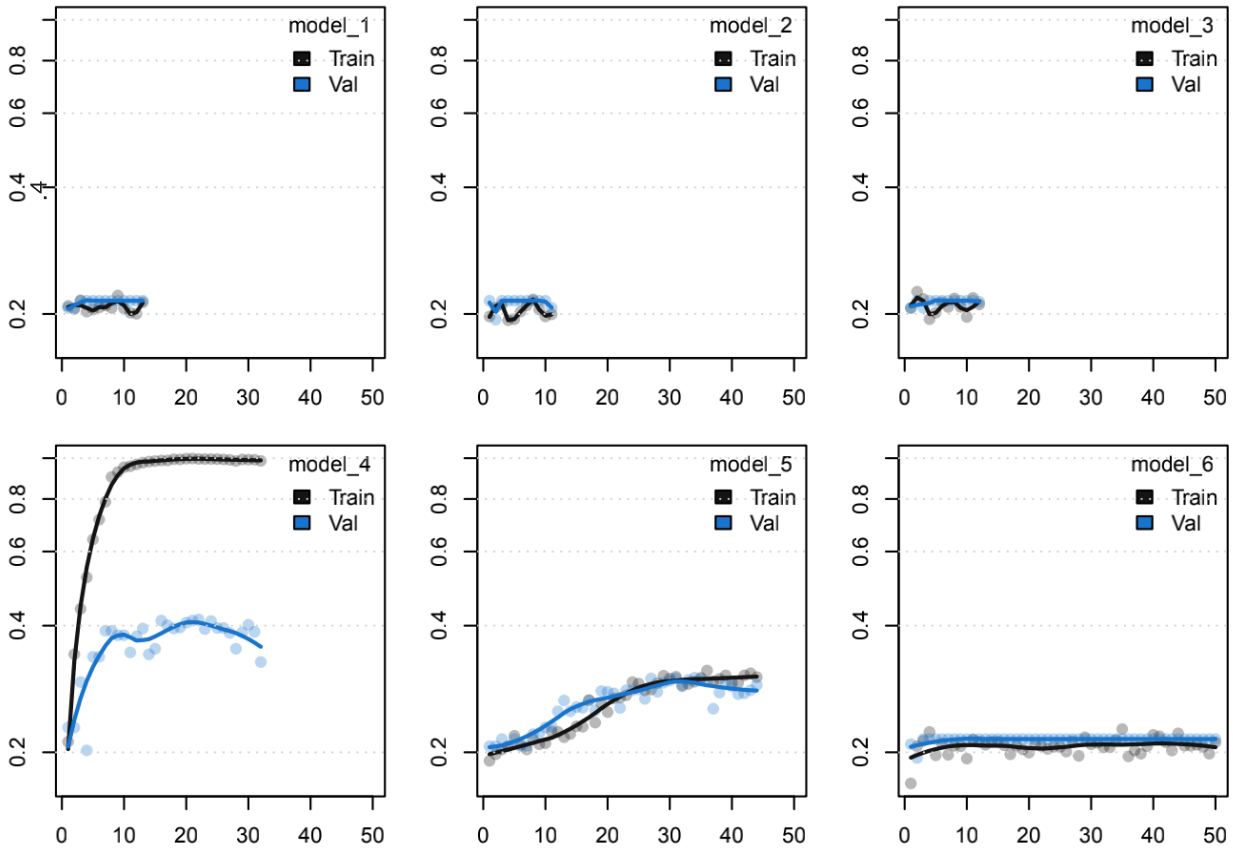
```
# store  outputs
out_loss <- list(fit_1$metrics$loss, fit_1$metrics$val_loss,
                 fit_2$metrics$loss, fit_2$metrics$val_loss,
                 fit_3$metrics$loss, fit_3$metrics$val_loss,
                 fit_4$metrics$loss, fit_4$metrics$val_loss,
                 fit_5$metrics$loss, fit_5$metrics$val_loss,
                 DNN1$metrics$loss, DNN1$metrics$val_loss)

out_acc<-list(fit_1$metrics$accuracy,fit_1$metrics$val_accuracy,
                 fit_2$metrics$accuracy,fit_2$metrics$val_accuracy,
                 fit_3$metrics$accuracy,fit_3$metrics$val_accuracy,
                 fit_4$metrics$accuracy,fit_4$metrics$val_accuracy,
                 fit_5$metrics$accuracy,fit_5$metrics$val_accuracy,
                 DNN1$metrics$accuracy, DNN1$metrics$val_accuracy)
```

```
cols <- c("gray8", "dodgerblue3") # set the colors for the plot
rg_acc <- range(unlist(out_acc)) # get the range for accuracy values
rg_loss <- range(unlist(out_loss)) # get the range for loss values
J <- length(out_loss) # get the length of the loss values

# create a function for smoothing lines
smooth_line <- function(y) {
   x <- 1:length(y) # set the x-axis values
```

```
  out <- predict( loess(y ~ x, span = 0.5) ) #apply loess smoothing to y values
  return(out)
}
ind <- matrix(1:J, nrow = J/2, byrow = TRUE) #create an index for subsetting the loss values
par(mfrow = c(2,3), mar = c(3,3, 0.4, 0.6)) # set the layout and margins for the plot

# loop through the index and plot the loss values
for ( i in 1:(J/2) ) {
  # convert to a matrix, append NAs for make lengths all equal
  out <- matrix( unlist(out_loss[ind[i,]]), ncol = 2 ) # subset the loss values and convert to a matrix
  out <- rbind( out, matrix(rep(NA, (ep - nrow(out))*2), ncol = 2) )#add NAs to make matrix lengths equal

  matplot(1:ep, out, pch = 19, ylab = "Loss", xlab = "Epochs",
          col = adjustcolor(cols, 0.3), ylim = rg_loss, log = "y")# plot the loss values
          matlines(apply(out, 2, smooth_line), lty = 1, col = cols, lwd = 2)# plot the smoothed lines
          legend("topright", legend = c("Train", "Val"),
          fill = cols, bty = "n", title = paste0("model_", i))# add a legend to the plot
          grid(ny = NULL, nx = NA) }  #add gridlines to the plot
```



```
#Set the layout and margins for the plot
par(mfrow = c(2,3), mar = c(3,3, 0.4, 0.6))

for ( i in 1:(J/2) ) {
  #matrix that contains the training and validation accuracy for each epoch
  out <- matrix( unlist(out_acc[ind[i,]]), ncol = 2 )#subset the accuracy values and convert to a matrix
  out <- rbind( out, matrix(rep(NA, (ep - nrow(out))*2), ncol = 2) )#add NAs to make matrix lengths equal
  #plots the accuracy for each epoch on the y-axis against the epoch number on the x-axis
  matplot(out, pch = 19, ylab = "Accuracy", xlab = "Epochs",
          col = adjustcolor(cols, 0.3), ylim = rg_acc, log = "y")
  #add a smoothed line
  matlines(apply(out, 2, smooth_line), lty = 1, col = cols, lwd = 2)legend("topright",
  legend = c("Train", "Val"),
          fill = cols, bty = "n", title = paste0("model_", i)) #Add a legend#add gridlines to
  the plot
  grid(ny = NULL, nx = NA) }
```



