Using Deep Learning for Detecting Soybean Diseases

Prithviraj Lakkakula

3/9/2022

Contents

Unstructured Data

Unlike traditional structured data, Images or pictures and the text data are considered as unstructured data. The analysis of unstructured data involves converting into structured data and then conduct analysis.

Deep Learning and Keras

Deep learning is a subfield of machine learning that uses neural networks. A simple neural network consists of an input layer, a hidden layer, and an output layer. The term 'deep learning' is used for a model neural networks that has more than one hidden layer is used for conducting an analysis.

Keras is a high-level neural network application programming interface (API) for deep learning . It uses Tensorflow by Google as a backend. A front end is user interface (what the user sees) and a backend is a server application and database that works behind the scenes to delier the information to the user.

Soybean Disease Image Data

In this post, I will illustrate image classification and recognition using Keras package using 20 images for each of the four diseases of soybean. The four soybean diseases include bacterial blight (BB), bacterial pustule (BP), downy mildew (DM), and sudden death (SD) syndrome. In other words, this analysis is essentially a deep learning supervised learning problem that involve labeling of disease images or pictures of soybeans in this case. A total of 80 disease images will be used in this illustration. First 20 images are bacterial blight, and each of the next 20 are disease images of bacterial pustule, downy mildew, and sudden death syndrome, respectively.

Before proceeding, first we need to download and call the libraries of the following packages. It is important to note that the line of code that starts with # is considered as comment in R. Please follow the steps below.

Step 1. Loading required R packages

```
#The line of R code that starts with '#' is a comment. For example, this is a comment.
#install.packages("BiocManager")
#BiocManager::install("EBImage")
library(EBImage) #EBImage, an R package used to handle and explore image data
library(keras)   #Keras is a high-level neural network API for deep learning
```

```
##
## Attaching package: 'keras'
```

```
## The following object is masked from 'package:EBImage':
##
##     normalize
```

Step 2. Read Images

The following chunk of R code reads all soybean disease images. In our case, it is a total of 80 disease images for our illustration purposes.

```
#setwd('/Volumes/RAJ/DLImages/idata')
images = list.files(pattern="*.JPG")
myimages <- list()
for (i in 1:length(images)) {myimages[[i]] <- readImage(images[i])}
```

Step 3. Exploring Soybean Disease Images

In the following chunk of R code, the *print* function provides an output that converts unstructured data, that is image, to structured data (numbers). In other words, the dimensions of the image is converted into data points (**pixels**). The **print** function provides an output that consists of dimentions (dim) for each of the four diseases. First observation is that the first and last images are of different size and second and third image are of the same size. The **dim** in the output consists of three numbers. For example, if you take the first image, the dimensions are **6016 times 4016 times 3** which when multiplied gives **72,480,768** pixels as shown in the histogram figure of first image. The number 6016 is width of that image, 4016 is the height of the image, and 3 indicates the number of channels. In our case, as we are dealing with color images, the number of channels are 3, indicating RBG (red, blue, green). If it were a grayscale image, the last value in the **dim** would take a value of 1 (not 3).

```
m <- c(1, 21, 41, 61)
for (i in m) {print(myimages[[i]])}
```

