

ASL (American Sign Language) - Alphabet Image Recognition

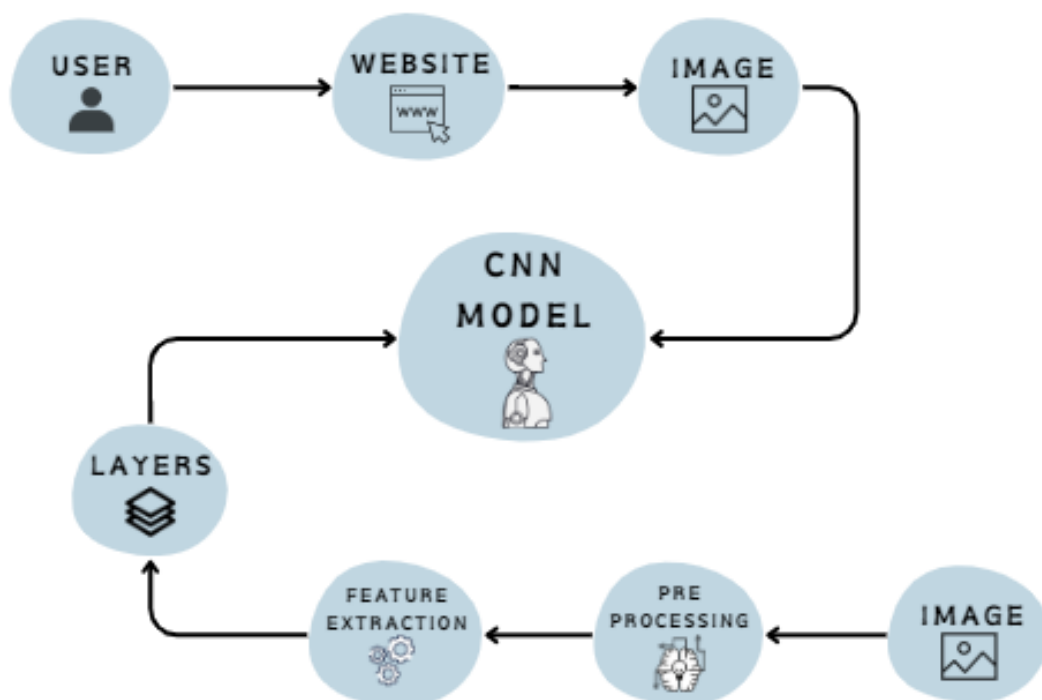
Introduction:

The American Sign Language (ASL) is the primary language used by deaf individuals in North America. It is a visual language that uses a combination of hand gestures, facial expressions, and body movements to convey meaning. In recent years, there has been an increasing interest in developing technologies to help bridge the communication gap between the deaf and hearing communities.

One such technology is ASL Alphabet Image Recognition, which is an image classification task that aims to recognize the ASL alphabet from images of hand signs. This project involves training a machine learning model to classify images of hand signs corresponding to the 26 letters of the English alphabet, as well as three additional classes for the signs for "space", "delete", and "nothing".

The trained model can be used to develop applications that can recognize the ASL alphabet from real-time video streams, which could be used to improve communication between the deaf and hearing communities.

Technical Architecture:



Prerequisites:

To complete this project, we must the following software's, concepts, and packages:

- ❖ Software's
 - Google colab
 - VS code
 - Spyder IDE.
- ❖ Deep Learning Concepts
 - CNN: A Convolutional Neural Network is a class of deep neural networks, most commonly applied to analysing visual imagery. For this project, we have used DenseNet architecture.
 - Flask: Flask is a popular Python web framework, meaning it is a third-party Python library used for developing web applications.
- ❖ Packages
 - TensorFlow
 - Keras