Analysis:

Comparing the above plots we can conclude that out of the six models implemented:

1. Model with only dense layers and no convolution layer has under fitting of the data (model_6) and CNN has an overall better performance for the data set available.

2. model_1, model_2 and model_4 significantly overfit indicating that the network might be too complex for the dataset.

3. Using batch normalization (model_4) does not prevent overfitting in the CNNs and is not the bestapproach for this data set.

4. CNNs trained with data augmentation (model_3,model_5) generally performed better.

5. Adding a new dense layer to the CNN model with data augmentation does not improve the model performance (model_5).

6. Overall, the CNN with 3 convolutional layers and one dense layer trained with data augmentation (model_3) had the best performance on the validation set and was shall be used to predict the test data.

Using the test data to evaluate the predictive performance of the best model.

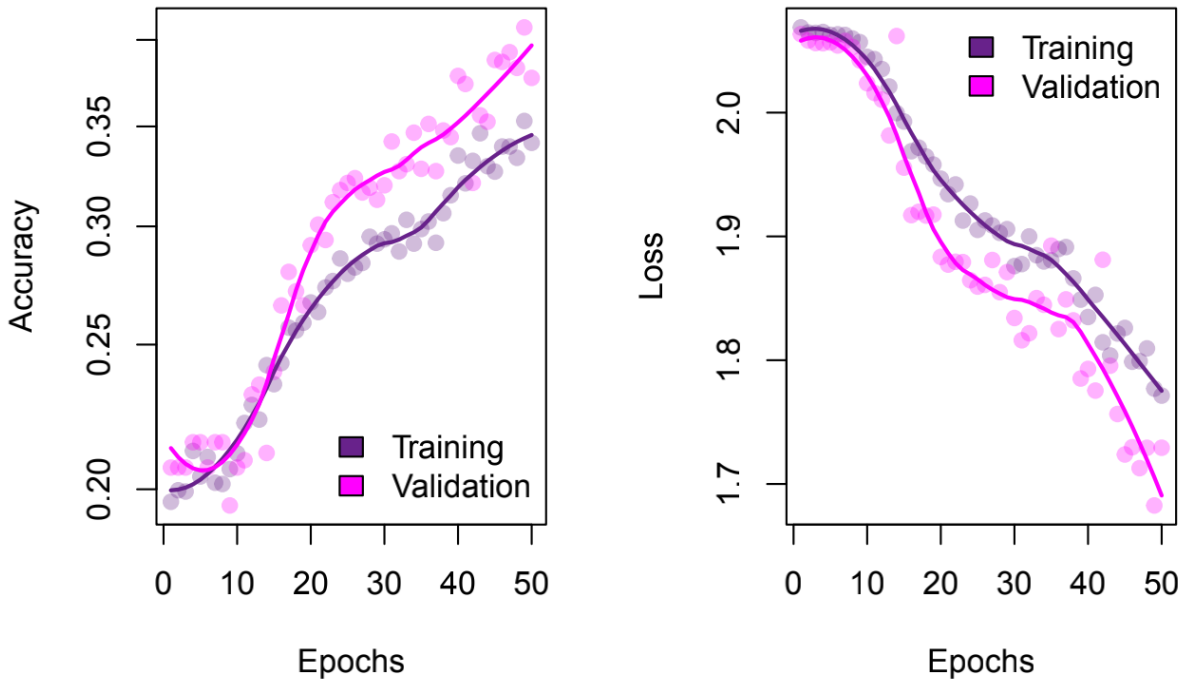
There are 851 images in the test data. In this case, for the test generator we set a batch size of 1, whilefor the fit generator a step of 851. In such way, all the images available in the test folder will be used for testing. This avoids the introduction of bias in the estimate of the predictive performance.

