ASL (American Sign Language) - Alphabet Image recognition

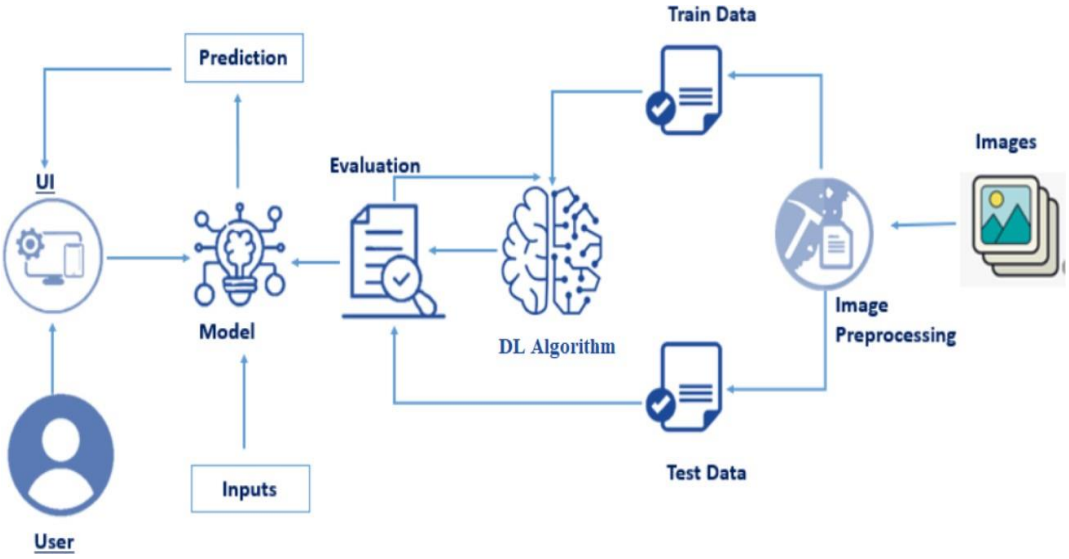
Introduction:

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:



CODING AND SOLUTIONING

```
[ ]  import numpy as np
     import pandas as pd

     # Input data files are available in the read-only "../input/" directory
     # For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory

     import os
```

Create the train and test data

COMMENTS

Before we can work on the data, let's start off by importing the necessary libraries.

