University of Tehran
College of Engineering
School of Electrical and Computer Engineering

Intelligent Systems

Dr.Hosseini

Homework 4

Soroush Mesforush Mashhad

SN:810198472

Dey 01

Contents

Abstract

In this project we begin with analyzing a multi layered perceptron neural network theoretically, then we verify our calculations via python.

In the second part we design an **MLP** neural network and change various elements in it and observe the accuracy for different states. Then we go on to add a **CNN** to the **MLP** design and observe the effects of pooling, batch normalization etc...

In the final section we first study EfficientNet then perform transfer learning on it.

1 Question 1 : Multi layered perceptron

1.1 Theoretical Analysis

Here we shall update the coefficients for two iterations.

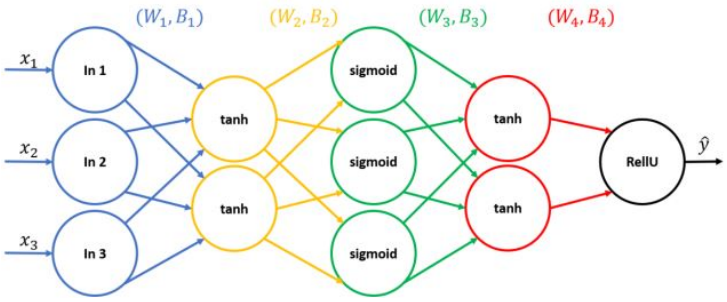

Figure 1: Multi Layered Perceptron

For my SID : 810198472 the values shall be as depicted below.

$$X_1 = \begin{bmatrix} 2 \\ 7 \\ 2 \end{bmatrix}, Y_1 = \begin{bmatrix} 2 \end{bmatrix}, \quad X_2 = \begin{bmatrix} 7 \\ 2 \\ 7 \end{bmatrix}, Y_2 = \begin{bmatrix} 7 \end{bmatrix}$$

$$W_1 = \begin{bmatrix} 0.12 & 0.27 \\ 0.32 & 0.47 \\ 0.52 & 0.47 \end{bmatrix}, \quad W_2 = \begin{bmatrix} 2.15 & 2.25 & 2.35 \\ 7.45 & 7.55 & 7.65 \end{bmatrix}$$

$$W_3 = \begin{bmatrix} 14.12 & 14.22 \\ 14.32 & 14.42 \\ 14.52 & 14.62 \end{bmatrix}, \quad W_4 = \begin{bmatrix} -4.84 \\ -4.64 \end{bmatrix}$$

$$B_1 = \begin{bmatrix} 0.21 \\ 0.72 \end{bmatrix}, \quad B_2 = \begin{bmatrix} 9.15 \\ 9.25 \\ 9.35 \end{bmatrix}, \quad B_3 = \begin{bmatrix} 0.37 \\ 0.47 \end{bmatrix}, \quad B_4 = \begin{bmatrix} 5.26 \end{bmatrix}$$

The error function is depicted as below.

$$E = \frac{1}{2}(\hat{y} - y)^2$$

We shall first start the feed forward with with X_1 and Y_1 as depicted below.

$$Z = \tanh(W_1^T X + B_1), \quad W_1^T X + B_1 = \begin{bmatrix} 0.12 & 0.32 & 0.52 \\ 0.27 & 0.47 & 0.47 \end{bmatrix} \begin{bmatrix} 2 \\ 7 \\ 2 \end{bmatrix} + \begin{bmatrix} 0.21 \\ 0.72 \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} 3.52 \\ 4.77 \end{bmatrix} + \begin{bmatrix} 0.21 \\ 0.72 \end{bmatrix} = \begin{bmatrix} 3.73 \\ 5.49 \end{bmatrix}, \quad Z = \tanh\left(\begin{bmatrix} 3.73 \\ 5.49 \end{bmatrix}\right) \longrightarrow Z = \begin{bmatrix} 0.99885 \\ 0.99997 \end{bmatrix}$$

$$K = Sigmoid(W_2^T Z + B_2), \quad W_2^T Z + B_2 = \begin{bmatrix} 2.15 & 7.45 \\ 2.25 & 7.55 \\ 2.35 & 7.65 \end{bmatrix} \begin{bmatrix} 0.99885 \\ 0.99997 \end{bmatrix} + \begin{bmatrix} 9.15 \\ 9.25 \\ 9.35 \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} 9.597 \\ 9.797 \\ 9.997 \end{bmatrix} + \begin{bmatrix} 9.15 \\ 9.25 \\ 9.35 \end{bmatrix} = \begin{bmatrix} 18.747 \\ 19.038 \\ 19.339 \end{bmatrix}, \quad K = Sigmoid\left(\begin{bmatrix} 18.747 \\ 19.038 \\ 19.339 \end{bmatrix}\right) = \begin{bmatrix} 0.9999 \\ 0.9999 \\ 0.9999 \end{bmatrix}$$

$$P = \tanh(W_3^T K + B_3), \quad W_3^T K + B_3 = \begin{bmatrix} 14.12 & 14.32 & 14.52 \\ 14.22 & 14.42 & 14.62 \end{bmatrix} \begin{bmatrix} 0.9999 \\ 0.9999 \\ 0.9999 \end{bmatrix} + \begin{bmatrix} 0.37 \\ 0.47 \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} 42.956 \\ 43.256 \end{bmatrix} + \begin{bmatrix} 0.37 \\ 0.47 \end{bmatrix} = \begin{bmatrix} 43.326 \\ 43.726 \end{bmatrix}, \quad P = \tanh\left(\begin{bmatrix} 43.326 \\ 43.726 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\hat{y} = Relu(W_4^T P + B_4), \quad W_4^T P + B_4 = \begin{bmatrix} -4.84 & -4.64 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 5.26 \end{bmatrix} = \begin{bmatrix} -4.22 \end{bmatrix}.$$

$$\hat{y} = Relu(-4.22) = max(-4.22, 0) = 0, \quad E = \frac{1}{2}(\hat{y} - y)^2 = 2$$

Now we shall perform back propagation. This was very difficult for me and I watched Andrew Ng's neural network's course. We studied in the lectures that we update parameters as follows.

$$W_n = W_n - lr\frac{\partial L}{\partial W_n}, \quad B_n = B_n - lr\frac{\partial L}{\partial B_n}$$

In the following picture I have included Andrew Ng's handwritten lecture of the backpropagation equations then I have tidied them up accordingly.

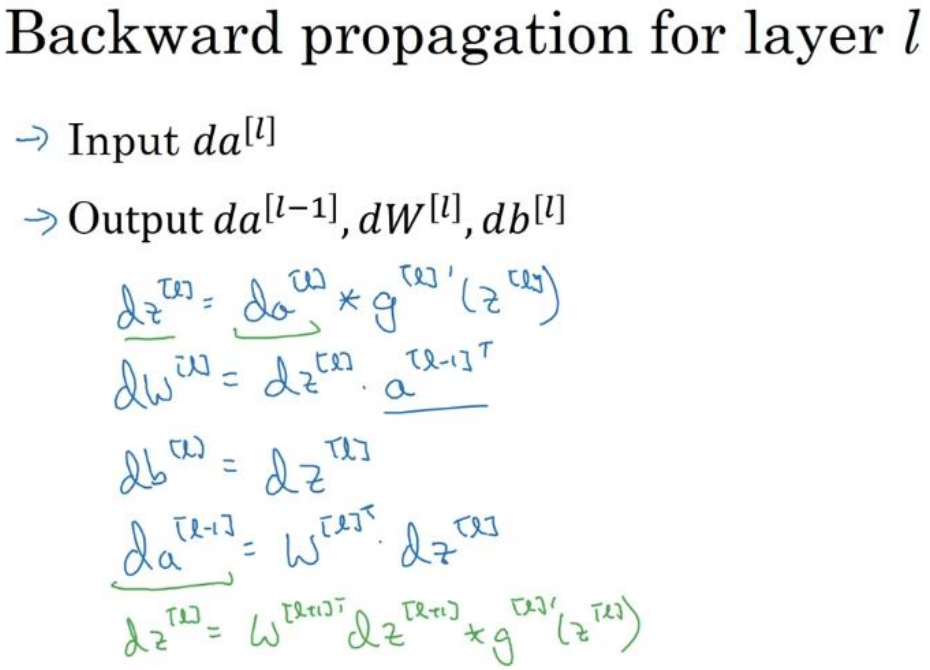

Figure 2: Backpropagation equations

$$
\begin{aligned}
&\frac{\partial L}{\partial z^{[l]}} = \frac{\partial L}{\partial a^{[l]}} \circ \frac{\partial a^{[l]}}{\partial z^{[l]}} \\
&\frac{\partial L}{\partial W^{[l]}}^{T} = \frac{\partial L}{\partial z^{[l]}} . (a^{[l-1]})^{T} \\
&\frac{\partial L}{\partial B^{[l]}} = \frac{\partial L}{\partial z^{[l]}} \\
&\frac{\partial L}{\partial a^{[l-1]}} = W^{[l]} \frac{\partial L}{\partial z^{[l]}} \\
&z^{[l]} = W^{[l]} . a^{[l-1]} + b^{[l]}
\end{aligned}
$$

Now we shall continue to find the necessary derivatives, before doing so we

must pay attention that.

$$f(z) = \frac{1}{1+e^{-z}} \longrightarrow f'(z) = 1 - f(z),$$
$$f(z) = \tanh(z) \longrightarrow f'(z) = 1 - f^2(z)$$
$$f(z) = Relu(z) \longrightarrow f'(z) = H(z), \quad H(z) : \text{Heaviside}$$

Now we shall perform some very lengthy derivatives.

Layer 4

$$\frac{\partial L}{\partial W^{[4]}}^T = \frac{\partial L}{\partial z^{[4]}}.(a^{[3]})^T, \quad \frac{\partial L}{\partial z^{[4]}} = \frac{\partial L}{\partial a^{[4]}} \circ \frac{\partial a^{[4]}}{\partial z^{[4]}} = (\hat{y} - y)H(W_4^T P + B_4)$$
$$\frac{\partial L}{\partial W^{[4]}}^T = (\hat{y} - y)H(W_4^T P + B_4)P^T, \quad \frac{\partial L}{\partial B^{[4]}} = (\hat{y} - y)H(W_4^T P + B_4)$$

Layer 3

$$\frac{\partial L}{\partial W^{[3]}}^T = \frac{\partial L}{\partial z^{[3]}}.(a^{[2]})^T, \quad \frac{\partial L}{\partial z^{[3]}} = \frac{\partial L}{\partial a^{[3]}} \circ \frac{\partial a^{[3]}}{\partial z^{[3]}} = \big(W_4(\hat{y} - y)H(W_4^T P + B_4)\big) \circ (1 - P^2)$$
$$\frac{\partial L}{\partial W^{[3]}}^T = \big(\big(W_4(\hat{y} - y)H(W_4^T P + B_4)\big) \circ (1 - P^2)\big).K^T,$$
$$\frac{\partial L}{\partial B^{[3]}} = \big(W_4(\hat{y} - y)H(W_4^T P + B_4)\big) \circ (1 - P^2)$$

Layer 2

$$\frac{\partial L}{\partial W^{[2]}}^T = \frac{\partial L}{\partial z^{[2]}}.(a^{[1]})^T,$$
$$\frac{\partial L}{\partial z^{[2]}} = \frac{\partial L}{\partial a^{[2]}} \circ \frac{\partial a^{[2]}}{\partial z^{[2]}} = \big(W_3\big(W_4(\hat{y} - y)H(W_4^T P + B_4)\big) \circ (1 - P^2)\big) \circ (K(1 - K))$$
$$\frac{\partial L}{\partial W^{[2]}}^T = \big(W_3\big(W_4(\hat{y} - y)H(W_4^T P + B_4)\big) \circ (1 - P^2)\big) \circ (K(1 - K)).Z^T,$$
$$\frac{\partial L}{\partial B^{[2]}} = \big(W_3\big(W_4(\hat{y} - y)H(W_4^T P + B_4)\big) \circ (1 - P^2)\big) \circ (K(1 - K))$$

Layer 1

$$\frac{\partial L}{\partial W^{[1]}}^T = \frac{\partial L}{\partial z^{[1]}}.(a^{[0]})^T,$$
$$\frac{\partial L}{\partial z^{[1]}} = \frac{\partial L}{\partial a^{[1]}} \circ \frac{\partial a^{[1]}}{\partial z^{[2]}} = \big(W_2\left(W_3\left(W_4(\hat{y}-y)H(W_4^T P + B_4)\right) \circ (1-P^2)\right) \circ (K(1-K))\big) \circ (1-Z^2)$$
$$\frac{\partial L}{\partial W^{[1]}}^T = \big(W_2\left(W_3\left(W_4(\hat{y}-y)H(W_4^T P + B_4)\right) \circ (1-P^2)\right) \circ (K(1-K))\big) \circ (1-Z^2).X^T,$$
$$\frac{\partial L}{\partial B^{[1]}} = \big(W_2\left(W_3\left(W_4(\hat{y}-y)H(W_4^T P + B_4)\right) \circ (1-P^2)\right) \circ (K(1-K))\big) \circ (1-Z^2)$$

Now for the first backpropagation we have:

$$\frac{\partial L}{\partial W^{[4]}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \frac{\partial L}{\partial B^{[4]}} = [0]$$

$$\frac{\partial L}{\partial W^{[3]}} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad \frac{\partial L}{\partial B^{[3]}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\frac{\partial L}{\partial W^{[2]}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \frac{\partial L}{\partial B^{[2]}} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\frac{\partial L}{\partial W^{[1]}} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad \frac{\partial L}{\partial B^{[1]}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The results are somewhat unusual, I have double and triple checked my answers and there is nothing wrong with it, the results are like this because of my SID, so when we perform the updating, nothing happens, we shall go on to give X_2 as the input.

$$Z = \tanh(W_1^T X + B_1), \quad W_1^T X + B_1 = \begin{bmatrix} 0.12 & 0.32 & 0.52 \\ 0.27 & 0.47 & 0.47 \end{bmatrix} \begin{bmatrix} 7 \\ 2 \\ 7 \end{bmatrix} + \begin{bmatrix} 0.21 \\ 0.72 \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} 5.12 \\ 6.12 \end{bmatrix} + \begin{bmatrix} 0.21 \\ 0.72 \end{bmatrix} = \begin{bmatrix} 5.33 \\ 6.84 \end{bmatrix}, \quad Z = \tanh\left(\begin{bmatrix} 5.33 \\ 6.84 \end{bmatrix}\right) \longrightarrow Z = \begin{bmatrix} 0.99995 \\ 0.99999 \end{bmatrix}$$

$$K = Sigmoid(W_2^T Z + B_2), \quad W_2^T Z + B_2 = \begin{bmatrix} 2.15 & 7.45 \\ 2.25 & 7.55 \\ 2.35 & 7.65 \end{bmatrix} \begin{bmatrix} 0.99995 \\ 0.99999 \end{bmatrix} + \begin{bmatrix} 9.15 \\ 9.25 \\ 9.35 \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} 9.6 \\ 9.8 \\ 9.99 \end{bmatrix} + \begin{bmatrix} 9.15 \\ 9.25 \\ 9.35 \end{bmatrix} = \begin{bmatrix} 18.75 \\ 19.05 \\ 19.34 \end{bmatrix}, \quad K = Sigmoid\left(\begin{bmatrix} 18.75 \\ 19.05 \\ 19.34 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$P = \tanh(W_3^T K + B_3), \quad W_3^T K + B_3 = \begin{bmatrix} 14.12 & 14.32 & 14.52 \\ 14.22 & 14.42 & 14.62 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0.37 \\ 0.47 \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} 42.96 \\ 43.26 \end{bmatrix} + \begin{bmatrix} 0.37 \\ 0.47 \end{bmatrix} = \begin{bmatrix} 43.33 \\ 43.73 \end{bmatrix}, \quad P = \tanh\left(\begin{bmatrix} 43.33 \\ 43.73 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\hat{y} = Relu(W_4^T P + B_4), \quad W_4^T P + B_4 = \begin{bmatrix} -4.84 & -4.64 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 5.26 \end{bmatrix} = \begin{bmatrix} -4.22 \end{bmatrix}.$$

$$\hat{y} = Relu(-4.22) = max(-4.22, 0) = 0, \quad E = \frac{1}{2}(\hat{y} - y)^2 = 24.5$$

The backpropagation is exactly like the previous iteration, the vanishing gradient problem happens again.

$$\frac{\partial L}{\partial W^{[4]}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \frac{\partial L}{\partial B^{[4]}} = [0]$$

$$\frac{\partial L}{\partial W^{[3]}} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad \frac{\partial L}{\partial B^{[3]}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\frac{\partial L}{\partial W^{[2]}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \frac{\partial L}{\partial B^{[2]}} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\frac{\partial L}{\partial W^{[1]}} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad \frac{\partial L}{\partial B^{[1]}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

1.2 Python Implementation

Here we have implemented what the did theoretically in python, the codes are as below.

```
def step(x):
if x>0:
return 1
else:
return 0

```

Figure 3: Step function

```
def AclayFeed(x,W1,W2,W3,W4,B1,B2,B3,B4):
Z = np.tanh(np.dot(np.transpose(W1),x)+B1)
K= 1/1+np.exp(-(np.dot(np.transpose(W2),Z)+B2))
P = np.tanh(np.dot(np.transpose(W3),K)+B3)
Yhat = np.maximum(0,np.dot(np.transpose(W4),P)+B4)
return Yhat[0][0]

```

Figure 4: Yhat calculator

```
def Gradient_CalcandUp(x,W1,W2,W3,W4,B1,B2,B3,B4,Y,lr):
Z = np.tanh(np.dot(np.transpose(W1),x)+B1)
K= 1/1+np.exp(-(np.dot(np.transpose(W2),Z)+B2))
P = np.tanh(np.dot(np.transpose(W3),K)+B3)
Yhat = np.maximum(0,np.dot(np.transpose(W4),P)+B4)
A = np.dot(np.transpose(W4),P)+B4
dW4 = np.transpose((Yhat - Y)*step(A)*np.transpose(P))
dB4 = (Yhat - Y)*step(A)
dW3 = np.transpose(np.matmul(np.multiply(W4 *(Yhat - Y)*A,1-P
  **2),np.transpose(K)))
dB3 = np.multiply(W4 *(Yhat - Y)*A,1-P**2)
TempW2=np.multiply(W4 *(Yhat - Y)*A,1-P**2)
dW2 = np.transpose(np.matmul(np.multiply(np.matmul(W3,TempW2),
  K*(1-K)),np.transpose(Z)))
dB2 =np.multiply(np.matmul(W3,TempW2),K*(1-K))
TempW1 = np.multiply(np.matmul(W3,TempW2),K*(1-K))
dW1 = np.transpose(np.matmul(np.multiply(np.matmul(W2,TempW1)
  ,1-Z**2),np.transpose(x)))
dB1 = np.multiply(np.matmul(W2,TempW1),1-Z**2)
W4-=lr*dW4
W3-=lr*dW3
W2-=lr*dW2
W1-=lr*dW1
B4-=lr*dB4
B3-=lr*dB3
B2-=lr*dB2
B1-=lr*dB1
return W4,W3,W2,W1,B4,B3,B2,B1

