Digital Image Processing Final Project Report: ASL Hand Gesture Image Classification using Convolutional Neural Network

By Arunkumar Ramachandran

ECE 5256

4/28/2021

Contents

Table of Figures

Table of Tables

1. Introduction

Gestures are a form of nonverbal communication in which visible bodily actions are used to communicate important messages, either in speech or together and in parallel with spoken words. Gestures include movement of the hands, face, or other parts of the body. Physical non-verbal communication such as purely expressive displays, proxemics, or displays of joint attention differ from gestures, which communicate specific messages.

2. Goal

The main goal of this project is to classify images according to their respective labels and train the model to obtain good accuracy.

3. Dataset

The data set is used from the site: https://www.kaggle.com/grassknoted/asl-alphabet
There are 87K images, which are of 200x200x3 pixels. The image of the classification can be shown as follows:

Figure 1 Images in my ASL hand gesture dataset

The dataset contains 29 classes, of which 26 are for letters A-Z and three classes for SPACE, DELETE, and NOTHING. I feel the dataset is challenging because of the diversity in the classes, and I might have to resize the images and make sure I do not lose much of the details from the image.

4. Preprocessing Dataset

This section describes the assumptions and procedures used to set up the analysis and obtain solutions for them.

4.1. Train-Test Split:

To tune the hyperparameters, a total of 10,000 values were sent and the train test split used can be shown in the following figure:

```
[ ]
##############################################################################################################################
#Normalizing 1

(trainX, valX, trainY, valY) = train_test_split(X[:10000], y[:10000], test_size=0.20, random_state=1)

trainX = trainX.astype('float32')/255.0
valX = valX.astype('float32')/255.0

print(trainX.shape)
print(trainY.shape)
print(valX.shape)
print(valY.shape)

(8000, 75, 75, 3)
(8000,)
(2000, 75, 75, 3)
(2000,)
```

Figure 2 Train-Test split to tune hyperparameters.

As seen in the above figure, 10,000 datapoints were split and 8000 datapoints were used for train set and 2000 was used for validation/test set.

4.2. Normalize Function

In this project, two normalization methods were used. The two normalization methods used are:

- Normalizing train and test data by dividing it with 255
- Min-Max Function

4.2.1. Normalizing train and test data by dividing it with 255

The train and test data were normalized by dividing it by 255. The function can be depicted as:

- trainX/255
- valX/255

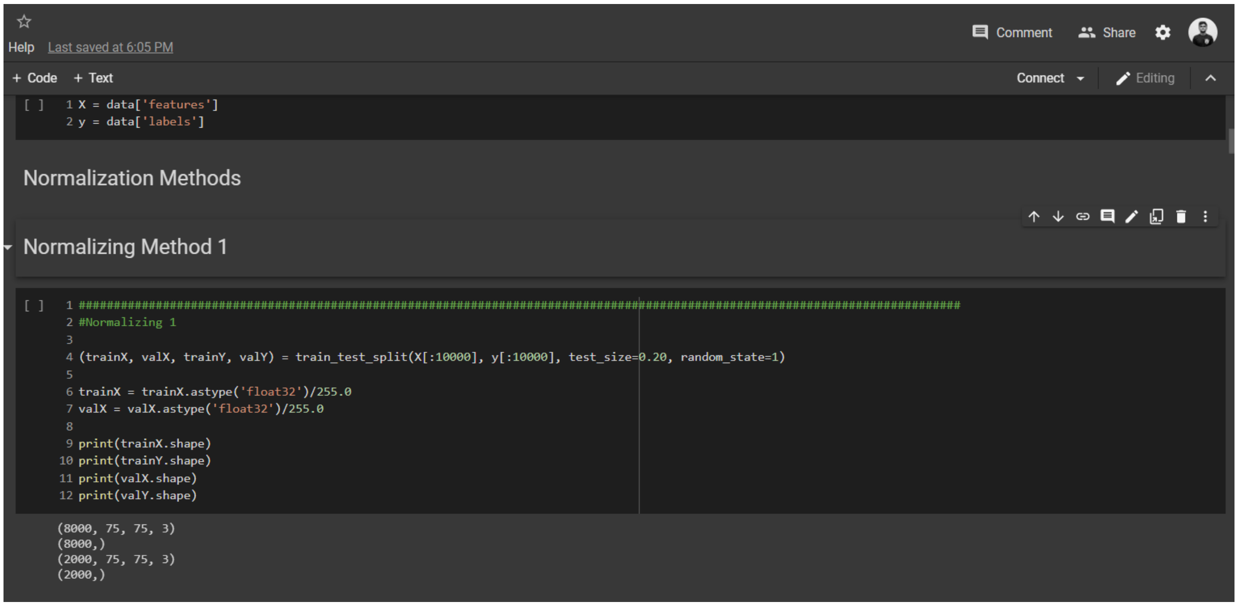


Figure 3 Normalization method dividing train and test data with 255.

4.2.2. Normalize Function using Mean And Standard Deviation:

The test and train set data were normalized by using the mean and standard deviation method.

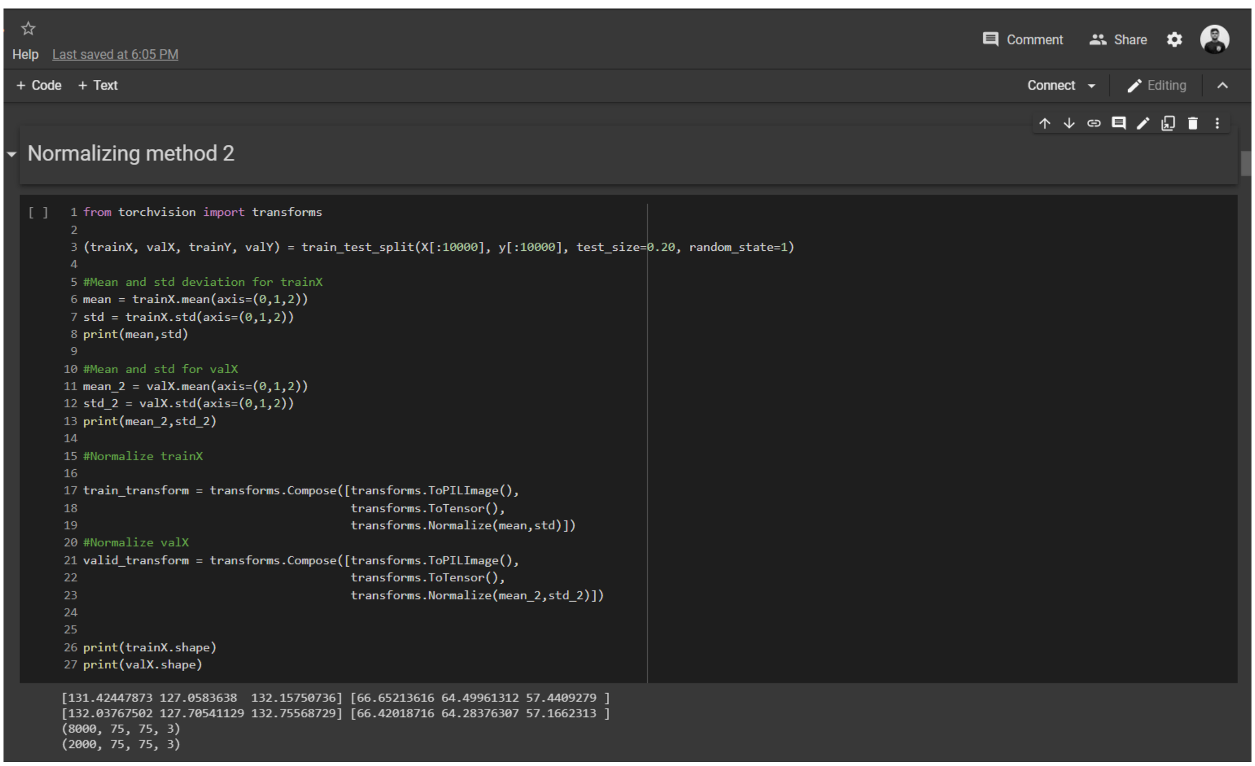


Figure 4 Normalization Using Mean and Standard Deviation.

4.3. Resize the images:

To pick the right image size, random values were given to test how the images look after resizing, and an appropriate size was selected for resizing all images. The images contained in the train and test dataset have been resized from (200*200*3) to (75*75*3).

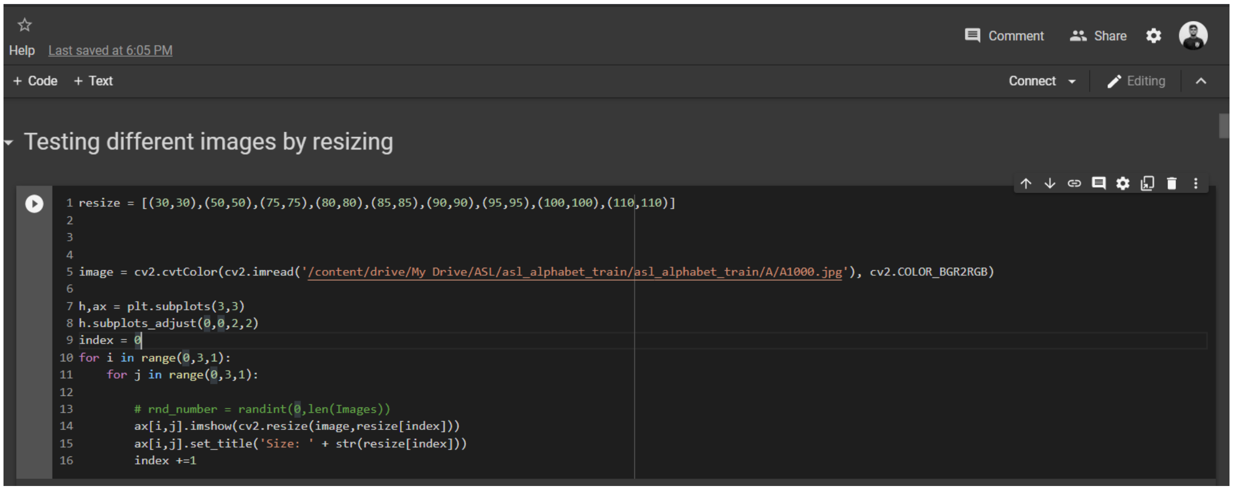


Figure 5 Testing Different sizes for Resizing.

The output after resizing the images in folder A can be shown as follows:

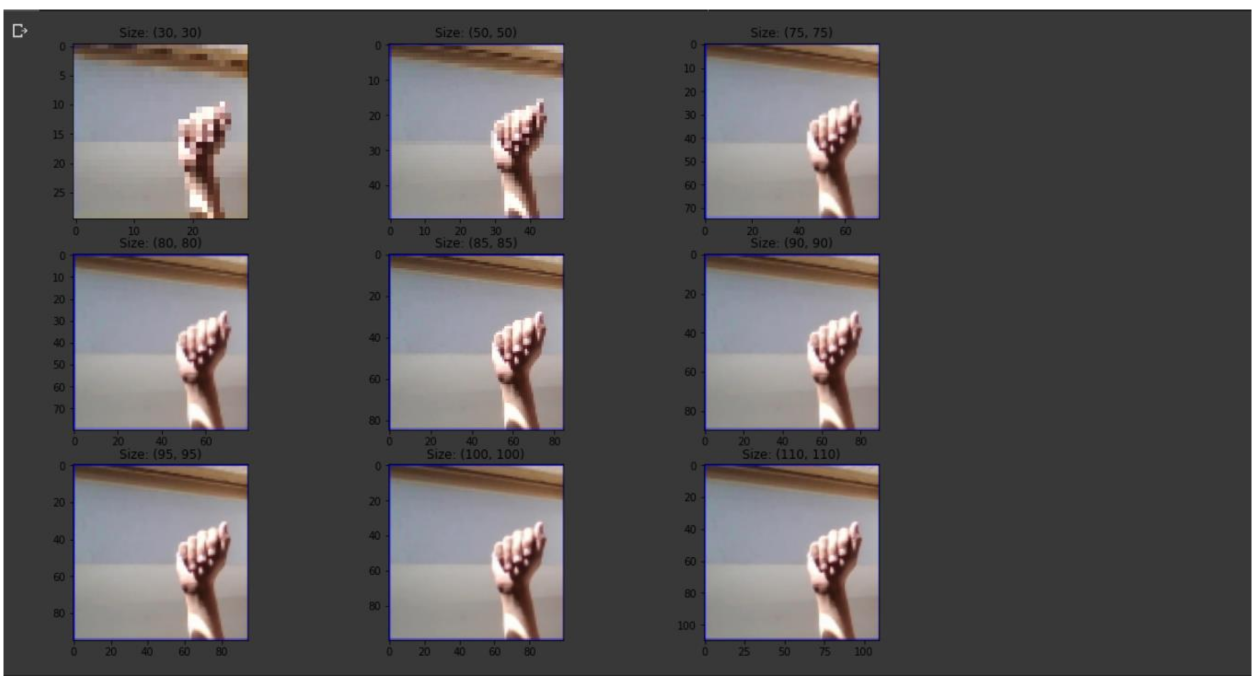
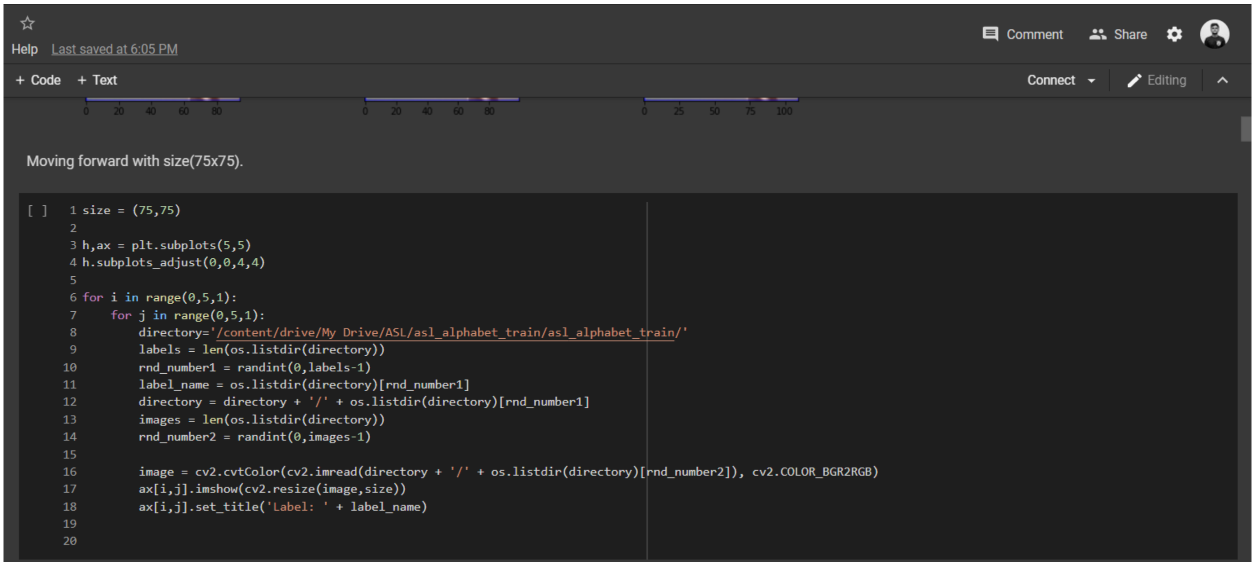


Figure 6 Images after resizing.

