Department of Electronics Engineering

EXPLORATORY PROJECT REPORT

Design a deep learning model to detect Sign Languages

Submitted By:

Vikrant (20095123)

Nelson Rahul Tigga (20095072)

Raghav Agrawal (20095140)

Abhinav Raj (20095001)

Mentored By:

Prof. Vishwambhar Nath Mishra

DEPARTMENT OF ELECTRONICS ENGINEERING

IIT (BHU) VARANASI

CERTIFICATE

This is to certify that this report **“Sing Language Detector”** is submitted by Vikrant, Nelson Rahul Tigga, Raghav Agrawal and Abhinav Raj who carried out the project work under the supervision of **Prof. Vishwambhar Nath Mishra**.

We approve this project for submission of the Exploratory Project, IIT (BHU) Varanasi.

Sign of Supervisor

Prof. Vishwambhar Nath Mishra

Department of Electronics Engineering

IIT (BHU) Varanasi

ACKNOWLEDGEMENT

I would like to acknowledge my indebtedness and it gives us immense pleasure to express our deepest sense of gratitude and sincere thanks to our highly respected and esteemed guide **Prof. Vishwambhar Nath Mishra** for his valuable guidance, encouragement, and help for accomplishing this work. His useful suggestions for this whole projectare acknowledged. We would also like to express our sincere thanks to all others who helped us directly or indirectly during this project work

STUDENT NAME:	DATE:
Vikrant (20095123)	05/05/2022
Nelson Rahul Tigga (20095072)	
Raghav Agrawal (20095140)	
Abhinav Raj (20095001)	

ABSTRACT

Sign language is one of the oldest and most natural form of language for communication, but since most people do not know sign language and interpreters are very difficult to come by we have come up with a real time method using neural networks for fingerspelling based American sign language. In our method, the hand is first passed through a filter and after the filter is applied the hand is passed through a classifier which predicts the class of the hand gestures. Our method provides 95.7 % accuracy for the 26 letters of the alphabet.

INDEX

INTRODUCTION

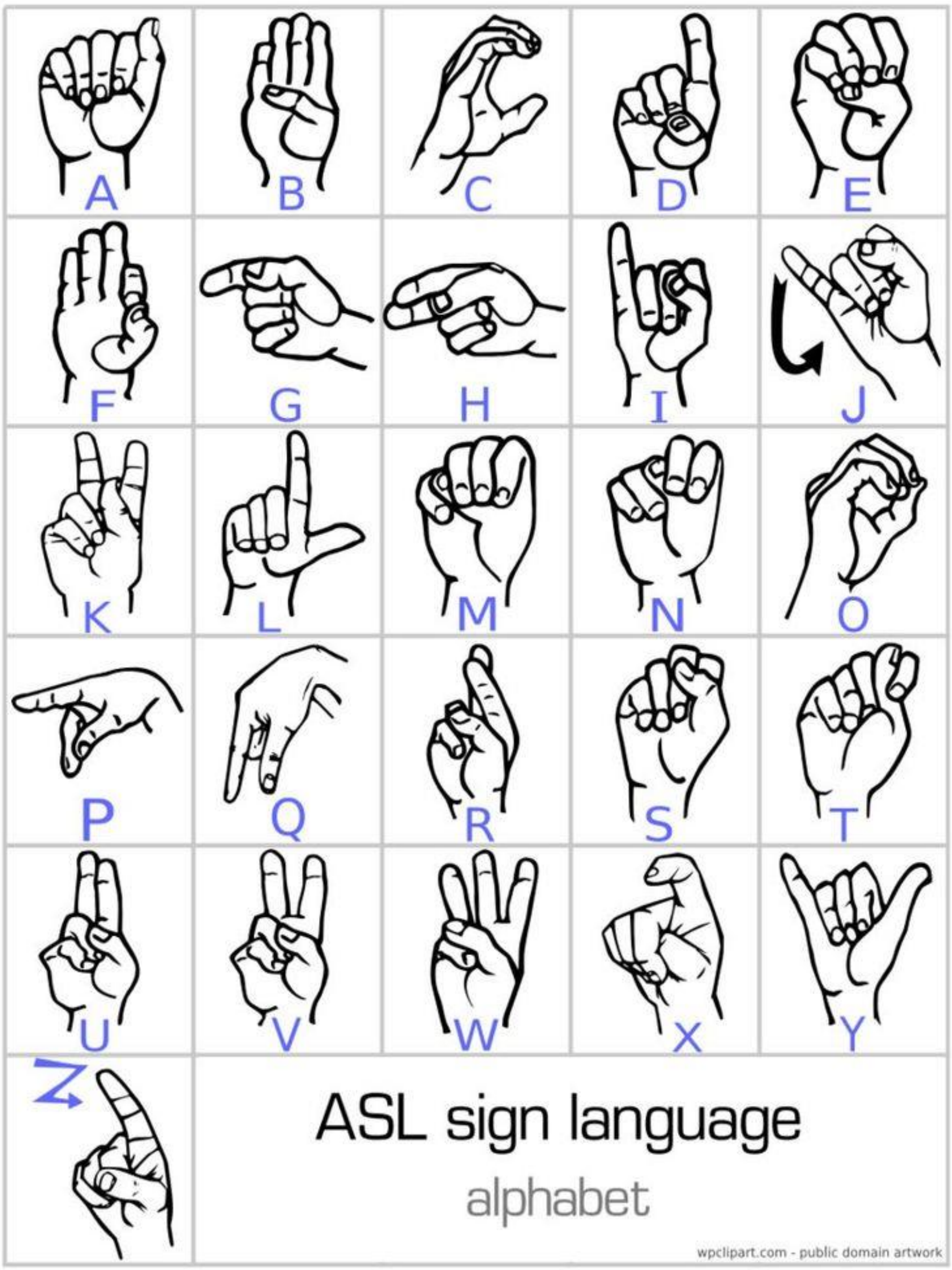
American Sign Language (ASL) is a complete, natural language that is expressed using the movement of hands and face. ASL provides the deaf community a way to interact within the community itself as well as to the outside world. However, not everyone knows about signs and gestures used in the sign language.