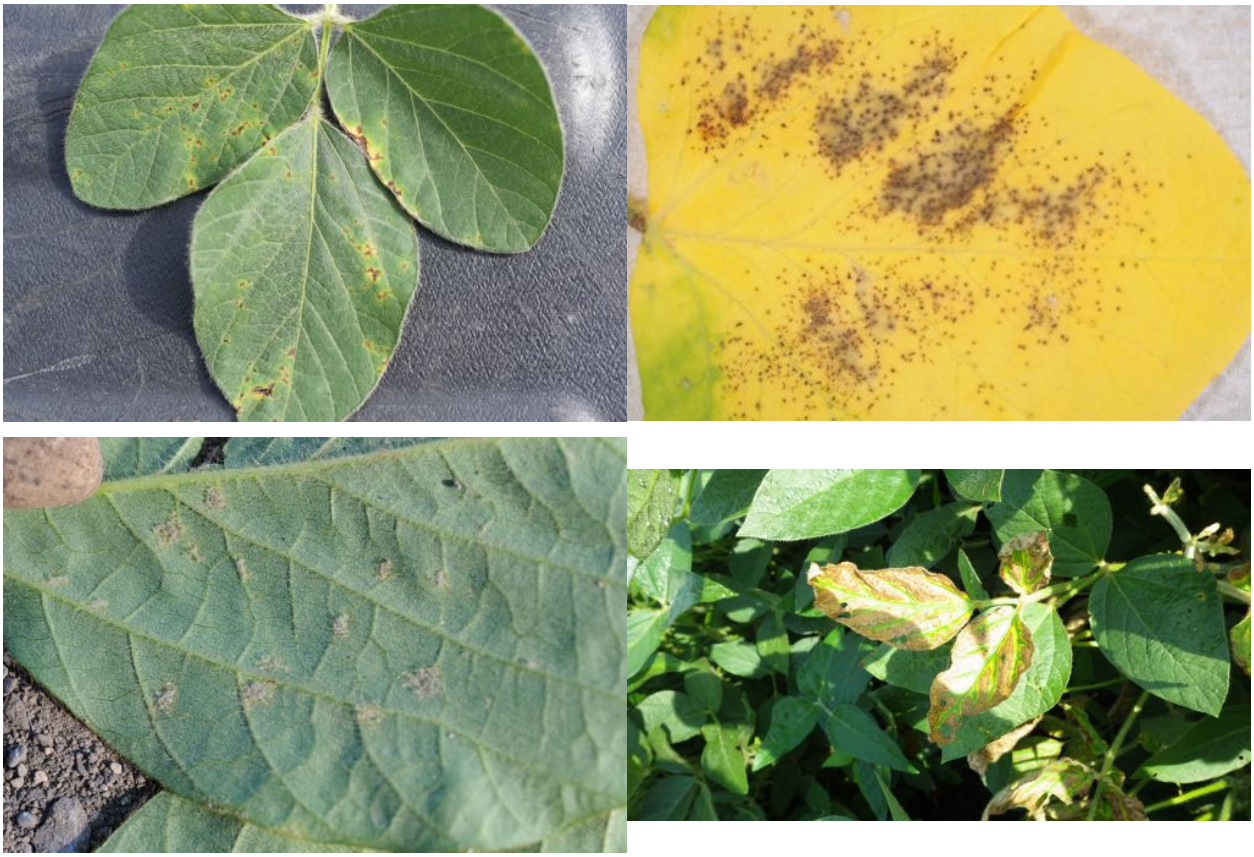
```
## Image
##   colorMode    : Color
##   storage.mode : double
##   dim          : 6016 4016 3
##   frames.total : 3
##   frames.render: 1
##
## imageData(object)[1:5,1:6,1]
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## [1,] 0.4352941 0.4313725 0.4235294 0.4117647 0.4000000 0.3960784
## [2,] 0.4392157 0.4352941 0.4274510 0.4156863 0.4039216 0.3960784
```

```
## [3,] 0.4392157 0.4352941 0.4274510 0.4196078 0.4078431 0.4000000
## [4,] 0.4235294 0.4235294 0.4235294 0.4196078 0.4117647 0.4078431
## [5,] 0.4196078 0.4156863 0.4196078 0.4235294 0.4156863 0.4078431
## Image
##   colorMode    : Color
##   storage.mode : double
##   dim          : 4288 2848 3
##   frames.total : 3
##   frames.render: 1
##
## imageData(object)[1:5,1:6,1]
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## [1,] 0.7921569 0.7921569 0.7960784 0.7960784 0.8000000 0.8000000
## [2,] 0.7882353 0.7921569 0.7921569 0.7921569 0.7960784 0.8000000
## [3,] 0.7882353 0.7921569 0.7921569 0.7921569 0.7960784 0.8000000
## [4,] 0.7882353 0.7921569 0.7921569 0.7921569 0.7960784 0.8000000
## [5,] 0.7882353 0.7921569 0.7921569 0.7921569 0.7960784 0.7960784
## Image
##   colorMode    : Color
##   storage.mode : double
##   dim          : 4288 2848 3
##   frames.total : 3
##   frames.render: 1
##
## imageData(object)[1:5,1:6,1]
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## [1,] 0.5647059 0.5607843 0.5607843 0.5607843 0.5647059 0.5647059
## [2,] 0.5725490 0.5686275 0.5647059 0.5568627 0.5647059 0.5686275
## [3,] 0.5803922 0.5764706 0.5725490 0.5647059 0.5686275 0.5686275
## [4,] 0.5725490 0.5686275 0.5647059 0.5686275 0.5686275 0.5607843
## [5,] 0.5725490 0.5647059 0.5490196 0.5647059 0.5686275 0.5647059
## Image
##   colorMode    : Color
##   storage.mode : double
##   dim          : 4608 2592 3
##   frames.total : 3
##   frames.render: 1
##
## imageData(object)[1:5,1:6,1]
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## [1,] 0.5098039 0.5254902 0.5450980 0.5411765 0.5333333 0.5333333
## [2,] 0.5215686 0.5215686 0.5333333 0.5450980 0.5372549 0.5333333
## [3,] 0.5215686 0.5176471 0.5333333 0.5450980 0.5411765 0.5372549
## [4,] 0.5137255 0.5098039 0.5215686 0.5411765 0.5450980 0.5450980
## [5,] 0.5098039 0.5058824 0.5137255 0.5254902 0.5450980 0.5529412
```

```
#print(myimages[[1]])
#display(myimages[[1]])
#summary(myimages[[1]])
#hist(myimages[[1]])
#str(myimages[[1]])
```

In the below, we plot an image of each of four diseases.

```
par(mfrow = c(2,2))
for (i in m) {plot(myimages[[i]])}
```