```

Figure 5: Gradient calculator

After this we shall continue to obtain the final results.

```
#First Iter
Y1=2
Yhat1=AclayFeed(x1,W1,W2,W3,W4,B1,B2,B3,B4)
print(f"Error for first feed forward: {ErrorCalc(Y1,Yhat1)}")
lr = 0.1
W4up1,W3up1,W2up1,W1up1,B4up1,B3up1,B2up1,B1up1=
  Gradient_CalcandUp(x1,W1,W2,W3,W4,B1,B2,B3,B4,Y1,lr)
print(f"Updated W4 after 1 iteration : {W4up1}")
print(f"Updated W3 after 1 iteration : {W3up1}")
print(f"Updated W2 after 1 iteration : {W2up1}")
print(f"Updated W1 after 1 iteration : {W1up1}")
print(f"Updated B4 after 1 iteration : {B4up1}")
print(f"Updated B3 after 1 iteration : {B3up1}")
print(f"Updated B2 after 1 iteration : {B2up1}")
print(f"Updated B1 after 1 iteration : {B1up1}")
##Second Iter
Y2=7
Yhat2=AclayFeed(x2,W1,W2,W3,W4,B1,B2,B3,B4)
print('
  ------------------------------------------------------------------
  ')
print(f"Error for second feed forward: {ErrorCalc(Y2,Yhat2)}")
lr = 0.1
W4up2,W3up2,W2up2,W1up2,B4up2,B3up2,B2up2,B1up2=
  Gradient_CalcandUp(x2,W1,W2,W3,W4,B1,B2,B3,B4,Y2,lr)
print(f"Updated W4 after 2 iterations : {W4up2}")
print(f"Updated W3 after 2 iterations : {W3up2}")
print(f"Updated W2 after 2 iterations : {W2up2}")
print(f"Updated W1 after 2 iterations : {W1up2}")
print(f"Updated B4 after 2 iterations : {B4up2}")
print(f"Updated B3 after 2 iterations : {B3up2}")
print(f"Updated B2 after 2 iterations : {B2up2}")
print(f"Updated B1 after 2 iterations : {B1up2}")

```

Figure 6: Error calculator

```
Error for first feed forward: 2.0
Updated W4 after 1 iteration : [[-4.84]
[-4.64]]
Updated W3 after 1 iteration : [[14.12 14.22]
[14.32 14.42]
[14.52 14.62]]
Updated W2 after 1 iteration : [[2.15 2.25 2.35]
[7.45 7.55 7.65]]
Updated W1 after 1 iteration : [[0.12 0.27]
[0.32 0.47]
[0.52 0.47]]
Updated B4 after 1 iteration : [[5.26]]
Updated B3 after 1 iteration : [[0.37]
[0.47]]
Updated B2 after 1 iteration : [[9.15]
[9.25]
[9.35]]
Updated B1 after 1 iteration : [[0.21]
[0.72]]
```

```
Error for second feed forward: 24.5
Updated W4 after 2 iterations : [[-4.84]
[-4.64]]
Updated W3 after 2 iterations : [[14.12 14.22]
[14.32 14.42]
[14.52 14.62]]
Updated W2 after 2 iterations : [[2.15 2.25 2.35]
[7.45 7.55 7.65]]
Updated W1 after 2 iterations : [[0.12 0.27]
[0.32 0.47]
[0.52 0.47]]
Updated B4 after 2 iterations : [[5.26]]
Updated B3 after 2 iterations : [[0.37]
[0.47]]
Updated B2 after 2 iterations : [[9.15]
[9.25]
[9.35]]
Updated B1 after 2 iterations : [[0.21]
[0.72]]
```

Figure 7: Error calculator

2 Question 2 : Application of neural networks in classification

2.1 Prerequisites

First of all, we shall depict the imported things used in this section.

```
import numpy as np
import pandas as pd
import seaborn as sn
from matplotlib import pyplot as plt
import random
from matplotlib import pyplot
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score , recall_score,
 f1_score, accuracy_score
from sklearn.model_selection import train_test_split
from keras.datasets import cifar10
from keras.callbacks import EarlyStopping
from keras.layers import BatchNormalization
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D , Flatten , Dense ,
  MaxPooling2D , Dropout
from tensorflow.keras.optimizers import SGD , Adam
```

Figure 8: Imported libraries

```
def Loader():
(X_train, Y_train), (X_test, Y_test) = cifar10.load_data()
return X_train, Y_train, X_test, Y_test

```

Figure 9: Loader function

Now we go on to print the first ten photos of the **Cifar-10** datasets.

```
Cifar_Labels = ['airplane', 'automobile', 'bird', 'cat', 'deer
  ', 'dog', 'frog', 'horse', 'ship', 'truck']
X_train, Y_train, X_test, Y_test = Loader()
for i in range(10):
plt.subplot(2,5,i+1)
plt.imshow(X_train[i])
plt.title(Cifar_Labels[Y_train[i][0]])

```

Figure 10: First ten photos

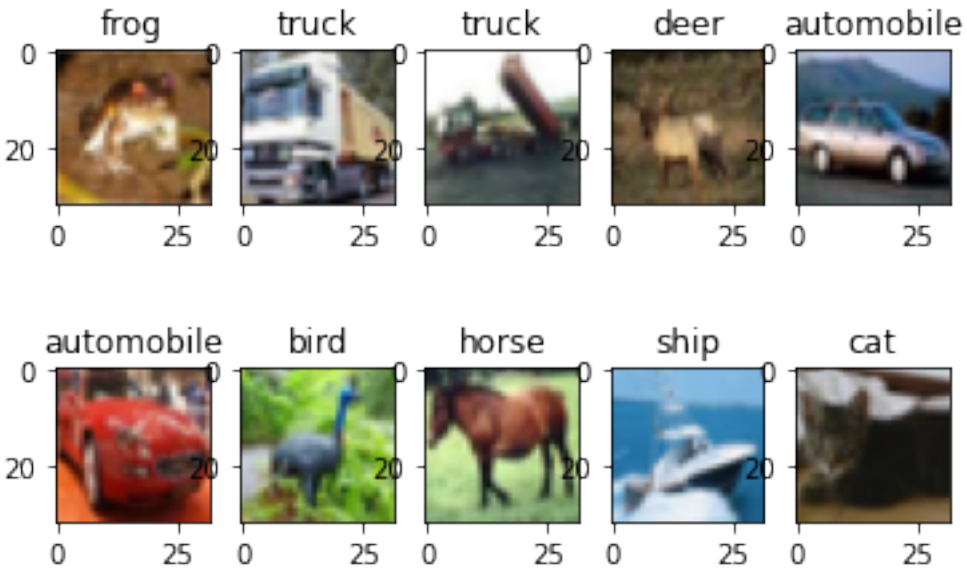

Figure 11: First ten photos

Now we shall define the rest of our needed functions including the **MLP**.

```
def preprocessing(X_train, Y_train, X_test, Y_test):
# Transform label indices to one-hot encoded vectors
Y_train = to_categorical(Y_train,num_classes = len(
  Cifar_Labels))
Y_test = to_categorical(Y_test,num_classes = len(Cifar_Labels)
  )
#Because working with 4 dimensions is not possible but is
  possible in cnns, we
#change the dimensions
X_train = np.reshape(X_train,(X_train.shape[0],32*32*3))
X_test = np.reshape(X_test,(X_test.shape[0],32*32*3))
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
#Here we perform normalization
X_train = X_train/255
X_test = X_test/255
#Test data must be kept unseen from the model until final
  evaluation hence we need
#validation dataset.
_, X_valid, _, Y_valid = train_test_split(X_train, Y_train,
test_size=0.2, random_state= 8)
return X_train, Y_train, X_test, Y_test, X_valid,Y_valid

```

Figure 12: Preprocessing

```
def Acc_plot(hist):
fig = plt.figure()
plt.plot(hist.history['accuracy'],'r')
plt.plot(hist.history['val_accuracy'],'b')
plt.title('Model Accuracy')
plt.ylabel('Acc')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'])
plt.grid()

```

Figure 13: Accuracy plot

```
def loss_plot(hist):
fig = plt.figure()
plt.plot(hist.history['loss'],'r')
plt.plot(hist.history['val_loss'],'b')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'])
plt.grid()

```

Figure 14: Loss plot

After this we have the functions needed to perform analysis after training the models.

```
def CompleteAnalysis(model , x_test , y_test ):
Y_pred = model.model.predict(x_test)
y_pred = np.argmax(Y_pred, axis=1)
Y_test = np.argmax(y_test, axis=1)
test_loss,test_accuracy = model.model.evaluate(x_test,y_test)
print('test loss = %f' % test_loss)
print('test accuracy = %f' % test_accuracy)
# accuracy: (tp + tn) / (p + n)
accuracy = accuracy_score(Y_test, y_pred)
print('Accuracy: %f' % accuracy)
# precision tp / (tp + fp)
precision = precision_score(Y_test, y_pred, average='macro')
print('Precision: %f' % precision)
# recall: tp / (tp + fn)
recall = recall_score(Y_test, y_pred, average='macro')
print('Recall: %f' % recall)
# f1: 2 tp / (2 tp + fp + fn)
f1 = f1_score(Y_test, y_pred, average='macro')
print('F1 score: %f' % f1)
#confMat
cm= confusion_matrix(Y_test, y_pred)
fig , ax = plt.subplots(figsize=(10,10))
sn.heatmap(cm, annot=True,square = True,ax=ax)
plt.xlabel('predicted value')
plt.ylabel('true value');

```

Figure 15: Complete Analysis(including precision, recall and f1-score)

2.2 Part 1 : Defining the network and testing for different batch sizes

Here we define the neural network as needed and observe the results with three different batch sizes.

2.2.1 The MLP Network

The network is depicted as follows.

```
def NeuralNetwork(Ac1,Ac2,Ac3,l,opt):
model = Sequential()
model.add(Dense(128,activation=Ac1,input_dim = 32*32*3))
model.add(Dense(128,activation=Ac2))
model.add(Dense(10,activation=Ac3))
model.compile(loss=l, optimizer=opt, metrics=['accuracy'])
return model

```

Figure 16: The MLP Network

In the next few pages I have included the code to train(for one example because they are exactly the same) and the results accordingly.

Batch Size = 32

The code to train is as follows.

```
X_train, Y_train, X_test, Y_test = Loader()
X_train, Y_train, X_test, Y_test, X_valid,Y_valid =
 preprocessing(X_train, Y_train, X_test, Y_test)
model32 = NeuralNetwork('relu','relu','softmax','
 categorical_crossentropy','sgd')
history32 = model32.fit(X_train,Y_train, epochs=15, batch_size
 =32, validation_data=(X_valid, Y_valid),verbose=1)

```

Figure 17: Code to train

```
Epoch 1/15
1563/1563 [==============================] - 16s 9ms/step - loss: 1.8948 - accuracy: 0.3189 - val_loss:
    1.7492 - val_accuracy: 0.3777
Epoch 2/15
1563/1563 [==============================] - 15s 9ms/step - loss: 1.7112 - accuracy: 0.3914 - val_loss:
    1.6516 - val_accuracy: 0.4172
Epoch 3/15
1563/1563 [==============================] - 14s 9ms/step - loss: 1.6311 - accuracy: 0.4238 - val_loss:
    1.6506 - val_accuracy: 0.4101
Epoch 4/15
1563/1563 [==============================] - 13s 9ms/step - loss: 1.5785 - accuracy: 0.4421 - val_loss:
    1.7122 - val_accuracy: 0.3937
Epoch 5/15
1563/1563 [==============================] - 13s 8ms/step - loss: 1.5369 - accuracy: 0.4561 - val_loss:
    1.5977 - val_accuracy: 0.4156
Epoch 6/15
1563/1563 [==============================] - 13s 8ms/step - loss: 1.5002 - accuracy: 0.4706 - val_loss:
    1.4773 - val_accuracy: 0.4691
Epoch 7/15
1563/1563 [==============================] - 13s 8ms/step - loss: 1.4681 - accuracy: 0.4800 - val_loss:
    1.4910 - val_accuracy: 0.4694
Epoch 8/15
1563/1563 [==============================] - 11s 7ms/step - loss: 1.4442 - accuracy: 0.4893 - val_loss:
    1.4178 - val_accuracy: 0.4942
Epoch 9/15
1563/1563 [==============================] - 11s 7ms/step - loss: 1.4192 - accuracy: 0.4977 - val_loss:
    1.3889 - val_accuracy: 0.5111
Epoch 10/15
1563/1563 [==============================] - 11s 7ms/step - loss: 1.3985 - accuracy: 0.5057 - val_loss:
    1.4229 - val_accuracy: 0.4960
Epoch 11/15
1563/1563 [==============================] - 10s 7ms/step - loss: 1.3774 - accuracy: 0.5134 - val_loss:
    1.3623 - val_accuracy: 0.5166
Epoch 12/15
1563/1563 [==============================] - 14s 9ms/step - loss: 1.3579 - accuracy: 0.5213 - val_loss:
    1.3133 - val_accuracy: 0.5336
Epoch 13/15
1563/1563 [==============================] - 14s 9ms/step - loss: 1.3422 - accuracy: 0.5243 - val_loss:
    1.3074 - val_accuracy: 0.5373
Epoch 14/15
1563/1563 [==============================] - 14s 9ms/step - loss: 1.3265 - accuracy: 0.5292 - val_loss:
    1.3111 - val_accuracy: 0.5342
Epoch 15/15
1563/1563 [==============================] - 14s 9ms/step - loss: 1.3094 - accuracy: 0.5362 - val_loss:
    1.3515 - val_accuracy: 0.5133
```

Figure 18: The Results for Batch = 32

Batch Size = 64

```
Epoch 1/15
782/782 [==============================] - 8s 9ms/step - loss: 1.9581 - accuracy: 0.2999 - val_loss:
    1.8961 - val_accuracy: 0.3291
Epoch 2/15
782/782 [==============================] - 8s 10ms/step - loss: 1.7871 - accuracy: 0.3704 - val_loss:
    1.8790 - val_accuracy: 0.3327
Epoch 3/15
782/782 [==============================] - 7s 9ms/step - loss: 1.7086 - accuracy: 0.3974 - val_loss:
    1.8043 - val_accuracy: 0.3522
Epoch 4/15
782/782 [==============================] - 8s 10ms/step - loss: 1.6525 - accuracy: 0.4188 - val_loss:
    1.7471 - val_accuracy: 0.3689
Epoch 5/15
782/782 [==============================] - 7s 9ms/step - loss: 1.6090 - accuracy: 0.4343 - val_loss:
    1.6453 - val_accuracy: 0.4077
Epoch 6/15
782/782 [==============================] - 8s 10ms/step - loss: 1.5742 - accuracy: 0.4462 - val_loss:
    1.6363 - val_accuracy: 0.4111
Epoch 7/15
782/782 [==============================] - 7s 9ms/step - loss: 1.5447 - accuracy: 0.4531 - val_loss:
    1.5514 - val_accuracy: 0.4416
Epoch 8/15
782/782 [==============================] - 7s 10ms/step - loss: 1.5163 - accuracy: 0.4668 - val_loss:
    1.5987 - val_accuracy: 0.4280
Epoch 9/15
782/782 [==============================] - 8s 10ms/step - loss: 1.4940 - accuracy: 0.4732 - val_loss:
    1.5079 - val_accuracy: 0.4683
Epoch 10/15
782/782 [==============================] - 6s 8ms/step - loss: 1.4699 - accuracy: 0.4821 - val_loss:
    1.4759 - val_accuracy: 0.4766
Epoch 11/15
782/782 [==============================] - 7s 9ms/step - loss: 1.4542 - accuracy: 0.4876 - val_loss:
    1.4351 - val_accuracy: 0.4933
Epoch 12/15
782/782 [==============================] - 7s 9ms/step - loss: 1.4374 - accuracy: 0.4934 - val_loss:
    1.5039 - val_accuracy: 0.4692
Epoch 13/15
782/782 [==============================] - 7s 8ms/step - loss: 1.4216 - accuracy: 0.4978 - val_loss:
    1.5382 - val_accuracy: 0.4617
Epoch 14/15
782/782 [==============================] - 7s 8ms/step - loss: 1.4086 - accuracy: 0.5022 - val_loss:
    1.5301 - val_accuracy: 0.4569
Epoch 15/15
782/782 [==============================] - 6s 8ms/step - loss: 1.3912 - accuracy: 0.5099 - val_loss:
    1.6270 - val_accuracy: 0.4274
```

Figure 19: The Results for Batch = 64

Batch Size = 256

```
Epoch 1/15
196/196 [==============================] - 4s 15ms/step - loss: 2.0790 - accuracy: 0.2523 - val_loss:
    1.9610 - val_accuracy: 0.2967
Epoch 2/15
196/196 [==============================] - 3s 14ms/step - loss: 1.9140 - accuracy: 0.3202 - val_loss:
    1.9085 - val_accuracy: 0.3077
Epoch 3/15
196/196 [==============================] - 3s 13ms/step - loss: 1.8535 - accuracy: 0.3457 - val_loss:
    1.8299 - val_accuracy: 0.3522
Epoch 4/15
196/196 [==============================] - 3s 13ms/step - loss: 1.8113 - accuracy: 0.3626 - val_loss:
    1.8077 - val_accuracy: 0.3552
Epoch 5/15
196/196 [==============================] - 3s 13ms/step - loss: 1.7784 - accuracy: 0.3735 - val_loss:
    1.7590 - val_accuracy: 0.3814
Epoch 6/15
196/196 [==============================] - 3s 14ms/step - loss: 1.7512 - accuracy: 0.3866 - val_loss:
    1.7479 - val_accuracy: 0.3850
Epoch 7/15
196/196 [==============================] - 3s 14ms/step - loss: 1.7260 - accuracy: 0.3956 - val_loss:
    1.8168 - val_accuracy: 0.3464
Epoch 8/15
196/196 [==============================] - 3s 14ms/step - loss: 1.7052 - accuracy: 0.4023 - val_loss:
    1.7585 - val_accuracy: 0.3658
Epoch 9/15
196/196 [==============================] - 3s 13ms/step - loss: 1.6877 - accuracy: 0.4087 - val_loss:
    1.7107 - val_accuracy: 0.3915
Epoch 10/15
196/196 [==============================] - 3s 13ms/step - loss: 1.6681 - accuracy: 0.4144 - val_loss:
    1.6977 - val_accuracy: 0.4014
Epoch 11/15
196/196 [==============================] - 3s 14ms/step - loss: 1.6545 - accuracy: 0.4203 - val_loss:
    1.6607 - val_accuracy: 0.4142
Epoch 12/15
196/196 [==============================] - 3s 14ms/step - loss: 1.6406 - accuracy: 0.4251 - val_loss:
    1.6391 - val_accuracy: 0.4260
Epoch 13/15
196/196 [==============================] - 3s 13ms/step - loss: 1.6238 - accuracy: 0.4327 - val_loss:
    1.6755 - val_accuracy: 0.4127
Epoch 14/15
196/196 [==============================] - 3s 13ms/step - loss: 1.6159 - accuracy: 0.4350 - val_loss:
    1.6197 - val_accuracy: 0.4300
Epoch 15/15
196/196 [==============================] - 3s 14ms/step - loss: 1.6018 - accuracy: 0.4403 - val_loss:
    1.6416 - val_accuracy: 0.4193
```

Figure 20: The Results for Batch = 256

As we can see the train and validation accuracy decreases with increasing the batch size, larger batch sizes take less time to train and but yield to lesser accuracies.