With the advent of Artificial Neural Networks and Deep Learning, it is now possible to build a system that can recognize objects or even objects of various categories (like red vs green apple). Utilizing this, here we have an application that uses a deep learning model trained on the ASL Dataset to predict the sign from the sign language given an input image or frame from a video feed.

Sign language is visual language and consists of three major components

Fingerspelling	**Word level sign language**	**Non manual features**
Used to spell words letter by letter.	Used for the majority of communication.	Facial expressions, tongue, mouth and body position.

Alphabet signs in American Sign Language are shown below:



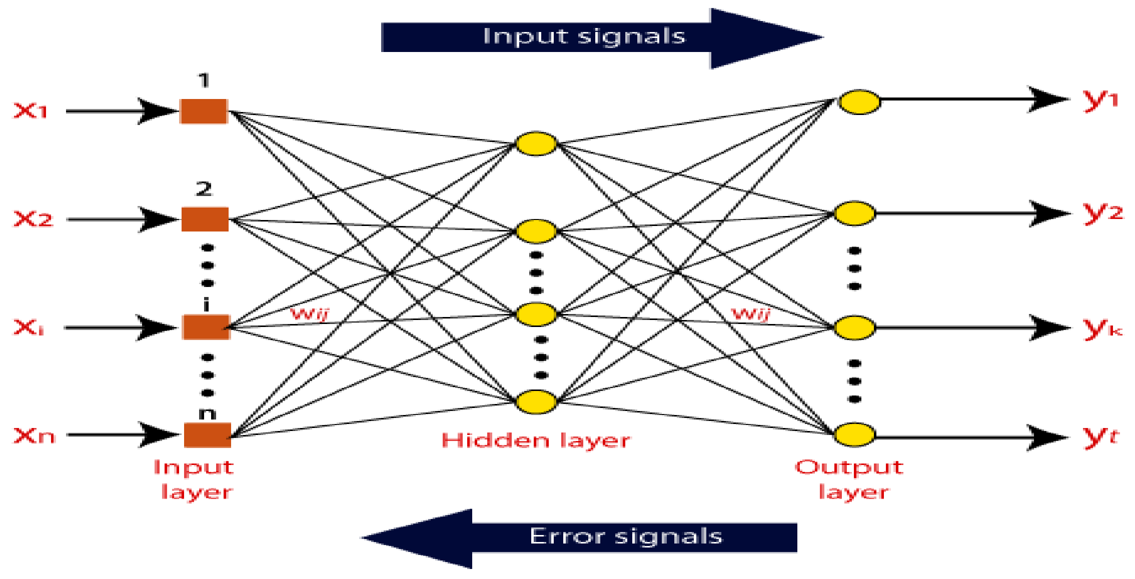
KEYWORDS AND DEFINITIONS

1. Feature Extraction and Representation

The representation of an image as a 3D matrix having dimension as of height and width of the image and the value of each pixel as depth (1 in case of Grayscale and 3 in case of RGB). Further, these pixel values are used for extracting useful features using CNN.

