CNN Hyperparameter Tuning.

Siddhesh Bagwe

The given dataset consists of images of different indoor scenes. Our job is to find a model which can best predict the labels according to our label data. The steps to design a model are

1) Preparing the data for modelling.
2) Fitting different models for the data.
3) Evaluating the model performances and selecting the best model.
4) Testing the model chosen.

Preparing the dataset

The given dataset consists of different images. For fitting this to a model we need to properly prepare the data. The data is already divided into train, validation, test. We normalize each image in the input to convert it from a range of [0,255] to [0,1]. Next we define the target size for each image which in this case is (64,64).

Batch size is another important parameter to be considered when designing a Neural Network. The batch size is a number of samples processed before the model is updated. Here, according to the dataset we have set the batch size to 32 so as to make the learning faster. Also the size of our dataset is pretty small so a bigger batch size would not be feasible. 32 would be a optimal batch_ size as going lower than that will not provide a good estimate of the gradients.

```
train_dir <- "train"
validation_dir <- "validation"
test_dir <- "test"
train_val <- "train_val"

## Creating train and validation data
train_datagen <- image_data_generator(rescale = 1./255)
validation_datagen <- image_data_generator(rescale = 1./255)
test_datagen <- image_data_generator(rescale = 1./255)

train_generator <- flow_images_from_directory(
train_dir,
train_datagen,
target_size = c(64, 64),
batch_size = 32,
class_mode = "binary"
)

validation_generator <- flow_images_from_directory(
validation_dir,
validation_datagen,
target_size = c(64,64),
batch_size = 32,
class_mode = "binary"
)
```

The output shows us that the data consists of 1713 training images and 851 images for validation data. Next, we start designing different models to fit our data on.

Model Fitting.

1)DNN.

```
model_dnn <- keras_model_sequential() %>%
  layer_dense(units = 64, input_shape = c(64,64,3), activation = "relu", name
= "layer_1",
  kernel_regularizer = regularizer_l2(0.01)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 32, activation = "relu", name = "layer_2",
  kernel_regularizer = regularizer_l2(0.01)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 10, activation = "sigmoid") %>%
  compile(loss = "sparse_categorical_crossentropy", metrics = "accuracy",
  optimizer = optimizer_adam(learning_rate = 0.001),
)
summary(model_dnn)
```

```
## Model: "sequential"
##
_____________________________________________________________________________
___
##  Layer (type)                    Output Shape                    Param
#
##
=============================================================================
===
##  layer_1 (Dense)                 (None, 64, 64, 64)              256
##  dropout_1 (Dropout)             (None, 64, 64, 64)              0
##  layer_2 (Dense)                 (None, 64, 64, 32)              2080
##  dropout (Dropout)               (None, 64, 64, 32)              0
##  flatten (Flatten)               (None, 131072)                  0
##  dense_1 (Dense)                 (None, 128)
16777344
##  dense (Dense)                   (None, 10)                      1290
##
=============================================================================
===
## Total params: 16,780,970
## Trainable params: 16,780,970
## Non-trainable params: 0
##
________________________________________________________________________________
___
```

The model above is a DNN model with 7 hidden layers (2 DNN, 2 dropout, 3 fully connected). The input shape is (64,64,3) which indicates that the input data is a 3-channel image with dimensions of 64x64 pixels.

- First DNN layer has a width of 64 with ReLU activation function and L2 regularization with strength 0.01
- Second DNN layer has a width of 32, having the same ReLU activation function with L2 regularization.
- A dropout layer is added after each of the 2 layers with a rate of 0.4 to avoid overfitting.
- Next a flatten layer is added that flattens the output from the previous layer into a 1D vector. This is done to get the output as labels for the predictions.
- Next layer is a DNN layer with 512 output units and a ReLU activation function. This layer takes the inputs from flatten layer and tries to again learn from it.
- Final layer is a DNN layer with 10 output unit and a sigmoid activation function. This is the output layer which gives the prediction of the 10 classes.