- **cv2** - We use cv2 to load image from a specified file and also resize it to the desired pixel size
- **tensorflow** - This library will help us construct and train a neural network that will do the classification for us
- **tqdm** - a smart progress meter
- The training set contains 87,000 images which are 200x200 pixels.There are 29 classes, of which 26 are for the letters A-Z and 3 classes for SPACE, DELETE and NOTHING.
- While reading the images, I have resized it to 50x 50 pixels. The labels_map dictionary maps each of the 29 classes to a corresponding number

```
import cv2
import tensorflow as tf
from tqdm import tqdm
import os
import numpy as np

train_dir = "../input/asl-alphabet/asl_alphabet_train/asl_alphabet_train/"
test_dir =  "../input/asl-alphabet/asl_alphabet_test/asl_alphabet_test/"
IMG_SIZE = 50
labels_map = {'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}
```

```
[ ] def create_train_data():
        x_train = []
        y_train = []
        for folder_name in os.listdir(train_dir):
            label = labels_map[folder_name]
            for image_filename in tqdm(os.listdir(train_dir + folder_name)):
                path = os.path.join(train_dir,folder_name,image_filename)
                img = cv2.resize(cv2.imread(path, cv2.IMREAD_GRAYSCALE),(IMG_SIZE, IMG_SIZE ))
                x_train.append(np.array(img))
                y_train.append(np.array(label))
        print("Done creating train data")
        return x_train, y_train
```

```
[ ] def create_test_data():
        x_test = []
        y_test = []
        for folder_name in os.listdir(test_dir):
            label = folder_name.replace("_test.jpg","")
            label = labels_map[label]
            path = os.path.join(test_dir,folder_name)
            img = cv2.resize(cv2.imread(path, cv2.IMREAD_GRAYSCALE),(IMG_SIZE, IMG_SIZE ))
            x_test.append(np.array(img))
            y_test.append(np.array(label))
        print("Done creating test data")
        return x_test,y_test
```

```
[ ] x_train, y_train= create_train_data()
    x_test,y_test = create_test_data()
```

```
100%|          | 3000/3000 [00:14<00:00, 211.42it/s]
100%|          | 3000/3000 [00:13<00:00, 219.73it/s]
100%|          | 3000/3000 [00:13<00:00, 218.30it/s]
100%|          | 3000/3000 [00:13<00:00, 214.64it/s]
100%|          | 3000/3000 [00:14<00:00, 213.55it/s]
100%|          | 3000/3000 [00:13<00:00, 220.80it/s]
100%|          | 3000/3000 [00:13<00:00, 224.30it/s]
100%|          | 3000/3000 [00:13<00:00, 221.95it/s]
100%|          | 3000/3000 [00:13<00:00, 218.74it/s]
100%|          | 3000/3000 [00:13<00:00, 219.04it/s]
100%|          | 3000/3000 [00:13<00:00, 214.95it/s]
100%|          | 3000/3000 [00:13<00:00, 222.88it/s]
100%|          | 3000/3000 [00:13<00:00, 220.27it/s]
100%|          | 3000/3000 [00:13<00:00, 222.29it/s]
100%|          | 3000/3000 [00:13<00:00, 223.57it/s]
```

```
100%|          | 3000/3000 [00:13<00:00, 223.57it/s]
100%|          | 3000/3000 [00:13<00:00, 224.06it/s]
100%|          | 3000/3000 [00:13<00:00, 217.95it/s]
100%|          | 3000/3000 [00:13<00:00, 217.52it/s]
100%|          | 3000/3000 [00:13<00:00, 217.82it/s]
100%|          | 3000/3000 [00:13<00:00, 226.01it/s]
100%|          | 3000/3000 [00:13<00:00, 226.02it/s]
100%|          | 3000/3000 [00:13<00:00, 230.13it/s]
100%|          | 3000/3000 [00:13<00:00, 223.91it/s]
100%|          | 3000/3000 [00:14<00:00, 213.30it/s]
100%|          | 3000/3000 [00:13<00:00, 228.75it/s]
100%|          | 3000/3000 [00:13<00:00, 228.19it/s]
100%|          | 3000/3000 [00:14<00:00, 208.75it/s]
100%|          | 3000/3000 [00:13<00:00, 223.04it/s]
100%|          | 3000/3000 [00:13<00:00, 229.50it/s]
Done creating train data
Done creating test data
```

COMMENTS

- The number of features are 25000 [50 x50 pixels] i.e considering each pixel as a feature for the image.We reshape the images into 1D arrays of 2500 pixels. Each one of those values will be an input node into our deep neural network. We also normalize the inputs to be a value between 0 and 1 (inclusive)
- The number of class are 29, which means there will be 29 neurons in the output layer each representing an output class A-Z, SPACE, NOTHING or DELETE

```
[ ] num_features = 2500
    num_classes = 29

    x_train, x_test = np.array(x_train, np.float32), np.array(x_test, np.float32)
    x_train, x_test = x_train.reshape([-1, num_features]), x_test.reshape([-1, num_features])
    x_train, x_test = x_train / 255., x_test / 255.
```

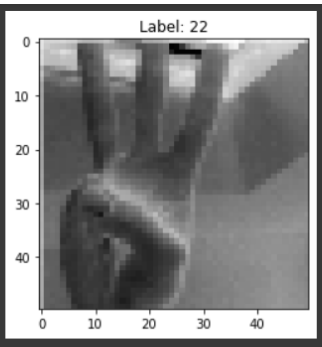
COMMENTS

- The training images are therefore a tensor of shape [87,000, 25000]. The training labels are a one-dimensional tensor of 87,000 labels that range from 0 to 28.
- Let's get a feel of what the images look like compressed to 50 x 50 pixels and in gray scale. Remember we have compressed it from 200 x 200 to 50 x 50. Looking at the image below it is a bit pixellated; but still looks good!

```
[ ] %matplotlib inline
    import matplotlib.pyplot as plt

    def display_image(num):
        label = y_train[num]
        plt.title('Label: %d' % (label))
        image = x_train[num].reshape([IMG_SIZE,IMG_SIZE])
        plt.imshow(image, cmap=plt.get_cmap('gray_r'))
        plt.show()
    display_image(5)
```