In the R code chunk below, the histogram of RBG channels are shown. It is clear that the intensity of RBG colors for each of the four disease images are quite different from the figures. Intensity values range between 0 and 1.

```
par(mfrow = c(1,1))
for (i in m) {hist(myimages[[i]])}
```

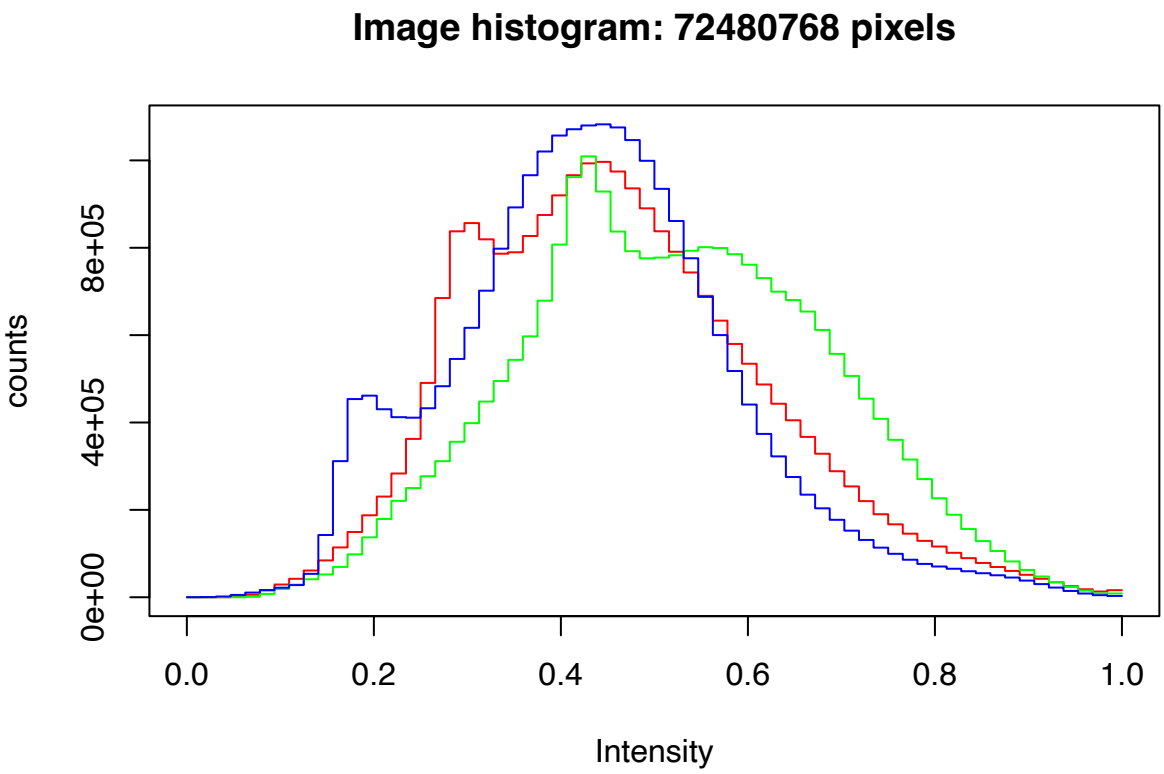


Image histogram: 72480768 pixels
counts
0e+00
4e+05
8e+05
0.0
0.2
0.4
0.6
0.8
1.0
Intensity

