ECE763 - Computer Vision Project -3

Team: Sai Srinivas Murthy Narayanam, Shreya Ramakanth, Sowjanya Srija Movva
snaray23@ncsu.edu sramaka@ncsu.edu smoovva3@ncsu.edu

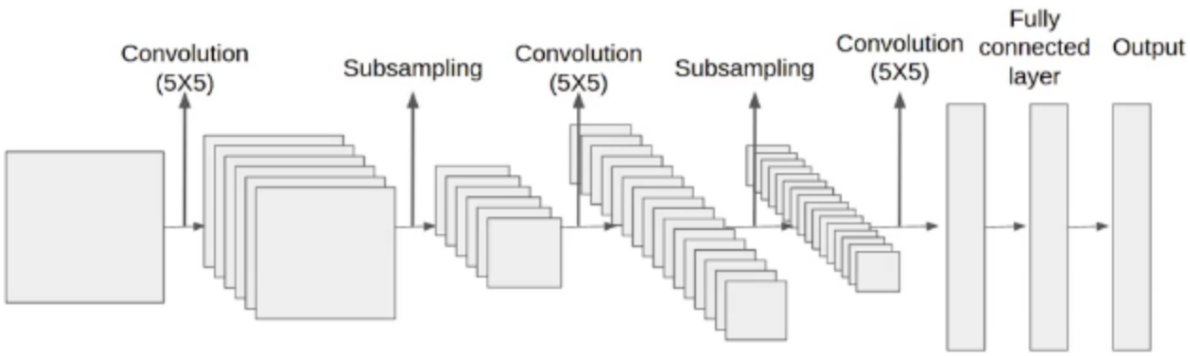
Note: Please check the end of the report for individual contribution for the project.

1.IMAGE DATA USED:

The Face-Nonface dataset was manually extracted from the FDDB faces dataset. The dataset consists of 2000 images for faces, and 2000 images for non-Face. For testing 800 images were used, of which 400 were of face and 400 were of Nonface. For training and testing, the images were cropped to dimensions of 20x20x1. The images used were gray scale images.

2.ARCHITECTURE OF NEURAL NETWORK USED:

LeNet-5 convolution neural networks have been used for babysitting. LeNet-5 has all the basic units of a convolutional neural network such as convolutional layer, pooling layer and full connection layer, laying a foundation for the future development of convolutional neural networks. lenet-5 consists of five layers. The image below shows the Architecture of LeNet-5.



Soure: Analytics Vidhya

3.DATA PRE-PROCESSING AND DATA AUGMENTATION:

A. Zero-centering the data distribution:

We use a matrix X which contains all the images after which the mean is found, and then subtract the mean image from all the images in X, doing this makes data distribution centered around 0.

```
xTrain -= np.mean(xTrain)
xVal -= np.mean(xVal)
```

B. DATA NORMALIZATION

For data normalization, matrix X is used to obtain the standard deviation of the image vectors in X, which is then divided by all the images in X to get normalized data. The images have been divided by 255 to give the value 1. The train and Val data are subtracted by the mean and divided by the standard deviation, to normalize the data with mean as 0 and standard deviation as 1.

```
def im_read_dir(path):
    list_images = []
    for imagepath in glob.glob(path+'/*.jpg'):
        image = cv2.imread(imagepath,0)
        image = image/255
        list_images.append(image.astype(np.float32))
        if len(list_images) == 2000:
            break
    return list_images
```

```
xTrain -= np.mean(xTrain)
xVal -= np.mean(xVal)
xTrain /= np.std(xTrain, axis = 0)
xVal /= np.std(xVal, axis = 0)
```

C. WEIGHT INITIALIZATION:

Initializing with Zeros

It can be seen from the above snippet that the loss is constant for all the 10 epochs. Using the zero initializer does not reduce the loss and it is constant throughout.

```
initializer = tf.keras.initializers.zeros()
class LeNet:
    def build(width, height, depth,category_ct, reg,lr):
        model = Sequential()
        inputShape = (height, width, depth)
        model.add(Conv2D(6, kernel_initializer=initializer,kernel_size=(5, 5), strides= (1,1), padding="same"
        model.add(Activation("relu"))
```

```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: 0.6932 - accuracy: 0.4893 - val_loss: 0.6932 - val_accur
acy: 0.4938
Epoch 2/10
85/85 [==============================] - 1s 6ms/step - loss: 0.6932 - accuracy: 0.4952 - val_loss: 0.6932 - val_accur
acy: 0.4938
Epoch 3/10
85/85 [==============================] - 1s 6ms/step - loss: 0.6932 - accuracy: 0.4923 - val_loss: 0.6932 - val_accur
acy: 0.4938
Epoch 4/10
85/85 [==============================] - 1s 6ms/step - loss: 0.6932 - accuracy: 0.4960 - val_loss: 0.6932 - val_accur
acy: 0.4938
Epoch 5/10
85/85 [==============================] - 1s 6ms/step - loss: 0.6932 - accuracy: 0.4864 - val_loss: 0.6932 - val_accur
acy: 0.4938
Epoch 6/10
85/85 [==============================] - 1s 6ms/step - loss: 0.6932 - accuracy: 0.4938 - val_loss: 0.6932 - val_accur
acy: 0.4938
Epoch 7/10
85/85 [==============================] - 1s 6ms/step - loss: 0.6932 - accuracy: 0.5011 - val_loss: 0.6932 - val_accur
acy: 0.4938
Epoch 8/10
85/85 [==============================] - 1s 6ms/step - loss: 0.6932 - accuracy: 0.5011 - val_loss: 0.6932 - val_accur
acy: 0.4938
Epoch 9/10
85/85 [==============================] - 1s 6ms/step - loss: 0.6932 - accuracy: 0.4989 - val_loss: 0.6932 - val_accur
acy: 0.4938
Epoch 10/10
85/85 [==============================] - 1s 6ms/step - loss: 0.6932 - accuracy: 0.4827 - val_loss: 0.6932 - val_accur
acy: 0.4938
```

Initializing with GlorotNormal

```
10
11 initializer = tf.keras.initializers.GlorotNormal(seed=None)
12
13 class LeNet:
14     def build(width, height, depth,category_ct, reg,lr):
15         model = Sequential()
16         inputShape = (height, width, depth)
17         model.add(Conv2D(6, kernel_initializer=initializer,kernel_size=(5, 5), strides= (1,1), padding="same",input
18         model.add(Activation("relu"))
```

```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: 0.6119 - accuracy: 0.6761 - val_loss: 0.5422 - val_accur
acy: 0.7771
Epoch 2/10
85/85 [==============================] - 1s 7ms/step - loss: 0.5409 - accuracy: 0.7735 - val_loss: 0.5011 - val_accur
acy: 0.7833
Epoch 3/10
85/85 [==============================] - 1s 7ms/step - loss: 0.4786 - accuracy: 0.7956 - val_loss: 0.4300 - val_accur
acy: 0.8208
Epoch 4/10
85/85 [==============================] - 1s 7ms/step - loss: 0.4309 - accuracy: 0.8140 - val_loss: 0.4217 - val_accur
acy: 0.8292
Epoch 5/10
85/85 [==============================] - 1s 7ms/step - loss: 0.3975 - accuracy: 0.8210 - val_loss: 0.3690 - val_accur
acy: 0.8667
Epoch 6/10
85/85 [==============================] - 1s 7ms/step - loss: 0.3661 - accuracy: 0.8393 - val_loss: 0.3422 - val_accur
acy: 0.8771
Epoch 7/10
85/85 [==============================] - 1s 6ms/step - loss: 0.3346 - accuracy: 0.8592 - val_loss: 0.3418 - val_accur
acy: 0.8500
Epoch 8/10
85/85 [==============================] - 1s 9ms/step - loss: 0.3159 - accuracy: 0.8687 - val_loss: 0.3025 - val_accur
acy: 0.8771
Epoch 9/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2951 - accuracy: 0.8746 - val_loss: 0.2789 - val_accur
acy: 0.8896
Epoch 10/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2786 - accuracy: 0.8838 - val_loss: 0.2787 - val_accur
acy: 0.8979
```

The Initial loss with glorot/xavier initialization is 0.62 and we can view it in the above figure. We can also observe that the loss is converging which is a good sign of a proper initialization.