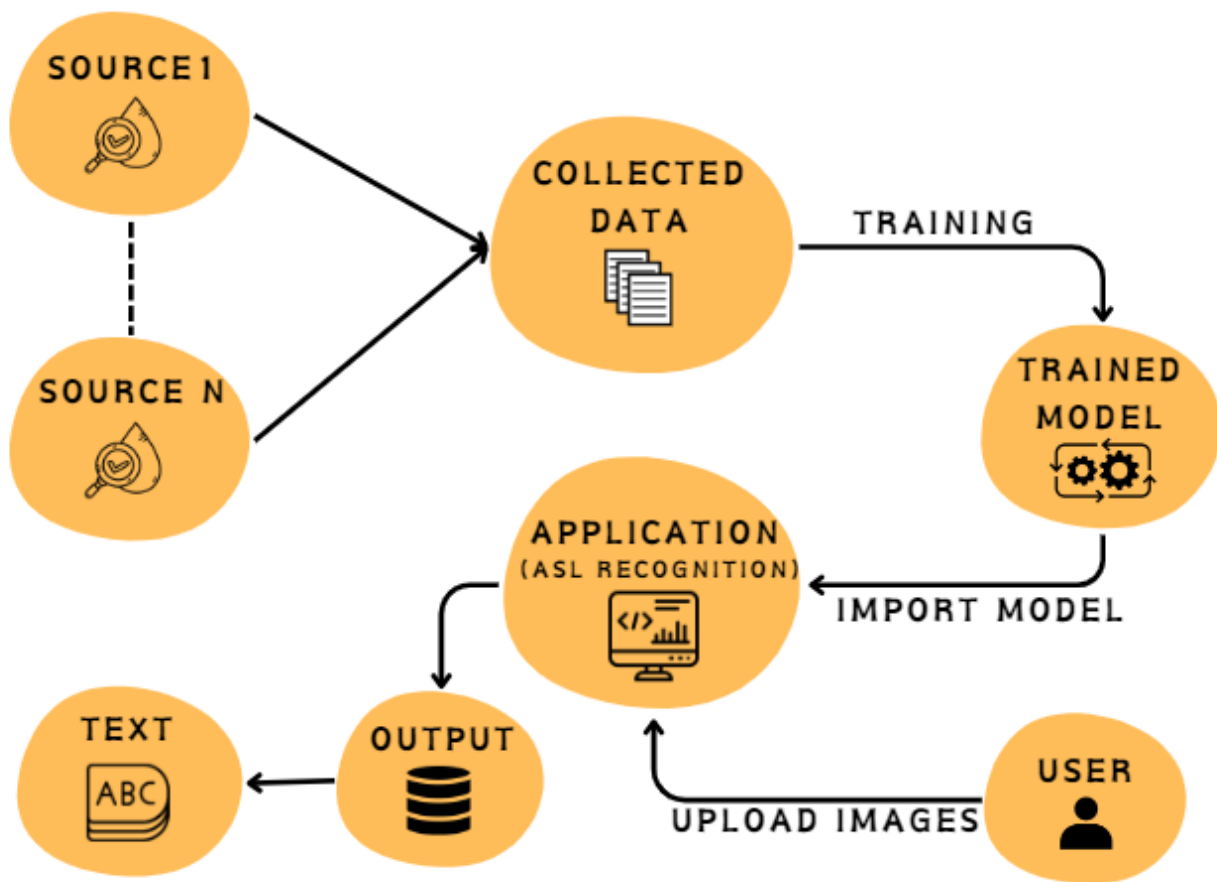
Project Objectives:

- Know fundamental concepts and techniques of Convolutional Neural Networks.
- Gain a broad understanding of ASL image data.
- Train and fine tune parameters for the best model accuracy.
- Know how to build a web application using the Flask framework.
- Build a complete website and interface with model to predict the sign language images in real time.

Project Flow:

- The user interacts with the UI(User Interface), i.e., our website to choose the image and upload it.
- The chosen image is analysed by the DenseNet model which is integrated with the flask application. After the analysis, the prediction is showcased on the website.

Our solution architecture looks as follows:



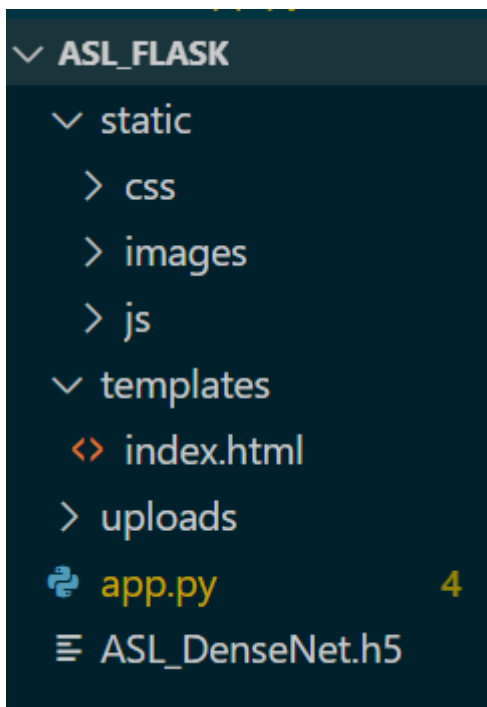
Steps involved to execute this are:

- **Data Collection:** Download the dataset using Google Colab from Kaggle.
- **Data Preprocessing:** Preprocess the data by rescaling, zooming, etc.
- **Model Building:**
 - Import the necessary libraries for building the CNN model.
 - Define the input shape of the image data.
 - Initialise the model and add layers to the model:
 - **Convolutional Layers:** Apply filters to the input image to create feature maps.
 - **Fully Connected Layers:** Flatten the output of the convolutional layers and apply fully connected layers to classify the images
 - Compile the model using Adam optimizer, loss function, and metrics to be used during training.
- **Model Training:** Train the model using the training set with the help of the ImageDataGenerator class to augment the images during training. Monitor the accuracy of the model on the validation set to avoid overfitting.

- Model Evaluation: Evaluate the performance of the trained model on the testing set.
- Model Deployment: Save the model for future use and deploy it in real-world applications.
- Flask Development: Develop a user interface and integrate it with the saved model using Flask.

Project Structure:

Create a Project folder which contains files as shown below

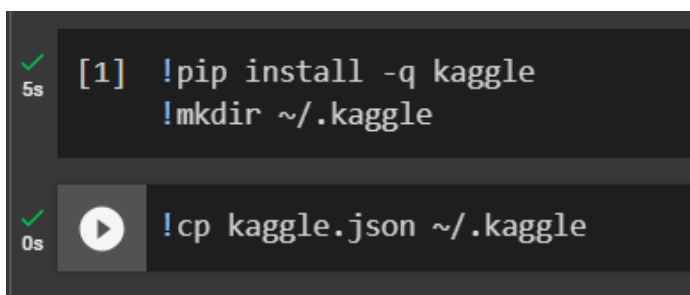


1. DATA COLLECTION

Kaggle data set link :

<https://www.kaggle.com/datasets/debashishsau/aslamerican-sign-language-alphabet-dataset>

Below three lines of code are used to set up the Kaggle API credentials in a Google Colab notebook. The first line installs the Kaggle. The second line creates a directory named .kaggle in the root directory of the Colab notebook. The third line copies the kaggle.json file (which contains the API credentials) to the .kaggle directory.



This code downloads the ASL Alphabet dataset from Kaggle.

```
!kaggle datasets download -d debashishsau/aslamerican-sign-language-aplhabet-dataset

Warning: Your Kaggle API key is readable by other users on this system! To fix this, you can run 'chmod 600 /root/.kaggle/kaggle.json'
Downloading aslamerican-sign-language-aplhabet-dataset.zip to /content
100% 4.19G/4.20G [00:53<00:00, 135MB/s]
100% 4.20G/4.20G [00:53<00:00, 84.3MB/s]
```

+ Code + Text

```
[ ] !unzip /content/aslamerican-sign-language-aplhabet-dataset.zip
```

Streaming output truncated to the last 5000 lines.

```
inflating: ASL_Alphabet_Dataset/asl_alphabet_train/space/space (2863).jpg
inflating: ASL_Alphabet_Dataset/asl_alphabet_train/space/space (2864).jpg
inflating: ASL_Alphabet_Dataset/asl_alphabet_train/space/space (2865).jpg
inflating: ASL_Alphabet_Dataset/asl_alphabet_train/space/space (2866).jpg
inflating: ASL_Alphabet_Dataset/asl_alphabet_train/space/space (2867).jpg
inflating: ASL_Alphabet_Dataset/asl_alphabet_train/space/space (2868).jpg
inflating: ASL_Alphabet_Dataset/asl_alphabet_train/space/space (2869).jpg
inflating: ASL_Alphabet_Dataset/asl_alphabet_train/space/space (287).jpg
inflating: ASL_Alphabet_Dataset/asl_alphabet_train/space/space (2870).jpg
inflating: ASL_Alphabet_Dataset/asl_alphabet_train/space/space (2871).jpg
```

2. DATA PREPARATION

- Installing Imagedata Generator

```
[ ] from keras.preprocessing.image import ImageDataGenerator
```

- Model Building

```
[ ] from keras.applications import DenseNet121
from keras.models import Sequential
from keras.layers import Flatten, Dense
from keras.optimizers import Adam
```