[ ]



Determining training and network parameters

COMMENTS

- These parameters or "hyperparameters" are ones we have to experiment with to improve upon the **accuracy** of the neural network. A little tweak could result in a huge difference!
- **learning rate** - controls how much to change the model in response to the estimated error each time the model weights are updated.
- **training steps** - number of epochs
- **batch size** - small chunks for each iteration of training
- **n hidden** - number of neurons in the hidden layer
- We use tf.data API to shuffle the data and divide it into batches

```
[ ]  # Training parameters.
     learning_rate = 0.001
     training_steps = 5000
     batch_size = 250
     display_step = 500

     # Network parameters.
     n_hidden =  300# Number of neurons.
```

```
[ ]  # Use tf.data API to shuffle and batch data.
     train_data = tf.data.Dataset.from_tensor_slices((x_train, y_train))
     train_data = train_data.repeat().shuffle(87000).batch(batch_size).prefetch(1)
```

Constructing the neural network

> 📝 COMMENTS
>
> - We start by initializing weights randomly for each layer in the neural network. We use the RandomNormal API to do so.
> - We can use bias to allow the activation function to be shifted to the left or right, to better fit the data. Bias makes it easier for the neural networks to fire. The biases are initialized to zero and are learnt during training.
> - The neural network consist of 2 hidden layers, with 300 neurons in each layer and one output layer followed by a **softmax** function that converts the weights of the neural network into probabilities for each class. Softmax helps turn the weights into probabilities and make it a classification problem.

```
[ ]  # Store layers weight & bias

     # A random value generator to initialize weights initially
     random_normal = tf.initializers.RandomNormal()

     weights = {
         'h1': tf.Variable(random_normal([num_features, n_hidden])),
         'h2': tf.Variable(random_normal([n_hidden, n_hidden])),
         'out': tf.Variable(random_normal([n_hidden, num_classes]))
     }
     biases = {
         'b': tf.Variable(tf.zeros([n_hidden])),
         'out': tf.Variable(tf.zeros([num_classes]))
     }
```

```
     def neural_nets(input_data):
         hidden_layer1 = tf.add(tf.matmul(input_data,weights['h1']),biases['b'])
         hidden_layer1 = tf.nn.sigmoid(hidden_layer1)

         hidden_layer2 = tf.add(tf.matmul(hidden_layer1,weights['h2']),biases['b'])
         hidden_layer2 = tf.nn.sigmoid(hidden_layer2)

         out_layer = tf.add(tf.matmul(hidden_layer1,weights['out']),biases['out'])

         return tf.nn.softmax(out_layer)
```

Defining our loss function and SGD Optimizer

COMMENTS

- The loss function for measuring the progress in gradient descent: **cross entropy**. It uses a logrithamic scale that penalizes incorrect classification more than the ones that are close.
- Finally we set up the **stocashtic gradient descent optimizer**. The gradient tape is a TensorFlow API that does **reverse mode auto differentiation**. It's the new way of setting up neural nets from scratch in Tensorflow 2.
- We define a function to measure the accuracy of the neural network. It does this by selecting the class with the highest probability and matching it to the true class.

```
[ ] def cross_entropy(y_pred, y_true):
        # Encode label to a one hot vector.
        y_true = tf.one_hot(y_true, depth=num_classes)
        # Clip prediction values to avoid log(0) error.
        y_pred = tf.clip_by_value(y_pred, 1e-9, 1.)
        # Compute cross-entropy.
        return tf.reduce_mean(-tf.reduce_sum(y_true * tf.math.log(y_pred)))
```

```
optimizer = tf.keras.optimizers.SGD(learning_rate)

def run_optimization(x, y):
    # Wrap computation inside a GradientTape for automatic differentiation.
    with tf.GradientTape() as g:
        pred = neural_nets(x)
        loss = cross_entropy(pred, y)

    # Variables to update, i.e. trainable variables.
    trainable_variables = list(weights.values()) + list(biases.values())

    # Compute gradients.
    gradients = g.gradient(loss, trainable_variables)

    # Update W and b following gradients.
    optimizer.apply_gradients(zip(gradients, trainable_variables))
```

```
[ ] def accuracy(y_pred, y_true):
        # Predicted class is the index of highest score in prediction vector (i.e. argmax).
        #print("argmax:",tf.argmax(y_pred,1))
        #print("cast",tf.cast(y_true, tf.int64))
        correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.cast(y_true, tf.int64))
        return tf.reduce_mean(tf.cast(correct_prediction, tf.float32), axis=-1)
```