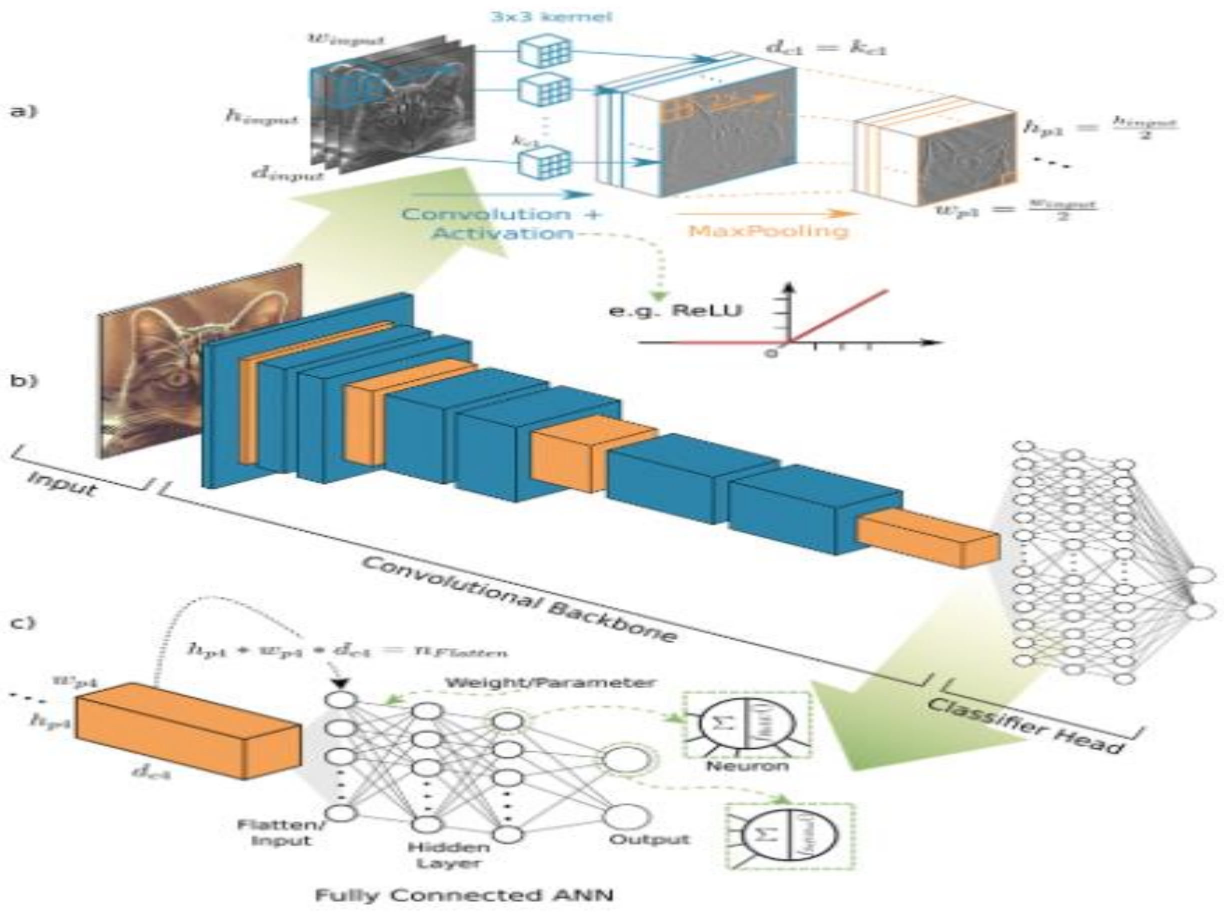
2. Artificial Neural Network

Artificial Neural Network is a connection of neurons, replicating the structure of human brain. Each connection of neuron transfers information to another neuron. Inputs are fed into first layer of neurons which processes it and transfers to another layer of neurons called as hidden layers. After processing of information through multiple layers of hidden layers, information is passed to final output layer. They are capable of learning and they have to be trained.



3. Convolution Neural Network

Unlike regular Neural Networks, in the layers of CNN, the neurons are arranged in 3 dimensions: width, height, depth. The neurons in a layer will only be connected to a small region of the layer (window size) before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would have dimensions (number of classes), because by the end of the CNN architecture we will reduce the full image into a single vector of class scores.



4. Convolutional Layer

In convolutional layer we take a small window size [typically of length 5*5] that extends to the depth of the input matrix. The layer consists of learnable filters of window size. During every iteration we slid the window by stride size [typically 1], and compute the dot product of filter entries and input values at a given position. As we continue this process well create a 2-Dimensional activation matrix that gives the response of that matrix at every spatial position.