- Test

```
[ ] from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import numpy as np
```

- Configuration

This snippet of code gives us the path to the train and test folders that are present in the ASL_Alphabet_Dataset folder.

```
[6] trainpath = "/content/ASL_Alphabet_Dataset/asl_alphabet_train"
testpath = "/content/ASL_Alphabet_Dataset/asl_alphabet_test"
```

3. DATA PREPROCESSING

- The following snippet of code shows the data augmentation. It generates image data generators using the ImageDataGenerator class from the Keras library for the train and test set of images. The generators take the images from the train and test folder and apply the same data augmentation technique of rescaling the image pixel values to a range of 0 to 1.

```
✓ 0s [6] trainpath = "/content/ASL_Alphabet_Dataset/asl_alphabet_train"
      testpath = "/content/ASL_Alphabet_Dataset/asl_alphabet_test"

✓ 0s [7] train_datagen = ImageDataGenerator(rescale = 1. / 255, zoom_range = 0.2, shear_range = 0.2)
      test_datagen = ImageDataGenerator(rescale = 1. / 255)

✓ 4s [8] train = train_datagen.flow_from_directory(trainpath, target_size = (108, 108), batch_size = 32)
      test = test_datagen.flow_from_directory(testpath, target_size = (108, 108), batch_size = 32)

Found 223074 images belonging to 29 classes.
Found 0 images belonging to 0 classes.
```

- Train labels/Indices

This line of code prints the class labels we have in the training and testing data.

```
✓ 0s ▶ train.class_indices

{ 'A': 0,
  'B': 1,
  'C': 2,
  'D': 3,
  'E': 4,
  'F': 5,
  'G': 6,
  'H': 7,
  'I': 8,
  'J': 9,
  'K': 10,
  'L': 11,
  'M': 12,
  'N': 13,
  'O': 14,
  'P': 15,
  'Q': 16,
  'R': 17,
  'S': 18,
  'T': 19,
  'U': 20,
  'V': 21,
  'W': 22,
  'X': 23,
  'Y': 24,
  'Z': 25,
  'del': 26,
  'nothing': 27,
  'space': 28 }
```

4. MODEL BUILDING

- Load the pre-trained DenseNet model.

```
✓ 4s ▶ densenet_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(108, 108, 3))
```

⏏ Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/densenet/densenet121_weights_tf_dim_ordering_tf_data_format.h5 [=====] - 0s 0us/step

- Initialise the model and add the layers to the model.

```
✓ 0s [17] model = Sequential()

✓ 2s [18] model.add(densenet_model)

✓ 0s [20] densenet_model.trainable = False

✓ 0s [21] model.add(Flatten())
      model.add(Dense(256, activation='relu'))
      model.add(Dense(29, activation='softmax'))
```

- Summary of the Model

```
✓ 0s [22] model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
densenet121 (Functional)	(None, 3, 3, 1024)	7037504
flatten_1 (Flatten)	(None, 9216)	0
dense_2 (Dense)	(None, 256)	2359552
dense_3 (Dense)	(None, 29)	7453

=====

Total params: 9404509 (35.88 MB)
Trainable params: 2367005 (9.03 MB)
Non-trainable params: 7037504 (26.85 MB)

=====

- Compile the model using optimizer and metrics.