As seen above, the images in folder A (containing hand gestures for alphabet A) have been resized according to the given sizes. Amongst them, size (75x75) was selected as a lot of information was not missing from the images, and it looked just about the right size. After selecting (75x75), I ran another code, as shown below, to check how other images looked. They can be seen as follows:




```
size = (75,75)

h,ax = plt.subplots(5,5)
h.subplots_adjust(0,0,4,4)

for i in range(0,5,1):
    for j in range(0,5,1):
        directory='/content/drive/My Drive/ASL/asl_alphabet_train/asl_alphabet_train/'
        labels = len(os.listdir(directory))
        rnd_number1 = randint(0,labels-1)
        label_name = os.listdir(directory)[rnd_number1]
        directory = directory + '/' + os.listdir(directory)[rnd_number1]
        images = len(os.listdir(directory))
        rnd_number2 = randint(0,images-1)

        image = cv2.cvtColor(cv2.imread(directory + '/' + os.listdir(directory)[rnd_number2]), cv2.COLOR_BGR2RGB)
        ax[i,j].imshow(cv2.resize(image,size))
        ax[i,j].set_title('Label: ' + label_name)
```

Figure 7 Code containing Images after resizing to (75x75)

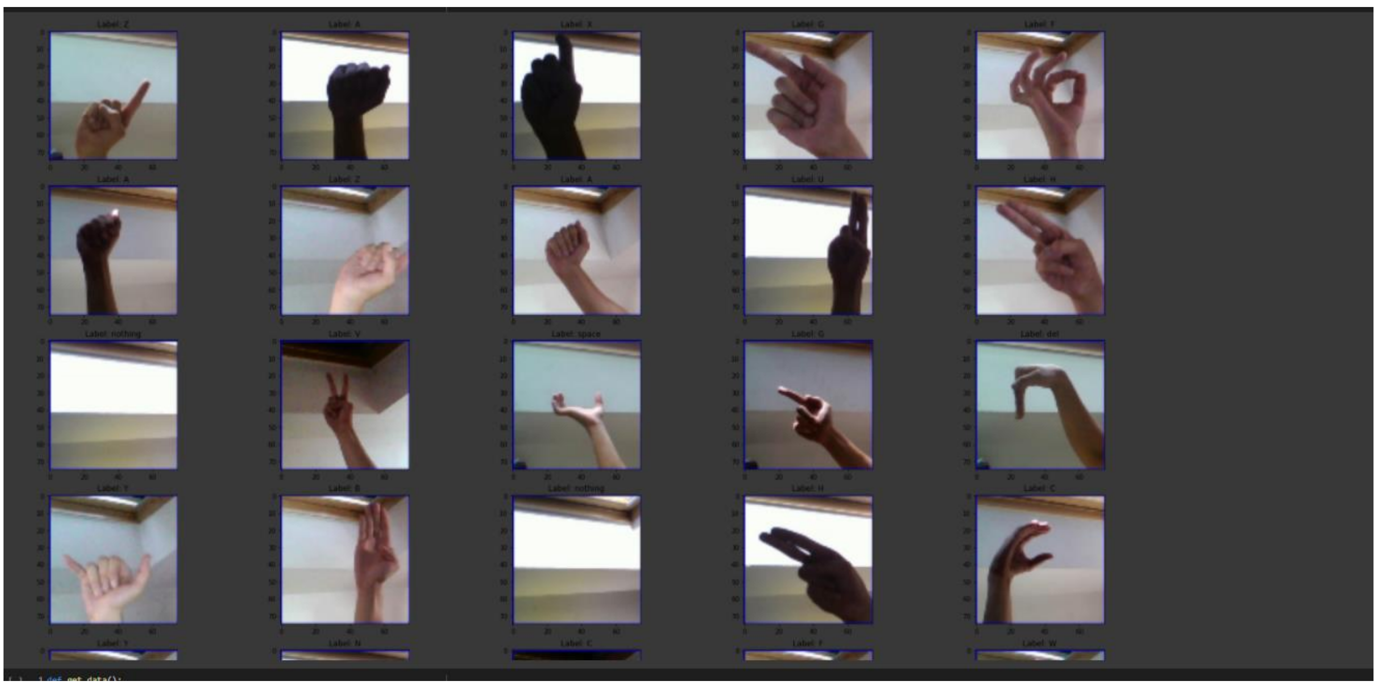


Figure 8 Images after resizing to (75x75)

5. Data Augmentation:

Different data augmentation methods were used in this project. Some of them are as follows:

- Horizontal and Vertical Shift Augmentation.
- Horizontal and Vertical Flip
- Random Rotation
- Random Brightness Augmentation
- Random Zoom Augmentation

5.1. Horizontal and Vertical Shift Augmentation:

A horizontal and vertical shift feature was used, and the images were tested with different values and then plotted.

5.1.1. Horizontal Shift Augmentation:

A horizontal shift augmentation used the “**width_shift_range”** function so that depending on the value provided, the images tend to move horizontally. The function and the range of values provided can be shown as follows:

Horizontal Shift:

```python
# example of zoom image augmentation
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from matplotlib import pyplot
# create data generator
datagen = ImageDataGenerator()

# example of horizontal shift image augmentation
# load the image
img = load_img('/content/drive/My Drive/ASL/asl_alphabet_train/asl_alphabet_train/A/A1000.jpg')
# convert to numpy array
data = img_to_array(img)
# expand dimension to one sample
samples = expand_dims(data, 0)
# create image data augmentation generator
datagen = ImageDataGenerator(width_shift_range=[-100,100])
# prepare iterator
it = datagen.flow(samples, batch_size=1)
# generate samples and plot
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # generate batch of images
    batch = it.next()
    # convert to unsigned integers for viewing
    image = batch[0].astype('uint8')
    # plot raw pixel data
    pyplot.imshow(image)
# show the figure
pyplot.show()
```

Figure 9 A code used in Horizontal shift augmentation.

The output of the augmentation can be shown as follows:

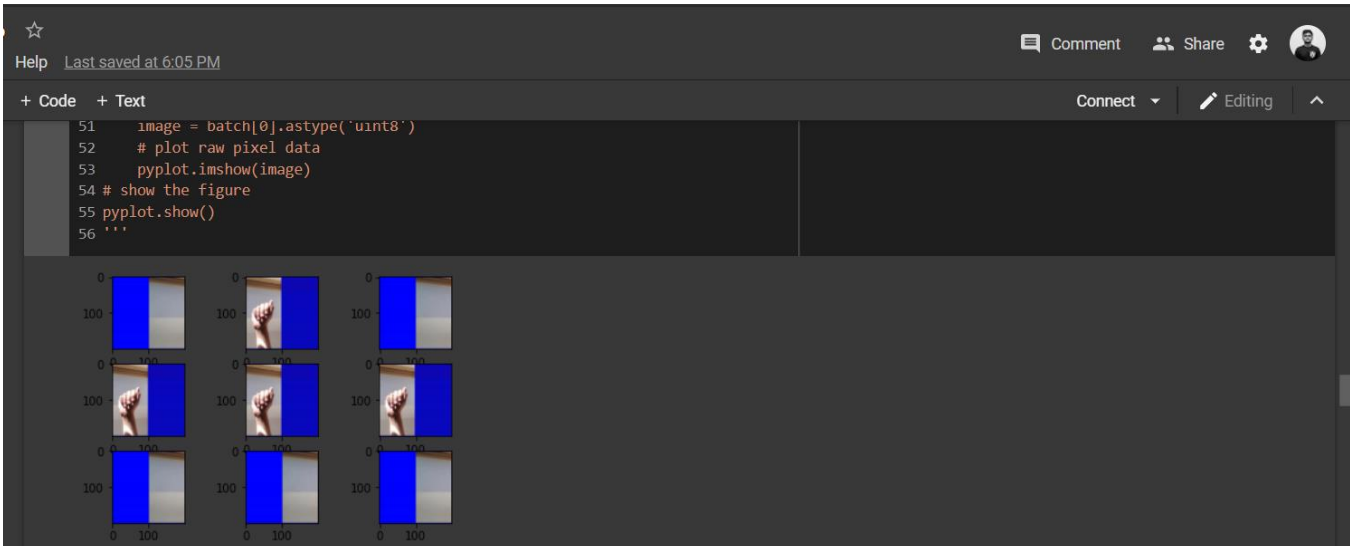
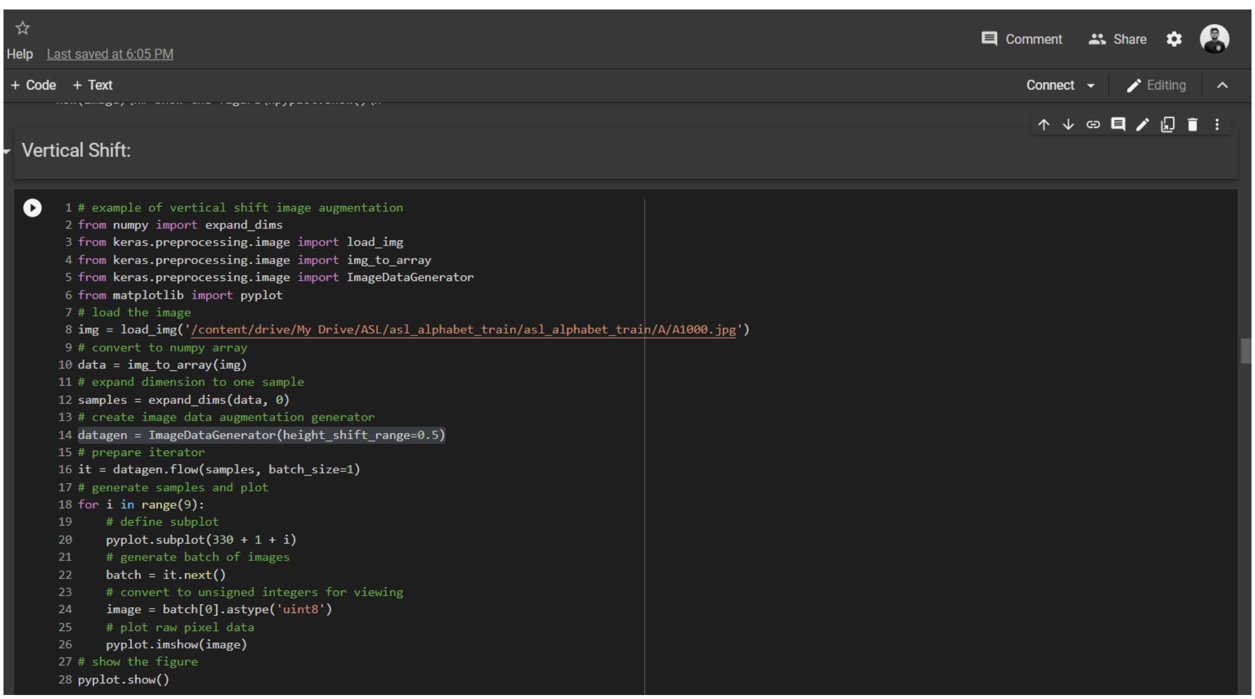


Figure 10 Output obtained after using Horizontal shift augmentation.

5.1.2. Vertical Shift Augmentation:

A Vertical shift augmentation used **"the height_shift_range"** function so that depending on the value provided, the images tend to move vertically. The function and the range of values provided can be shown in the following figure.




```
# example of vertical shift image augmentation
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load the image
img = load_img('/content/drive/My Drive/ASL/asl_alphabet_train/asl_alphabet_train/A/A1000.jpg')
# convert to numpy array
data = img_to_array(img)
# expand dimension to one sample
samples = expand_dims(data, 0)
# create image data augmentation generator
datagen = ImageDataGenerator(height_shift_range=0.5)
# prepare iterator
it = datagen.flow(samples, batch_size=1)
# generate samples and plot
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # generate batch of images
    batch = it.next()
    # convert to unsigned integers for viewing
    image = batch[0].astype('uint8')
    # plot raw pixel data
    pyplot.imshow(image)
# show the figure
pyplot.show()
```

Figure 11 Code used in vertical shift augmentation.

The output of the augmentation can be shown as follows:

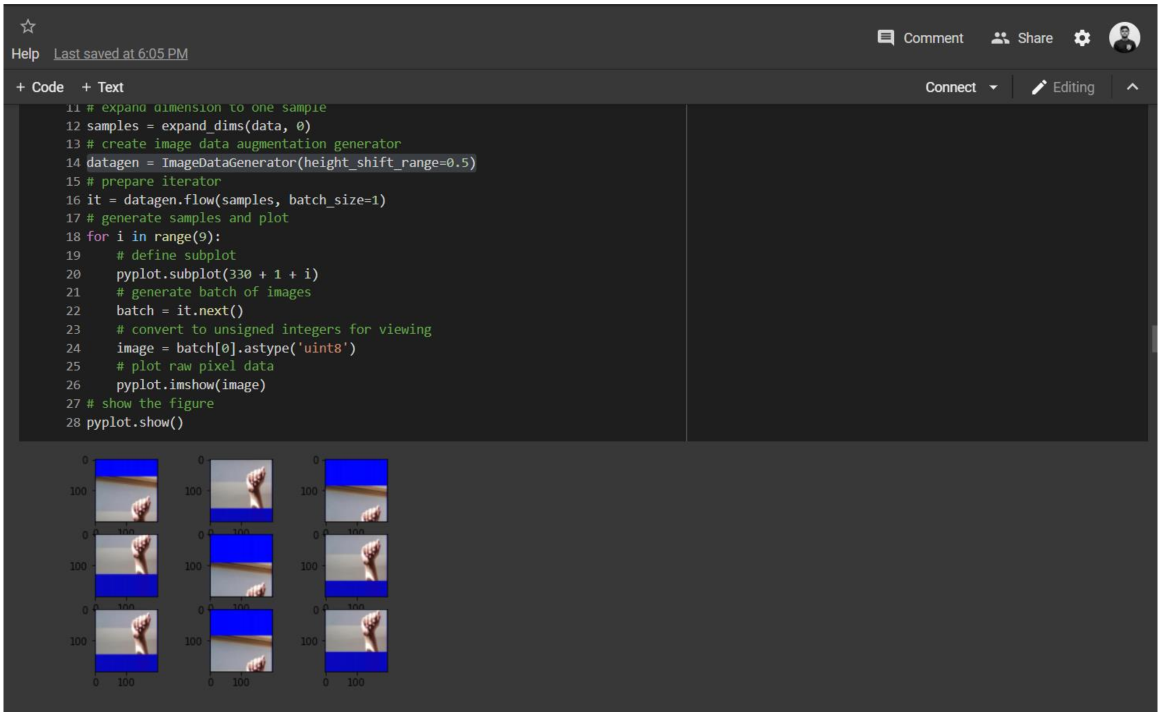


Figure 12 Output obtained after using vertical shift augmentation.

5.2. Horizontal and Vertical Flip Augmentation:

A horizontal and vertical flip feature was used, and the images were tested with different values and then plotted.