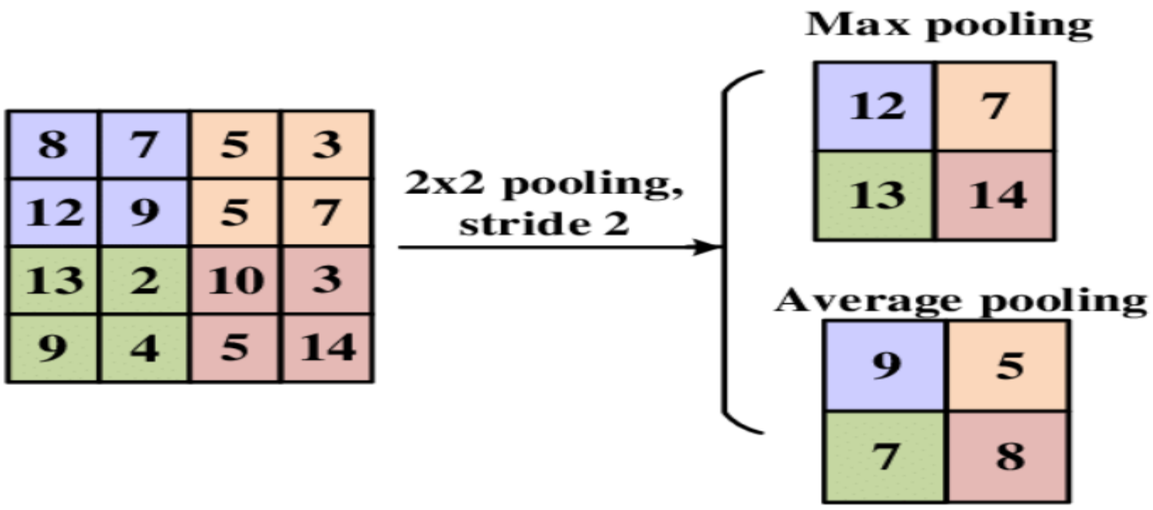
That is, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color.

5. Pooling Layer

We use pooling layer to decrease the size of activation matrix and ultimately reduce the learnable parameters. There are two types of pooling layers:

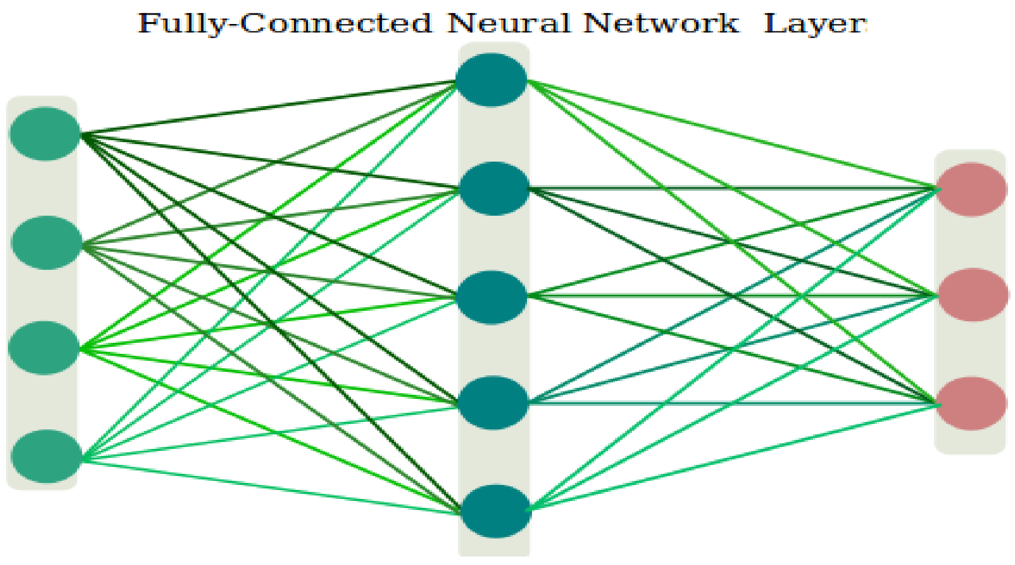
a) Max Pooling: In max pooling we take a window size [for example window of size 2*2], and only take the maximum of 4 values. Well lid this window and continue this process, so well finally get a activation matrix half of its original Size.

b) Average Pooling: In average pooling we take average of all value in a window



6. Fully Connected Layer

In convolution layer neurons are connected only to a local region, while in a fully connected region, well connect the all the inputs to neurons



7. Final Output Layer

After getting values from fully connected layer, we'll connect them to final layer of neurons having count equal to total number of classes, that will predict the probability of each image to be in different classes.

8. Tensor Flow

Tensor flow is an open source software library for numerical computation. First we define the nodes of the computation graph, then inside a session, the actual computation takes place. Tensor Flow is widely used in Machine Learning.

9. Keras

Keras is a high-level neural networks library written in python that works as a wrapper to Tensor Flow. It contains implementations of commonly used neural network elements like layers, objective, activation functions, optimizers, and tools to make working with images and text data easier.

10. OpenCV

OpenCV(Open Source Computer Vision) is an open source library of programming functions used for real-time computer-vision. It is mainly used for image processing, video capture and analysis for features like face and object recognition. It is written in C++ which is its primary interface, however bindings are available for Python, Java, MATLAB/OCTAVE.

APPROACH TO BULID THE MODEL

We have utilized a method called **Data Augmentation** and **Transfer Learning** to create a deep learning model for the American Sign Language Detection.

Data Augmentation

Data Augmentation is simple technique that is widely used to increase the diversity of the training data set.

The model was on the Kaggle dataset of ASL alphabet. The dataset contains *87,000* images which are 200x200 pixels, divided into *29* classes (*26* English Alphabets and *3* additional signs of SPACE, DELETE and NOTHING).