Initializing with RandomNormal

```
initializer = tf.keras.initializers.RandomNormal(mean=0, stddev=1, seed=None)

class LeNet:
    def build(width, height, depth,category_ct, reg,lr):
        model = Sequential()
        inputShape = (height, width, depth)
        model.add(Conv2D(6, kernel_initializer=initializer,kernel_size=(5, 5), strides= (1,1), padding="same",input
        model.add(Activation("relu"))
```

```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: 0.6028 - accuracy: 0.6518 - val_loss: 0.5131 - val_accur
acy: 0.7917
Epoch 2/10
85/85 [==============================] - 1s 6ms/step - loss: 0.5060 - accuracy: 0.7772 - val_loss: 0.4422 - val_accur
acy: 0.8458
Epoch 3/10
85/85 [==============================] - 1s 6ms/step - loss: 0.4351 - accuracy: 0.8195 - val_loss: 0.3956 - val_accur
acy: 0.8583
Epoch 4/10
85/85 [==============================] - 1s 6ms/step - loss: 0.3828 - accuracy: 0.8371 - val_loss: 0.3472 - val_accur
acy: 0.8750
Epoch 5/10
85/85 [==============================] - 1s 6ms/step - loss: 0.3426 - accuracy: 0.8548 - val_loss: 0.3274 - val_accur
acy: 0.8750
Epoch 6/10
85/85 [==============================] - 1s 6ms/step - loss: 0.3164 - accuracy: 0.8684 - val_loss: 0.3172 - val_accur
acy: 0.8896
Epoch 7/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2923 - accuracy: 0.8842 - val_loss: 0.2921 - val_accur
acy: 0.8917
Epoch 8/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2773 - accuracy: 0.8886 - val_loss: 0.3068 - val_accur
acy: 0.8813
Epoch 9/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2601 - accuracy: 0.8963 - val_loss: 0.2617 - val_accur
acy: 0.9042
Epoch 10/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2531 - accuracy: 0.8919 - val_loss: 0.2378 - val_accur
acy: 0.9292
```

It can be seen from the above snippet that the loss considerably decreases over the 10 epochs. Using Random normal decreases, the accuracy but initial loss seems to be too high.

Initializing with Henormal

```
#initializer = tf.keras.initializers.RandomNormal(mean=0, stddev=1, seed=None)
#initializer = tf.keras.initializers.zeros()
#initializer = tf.keras.initializers.GlorotNormal(seed=None)

initializer = tf.keras.initializers.HeNormal(seed=None)

class LeNet:
    def build(width, height, depth,category_ct, reg,lr):
        model = Sequential()
        inputShape = (height, width, depth)
        model.add(Conv2D(6, kernel_initializer=initializer,kernel_size=(5, 5), strides= (1,1), padding="same",input
        model.add(Activation("relu"))
```

```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: 0.6323 - accuracy: 0.6603 - val_loss: 0.5848 - val_accur
acy: 0.7458
Epoch 2/10
85/85 [==============================] - 1s 7ms/step - loss: 0.5363 - accuracy: 0.7643 - val_loss: 0.6106 - val_accur
acy: 0.7396
Epoch 3/10
85/85 [==============================] - 1s 7ms/step - loss: 0.4994 - accuracy: 0.7838 - val_loss: 0.4860 - val_accur
acy: 0.8083
Epoch 4/10
85/85 [==============================] - 1s 6ms/step - loss: 0.4399 - accuracy: 0.8040 - val_loss: 0.3818 - val_accur
acy: 0.8438
Epoch 5/10
85/85 [==============================] - 1s 7ms/step - loss: 0.3895 - accuracy: 0.8257 - val_loss: 0.3480 - val_accur
acy: 0.8583
Epoch 6/10
85/85 [==============================] - 1s 7ms/step - loss: 0.3520 - accuracy: 0.8438 - val_loss: 0.3662 - val_accur
acy: 0.8417
Epoch 7/10
85/85 [==============================] - 1s 6ms/step - loss: 0.3287 - accuracy: 0.8596 - val_loss: 0.3009 - val_accur
acy: 0.8750
Epoch 8/10
85/85 [==============================] - 0s 6ms/step - loss: 0.3075 - accuracy: 0.8673 - val_loss: 0.2713 - val_accur
acy: 0.9021
Epoch 9/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2844 - accuracy: 0.8794 - val_loss: 0.2681 - val_accur
acy: 0.8833
Epoch 10/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2706 - accuracy: 0.8824 - val_loss: 0.2686 - val_accur
acy: 0.8833
```

It can be seen from the above snippet that the loss considerably decreases but not so much over the 10 epochs.

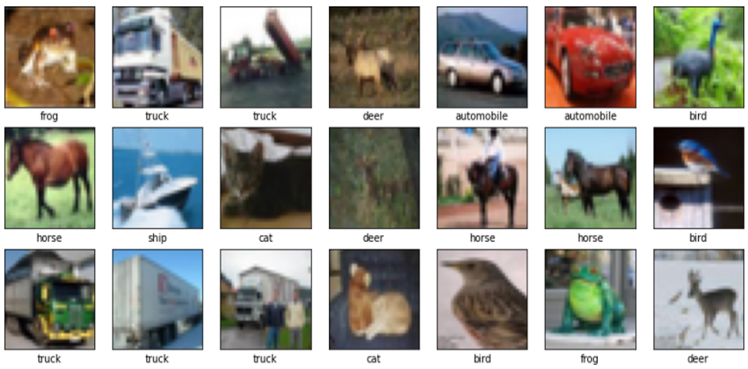
From the observations above, we felt that the GlorotNormal is a better weight initializer as it converges better with low initial loss and has better accuracy than the others.

4. DATA AUGMENTATION:

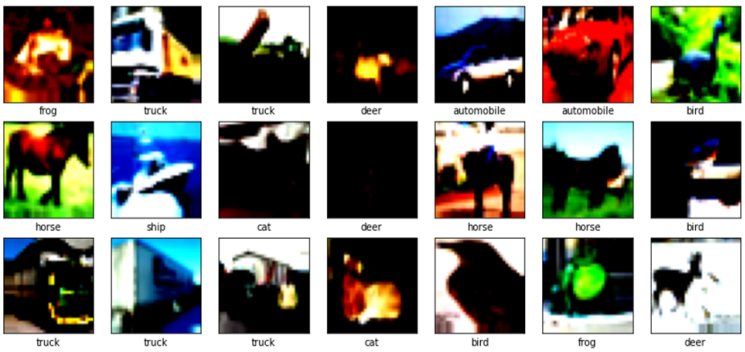
We have applied data augmentation to our model and also not have applied augmentation to our model, to see which suits the best.

CIFAR10 Dataset

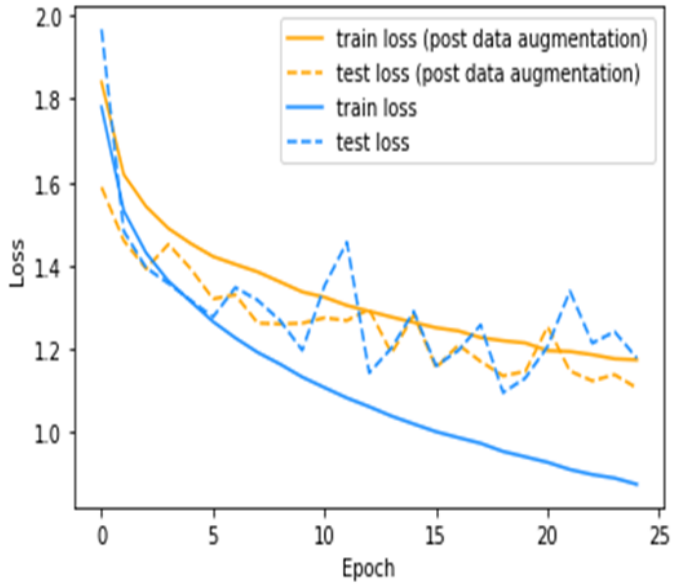
CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class. (It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.)



Visualization prior to data augmentation



Visualization post data augmentation



Loss vs Epoch plot

FDDB Dataset

```
# Data Augmentation
width_shift = 3/32
height_shift = 3/32
flip = True
data_aug = ImageDataGenerator(horizontal_flip=flip,
                              rotation_range=35, width_shift_range=0.1,height_shift_range=0.1,
                              zoom_range=0.3,shear_range=0.1, fill_mode="reflect")
```

For data augmentation we have used the following morphological changes: Flip, Rotate, Height Shift, Width shift, Zoom, Shear Range. We have performed both with and without augmentation.