Training the neural network

> 📝 COMMENTS
>
> - Now that we have everything set up, let's try to run it!
> - We train the model in 4000 epochs or training steps. At each step we run the optimization function on a small chunk of training data 250 records, in our case
> - Every 100 steps we display the current **Loss function and Accuracy** of the model

```
[ ]  # Run training for the given number of steps.
     for step, (batch_x, batch_y) in enumerate(train_data.take(training_steps), 1):
         # Run the optimization to update W and b values.
         run_optimization(batch_x, batch_y)

         if step % display_step == 0:
             pred = neural_nets(batch_x)
             loss = cross_entropy(pred, batch_y)
             acc = accuracy(pred, batch_y)
             print("Training epoch: %i, Loss: %f, Accuracy: %f" % (step, loss, acc))
```

```
Training epoch: 500, Loss: 239.489151, Accuracy: 0.736000
Training epoch: 1000, Loss: 218.471893, Accuracy: 0.768000
Training epoch: 1500, Loss: 194.523163, Accuracy: 0.772000
Training epoch: 2000, Loss: 183.343811, Accuracy: 0.796000
Training epoch: 2500, Loss: 189.522919, Accuracy: 0.800000
Training epoch: 3000, Loss: 143.437256, Accuracy: 0.876000
Training epoch: 3500, Loss: 162.636841, Accuracy: 0.812000
Training epoch: 4000, Loss: 214.353699, Accuracy: 0.696000
Training epoch: 4500, Loss: 116.986954, Accuracy: 0.848000
Training epoch: 5000, Loss: 135.456482, Accuracy: 0.872000
```

Validating the model

```
[ ]  # Test model on validation set.
     pred = neural_nets(x_test)
     print("Test Accuracy: %f" % accuracy(pred, y_test))
```

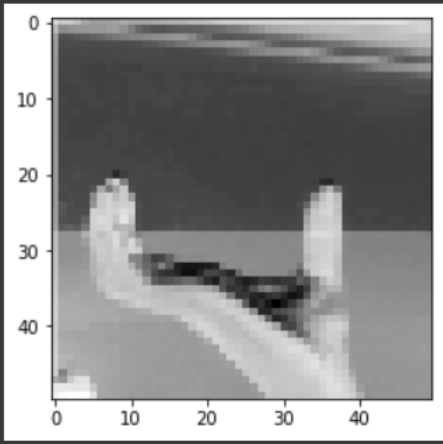
```
Test Accuracy: 1.000000
```

> 📊 My insight
>
> - The highest accuracy I have got is **1.0** meaning all the images have been correctly classified in the test set. This might change each time we train the model because each time we start with different random weights and biases.
> - We can tweak the hyperparameters and the layers and neurons in the network to try out different topologies to see if they give better results.