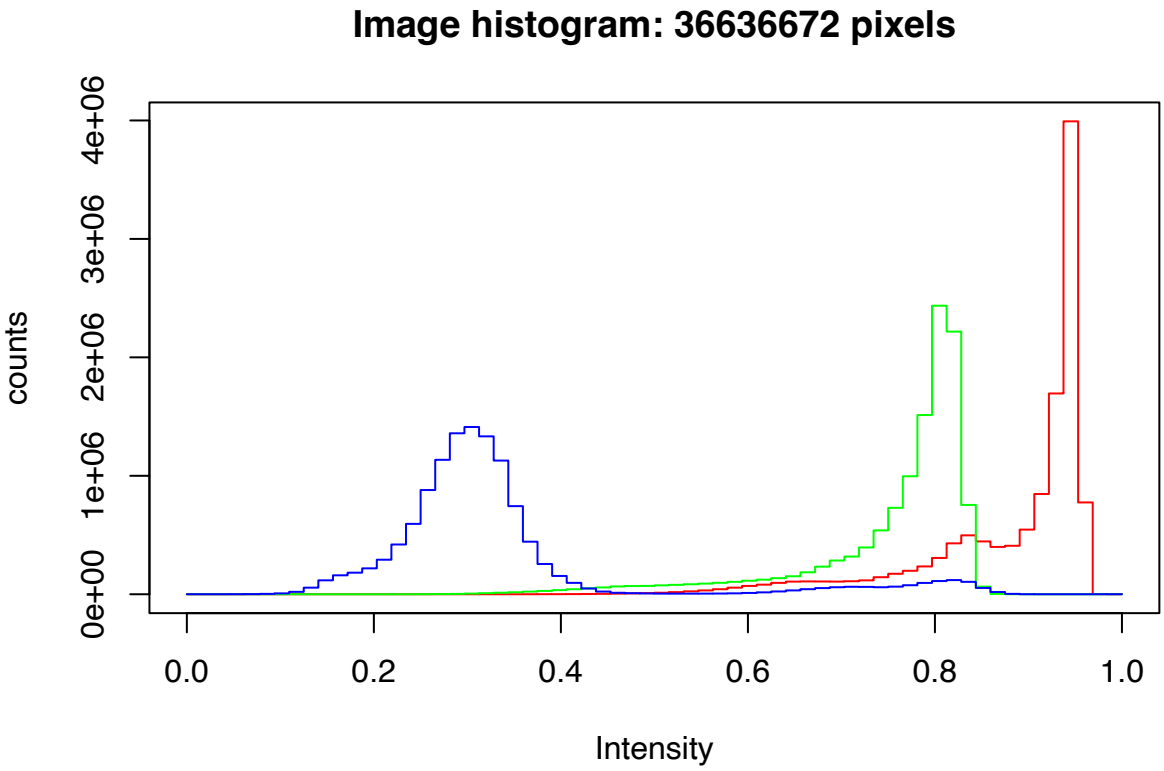


Image histogram: 36636672 pixels
counts
0e+00
1e+06
2e+06
3e+06
4e+06
0.0
0.2
0.4
0.6
0.8
1.0
Intensity

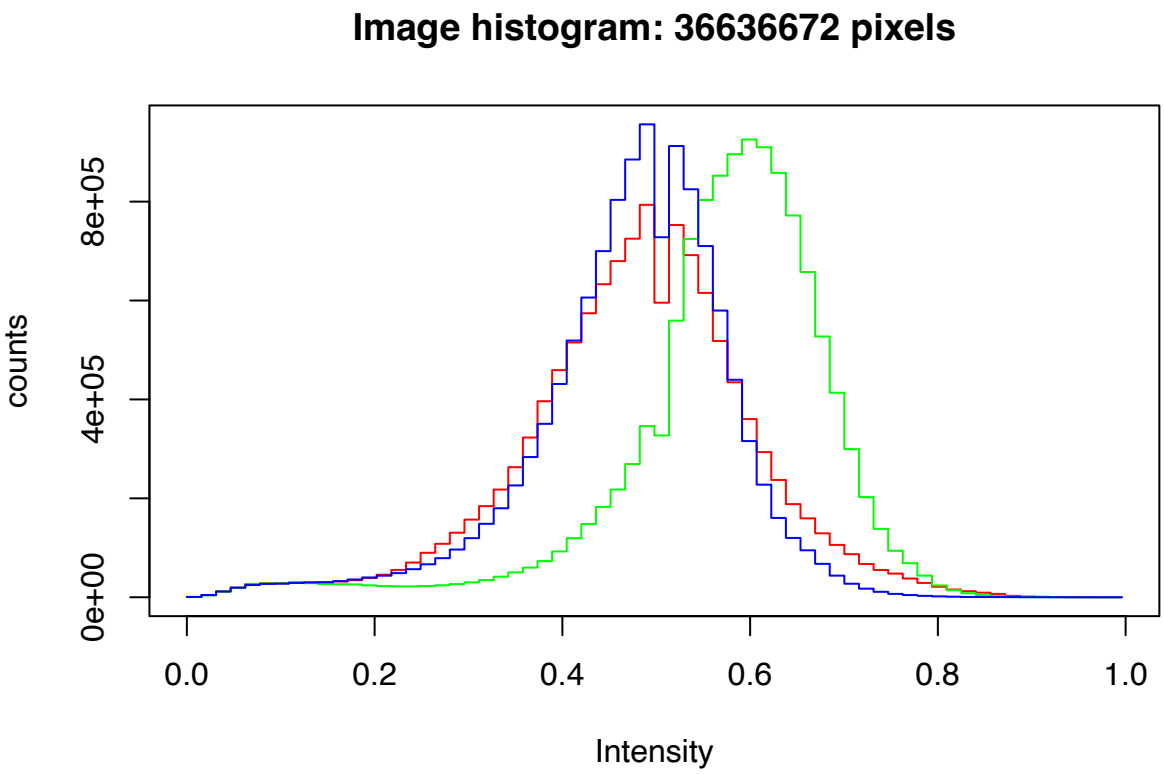
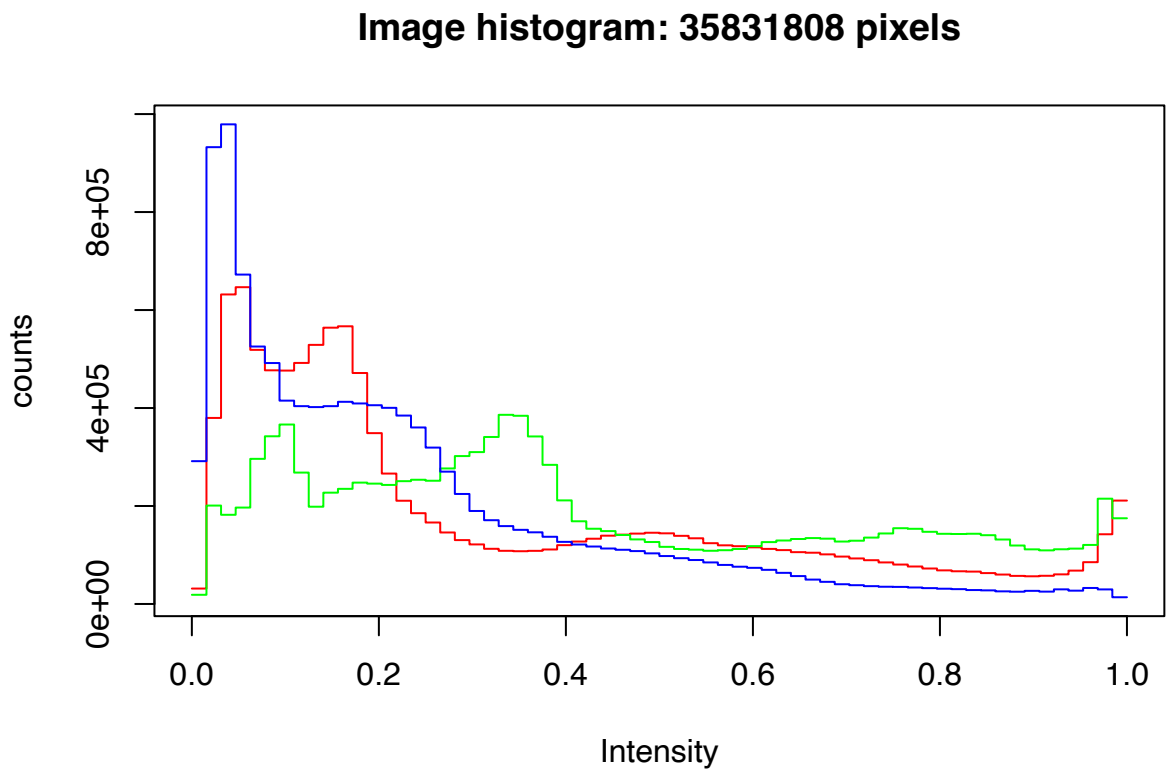