As seen from the graphs for our model, it performs better without data augmentation. Hence, we choose not to use data augmentation further.

```
model = LeNet.build(20,20,1, category_ct,0,0.01)
trained_model = model.fit(xTrain, yTrain,validation_data=(xVal, yVal),epochs=ep, verbose=1,batch_size=32)

# With Data Augmentation
data_aug.fit(xTrain.reshape(2720, 20, 20, 1))
trained_model_aug = model.fit(data_aug.flow(xTrain.reshape(2720, 20, 20, 1), yTrain, batch_size=32),
                              validation_data=(xVal, yVal),epochs=ep, verbose=1)
```

With Data Augmentation

Output of model trained with data augmentation

```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: 0.4207 - accuracy: 0.7971 - val_loss: 0.2867 - val_accur
acy: 0.8979
Epoch 2/10
85/85 [==============================] - 1s 8ms/step - loss: 0.4050 - accuracy: 0.8158 - val_loss: 0.2825 - val_accur
acy: 0.8958
Epoch 3/10
85/85 [==============================] - 1s 8ms/step - loss: 0.3905 - accuracy: 0.8250 - val_loss: 0.2946 - val_accur
acy: 0.8729
Epoch 4/10
85/85 [==============================] - 1s 8ms/step - loss: 0.3944 - accuracy: 0.8169 - val_loss: 0.2757 - val_accur
acy: 0.8958
Epoch 5/10
85/85 [==============================] - 1s 8ms/step - loss: 0.3909 - accuracy: 0.8254 - val_loss: 0.3031 - val_accur
acy: 0.8625
Epoch 6/10
85/85 [==============================] - 1s 8ms/step - loss: 0.3891 - accuracy: 0.8283 - val_loss: 0.2845 - val_accur
acy: 0.8792
Epoch 7/10
85/85 [==============================] - 1s 8ms/step - loss: 0.3783 - accuracy: 0.8342 - val_loss: 0.2802 - val_accur
acy: 0.8875
Epoch 8/10
85/85 [==============================] - 1s 8ms/step - loss: 0.3723 - accuracy: 0.8338 - val_loss: 0.2771 - val_accur
acy: 0.8896
Epoch 9/10
85/85 [==============================] - 1s 8ms/step - loss: 0.3717 - accuracy: 0.8342 - val_loss: 0.2734 - val_accur
acy: 0.8854
Epoch 10/10
85/85 [==============================] - 1s 8ms/step - loss: 0.3744 - accuracy: 0.8360 - val_loss: 0.2713 - val_accur
acy: 0.8958
```

Without Augmentation

Model Trained without data augmentation

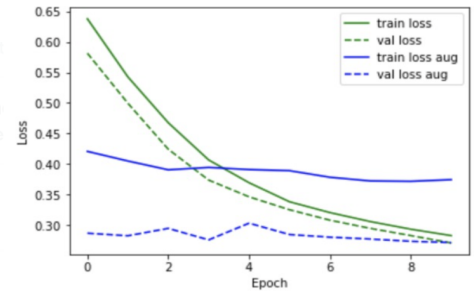
```
Epoch 1/10
85/85 [==============================] - 1s 5ms/step - loss: 0.6379 - accuracy: 0.6485 - val_loss: 0.5816 - val_accur
acy: 0.7521
Epoch 2/10
85/85 [==============================] - 0s 4ms/step - loss: 0.5429 - accuracy: 0.7643 - val_loss: 0.4998 - val_accur
acy: 0.7896
Epoch 3/10
85/85 [==============================] - 0s 4ms/step - loss: 0.4677 - accuracy: 0.7989 - val_loss: 0.4243 - val_accur
acy: 0.8229
Epoch 4/10
85/85 [==============================] - 0s 3ms/step - loss: 0.4067 - accuracy: 0.8202 - val_loss: 0.3737 - val_accur
acy: 0.8417
Epoch 5/10
85/85 [==============================] - 0s 4ms/step - loss: 0.3693 - accuracy: 0.8412 - val_loss: 0.3463 - val_accur
acy: 0.8521
Epoch 6/10
85/85 [==============================] - 0s 3ms/step - loss: 0.3382 - accuracy: 0.8588 - val_loss: 0.3251 - val_accur
acy: 0.8708
Epoch 7/10
85/85 [==============================] - 0s 3ms/step - loss: 0.3204 - accuracy: 0.8621 - val_loss: 0.3080 - val_accur
acy: 0.8833
Epoch 8/10
85/85 [==============================] - 0s 4ms/step - loss: 0.3057 - accuracy: 0.8728 - val_loss: 0.2946 - val_accur
acy: 0.8896
Epoch 9/10
85/85 [==============================] - 0s 4ms/step - loss: 0.2932 - accuracy: 0.8735 - val_loss: 0.2828 - val_accur
acy: 0.8938
Epoch 10/10
85/85 [==============================] - 0s 4ms/step - loss: 0.2827 - accuracy: 0.8783 - val_loss: 0.2707 - val_accur
acy: 0.8938
```

```python
fig = plt.figure()

plt.plot(trained_model.history['loss'],
         label='train loss',
         c='green', ls='-')
plt.plot(trained_model.history['val_loss'],
         label='val loss',
         c='green',ls='--')

plt.plot(trained_model_aug.history['loss'],
         label='train loss aug',
         c='blue', ls='-')
plt.plot(trained_model_aug.history['val_loss'],
         label='val loss aug',
         c='blue',ls='--')

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()
```



We can observe from the above graphs that, the CIFAR dataset does well with the data augmentation. The initial training loss is lower with augmentation and the test loss also is lower post data augmentation. This makes it clear that the data augmentation could be advantageous like the case with CIFAR and not so great like our face dataset. Even though face dataset has a lower initial loss with augmentation, the validation loss of the face dataset seems to be lower for the dataset without augmentation. Hence we would not proceed to augment the data in the further steps.

5. ADDING BATCH NORMALIZATION.

We have applied batch normalization to our model and also not have applied batch normalization to our model, to see which suits the best.

```
class LeNet:
    def build(width, height, depth,category_ct, reg,lr):
        model = Sequential()
        inputShape = (height, width, depth)
        model.add(Conv2D(6, kernel_initializer=initializer,kernel_size=(5, 5), strides= (1,1), padding="same",input
        model.add(Activation("relu"))
#         model.add(Activation("LeakyReLU"))
#         model.add(Activation("tanh"))
        model.add(AveragePooling2D(pool_size=(2, 2),strides= (2,2)))
        model.add(Conv2D(16,kernel_initializer=initializer,kernel_size= (5, 5), strides= (1,1), padding="valid",inp
        model.add(Activation("relu"))
#         model.add(Activation("LeakyReLU"))
#         model.add(Activation("tanh"))
        model.add(AveragePooling2D(pool_size=(2, 2),strides= (2,2)))
        model.add(Flatten())
        model.add(Dense(120, activation='relu',activity_regularizer=regularizers.l2(0)))
        model.add(Dense(84, activation='relu',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(120, activation='LeakyReLU',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(84, activation='LeakyReLU',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(120, activation='tanh',activity_regularizer=regularizers.l2(reg)))
#         model.add(Dense(84, activation='tanh',activity_regularizer=regularizers.l2(reg)))
        model.add(BatchNormalization())
        model.add(Dense(category_ct,activation = 'softmax'))
        model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.SGD(learni
#         model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.RMSprop(
#         model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.Adam(lea
        return model
```

```
Epoch 1/10
85/85 [==============================] - 1s 5ms/step - loss: 0.4009 - accuracy: 0.8213 - val_loss: 0.5448 - val_accur
acy: 0.8792
Epoch 2/10
85/85 [==============================] - 0s 4ms/step - loss: 0.3029 - accuracy: 0.8724 - val_loss: 0.4791 - val_accur
acy: 0.8646
Epoch 3/10
85/85 [==============================] - 0s 4ms/step - loss: 0.2640 - accuracy: 0.8956 - val_loss: 0.4195 - val_accur
acy: 0.8750
Epoch 4/10
85/85 [==============================] - 0s 4ms/step - loss: 0.2359 - accuracy: 0.9070 - val_loss: 0.3143 - val_accur
```

As we introduced batch normalization the initial loss has decreased from 0.63 to 0.400, which decreased significantly. So, batch normalization was very beneficial to our model so we have used it further.

