SIGN LANGUAGE RECOGNITION

Machine Learning Project Report - 2021

MEDTOUREASY, NEW DELHI

Abhi Lad - abhilad1009@gmail.com

MedTourEasy

About

MedTourEasy, a global healthcare company, provides you the informational resources needed to evaluate your global options. It helps you find the right healthcare solution based on specific health needs, affordable care while meeting the quality standards that you expect to have in healthcare. MedTourEasy improves access to healthcare for people everywhere. It is an easy to use platform and service that helps patients to get medical second opinions and to schedule affordable, high-quality medical treatment abroad.

The MTE Proposition

Accessibility: We connect you with trusted health care providers and we partner with internationally accredited institutions. We are committed to upholding patient privacy and use state-of-the-art encryption in all transactions.

Transparency: We believe that when making health decisions, the more information you have the better. Hospitals are listed with accreditation levels, staff experience, facility pictures, procedure prices and reviews from former patients. By using your personal dashboard, you contact the hospital's staff directly.

Privacy: We do not and never will sell your personal data. Simply send us an inquiry and we will help you get a personalized quote from a specialist. Each quote includes a treatment plan and cost estimate.

ACKNOWLEDGMENTS

I am very grateful to MedTourEasy to provide me with an opportunity to learn about Machine Learning and work on project to further my knowledge and help me gain real world experience. This experience provided me with skills required to work in corporate environment and also paved a way for my career in Machine Learning.

I convey my thanks to Mr. Ankit Hasija, the training head at MedTourEasy for laying out necessary training coursework and providing a comprehensive guide for Machine Learning project. I also thank him for providing a seamless learning environment which flattened the learning curve for fresher like me. Finally, I would like to thank the MedTourEasy team for providing subsidized access to Datacamp training modules, which were necessary for completing the certification course.

ABSTRACT

Owing to the advancements in computational capabilities, hardware and software, many human cognitive abilities like vision, speech etc. which were not possible before are now developing rapidly. NLP based machine learning algorithms have now enabled physically handicapped people to get work done effortlessly which seemed impossible before. Computer vision assisted walking sticks are making commute for blind people even safer. One such problem is also faced by deaf or hard-of-hearing people, which is to communicate with people not familiar with sign language. ASL (American Sign Language) is one of the most widely used language used by such people. ASL is well defined and very expressive, which makes it suitable candidate for computer vision tasks. Our goal here is to make a machine learning model which can predict hand gestures from ASL and convert it in English vocabulary.

Table of Contents

1. INTRODUCTION

1.1 About the Project

American Sign Language (ASL) is the primary language used by many deaf individuals in North America, and it is also used by hard-of-hearing and hearing individuals. The language is as rich as spoken languages and employs signs made with the hand, along with facial gestures and bodily postures.
The goal of this project is to make a convolutional neural network which takes a hand gesture image as inputs and classify them into specific category of ASL letters.

1.2 Project Workflow

The project can be divided into following high level modules:
1) Data loading and visualization
2) Data preprocessing
3) Defining ML mode
4) Training and testing

The figure 1.1 given below shows all the major tasks performed in this project.