Image histogram: 36636672 pixels
counts
0e+00
4e+05
8e+05
0.0
0.2
0.4
0.6
0.8
1.0
Intensity



Step 4. Resizing and Reshaping Images

As we already know, the size of the images are different. As part of data preparation, one needs to convert all the images into a one fixed size. Here we are converting their size and reshaping into **36 times 36 times 3**.

```
for (i in 1:length(images)) {myimages[[i]] <- resize(myimages[[i]], 36, 36)}
for (i in 1:length(images)) {myimages[[i]] <- array_reshape(myimages[[i]], c(36, 36, 3))}
```

Step 5. Row Binding All the Images into Training, Validation, and Test Sets

In this step, we bind all the images into rows and divide them into three different sets, including training, validation, and test sets. The training set contains the first 14 images of each of the diseases. The validation set consists of next 2 images while the final 4 images of each disease images are placed in test set.

```
library(tensorflow)
#training set
tr <- c(1:14, 21:34, 41:54, 61:74)
x.train <- NULL
for (i in tr) {x.train <- rbind(x.train, myimages[[i]])}
str(x.train)
```

```
##  num [1:56, 1:3888] 0.482 0.45 0.271 0.259 0.717 ...
```

```
#validation set
va <- c(15:16, 35:36, 55:56, 75:76)
x.valid <- NULL
for (i in va) {x.valid <- rbind(x.valid, myimages[[i]])}

#test set
te <- c(17:20, 37:40, 57:60, 77:80)
x.test <- NULL
for (i in te) {x.test <- rbind(x.test, myimages[[i]])}
```

In the code chunk shown below, we are defining the Y variables for each of the diseases. That is, in the training set, the first 14 images are related to BB disease, while the next 14 images related to BP disease and so on. Similarly, for the validation set, the first 2 images with BB disease and so on. Finally, the test set consists of first 4 images with BB disease, next four with BP disease and so on.

```
y.train <- c(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
             2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,3,3,3,3,3,3)
y.valid <- c(0,0,1,1,2,2,3,3)
y.test <- c(0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3)
```