6. ACTIVATION FUNCTION :

We have tested 3 different activation functions to see which suits best to our model

ReLU.

```python
# initializer = tf.keras.initializers.HeNormal(seed=None)
# initializer = tf.keras.initializers.Zeros()
class LeNet:
    def build(width, height, depth,category_ct, reg,lr):
        model = Sequential()
        inputShape = (height, width, depth)
        model.add(Conv2D(6, kernel_initializer=initializer,kernel_size=(5, 5), strides= (1,1), padding="same",input
        model.add(Activation("relu"))
#         model.add(Activation("LeakyReLU"))
#         model.add(Activation("tanh"))
        model.add(AveragePooling2D(pool_size=(2, 2),strides= (2,2)))
        model.add(Conv2D(16,kernel_initializer=initializer,kernel_size= (5, 5), strides= (1,1), padding="valid",inp
        model.add(Activation("relu"))
#         model.add(Activation("LeakyReLU"))
#         model.add(Activation("tanh"))
        model.add(AveragePooling2D(pool_size=(2, 2),strides= (2,2)))
        model.add(Flatten())
        model.add(Dense(120, activation='relu',activity_regularizer=regularizers.l2(0)))
        model.add(Dense(84, activation='relu',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(120, activation='LeakyReLU',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(84, activation='LeakyReLU',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(120, activation='tanh',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(84, activation='tanh',activity_regularizer=regularizers.l2(0)))
        model.add(BatchNormalization())
        model.add(Dense(category_ct,activation = 'softmax'))
        model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.SGD(learni
#         model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.RMSprop(
#         model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.Adam(lea
        return model


ep = 10
```

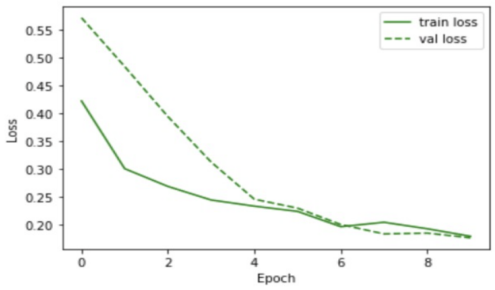
```
Epoch 1/10
85/85 [==============================] - 1s 5ms/step - loss: 0.4222 - accuracy: 0.8026 - val_loss: 0.5710 - val_accur
acy: 0.8542
Epoch 2/10
85/85 [==============================] - 0s 4ms/step - loss: 0.3010 - accuracy: 0.8721 - val_loss: 0.4837 - val_accur
acy: 0.8979
Epoch 3/10
85/85 [==============================] - 0s 4ms/step - loss: 0.2692 - accuracy: 0.8886 - val_loss: 0.3941 - val_accur
acy: 0.9083
Epoch 4/10
85/85 [==============================] - 0s 3ms/step - loss: 0.2449 - accuracy: 0.9037 - val_loss: 0.3128 - val_accur
acy: 0.9187
Epoch 5/10
85/85 [==============================] - 0s 4ms/step - loss: 0.2339 - accuracy: 0.9037 - val_loss: 0.2463 - val_accur
acy: 0.9187
Epoch 6/10
85/85 [==============================] - 0s 4ms/step - loss: 0.2245 - accuracy: 0.9132 - val_loss: 0.2303 - val_accur
acy: 0.9167
Epoch 7/10
85/85 [==============================] - 0s 4ms/step - loss: 0.1972 - accuracy: 0.9272 - val_loss: 0.2010 - val_accur
acy: 0.9333
Epoch 8/10
85/85 [==============================] - 0s 4ms/step - loss: 0.2050 - accuracy: 0.9158 - val_loss: 0.1842 - val_accur
acy: 0.9333
Epoch 9/10
85/85 [==============================] - 0s 4ms/step - loss: 0.1935 - accuracy: 0.9283 - val_loss: 0.1858 - val_accur
acy: 0.9375
Epoch 10/10
85/85 [==============================] - 0s 4ms/step - loss: 0.1800 - accuracy: 0.9272 - val_loss: 0.1772 - val_accur
acy: 0.9396
```

```
fig = plt.figure()

plt.plot(trained_model.history['loss'],
         label='train loss',
         c='green', ls='-')
plt.plot(trained_model.history['val_loss'],
         label='val loss',
         c='green',ls='--')

# plt.plot(trained_model_aug.history['loss'],
#          label='train loss aug',
#          c='blue', ls='-')
# plt.plot(trained_model_aug.history['val_loss'],
#          label='val loss aug',
#          c='blue',ls='--')

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()
```



Leaky ReLU

```
# initializer = tf.keras.initializers.RandomNormal(mean=0, stddev=1, seed=None)
initializer = tf.keras.initializers.GlorotNormal(seed=None)
# initializer = tf.keras.initializers.HeNormal(seed=None)
# initializer = tf.keras.initializers.Zeros()
class LeNet:
    def build(width, height, depth,category_ct, reg,lr):
        model = Sequential()
        inputShape = (height, width, depth)
        model.add(Conv2D(6, kernel_initializer=initializer,kernel_size=(5, 5), strides= (1,1), padding="same",input
#         model.add(Activation("relu"))
        model.add(Activation("LeakyReLU"))
#         model.add(Activation("tanh"))
        model.add(AveragePooling2D(pool_size=(2, 2),strides= (2,2)))
        model.add(Conv2D(16,kernel_initializer=initializer,kernel_size= (5, 5), strides= (1,1), padding="valid",inp
#         model.add(Activation("relu"))
        model.add(Activation("LeakyReLU"))
#         model.add(Activation("tanh"))
        model.add(AveragePooling2D(pool_size=(2, 2),strides= (2,2)))
        model.add(Flatten())
#         model.add(Dense(120, activation='relu',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(84, activation='relu',activity_regularizer=regularizers.l2(0)))
        model.add(Dense(120, activation='LeakyReLU',activity_regularizer=regularizers.l2(0)))
        model.add(Dense(84, activation='LeakyReLU',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(120, activation='tanh',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(84, activation='tanh',activity_regularizer=regularizers.l2(0)))
        model.add(BatchNormalization())
        model.add(Dense(category_ct,activation = 'softmax'))
        model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.SGD(learni
#         model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.RMSprop(
#         model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.Adam(lea
        return model
```

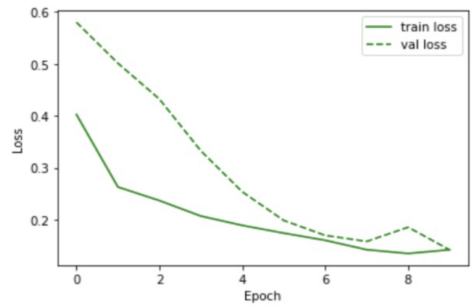
```
Epoch 1/10
85/85 [==============================] - 1s 5ms/step - loss: 0.4025 - accuracy: 0.8121 - val_loss: 0.5802 - val_accur
acy: 0.8896
Epoch 2/10
85/85 [==============================] - 0s 4ms/step - loss: 0.2628 - accuracy: 0.8882 - val_loss: 0.5014 - val_accur
acy: 0.9333
Epoch 3/10
85/85 [==============================] - 0s 4ms/step - loss: 0.2367 - accuracy: 0.9018 - val_loss: 0.4323 - val_accur
acy: 0.9229
Epoch 4/10
85/85 [==============================] - 0s 4ms/step - loss: 0.2069 - accuracy: 0.9202 - val_loss: 0.3325 - val_accur
acy: 0.9271
Epoch 5/10
85/85 [==============================] - 0s 5ms/step - loss: 0.1888 - accuracy: 0.9305 - val_loss: 0.2535 - val_accur
acy: 0.9396
Epoch 6/10
85/85 [==============================] - 0s 4ms/step - loss: 0.1740 - accuracy: 0.9386 - val_loss: 0.1984 - val_accur
acy: 0.9333
Epoch 7/10
85/85 [==============================] - 0s 4ms/step - loss: 0.1601 - accuracy: 0.9397 - val_loss: 0.1695 - val_accur
acy: 0.9417
Epoch 8/10
85/85 [==============================] - 0s 4ms/step - loss: 0.1420 - accuracy: 0.9529 - val_loss: 0.1580 - val_accur
acy: 0.9458
Epoch 9/10
85/85 [==============================] - 0s 4ms/step - loss: 0.1347 - accuracy: 0.9540 - val_loss: 0.1854 - val_accur
```

```python
fig = plt.figure()

plt.plot(trained_model.history['loss'],
         label='train loss',
         c='green', ls='-')
plt.plot(trained_model.history['val_loss'],
         label='val loss',
         c='green',ls='--')

# plt.plot(trained_model_aug.history['loss'],
#          label='train loss aug',
#          c='blue', ls='-')
# plt.plot(trained_model_aug.history['val_loss'],
#          label='val loss aug',
#          c='blue',ls='--')

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()
```