5.2.1. Horizontal Flip Augmentation:

A horizontal flip augmentation used the "**horizontal_flip**" function so that depending on the value provided, the images tend to flip horizontally. The function and a Boolean value (True) provided can be shown as follows:

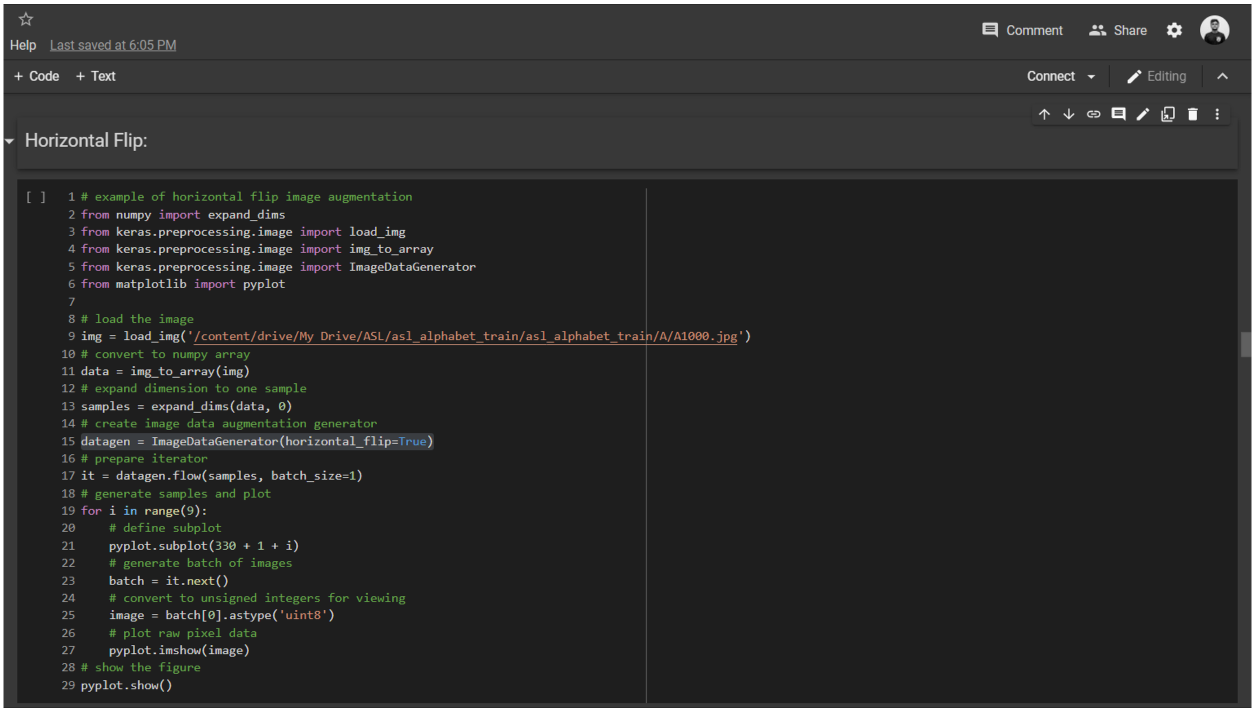


Figure 13 Code used in Horizontal flip augmentation.

The output of the augmentation can be shown as follows:

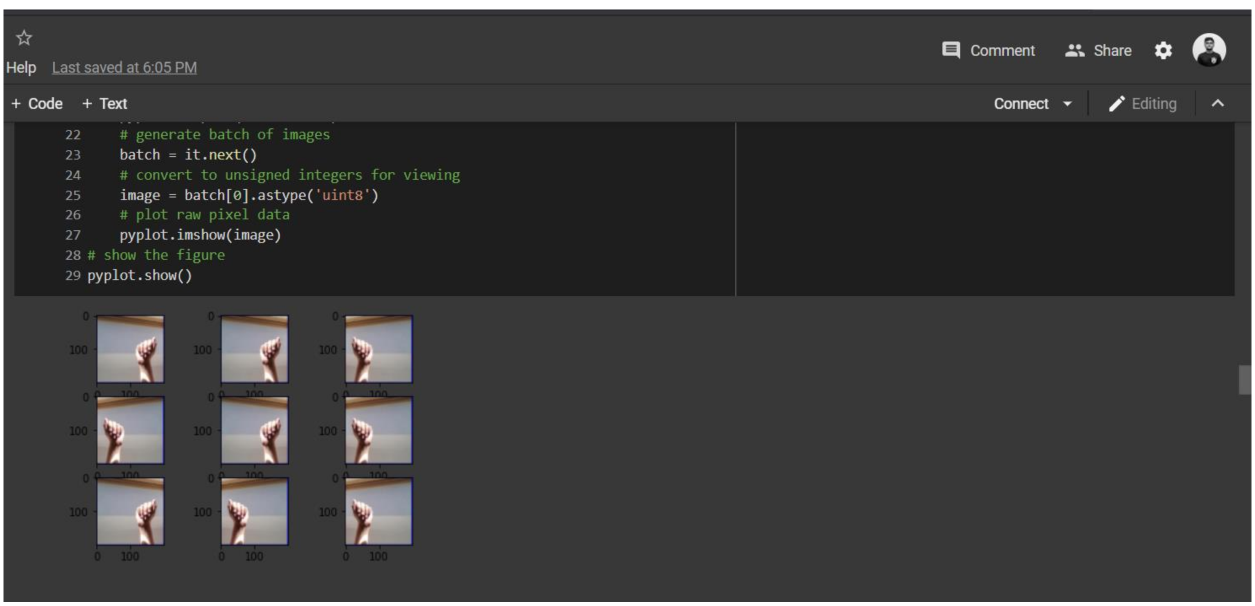


Figure 14 Output obtained after using horizontal flip augmentation.

5.2.2. Vertical Flip Augmentation:

A vertical flip augmentation used the "**vertical_flip**" function so that depending on the value provided, the images tend to move vertically. The function and a Boolean value (True) provided can be shown as follows:

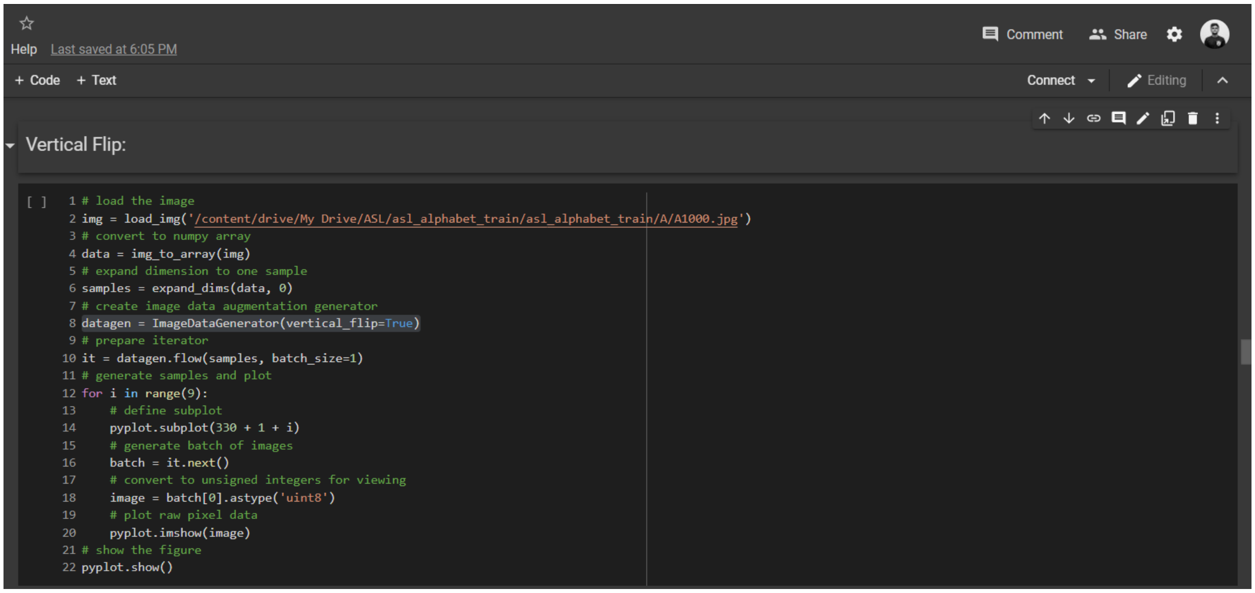


Figure 15 Code used in Vertical flip augmentation.

The output of the augmentation can be shown as follows:

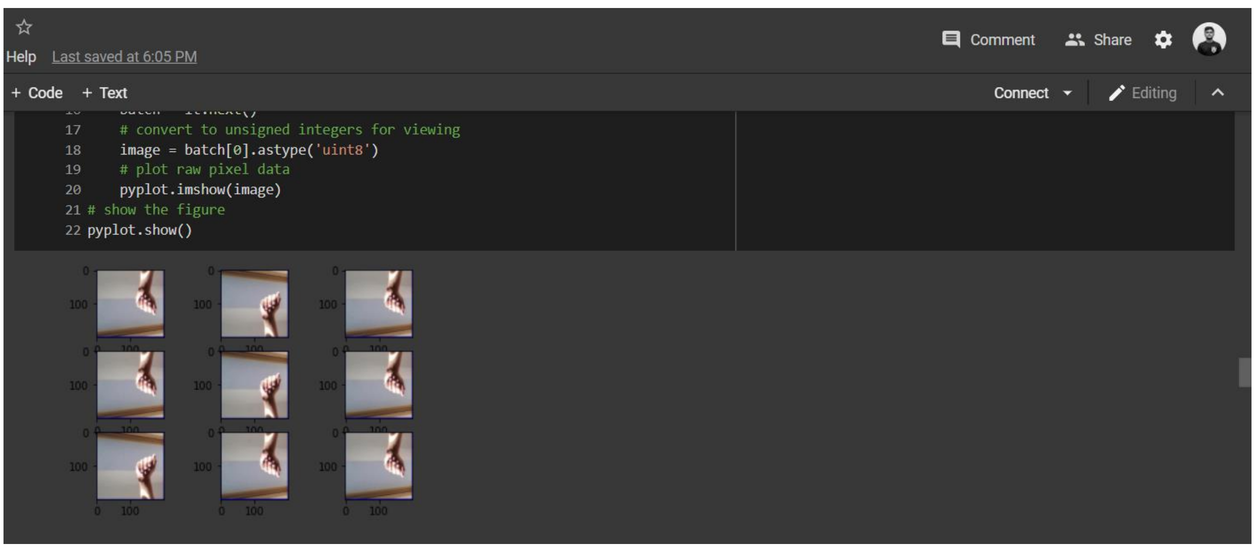


Figure 16 Output obtained after using vertical shift augmentation.

5.3. Random Rotation:

A random rotation augmentation used the **"rotation_range"** function so that depending on the value provided, the images tend to rotate. The function and the range of values provided can be shown as follows:

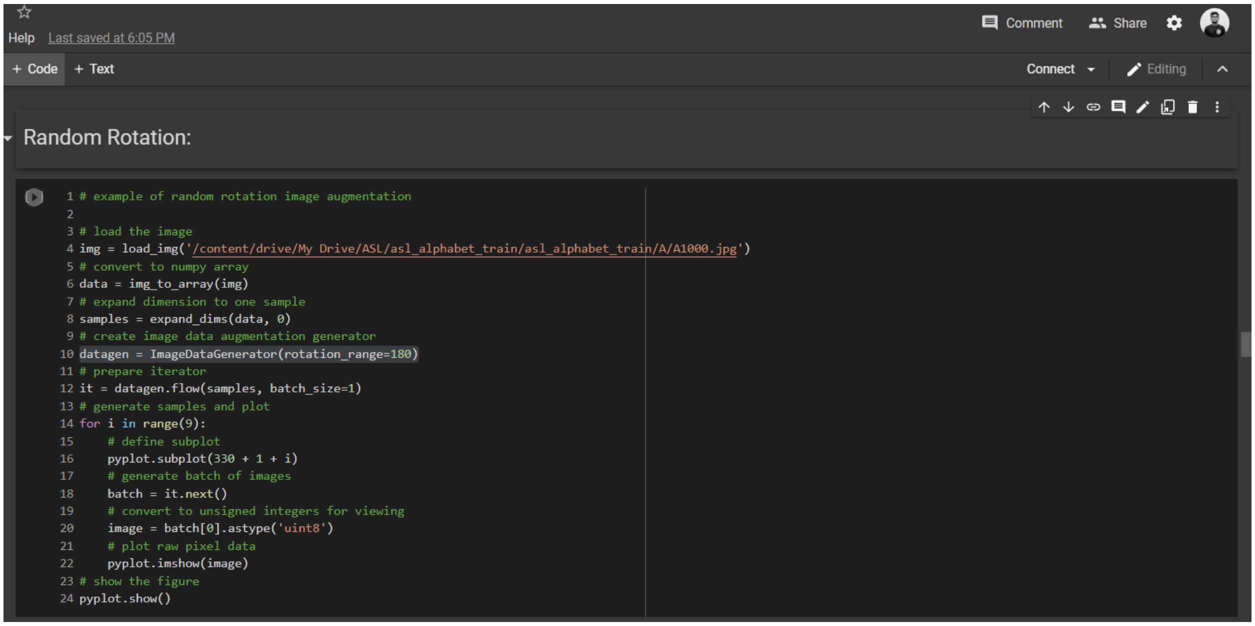


Figure 17 Code used in random rotation augmentation.

The output of the augmentation can be shown as follows:

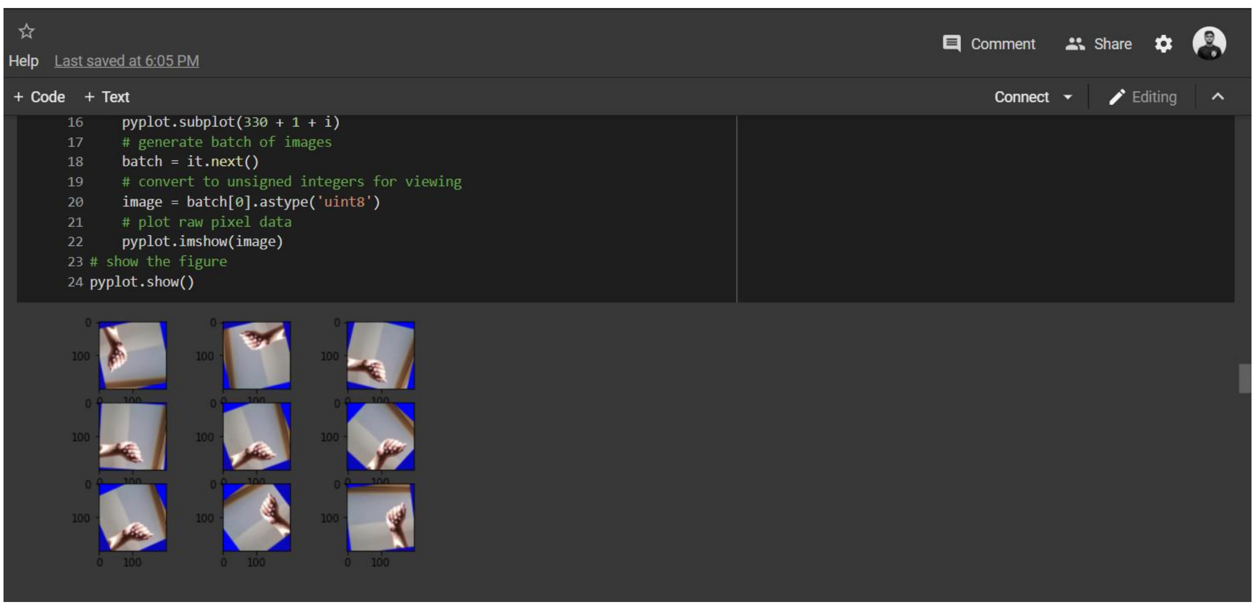


Figure 18 Output obtained after rotation augmentation.