Step 6. One-Hot Encoding/Binary/Dummy Variables

Here, the binary or dummy variables are defined for each of the four diseases. In machine learning terms, this is also called one-hot encoding.

```
train.labels <- to_categorical(y.train)
```

```
## Loaded Tensorflow version 2.8.0
```

```
valid.labels <- to_categorical(y.valid)
test.labels <- to_categorical(y.test)

train.labels
```

```
##       [,1] [,2] [,3] [,4]
##  [1,]    1    0    0    0
##  [2,]    1    0    0    0
##  [3,]    1    0    0    0
##  [4,]    1    0    0    0
##  [5,]    1    0    0    0
##  [6,]    1    0    0    0
##  [7,]    1    0    0    0
##  [8,]    1    0    0    0
##  [9,]    1    0    0    0
## [10,]    1    0    0    0
## [11,]    1    0    0    0
## [12,]    1    0    0    0
## [13,]    1    0    0    0
## [14,]    1    0    0    0
## [15,]    0    1    0    0
## [16,]    0    1    0    0
```

```
## [17,]    0    1    0    0
## [18,]    0    1    0    0
## [19,]    0    1    0    0
## [20,]    0    1    0    0
## [21,]    0    1    0    0
## [22,]    0    1    0    0
## [23,]    0    1    0    0
## [24,]    0    1    0    0
## [25,]    0    1    0    0
## [26,]    0    1    0    0
## [27,]    0    1    0    0
## [28,]    0    1    0    0
## [29,]    0    0    1    0
## [30,]    0    0    1    0
## [31,]    0    0    1    0
## [32,]    0    0    1    0
## [33,]    0    0    1    0
## [34,]    0    0    1    0
## [35,]    0    0    1    0
## [36,]    0    0    1    0
## [37,]    0    0    1    0
## [38,]    0    0    1    0
## [39,]    0    0    1    0
## [40,]    0    0    1    0
## [41,]    0    0    1    0
## [42,]    0    0    1    0
## [43,]    0    0    0    1
## [44,]    0    0    0    1
## [45,]    0    0    0    1
## [46,]    0    0    0    1
## [47,]    0    0    0    1
## [48,]    0    0    0    1
## [49,]    0    0    0    1
## [50,]    0    0    0    1
## [51,]    0    0    0    1
## [52,]    0    0    0    1
## [53,]    0    0    0    1
## [54,]    0    0    0    1
## [55,]    0    0    0    1
## [56,]    0    0    0    1
```

```
valid.labels
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    1    0    0    0
## [3,]    0    1    0    0
## [4,]    0    1    0    0
## [5,]    0    0    1    0
## [6,]    0    0    1    0
## [7,]    0    0    0    1
## [8,]    0    0    0    1
```

```
test.labels
```

```
##       [,1] [,2] [,3] [,4]
##  [1,]    1    0    0    0
##  [2,]    1    0    0    0
##  [3,]    1    0    0    0
##  [4,]    1    0    0    0
##  [5,]    0    1    0    0
##  [6,]    0    1    0    0
##  [7,]    0    1    0    0
##  [8,]    0    1    0    0
##  [9,]    0    0    1    0
## [10,]    0    0    1    0
## [11,]    0    0    1    0
## [12,]    0    0    1    0
## [13,]    0    0    0    1
## [14,]    0    0    0    1
## [15,]    0    0    0    1
## [16,]    0    0    0    1
```

Step 7. Model Building

In model building, we start by creating a sequential model and then add various layers. ReLu (Rectified Linear Unit) is used as an activation function for hidden layers. The model is acivated with **36 times 36 times 3** equal 3888, which goes into the **input_shape**. For the output layer, we use **softmax** as an activation function. Finally, the summary of the model is obtained from the **summary** function.

```
str(x.train)
```

```
##  num [1:56, 1:3888] 0.482 0.45 0.271 0.259 0.717 ...
```

```
model1 <- keras_model_sequential()
model1 %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(3888)) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dense(units = 4, activation = 'softmax')

summary(model1)
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  dense_2 (Dense)                    (None, 256)                     995584
##
##  dense_1 (Dense)                    (None, 128)                     32896
##
##  dense (Dense)                      (None, 4)                       516
##
## ================================================================================
## Total params: 1,028,996
```

```
## Trainable params: 1,028,996
## Non-trainable params: 0
## _________________________________________________________________________________
```

Step 8. Showing the Calculations of Total Number of Parameters

The R code chunk below explains how we got the total number of parameters as 1028996 in the above step. The number that is added for each of the line below are intercepts.

```
(3888*256)+256
```

```
## [1] 995584
```

```
(128*256)+128
```

```
## [1] 32896
```

```
(128*4)+4
```

```
## [1] 516
```

```
#Total number of parameters = 1028996
```

Step 8. Compile the Model

In this step, we compile the model. The **categorical_crossentropy** is used for loss as we are doing a multi-class classification model. **Adam** is used as an optimizer while the **accuracy** is used as a metric.

```
model1 %>%
  compile(loss = 'categorical_crossentropy',
          optimizer = 'adam',
          metrics = 'accuracy')
```