2.3 Part 2 : Changing activation functions

Here we shall change the activation functions for the layers, 3 times.

2.3.1 Sigmoid

```
Epoch 1/15
782/782 [==============================] - 7s 8ms/step - loss: 2.2887 - accuracy: 0.1620 - val_loss:
    2.2634 - val_accuracy: 0.1799
Epoch 2/15
782/782 [==============================] - 3s 4ms/step - loss: 2.2374 - accuracy: 0.2293 - val_loss:
    2.2067 - val_accuracy: 0.2756
Epoch 3/15
782/782 [==============================] - 3s 4ms/step - loss: 2.1701 - accuracy: 0.2582 - val_loss:
    2.1320 - val_accuracy: 0.2792
Epoch 4/15
782/782 [==============================] - 3s 4ms/step - loss: 2.0983 - accuracy: 0.2745 - val_loss:
    2.0671 - val_accuracy: 0.2806
Epoch 5/15
782/782 [==============================] - 3s 4ms/step - loss: 2.0415 - accuracy: 0.2854 - val_loss:
    2.0198 - val_accuracy: 0.2892
Epoch 6/15
782/782 [==============================] - 3s 4ms/step - loss: 1.9987 - accuracy: 0.2954 - val_loss:
    1.9852 - val_accuracy: 0.3018
Epoch 7/15
782/782 [==============================] - 3s 4ms/step - loss: 1.9657 - accuracy: 0.3025 - val_loss:
    1.9520 - val_accuracy: 0.3062
Epoch 8/15
782/782 [==============================] - 3s 4ms/step - loss: 1.9396 - accuracy: 0.3133 - val_loss:
    1.9298 - val_accuracy: 0.3068
Epoch 9/15
782/782 [==============================] - 3s 3ms/step - loss: 1.9187 - accuracy: 0.3194 - val_loss:
    1.9121 - val_accuracy: 0.3175
Epoch 10/15
782/782 [==============================] - 3s 4ms/step - loss: 1.9022 - accuracy: 0.3254 - val_loss:
    1.8946 - val_accuracy: 0.3250
Epoch 11/15
782/782 [==============================] - 3s 4ms/step - loss: 1.8880 - accuracy: 0.3324 - val_loss:
    1.8808 - val_accuracy: 0.3335
Epoch 12/15
782/782 [==============================] - 3s 3ms/step - loss: 1.8756 - accuracy: 0.3358 - val_loss:
    1.8678 - val_accuracy: 0.3335
Epoch 13/15
782/782 [==============================] - 3s 3ms/step - loss: 1.8640 - accuracy: 0.3407 - val_loss:
    1.8618 - val_accuracy: 0.3393
Epoch 14/15
782/782 [==============================] - 3s 4ms/step - loss: 1.8532 - accuracy: 0.3456 - val_loss:
    1.8479 - val_accuracy: 0.3459
Epoch 15/15
782/782 [==============================] - 3s 4ms/step - loss: 1.8432 - accuracy: 0.3475 - val_loss:
    1.8393 - val_accuracy: 0.3512


```

Figure 21: Sigmoid Activation

Accuracy Plot

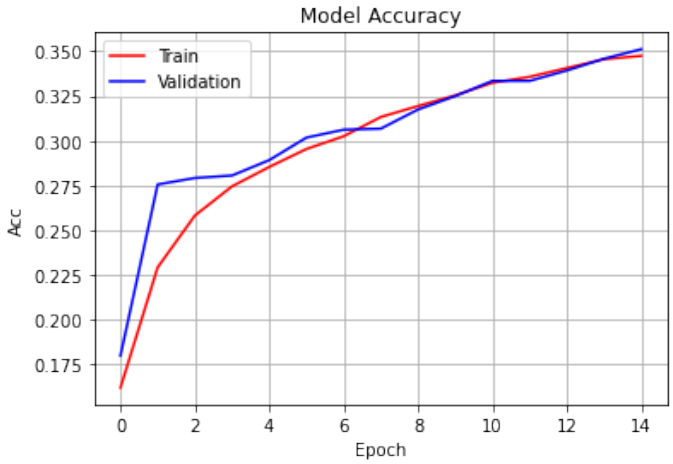

Figure 22: Accuracy Plot

Loss Plot

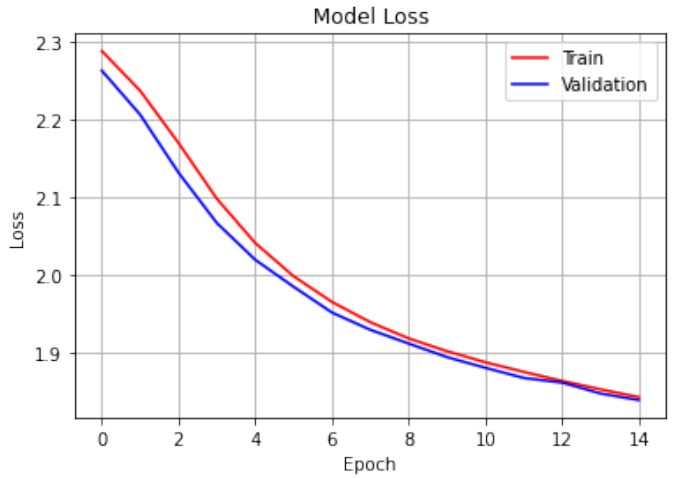

Figure 23: Loss Plot

Confusion matrix and accuracy and error

```
1563/1563 [==============================] - 3s 2ms/step
1563/1563 [==============================] - 3s 2ms/step - loss: 1.8407 - accuracy: 0.3526
test loss = 1.840652
test accuracy = 0.352620

```

Figure 24: Accuracy and error for Sigmoid

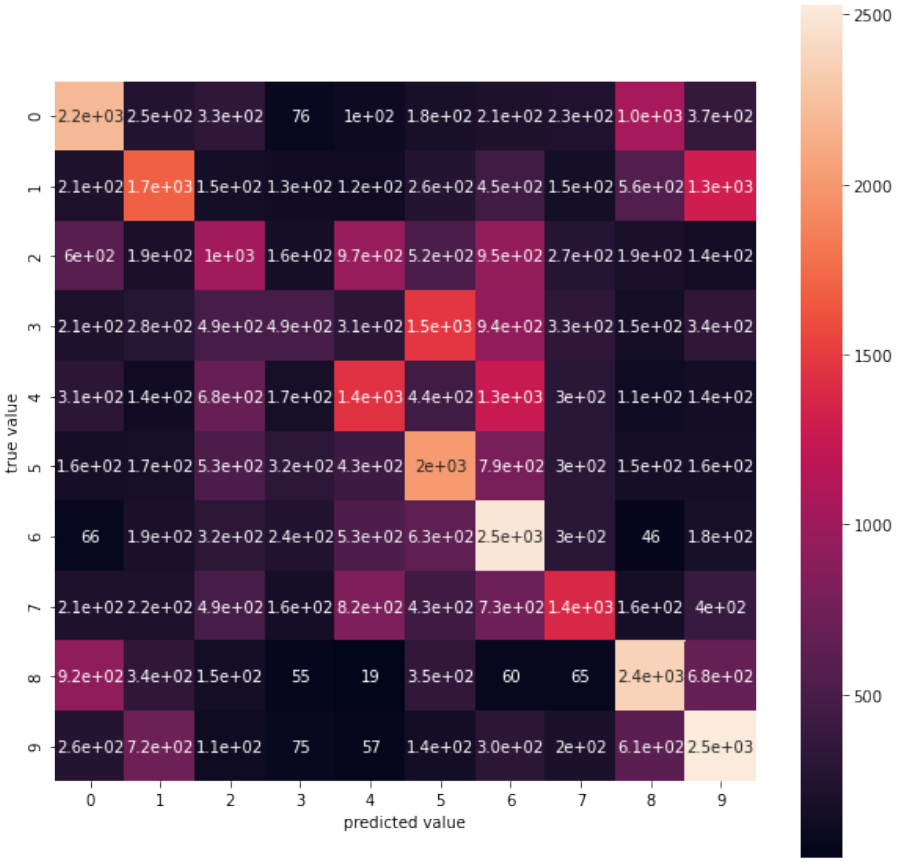

Figure 25: Confusion Matrix

2.3.2 tanh

```
Epoch 1/15
782/782 [==============================] - 3s 4ms/step - loss: 1.9343 - accuracy: 0.3067 - val_loss:
    1.8745 - val_accuracy: 0.3464
Epoch 2/15
782/782 [==============================] - 3s 3ms/step - loss: 1.7746 - accuracy: 0.3737 - val_loss:
    1.7369 - val_accuracy: 0.3925
Epoch 3/15
782/782 [==============================] - 3s 4ms/step - loss: 1.7095 - accuracy: 0.3993 - val_loss:
    1.7124 - val_accuracy: 0.3949
Epoch 4/15
782/782 [==============================] - 3s 4ms/step - loss: 1.6585 - accuracy: 0.4150 - val_loss:
    1.6400 - val_accuracy: 0.4169
Epoch 5/15
782/782 [==============================] - 3s 4ms/step - loss: 1.6211 - accuracy: 0.4273 - val_loss:
    1.6150 - val_accuracy: 0.4242
Epoch 6/15
782/782 [==============================] - 3s 4ms/step - loss: 1.5914 - accuracy: 0.4384 - val_loss:
    1.6131 - val_accuracy: 0.4212
Epoch 7/15
782/782 [==============================] - 3s 4ms/step - loss: 1.5663 - accuracy: 0.4466 - val_loss:
    1.5970 - val_accuracy: 0.4265
Epoch 8/15
782/782 [==============================] - 3s 4ms/step - loss: 1.5446 - accuracy: 0.4541 - val_loss:
    1.5479 - val_accuracy: 0.4451
Epoch 9/15
782/782 [==============================] - 3s 3ms/step - loss: 1.5247 - accuracy: 0.4612 - val_loss:
    1.5227 - val_accuracy: 0.4623
Epoch 10/15
782/782 [==============================] - 3s 4ms/step - loss: 1.5083 - accuracy: 0.4643 - val_loss:
    1.5500 - val_accuracy: 0.4513
Epoch 11/15
782/782 [==============================] - 3s 3ms/step - loss: 1.4910 - accuracy: 0.4764 - val_loss:
    1.6814 - val_accuracy: 0.3894
Epoch 12/15
782/782 [==============================] - 3s 3ms/step - loss: 1.4766 - accuracy: 0.4782 - val_loss:
    1.5115 - val_accuracy: 0.4608
Epoch 13/15
782/782 [==============================] - 3s 3ms/step - loss: 1.4613 - accuracy: 0.4850 - val_loss:
    1.5374 - val_accuracy: 0.4525
Epoch 14/15
782/782 [==============================] - 3s 4ms/step - loss: 1.4452 - accuracy: 0.4891 - val_loss:
    1.4481 - val_accuracy: 0.4845
Epoch 15/15
782/782 [==============================] - 3s 3ms/step - loss: 1.4311 - accuracy: 0.4957 - val_loss:
    1.5060 - val_accuracy: 0.4570


```

Figure 26: tanh Activation

Accuracy Plot

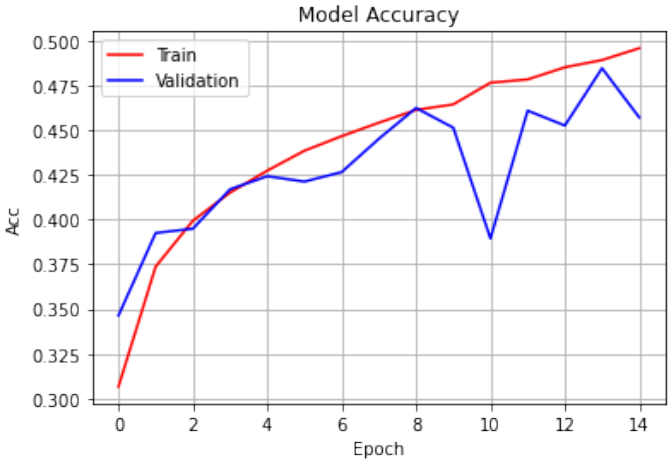

Figure 27: Accuracy Plot

Loss Plot

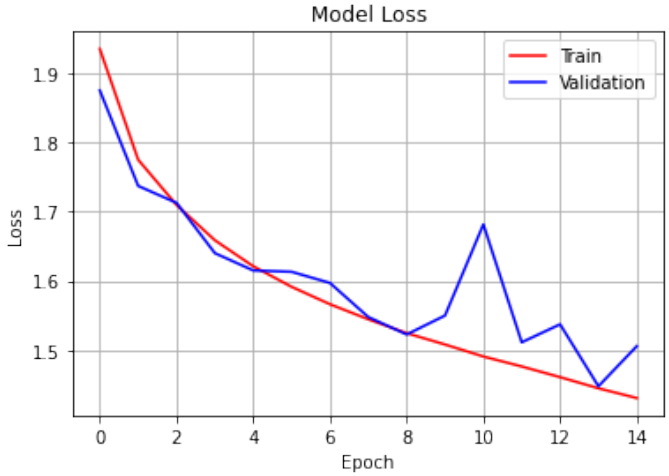

Figure 28: Loss Plot

Confusion matrix and accuracy and error

```
1563/1563 [==============================] - 2s 2ms/step
1563/1563 [==============================] - 2s 2ms/step -
 loss: 1.5082 - accuracy: 0.4604
test loss = 1.508156
test accuracy = 0.460420

```

Figure 29: Accuracy and error for tanh

Figure 30: Confusion Matrix

2.3.3 Selu

```
Epoch 1/15
782/782 [==============================] - 3s 4ms/step - loss: 1.9480 - accuracy: 0.3061 - val_loss:
    1.9291 - val_accuracy: 0.2995
Epoch 2/15
782/782 [==============================] - 3s 4ms/step - loss: 1.7708 - accuracy: 0.3736 - val_loss:
    1.9125 - val_accuracy: 0.3250
Epoch 3/15
782/782 [==============================] - 3s 4ms/step - loss: 1.7001 - accuracy: 0.4004 - val_loss:
    1.6834 - val_accuracy: 0.4030
Epoch 4/15
782/782 [==============================] - 3s 4ms/step - loss: 1.6496 - accuracy: 0.4167 - val_loss:
    1.8410 - val_accuracy: 0.3571
Epoch 5/15
782/782 [==============================] - 3s 4ms/step - loss: 1.6107 - accuracy: 0.4302 - val_loss:
    1.6446 - val_accuracy: 0.4066
Epoch 6/15
782/782 [==============================] - 3s 4ms/step - loss: 1.5779 - accuracy: 0.4443 - val_loss:
    1.6786 - val_accuracy: 0.4084
Epoch 7/15
782/782 [==============================] - 3s 4ms/step - loss: 1.5564 - accuracy: 0.4507 - val_loss:
    1.6879 - val_accuracy: 0.3967
Epoch 8/15
782/782 [==============================] - 3s 4ms/step - loss: 1.5307 - accuracy: 0.4598 - val_loss:
    1.8416 - val_accuracy: 0.3590
Epoch 9/15
782/782 [==============================] - 3s 3ms/step - loss: 1.5115 - accuracy: 0.4680 - val_loss:
    1.5949 - val_accuracy: 0.4115
Epoch 10/15
782/782 [==============================] - 3s 4ms/step - loss: 1.4937 - accuracy: 0.4730 - val_loss:
    1.5157 - val_accuracy: 0.4605
Epoch 11/15
782/782 [==============================] - 3s 4ms/step - loss: 1.4725 - accuracy: 0.4814 - val_loss:
    1.5171 - val_accuracy: 0.4559
Epoch 12/15
782/782 [==============================] - 3s 4ms/step - loss: 1.4564 - accuracy: 0.4866 - val_loss:
    1.5869 - val_accuracy: 0.4358
Epoch 13/15
782/782 [==============================] - 3s 4ms/step - loss: 1.4400 - accuracy: 0.4924 - val_loss:
    1.4580 - val_accuracy: 0.4755
Epoch 14/15
782/782 [==============================] - 3s 4ms/step - loss: 1.4286 - accuracy: 0.4980 - val_loss:
    1.4395 - val_accuracy: 0.4798
Epoch 15/15
782/782 [==============================] - 3s 4ms/step - loss: 1.4134 - accuracy: 0.5022 - val_loss:
    1.4289 - val_accuracy: 0.4892


```

Figure 31: Selu Activation

Accuracy Plot

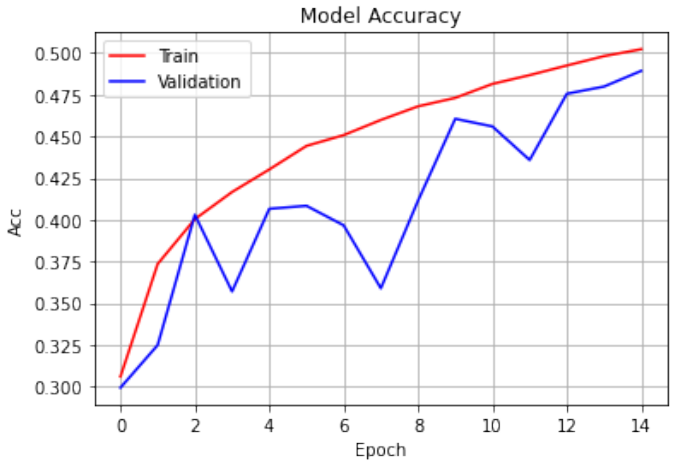

Figure 32: Accuracy Plot

Loss Plot

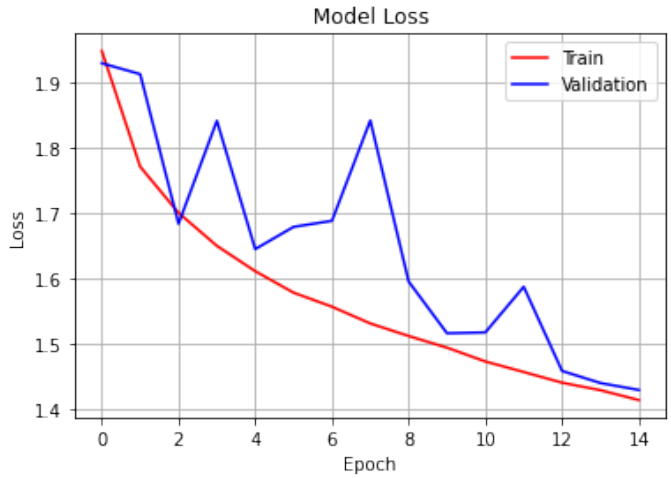

Figure 33: Loss Plot

Confusion matrix and accuracy and error

```
1563/1563 [==============================] - 2s 2ms/step
1563/1563 [==============================] - 3s 2ms/step - loss: 1.4360 - accuracy: 0.4880
test loss = 1.436009
test accuracy = 0.488000