```
▶ model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

5. MODEL TRAINING

```
1 model.fit(train, steps_per_epoch = 100, epochs = 45, validation_data = test)
```

```
Epoch 1/45
100/100 [=====] - 520s 5s/step - loss: 2.4617 - accuracy: 0.3613
Epoch 2/45
100/100 [=====] - 480s 5s/step - loss: 1.0630 - accuracy: 0.6847
Epoch 3/45
100/100 [=====] - 506s 5s/step - loss: 0.8007 - accuracy: 0.7572
Epoch 4/45
100/100 [=====] - 482s 5s/step - loss: 0.6588 - accuracy: 0.8141
Epoch 5/45
100/100 [=====] - 478s 5s/step - loss: 0.5311 - accuracy: 0.8403
Epoch 6/45
100/100 [=====] - 482s 5s/step - loss: 0.4857 - accuracy: 0.8484
Epoch 7/45
100/100 [=====] - 482s 5s/step - loss: 0.4738 - accuracy: 0.8537
Epoch 8/45
100/100 [=====] - 476s 5s/step - loss: 0.5580 - accuracy: 0.8309
Epoch 9/45
100/100 [=====] - 471s 5s/step - loss: 0.3819 - accuracy: 0.8838
Epoch 10/45
100/100 [=====] - 485s 5s/step - loss: 0.3528 - accuracy: 0.8916
Epoch 11/45
100/100 [=====] - 458s 5s/step - loss: 0.3054 - accuracy: 0.9009
Epoch 12/45
100/100 [=====] - 454s 5s/step - loss: 0.3408 - accuracy: 0.8991
Epoch 13/45
100/100 [=====] - 464s 5s/step - loss: 0.2567 - accuracy: 0.9212
Epoch 14/45
100/100 [=====] - 454s 5s/step - loss: 0.2963 - accuracy: 0.9100
Epoch 15/45
100/100 [=====] - 456s 5s/step - loss: 0.3246 - accuracy: 0.8991
Epoch 16/45
100/100 [=====] - 451s 5s/step - loss: 0.3096 - accuracy: 0.9062
```

```
model.fit(train, steps_per_epoch = 100, epochs = 45, validation_data = test)
```

```
Epoch 16/45
100/100 [=====] - 451s 5s/step - loss: 0.3096 - accuracy: 0.9062
Epoch 17/45
100/100 [=====] - 460s 5s/step - loss: 0.2730 - accuracy: 0.9216
Epoch 18/45
100/100 [=====] - 463s 5s/step - loss: 0.2688 - accuracy: 0.9228
Epoch 19/45
100/100 [=====] - 463s 5s/step - loss: 0.2482 - accuracy: 0.9222
Epoch 20/45
100/100 [=====] - 469s 5s/step - loss: 0.3183 - accuracy: 0.8988
Epoch 21/45
100/100 [=====] - 475s 5s/step - loss: 0.2512 - accuracy: 0.9237
Epoch 22/45
100/100 [=====] - 455s 5s/step - loss: 0.2507 - accuracy: 0.9303
Epoch 23/45
100/100 [=====] - 460s 5s/step - loss: 0.2350 - accuracy: 0.9319
Epoch 24/45
100/100 [=====] - 458s 5s/step - loss: 0.2460 - accuracy: 0.9231
Epoch 25/45
100/100 [=====] - 469s 5s/step - loss: 0.1844 - accuracy: 0.9375
Epoch 26/45
100/100 [=====] - 460s 5s/step - loss: 0.2047 - accuracy: 0.9319
Epoch 27/45
100/100 [=====] - 461s 5s/step - loss: 0.2295 - accuracy: 0.9316
Epoch 28/45
100/100 [=====] - 466s 5s/step - loss: 0.2013 - accuracy: 0.9378
Epoch 29/45
100/100 [=====] - 460s 5s/step - loss: 0.2113 - accuracy: 0.9362
Epoch 30/45
100/100 [=====] - 457s 5s/step - loss: 0.2060 - accuracy: 0.9428
Epoch 31/45
100/100 [=====] - 449s 4s/step - loss: 0.2795 - accuracy: 0.9234
Epoch 32/45
```



```

Epoch 32/45
100/100 [=====] - 454s 5s/step - loss: 0.1805 - accuracy: 0.9463
Epoch 33/45
100/100 [=====] - 448s 4s/step - loss: 0.1723 - accuracy: 0.9466
Epoch 34/45
100/100 [=====] - 449s 4s/step - loss: 0.2085 - accuracy: 0.9438
Epoch 35/45
100/100 [=====] - 450s 4s/step - loss: 0.2956 - accuracy: 0.9187
Epoch 36/45
100/100 [=====] - 452s 5s/step - loss: 0.2136 - accuracy: 0.9337
Epoch 37/45
100/100 [=====] - 452s 5s/step - loss: 0.1503 - accuracy: 0.9534
Epoch 38/45
100/100 [=====] - 449s 4s/step - loss: 0.1458 - accuracy: 0.9534
Epoch 39/45
100/100 [=====] - 447s 4s/step - loss: 0.2150 - accuracy: 0.9394
Epoch 40/45
100/100 [=====] - 448s 4s/step - loss: 0.1792 - accuracy: 0.9475
Epoch 41/45
100/100 [=====] - 452s 5s/step - loss: 0.1736 - accuracy: 0.9513
Epoch 42/45
100/100 [=====] - 452s 5s/step - loss: 0.1575 - accuracy: 0.9528
Epoch 43/45
100/100 [=====] - 453s 5s/step - loss: 0.2216 - accuracy: 0.9406
Epoch 44/45
100/100 [=====] - 447s 4s/step - loss: 0.2127 - accuracy: 0.9378
Epoch 45/45
100/100 [=====] - 467s 5s/step - loss: 0.1417 - accuracy: 0.9581
<keras.src.callbacks.History at 0x7bf973e20880>