The model is compiled with a sparse_categorical cross-entropy loss function, an accuracy metric, and the Adam optimizer with a learning rate of 0.001. The learning rate is kept low to improve the learning capability of the model

The summary function prints out a summary of the model architecture, including the number of parameters in each layer and the total number of trainable parameters.

```
set.seed(21200534)
fit_dnn <- model_dnn %>% fit(
train_generator,
steps_per_epoch = 1713/32,
epochs = 50,

validation_data = validation_generator,
validation_steps = 851/32
)
```

We fit the model compiled above using our dataset. the parameters needed while fitting the model are

- steps_per_epoch - Total number of steps taken in one epoch. We set it to 1713/32 which is the number of training samples / batch size. We do this to improve efficiency and randomization of the data
- epochs - An epoch in a neural network is the training of the neural network with all the training data for one cycle. Here we fit the model with 50 epochs.

- validation_steps - Total number of steps used for validation. We set this again to number of validation samples/batch_size

We use this particular values as it is a typical choice for steps_per_epoch and validation_steps. The epochs are set to 50 so that the model trains and learns from the data properly.

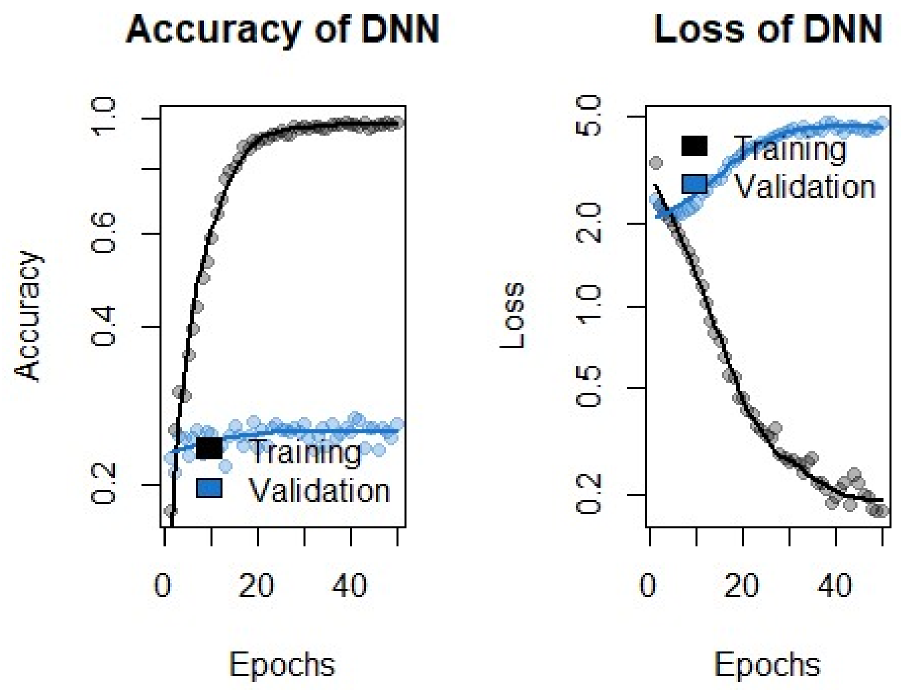
```
model_dnn %>% evaluate(train_generator)

##      loss  accuracy
## 0.1179833 0.9970812

model_dnn %>% evaluate(validation_generator)

##      loss  accuracy
## 4.6670618 0.2608696
```

The model fit gives us a training accuracy of 0.9970812 and loss of 0.1179833 Similarly we get a validation accuracy of 0.2608696 and loss of 4.6670618.

The training loss and accuracy indicate how well the model is performing on the training data, while the validation loss and accuracy give an indication of how well the model is generalizing to new, unseen data. In this case, the training accuracy is pretty high, while the validation accuracy is comparatively low which suggests that the model is overfitting as it is not able to perform on new data. We can check this using the accuracy and loss plots as well.

```
smooth_line <- function(y) {
x <- 1:length(y)
out <- predict( loess(y ~ x) )
return(out)
}
# check learning curves
out <- cbind(fit_dnn$metrics$accuracy,
fit_dnn$metrics$val_accuracy,
fit_dnn$metrics$loss,
fit_dnn$metrics$val_loss)
cols <- c("black", "dodgerblue3")
par(mfrow = c(1,2))
# accuracy
matplot(out[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3),
log = "y", main = "Accuracy of DNN")
matlines(apply(out[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = cols, bty = "n")

matplot(out[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",
col = adjustcolor(cols, 0.3),
log = "y", main = "Loss of DNN")
matlines(apply(out[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
```

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



The plots show the accuracy and loss of the training and validation data over the epochs. We can clearly see the difference between the training and validation accuracy as well as the loss showing the overfitting in the model.