```

Figure 34: Accuracy and error for Selu

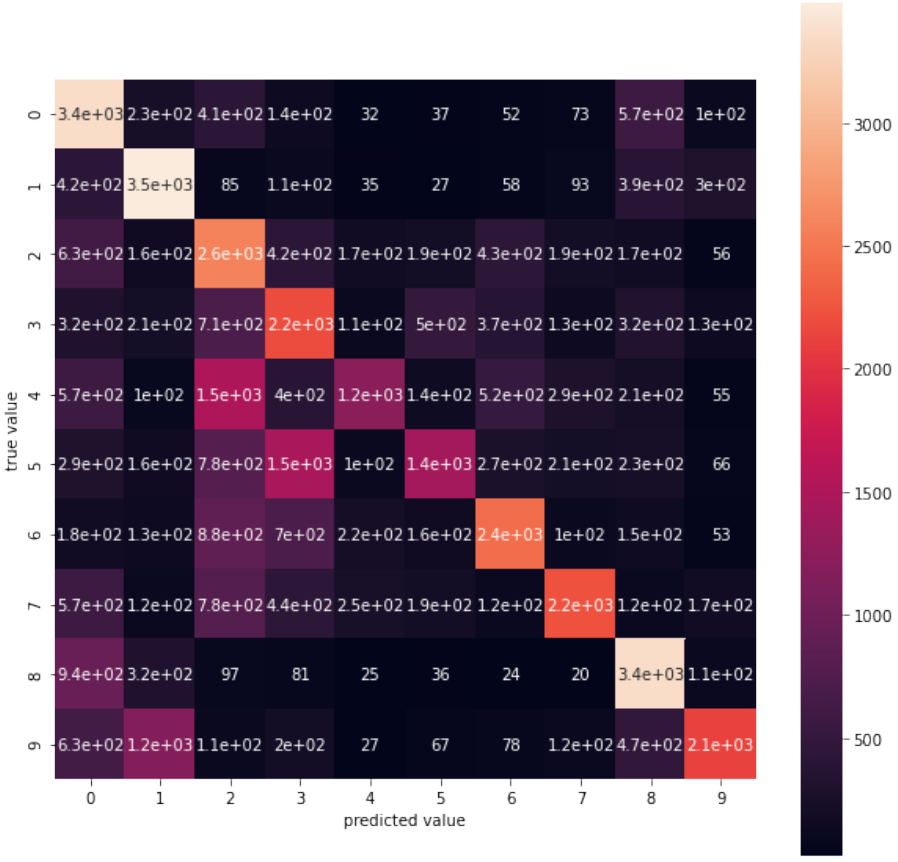

Figure 35: Confusion Matrix

As we can see still the *relu* activation works better than the rest. Now we shall talk about the pros and cons of the used activations here.

Sigmoid Advantages

- It is one of the best normalized functions
- Has a clear prediction
- The gradient is smooth

Sigmoid Disdvantages

- Computationally expensive
- Vanishing gradient can occur
- Always gives positive values

tanh **Advantages**

- It is zero centric
- The gradient is smooth

tanh **Disdvantages**

- Computationally expensive
- Vanishing gradient can occur

Selu

It is very similar to Relu but with an advantage, internal normalization is faster than external normalization, which means the network converges faster.

2.4 Part 3 : Changing the loss functions

Here we change the error function, 2 times.

2.4.1 Poisson loss

```
Epoch 1/15
782/782 [==============================] - 4s 4ms/step - loss: 0.3218 - accuracy: 0.2045 - val_loss:
    0.3132 - val_accuracy: 0.2530
Epoch 2/15
782/782 [==============================] - 3s 3ms/step - loss: 0.3071 - accuracy: 0.2743 - val_loss:
    0.3025 - val_accuracy: 0.2835
Epoch 3/15
782/782 [==============================] - 3s 3ms/step - loss: 0.2987 - accuracy: 0.3042 - val_loss:
    0.2958 - val_accuracy: 0.3052
Epoch 4/15
782/782 [==============================] - 3s 3ms/step - loss: 0.2935 - accuracy: 0.3208 - val_loss:
    0.2919 - val_accuracy: 0.3212
Epoch 5/15
782/782 [==============================] - 3s 3ms/step - loss: 0.2900 - accuracy: 0.3358 - val_loss:
    0.2886 - val_accuracy: 0.3381
Epoch 6/15
782/782 [==============================] - 3s 3ms/step - loss: 0.2872 - accuracy: 0.3466 - val_loss:
    0.2861 - val_accuracy: 0.3450
Epoch 7/15
782/782 [==============================] - 3s 3ms/step - loss: 0.2850 - accuracy: 0.3554 - val_loss:
    0.2841 - val_accuracy: 0.3541
Epoch 8/15
782/782 [==============================] - 3s 3ms/step - loss: 0.2831 - accuracy: 0.3610 - val_loss:
    0.2824 - val_accuracy: 0.3549
Epoch 9/15
782/782 [==============================] - 3s 3ms/step - loss: 0.2814 - accuracy: 0.3671 - val_loss:
    0.2809 - val_accuracy: 0.3637
Epoch 10/15
782/782 [==============================] - 3s 3ms/step - loss: 0.2800 - accuracy: 0.3717 - val_loss:
    0.2796 - val_accuracy: 0.3657
Epoch 11/15
782/782 [==============================] - 3s 3ms/step - loss: 0.2786 - accuracy: 0.3753 - val_loss:
    0.2781 - val_accuracy: 0.3746
Epoch 12/15
782/782 [==============================] - 2s 3ms/step - loss: 0.2774 - accuracy: 0.3807 - val_loss:
    0.2770 - val_accuracy: 0.3783
Epoch 13/15
782/782 [==============================] - 2s 3ms/step - loss: 0.2763 - accuracy: 0.3836 - val_loss:
    0.2757 - val_accuracy: 0.3815
Epoch 14/15
782/782 [==============================] - 2s 3ms/step - loss: 0.2752 - accuracy: 0.3884 - val_loss:
    0.2753 - val_accuracy: 0.3846
Epoch 15/15
782/782 [==============================] - 3s 3ms/step - loss: 0.2742 - accuracy: 0.3930 - val_loss:
    0.2734 - val_accuracy: 0.3929


```

Figure 36: Poissin Loss

Accuracy Plot

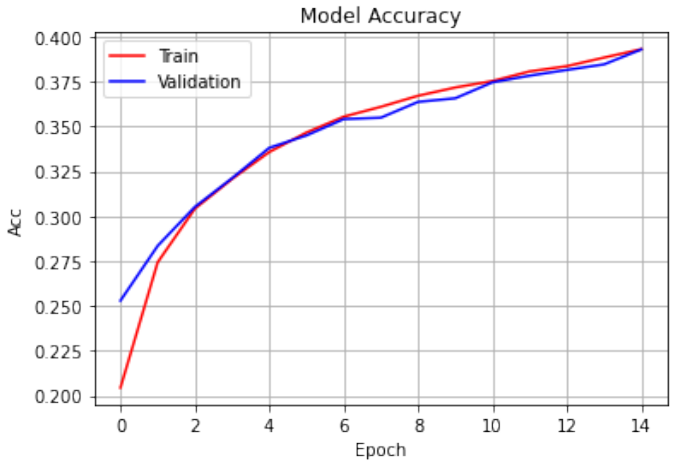

Figure 37: Accuracy Plot

Loss Plot

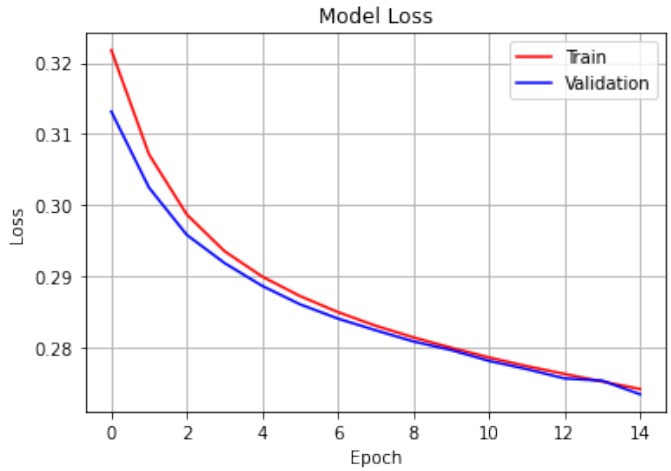

Figure 38: Loss Plot

Confusion matrix and accuracy and error

```
1563/1563 [==============================] - 2s 1ms/step
1563/1563 [==============================] - 3s 2ms/step - loss: 0.2735 - accuracy: 0.3956
test loss = 0.273509
test accuracy = 0.395600

```

Figure 39: Accuracy and error for Poisson Loss

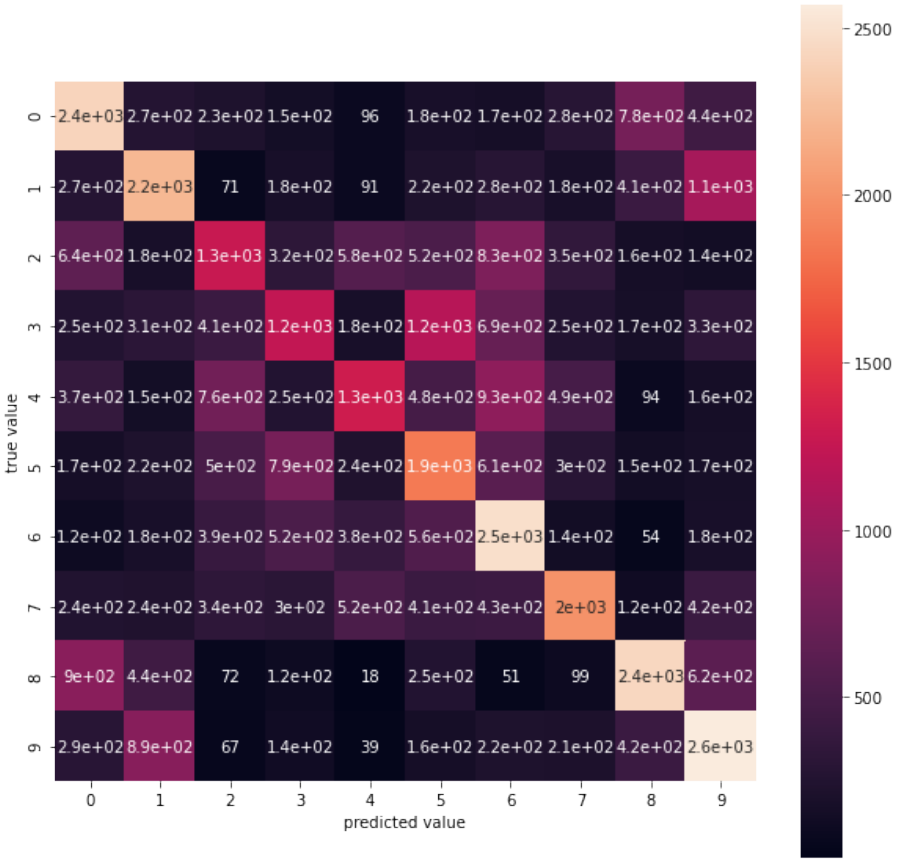

Figure 40: Confusion Matrix

2.4.2 Mean Squared Error loss

```
Epoch 1/15
782/782 [==============================] - 3s 4ms/step - loss: 0.0899 - accuracy: 0.1273 - val_loss:
    0.0892 - val_accuracy: 0.1582
Epoch 2/15
782/782 [==============================] - 3s 4ms/step - loss: 0.0886 - accuracy: 0.1836 - val_loss:
    0.0880 - val_accuracy: 0.2028
Epoch 3/15
782/782 [==============================] - 3s 4ms/step - loss: 0.0876 - accuracy: 0.2138 - val_loss:
    0.0871 - val_accuracy: 0.2289
Epoch 4/15
782/782 [==============================] - 3s 4ms/step - loss: 0.0866 - accuracy: 0.2361 - val_loss:
    0.0862 - val_accuracy: 0.2438
Epoch 5/15
782/782 [==============================] - 3s 3ms/step - loss: 0.0858 - accuracy: 0.2477 - val_loss:
    0.0854 - val_accuracy: 0.2546
Epoch 6/15
782/782 [==============================] - 3s 3ms/step - loss: 0.0850 - accuracy: 0.2574 - val_loss:
    0.0847 - val_accuracy: 0.2590
Epoch 7/15
782/782 [==============================] - 3s 3ms/step - loss: 0.0843 - accuracy: 0.2663 - val_loss:
    0.0840 - val_accuracy: 0.2690
Epoch 8/15
782/782 [==============================] - 3s 3ms/step - loss: 0.0837 - accuracy: 0.2763 - val_loss:
    0.0834 - val_accuracy: 0.2780
Epoch 9/15
782/782 [==============================] - 3s 3ms/step - loss: 0.0831 - accuracy: 0.2859 - val_loss:
    0.0828 - val_accuracy: 0.2847
Epoch 10/15
782/782 [==============================] - 2s 3ms/step - loss: 0.0825 - accuracy: 0.2931 - val_loss:
    0.0823 - val_accuracy: 0.2980
Epoch 11/15
782/782 [==============================] - 2s 3ms/step - loss: 0.0820 - accuracy: 0.3019 - val_loss:
    0.0819 - val_accuracy: 0.3028
Epoch 12/15
782/782 [==============================] - 2s 3ms/step - loss: 0.0816 - accuracy: 0.3096 - val_loss:
    0.0814 - val_accuracy: 0.3098
Epoch 13/15
782/782 [==============================] - 2s 3ms/step - loss: 0.0812 - accuracy: 0.3174 - val_loss:
    0.0810 - val_accuracy: 0.3153
Epoch 14/15
782/782 [==============================] - 2s 3ms/step - loss: 0.0808 - accuracy: 0.3239 - val_loss:
    0.0807 - val_accuracy: 0.3218
Epoch 15/15
782/782 [==============================] - 2s 3ms/step - loss: 0.0804 - accuracy: 0.3283 - val_loss:
    0.0802 - val_accuracy: 0.3269


```

Figure 41: Mean Squared Error Loss

Accuracy Plot

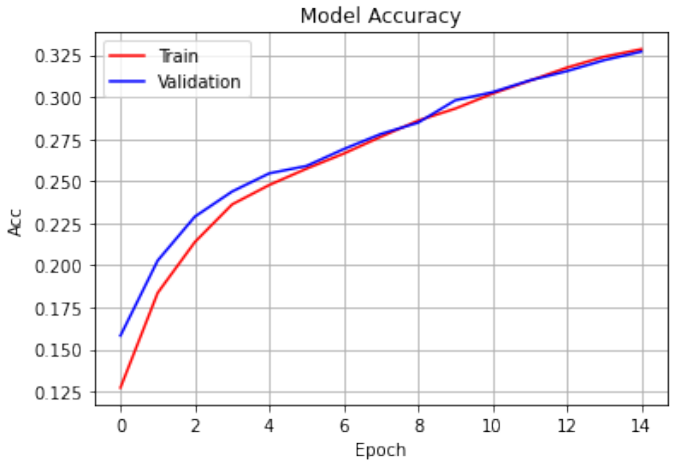

Figure 42: Accuracy Plot

Loss Plot

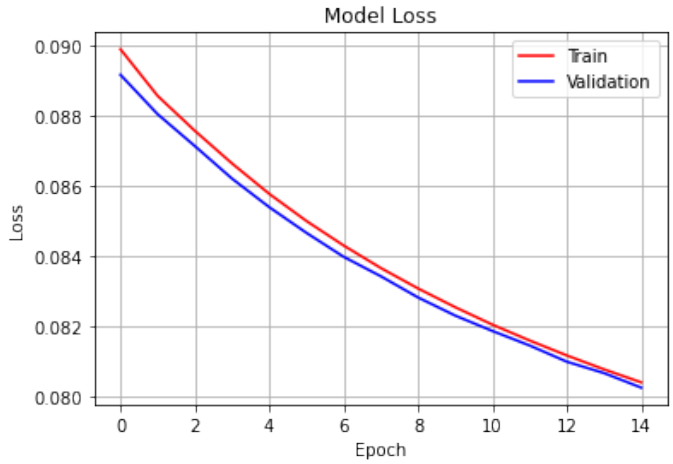

Figure 43: Loss Plot

Confusion matrix and accuracy and error

```
1563/1563 [==============================] - 2s 1ms/step
1563/1563 [==============================] - 3s 2ms/step - loss: 0.0802 - accuracy: 0.3327
test loss = 0.080179
test accuracy = 0.332700

```

Figure 44: Accuracy and error for Mean Squared Error Loss

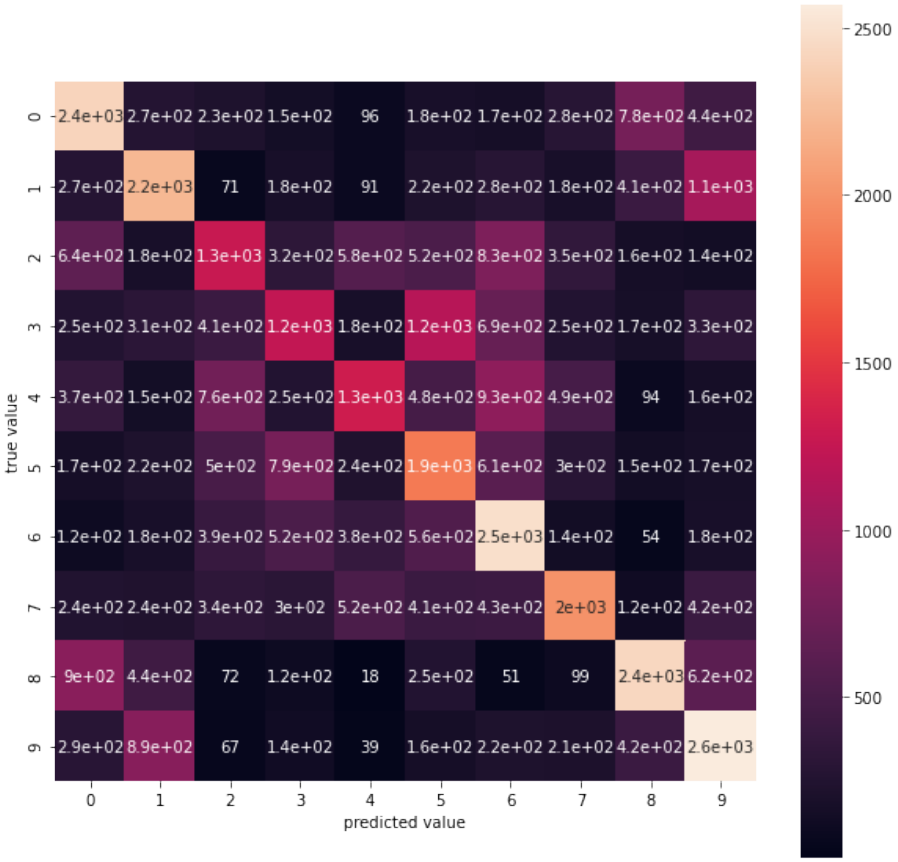

Figure 45: Confusion Matrix

Our results with categorical_crossentropy is still much better than these other two loss functions, I guess this because of the nature of our problem and the dataset we are using, for this problem the categorical_crossentropy loss is much better due to us using the **one-hot encoded** version of labels.