Tanh

```
# initializer = tf.keras.initializers.RandomNormal(mean=0, stddev=1, seed=None)
initializer = tf.keras.initializers.GlorotNormal(seed=None)
# initializer = tf.keras.initializers.HeNormal(seed=None)
# initializer = tf.keras.initializers.Zeros()
class LeNet:
    def build(width, height, depth,category_ct, reg,lr):
        model = Sequential()
        inputShape = (height, width, depth)
        model.add(Conv2D(6, kernel_initializer=initializer,kernel_size=(5, 5), strides= (1,1), padding="same",input
#         model.add(Activation("relu"))
#         model.add(Activation("LeakyReLU"))
        model.add(Activation("tanh"))
        model.add(AveragePooling2D(pool_size=(2, 2),strides= (2,2)))
        model.add(Conv2D(16,kernel_initializer=initializer,kernel_size= (5, 5), strides= (1,1), padding="valid",inp
#         model.add(Activation("relu"))
#         model.add(Activation("LeakyReLU"))
        model.add(Activation("tanh"))
        model.add(AveragePooling2D(pool_size=(2, 2),strides= (2,2)))
        model.add(Flatten())
#         model.add(Dense(120, activation='relu',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(84, activation='relu',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(120, activation='LeakyReLU',activity_regularizer=regularizers.l2(0)))
#         model.add(Dense(84, activation='LeakyReLU',activity_regularizer=regularizers.l2(0)))
        model.add(Dense(120, activation='tanh',activity_regularizer=regularizers.l2(reg)))
        model.add(Dense(84, activation='tanh',activity_regularizer=regularizers.l2(reg)))
        model.add(BatchNormalization())
        model.add(Dense(category_ct,activation = 'softmax'))
        model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.SGD(learni
#         model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.RMSprop(
#         model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.Adam(lea
        return model
```

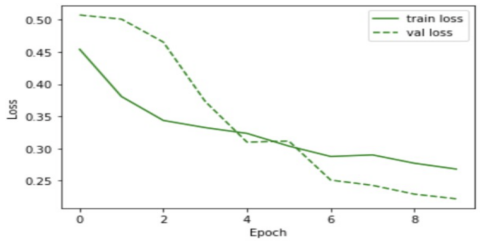
```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: 0.4541 - accuracy: 0.7923 - val_loss: 0.5074 - val_accur
acy: 0.7479
Epoch 2/10
85/85 [==============================] - 0s 5ms/step - loss: 0.3806 - accuracy: 0.8445 - val_loss: 0.5009 - val_accur
acy: 0.6812
Epoch 3/10
85/85 [==============================] - 0s 5ms/step - loss: 0.3434 - accuracy: 0.8581 - val_loss: 0.4652 - val_accur
acy: 0.7354
Epoch 4/10
85/85 [==============================] - 0s 5ms/step - loss: 0.3323 - accuracy: 0.8588 - val_loss: 0.3731 - val_accur
acy: 0.8083
Epoch 5/10
85/85 [==============================] - 0s 5ms/step - loss: 0.3233 - accuracy: 0.8643 - val_loss: 0.3097 - val_accur
acy: 0.8521
Epoch 6/10
85/85 [==============================] - 0s 5ms/step - loss: 0.3036 - accuracy: 0.8757 - val_loss: 0.3114 - val_accur
acy: 0.8375
Epoch 7/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2874 - accuracy: 0.8853 - val_loss: 0.2507 - val_accur
acy: 0.9042
Epoch 8/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2898 - accuracy: 0.8857 - val_loss: 0.2425 - val_accur
acy: 0.8979
Epoch 9/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2769 - accuracy: 0.8908 - val_loss: 0.2288 - val_accur
acy: 0.9187
Epoch 10/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2678 - accuracy: 0.8971 - val_loss: 0.2216 - val_accur
acy: 0.9146
```

```
fig = plt.figure()

plt.plot(trained_model.history['loss'],
         label='train loss',
         c='green', ls='-')
plt.plot(trained_model.history['val_loss'],
         label='val loss',
         c='green',ls='--')

# plt.plot(trained_model_aug.history['loss'],
#          label='train loss aug',
#          c='blue', ls='-')
# plt.plot(trained_model_aug.history['val_loss'],
#          label='val loss aug',
#          c='blue',ls='--')

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()
```



We have implemented 3 activation functions. They are ReLu, leakyReLu and Tanh. Among the three we choose LeakyreLu because as we can see the loss has become the lowest with leakyrelu when compared with the other two. The next best is Relu and Tanh doesn't not decrease the loss much.

7. OPTIMIZERS

Up until now, SGD has been used as the optimizer to update weights. So let us now try to use different optimizers and see if there is any improvement in the performance.

RMS Prop

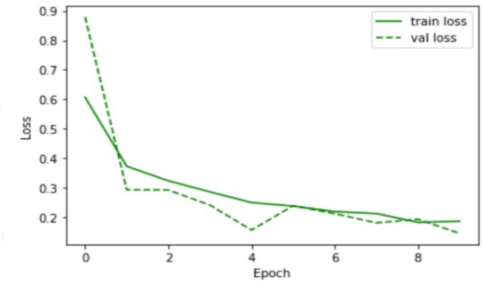
```
Epoch 1/10
85/85 [==============================] - 2s 8ms/step - loss: 0.6066 - accuracy: 0.7533 - val_loss: 0.8790 - val_accur
acy: 0.7917
Epoch 2/10
85/85 [==============================] - 0s 5ms/step - loss: 0.3728 - accuracy: 0.8449 - val_loss: 0.2933 - val_accur
acy: 0.8979
Epoch 3/10
85/85 [==============================] - 0s 5ms/step - loss: 0.3236 - accuracy: 0.8728 - val_loss: 0.2922 - val_accur
acy: 0.9083
Epoch 4/10
85/85 [==============================] - 0s 6ms/step - loss: 0.2861 - accuracy: 0.8912 - val_loss: 0.2411 - val_accur
acy: 0.9125
Epoch 5/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2498 - accuracy: 0.9004 - val_loss: 0.1565 - val_accur
acy: 0.9396
Epoch 6/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2378 - accuracy: 0.9062 - val_loss: 0.2390 - val_accur
acy: 0.8979
Epoch 7/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2193 - accuracy: 0.9169 - val_loss: 0.2121 - val_accur
acy: 0.9187
Epoch 8/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2125 - accuracy: 0.9184 - val_loss: 0.1811 - val_accur
acy: 0.9354
Epoch 9/10
85/85 [==============================] - 0s 5ms/step - loss: 0.1827 - accuracy: 0.9298 - val_loss: 0.1933 - val_accur
acy: 0.9312
Epoch 10/10
85/85 [==============================] - 0s 5ms/step - loss: 0.1866 - accuracy: 0.9294 - val_loss: 0.1452 - val_accur
acy: 0.9375
```

```python
fig = plt.figure()

plt.plot(trained_model.history['loss'],
         label='train loss',
         c='green', ls='-')
plt.plot(trained_model.history['val_loss'],
         label='val loss',
         c='green',ls='--')

# plt.plot(trained_model_aug.history['loss'],
#          label='train loss aug',
#          c='blue', ls='-')
# plt.plot(trained_model_aug.history['val_loss'],
#          label='val loss aug',
#          c='blue',ls='--')

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()
```