We will try using the CNN model and check if the CNN model could better fit the data.

2. CNN.

```
model_cnn <- 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_dropout(rate = 0.4) %>%
layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
layer_max_pooling_2d(pool_size = c(2, 2)) %>%
layer_dropout(rate = 0.4) %>%
layer_conv_2d(filters = 256, kernel_size = c(3, 3), activation = "relu") %>%
layer_max_pooling_2d(pool_size = c(2, 2)) %>%
layer_dropout(rate = 0.4) %>%


# fully connected layers
```

```
layer_flatten() %>%
layer_dense(units = 512, activation = "relu") %>%
layer_dense(units = 50, activation = "sigmoid") %>%
#
# compile
compile(
loss = "sparse_categorical_crossentropy",
metrics = "accuracy",
optimizer = optimizer_adam(learning_rate = 0.001)
)
summary(model_cnn)
```

```
## Model: "sequential_1"
##
________________________________________________________________________________
___
##  Layer (type)                       Output Shape                     Param
#
##
================================================================================
===
##  conv2d_2 (Conv2D)                  (None, 60, 60, 64)               4864
##  max_pooling2d_2 (MaxPooling2D)     (None, 30, 30, 64)               0
##  dropout_4 (Dropout)                (None, 30, 30, 64)               0
##  conv2d_1 (Conv2D)                  (None, 28, 28, 128)              73856
##  max_pooling2d_1 (MaxPooling2D)     (None, 14, 14, 128)              0
##  dropout_3 (Dropout)                (None, 14, 14, 128)              0
##  conv2d (Conv2D)                    (None, 12, 12, 256)              295168
##  max_pooling2d (MaxPooling2D)       (None, 6, 6, 256)                0
##  dropout_2 (Dropout)                (None, 6, 6, 256)                0
##  flatten_1 (Flatten)                (None, 9216)                     0
##  dense_3 (Dense)                    (None, 512)
4719104
##  dense_2 (Dense)                    (None, 50)                       25650
##
================================================================================
===
## Total params: 5,118,642
## Trainable params: 5,118,642
## Non-trainable params: 0
##
________________________________________________________________________________
___
```

The model consists of three convolutional layers and three max-pooling layers. I decided to go with this configuration after trial and error using various layer configurations. The input to the model is a 64x64 pixel RGB image, and the output is a probability distribution over 10 classes.

The first convolutional layer has 64 filters of size 5x5 which results in a feature map of size 60x60, this is designed as the input shape is 64 as well. We will simultaneously keep increasing the number of filters in the next layers to improve the learning capability. A max-pooling layer with a pool size of 2x2 follows, which reduces the feature map's size to 30x30. A dropout layer is added so that we can prevent overfitting of the model.

The second convolutional layer has 128 filters of size 3x3, we reduce the kernel size so as to keep the computing time in check This results in a feature map of size 28x28. Another max-pooling layer follows, reducing the feature map size to 14x14. A dropout layer is applied after the max-pooling layer to reduce overfitting.

The third convolutional layer has 256 filters of size 3x3, resulting in a feature map of size 12x12. Another max-pooling layer follows, reducing the feature map size to 6x6. A dropout layer is applied after the max-pooling layer to reduce overfitting.

The output is flattened to a 1-dimensional vector after the max-pooling layer, which is then passed through two fully connected layers with 512 and 10 units, respectively. The ReLU activation function is used in the first fully connected layer, while the softmax activation function is used in the second fully connected layer to output the probability distribution over the 10 classes.