5.4. Random Brightness Augmentation:

A random brightness augmentation used the "**brightness_range"** function so that depending on the value provided, the images tend to rotate. The function and the range of values provided can be shown as follows:

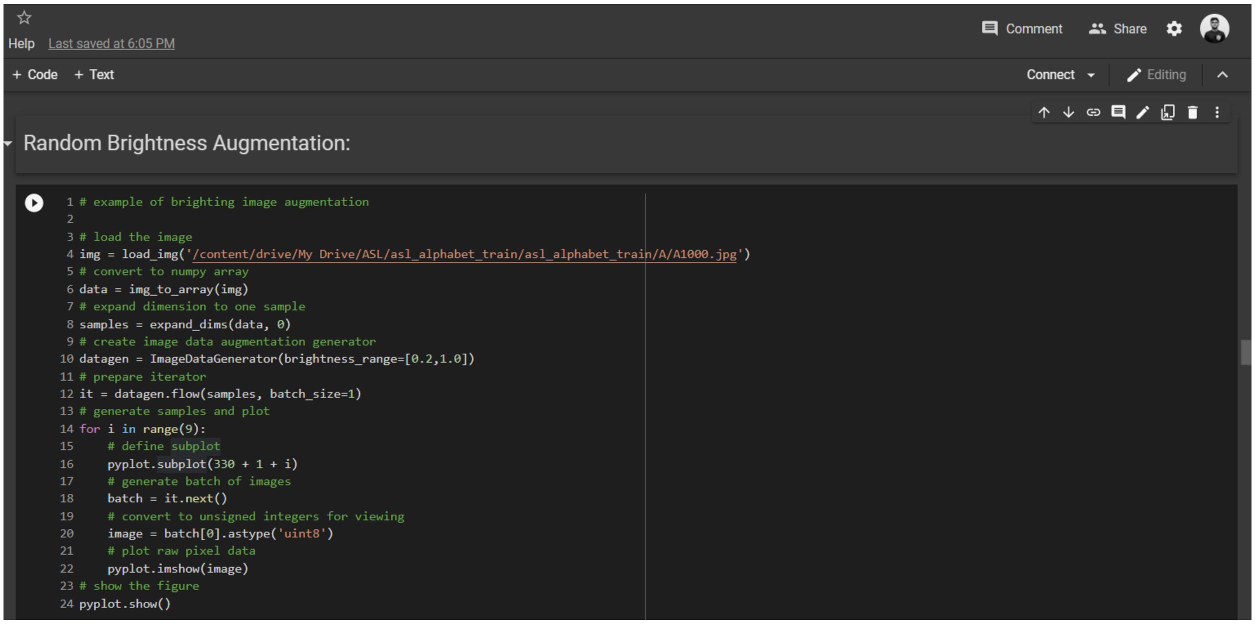


Figure 19 Code used in random brightness augmentation.

The output of the augmentation can be shown as follows:

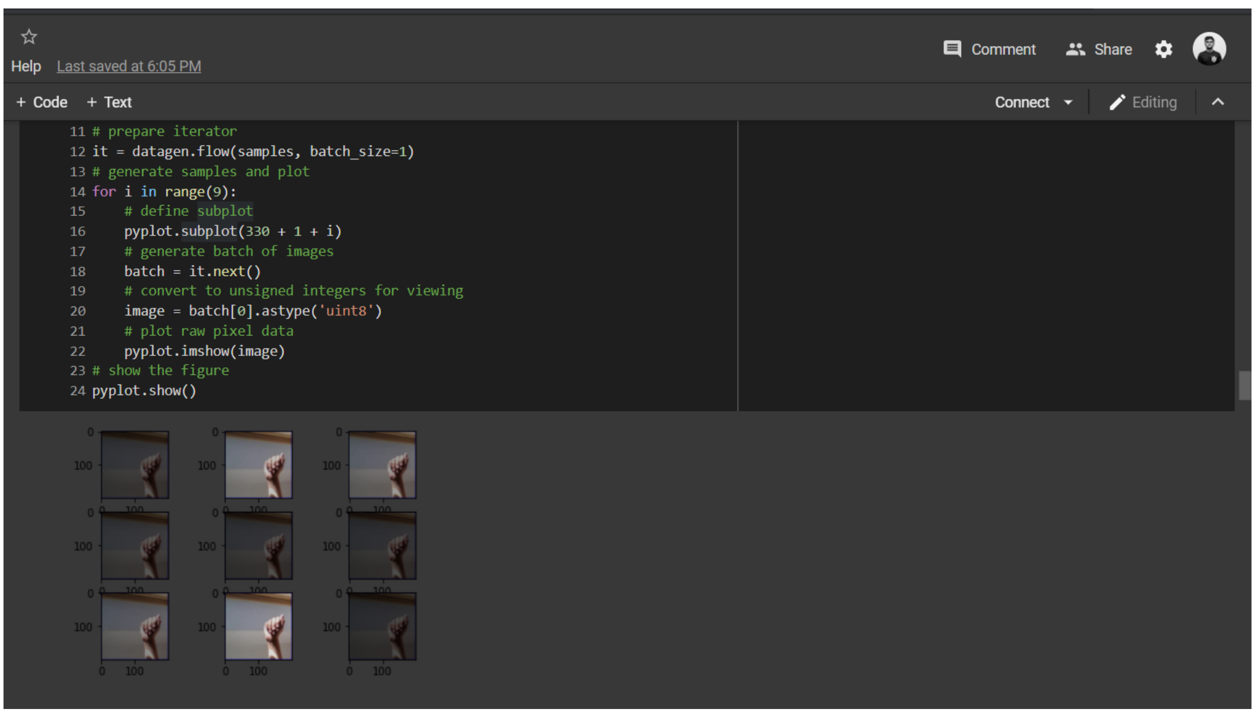


Figure 20 Output obtained after using random brightness augmentation.

5.5. Random Zoom Augmentation:

A random brightness augmentation used the "**zoom_range**" function so that depending on the value provided, the images tend to rotate. The function and the range of values provided can be shown as follows:

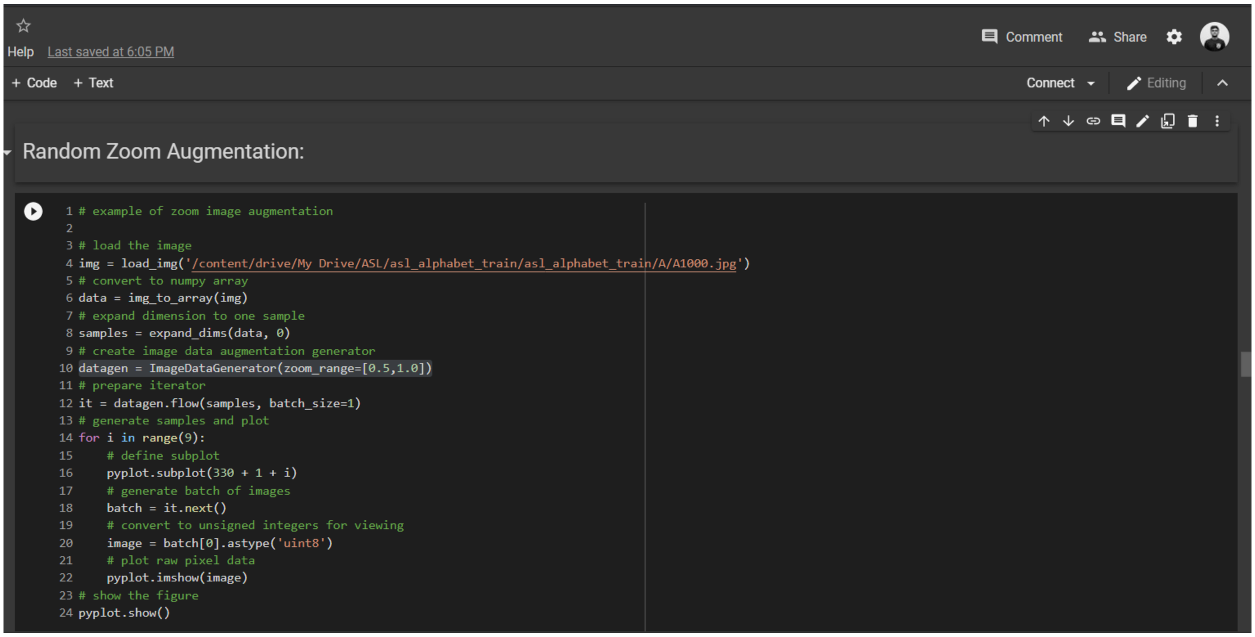


Figure 21 Code used in random zoom augmentation.

The output of the augmentation can be shown as follows:

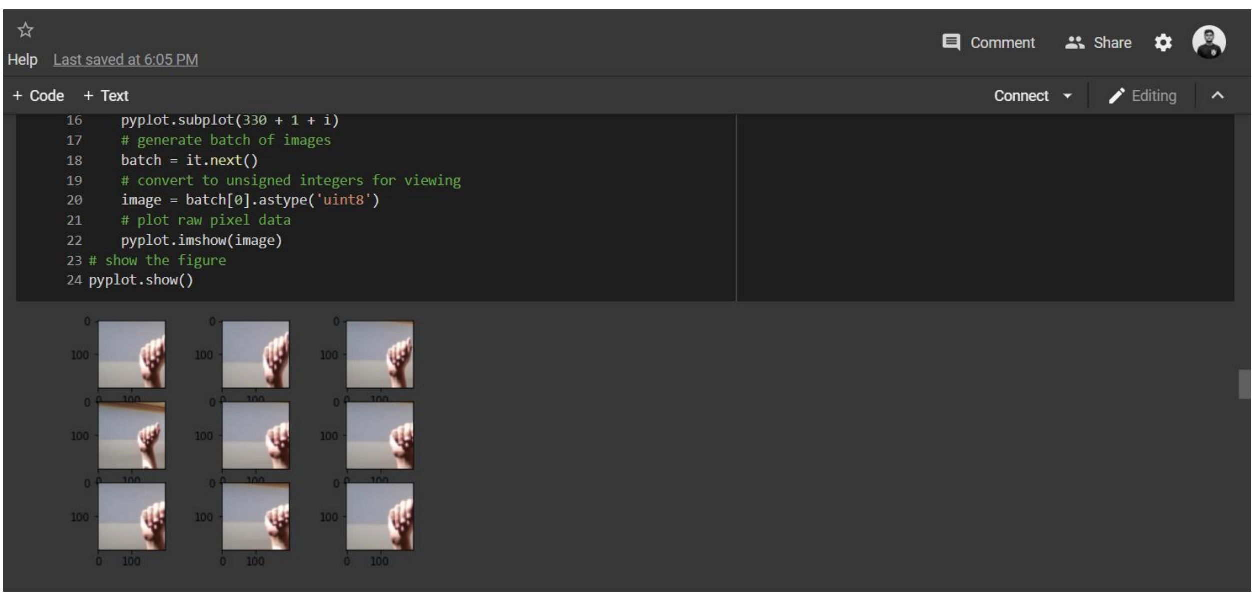


Figure 22 Output obtained after using random zoom augmentation.

5.6. Data Augmentation on different network architectures:

ImageDataGenerator function was used for different nets to calculate the accuracy values on the train set. The nets used and their accuracy can be given as follows:

Table 1 Network Architecture and accuracy values after using image generator.

Network Architecture	Accuracy
VGG19	0.85
VGG16	0.93
AlexNet	0.78
ResNet50	0.69

6. Network Architecture

Different CNN Network architectures were used before tuning the hyperparameters. They are as follows:

- LeNet
- AlexNet
- VGG16
- VGG19
- MiniVGG

Out of these, VGG19 gave the highest accuracy. After selecting that, the rest of the hyperparameters were tuned. The accuracy values obtained for each can be shown in the following table:

Table 2 Network Architecture and their accuracy.

Network Architecture	Accuracy
LeNet	0.90
AlexNet	0.90
VGG16	0.94
VGG19	0.98
MiniVGG	0.91

7. Modern CNN Network Architecture

To test a modern CNN architecture, I used a ResNet50 network architecture and trained the model. The ResNet50 is used primarily to avoid vanishing gradients. It uses the shortcut or skip connection, allowing the gradient to be directly backpropagated to earlier layers. The code is divided into two blocks:

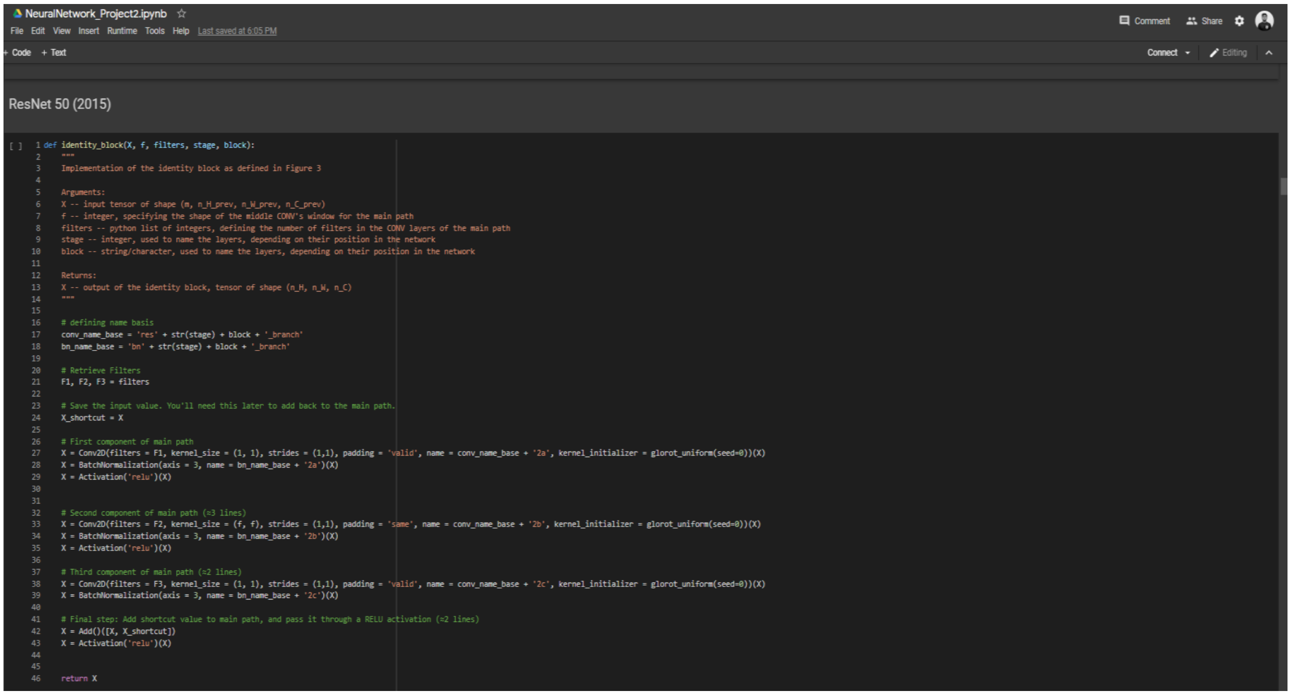
a) Identity Code:

Introduce Conv 2D, back normalization, activation function as a 1st component, 2nd component, and 3rd component, and later the shortcut and input are added together.

b) Conv Code:

The code ensures that the input and output dimensions don't match up.