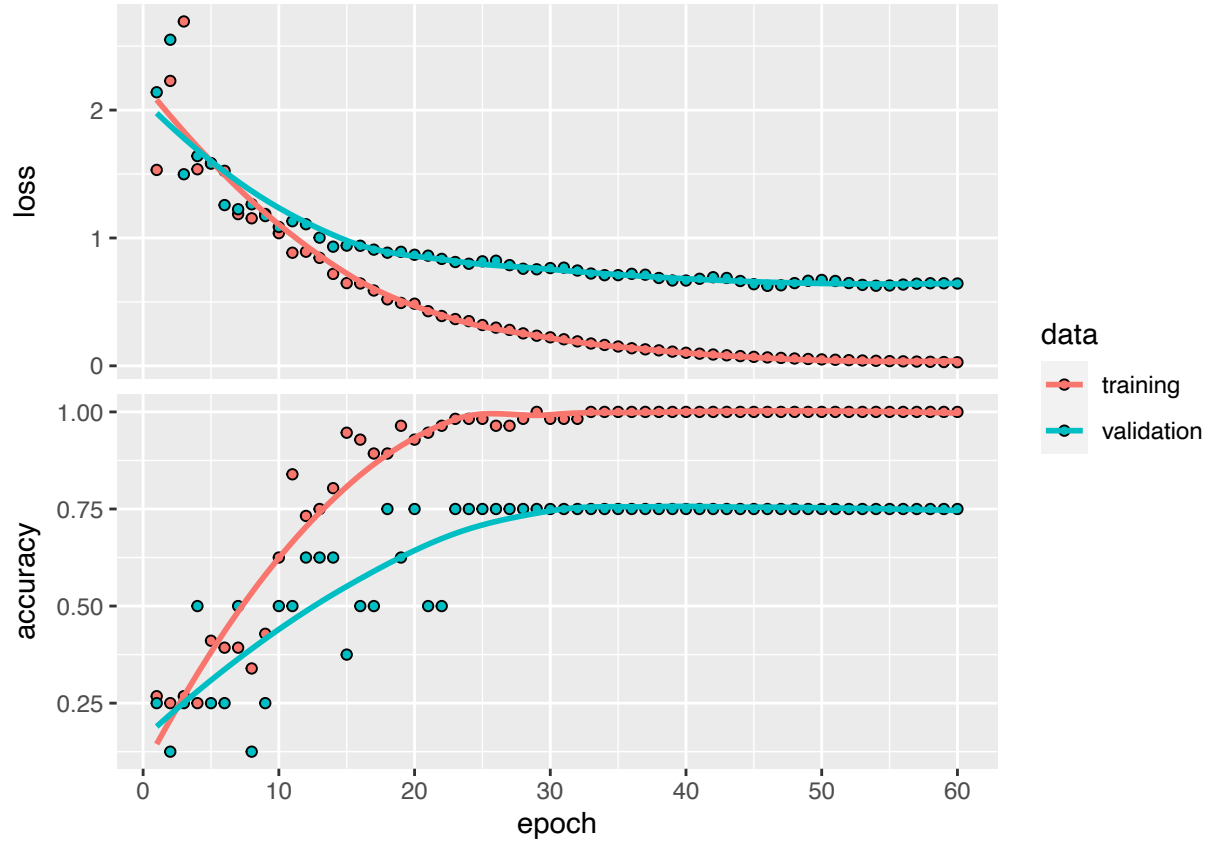
Step 9. Fit the Model

In this step, we fit the model. The plot consists of two panels. The top panel shows loss while the lower panel shows the accuracy for both training and validation dataset across the number of epochs on the x-axis. From the figure, key takeawsy is that at about 22 epochs the accuracy of the classification of disease images remains more or less same for the rest of epochs.

```
history <- model1 %>%
  fit(x.train,
      train.labels,
      epochs = 60,
      batch_size = 65,
      validation_data = list(x.valid, valid.labels))

plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Step 10. Model Evaluation and Prediction - Train Data

Here, confusion matrix and prediction probabilities of the results on training data is presented.

```
# Model Evaluation and Prediction - Train Data
model1 %>% evaluate(x.train, train.labels)
```

```
##       loss   accuracy
## 0.02698384 1.00000000
```

```
#confusion matrix
pred <- model1 %>% predict(x.train) %>% k_argmax()
table(Predicted = as.numeric(pred), Actual = y.train)
```

```
##           Actual
## Predicted  0  1  2  3
##         0 14  0  0  0
##         1  0 14  0  0
##         2  0  0 14  0
##         3  0  0  0 14
```

```r
#Prediction probabilities
prob <- model1 %>% predict(x.train)
cbind(round(prob, 3), Predicted_class = as.numeric(pred), Actual = y.train)
```