```

➤ Save the model

```
model.save("densenet_model.h5")
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3079: UserWarning: You are saving your model as an HDF5 file via `model.save()`. The current default is to save as a Keras 3 `.keras` file. To save in the older format use `save_api.save_model(model, filepath)` instead.
```

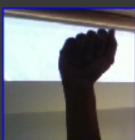
6. LOAD AND TEST THE MODEL

```
[1] from tensorflow.keras.models import load_model
    from tensorflow.keras.preprocessing import image
    import numpy as np
```

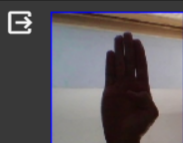
```
[ ] model = load_model("/content/densenet_model.h5")
```

```
[2] img1 = image.load_img(r"/content/A_test.jpg", target_size = (108, 108))
    img2 = image.load_img(r"/content/B_test.jpg", target_size = (108, 108))
    img3 = image.load_img(r"/content/C_test.jpg", target_size = (108, 108))
```


```
[3] img1
```



```
img2
```



```
[5] img3
```



```
[6] x1 = image.img_to_array(img1)
x2 = image.img_to_array(img2)
x3 = image.img_to_array(img3)
```

```
x1
```

```
array([[[ 1.,  1., 255.],
        [ 2.,  0., 247.],
        [ 0.,  2., 251.],
        ...,
        [ 6.,  2., 245.],
        [ 5.,  0., 241.],
        [ 0.,  5., 249.]],
       [[ 1.,  0., 241.],
        [ 82., 76., 186.],
        [ 90., 80., 140.],
        ...,
        [115., 108., 178.],
        [120., 112., 189.],
        [ 12., 12., 170.]],
       [[ 0.,  2., 245.],
```

```
[12] x1 = np.expand_dims(x1,axis=0)
x2 = np.expand_dims(x2,axis=0)
x3 = np.expand_dims(x3,axis=0)
```

```
[13] pred1 = model.predict(x1)
pred2 = model.predict(x2)
pred3 = model.predict(x3)
```

```
1/1 [=====] - 1s 746ms/step
1/1 [=====] - 0s 260ms/step
1/1 [=====] - 0s 238ms/step
```

Output Prediction:

```
[15] pred_class1 = np.argmax(pred1, axis=1)
pred_class2 = np.argmax(pred2, axis=1)
pred_class3 = np.argmax(pred3, axis=1)
```

```
pred_class1
```

```
array([0])
```

```
[17] index = ['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']
```

```
[18] result1 = str(index[pred_class1[0]])
result2 = str(index[pred_class2[0]])
result3 = str(index[pred_class3[0]])
print(result1, result2, result3)
```

```
A B C
```