In order to train the model for better real-world scenarios, we have augmented the data using brightness shift (ranging in *20%* darker lighting conditions) and zoom shift (zooming out up to *120%*).

Transfer Learning

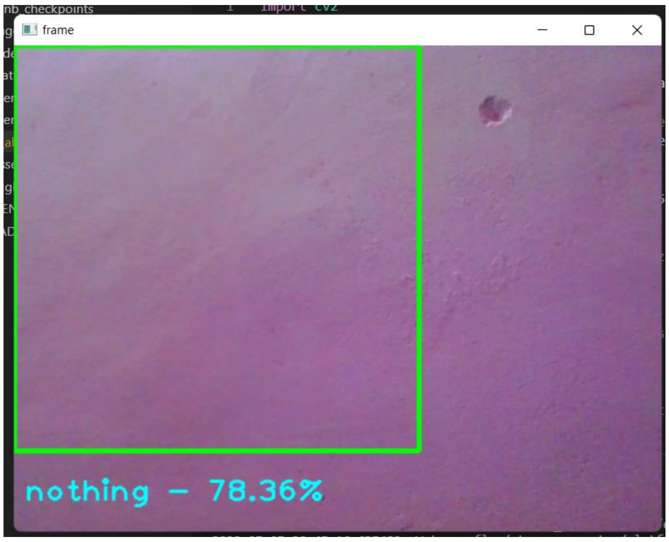
Transfer learning is a machine learning technique that uses a pre-trained model on some other same type of task.

Some transfer learning models that are used with image data are:
i) Oxford VGG Model
ii) Google Inception Model
iii) Microsoft ResNet Model

Our model's architecture uses Google Inception v3 as the base model. The first 248 out of 311 layers of the model (i.e. up to the third last inception block) are locked, leaving only the last 2 inception blocks for training. We've also removed the fully connected layers at the top of Inception network and added our own set of fully connected layers after the inception network so as to conform the neural network for our application. We've added 2 fully connected layers, one consisting of 1024 ReLU units and the other of 29 Softmax units for the prediction of 29 classes. The model is then trained on the set of new images for the ASL Application.

Model Training

- For training, Categorical Crossentropy was used to measure the loss along with Stochastic Gradient Descent optimizer (with learning rate of 0.0001 and momentum of 0.9) to optimize our model. The model is trained for 24 epochs.
- After the model is trained, it is then loaded in the application. OpenCV is used to capture frames from a video feed. The application provides an area (inside the green rectangle) where the signs are to be presented to be detected or recognized.



Model Testing

- The signs are captured in frames using OpenCV, the frame is processed for the model and then fed to the model. Based on the sign made, the model predicts the sign captured.
- If the model predicts a sign with a confidence greater than `20%`, the prediction is presented to the user (`LOW` confidence sign predictions are predictions above `20%` to `50%` confidence which are presented with a `Maybe [sign] - [confidence]` output and `HIGH` confidence sign predictions are above `50%` confidence and presented with a `[sign] - [confidence]` output where `[sign]` is the model predicted sign and `[confidence]` is the model's confidence for that prediction). Else, the model displays `nothing` as output.

Code to train the model

Importing tensorflow and checking tensorflow:

```
import tensorflow as tf

print(tf.__version__)
```

Installing Kaggle so as to download the dataset using Kaggle API:

```
!pip install -q kaggle
```

Setting up the kaggle.json authentication file enabling us to download the dataset:

```
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
```

Downloading the dataset using the API:

```
!kaggle datasets download -d grassknoted/asl-alphabet
```

Extracting the contents:

```
!unzip asl-alphabet.zip
```

Specifying train and test directories:

```
# Specifying the training and test directories

TRAINING_DIR = './asl_alphabet_train/asl_alphabet_train/'
TEST_DIR = './asl_alphabet_test/asl_alphabet_test/'
```

Looking at some random images from the dataset:

```
# Printing 5 random images from any training category or from a specified
category
%matplotlib inline

import cv2
import os
import random
import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

number_of_rows = 1
number_of_columns = 5

categories = os.listdir(TRAINING_DIR)

random.seed(13)

category = categories[random.randint(1, 30)]
# category = 'A'

for i in range(number_of_columns):
  subplot = plt.subplot(number_of_rows, number_of_columns, i + 1)
  subplot.axis('Off')
  subplot.set_title(category)
  image_path = os.path.join(
      TRAINING_DIR,
      str(category),
      str(category) + str(random.randint(1, 1000)) + '.jpg'
  )
  image = mpimg.imread(image_path)
  plt.imshow(image)

plt.show()
```