Adam

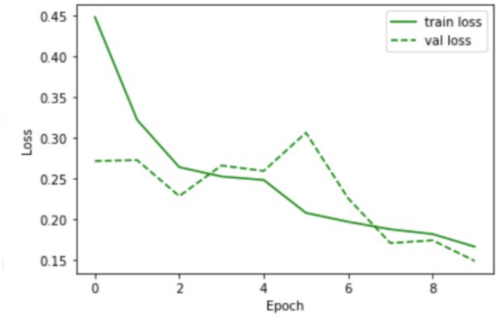
```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: 0.4484 - accuracy: 0.8125 - val_loss: 0.2712 - val_accur
acy: 0.8958
Epoch 2/10
85/85 [==============================] - 0s 5ms/step - loss: 0.3219 - accuracy: 0.8684 - val_loss: 0.2723 - val_accur
acy: 0.9146
Epoch 3/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2636 - accuracy: 0.8989 - val_loss: 0.2280 - val_accur
acy: 0.9125
Epoch 4/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2521 - accuracy: 0.9026 - val_loss: 0.2656 - val_accur
acy: 0.9167
Epoch 5/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2478 - accuracy: 0.9099 - val_loss: 0.2590 - val_accur
acy: 0.9125
Epoch 6/10
85/85 [==============================] - 0s 5ms/step - loss: 0.2073 - accuracy: 0.9246 - val_loss: 0.3060 - val_accur
acy: 0.8813
Epoch 7/10
85/85 [==============================] - 0s 5ms/step - loss: 0.1963 - accuracy: 0.9279 - val_loss: 0.2253 - val_accur
acy: 0.9146
Epoch 8/10
85/85 [==============================] - 0s 5ms/step - loss: 0.1871 - accuracy: 0.9290 - val_loss: 0.1701 - val_accur
acy: 0.9312
Epoch 9/10
85/85 [==============================] - 0s 5ms/step - loss: 0.1813 - accuracy: 0.9324 - val_loss: 0.1735 - val_accur
acy: 0.9312
Epoch 10/10
85/85 [==============================] - 0s 5ms/step - loss: 0.1657 - accuracy: 0.9430 - val_loss: 0.1482 - val_accur
acy: 0.9458
```

```
fig = plt.figure()

plt.plot(trained_model.history['loss'],
         label='train loss',
         c='green', ls='-')
plt.plot(trained_model.history['val_loss'],
         label='val loss',
         c='green',ls='--')

# plt.plot(trained_model_aug.history['loss'],
#          label='train loss aug',
#          c='blue', ls='-')
# plt.plot(trained_model_aug.history['val_loss'],
#          label='val loss aug',
#          c='blue',ls='--')

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()
```



We can observe that the Adam optimizers learning curve looks bad as the validation curve diverges and converges back which is not a really good trend we wish for. RMSprop does better than Adam but still has some divergence in some points but is equally as good as SGD. so for these reasons, for further steps we choose SGD as our final optimizer for the network.

8. LAYOUT OF THE SOLUTION:

a) Babysitting and training process:

The babysitting process consists of various checks and parameters that need to be performed and estimated:

i) Disabling regularization and checking for initial Loss

The regularization has been set to 0

```
model.add(Dense(120, activation='LeakyReLU',activity_regularizer=regularizers.l2(0)))
model.add(Dense(84, activation='LeakyReLU',activity_regularizer=regularizers.l2(0)))
```

```
Epoch 1/10
85/85 [==============================] - 2s 13ms/step - loss: 0.4627 - accuracy: 0.7824 - val_loss: 0.6163 - val_accu
racy: 0.8167
Epoch 2/10
85/85 [==============================] - 1s 6ms/step - loss: 0.3462 - accuracy: 0.8500 - val_loss: 0.5561 - val_accur
acy: 0.8562
Epoch 3/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2997 - accuracy: 0.8739 - val_loss: 0.4904 - val_accur
acy: 0.8771
Epoch 4/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2797 - accuracy: 0.8846 - val_loss: 0.4214 - val_accur
acy: 0.8938
Epoch 5/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2553 - accuracy: 0.9055 - val_loss: 0.3331 - val_accur
acy: 0.9208
Epoch 6/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2464 - accuracy: 0.9037 - val_loss: 0.2837 - val_accur
acy: 0.9187
Epoch 7/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2310 - accuracy: 0.9151 - val_loss: 0.2432 - val_accur
acy: 0.9312
Epoch 8/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2200 - accuracy: 0.9151 - val_loss: 0.2190 - val_accur
acy: 0.9146
Epoch 9/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2134 - accuracy: 0.9213 - val_loss: 0.1932 - val_accur
acy: 0.9229
Epoch 10/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2060 - accuracy: 0.9235 - val_loss: 0.2050 - val_accur
acy: 0.9250
```

ii)Introducing regularization of (0.01)

```
    model.add(Dense(120, activation='LeakyReLU',activity_regularizer=regularizers.l2(0.01)))
    model.add(Dense(84, activation='LeakyReLU',activity_regularizer=regularizers.l2(0.01)))
```

```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: 0.4416 - accuracy: 0.8037 - val_loss: 0.6007 - val_accur
acy: 0.8500
Epoch 2/10
85/85 [==============================] - 1s 7ms/step - loss: 0.3105 - accuracy: 0.8739 - val_loss: 0.5472 - val_accur
acy: 0.9104
Epoch 3/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2671 - accuracy: 0.9011 - val_loss: 0.4595 - val_accur
acy: 0.9000
Epoch 4/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2495 - accuracy: 0.9026 - val_loss: 0.4056 - val_accur
acy: 0.9208
Epoch 5/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2244 - accuracy: 0.9210 - val_loss: 0.2904 - val_accur
acy: 0.9375
Epoch 6/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2106 - accuracy: 0.9210 - val_loss: 0.2955 - val_accur
acy: 0.9250
Epoch 7/10
85/85 [==============================] - 1s 6ms/step - loss: 0.1983 - accuracy: 0.9268 - val_loss: 0.1989 - val_accur
acy: 0.9417
Epoch 8/10
85/85 [==============================] - 1s 6ms/step - loss: 0.1874 - accuracy: 0.9335 - val_loss: 0.2017 - val_accur
acy: 0.9333
Epoch 9/10
85/85 [==============================] - 1s 6ms/step - loss: 0.1825 - accuracy: 0.9353 - val_loss: 0.2187 - val_accur
acy: 0.9187
Epoch 10/10
85/85 [==============================] - 1s 6ms/step - loss: 0.1698 - accuracy: 0.9386 - val_loss: 0.1556 - val_accur
acy: 0.9438
```

The initial loss decreased slightly when regularization was set to 0.01

iii)Introducing regularization of (1)

```
        model.add(Dense(120, activation='LeakyReLU',activity_regularizer=regularizers.l2(1)))
        model.add(Dense(84, activation='LeakyReLU',activity_regularizer=regularizers.l2(1)))
```

```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: 0.5652 - accuracy: 0.7982 - val_loss: 0.7068 - val_accur
acy: 0.8604
Epoch 2/10
85/85 [==============================] - 1s 6ms/step - loss: 0.3907 - accuracy: 0.8665 - val_loss: 0.6734 - val_accur
acy: 0.7875
Epoch 3/10
85/85 [==============================] - 1s 6ms/step - loss: 0.3661 - accuracy: 0.8702 - val_loss: 0.6144 - val_accur
acy: 0.9083
Epoch 4/10
85/85 [==============================] - 1s 6ms/step - loss: 0.3270 - accuracy: 0.8938 - val_loss: 0.5435 - val_accur
acy: 0.9104
Epoch 5/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2987 - accuracy: 0.9085 - val_loss: 0.4993 - val_accur
acy: 0.8896
Epoch 6/10
85/85 [==============================] - 1s 6ms/step - loss: 0.2906 - accuracy: 0.9074 - val_loss: 0.4190 - val_accur
acy: 0.8896
Epoch 7/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2822 - accuracy: 0.9132 - val_loss: 0.3102 - val_accur
acy: 0.9250
Epoch 8/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2644 - accuracy: 0.9195 - val_loss: 0.2702 - val_accur
acy: 0.9167
Epoch 9/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2613 - accuracy: 0.9169 - val_loss: 0.2801 - val_accur
acy: 0.8979
Epoch 10/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2409 - accuracy: 0.9342 - val_loss: 0.2509 - val_accur
acy: 0.9104
```

The initial loss increased again when the regularization was set to 1.

iv)Introducing regularization of (100)

```
model.add(Dense(120, activation='LeakyReLU',activity_regularizer=regularizers.l2(100)))
model.add(Dense(84, activation='LeakyReLU',activity_regularizer=regularizers.l2(100)))
```

```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: nan - accuracy: 0.4967 - val_loss: nan - val_accuracy:
0.5063
Epoch 2/10
85/85 [==============================] - 1s 6ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 3/10
85/85 [==============================] - 1s 7ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 4/10
85/85 [==============================] - 1s 7ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 5/10
85/85 [==============================] - 1s 6ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 6/10
85/85 [==============================] - 1s 7ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 7/10
85/85 [==============================] - 1s 7ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 8/10
85/85 [==============================] - 1s 6ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 9/10
85/85 [==============================] - 1s 7ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 10/10
85/85 [==============================] - 1s 7ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
```

We can observe that the loss is going really high if the regularization is > 1. Hence the loss should be negative powers of 10. But we need to find an optimal value for it.