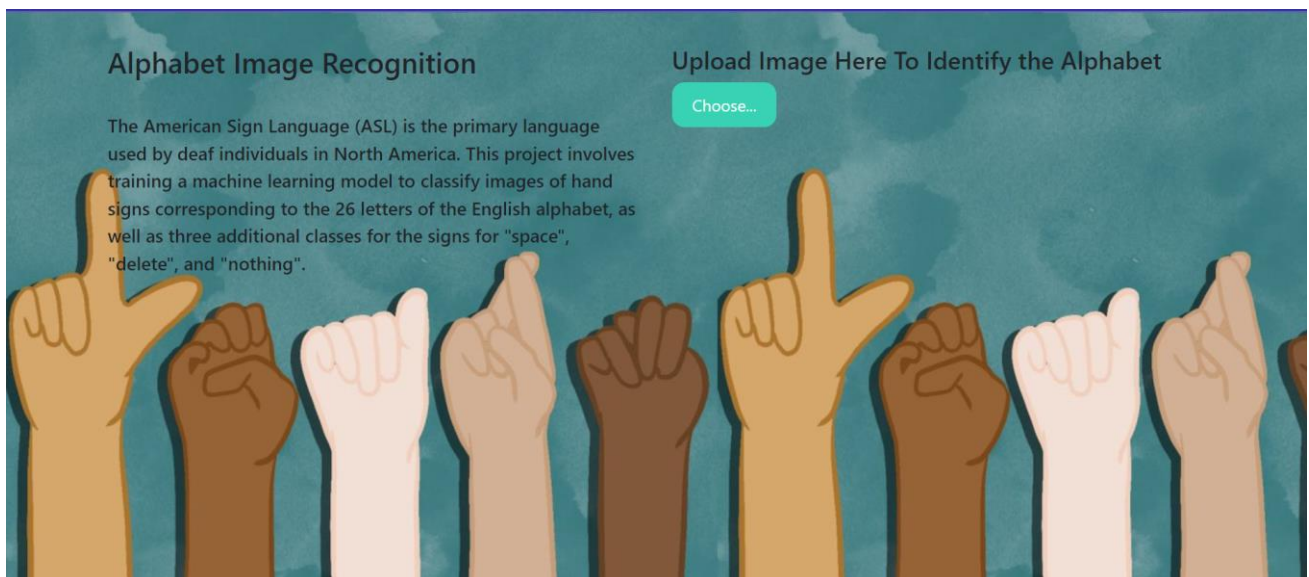
7. APPLICATION BUILDING

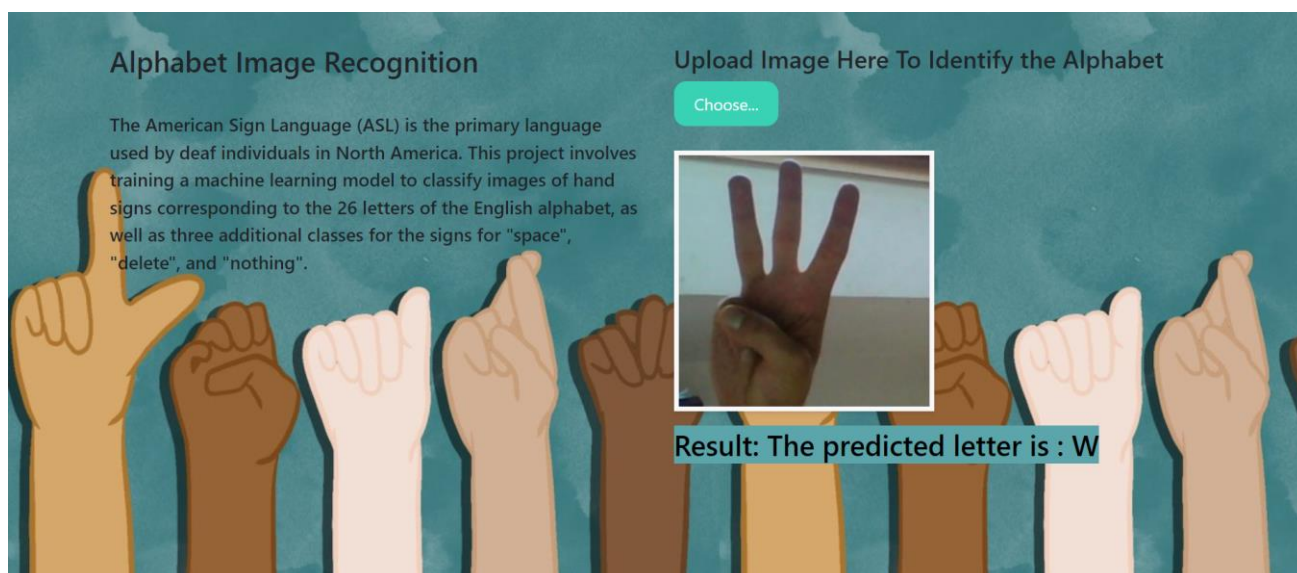
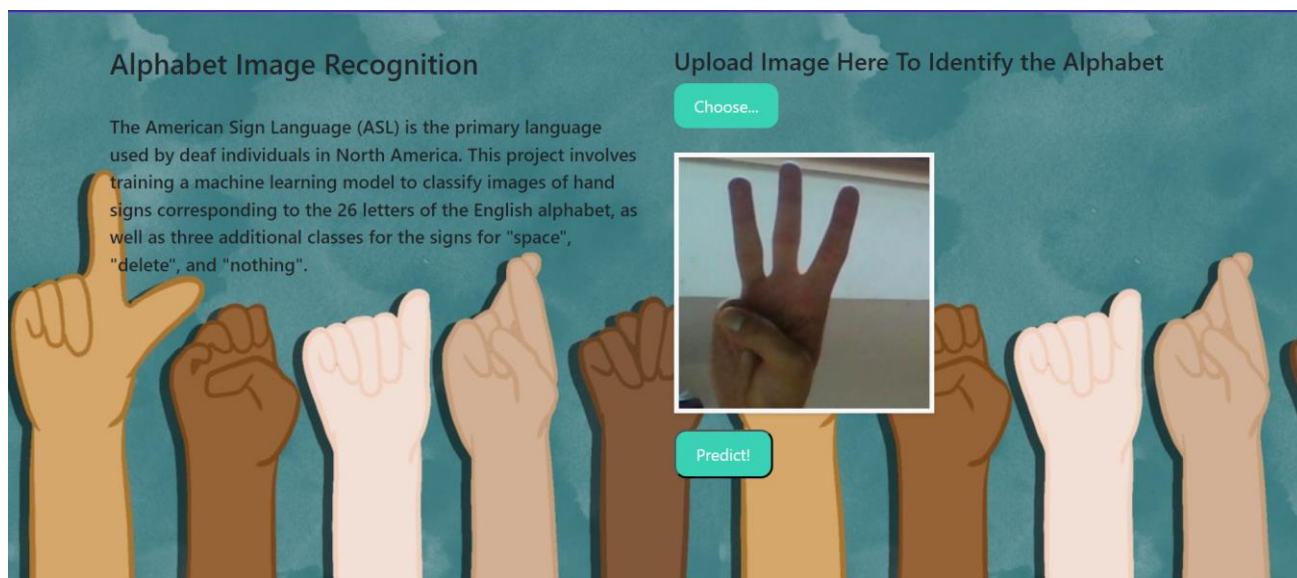
Building our flask application which will be running in our local browser with a user interface. In the flask application, the input(image) is taken from the HTML page. This input is given to the model to predict the alphabet class. Whenever the user interacts with the UI and selects the "Choose..." button, a dialog box of the computer's files open enabling them to choose an image. Our website page accepts .png, .jpg and .jpeg file extensions. After choosing the image, a "Predict" button appears, and on clicking the "Predict" button, the letter is predicted