Augmenting the data with brightness and zoom ranges:

```
# Preparing ImageDataGenerator object for training the model
from tensorflow.keras.preprocessing.image import ImageDataGenerator

IMAGE_SIZE = 200
BATCH_SIZE = 64

data_generator = ImageDataGenerator(
    samplewise_center=True,
    samplewise_std_normalization=True,
    brightness_range=[0.8, 1.0],
    zoom_range=[1.0, 1.2],
    validation_split=0.1
)

train_generator = data_generator.flow_from_directory(TRAINING_DIR,
target_size=(IMAGE_SIZE, IMAGE_SIZE), shuffle=True, seed=13,
                                                     class_mode='categorical',
batch_size=BATCH_SIZE, subset="training")

validation_generator = data_generator.flow_from_directory(TRAINING_DIR,
target_size=(IMAGE_SIZE, IMAGE_SIZE), shuffle=True, seed=13,
                                                     class_mode='categorical',
batch_size=BATCH_SIZE, subset="validation")
```

Downloading custom weight file if required:

```
!wget --no-check-certificate \
    https://storage.googleapis.com/mledu-
datasets/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5 \
    -O /content/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5
```

Preparing Inception V3 Network for transfer learning:

```
# Loading inception v3 network for transfer learning
from tensorflow.keras import layers
from tensorflow.keras import Model

from tensorflow.keras.applications.inception_v3 import InceptionV3

WEIGHTS_FILE = './inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5'

inception_v3_model = InceptionV3(
    input_shape = (IMAGE_SIZE, IMAGE_SIZE, 3),
    include_top = False,
    weights = 'imagenet'
)

# Not required --> inception_v3_model.load_weights(WEIGHTS_FILE)

# Enabling the top 2 inception blocks to train
for layer in model.layers[:249]:
    layer.trainable = False
for layer in model.layers[249:]:
    layer.trainable = True

# Checking model summary to pick a layer (if required)
inception_v3_model.summary()
```

Choosing the inception output layer:

```
# Choosing the output layer to be merged with our FC layers (if required)
inception_output_layer = inception_v3_model.get_layer('mixed7')
print('Inception model output shape:', inception_output_layer.output_shape)

# Not required --> inception_output = inception_output_layer.output
inception_output = inception_v3_model.output
```

Adding our own set of fully connected layers at the end of Inception V3 Network:

```
from tensorflow.keras.optimizers import RMSprop, Adam, SGD

x = layers.GlobalAveragePooling2D()(inception_output)
x = layers.Dense(1024, activation='relu')(x)
# Not required --> x = layers.Dropout(0.2)(x)
x = layers.Dense(29, activation='softmax')(x)

model = Model(inception_v3_model.input, x)

model.compile(
    optimizer=SGD(lr=0.0001, momentum=0.9),
    loss='categorical_crossentropy',
    metrics=['acc']
)
```

Model summary:

```
# Watch the new model summary
model.summary()
```

Setting up a callback function in order to stop training at a particular threshold:

```
# Creating a callback to stop model training after reaching a threshold
accuracy

LOSS_THRESHOLD = 0.2
ACCURACY_THRESHOLD = 0.95

class ModelCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs={}):
    if logs.get('val_loss') <= LOSS_THRESHOLD and logs.get('val_acc') >=
ACCURACY_THRESHOLD:
      print("\nReached", ACCURACY_THRESHOLD * 100, "accuracy, Stopping!")
      self.model.stop_training = True

callback = ModelCallback()
```

Training the model:

```
history = model.fit_generator(
    train_generator,
    validation_data=validation_generator,
    steps_per_epoch=100,
    validation_steps=50,
    epochs=50,
    callbacks=[callback]
)
```

Training Accuracy vs Validation Accuracy:

```
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'r', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend(loc=0)
plt.figure()


plt.show()
```

Training Loss vs Validation Loss:

```
plt.plot(epochs, loss, 'r', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend(loc=0)
plt.figure()
```

Saving the model:

```
# Saving the model
MODEL_NAME = 'models/asl_alphabet_{}.h5'.format(9575)
model.save(MODEL_NAME)
```

Testing the model:

```
import cv2
import numpy as np
import os
import matplotlib.pyplot as plt

classes = os.listdir(TRAINING_DIR)
classes.sort()

for i, test_image in enumerate(os.listdir(TEST_DIR)):
    image_location = TEST_DIR + test_image
    img = cv2.imread(image_location)
    img = cv2.resize(img, (IMAGE_SIZE, IMAGE_SIZE))
    plt.figure()
    plt.axis('Off')
    plt.imshow(img)
    img = np.array(img) / 255.
    img = img.reshape((1, IMAGE_SIZE, IMAGE_SIZE, 3))
    img = data_generator.standardize(img)
    prediction = np.array(model.predict(img))
    actual = test_image.split('_')[0]
    predicted = classes[prediction.argmax()]
    print('Actual class: {} \n Predicted class: {}'.format(actual, predicted))
    plt.show()
```

Calculating the test accuracy:

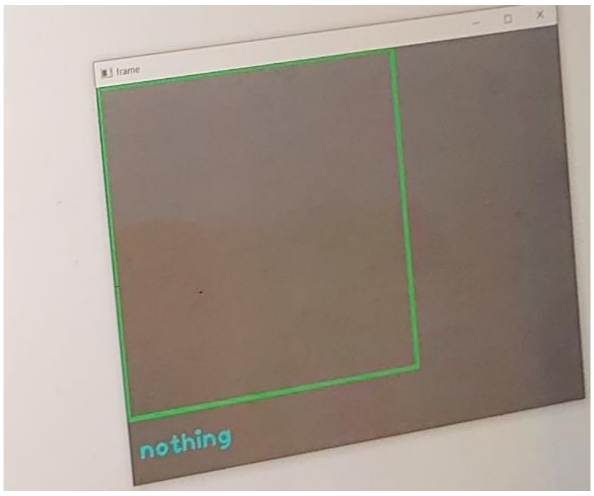
```
test_images = os.listdir(TEST_DIR)
total_test_cases = len(test_images)
total_correctly_classified = 0
total_misclassified = 0
for i, test_image in enumerate(test_images):
    image_location = TEST_DIR + test_image
    img = cv2.imread(image_location)
    img = cv2.resize(img, (IMAGE_SIZE, IMAGE_SIZE))
    img = np.array(img) / 255.
    img = img.reshape((1, IMAGE_SIZE, IMAGE_SIZE, 3))
    img = data_generator.standardize(img)
    prediction = np.array(model.predict(img))
    actual = test_image.split('_')[0]
    predicted = classes[prediction.argmax()]
    print('Actual class: {} - Predicted class: {}'.format(
        actual, predicted), end=' ')
    if actual == predicted:
      print('PASS!')
      total_correctly_classified += 1
    else:
      print('FAIL!')
      total_misclassified += 1
print("=" * 20)
test_accuracy = (total_correctly_classified / total_test_cases) * 100
test_error_rate = (total_misclassified / total_test_cases) * 100

print('Test accuracy (%):', test_accuracy)
print('Test error rate (%):', test_error_rate)
print('Number of misclassified classes:', total_misclassified)
print('Number of correctly classified classes', total_correctly_classified)
```

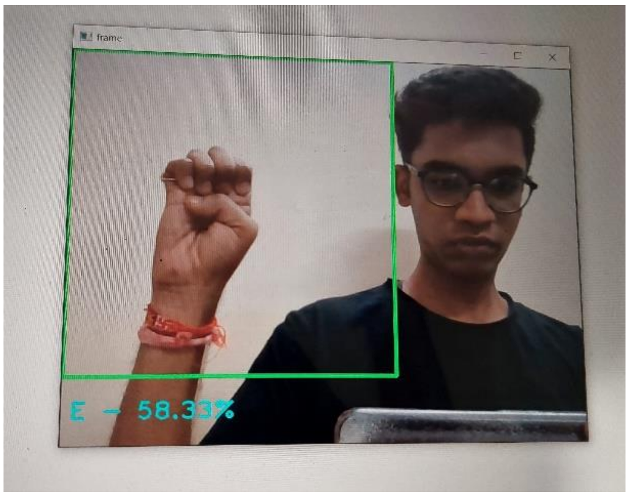
Code to use the model

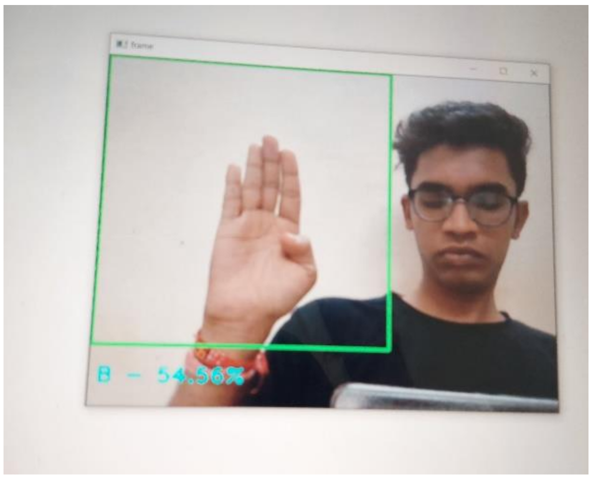
```
import cv2
import numpy as np

from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Prepare data generator for standardizing frames before sending them into the
model.
data_generator = ImageDataGenerator(samplewise_center=True,
samplewise_std_normalization=True)

# Loading the model.
MODEL_NAME = 'models/asl_alphabet_{}.h5'.format(9575)
model = load_model(MODEL_NAME)

# Setting up the input image size and frame crop size.
IMAGE_SIZE = 200
CROP_SIZE = 400

# Creating list of available classes stored in classes.txt.
classes_file = open("classes.txt")
classes_string = classes_file.readline()
classes = classes_string.split()
classes.sort()  # The predict function sends out output in sorted order.

# Preparing cv2 for webcam feed
cap = cv2.VideoCapture(0)

while(True):
    # Capture frame-by-frame.
    ret, frame = cap.read()

    # Target area where the hand gestures should be.
    cv2.rectangle(frame, (0, 0), (CROP_SIZE, CROP_SIZE), (0, 255, 0), 3)

    # Preprocessing the frame before input to the model.
    cropped_image = frame[0:CROP_SIZE, 0:CROP_SIZE]
    resized_frame = cv2.resize(cropped_image, (IMAGE_SIZE, IMAGE_SIZE))
    reshaped_frame = (np.array(resized_frame)).reshape((1, IMAGE_SIZE,
IMAGE_SIZE, 3))
    frame_for_model = data_generator.standardize(np.float64(reshaped_frame))

    # Predicting the frame.
    prediction = np.array(model.predict(frame_for_model))
    predicted_class = classes[prediction.argmax()]      # Selecting the max
confidence index.

    # Preparing output based on the model's confidence.
    prediction_probability = prediction[0, prediction.argmax()]
    if prediction_probability > 0.5:
        # High confidence.
        cv2.putText(frame, '{} - {:.2f}%'.format(predicted_class,
prediction_probability * 100),
                                    (10, 450), 1, 2, (255, 255, 0), 2,
cv2.LINE_AA)
    elif prediction_probability > 0.2 and prediction_probability <= 0.5:
        # Low confidence.
        cv2.putText(frame, 'Maybe {}... - {:.2f}%'.format(predicted_class,
prediction_probability * 100),
                                    (10, 450), 1, 2, (0, 255, 255), 2,
cv2.LINE_AA)
    else:
        # No confidence.
        cv2.putText(frame, classes[-2], (10, 450), 1, 2, (255, 255, 0), 2,
cv2.LINE_AA)

    # Display the image with prediction.
    cv2.imshow('frame', frame)

    # Press q to quit
    k = cv2.waitKey(1) & 0xFF
    if k == ord('q'):
        break

# When everything done, release the capture.
cap.release()
cv2.destroyAllWindows()
```