Poisson Regression

The Poisson regression is best used for modeling events where the outcomes are counts. To be more precise this loss function is best used when the outcomes are count data, something like non-negative integers, in this problem this is not our case therefore we don't get very good results.

Mean Squared Loss

As we know and can also interpret from the name of this loss function, it is normally used in regression task where the goal is to minimize the expected value of some sort of function on our training data, as shown in the **Stochastic Processes** course this loss is formulated as below.

$$MSE = \frac{1}{n}\sum(y - \hat{y})^2$$

In other words this loss function is used when we want to check how close are estimates are to actual values.

In our project, comparing all loss functions, the best one shall be the **categorical_crossentropy** loss.

2.5 Part 4 : Changing the optimizers

Here we shall change the optimizer of the network, 2 times.

2.5.1 Adam

```
Epoch 1/15
782/782 [==============================] - 5s 6ms/step - loss: 1.8580 - accuracy: 0.3291 - val_loss:
    1.7640 - val_accuracy: 0.3550
Epoch 2/15
782/782 [==============================] - 4s 5ms/step - loss: 1.6807 - accuracy: 0.3988 - val_loss:
    1.6233 - val_accuracy: 0.4169
Epoch 3/15
782/782 [==============================] - 4s 5ms/step - loss: 1.6076 - accuracy: 0.4239 - val_loss:
    1.5285 - val_accuracy: 0.4551
Epoch 4/15
782/782 [==============================] - 4s 5ms/step - loss: 1.5559 - accuracy: 0.4450 - val_loss:
    1.5821 - val_accuracy: 0.4325
Epoch 5/15
782/782 [==============================] - 4s 5ms/step - loss: 1.5287 - accuracy: 0.4573 - val_loss:
    1.4826 - val_accuracy: 0.4731
Epoch 6/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4997 - accuracy: 0.4656 - val_loss:
    1.4845 - val_accuracy: 0.4711
Epoch 7/15
782/782 [==============================] - 4s 6ms/step - loss: 1.4774 - accuracy: 0.4738 - val_loss:
    1.4901 - val_accuracy: 0.4686
Epoch 8/15
782/782 [==============================] - 4s 6ms/step - loss: 1.4579 - accuracy: 0.4801 - val_loss:
    1.4195 - val_accuracy: 0.4884
Epoch 9/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4302 - accuracy: 0.4893 - val_loss:
    1.4157 - val_accuracy: 0.4881
Epoch 10/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4206 - accuracy: 0.4903 - val_loss:
    1.4093 - val_accuracy: 0.4929
Epoch 11/15
782/782 [==============================] - 4s 6ms/step - loss: 1.3992 - accuracy: 0.5002 - val_loss:
    1.3666 - val_accuracy: 0.5150
Epoch 12/15
782/782 [==============================] - 4s 5ms/step - loss: 1.3915 - accuracy: 0.4996 - val_loss:
    1.3928 - val_accuracy: 0.4966
Epoch 13/15
782/782 [==============================] - 4s 5ms/step - loss: 1.3774 - accuracy: 0.5057 - val_loss:
    1.3549 - val_accuracy: 0.5232
Epoch 14/15
782/782 [==============================] - 4s 5ms/step - loss: 1.3663 - accuracy: 0.5114 - val_loss:
    1.3897 - val_accuracy: 0.5019
Epoch 15/15
782/782 [==============================] - 4s 5ms/step - loss: 1.3532 - accuracy: 0.5146 - val_loss:
    1.3468 - val_accuracy: 0.5167


```

Figure 46: Adam Optimizer

2.5.2 RMSprop

```
Epoch 1/15
782/782 [==============================] - 4s 5ms/step - loss: 1.9534 - accuracy: 0.2977 - val_loss:
    1.8734 - val_accuracy: 0.3299
Epoch 2/15
782/782 [==============================] - 4s 5ms/step - loss: 1.7362 - accuracy: 0.3749 - val_loss:
    1.8528 - val_accuracy: 0.3441
Epoch 3/15
782/782 [==============================] - 4s 5ms/step - loss: 1.6628 - accuracy: 0.4033 - val_loss:
    1.6178 - val_accuracy: 0.4163
Epoch 4/15
782/782 [==============================] - 4s 5ms/step - loss: 1.6131 - accuracy: 0.4245 - val_loss:
    1.7880 - val_accuracy: 0.3693
Epoch 5/15
782/782 [==============================] - 4s 5ms/step - loss: 1.5732 - accuracy: 0.4409 - val_loss:
    1.5200 - val_accuracy: 0.4563
Epoch 6/15
782/782 [==============================] - 4s 5ms/step - loss: 1.5441 - accuracy: 0.4479 - val_loss:
    1.6083 - val_accuracy: 0.4284
Epoch 7/15
782/782 [==============================] - 3s 4ms/step - loss: 1.5176 - accuracy: 0.4593 - val_loss:
    1.5595 - val_accuracy: 0.4421
Epoch 8/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4970 - accuracy: 0.4672 - val_loss:
    1.5623 - val_accuracy: 0.4330
Epoch 9/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4840 - accuracy: 0.4732 - val_loss:
    1.5004 - val_accuracy: 0.4614
Epoch 10/15
782/782 [==============================] - 3s 4ms/step - loss: 1.4679 - accuracy: 0.4770 - val_loss:
    1.4534 - val_accuracy: 0.4788
Epoch 11/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4498 - accuracy: 0.4851 - val_loss:
    1.5108 - val_accuracy: 0.4575
Epoch 12/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4394 - accuracy: 0.4883 - val_loss:
    1.4865 - val_accuracy: 0.4785
Epoch 13/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4268 - accuracy: 0.4922 - val_loss:
    1.6755 - val_accuracy: 0.4242
Epoch 14/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4184 - accuracy: 0.4972 - val_loss:
    1.3920 - val_accuracy: 0.5095
Epoch 15/15
782/782 [==============================] - 3s 4ms/step - loss: 1.4104 - accuracy: 0.4990 - val_loss:
    1.5326 - val_accuracy: 0.4620
```

Figure 47: RMSprop Optimizer

As we can see the **Adam** optimizer is a little better than the other ones, this is an SGD based optimizer which is frequently used in neural networks.

2.6 Part 5 : The best model

After studying the previous parts, we conclude that using **Relu** and **Softmax** activations, the **categorical_crossentropy** loss function and the **Adam** optimizer yields to the best networks.

We shall go on to train this model and get the accuracy for the test data. After this we shall check its precision, recall and f-score.

The code to train is as follows.

```
X_train, Y_train, X_test, Y_test = Loader()
X_train, Y_train, X_test, Y_test, X_valid, Y_valid =
  preprocessing(X_train, Y_train, X_test, Y_test)
model64adam = NeuralNetwork('relu','relu','softmax','
  categorical_crossentropy','adam')
history64adam = model64adam.fit(X_train,Y_train, epochs=15,
  batch_size=64, validation_data=(X_valid, Y_valid),verbose=1)

```

Figure 48: Code to train

```
1563/1563 [==============================] - 2s 1ms/step
1563/1563 [==============================] - 3s 2ms/step -
  loss: 1.3097 - accuracy: 0.5346
test loss = 1.309681
test accuracy = 0.534580
Accuracy: 0.534580
Precision: 0.535605
Recall: 0.534580
F1 score: 0.531291

```

Figure 49: Complete Analysis

```
Epoch 1/15
782/782 [==============================] - 5s 5ms/step - loss: 1.8566 - accuracy: 0.3291 - val_loss:
    1.7294 - val_accuracy: 0.3751
Epoch 2/15
782/782 [==============================] - 4s 5ms/step - loss: 1.6843 - accuracy: 0.3971 - val_loss:
    1.6944 - val_accuracy: 0.3895
Epoch 3/15
782/782 [==============================] - 4s 5ms/step - loss: 1.6147 - accuracy: 0.4220 - val_loss:
    1.6166 - val_accuracy: 0.4195
Epoch 4/15
782/782 [==============================] - 4s 5ms/step - loss: 1.5671 - accuracy: 0.4405 - val_loss:
    1.5939 - val_accuracy: 0.4327
Epoch 5/15
782/782 [==============================] - 4s 5ms/step - loss: 1.5293 - accuracy: 0.4526 - val_loss:
    1.5435 - val_accuracy: 0.4489
Epoch 6/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4946 - accuracy: 0.4676 - val_loss:
    1.4566 - val_accuracy: 0.4780
Epoch 7/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4755 - accuracy: 0.4745 - val_loss:
    1.4519 - val_accuracy: 0.4825
Epoch 8/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4519 - accuracy: 0.4811 - val_loss:
    1.4569 - val_accuracy: 0.4805
Epoch 9/15
782/782 [==============================] - 4s 5ms/step - loss: 1.4325 - accuracy: 0.4913 - val_loss:
    1.4046 - val_accuracy: 0.4980
Epoch 10/15
782/782 [==============================] - 4s 6ms/step - loss: 1.4058 - accuracy: 0.4974 - val_loss:
    1.3695 - val_accuracy: 0.5075
Epoch 11/15
782/782 [==============================] - 4s 5ms/step - loss: 1.3995 - accuracy: 0.5028 - val_loss:
    1.3467 - val_accuracy: 0.5180
Epoch 12/15
782/782 [==============================] - 4s 5ms/step - loss: 1.3784 - accuracy: 0.5091 - val_loss:
    1.3615 - val_accuracy: 0.5125
Epoch 13/15
782/782 [==============================] - 4s 5ms/step - loss: 1.3757 - accuracy: 0.5114 - val_loss:
    1.3614 - val_accuracy: 0.5222
Epoch 14/15
782/782 [==============================] - 4s 5ms/step - loss: 1.3526 - accuracy: 0.5187 - val_loss:
    1.3659 - val_accuracy: 0.5170
Epoch 15/15
782/782 [==============================] - 4s 5ms/step - loss: 1.3420 - accuracy: 0.5242 - val_loss:
    1.3011 - val_accuracy: 0.5394


```

Figure 50: The best model

Now we shall take a look at the confusion matrix.

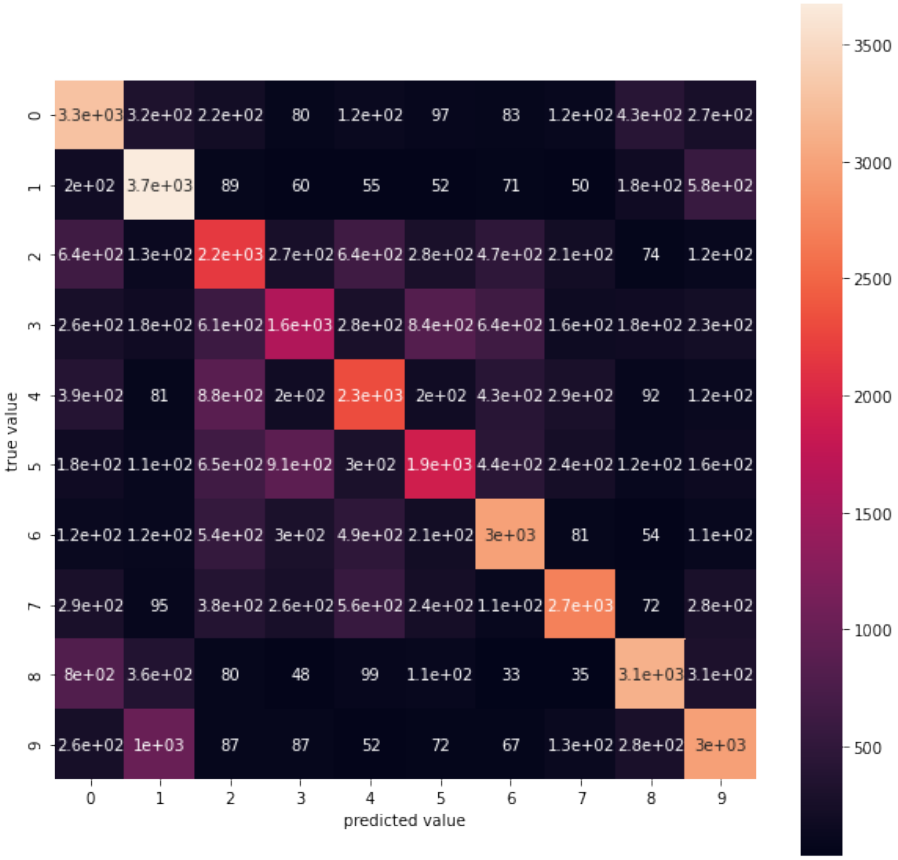

Figure 51: Confusion Matrix

This concludes are analysis of the **MLP**, next we shall move on to the **MLP+CNN** model.

MLP+CNN

First of all we shall include our new preprocessing method.

```
def preprocessingcnn(X_train, Y_train, X_test, Y_test):
# Transform label indices to one-hot encoded vectors
Y_train = to_categorical(Y_train,num_classes = len(Cifar_Labels))
Y_test = to_categorical(Y_test,num_classes = len(Cifar_Labels))
#Because working with 4 dimensions is not possible but is possible in cnns, we
#change the dimensions
# X_train = np.reshape(X_train,(X_train.shape[0],32*32*3))
#X_test = np.reshape(X_test,(X_test.shape[0],32*32*3))
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
#Here we perform normalization
X_train = X_train/255
X_test = X_test/255
#Test data must be kept unseen from the model until final evaluation hence we need
#validation dataset.
_, X_valid, _, Y_valid = train_test_split(X_train, Y_train,
test_size=0.2, random_state= 8)
return X_train, Y_train, X_test, Y_test, X_valid,Y_valid

```

Figure 52: Preprocessing

2.7 Part 1 (CNN) : The preliminary CNN

The designed network is depicted below.

```
def NeuralNetworkcnn(l,opt):
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', input_shape=(32,
  32, 3), activation="relu"))
model.add(Conv2D(32, (3, 3), activation="relu"))
model.add(Flatten())
model.add(Dense(128,activation='relu',input_dim = 32*32*3))
model.add(Dense(128,activation='relu'))
model.add(Dense(10, activation="softmax"))
model.compile(loss=l, optimizer=opt, metrics=['accuracy'])
return model

```

Figure 53: CNN

The results are better than the **MLP** with about 10% more test accuracy and less error.

2.8 Part 2 (CNN) : Pooling and Batch Normalization

Firstly I shall explain the batch normalization and max pooling layers in convolutional neural networks.

2.8.1 Batch Normalization

Batch normalization is a normalization technique, this technique is utilized between the layers of a neural network instead of normalizing the raw data, batch normalization is done along mini-batches instead of the entire dataset.It is useful for speeding up training and using higher learning rates, making learning easier. It can be formulated as below.

$$z^N = \left(\frac{z - m_z}{s_z} \right), \quad s_z : \text{Standard Deviation}$$

2.8.2 Pooling

We know that the pooling operation involves sliding a two-dimensional filter over each channel of feature map and summarising the features lying within the region covered by the filter.

Pooling layers are used to reduce the dimensions of feature maps. So pooling reduces the number of parameters to learn and the amount of computation needed.

Also it is important to imply that pooling summarises the features present in a region of the feature map which is created by the CNN.

There many types of pooling such as max pooling, average pooling, global pooling etc...

In this project I have used max pooling.