The model has 5,118,642 trainable parameters, and all layers are trainable, allowing the model to learn from the data and optimize its performance on the classification task.

```
set.seed(21200534)
fit <- model_cnn %>% fit(
train_generator,
steps_per_epoch = 1713/32,
epochs = 50,
validation_data = validation_generator,
validation_steps = 851/32
)
```

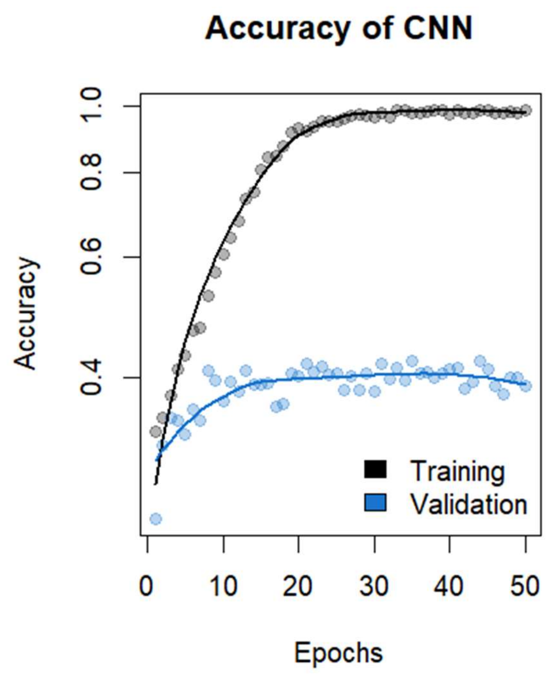
We fit the model using steps_per_epoch as 1713/32 which is the total number of training images/ batch_size and validation_steps as 851/32 which is the validation samples/batch_size and run it for 50 epochs.

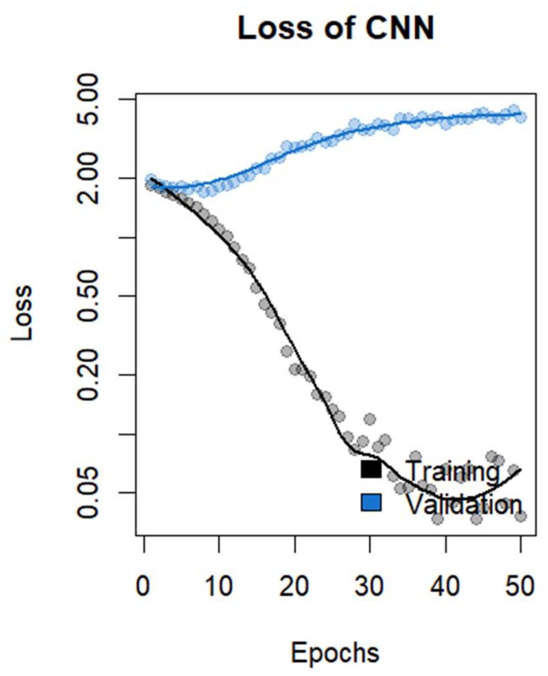
```
model_cnn %>% evaluate(train_generator)

##       loss  accuracy
## 0.004266919 0.998248696

model_cnn %>% evaluate(validation_generator)

##       loss  accuracy
## 4.0939145 0.3901293
```

The evaluate function gives us the accuracy and loss of the training as well as the validation data. Here we can see that the training data has accuracy of 0.998248696 and loss of 0.004266919 while for validation data the accuracy is `0.3901293` with a loss of

4.0939145. This shows that the model is still overfitting as the difference between the train and validation accuracy is significant. We will check this using the plot of accuracy and loss to further check the model.

```
smooth_line <- function(y) {
x <- 1:length(y)
out <- predict( loess(y ~ x) )
return(out)
}
# check learning curves
out <- cbind(fit$metrics$accuracy,
fit$metrics$val_accuracy,
fit$metrics$loss,
fit$metrics$val_loss)
cols <- c("black", "dodgerblue3")
par(mfrow = c(1,2))
# accuracy
matplot(out[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3),
log = "y", main = "Accuracy of CNN")
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),
log = "y", main ="Loss of CNN")
matlines(apply(out[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = cols, bty = "n")
```





The plots show that the model is overfitting as the validation accuracy is significantly less as compared to the training accuracy. We will try some other techniques to improve the model performance. We will try batch normalization and data augmentation for this CNN so as to check whether they can help improve our model's performance.