(v)Introducing small regularization and finding the learning rate that makes the loss go down:

a) learning rate= 0.0001

```
model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.SGD(0.0001))
```

When the learning rate was set to 0.0001, the loss reduced from 0.9621 to 0.4810.

```
Epoch 1/10
85/85 [==============================] - 1s 9ms/step - loss: 0.9621 - accuracy: 0.5199 - val_loss: 0.6999 - val_accur
acy: 0.5396
Epoch 2/10
85/85 [==============================] - 1s 7ms/step - loss: 0.8019 - accuracy: 0.5864 - val_loss: 0.6837 - val_accur
acy: 0.6250
Epoch 3/10
85/85 [==============================] - 1s 6ms/step - loss: 0.6916 - accuracy: 0.6324 - val_loss: 0.6560 - val_accur
acy: 0.6604
Epoch 4/10
85/85 [==============================] - 1s 7ms/step - loss: 0.6230 - accuracy: 0.6798 - val_loss: 0.6182 - val_accur
acy: 0.7021
Epoch 5/10
85/85 [==============================] - 1s 6ms/step - loss: 0.5764 - accuracy: 0.7154 - val_loss: 0.5749 - val_accur
acy: 0.7229
Epoch 6/10
85/85 [==============================] - 1s 6ms/step - loss: 0.5526 - accuracy: 0.7257 - val_loss: 0.5367 - val_accur
acy: 0.7521
Epoch 7/10
85/85 [==============================] - 1s 6ms/step - loss: 0.5250 - accuracy: 0.7548 - val_loss: 0.5064 - val_accur
acy: 0.7708
Epoch 8/10
85/85 [==============================] - 1s 6ms/step - loss: 0.5056 - accuracy: 0.7603 - val_loss: 0.4843 - val_accur
acy: 0.7812
Epoch 9/10
85/85 [==============================] - 1s 6ms/step - loss: 0.4917 - accuracy: 0.7699 - val_loss: 0.4691 - val_accur
acy: 0.7875
Epoch 10/10
85/85 [==============================] - 1s 6ms/step - loss: 0.4810 - accuracy: 0.7809 - val_loss: 0.4569 - val_accur
acy: 0.7958
```

b) Learning rate = 0.01

```
model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.SGD(0.01))
```

When the learning rate was set to 0.01, the loss reduced from 0.4170 to 0.1997.

```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: 0.4170 - accuracy: 0.8199 - val_loss: 0.6190 - val_accur
acy: 0.8229
Epoch 2/10
85/85 [==============================] - 1s 6ms/step - loss: 0.3117 - accuracy: 0.8724 - val_loss: 0.5411 - val_accur
acy: 0.8729
Epoch 3/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2726 - accuracy: 0.8960 - val_loss: 0.4764 - val_accur
acy: 0.8979
Epoch 4/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2561 - accuracy: 0.8967 - val_loss: 0.3889 - val_accur
acy: 0.9146
Epoch 5/10
85/85 [==============================] - 1s 8ms/step - loss: 0.2328 - accuracy: 0.9092 - val_loss: 0.3082 - val_accur
acy: 0.9354
Epoch 6/10
85/85 [==============================] - 0s 6ms/step - loss: 0.2201 - accuracy: 0.9195 - val_loss: 0.2387 - val_accur
acy: 0.9312
Epoch 7/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2110 - accuracy: 0.9250 - val_loss: 0.2026 - val_accur
acy: 0.9375
Epoch 8/10
85/85 [==============================] - 1s 6ms/step - loss: 0.1997 - accuracy: 0.9243 - val_loss: 0.1835 - val_accur
acy: 0.9479
```

c) Learning rate= 1000

```
model.compile(loss=categorical_crossentropy,metrics=['accuracy'], optimizer= tf.keras.optimizers.SGD(1000))
```

When the learning rate was set to 1000, the value was too large as the loss has become Nan as shown below.

```
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: nan - accuracy: 0.4996 - val_loss: nan - val_accuracy:
0.5063
Epoch 2/10
85/85 [==============================] - 1s 6ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 3/10
85/85 [==============================] - 1s 6ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 4/10
85/85 [==============================] - 1s 6ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 5/10
85/85 [==============================] - 1s 6ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 6/10
85/85 [==============================] - 1s 6ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 7/10
85/85 [==============================] - 1s 7ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 8/10
85/85 [==============================] - 1s 6ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 9/10
85/85 [==============================] - 1s 6ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 10/10
85/85 [==============================] - 1s 6ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 1/10
85/85 [==============================] - 1s 11ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 2/10
85/85 [==============================] - 1s 10ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
Epoch 3/10
85/85 [==============================] - 1s 10ms/step - loss: nan - accuracy: 0.4989 - val_loss: nan - val_accuracy:
0.5063
```

As the learning rate decreases , the accuracy gos up. But the learning rate should not be too low or too high. For this we run a grid search for both learning rate and regularization and check which combination of values can give us best results.

v) Running Grid search:

We use interval values of both learning rate and regularization with 10 epochs. We varied the range of reg and lr with the following code and ran a grid search.

```
for ct in range(2,3):
    for iterr in range (0,10):
        keras.backend.clear_session()
        reg = random()*pow(10,np.random.randint(-5,0))
        lr = random()*pow(10,-ct)
        print("learning rate:",lr,"reg",reg)
        model = LeNet.build(20,20,1,2,reg,lr)
        model.fit(xTrain, yTrain,validation_data=(xVal, yVal),epochs=ep, verbose=1)
        iterr = iterr+1
```

Learning rate = 10^-2 to 10 ^-4

Regularization = 10^0 to 10^-5

The following are the training outputs:

```
learning rate: 0.008727774042468007 reg 0.021470685266561918
Epoch 1/10
85/85 [==============================] - 1s 8ms/step - loss: 0.4334 - accuracy: 0.8180 - val_loss: 0.6357 - val_accur
acy: 0.8813
Epoch 2/10
85/85 [==============================] - 1s 6ms/step - loss: 0.3007 - accuracy: 0.8809 - val_loss: 0.5865 - val_accur
acy: 0.8896
Epoch 3/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2632 - accuracy: 0.9026 - val_loss: 0.4885 - val_accur
acy: 0.9187
Epoch 4/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2537 - accuracy: 0.9074 - val_loss: 0.4158 - val_accur
acy: 0.9083
Epoch 5/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2290 - accuracy: 0.9151 - val_loss: 0.3229 - val_accur
acy: 0.9250
Epoch 6/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2195 - accuracy: 0.9202 - val_loss: 0.2511 - val_accur
acy: 0.9312
```

```
Epoch 7/10
85/85 [==============================] - 1s 8ms/step - loss: 0.2013 - accuracy: 0.9283 - val_loss: 0.2455 - val_accur
acy: 0.9333
Epoch 8/10
85/85 [==============================] - 1s 7ms/step - loss: 0.2041 - accuracy: 0.9243 - val_loss: 0.2037 - val_accur
acy: 0.9354
Epoch 9/10
85/85 [==============================] - 1s 7ms/step - loss: 0.1845 - accuracy: 0.9412 - val_loss: 0.2627 - val_accur
acy: 0.9104
Epoch 10/10
85/85 [==============================] - 1s 7ms/step - loss: 0.1737 - accuracy: 0.9390 - val_loss: 0.1736 - val_accur
acy: 0.9375
```

So, The optimal learning rate and regularization obtained for this best value is **:**

```
learning rate: 0.008727774042468007 reg 0.021470685266561918
```

9. TESTING THE NETWORK FACE DATASET:

Now, we extend a similar LeNet 5 architecture to train the data on the FaceNonface dataset. We validated 480 images. We have used 800 images for testing- 400 for face and 400 for non-face.

Babysitting follows the same approach, wherein the hyperparameters like regularization and learning rate are obtained using the coarse-fine search strategy.