The designed network is depicted below.

```
def NeuralNetworkcnnpoolbatchnorm(l,opt):
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', input_shape=(32,
 32, 3), activation="relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), activation="relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(BatchNormalization())
model.add(Dense(128,activation='relu',input_dim = 32*32*3))
model.add(Dense(128,activation='relu'))
model.add(Dense(10, activation="softmax"))
model.compile(loss=l, optimizer=opt, metrics=['accuracy'])
return model

```

Figure 54: CNN with max pooling and batch normalization

The code to train and results are as follows.

```
X_train, Y_train, X_test, Y_test = Loader()
X_train, Y_train, X_test, Y_test, X_valid,Y_valid =
 preprocessingcnn(X_train, Y_train, X_test, Y_test)
model64adamcnnbp = NeuralNetworkcnnpoolbatchnorm('
 categorical_crossentropy','adam')
history64adamcnnbp = model64adamcnnbp.fit(X_train,Y_train,
 epochs=15, batch_size=64, validation_data=(X_valid, Y_valid),
 verbose=1)

```

Figure 55: Code to train

Train Results

```
Epoch 1/15
782/782 [==============================] - 31s 38ms/step - loss: 1.2756 - accuracy: 0.5460 - val_loss: 0.9410 - val_accuracy: 0.6702
Epoch 2/15
782/782 [==============================] - 30s 39ms/step - loss: 0.9146 - accuracy: 0.6784 - val_loss: 0.8208 - val_accuracy: 0.7046
Epoch 3/15
782/782 [==============================] - 29s 37ms/step - loss: 0.7608 - accuracy: 0.7303 - val_loss: 0.6284 - val_accuracy: 0.7791
Epoch 4/15
782/782 [==============================] - 30s 38ms/step - loss: 0.6422 - accuracy: 0.7708 - val_loss: 0.5646 - val_accuracy: 0.7972
Epoch 5/15
782/782 [==============================] - 29s 37ms/step - loss: 0.5513 - accuracy: 0.8042 - val_loss: 0.4384 - val_accuracy: 0.8470
Epoch 6/15
782/782 [==============================] - 29s 37ms/step - loss: 0.4718 - accuracy: 0.8312 - val_loss: 0.3684 - val_accuracy: 0.8715
Epoch 7/15
782/782 [==============================] - 30s 38ms/step - loss: 0.3989 - accuracy: 0.8580 - val_loss: 0.4330 - val_accuracy: 0.8453
Epoch 8/15
782/782 [==============================] - 31s 40ms/step - loss: 0.3500 - accuracy: 0.8758 - val_loss: 0.2699 - val_accuracy: 0.9062
Epoch 9/15
782/782 [==============================] - 30s 38ms/step - loss: 0.3027 - accuracy: 0.8913 - val_loss: 0.3269 - val_accuracy: 0.8838
Epoch 10/15
782/782 [==============================] - 29s 37ms/step - loss: 0.2583 - accuracy: 0.9075 - val_loss: 0.2057 - val_accuracy: 0.9303
Epoch 11/15
782/782 [==============================] - 29s 37ms/step - loss: 0.2347 - accuracy: 0.9154 - val_loss: 0.3131 - val_accuracy: 0.8915
Epoch 12/15
782/782 [==============================] - 29s 37ms/step - loss: 0.2066 - accuracy: 0.9281 - val_loss: 0.1484 - val_accuracy: 0.9482
Epoch 13/15
782/782 [==============================] - 30s 38ms/step - loss: 0.1833 - accuracy: 0.9344 - val_loss: 0.1953 - val_accuracy: 0.9312
Epoch 14/15
782/782 [==============================] - 30s 38ms/step - loss: 0.1777 - accuracy: 0.9371 - val_loss: 0.1594 - val_accuracy: 0.9443
Epoch 15/15
782/782 [==============================] - 30s 38ms/step - loss: 0.1610 - accuracy: 0.9422 - val_loss: 0.1982 - val_accuracy: 0.9314

```

Figure 56: CNN with max pooling and batch normalization

Accuracy Plot

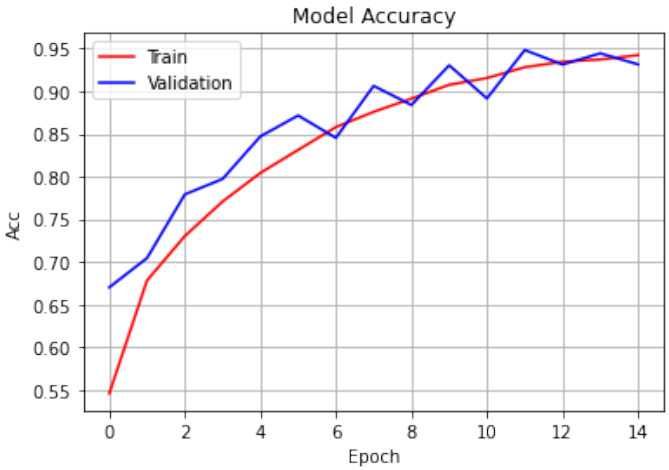

Figure 57: Accuracy Plot

Loss Plot

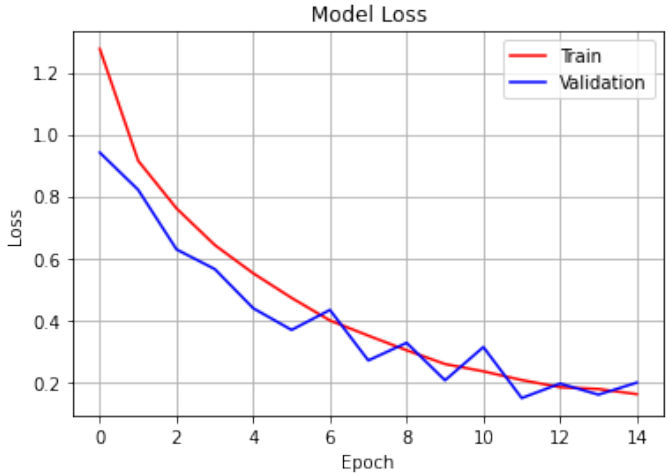

Figure 58: Loss Plot

Confusion matrix and accuracy and error

```
313/313 [==============================] - 2s 6ms/step
313/313 [==============================] - 2s 6ms/step - loss:
  1.6031 - accuracy: 0.6761
test loss = 1.603063
test accuracy = 0.676100
Accuracy: 0.676100
Precision: 0.700380
Recall: 0.676100
F1 score: 0.681632

```

Figure 59: Accuracy and error for CNN with max pooling and batch normalization

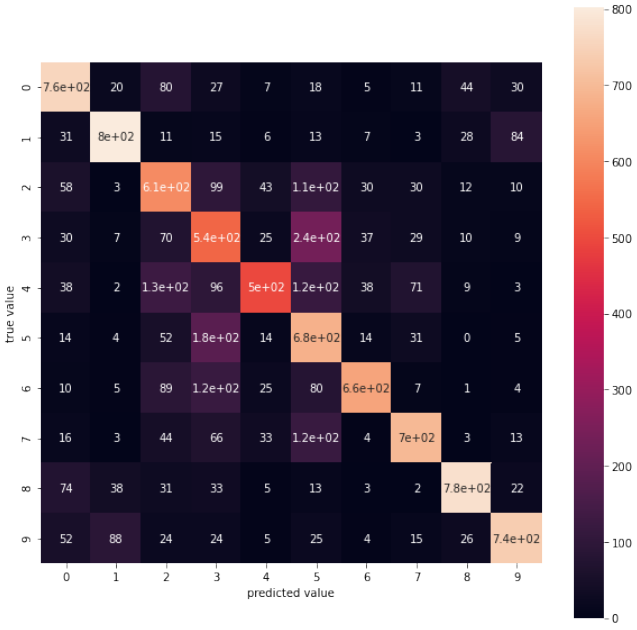

Figure 60: Confusion Matrix

As we can see both the loss and accuracy has improved compared to the MLP architecture.

2.9 Part 3 (CNN) : Adding dropout

Firstly I shall give some explanation about dropout and what it is.

2.9.1 Dropout

As instructed in the lectures, the term dropout means dropping out the nodes in a neural network, which can be the input or hidden layer nodes. All the forward and backward connections are removed temporarily so we generally create a new architecture.

Dropout is very useful in solving the overfitting problem, we know that in overfitting neurons might change in a way that fixes the mistakes of other neurons, dropout prevents neurons fixing the mistakes of other neurons, hence by randomly dropping a few nodes, it forces the layers to take more or less responsibility for the input by taking a probabilistic approach.

So to conclude our explanation we say that we use dropout to fix the overfitting problem.

The designed network is depicted below.

```
def NeuralNetworkcnndrop(l,opt):
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', input_shape=(32,
  32, 3), activation="relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(32, (3, 3), activation="relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(BatchNormalization())
model.add(Dropout(0.25))
model.add(Dense(128,activation='relu',input_dim = 32*32*3))
model.add(Dense(128,activation='relu'))
model.add(Dense(10, activation="softmax"))
model.compile(loss=l, optimizer=opt, metrics=['accuracy'])
return model

```

Figure 61: CNN with max pooling and batch normalization and dropout

The code to train and results are as follows.

```
X_train, Y_train, X_test, Y_test = Loader()
X_train, Y_train, X_test, Y_test, X_valid,Y_valid = preprocessingcnn(X_train, Y_train, X_test, Y_test)
model64adamcnnbpd = NeuralNetworkcnndrop('categorical_crossentropy','adam')
history64adamcnnbpd = model64adamcnnbpd.fit(X_train,Y_train, epochs=15, batch_size=64, validation_data=(
    X_valid, Y_valid),verbose=1)

```

Figure 62: Code to train

Train Results

```
Epoch 1/15
782/782 [==============================] - 37s 45ms/step - loss: 1.4404 - accuracy: 0.4823 - val_loss:
    1.3081 - val_accuracy: 0.5348
Epoch 2/15
782/782 [==============================] - 34s 44ms/step - loss: 1.1163 - accuracy: 0.6028 - val_loss:
    1.2351 - val_accuracy: 0.5748
Epoch 3/15
782/782 [==============================] - 34s 44ms/step - loss: 0.9943 - accuracy: 0.6451 - val_loss:
    0.7949 - val_accuracy: 0.7186
Epoch 4/15
782/782 [==============================] - 34s 43ms/step - loss: 0.9184 - accuracy: 0.6743 - val_loss:
    0.8778 - val_accuracy: 0.6910
Epoch 5/15
782/782 [==============================] - 35s 45ms/step - loss: 0.8606 - accuracy: 0.6938 - val_loss:
    0.7139 - val_accuracy: 0.7553
Epoch 6/15
782/782 [==============================] - 33s 42ms/step - loss: 0.8170 - accuracy: 0.7138 - val_loss:
    0.6629 - val_accuracy: 0.7566
Epoch 7/15
782/782 [==============================] - 34s 43ms/step - loss: 0.7748 - accuracy: 0.7252 - val_loss:
    0.6814 - val_accuracy: 0.7533
Epoch 8/15
782/782 [==============================] - 33s 42ms/step - loss: 0.7454 - accuracy: 0.7350 - val_loss:
    0.7308 - val_accuracy: 0.7425
Epoch 9/15
782/782 [==============================] - 33s 42ms/step - loss: 0.7265 - accuracy: 0.7419 - val_loss:
    0.6277 - val_accuracy: 0.7802
Epoch 10/15
782/782 [==============================] - 33s 42ms/step - loss: 0.7031 - accuracy: 0.7512 - val_loss:
    0.5166 - val_accuracy: 0.8168
Epoch 11/15
782/782 [==============================] - 33s 42ms/step - loss: 0.6795 - accuracy: 0.7598 - val_loss:
    0.4891 - val_accuracy: 0.8314
Epoch 12/15
782/782 [==============================] - 34s 44ms/step - loss: 0.6637 - accuracy: 0.7643 - val_loss:
    0.4912 - val_accuracy: 0.8278
Epoch 13/15
782/782 [==============================] - 34s 43ms/step - loss: 0.6470 - accuracy: 0.7701 - val_loss:
    0.4605 - val_accuracy: 0.8353
Epoch 14/15
782/782 [==============================] - 34s 44ms/step - loss: 0.6355 - accuracy: 0.7740 - val_loss:
    0.4919 - val_accuracy: 0.8290
Epoch 15/15
782/782 [==============================] - 34s 44ms/step - loss: 0.6223 - accuracy: 0.7798 - val_loss:
    0.4260 - val_accuracy: 0.8478

```

Figure 63: CNN with max pooling and batch normalization and dropout

Accuracy Plot

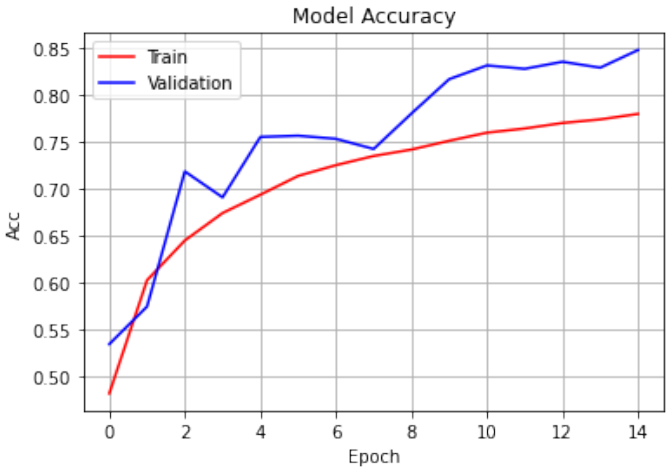

Figure 64: Accuracy Plot

Loss Plot

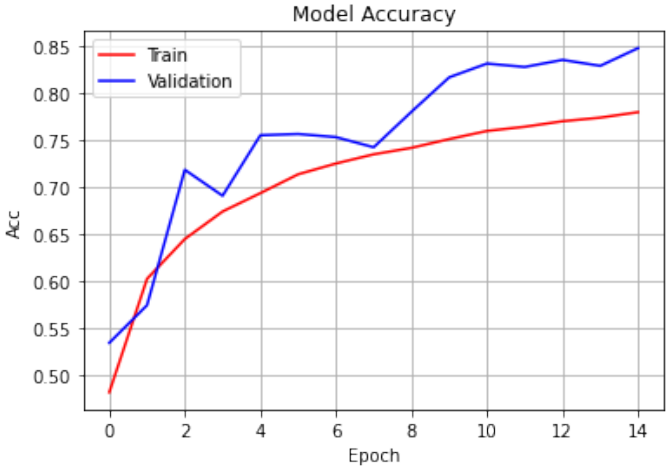

Figure 65: Loss Plot

Confusion matrix and accuracy and error

```
313/313 [==============================] - 2s 6ms/step
313/313 [==============================] - 2s 6ms/step - loss:
  0.7780 - accuracy: 0.7298
test loss = 0.778038
test accuracy = 0.729800
Accuracy: 0.729800
Precision: 0.747035
Recall: 0.729800
F1 score: 0.731516

```

Figure 66: Accuracy and error for CNN with max pooling and batch normalization

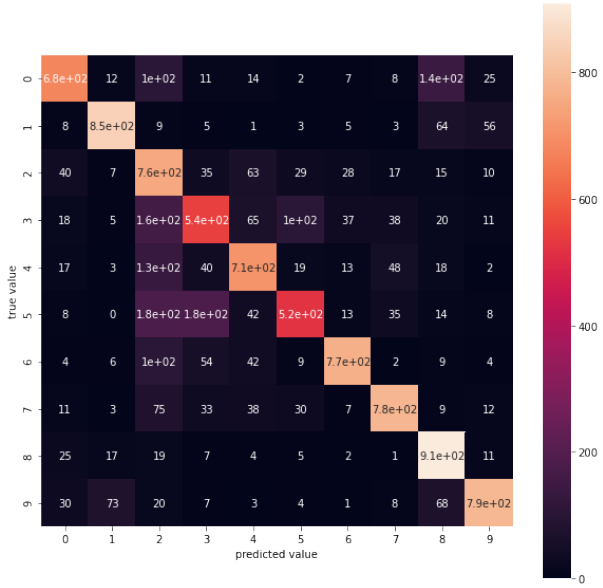

Figure 67: Confusion Matrix

We see that the overfitting problem is fixed and we get 72.98% accuracy.

2.10 Part 4 (CNN) : Early Stopping

Here I shall first explain early stopping.

2.10.1 Early Stopping

As depicted in every neural networks course, early stopping is a technique used for regularization for neural networks that stops training on a certain epoch when the parameter updates no longer begin to yield improves on our validation set.

Metrics such is minimum changes in the monitored quantity and number of epochs with no improvement and baseline values for the monitored quantity can all be metrics in early stopping a neural network.

A very important graph in early stopping is shown below.

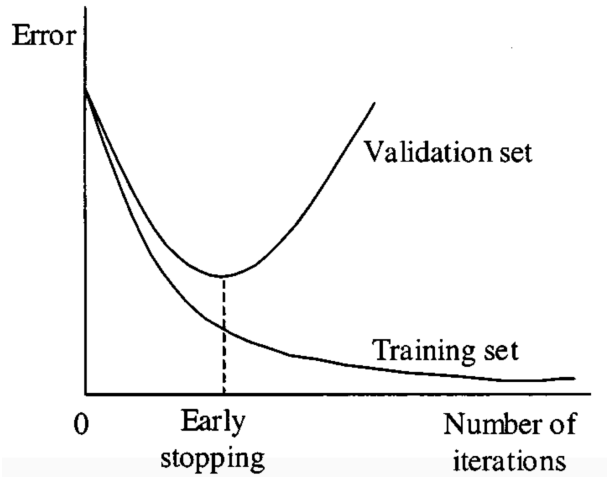

Figure 68: Early stopping

Now we shall implement early stopping as the final part of this section of the assignment.

The designed network is depicted below.

```
def NeuralNetworkcnnEarlyStop(l,opt):
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', input_shape=(32,
 32, 3), activation="relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(32, (3, 3), activation="relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(BatchNormalization())
model.add(Dropout(0.25))
model.add(Dense(128,activation='relu',input_dim = 32*32*3))
model.add(Dense(128,activation='relu'))
model.add(Dense(10, activation="softmax"))
model.compile(loss=l, optimizer=opt, metrics=['accuracy'])
return model

```

Figure 69: CNN Early Stopping

The code to train and results are as follows.

```
X_train, Y_train, X_test, Y_test = Loader()
X_train, Y_train, X_test, Y_test, X_valid,Y_valid = preprocessingcnn(X_train, Y_train, X_test, Y_test)
model64adamcnnerl = NeuralNetworkcnnEarlyStop('categorical_crossentropy','adam')
es = EarlyStopping(monitor='val_accuracy', mode='max',min_delta=1,verbose=1, patience=5)
history64adamcnnerl = model64adamcnnerl.fit(X_train,Y_train, epochs=15, batch_size=64, validation_data=(
    X_valid, Y_valid),callbacks=[es],verbose=1)

```

Figure 70: Code to train

Train Results

```
Epoch 1/15
782/782 [==============================] - 35s 43ms/step - loss: 1.4765 - accuracy: 0.4707 - val_loss:
    1.3721 - val_accuracy: 0.5186
Epoch 2/15
782/782 [==============================] - 33s 43ms/step - loss: 1.1906 - accuracy: 0.5742 - val_loss:
    1.3616 - val_accuracy: 0.5255
Epoch 3/15
782/782 [==============================] - 36s 45ms/step - loss: 1.0452 - accuracy: 0.6292 - val_loss:
    1.1625 - val_accuracy: 0.6078
Epoch 4/15
782/782 [==============================] - 37s 47ms/step - loss: 0.9614 - accuracy: 0.6553 - val_loss:
    0.8053 - val_accuracy: 0.7223
Epoch 5/15
782/782 [==============================] - 34s 43ms/step - loss: 0.8970 - accuracy: 0.6813 - val_loss:
    0.8304 - val_accuracy: 0.7072
Epoch 6/15
782/782 [==============================] - 34s 43ms/step - loss: 0.8488 - accuracy: 0.6985 - val_loss:
    0.6863 - val_accuracy: 0.7601
Epoch 6: early stopping