The code can be shown as follows:



NeuralNetwork_Project2.ipynb

File Edit View Insert Runtime Tools Help Last saved at 6:05 PM

+ Code + Text

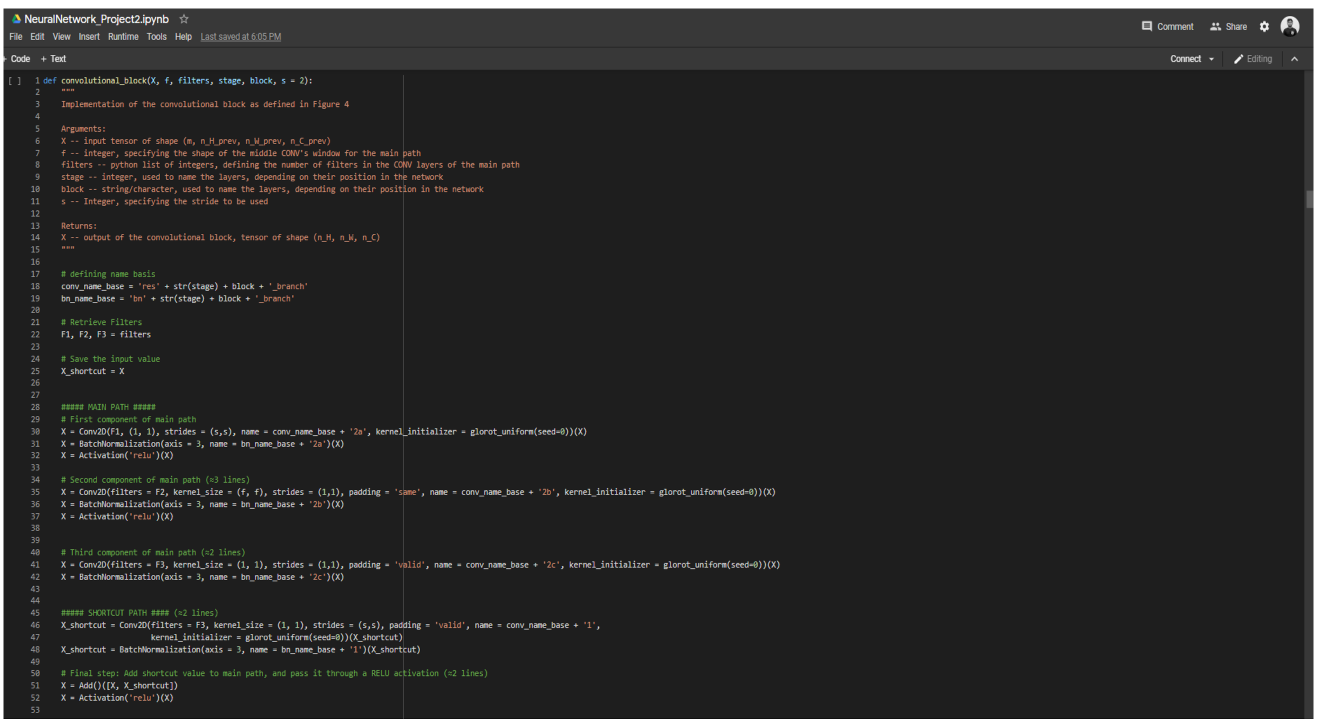
Comment Share

Connect Editing

ResNet 50 (2015)

```python
def identity_block(X, f, filters, stage, block):
    """
    Implementation of the identity block as defined in Figure 3

    Arguments:
    X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
    f -- integer, specifying the shape of the middle CONV's window for the main path
    filters -- python list of integers, defining the number of filters in the CONV layers of the main path
    stage -- integer, used to name the layers, depending on their position in the network
    block -- string/character, used to name the layers, depending on their position in the network

    Returns:
    X -- output of the identity block, tensor of shape (n_H, n_W, n_C)
    """

    # defining name basis
    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    # Retrieve Filters
    F1, F2, F3 = filters

    # Save the input value. You'll need this later to add back to the main path.
    X_shortcut = X

    # First component of main path
    X = Conv2D(filters = F1, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name = conv_name_base + '2a', kernel_initializer = glorot_uniform(seed=0))(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2a')(X)
    X = Activation('relu')(X)


    # Second component of main path (≈3 lines)
    X = Conv2D(filters = F2, kernel_size = (f, f), strides = (1,1), padding = 'same', name = conv_name_base + '2b', kernel_initializer = glorot_uniform(seed=0))(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
    X = Activation('relu')(X)

    # Third component of main path (≈2 lines)
    X = Conv2D(filters = F3, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name = conv_name_base + '2c', kernel_initializer = glorot_uniform(seed=0))(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)

    # Final step: Add shortcut value to main path, and pass it through a RELU activation (≈2 lines)
    X = Add()([X, X_shortcut])
    X = Activation('relu')(X)


    return X
```

Figure 23 Identity Block Of ResNet50 Architecture.



NeuralNetwork_Project2.ipynb

File Edit View Insert Runtime Tools Help Last saved at 6:05 PM

+ Code + Text

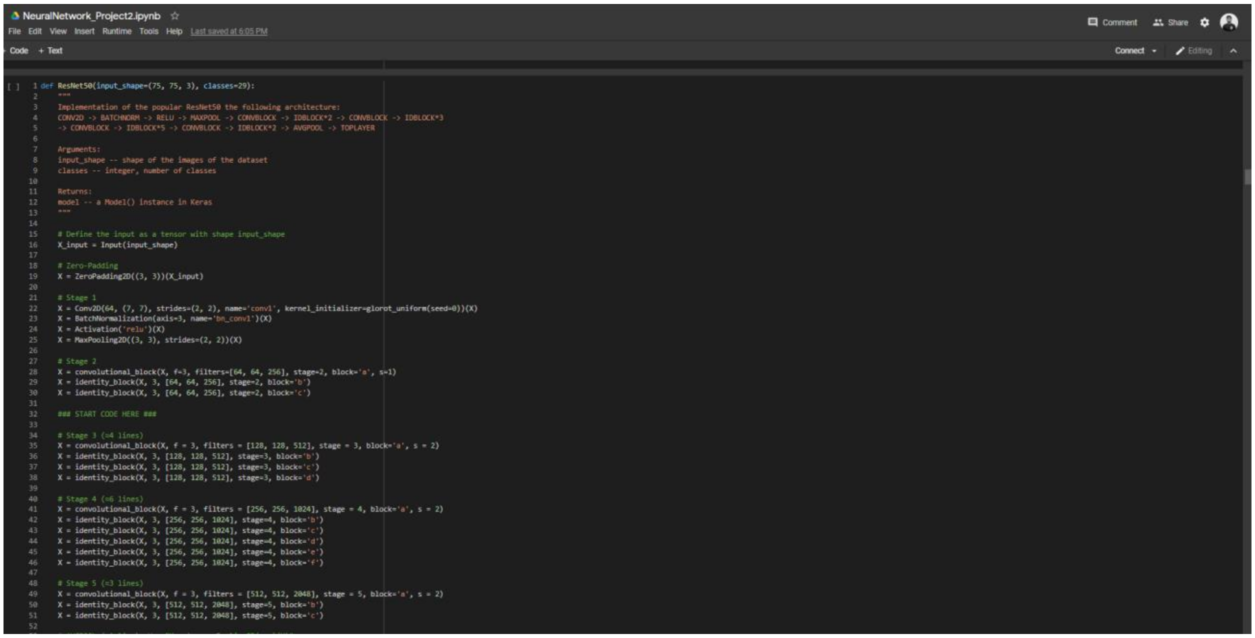
Comment Share

Connect Editing

```python
def convolutional_block(X, f, filters, stage, block, s = 2):
    """
    Implementation of the convolutional block as defined in Figure 4

    Arguments:
    X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
    f -- integer, specifying the shape of the middle CONV's window for the main path
    filters -- python list of integers, defining the number of filters in the CONV layers of the main path
    stage -- integer, used to name the layers, depending on their position in the network
    block -- string/character, used to name the layers, depending on their position in the network
    s -- Integer, specifying the stride to be used

    Returns:
    X -- output of the convolutional block, tensor of shape (n_H, n_W, n_C)
    """

    # defining name basis
    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    # Retrieve Filters
    F1, F2, F3 = filters

    # Save the input value
    X_shortcut = X


    ##### MAIN PATH #####
    # First component of main path
    X = Conv2D(F1, (1, 1), strides = (s,s), name = conv_name_base + '2a', kernel_initializer = glorot_uniform(seed=0))(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2a')(X)
    X = Activation('relu')(X)

    # Second component of main path (≈3 lines)
    X = Conv2D(filters = F2, kernel_size = (f, f), strides = (1,1), padding = 'same', name = conv_name_base + '2b', kernel_initializer = glorot_uniform(seed=0))(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
    X = Activation('relu')(X)


    # Third component of main path (≈2 lines)
    X = Conv2D(filters = F3, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name = conv_name_base + '2c', kernel_initializer = glorot_uniform(seed=0))(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)


    ##### SHORTCUT PATH #### (≈2 lines)
    X_shortcut = Conv2D(filters = F3, kernel_size = (1, 1), strides = (s,s), padding = 'valid', name = conv_name_base + '1',
                        kernel_initializer = glorot_uniform(seed=0))(X_shortcut)
    X_shortcut = BatchNormalization(axis = 3, name = bn_name_base + '1')(X_shortcut)

    # Final step: Add shortcut value to main path, and pass it through a RELU activation (≈2 lines)
    X = Add()([X, X_shortcut])
    X = Activation('relu')(X)

```

Figure 24 Convolutional Block Of ResNet50 Architecture.

The last code contains the Resnet50 function, which contains both the identity and the convolution block.



```
def ResNet50(input_shape=(75, 75, 3), classes=29):
    """
    Implementation of the popular ResNet50 the following architecture:
    CONV2D -> BATCHNORM -> RELU -> MAXPOOL -> CONVBLOCK -> IDBLOCK*2 -> CONVBLOCK -> IDBLOCK*3
    -> CONVBLOCK -> IDBLOCK*5 -> CONVBLOCK -> IDBLOCK*2 -> AVGPOOL -> TOPLAYER

    Arguments:
    input_shape -- shape of the images of the dataset
    classes -- integer, number of classes

    Returns:
    model -- a Model() instance in Keras
    """

    # Define the input as a tensor with shape input_shape
    X_input = Input(input_shape)

    # Zero-Padding
    X = ZeroPadding2D((3, 3))(X_input)

    # Stage 1
    X = Conv2D(64, (7, 7), strides=(2, 2), name='conv1', kernel_initializer=glorot_uniform(seed=0))(X)
    X = BatchNormalization(axis=3, name='bn_conv1')(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((3, 3), strides=(2, 2))(X)

    # Stage 2
    X = convolutional_block(X, f=3, filters=[64, 64, 256], stage=2, block='a', s=1)
    X = identity_block(X, 3, [64, 64, 256], stage=2, block='b')
    X = identity_block(X, 3, [64, 64, 256], stage=2, block='c')

    ### START CODE HERE ###

    # Stage 3 (≈4 lines)
    X = convolutional_block(X, f = 3, filters = [128, 128, 512], stage = 3, block='a', s = 2)
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='b')
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='c')
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='d')

    # Stage 4 (≈6 lines)
    X = convolutional_block(X, f = 3, filters = [256, 256, 1024], stage = 4, block='a', s = 2)
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='b')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='c')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='d')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='e')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='f')

    # Stage 5 (≈3 lines)
    X = convolutional_block(X, f = 3, filters = [512, 512, 2048], stage = 5, block='a', s = 2)
    X = identity_block(X, 3, [512, 512, 2048], stage=5, block='b')
    X = identity_block(X, 3, [512, 512, 2048], stage=5, block='c')
```

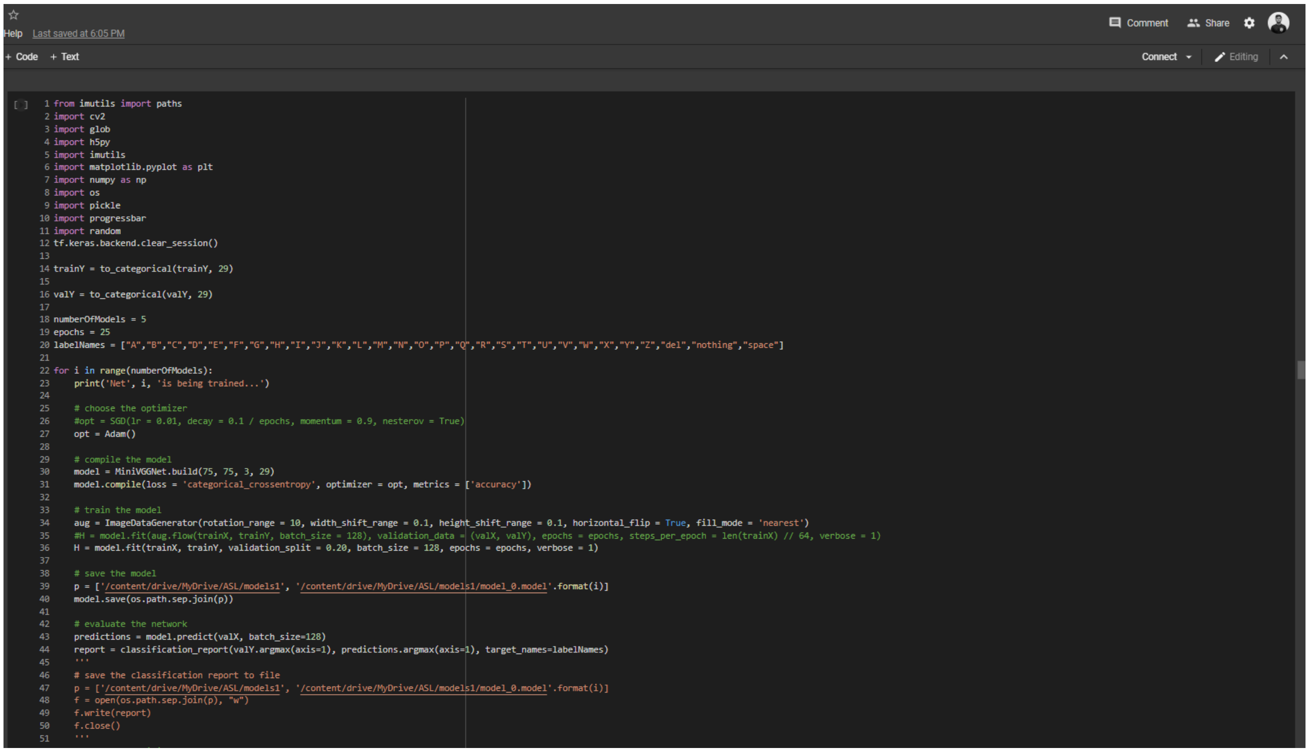
Figure 25 ResNet50 Architecture containing both identity and convolution block.

A final accuracy of 0.90 was obtained on running the code.