3. CNN with Batch Normalization.

```
model_cnn1 <- 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_dropout(rate = 0.4) %>%
layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
layer_max_pooling_2d(pool_size = c(2, 2)) %>%
layer_dropout(rate = 0.4) %>%
layer_conv_2d(filters = 256, kernel_size = c(3, 3), activation = "relu") %>%
layer_max_pooling_2d(pool_size = c(2, 2)) %>%
layer_dropout(rate = 0.4) %>%
layer_batch_normalization() %>%


# fully connected layers
layer_flatten() %>%
layer_batch_normalization() %>%
layer_dense(units = 512, activation = "relu") %>%
layer_dense(units = 10, activation = "sigmoid") %>%
#
```

```
# compile
compile(
loss = "sparse_categorical_crossentropy",
metrics = "accuracy",
optimizer = optimizer_adam(learning_rate = 0.001)
)
summary(model_cnn1)
```

```
## Model: "sequential_2"
##
_____________________________________________________________________________
___
##  Layer (type)                  Output Shape                Param #    
Trainable
##
=============================================================================
===
##  conv2d_5 (Conv2D)             (None, 60, 60, 64)          4864       Y
##  max_pooling2d_5 (MaxPooling2D  (None, 30, 30, 64)         0          Y
##  )
##  dropout_7 (Dropout)           (None, 30, 30, 64)          0          Y
##  conv2d_4 (Conv2D)             (None, 28, 28, 128)         73856      Y
##  max_pooling2d_4 (MaxPooling2D  (None, 14, 14, 128)        0          Y
##  )
##  dropout_6 (Dropout)           (None, 14, 14, 128)         0          Y
##  conv2d_3 (Conv2D)             (None, 12, 12, 256)         295168     Y
##  max_pooling2d_3 (MaxPooling2D  (None, 6, 6, 256)          0          Y
##  )
##  dropout_5 (Dropout)           (None, 6, 6, 256)           0          Y
##  batch_normalization_1 (BatchN  (None, 6, 6, 256)          1024       Y
##  ormalization)
##  flatten_2 (Flatten)           (None, 9216)                0          Y
##  batch_normalization (BatchNor  (None, 9216)               36864      Y
##  malization)
##  dense_5 (Dense)               (None, 512)                 4719104    Y
##  dense_4 (Dense)               (None, 10)                  5130       Y
##
=============================================================================
===
## Total params: 5,136,010
## Trainable params: 5,117,066
## Non-trainable params: 18,944
##
_____________________________________________________________________________
___
```

The model consists of three convolutional layers and three max-pooling layers. The input to the model is a 64x64 pixel RGB image, and the output is a probability distribution over 10 classes.

The first convolutional layer has 64 filters of size 5x5 which results in a feature map of size 60x60, this is designed as the input shape is 64 as well. We will simultaneously keep increasing the number of filters in the next layers to improve the learning capability. A max-pooling layer with a pool size of 2x2 follows, which reduces the feature map's size to 30x30. A dropout layer is added so that we can prevent overfitting of the model.

The second convolutional layer has 128 filters of size 3x3, we reduce the kernel size so as to keep the computing time in check This results in a feature map of size 28x28. Another max-pooling layer follows, reducing the feature map size to 14x14. A dropout layer is applied after the max-pooling layer to reduce overfitting.

The third convolutional layer has 256 filters of size 3x3, resulting in a feature map of size 12x12. Another max-pooling layer follows, reducing the feature map size to 6x6. A dropout layer is applied after the max-pooling layer to reduce overfitting.

In this model we add a batch normalization error after the convolutional layers. This layer helps the model to be stable and train faster. Batch normalization is used to normalize the layers by scaling and centering the data after the convolutional activity.

The output is then flattened to a 1-dimensional vector after the max-pooling layer, which is then passed through two fully connected layers with 512 and 10 units, respectively. The ReLU activation function is used in the first fully connected layer, while the softmax activation function is used in the second fully connected layer to output the probability distribution over the 10 classes.