I have clubbed the training and validation data set to a new folder in the data_indoor directory under "train&validation" to test the predictive performance of the model selected.

```
#new validation  set
steps_epoch <- round((1720 + 860)/20)

# reset  train  generator
train_gen_aug <- flow_images_from_directory(directory = "data_indoor/train&validation/",
                                            generator = data_augment,
                                            target_size = c(64, 64),
                                            batch_size = 20,
                                            class_mode = "categorical")


#Test generator
test_generator <- flow_images_from_directory(directory = "data_indoor/test/",
                                             generator = image_data_generator(rescale = 1/255),
                                             target_size = c(64, 64),
                                             batch_size = 1,
                                             class_mode = "categorical",
                                             shuffle = FALSE)
```

```
#Fit the  selected  model
fit <- model_3 %>% fit(train_gen_aug,steps_per_epoch = steps_epoch,
                       epochs = ep,
                       validation_data = test_generator,
                       validation_steps = 851,
                       callbacks = list(callback_early_stopping(monitor = "val_accuracy",patience =
                       10,restore_best_weights = TRUE)))
```

```
# Plot the accuracy and loss graphs
out <- cbind(fit$metrics$accuracy,
                    fit$metrics$val_accuracy,
                    fit$metrics$loss,
                    fit$metrics$val_loss)

cols <- c("darkorchid4", "magenta")

par(mfrow = c(1,2))

# accuracy
matplot(out[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",col =
          adjustcolor(cols, 0.3), log = "y")

matlines(apply(out[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)

legend("bottomright", legend = c("Training", "Validation"),
          fill = cols, bty = "n")

# loss
matplot(out[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",col = adjustcolor(cols, 0.3))
matlines(apply(out[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)


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



From the above plot we can confirm that the model is an overall good fit with sufficient gap between the learning curves for accuracy and loss plots. We must now compute the accuracy metrics. We take 851 steps to consider all images in the test data set.

```
# Get true classes from the test generator
true_classes <- test_generator$classes

# Get the class labels from the test generator
labels <- names(test_generator$class_indices)
```

```
# Predict the class probabilities for the test set using the trained model
pred <- model_3 %>% predict_generator(test_generator, steps = 851)
```

```
# Get the predicted class for each sample
pred_class <- factor(max.col(pred), labels = )

# Tabulate the predicted vs true classes
class <- factor(true_classes, labels = labels)tab <-
table(class, pred_class)

# Calculate class-specific accuracy and add to the table
cbind(tab, c_acc = diag(tab)/rowSums(tab))
```

##	bathroom	bedroom	closet	corridor	dining_room	garage	kitchen	living_room	c_acc
## bathroom	7	17	0	5	0	0	8	12	0.14285714
## bedroom	1	43	9	12	2	0	33	65	0.41212121
## children_room	0	2	3	5	0	0	5	13	0.07142857
## closet	0	1	22	4	0	0	2	4	0.12121212
## corridor	2	4	3	66	1	0	4	6	0.01162791
## dining_room	0	2	7	2	8	0	10	39	0.25000000
## garage	0	1	7	8	0	0	4	5	0.16000000
## kitchen	1	16	15	12	4	0	80	55	0.30054645
## living_room	0	17	17	5	3	1	38	95	0.39777273
## stairs	1	3	5	7	4	0	5	13	0.05263158

This calculates the class-specific accuracy by dividing the diagonal elements of the table (i.e., the counts ofcorrectly predicted samples) by the row sums (i.e., the total number of samples for each true class. Bedroom and the living room have the highest class accuracy.

```
# test accuracy
sum(diag(tab))/sum(tab)
```

```
## [1] 0.3374853
```

```
# proportion of "false positives"
1 - diag(tab)/colSums(tab)
```

```
##bathroom   bedroom   closet    corridor   dining_room garage   kitchen    living_room
## 0.4166667 0.5943396 0.9659091 0.9582540  0.9674555 1.0000000 0.9788360 0.9733333
```

- The model's accuracy test score is approximately 0.337 which is not a very good score for the model

performance and is an indication of a small training set.

- From the data of false positives, Closet,Corridor,and kitchen are generally easier to identify accurately.There are no samples classified for the children room, garage and stairs with no accuracy stating thatthere is a lack of training samples for this class.
- Additionally, there is a high percentage of images labeled as dining_room and living_room.