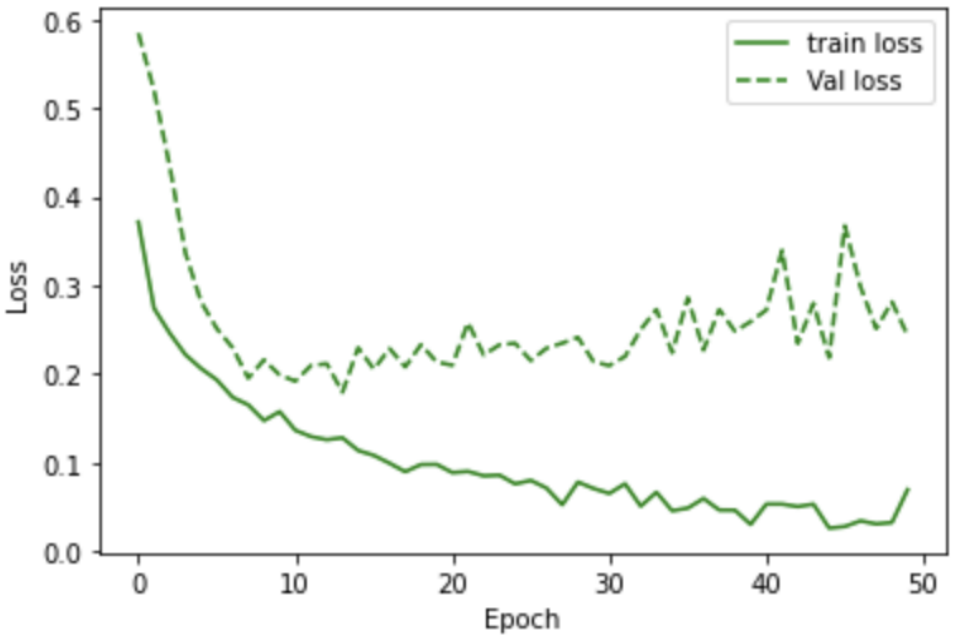
We now ran the search to find the best epoch value for 50 epochs. Then we checked for the best epoch value using the graph and then set that epoch value as the final one and started testing the model.

The values for training for 50 epochs look like:

```
Epoch 1/50
85/85 [==============================] - 1s 8ms/step - loss: 0.3719 - accuracy: 0.8415 - val_loss: 0.5847 - val_accur
acy: 0.8479
Epoch 2/50
85/85 [==============================] - 1s 6ms/step - loss: 0.2742 - accuracy: 0.8934 - val_loss: 0.5200 - val_accur
acy: 0.8979
Epoch 3/50
85/85 [==============================] - 1s 6ms/step - loss: 0.2462 - accuracy: 0.9081 - val_loss: 0.4343 - val_accur
acy: 0.9042
Epoch 4/50
85/85 [==============================] - 1s 7ms/step - loss: 0.2220 - accuracy: 0.9184 - val_loss: 0.3366 - val_accur
acy: 0.9229
Epoch 5/50
85/85 [==============================] - 1s 7ms/step - loss: 0.2064 - accuracy: 0.9217 - val_loss: 0.2814 - val_accur
acy: 0.9208
Epoch 6/50
85/85 [==============================] - 1s 6ms/step - loss: 0.1936 - accuracy: 0.9324 - val_loss: 0.2513 - val_accur
acy: 0.9104
Epoch 7/50
```

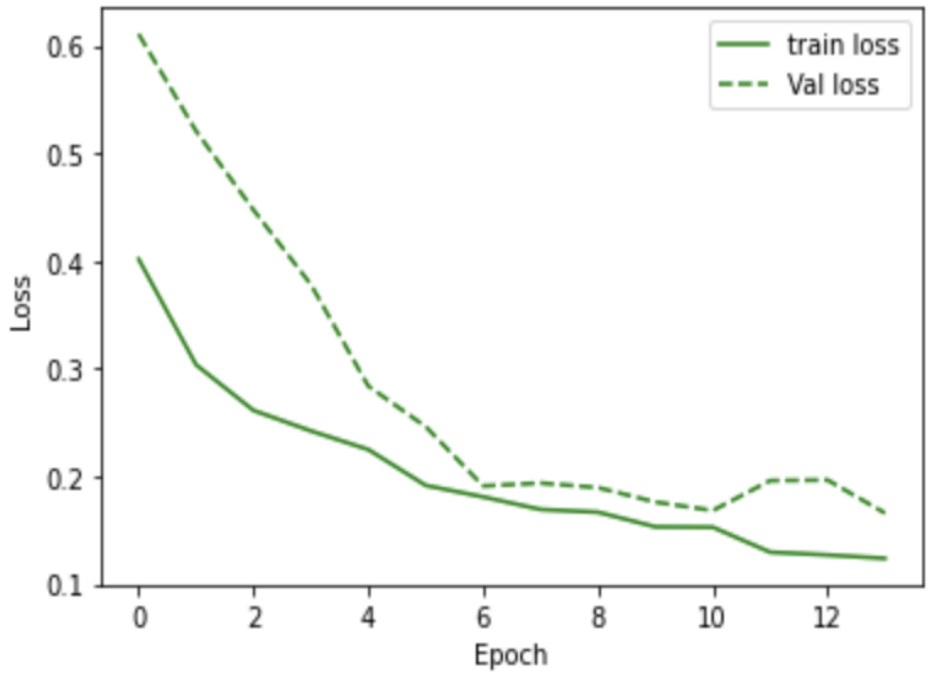
```
racy: 0.9292
Epoch 45/50
43/43 [==============================] - 1s 20ms/step - loss: 0.2243 - accuracy: 0.9081 - val_loss: 0.2056 - val_accu
racy: 0.9250
Epoch 46/50
43/43 [==============================] - 1s 20ms/step - loss: 0.2314 - accuracy: 0.9125 - val_loss: 0.2288 - val_accu
racy: 0.9104
Epoch 47/50
43/43 [==============================] - 1s 20ms/step - loss: 0.2254 - accuracy: 0.9088 - val_loss: 0.1953 - val_accu
racy: 0.9250
Epoch 48/50
43/43 [==============================] - 1s 23ms/step - loss: 0.2293 - accuracy: 0.9110 - val_loss: 0.2030 - val_accu
racy: 0.9083
Epoch 49/50
43/43 [==============================] - 1s 21ms/step - loss: 0.2129 - accuracy: 0.9206 - val_loss: 0.2006 - val_accu
racy: 0.9229
Epoch 50/50
43/43 [==============================] - 1s 20ms/step - loss: 0.2205 - accuracy: 0.9129 - val_loss: 0.2185 - val_accu
racy: 0.9104
```

Final validation accuracy of 91.04% is obtained at 50 epochs.



After obtaining the final Validation accuracy, the least loss has been searched in the val loss, through which we have obtained that epoch 14 had the least loss. After which epoch has been set to 14 and trained. We have obtained the graph below.

Learning curve for final mode with 14 epochs:



We follow a similar procedure for babysitting the network, and on testing we obtain the following output parameters:

```
t_data -= np.mean(t_data)
t_data /= np.std(t_data, axis = 0)

y_predicted = []
op=[0]*len(t_data)
ct=0
for i in t_data:
    i = np.expand_dims(i, axis=0)
    pr = model.predict(i)[0]
    pred=int(np.argmax(pr))
    y_predicted.append(pred)
    ct+=1

ct1 = 0
for i in range (0,len(t_labels),1):
    if y_predicted[i] == t_labels[i]:
        ct1 +=1

y_true = t_labels
target_names = ['Face', 'Non_Face']
print(classification_report(y_true, y_predicted, target_names=target_names))
```

Using the above code snippet, we have tested the model with the 800-image test set and derived final accuracy and the F1 scores from the below classification report**.**

	precision	recall	f1-score	support
Face	0.92	0.90	0.91	400
Non_Face	0.90	0.92	0.91	400
accuracy			0.91	800
macro avg	0.91	0.91	0.91	800
weighted avg	0.91	0.91	0.91	800

Final Config of Neural Networks:

Normalization: Yes

Architecture: LeNet

Data Augmentation: No

Activation: LeakyRelu

Batch Normalization: Yes

Optimizer: SGD

Epoch: 14

Batch Size : 64

Learning rate: 0.0087

Regularization: 0.0214

Final metrics:

Accuracy: 91%

F1 Score: 0.91

Individual contribution for the project:

Sai Srinivas Murthy Narayanam: Implementing LeNet, varying/ exploring initialization, optimizers, batch norm, activations, code for grid search of lr and reg parameters.

Shreya Ramakanth: Normalization, Data Augmentation –Face dataset and CIFAR dataset, fine search to find optimal epoch, code for testing the model and generating classification report.

Sowjanya Srija Movva: reading data, splitting datasets into train, test and val, varying different ranges of learning rate and regularizations, finding the best lr and reg from the search and plotting the graphs to check results.

Discussions and Report: as far as the discussions are concerned, we have all participated and contributed equally to all the group discussions, and also had regular group meetings. For the report, we have all contributed equally – working on a shared google doc and parallelly working on it – taking screenshots of our individual observations and results (mentioned above), syncing often, and bringing together a report that covers all parts of our explorations.