The model has 5,136,010 parameters, out of which 5,117,066 are trainable

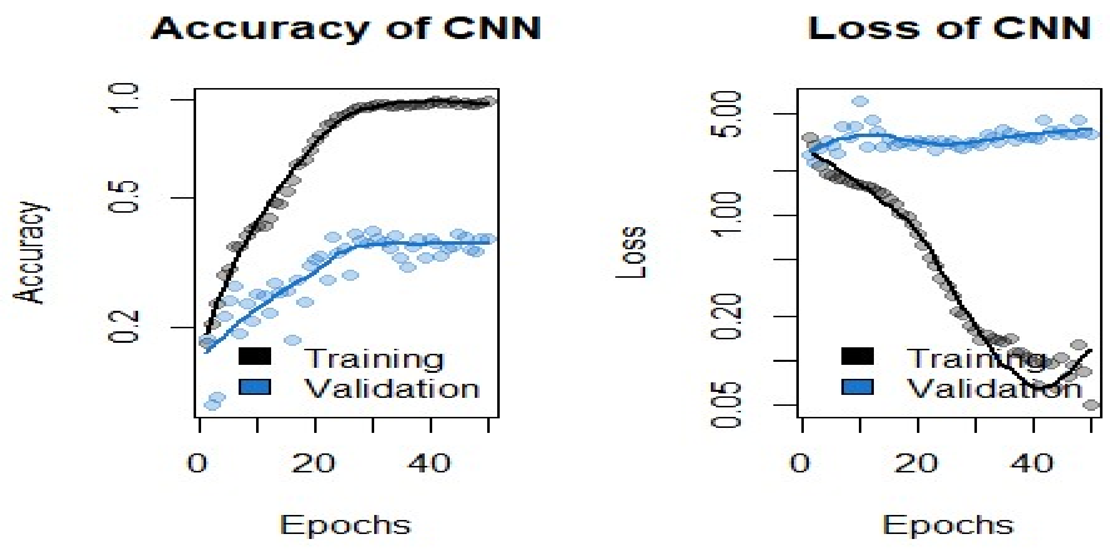
```
set.seed(21200534)
fit1 <- model_cnn1 %>% fit(
train_generator,
steps_per_epoch = 1713/32,
epochs = 50,
validation_data = validation_generator,
validation_steps = 851/32
)
```

Again we fit the model for 50 epochs using the steps_per_epoch and validation_steps as the previous model.

```
model_cnn1 %>% evaluate(train_generator)

##       loss   accuracy
## 0.01887343 0.99649739

model_cnn1 %>% evaluate(validation_generator)

##      loss  accuracy
## 3.5302458 0.3713278
```

The evaluate function gives us the accuracy and loss of the training as well as the validation data. Even the batch normalization doesn't help the overfitting of the model as we still see a high difference between the train and validation data. We will plot the accuracy and loss to get a better idea of the model performance.

```r
smooth_line <- function(y) {
x <- 1:length(y)
out <- predict( loess(y ~ x) )
return(out)
}
# check learning curves
out <- cbind(fit1$metrics$accuracy,
fit1$metrics$val_accuracy,
fit1$metrics$loss,
fit1$metrics$val_loss)
cols <- c("black", "dodgerblue3")
par(mfrow = c(1,2))
# accuracy
matplot(out[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3),
log = "y", main = "Accuracy of CNN")
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),
log = "y", main ="Loss of CNN")
matlines(apply(out[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = cols, bty = "n")
```



The plots show the accuracy and loss of the training and validation data over the epochs. We see the difference between the training and the validation accuracy suggesting the model is overfitting. After trying several combinations, we reach a conclusion that this problem is caused as the sample data is too small. We will now try using data augmentation to improve the model performance.