8. Ensemble Learning:

8.1. Mini VGG:

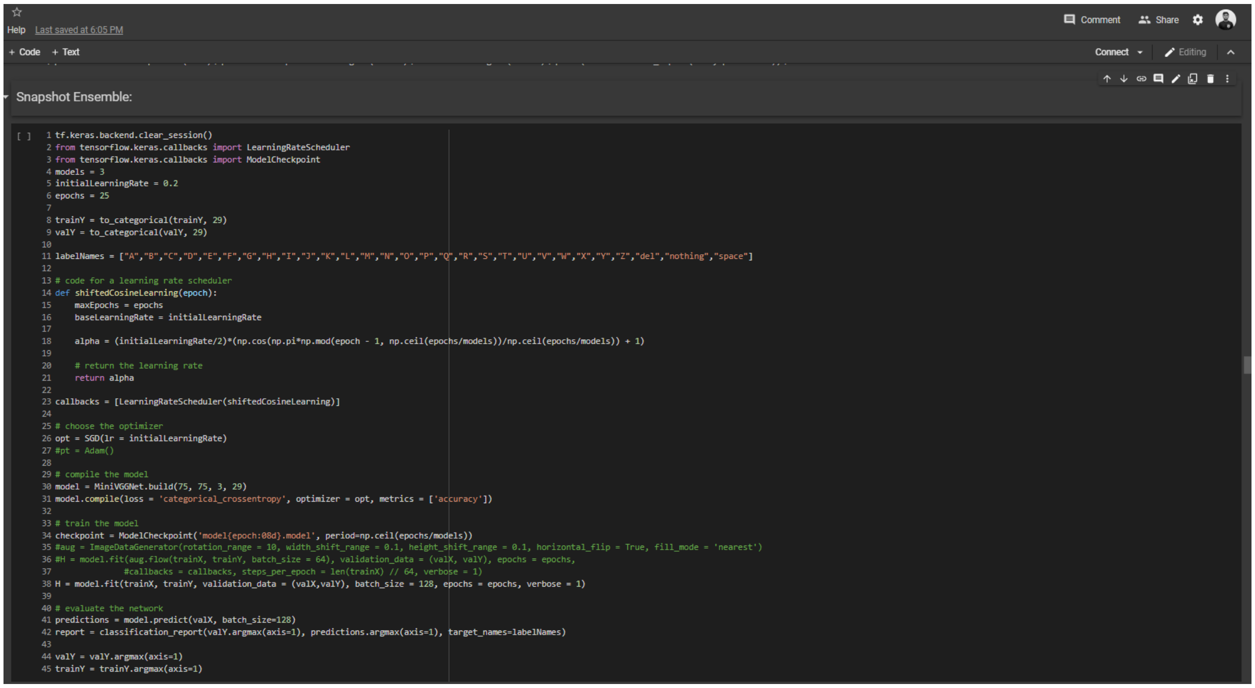
The code of my ensemble using Mini VGG can be shown as follows:



```python
from imutils import paths
import cv2
import glob
import h5py
import imutils
import matplotlib.pyplot as plt
import numpy as np
import os
import pickle
import progressbar
import random
tf.keras.backend.clear_session()

trainY = to_categorical(trainY, 29)

valY = to_categorical(valY, 29)

numberOfModels = 5
epochs = 25
labelNames = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O","P","Q","R","S","T","U","V","W","X","Y","Z","del","nothing","space"]

for i in range(numberOfModels):
    print('Net', i, 'is being trained...')

    # choose the optimizer
    #opt = SGD(lr = 0.01, decay = 0.1 / epochs, momentum = 0.9, nesterov = True)
    opt = Adam()

    # compile the model
    model = MiniVGGNet.build(75, 75, 3, 29)
    model.compile(loss = 'categorical_crossentropy', optimizer = opt, metrics = ['accuracy'])

    # train the model
    aug = ImageDataGenerator(rotation_range = 10, width_shift_range = 0.1, height_shift_range = 0.1, horizontal_flip = True, fill_mode = 'nearest')
    #H = model.fit(aug.flow(trainX, trainY, batch_size = 128), validation_data = (valX, valY), epochs = epochs, steps_per_epoch = len(trainX) // 64, verbose = 1)
    H = model.fit(trainX, trainY, validation_split = 0.20, batch_size = 128, epochs = epochs, verbose = 1)

    # save the model
    p = ['/content/drive/MyDrive/ASL/models1', '/content/drive/MyDrive/ASL/models1/model_0.model'.format(i)]
    model.save(os.path.sep.join(p))

    # evaluate the network
    predictions = model.predict(valX, batch_size=128)
    report = classification_report(valY.argmax(axis=1), predictions.argmax(axis=1), target_names=labelNames)
    '''
    # save the classification report to file
    p = ['/content/drive/MyDrive/ASL/models1', '/content/drive/MyDrive/ASL/models1/model_0.model'.format(i)]
    f = open(os.path.sep.join(p), "w")
    f.write(report)
    f.close()
    '''
```

Figure 26 Ensemble of nets output of MiniVGG.

To test a different type of ensemble, a snapshot ensemble was used to train my model:



Snapshot Ensemble:

```python
tf.keras.backend.clear_session()
from tensorflow.keras.callbacks import LearningRateScheduler
from tensorflow.keras.callbacks import ModelCheckpoint
models = 3
initialLearningRate = 0.2
epochs = 25

trainY = to_categorical(trainY, 29)
valY = to_categorical(valY, 29)

labelNames = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O","P","Q","R","S","T","U","V","W","X","Y","Z","del","nothing","space"]

# code for a learning rate scheduler
def shiftedCosineLearning(epoch):
    maxEpochs = epochs
    baseLearningRate = initialLearningRate

    alpha = (initialLearningRate/2)*(np.cos(np.pi*np.mod(epoch - 1, np.ceil(epochs/models))/np.ceil(epochs/models)) + 1)

    # return the learning rate
    return alpha

callbacks = [LearningRateScheduler(shiftedCosineLearning)]

# choose the optimizer
opt = SGD(lr = initialLearningRate)
#pt = Adam()

# compile the model
model = MiniVGGNet.build(75, 75, 3, 29)
model.compile(loss = 'categorical_crossentropy', optimizer = opt, metrics = ['accuracy'])

# train the model
checkpoint = ModelCheckpoint('model{epoch:08d}.model', period=np.ceil(epochs/models))
#aug = ImageDataGenerator(rotation_range = 10, width_shift_range = 0.1, height_shift_range = 0.1, horizontal_flip = True, fill_mode = 'nearest')
#H = model.fit(aug.flow(trainX, trainY, batch_size = 64), validation_data = (valX, valY), epochs = epochs,
              #callbacks = callbacks, steps_per_epoch = len(trainX) // 64, verbose = 1)
H = model.fit(trainX, trainY, validation_data = (valX,valY), batch_size = 128, epochs = epochs, verbose = 1)

# evaluate the network
predictions = model.predict(valX, batch_size=128)
report = classification_report(valY.argmax(axis=1), predictions.argmax(axis=1), target_names=labelNames)

valY = valY.argmax(axis=1)
trainY = trainY.argmax(axis=1)
```

Figure 27 Snapshot Ensemble of nets output of MIniVGG.

8.2. VGG16:

The code of my ensemble in VGG16 can be shown as follows:

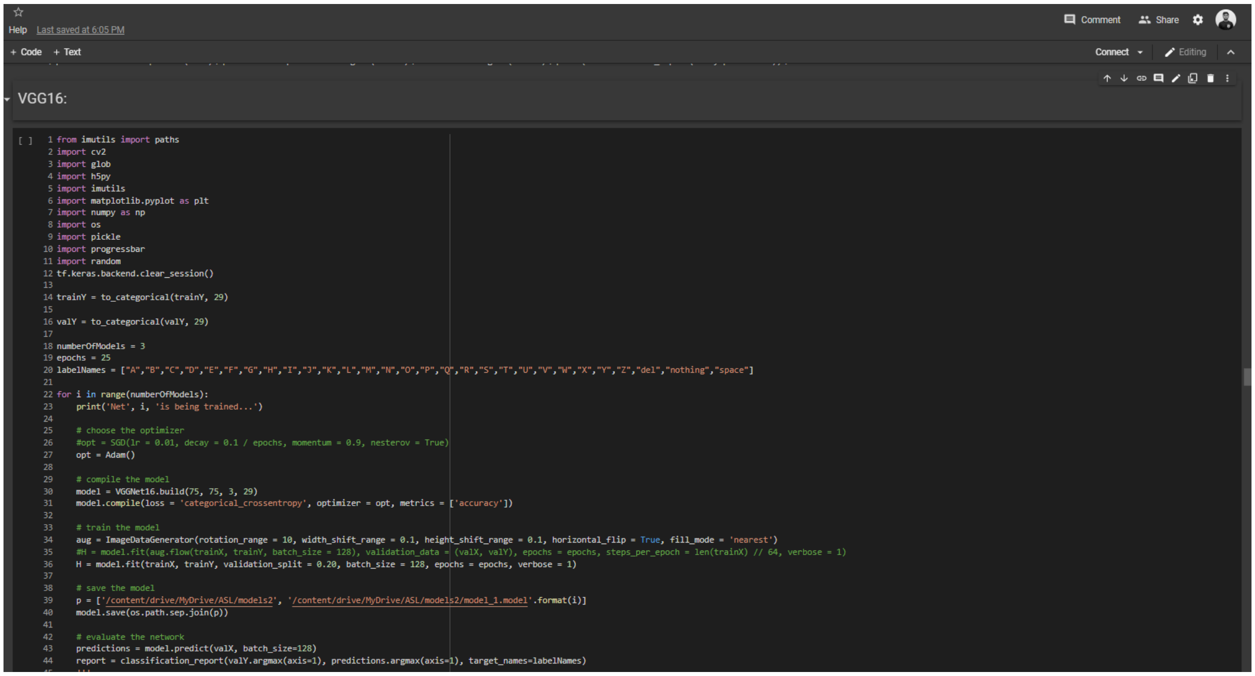


Figure 28 Ensemble of nets output of VGG16.

To test a different type of ensemble, a snapshot ensemble was used to train my model:

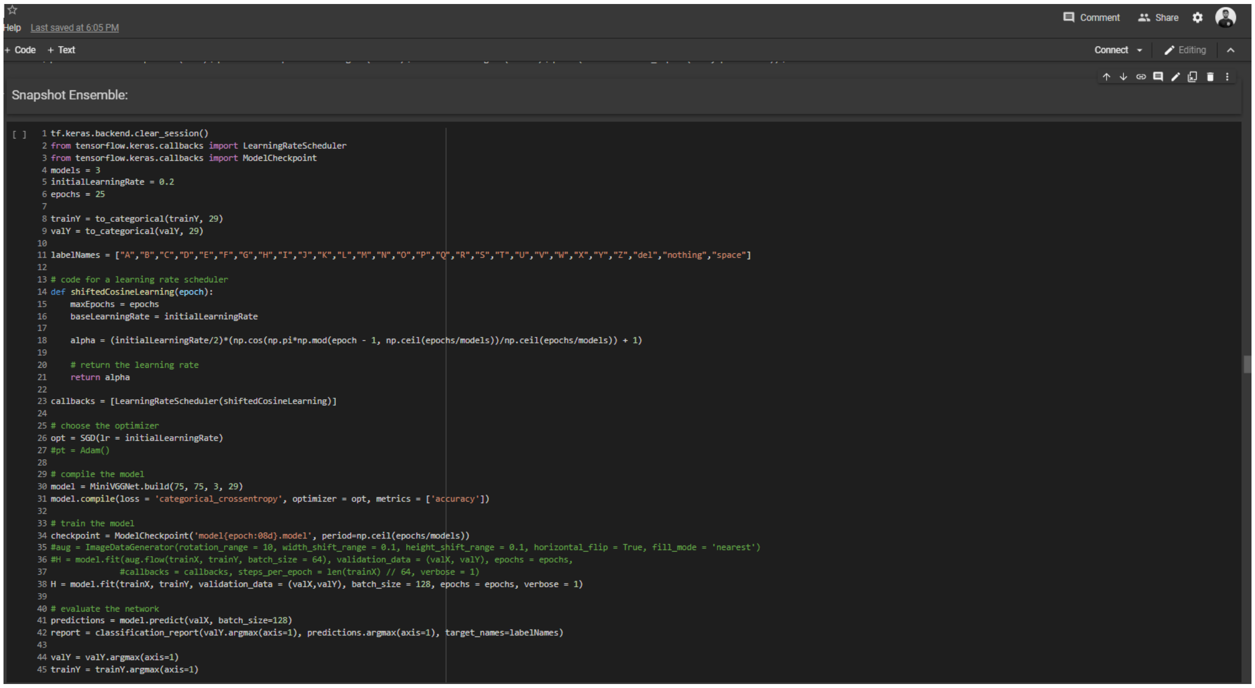
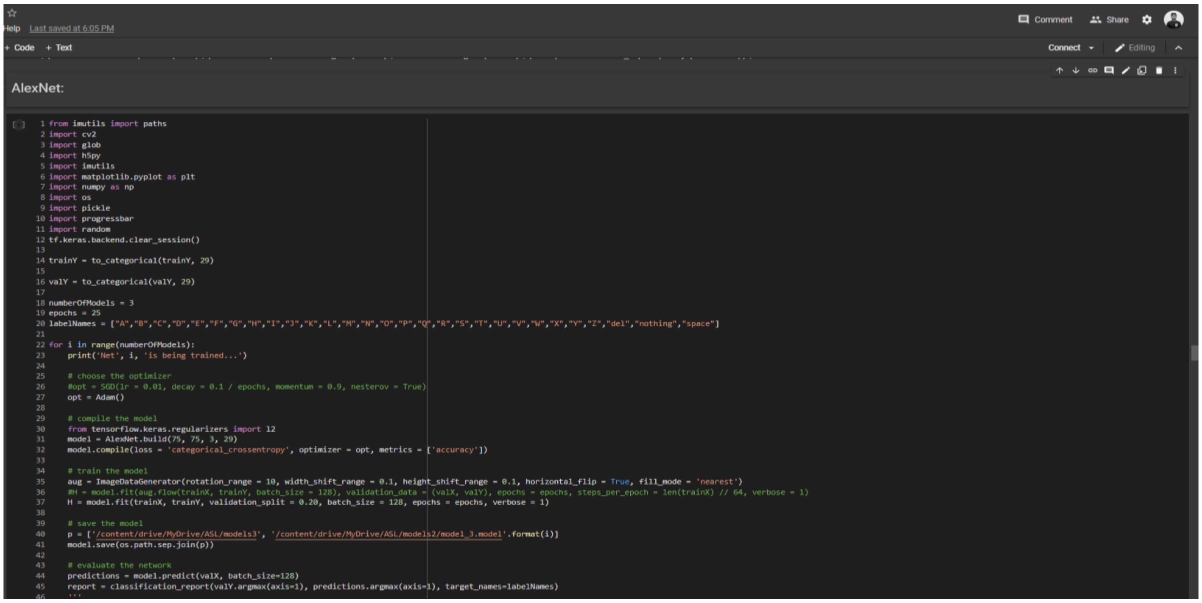


Figure 29 Snapshot Ensemble of nets output of VGG16.