Activity 1: Create HTML Page

- o We used HTML - index.html to create the front end part of the web page.
- o We also used JavaScript - main.js and CSS - main.css to enhance our functionality and view of HTML pages respectively.

The flow of using our website:





This is a Python script for a Flask web application that loads a pre-trained deep learning model for image classification and makes predictions on images uploaded by the user. The app has two routes ('/' - index.html, '/predict' - for prediction on index.html). The main prediction functionality is implemented in the '/predict' route, where the uploaded image is loaded, pre-processed, and passed through the model for prediction. The predicted result is then displayed on the index page. The app can be run by executing the script, and it will start a local server accessible through a web browser.

```

1
2 import numpy as np
3 import os
4 from tensorflow.keras.models import load_model
5 from tensorflow.keras.preprocessing import image
6 from flask import Flask, request, render_template
7
8 app = Flask(__name__)
9
10 model = load_model("ASL_DenseNet.h5", compile = False)
11
12 @app.route('/')
13 def index():
14     return render_template('index.html')
15
16 @app.route('/predict', methods = ['GET', 'POST'])
17 def upload():
18     if request.method == 'POST':
19         f = request.files['image']
20         print("current path")
21         basepath = os.path.dirname(__file__)
22         print("current path", basepath)
23         filepath = os.path.join(basepath, 'uploads', f.filename)
24         print("upload folder is ", filepath)
25         f.save(filepath)
26
27         img = image.load_img(filepath, target_size = (108, 108))
28         x = image.img_to_array(img)
29         print(x)
30         x = np.expand_dims(x, axis = 0)
31         print(x)
32         y = model.predict(x)
33         preds = np.argmax(y, axis=1)
34         print("prediction", preds)
35         index = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M']
36         text = "The predicted letter is : " + str(index[preds[0]])
37         return text
38
39 if __name__ == '__main__':
40     app.run(debug = False, threaded = False)
41

```

To run this Flask application, simply navigate to the project directory in the terminal and run the command "python app.py". This will start the Flask server, and you can access the web application by visiting the local host address in your web browser. Once you upload an image and submit the form, the application will use the trained model to predict the species of the plant in the image and display the result on the page.