4. CNN with Data Augmentation.

```
data_augment <- image_data_generator(
rescale = 1/255,
rotation_range = 40,
width_shift_range = 0.2,
height_shift_range = 0.2,
shear_range = 0.2,
zoom_range = 0.2,
horizontal_flip = TRUE,
fill_mode = "nearest"
)
```

The given dataset has less samples which is causing our model to overfit. To overcome this problem, we use data augmentation to improve the generalization of the data. 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. Here along with rescaling the images to 1/255 we also use several other parameters as below

rotation_range - This parameter will help to rotate the image in any direction by 40 degrees

width_shift_range and height_shift_range - This parameter shifts the image horizontally and vertically by 20% in either direction.

shear_range - This parameter offers shearing transformation by 20%.

zoom_range - This parameter randomly zooms in and out of the image by 20%

horizontally_flip - This parameter randomly flips the image horizontally.

fill_mode - This parameter specifies that any new pixels created must be filled by the nearest pixel of the original image,

By applying these random transformations to the input images, the data generator can create a virtually infinite number of new images for training deep learning models, which helps improve the model’s performance and generalization.

```
train_generator1 <- flow_images_from_directory(
train_dir,
data_augment,
target_size = c(64,64),
batch_size = 32,
class_mode = "binary"
)
```

```
validation_generator1 <- flow_images_from_directory(
validation_dir,
data_augment,
target_size = c(64,64),
batch_size = 32,
class_mode = "binary"
)
```

We use the data augmentation to create the train and validtion generator keeping the target_zize and batch_size same as before. We will use this data to fit the CNN model without batch_normalization as it provided with a better accuracy.

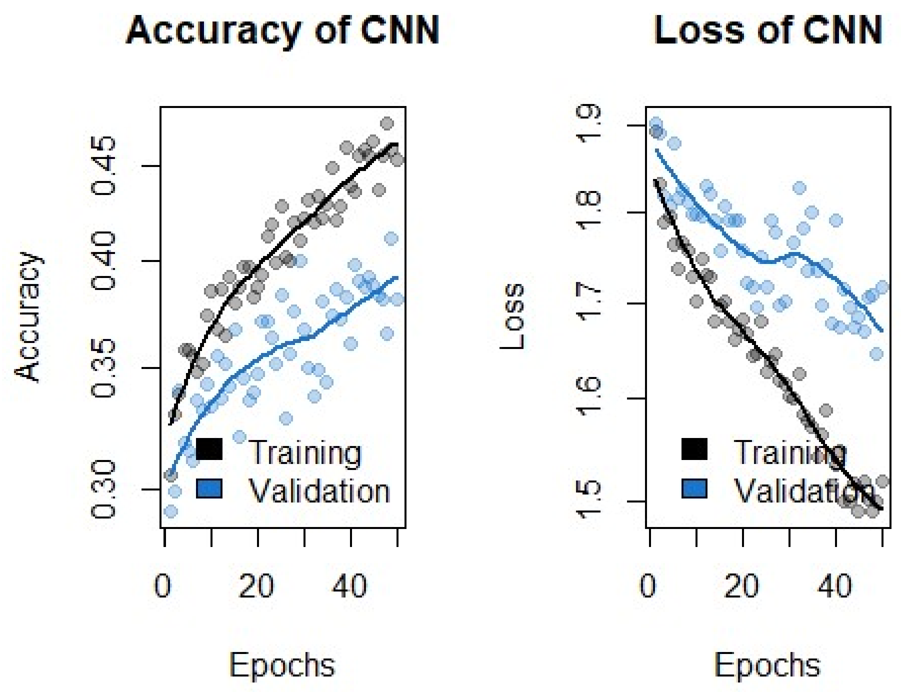
```
set.seed(21200534)
fit2 <- model_cnn %>% fit(
train_generator1,
steps_per_epoch = 1713/32,
epochs = 50,
validation_data = validation_generator1,
validation_steps = 852/32
)
```

Fitting the CNN model with new tain and validation generator for 50 epochs and steps_per_epoch as 1713/32 and validation as 851/32.

```
model_cnn %>% evaluate(train_generator1)

##      loss  accuracy
## 1.4389945 0.4839463

model_cnn %>% evaluate(validation_generator1)

##     loss accuracy
## 1.709664 0.373678
```