Results

Output Images



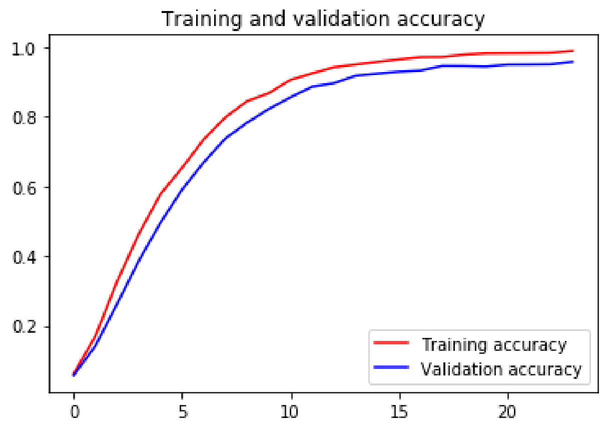


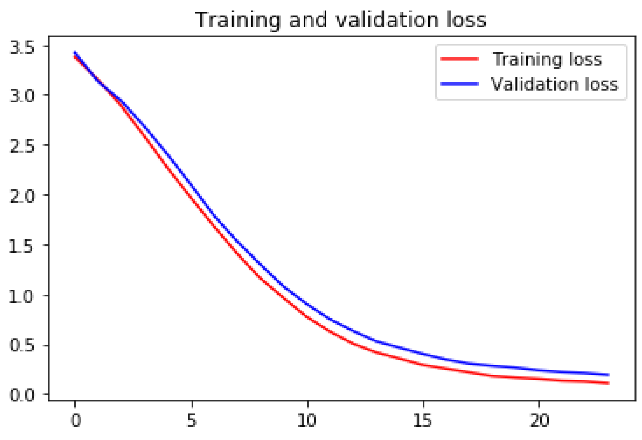


Model Accuracy

Name	Accuracy
Training Accuracy	0.9887 or 98.87%
Training Loss	0.1100
Validation Accuracy	0.9575 or 95.75%
Validation Loss	0.1926
Test Accuracy	96.43

Graphical representation





LIMITATIONS, CONCLUSION AND FUTURE SCOPE

Limitation of the model

- The model works well only in good lighting conditions.
- Plain background is needed for the model to detect with accuracy

Conclusion

In this report, a functional real time vision based American sign language recognition for D&M people have been developed for asl alphabets. We achieved final accuracy of 96.43.0% on our dataset. We are able to improve our prediction after implementing two layers of algorithms in which we verify and predict symbols which are more similar to each other. This way we are able to detect almost all the symbols provided that they are shown properly.

Future Scope

- We are planning to achieve higher accuracy even in case of complex background by trying out various background subtraction algorithms
- We will also be focused on how to get better results in low lightening conditions.

REFERENCES

[1] https://www.kaggle.com/datasets/grassknoted/asl-alphabet

[2] https://keras.io/

[3] https://iq.opengenus.org/inception-v3-model-architecture/

[4] https://machinelearningmastery.com/transfer-learning-for-deep-learning/

[5] https://opencv.org/

[6] https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53