```

Figure 71: CNN Early Stopping

Accuracy Plot

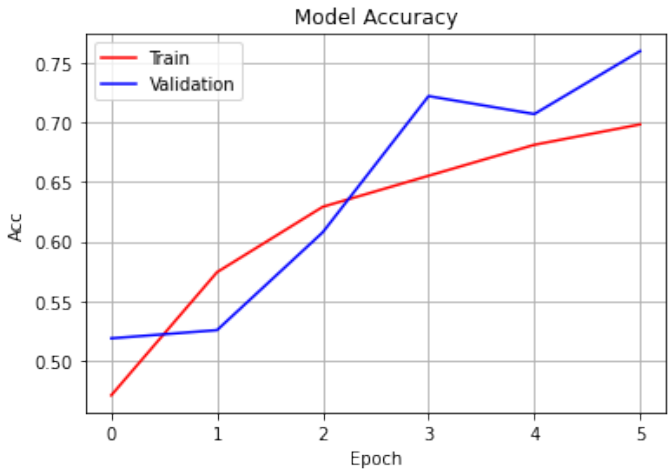

Figure 72: Accuracy Plot

Loss Plot

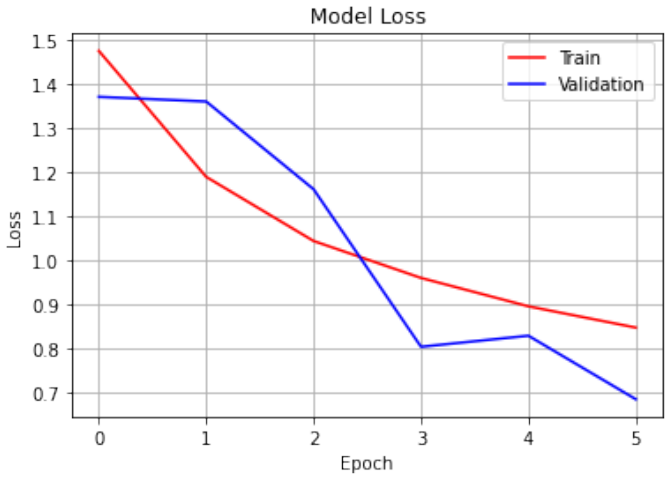

Figure 73: Loss Plot

Confusion matrix and accuracy and error

```
313/313 [==============================] - 2s 6ms/step
313/313 [==============================] - 2s 6ms/step - loss:
  0.8985 - accuracy: 0.6888
test loss = 0.898465
test accuracy = 0.688800
Accuracy: 0.688800
Precision: 0.711041
Recall: 0.688800
F1 score: 0.692636

```

Figure 74: Accuracy and error for CNN early stopping

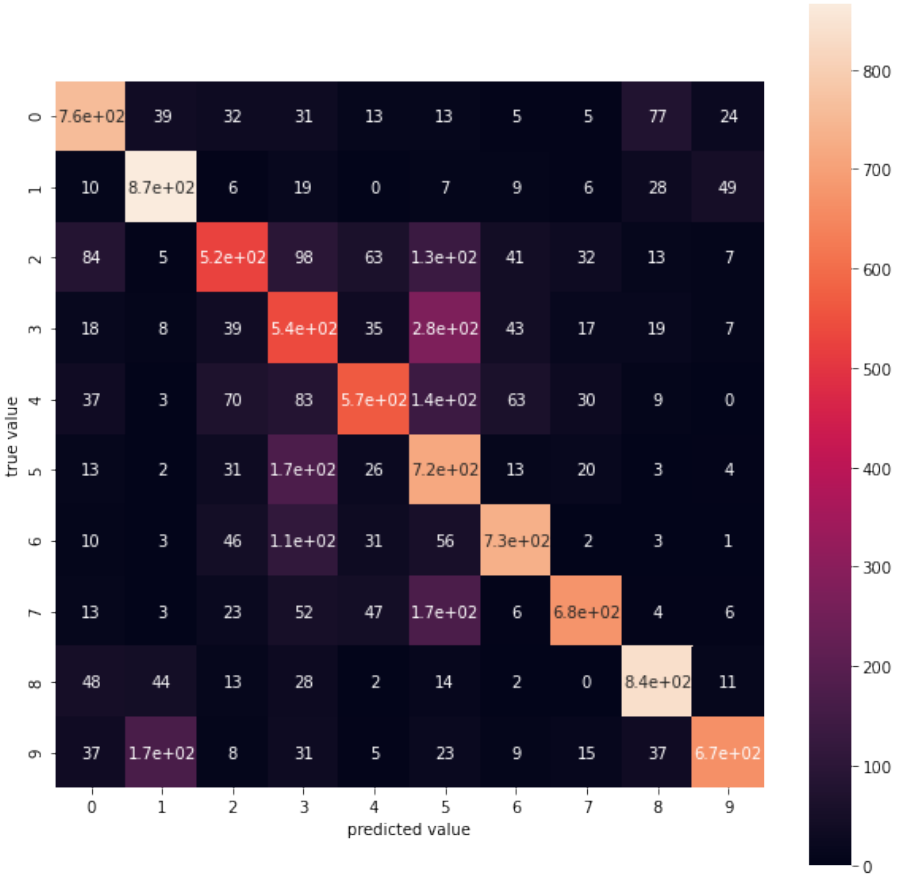

Figure 75: Confusion Matrix

3 Question 3 : Transfer Learning for EfficientNet network

3.1 Part A : Studying EfficientNet

Here we shall give some preliminary explanation about convolutional neural networks to build up our explanation for the EfficientNet architecture.

We know CNNs are normally developed at a fixed resource budget, and then scaled up to achieve better accuracy if more resources are available. In the EfficientNet paper a new scaling method that uniformly scales all dimensions of depth/width/resolution using a simple yet highly effective compound coefficient is proposed.

It is of common knowledge that scaling up CNNs is widely used to get better accuracy for example, ResNet can be scaled up from ResNet-18 to ResNet-200 by using more layers.

We have many ways to scale up CNNs, the most common way is to perform scaling by depth. In the given paper the main question is that is there a principled method to scale up ConvNets that can achieve better accuracy and efficiency? We can see some scaling methods in the following picture.

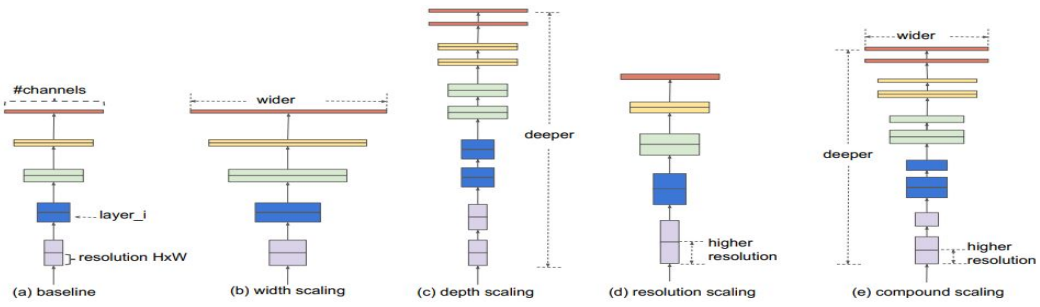

Figure 76: Different scaling models

3.1.1 Network Architecture

As explained in the paper due to the fact that model scaling does not change layer operators in baseline network, having a good baseline network is crucial, a new mobile-size baseline, called EfficientNet has been developed.

The baseline network has been developed by leveraging a multi-objective neural architecture search that optimizes both accuracy and FLOPS(Floating Point Operations Per Second). We optimize FLOPS rather than latency since we are not targeting any specific hardware device. This search produces an efficient network, which is named EfficientNet-B0.

We can see the EfficientNet-B0 baseline network table and structure below.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	28×28	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Figure 77: EfficientNet-B0 baseline network

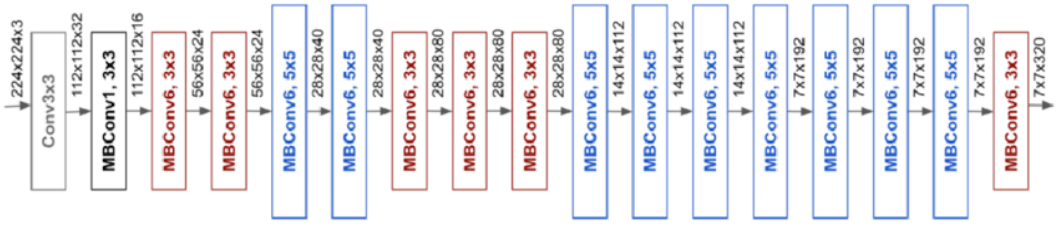

Figure 78: EfficientNet-B0 baseline network

3.1.2 Different EfficientNet Architectures

We must pay sincere attention that the **EfficientNetB0** model is the baseline, a full family of EfficientNet model have been developed based on **EfficientNetB0**, these models are named from **EfficientNetB1** to **EfficientNetB7**.

In the following photos we can see the difference between the architectures of these models.

EfficientNetB0

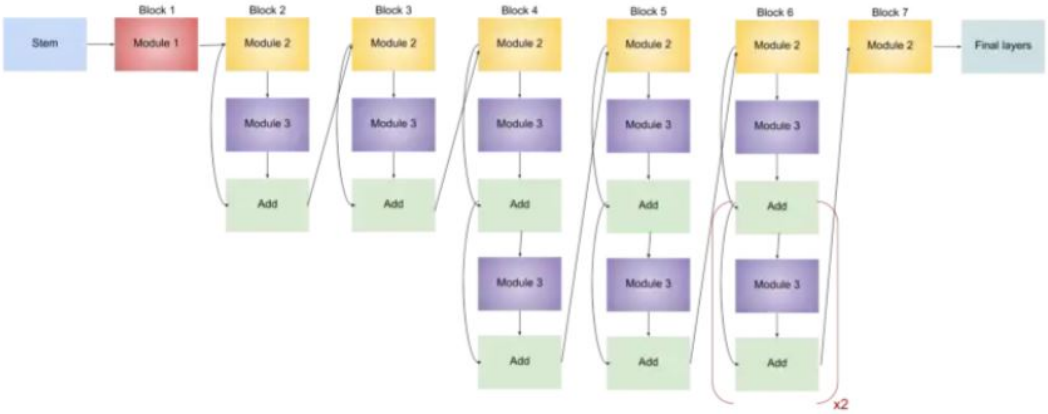

Figure 79: EfficientNet-B0 model

EfficientNetB1

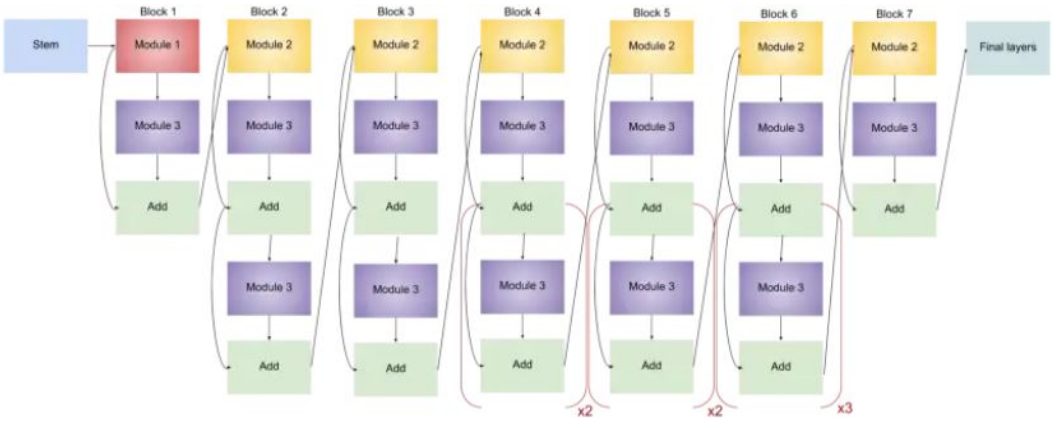

Figure 80: EfficientNet-B1 model

EfficientNetB2

Exactly the same as B1 but with more parameters.

EfficientNetB3

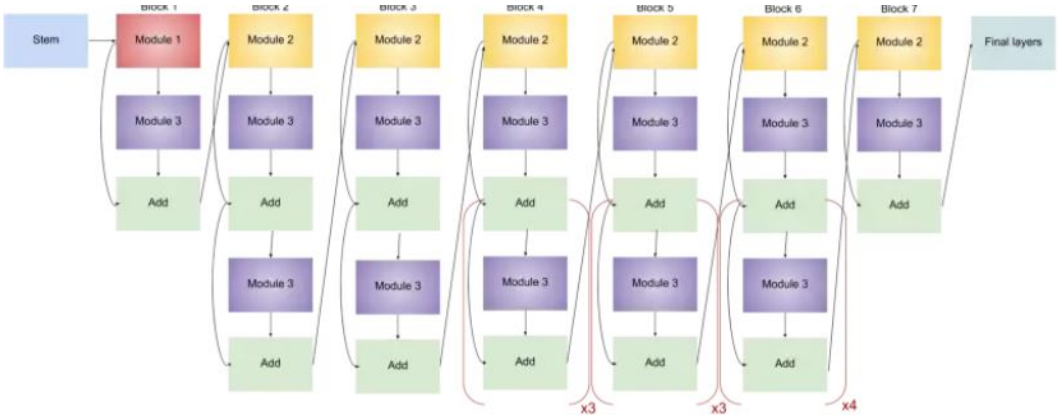

Figure 81: EfficientNet-B3 model

EfficientNetB4

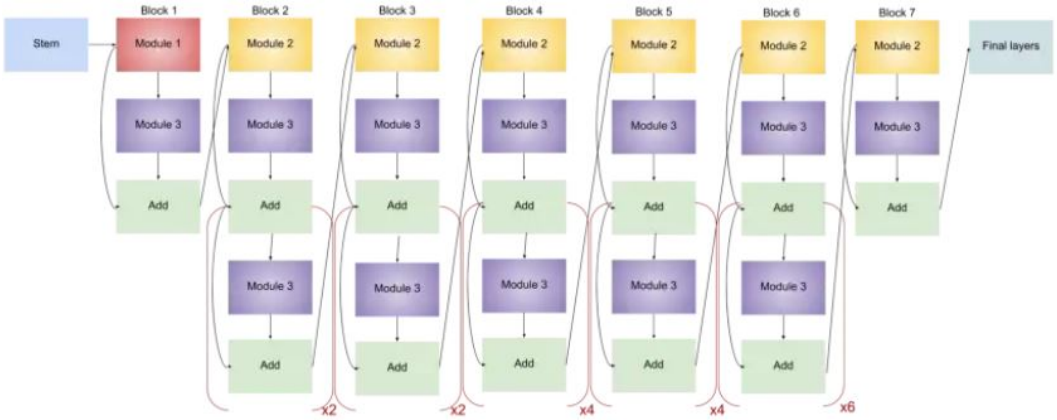

Figure 82: EfficientNet-B4 model

EfficientNetB5

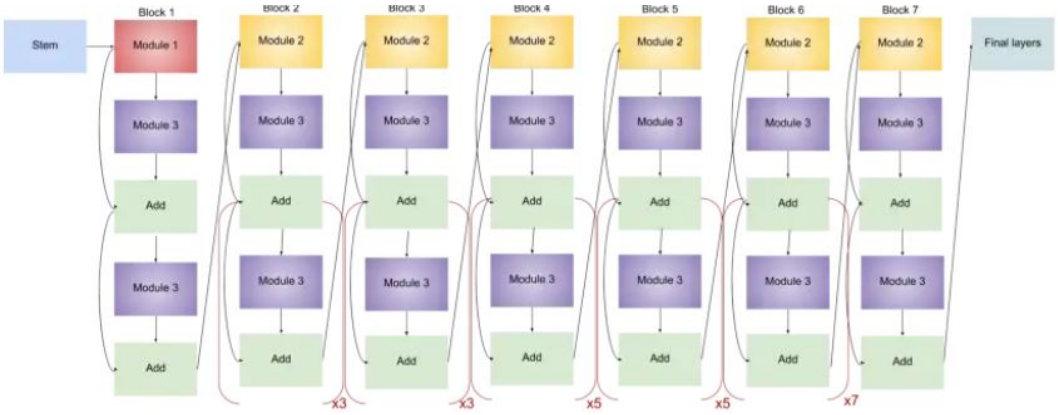

Figure 83: EfficientNet-B5 model

EfficientNetB6

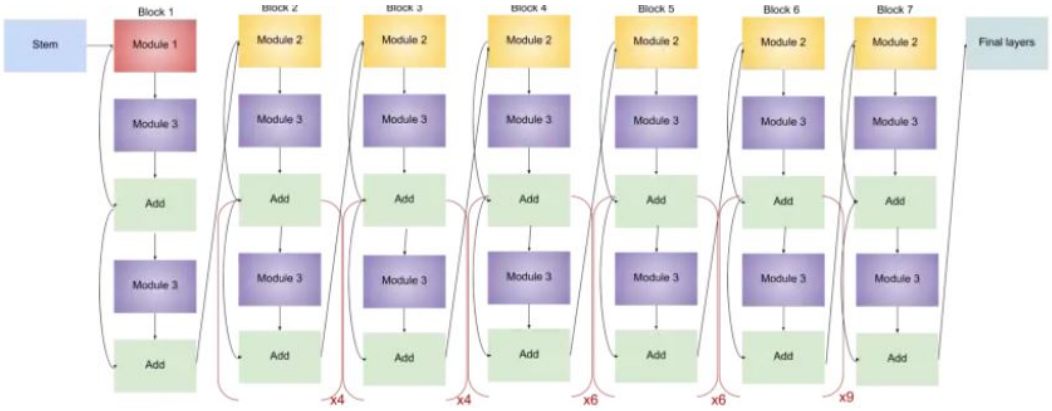

Figure 84: EfficientNet-B6 model

EfficientNetB7

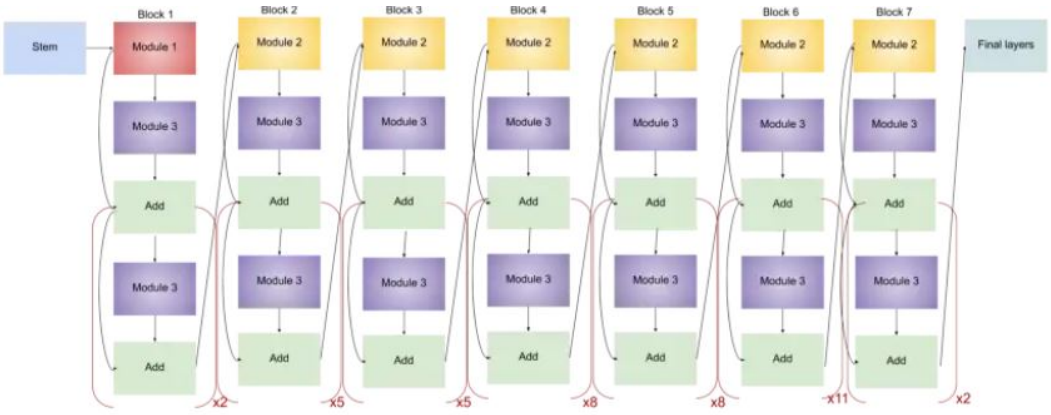

Figure 85: EfficientNet-B7 model

The main difference is in the number of channels which is depicted in the table below.