The evaluate function gives us the accuracy and loss of the training as well as the validation data. The training accuracy is 0.4839463 with a loss of 1.4389945. The validation accuracy is 0.373678 and loss is1.709664. The results show that the model performance has improved a lot with data augmentation and thus we can confirm that the problem of overfitting was caused by low sample size of the data. Let's see the plots to understand the performance.

```
smooth_line <- function(y) {
x <- 1:length(y)
out <- predict( loess(y ~ x) )
return(out)
}
# check learning curves
out <- cbind(fit2$metrics$accuracy,
fit2$metrics$val_accuracy,
fit2$metrics$loss,
fit2$metrics$val_loss)
cols <- c("black", "dodgerblue3")
par(mfrow = c(1,2))

# accuracy
matplot(out[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3),
log = "y", main = "Accuracy of CNN")
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),
log = "y", main ="Loss of CNN")
matlines(apply(out[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = cols, bty = "n")
```



The plots show the accuracy and loss of the training and validation data over the epochs. From the plot we can see that there is a consistency between the training and validation accuracy suggesting that the model has done a good job in generalizing the new data. Although there is a slight overfitting, this is by far the best model we have used.

Model Testing

We will use the CNN model with data augmentation to check its performance on the test data. The test data will be entirely new for the model and check how well it can predict the output.

```
train_generator2 <- flow_images_from_directory(
train_val,
data_augment,
target_size = c(64,64),
batch_size = 32,
class_mode = "binary"
)

test_generator2 <- flow_images_from_directory(
test_dir,
data_augment,
target_size = c(64,64),
batch_size = 32,
class_mode = "binary"
)
```

For testing the model on the test_data we use the train and validation data together to train our model. For this we create a new train_generator and test_generator

```
set.seed(21200534)
fit3 <- model_cnn %>% fit(
train_generator2,
steps_per_epoch = 2564/32,
epochs = 50
)
```

The model is fit for 50 epochs. As we have combined the train and validation data for training the model, we don't pass any validation data. The steps_per_epoch is set to 2564/32 as the training size now is 2564.

```
model_cnn %>% evaluate(train_generator2)

##      loss  accuracy
## 1.1928511 0.5881435

model_cnn %>% evaluate(test_generator2)

##     loss accuracy
## 1.605123 0.439483
```

The evaluate function gives us the accuracy and loss of the training as well as the new test data. The training accuracy is 0.5881435 with a loss of 1.1928511. The test accuracy is 0.439483 with a loss of 1.605123. This shows that there is a slight overfitting of the data but overall the model is able to generalize the data and provide predictions.

```r
tab<-table((test_generator2$classes)+1,model_cnn %>% predict(test_generator2)
%>% max.col())
acc <- diag(tab)/rowSums(tab)
cbind(tab, acc)
```

```
##    1  2 3 4  5  6 7  8  9 10        acc
## 1  0 11 1 0  5  3 1 15 10  3 0.00000000
## 2  5 32 7 5 24  7 2 39 39  5 0.19393939
## 3  1  6 1 3  2  1 0  3 10  1 0.03571429
## 4  2  5 3 1  3  4 0  5  5  5 0.03030303
## 5  1 15 1 5 11  2 1 22 24  4 0.12790698
## 6  4  7 2 4 10  1 2 21 15  2 0.01470588
## 7  0  8 0 1  3  1 1  4  6  1 0.04000000
## 8  3 13 4 6 32 14 6 56 41  8 0.30601093
## 9  7 24 3 5 22 11 4 36 56  8 0.31818182
## 10 2  5 2 0  5  2 0 10  6  6 0.15789474
```

The table shows the correlation matrix of the predictions and the original test data. The diagonal elements show the true values predicted by our model for the test data. The accuracy coloumn shows what is the accuracy of our model in predicting each class. We see that the accuracy of the model in not quite good in the prediction of the test data.

Conclusion

For the given data, we tried to employ a Neural network in various configurations to help in the predictions. The model that we predicted were as follows

1. Deep Neural Network
2. Convolutional Neural Network
3. Convolutional Neural Network with Batch Normalization.
4. Convolutional Neural Network with Data Augmentation.

Overall, the results were pretty similar for the first 3 models, with a huge difference between the test and validation accuracy, showing that the model was overfitting. However, data augmentation improved the model performance considerably showing that tha data in this case was too small for any model to learn and generalize. Even after using the data augmentation, the model could predict the test data with an accuracy of only 0.439483. This shows that the model was still not optimal and the data needs to have more samples to improve the model performance.