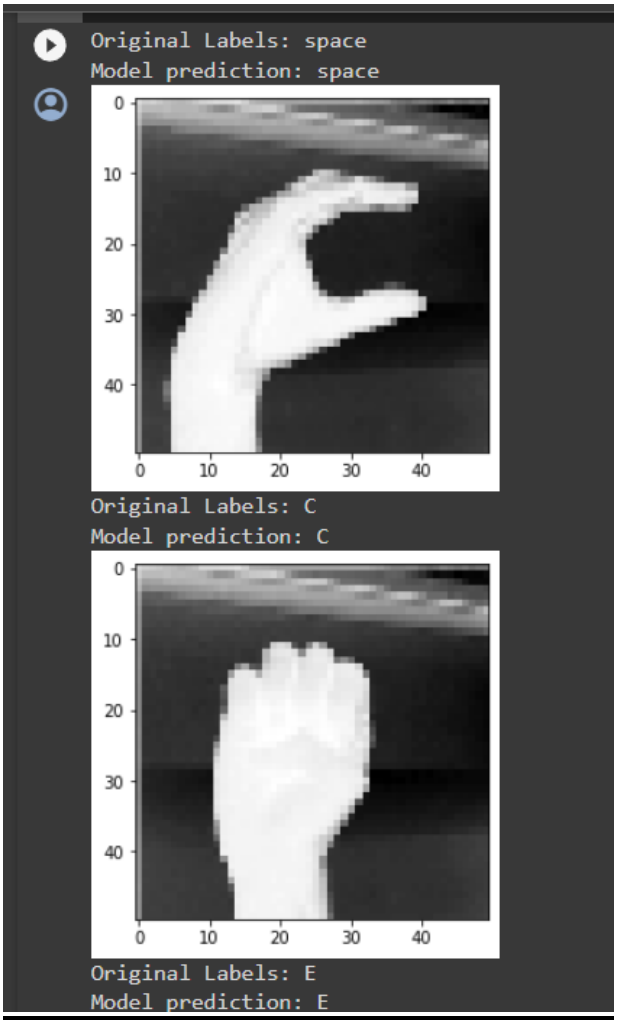
```
[ ]  def get_key(val):
         for key, value in labels_map.items():
             if val == value:
                 return key
```

Results

```
n_images = 28
predictions = neural_nets(x_test)
for i in range(n_images):
    model_prediction = np.argmax(predictions.numpy()[i])
    plt.imshow(np.reshape(x_test[i], [50, 50]), cmap='gray_r')
    plt.show()
    print("Original Labels: %s" % get_key(y_test[i]))
    print("Model prediction: %s" % get_key(model_prediction))
```



```
Original Labels: space
Model prediction: space
```



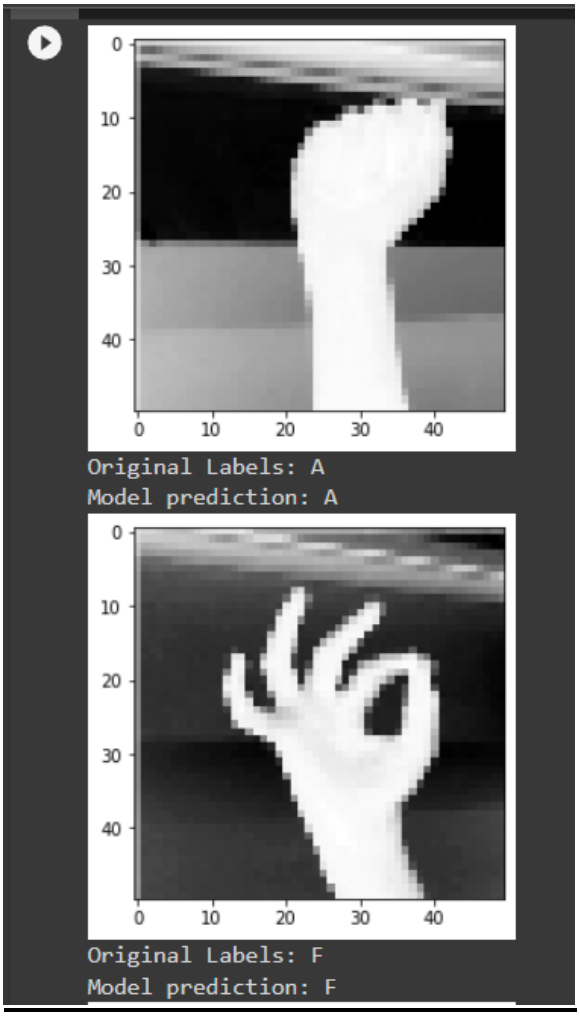
```
Original Labels: space
Model prediction: space
```



```
Original Labels: C
Model prediction: C
```



```
Original Labels: E
Model prediction: E
```




```
Original Labels: A
Model prediction: A
```



```
Original Labels: F
Model prediction: F
```