Stage	B1	B2	B3	B4	B5	B6	B7
1	32	32	40	48	48	56	64
2	16	16	24	24	24	32	32
3	24	24	32	32	40	40	48
4	40	48	48	56	64	72	80
5	80	88	96	112	128	144	160
6	112	120	136	160	176	200	224
7	192	208	232	272	304	344	384
8	320	352	384	448	512	576	640
9	1280	1408	1536	1792	2048	2304	2560

Figure 86: EfficientNet models

In the next table we can see the EfficientNet performance results on ImageNet.

Model	Top-1 Acc.	Top-5 Acc.	#Params	Ratio-to-EfficientNet	#FLOPS	Ratio-to-EfficientNet
EfficientNet-B0	**76.3%**	**93.2%**	**5.3M**	**1x**	**0.39B**	**1x**
ResNet-50 (He et al., 2016)	76.0%	93.0%	26M	4.9x	4.1B	11x
DenseNet-169 (Huang et al., 2017)	76.2%	93.2%	14M	2.6x	3.5B	8.9x
EfficientNet-B1	**78.8%**	**94.4%**	**7.8M**	**1x**	**0.70B**	**1x**
ResNet-152 (He et al., 2016)	77.8%	93.8%	60M	7.6x	11B	16x
DenseNet-264 (Huang et al., 2017)	77.9%	93.9%	34M	4.3x	6.0B	8.6x
Inception-v3 (Szegedy et al., 2016)	78.8%	94.4%	24M	3.0x	5.7B	8.1x
Xception (Chollet, 2017)	79.0%	94.5%	23M	3.0x	8.4B	12x
EfficientNet-B2	**79.8%**	**94.9%**	**9.2M**	**1x**	**1.0B**	**1x**
Inception-v4 (Szegedy et al., 2017)	80.0%	95.0%	48M	5.2x	13B	13x
Inception-resnet-v2 (Szegedy et al., 2017)	80.1%	95.1%	56M	6.1x	13B	13x
EfficientNet-B3	**81.1%**	**95.5%**	**12M**	**1x**	**1.8B**	**1x**
ResNeXt-101 (Xie et al., 2017)	80.9%	95.6%	84M	7.0x	32B	18x
PolyNet (Zhang et al., 2017)	81.3%	95.8%	92M	7.7x	35B	19x
EfficientNet-B4	**82.6%**	**96.3%**	**19M**	**1x**	**4.2B**	**1x**
SENet (Hu et al., 2018)	82.7%	96.2%	146M	7.7x	42B	10x
NASNet-A (Zoph et al., 2018)	82.7%	96.2%	89M	4.7x	24B	5.7x
AmoebaNet-A (Real et al., 2019)	82.8%	96.1%	87M	4.6x	23B	5.5x
PNASNet (Liu et al., 2018)	82.9%	96.2%	86M	4.5x	23B	6.0x
EfficientNet-B5	**83.3%**	**96.7%**	**30M**	**1x**	**9.9B**	**1x**
AmoebaNet-C (Cubuk et al., 2019)	83.5%	96.5%	155M	5.2x	41B	4.1x
EfficientNet-B6	**84.0%**	**96.9%**	**43M**	**1x**	**19B**	**1x**
EfficientNet-B7	**84.4%**	**97.1%**	**66M**	**1x**	**37B**	**1x**
GPipe (Huang et al., 2018)	84.3%	97.0%	557M	8.4x	-	-

Figure 87: EfficientNet performance results on ImageNet

3.1.3 Preliminary preprocessing on input image

We know that we must perform preprocessing for images before feeding them into a neural network, the image must match the input size of the network.

We know that image preprocessing includes adjustments to the size, orientation, and color and some other things. We perform preprocessing to increase the image's quality in order to have a better analysis. Preprocessing aids us to eliminate unneeded distortions and noises and improve qualities which are more important for recognition or any other application that we are looking for.

All in all, for preprocessing we must perform as such as resizing, random flipping, grayscaling and etc...

3.1.4 EfficientNet Pros compared to other models

We begin by taking a look at the accuracy per number of parameters plot as given in the paper.

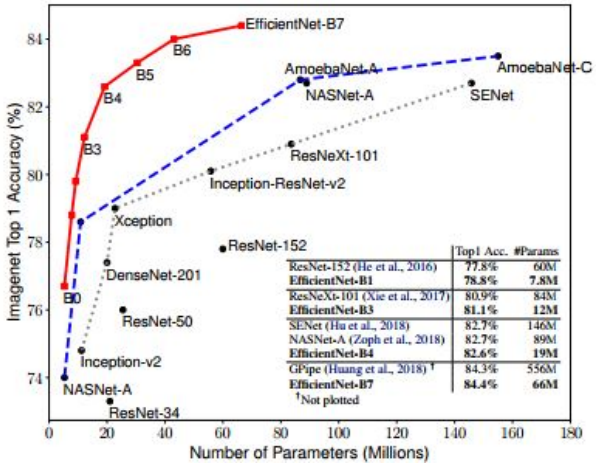

Figure 88: Imagenet accuracy for different models

As we can see **EfficinetNet** models yield to higher accuracies with lesser

parameters compared to different models such as **ResNet**, **SENet**, etc...

Another diagram can be seen below.

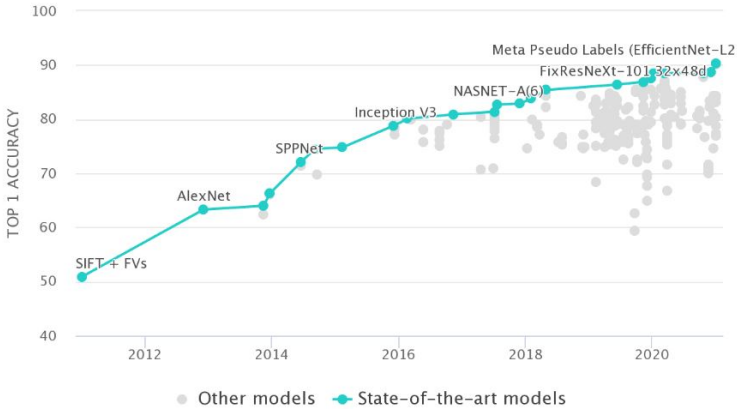

Figure 89: Different models comparison

As explained in the given paper, **EfficientNet** uses compound coefficients to scale models up in a very effective manner, instead of randomly scaling width, depth or resolution scaling is done cleverly with a set of fixed scaling coefficients.

For this model, transfer learning saves our time and computational energy as do other models, but we have better accuracies do to the clever scaling method explained.

3.2 Part B : Network Implementation with transfer learning

Here we implemented the model using keras then tested it for a photo taken by myself, the codes and results are as follows.

```
1    from tensorflow.keras.utils import load_img
2    from tensorflow.keras.utils import img_to_array
3    from tensorflow.keras.applications import EfficientNetB0
4    from keras.applications.efficientnet import preprocess_input
5    from keras.applications.efficientnet import
     decode_predictions
6    modelEffnet = EfficientNetB0(weights='imagenet')
7
```

Figure 90: Defining the EfficientNetB0 model

```
1  MyImage = load_img('Teapot.jpg',target_size = (224, 224))
2  MyImage = img_to_array(MyImage)
3  MyImage = MyImage.reshape((1, MyImage.shape[0], MyImage.shape
     [1], MyImage.shape[2]))
4  MyImage = preprocess_input(MyImage)
5  ypred = modelEffnet.predict(MyImage)
6  DecodedLabels = decode_predictions(ypred)
7
```

Figure 91: Performing predictions

After this we shall print the labels with the most probability as follows.

```
#Most Probability #1
DecodedLabel1 = DecodedLabels[0][0]
print(f"Pred with first most probability is : {DecodedLabel1[1]}\nAccuracy : {DecodedLabel1[2]*100}%")
#Most Probability #2
DecodedLabel2 = DecodedLabels[0][1]
print(f"Pred with second most probability is : {DecodedLabel2[1]} \nAccuracy : {DecodedLabel2[2]*100}%")
#Most Probability #3
DecodedLabel3 = DecodedLabels[0][2]
print(f"Pred with third most probability is : {DecodedLabel3[1]} \nAccuracy : {DecodedLabel3[2]*100}%")
The results:
Pred with first most probability is : teapot
Accuracy : 77.83220410346985%
Pred with second most probability is : coffeepot
Accuracy : 5.684347823262215%
Pred with third most probability is : water_jug
Accuracy : 2.3913515731692314%

```

Figure 92: Code and result for prediction printing

Figure 93: Input to the model

As we can see the predictions are close to the real thing.

3.3 Part C : Fixing a common problem

To fix the problem of input images not available in the original dataset, I have designed a function in which a threshold is observed and if the probability is less than the said threshold then the photo shall be labeled as **Others**.

The function code is depicted below.

```
def PhotoinModelChecker(DecodedLabels):
for i in range(len(DecodedLabels[0])):
if DecodedLabels[0][i][2]<0.5:
DecodedLabels[0][i] = list(DecodedLabels[0][i])
DecodedLabels[0][i][1] = 'others'
DecodedLabels[0][i][2] = '-'
DecodedLabels[0][i] = tuple(DecodedLabels[0][i])
return DecodedLabels

```

Figure 94: Code and result for prediction printing

To test this I gave the following photo to the network and obtained the following results.

Figure 95: Input to the model not in the dataset

The code and results are as follows.

```
#Most Probability #1
DecodedLabel1 = DecodedLabels[0][0]
print(f"Pred with first most probability is : {DecodedLabel1
  [1]}\nAccuracy : {DecodedLabel1[2]*100}%")
#Most Probability #2
DecodedLabel2 = DecodedLabels[0][1]
print(f"Pred with second most probability is : {DecodedLabel2
  [1]} \nAccuracy : {DecodedLabel2[2]*100}%")
#Most Probability #3
DecodedLabel3 = DecodedLabels[0][2]
print(f"Pred with third most probability is : {DecodedLabel3
  [1]} \nAccuracy : {DecodedLabel3[2]*100}%")
The results:
Pred with first most probability is : others
Accuracy : -%
Pred with second most probability is : others
Accuracy : -%
Pred with third most probability is : others
Accuracy : -%

```

Figure 96: Code and result for prediction printing

As we can see our implementation works, we shall now move on to the final part of this project.

3.4 Part 4 : Retraining the model on a new dataset

Here I used the **dogs-vs-cats** dataset which is available on Kaggle, then we perform transfer learning on this as depicted below.

```
! pip install -q kaggle
! mkdir ~/.kaggle
! cp kaggle.json ~/.kaggle/
! chmod 600 ~/.kaggle/kaggle.json
! kaggle datasets download -d shaunthesheep/microsoft-catsvsdogs-dataset

```

Figure 97: Downloading the dataset from Kaggle

Two of the photos in this dataset are faulty so I have removed them.

```
! unzip -q /content/microsoft-catsvsdogs-dataset.zip
! rm PetImages/Cat/666.jpg
! rm PetImages/Dog/11702.jpg
! rm PetImages/Cat/*.db
! rm PetImages/Dog/*.db

```

Figure 98: Deleting faulty images from dataset

Now we shall perform the train and validation split.

```
image_generator = ImageDataGenerator(validation_split=0.3, rescale=1/255.)

train = image_generator.flow_from_directory(batch_size=128,
directory='/content/PetImages',
shuffle=True,
subset="training",
class_mode="binary",
target_size=(224, 224),
classes = ["Cat", "Dog"])

valid = image_generator.flow_from_directory(batch_size=128,
directory='/content/PetImages',
shuffle=True,
subset="validation",
target_size=(224, 224),
class_mode="binary",
classes = ["Cat", "Dog"])

```

Figure 99: Train and valid split

Now we shall design the model.

```
base_model =  EfficientNetB0(input_shape=(224,224,3), weights=
  'imagenet', include_top=False)
model=Sequential()
model.add(base_model)
for layer in model.layers:
layer.trainable = False
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(1,kernel_initializer='he_uniform'))
model.add(keras.layers.Activation('sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy',
  metrics=['accuracy'])

model.summary()

```

Figure 100: The model

Now we shall train it.

```
137/137 [==============================] - 105s 715ms/step - loss: 1.1253 - accuracy: 0.5014 - val_loss:
    0.6992 - val_accuracy: 0.5000
Epoch 2/20
137/137 [==============================] - 87s 636ms/step - loss: 0.7618 - accuracy: 0.5078 - val_loss:
    0.6893 - val_accuracy: 0.5447
Epoch 3/20
137/137 [==============================] - 88s 643ms/step - loss: 0.7309 - accuracy: 0.5053 - val_loss:
    0.7074 - val_accuracy: 0.5011
Epoch 4/20
137/137 [==============================] - 87s 637ms/step - loss: 0.7692 - accuracy: 0.5066 - val_loss:
    0.6868 - val_accuracy: 0.5940
Epoch 5/20
137/137 [==============================] - 87s 635ms/step - loss: 0.7900 - accuracy: 0.5075 - val_loss:
    1.0678 - val_accuracy: 0.5000
Epoch 6/20
137/137 [==============================] - 101s 739ms/step - loss: 0.7862 - accuracy: 0.4958 - val_loss:
    0.7098 - val_accuracy: 0.5000
Epoch 7/20
137/137 [==============================] - 87s 635ms/step - loss: 0.7820 - accuracy: 0.5129 - val_loss:
    0.7739 - val_accuracy: 0.5007
Epoch 8/20
137/137 [==============================] - 87s 636ms/step - loss: 0.8326 - accuracy: 0.5057 - val_loss:
    0.6828 - val_accuracy: 0.5886
Epoch 9/20
137/137 [==============================] - 87s 636ms/step - loss: 0.7549 - accuracy: 0.5159 - val_loss:
    0.6834 - val_accuracy: 0.5750
Epoch 10/20
137/137 [==============================] - 86s 630ms/step - loss: 0.7436 - accuracy: 0.5183 - val_loss:
    0.6891 - val_accuracy: 0.5195
Epoch 11/20
137/137 [==============================] - 87s 634ms/step - loss: 0.8153 - accuracy: 0.5073 - val_loss:
    0.7324 - val_accuracy: 0.5033
Epoch 12/20
137/137 [==============================] - 87s 633ms/step - loss: 0.7811 - accuracy: 0.5144 - val_loss:
    0.6810 - val_accuracy: 0.5723
Epoch 13/20
137/137 [==============================] - 87s 635ms/step - loss: 0.7573 - accuracy: 0.5069 - val_loss:
    0.6925 - val_accuracy: 0.5000
Epoch 14/20
137/137 [==============================] - 86s 631ms/step - loss: 0.7745 - accuracy: 0.5181 - val_loss:
    0.7713 - val_accuracy: 0.5017
Epoch 15/20
137/137 [==============================] - 87s 635ms/step - loss: 0.7708 - accuracy: 0.5154 - val_loss:
    0.7525 - val_accuracy: 0.5039
Epoch 16/20
137/137 [==============================] - 86s 631ms/step - loss: 0.7574 - accuracy: 0.5124 - val_loss:
    0.8120 - val_accuracy: 0.5000
Epoch 17/20
137/137 [==============================] - 86s 631ms/step - loss: 0.7481 - accuracy: 0.5222 - val_loss:
    0.7015 - val_accuracy: 0.5000
Epoch 18/20
137/137 [==============================] - 88s 642ms/step - loss: 0.7952 - accuracy: 0.5137 - val_loss:
    0.6866 - val_accuracy: 0.5036
Epoch 19/20
137/137 [==============================] - 87s 637ms/step - loss: 0.7536 - accuracy: 0.5175 - val_loss:
    0.8200 - val_accuracy: 0.5000
Epoch 20/20
137/137 [==============================] - 86s 627ms/step - loss: 0.7613 - accuracy: 0.5187 - val_loss:
    0.8220 - val_accuracy: 0.5013

```

Figure 101: Training on dogs-vs-cats

Accuracy Plot

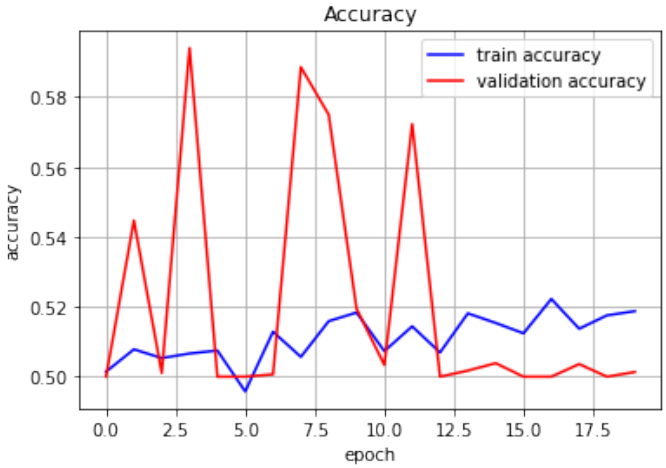

Figure 102: Accuracy Plot

Loss Plot

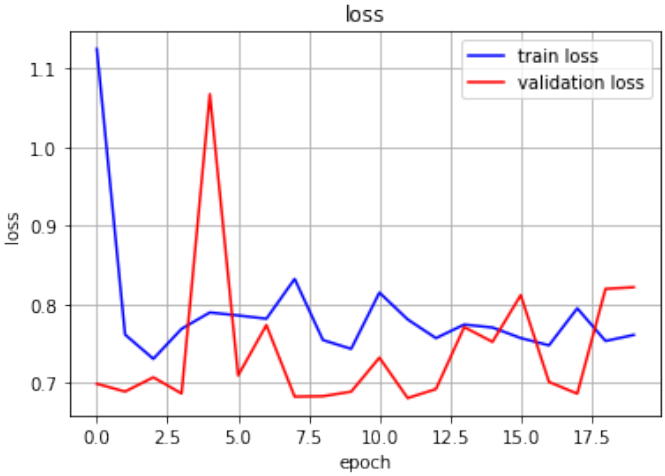

Figure 103: Loss Plot

References

[1] Reshad Hosseini, *Intelligent Systems Lecture Notes, Fall 01*

[2] Ali Olfat, *Stochastic Processes Lecture Notes, Fall 01*

[3] Activation functions advantages and disadvantages

[4] Activation functions in neural networks

[5] Batch Normalization

[6] Batch Normalization in Keras

[7] An introduction to pooling

[8] Dropout in Neural Networks

[9] Why is random forest an improvement of decision tree?

[10] Early stopping in Keras

[11] dogs-vs-cats dataset

[12] EfficientNet in Keras

[13] Image classification via fine-tuning with EfficientNet

[14] Image Data preprocessing

[15] Poisson Regression

[16] Poisson Regression

[17] Mean Squared Error vs Cross entropy loss function