```
##                                Predicted_class Actual
##  [1,] 0.980 0.000 0.016 0.004               0      0
##  [2,] 0.967 0.000 0.018 0.015               0      0
##  [3,] 0.950 0.000 0.033 0.017               0      0
##  [4,] 0.992 0.000 0.004 0.004               0      0
##  [5,] 0.988 0.000 0.004 0.008               0      0
##  [6,] 0.994 0.000 0.002 0.004               0      0
##  [7,] 0.991 0.000 0.004 0.005               0      0
##  [8,] 0.988 0.000 0.009 0.003               0      0
##  [9,] 0.983 0.000 0.008 0.009               0      0
## [10,] 0.980 0.000 0.010 0.010               0      0
## [11,] 0.943 0.000 0.038 0.019               0      0
## [12,] 0.968 0.000 0.018 0.014               0      0
## [13,] 0.862 0.001 0.092 0.045               0      0
## [14,] 0.920 0.000 0.031 0.048               0      0
## [15,] 0.000 0.996 0.000 0.004               1      1
## [16,] 0.000 0.999 0.000 0.001               1      1
## [17,] 0.000 0.998 0.000 0.001               1      1
## [18,] 0.000 0.998 0.000 0.002               1      1
## [19,] 0.000 0.999 0.000 0.001               1      1
## [20,] 0.000 0.999 0.000 0.001               1      1
## [21,] 0.000 0.999 0.000 0.001               1      1
## [22,] 0.000 0.998 0.000 0.002               1      1
## [23,] 0.000 0.993 0.000 0.007               1      1
## [24,] 0.000 0.991 0.000 0.009               1      1
## [25,] 0.000 0.999 0.000 0.001               1      1
## [26,] 0.000 0.982 0.000 0.018               1      1
## [27,] 0.000 0.998 0.000 0.002               1      1
## [28,] 0.000 0.998 0.000 0.002               1      1
## [29,] 0.061 0.000 0.935 0.004               2      2
## [30,] 0.042 0.001 0.921 0.037               2      2
## [31,] 0.021 0.001 0.961 0.017               2      2
## [32,] 0.020 0.001 0.957 0.022               2      2
## [33,] 0.009 0.000 0.989 0.003               2      2
## [34,] 0.006 0.000 0.992 0.002               2      2
## [35,] 0.005 0.000 0.994 0.001               2      2
## [36,] 0.003 0.000 0.996 0.001               2      2
## [37,] 0.032 0.001 0.948 0.019               2      2
## [38,] 0.010 0.001 0.985 0.004               2      2
## [39,] 0.017 0.000 0.981 0.001               2      2
## [40,] 0.033 0.000 0.961 0.006               2      2
## [41,] 0.006 0.000 0.991 0.003               2      2
## [42,] 0.019 0.000 0.973 0.008               2      2
## [43,] 0.007 0.004 0.008 0.981               3      3
## [44,] 0.007 0.000 0.001 0.992               3      3
## [45,] 0.004 0.000 0.001 0.995               3      3
## [46,] 0.012 0.000 0.005 0.983               3      3
## [47,] 0.016 0.013 0.036 0.935               3      3
## [48,] 0.004 0.006 0.007 0.982               3      3
```

```
## [49,] 0.084 0.001 0.021 0.894              3      3
## [50,] 0.026 0.005 0.019 0.951              3      3
## [51,] 0.013 0.005 0.011 0.970              3      3
## [52,] 0.018 0.009 0.005 0.969              3      3
## [53,] 0.028 0.007 0.008 0.957              3      3
## [54,] 0.010 0.008 0.003 0.979              3      3
## [55,] 0.013 0.009 0.011 0.967              3      3
## [56,] 0.032 0.009 0.018 0.941              3      3
```

Step 11. Evaluation and Prediction on the Test Data

In this final step, the confusion matrix and prediction probabilities of the model evaluated on the test data is presented.

```
# Evaluation and prediction on the test data
model1 %>% evaluate(x.test, test.labels)
```

```
##      loss  accuracy
## 0.7907364 0.6875000
```

```
#confusion matrix
pred <- model1 %>% predict(x.test) %>% k_argmax()
table(Predicted = as.numeric(pred), Actual = y.test)
```

```
##          Actual
## Predicted 0 1 2 3
##         1 0 4 0 0
##         2 4 0 4 1
##         3 0 0 0 3
```

```
#prediction probabilities
prob <- model1 %>% predict(x.test)
cbind(round(prob, 2), Predicted_class = as.numeric(pred), Actual = y.test)
```

```
##                          Predicted_class Actual
##  [1,] 0.11 0.01 0.67 0.21               2      0
##  [2,] 0.16 0.01 0.70 0.14               2      0
##  [3,] 0.15 0.01 0.68 0.17               2      0
##  [4,] 0.13 0.01 0.74 0.13               2      0
##  [5,] 0.00 1.00 0.00 0.00               1      1
##  [6,] 0.00 1.00 0.00 0.00               1      1
##  [7,] 0.00 1.00 0.00 0.00               1      1
##  [8,] 0.00 1.00 0.00 0.00               1      1
##  [9,] 0.09 0.00 0.83 0.07               2      2
## [10,] 0.04 0.00 0.91 0.05               2      2
## [11,] 0.07 0.00 0.87 0.06               2      2
## [12,] 0.06 0.00 0.89 0.05               2      2
## [13,] 0.12 0.00 0.85 0.03               2      3
## [14,] 0.06 0.13 0.20 0.61               3      3
## [15,] 0.01 0.00 0.02 0.97               3      3
## [16,] 0.00 0.00 0.00 0.99               3      3
```

Step 12. Results and Fine Tuning the Model

The training results show good results where the accuracy is quite high but when it comes to the results of test set there are some misclassified disease images. The model in steps 7 and 9 are adjusted with different number of neurons and fine tune to make sure the deep learning model is not overfitted and thereby obtain better results of correctly classified disease images.