8.3. AlexNet:

The code of my ensemble in AlexNet can be shown as follows:

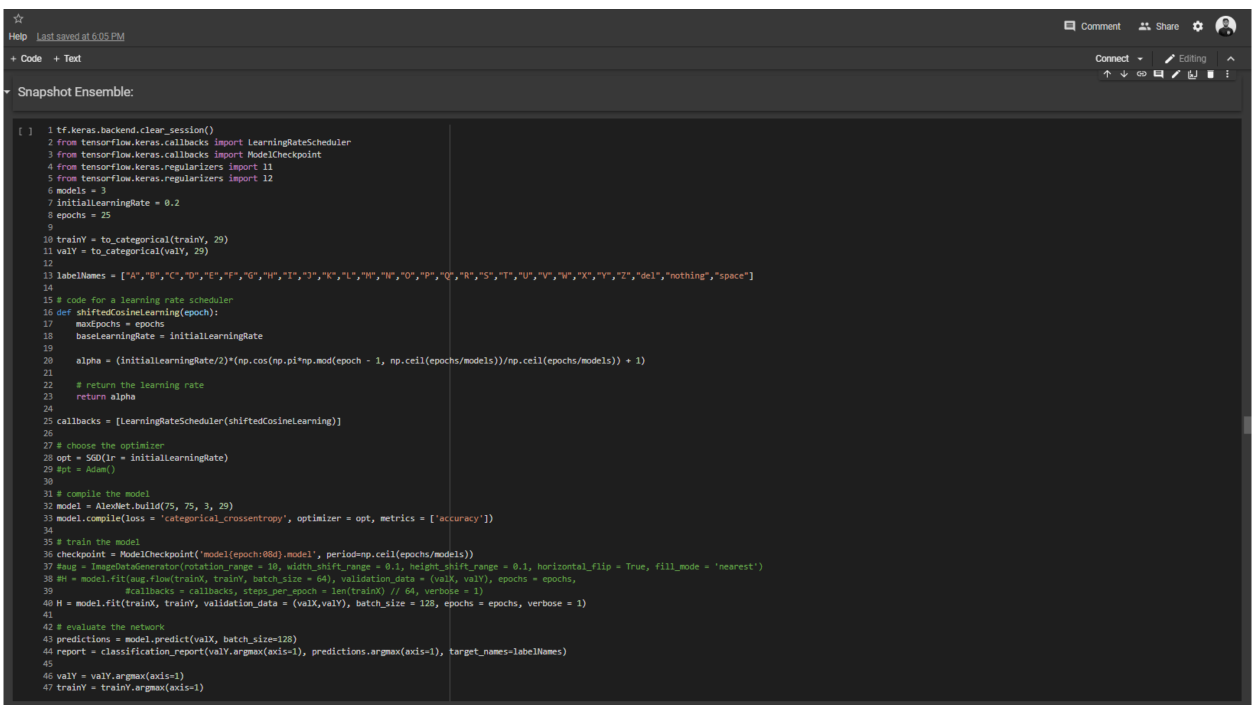


AlexNet:

```python
from imutils import paths
import cv2
import glob
import h5py
import imutils
import matplotlib.pyplot as plt
import numpy as np
import os
import pickle
import progressbar
import random
tf.keras.backend.clear_session()

trainY = to_categorical(trainY, 29)

valY = to_categorical(valY, 29)

numberOfModels = 3
epochs = 25
labelNames = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O","P","Q","R","S","T","U","V","W","X","Y","Z","del","nothing","space"]

for i in range(numberOfModels):
    print('Net', i, 'is being trained...')

    # choose the optimizer
    #opt = SGD(lr = 0.01, decay = 0.1 / epochs, momentum = 0.9, nesterov = True)
    opt = Adam()

    # compile the model
    from tensorflow.keras.regularizers import l2
    model = AlexNet.build(75, 75, 3, 29)
    model.compile(loss = 'categorical_crossentropy', optimizer = opt, metrics = ['accuracy'])

    # train the model
    aug = ImageDataGenerator(rotation_range = 10, width_shift_range = 0.1, height_shift_range = 0.1, horizontal_flip = True, fill_mode = 'nearest')
    #H = model.fit(aug.flow(trainX, trainY, batch_size = 128), validation_data = (valX, valY), epochs = epochs, steps_per_epoch = len(trainX) // 64, verbose = 1)
    H = model.fit(trainX, trainY, validation_split = 0.20, batch_size = 128, epochs = epochs, verbose = 1)

    # save the model
    p = ['/content/drive/MyDrive/ASL/models3', '/content/drive/MyDrive/ASL/models2/model_3.model'.format(i)]
    model.save(os.path.sep.join(p))

    # evaluate the network
    predictions = model.predict(valX, batch_size=128)
    report = classification_report(valY.argmax(axis=1), predictions.argmax(axis=1), target_names=labelNames)
    '''
```

Figure 30 Ensemble of nets output of AlexNet.

To test a different type of ensemble, a snapshot ensemble was used to train my model:



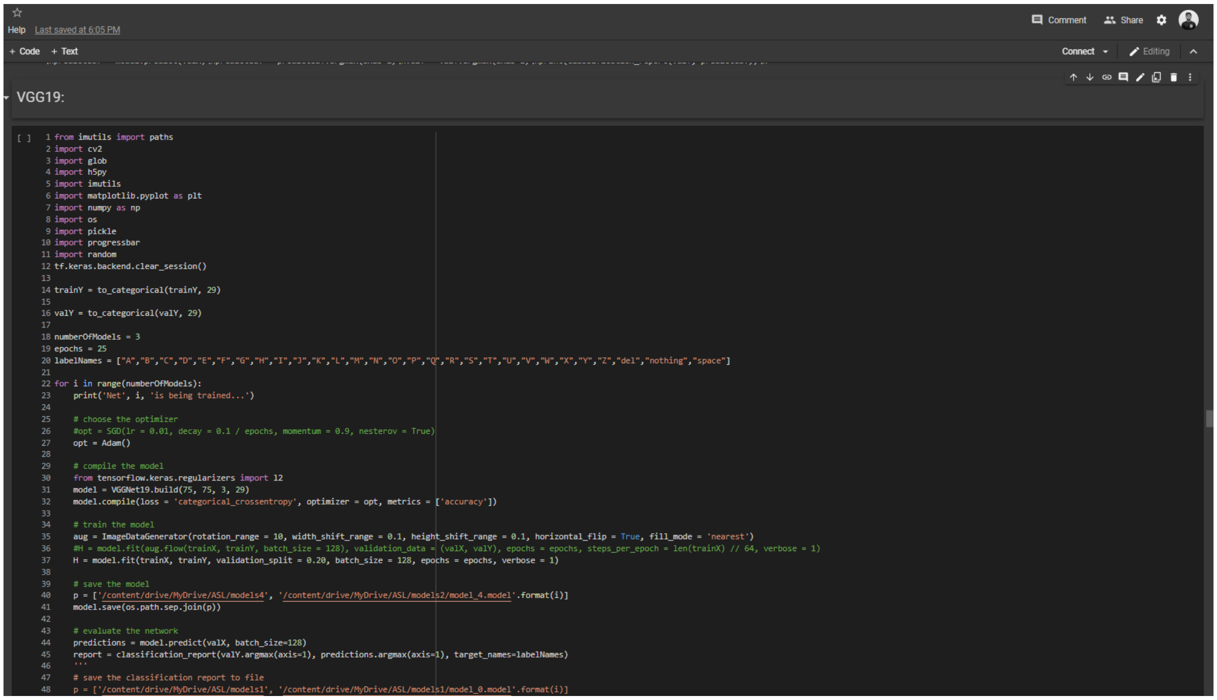
Snapshot Ensemble:

```python
tf.keras.backend.clear_session()
from tensorflow.keras.callbacks import LearningRateScheduler
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.regularizers import l1
from tensorflow.keras.regularizers import l2
models = 3
initialLearningRate = 0.2
epochs = 25

trainY = to_categorical(trainY, 29)
valY = to_categorical(valY, 29)

labelNames = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O","P","Q","R","S","T","U","V","W","X","Y","Z","del","nothing","space"]

# code for a learning rate scheduler
def shiftedCosineLearning(epoch):
    maxEpochs = epochs
    baseLearningRate = initialLearningRate

    alpha = (initialLearningRate/2)*(np.cos(np.pi*np.mod(epoch - 1, np.ceil(epochs/models))/np.ceil(epochs/models)) + 1)

    # return the learning rate
    return alpha

callbacks = [LearningRateScheduler(shiftedCosineLearning)]

# choose the optimizer
opt = SGD(lr = initialLearningRate)
#pt = Adam()

# compile the model
model = AlexNet.build(75, 75, 3, 29)
model.compile(loss = 'categorical_crossentropy', optimizer = opt, metrics = ['accuracy'])

# train the model
checkpoint = ModelCheckpoint('model{epoch:08d}.model', period=np.ceil(epochs/models))
#aug = ImageDataGenerator(rotation_range = 10, width_shift_range = 0.1, height_shift_range = 0.1, horizontal_flip = True, fill_mode = 'nearest')
#H = model.fit(aug.flow(trainX, trainY, batch_size = 64), validation_data = (valX, valY), epochs = epochs,
            #callbacks = callbacks, steps_per_epoch = len(trainX) // 64, verbose = 1)
H = model.fit(trainX, trainY, validation_data = (valX,valY), batch_size = 128, epochs = epochs, verbose = 1)

# evaluate the network
predictions = model.predict(valX, batch_size=128)
report = classification_report(valY.argmax(axis=1), predictions.argmax(axis=1), target_names=labelNames)

valY = valY.argmax(axis=1)
trainY = trainY.argmax(axis=1)
```

Figure 31 Snapshot Ensemble of nets output of AlexNet.

8.4. VGG19:

The code of my ensemble in VGG19 can be shown as follows:

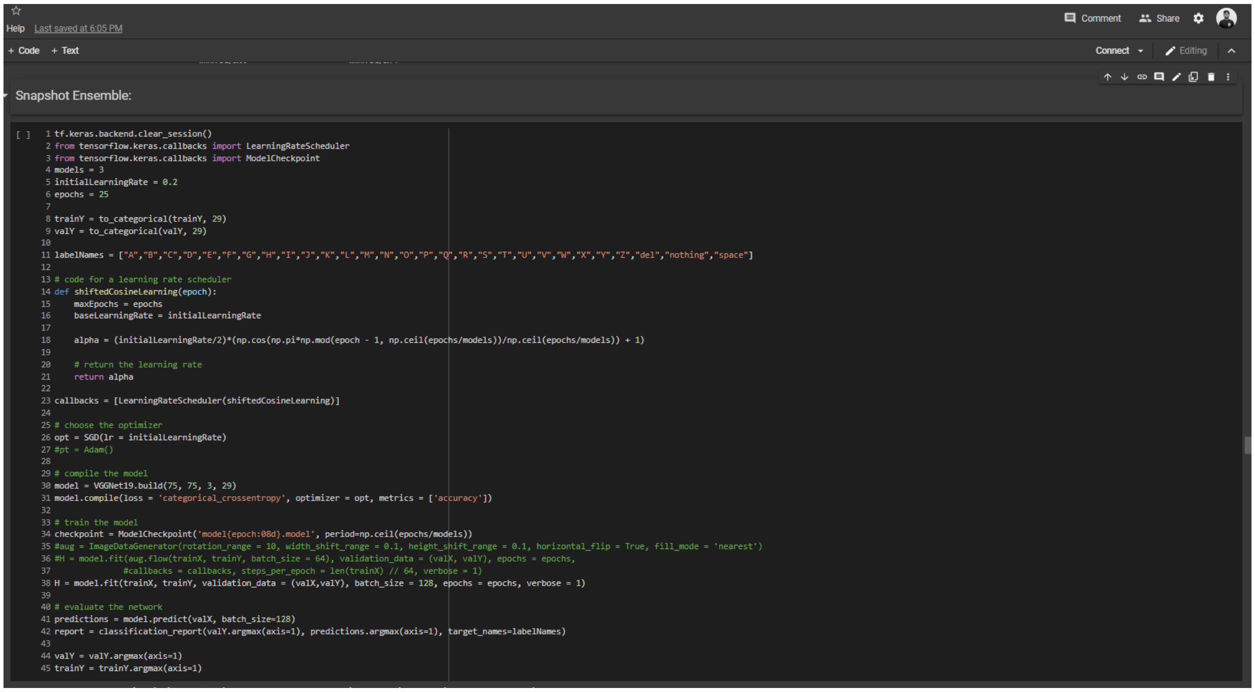


VGG19:

```python
from imutils import paths
import cv2
import glob
import h5py
import imutils
import matplotlib.pyplot as plt
import numpy as np
import os
import pickle
import progressbar
import random
tf.keras.backend.clear_session()

trainY = to_categorical(trainY, 29)

valY = to_categorical(valY, 29)

numberOfModels = 3
epochs = 25
labelNames = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O","P","Q","R","S","T","U","V","W","X","Y","Z","del","nothing","space"]

for i in range(numberOfModels):
    print('Net', i, 'is being trained...')

    # choose the optimizer
    #opt = SGD(lr = 0.01, decay = 0.1 / epochs, momentum = 0.9, nesterov = True)
    opt = Adam()

    # compile the model
    from tensorflow.keras.regularizers import l2
    model = VGGNet19.build(75, 75, 3, 29)
    model.compile(loss = 'categorical_crossentropy', optimizer = opt, metrics = ['accuracy'])

    # train the model
    aug = ImageDataGenerator(rotation_range = 10, width_shift_range = 0.1, height_shift_range = 0.1, horizontal_flip = True, fill_mode = 'nearest')
    #H = model.fit(aug.flow(trainX, trainY, batch_size = 128), validation_data = (valX, valY), epochs = epochs, steps_per_epoch = len(trainX) // 64, verbose = 1)
    H = model.fit(trainX, trainY, validation_split = 0.20, batch_size = 128, epochs = epochs, verbose = 1)

    # save the model
    p = ['/content/drive/MyDrive/ASL/models4', '/content/drive/MyDrive/ASL/models2/model_4.model'.format(i)]
    model.save(os.path.sep.join(p))

    # evaluate the network
    predictions = model.predict(valX, batch_size=128)
    report = classification_report(valY.argmax(axis=1), predictions.argmax(axis=1), target_names=labelNames)
    '''
    # save the classification report to file
    p = ['/content/drive/MyDrive/ASL/models1', '/content/drive/MyDrive/ASL/models1/model_0.model'.format(i)]
```

Figure 32 Ensemble of nets output of VGG19.

To test a different type of ensemble, a snapshot ensemble was used to train my model:



Snapshot Ensemble:

```python
tf.keras.backend.clear_session()
from tensorflow.keras.callbacks import LearningRateScheduler
from tensorflow.keras.callbacks import ModelCheckpoint
models = 3
initialLearningRate = 0.2
epochs = 25

trainY = to_categorical(trainY, 29)
valY = to_categorical(valY, 29)

labelNames = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O","P","Q","R","S","T","U","V","W","X","Y","Z","del","nothing","space"]

# code for a learning rate scheduler
def shiftedCosineLearning(epoch):
    maxEpochs = epochs
    baseLearningRate = initialLearningRate

    alpha = (initialLearningRate/2)*(np.cos(np.pi*np.mod(epoch - 1, np.ceil(epochs/models))/np.ceil(epochs/models)) + 1)

    # return the learning rate
    return alpha

callbacks = [LearningRateScheduler(shiftedCosineLearning)]

# choose the optimizer
opt = SGD(lr = initialLearningRate)
#pt = Adam()

# compile the model
model = VGGNet19.build(75, 75, 3, 29)
model.compile(loss = 'categorical_crossentropy', optimizer = opt, metrics = ['accuracy'])

# train the model
checkpoint = ModelCheckpoint('model{epoch:08d}.model', period=np.ceil(epochs/models))
#aug = ImageDataGenerator(rotation_range = 10, width_shift_range = 0.1, height_shift_range = 0.1, horizontal_flip = True, fill_mode = 'nearest')
#H = model.fit(aug.flow(trainX, trainY, batch_size = 64), validation_data = (valX, valY), epochs = epochs,
            #callbacks = callbacks, steps_per_epoch = len(trainX) // 64, verbose = 1)
H = model.fit(trainX, trainY, validation_data = (valX,valY), batch_size = 128, epochs = epochs, verbose = 1)

# evaluate the network
predictions = model.predict(valX, batch_size=128)
report = classification_report(valY.argmax(axis=1), predictions.argmax(axis=1), target_names=labelNames)

valY = valY.argmax(axis=1)
trainY = trainY.argmax(axis=1)
```

Figure 33 Snapshot Ensemble of nets output of VGG19.

The accuracy values obtained can be shown in the table below:

Table 3 Comparison between Ensemble of Nets and Snapshot Ensemble.

Network Architecture	Ensemble Of Nets	Snapshot Ensemble
MiniVGG	0.74	0.94
VGG16	0.92	0.84
AlexNet	0.77	0.41
VGG19	0.97	0.67

9. Activation Function

The activation function used in this project are as follows:

Table 4 Activation Function and their accuracy.

Activation Function	Accuracy
Sigmoid	0.49
ReLU	0.79
tanh	0.74
Softmax	0.03

Amongst these, ReLU showed better accuracy than the other activation functions.

10. Loss Function

The Loss functions used in this project are as follows:

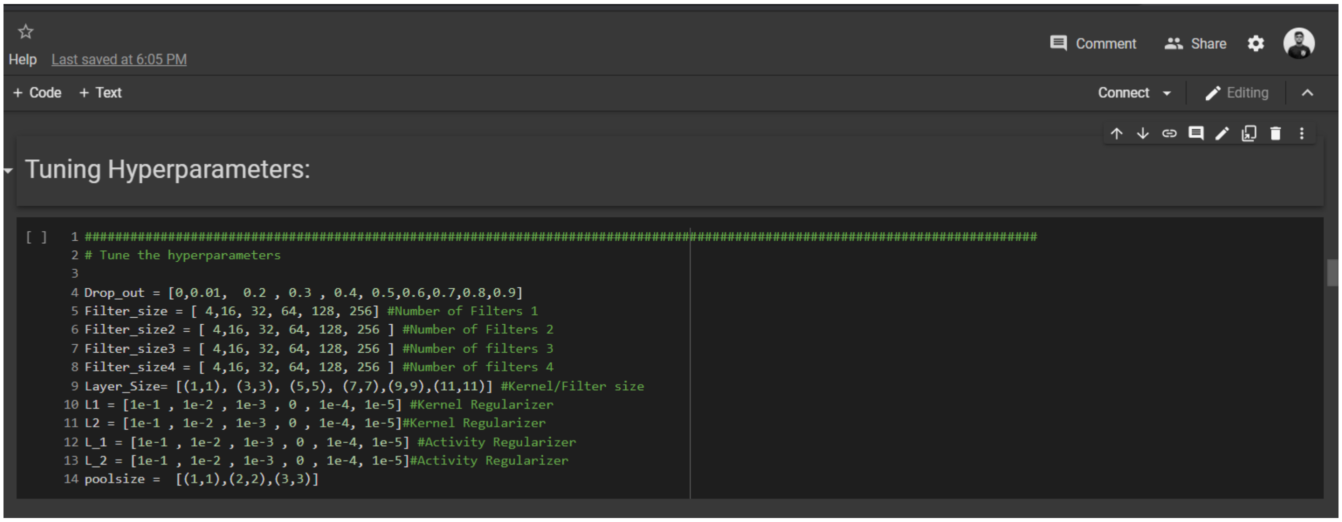
Table 5 Loss Function and their accuracy.

Loss Function	Accuracy
Mean_squared_Error	0.80
Categorical_crossentropy	0.89
Binary_categorical_crossentropy	0.87

Amongst these, category_crossentropy was selected as it gave a high accuracy compared to others. According to an online source, category_crossentropy is an ideal choice for multiclass image classification problems.

11. Tune hyperparameters

After selecting the right network architecture, activation, and loss function, the hyperparameters were tuned. To tune the hyperparameters, specific values for dropout, kernel size, layer size, and L1/L2 were given to choose the best one out of these values depending on the accuracy they printed. The values used are as shown below:




```
##########################################################################################################################
# Tune the hyperparameters

Drop_out = [0,0.01,  0.2 , 0.3 , 0.4, 0.5,0.6,0.7,0.8,0.9]
Filter_size = [ 4,16, 32, 64, 128, 256] #Number of Filters 1
Filter_size2 = [ 4,16, 32, 64, 128, 256 ] #Number of Filters 2
Filter_size3 = [ 4,16, 32, 64, 128, 256 ] #Number of filters 3
Filter_size4 = [ 4,16, 32, 64, 128, 256 ] #Number of filters 4
Layer_Size= [(1,1), (3,3), (5,5), (7,7),(9,9),(11,11)] #Kernel/Filter size
L1 = [1e-1 , 1e-2 , 1e-3 , 0 , 1e-4, 1e-5] #Kernel Regularizer
L2 = [1e-1 , 1e-2 , 1e-3 , 0 , 1e-4, 1e-5]#Kernel Regularizer
L_1 = [1e-1 , 1e-2 , 1e-3 , 0 , 1e-4, 1e-5] #Activity Regularizer
L_2 = [1e-1 , 1e-2 , 1e-3 , 0 , 1e-4, 1e-5]#Activity Regularizer
poolsize =  [(1,1),(2,2),(3,3)]
```

Figure 34 Tuning Hyperparameter.

As seen in the above figure, to select a value for dropout, I have initialized the 'Drop_out' variable with some values, i.e., from 0 to 0.9. Based on these values, a for loop was implemented on my network architecture, and based on the highest test accuracy obtained, the value corresponding to that accuracy was selected. The chosen values and their respected accuracy can be shown in table 8.

12. Optimizer

The optimizers used in this project can be shown as follows:

Table 6 Optimizer and their accuracy.

Optimizer	Accuracy
Adam	0.67
SGD (0.01)	0.65
SGD (0.001)	0.11
Adagrad	0.04
Adadelta	0.04
Rmsprop	0.92
Adamax	0.35
Nadam	0.65

Amongst these, Rmsprop gave the highest accuracy. Although it did give a good accuracy, Adam was used as Rmsprop for some reason gave unstable values. Now, after tuning all the hyperparameters, a final code was run to check the accuracy.

13. Kernel Weight Initialization Method:

The kernel weight initialization methods used in this project are as follows:

Table 7 Kernel Initializer and their accuracy.

Kernel Initializer	Accuracy
Glorot_Uniform	0.78
Random_normal	0.61
Random_uniform	0.78
Truncated_Normal	0.83
Zeros	0.03
Ones	0.05
Glorot_normal	0.89
Constant	0.03

Amongst these, glorot_normal gave the highest accuracy and hence it was used to tune my final model.

14. Final Code

Using the above values, and with the activation, network architecture, weight initialization methods and tuned hyperparameters, a final code is drafted to check the accuracy of the test, train, and validation. Before running the final code, more datapoints were used. A total of 70,000 images from the dataset were used as shown in the figure below:

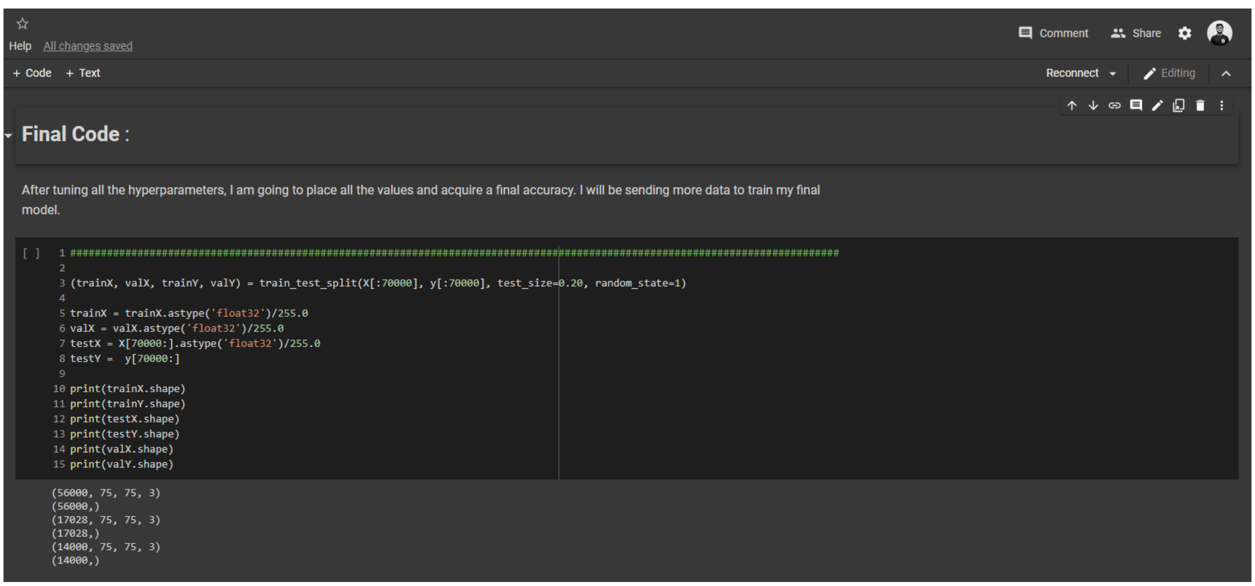


Figure 35 70000 dataset values sent for training final model.

As seen in the above figure, 56000 dataset values were used for training,17028 were used for test set and 14000 datasets was used for validation. The accuracy obtained can be shown in the following figure:

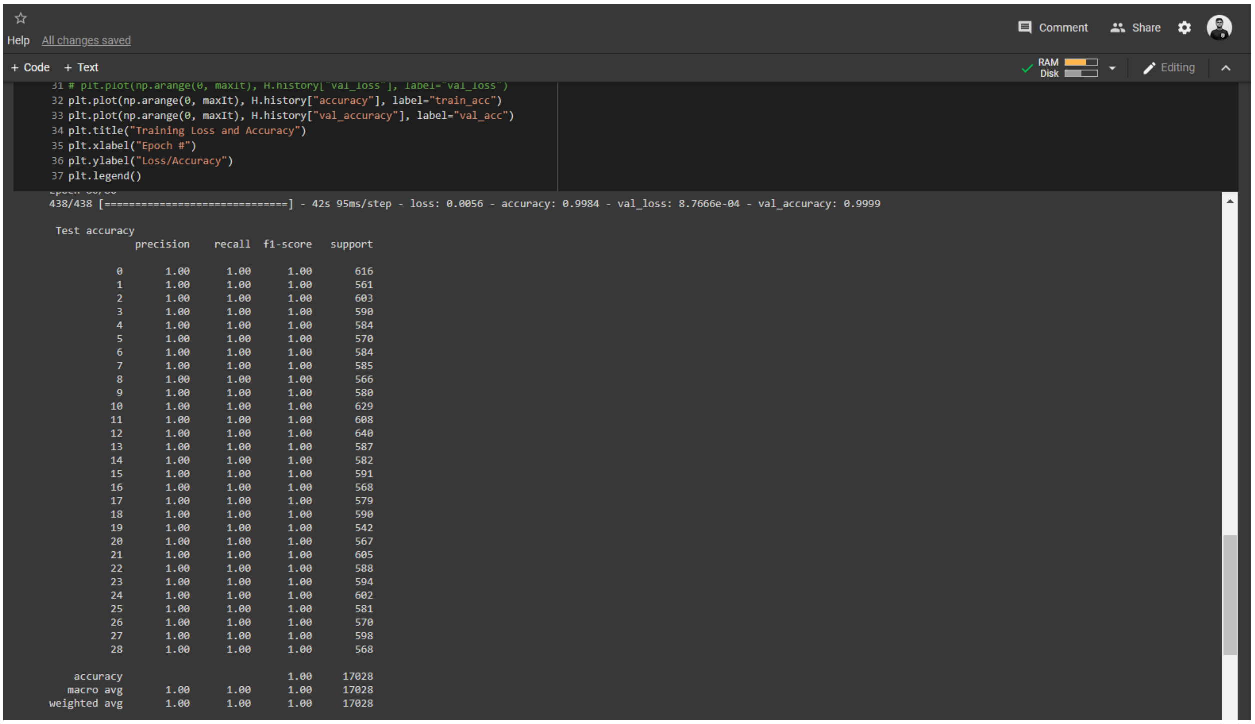


Figure 36 Final Accuracy obtained after training model

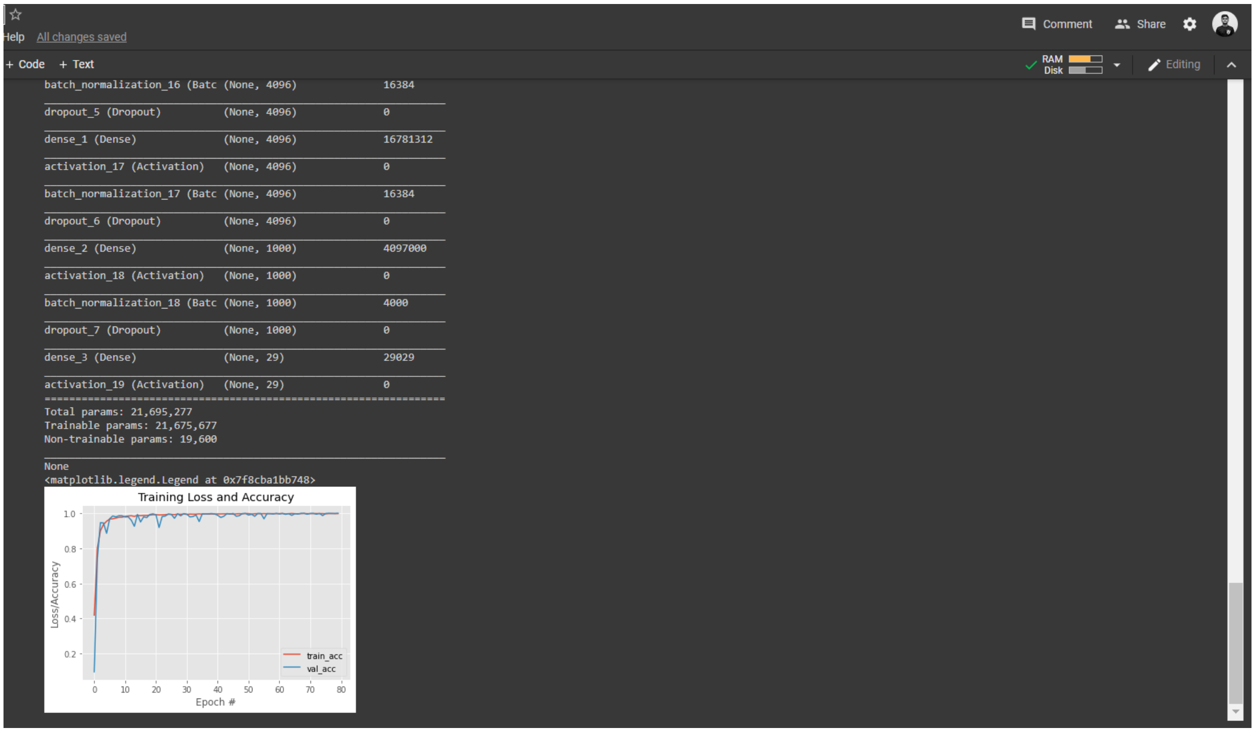


Figure 37 Graph of Final model obtained after training.

As seen in the above figure, the accuracy obtained for the test, train, and validation are:

- Train set – 0.98
- Test set – 1.00
- Validation Set – 0.99

15. Discussions

As seen above, the accuracy values touched 99 percent. The best values selected for tuning the hyperparameters can be shown in the following table.

Table 8 Best values obtained on tuning the hyperparameters.

Hyperparameters	Value Selected	Accuracy
Dropout	0.2	0.93
Filter_size (Kernel Size)	32	0.92
Filter_size2(Kernel Size)	16	0.90
Filter_size3(Kernel Size)	64	0.90
Filter_size4(Kernel Size)	32	0.85
Layer Size	(3,3)	0.86
L1 & L2 (Kernel Regularizer)	L1 & L2 = 0	0.83 & 0.79
L_1 & L_2 (Activity Regularizer)	L1 & L2 = $1 * e^{-5}$	0.60 & 0.78
Pool size	(2,2)	0.86