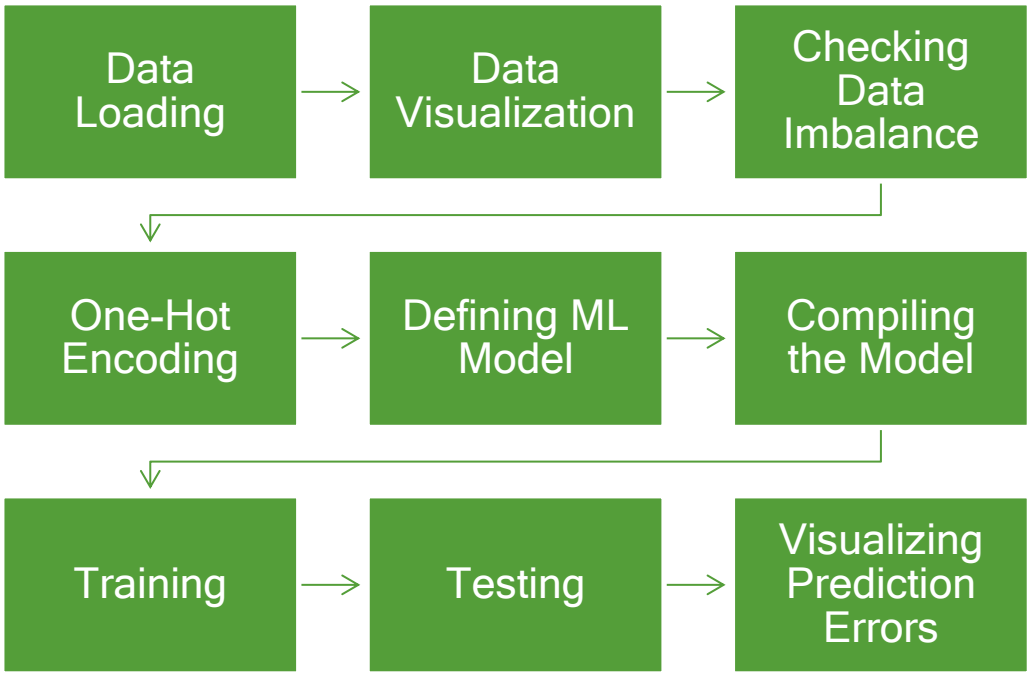

Figure 1.1: The workflow described as major tasks in the project.

1.3 System Overview

The Machine Learning system as an overview from user's perspective can be shown as given below in figure 1.2.

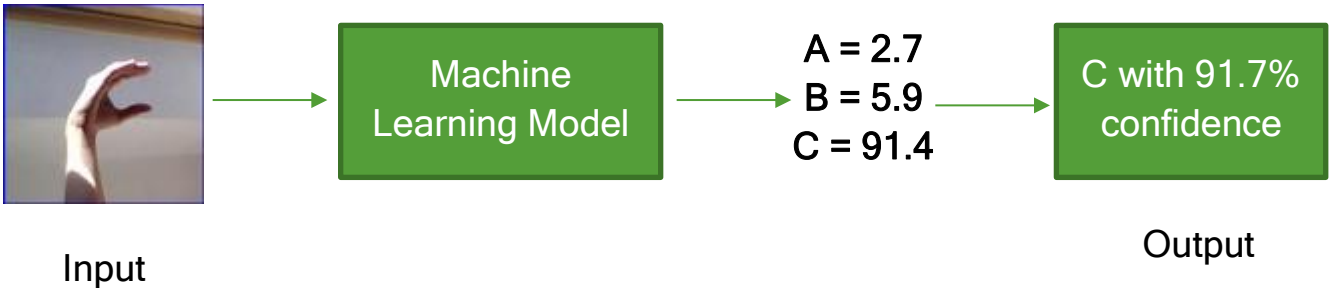

Figure 1.2: The overview of system from user's perspective.

The user feeds in images of hand signs as an input to the Machine Learning model. The ML model performs necessary operations and outputs probability values which indicates the confidence with which the model can assign a given label to the input image. The maximum of the probability values is used as the output label for the given model and is displayed to the user. The data processing part is not visible to the user and is thus left out from user overview of the system.

2. DATA AND METHODS

2.1 Data Description

The ASL dataset used in this project is a scaled down version of the complete ASL and has only 3 classes, i.e. letters A, B and C. Each class of data contains 1000 rgb images with dimensions 100x100 pixels. Images inside each class vary on number of factors. These factors include light exposure, hand placement in the frame and hand orientation. The data is not divided into training and testing sets and thus the results of training and testing section are depended on the splitting of data during the runtime.

2.2 Importing and Processing Data

2.2.1 Loading the dataset

All the functions for importing the dataset from respective folders are defined in a separate python module 'sign_language.py'. This module consists of two important functions:
1) path_to_tensor( )
2) load_data( )

The path_to_tensor( ) function uses keras image preprocessing module to load the image from specified path using image.load_img( ) function. The image is then converted to numpy array which can be used as input to ML model by using image.img_to_array( ) function. All the images are rescaled to desired dimension while loading, 50x50 for our model.

```
def path_to_tensor(img_path, size):
    # loads RGB image as PIL.Image.Image type
    img = image.load_img(img_path, target_size=(size, size))
    # convert PIL.Image.Image type to 3D tensor
    x = image.img_to_array(img)
    # convert 3D tensor to 4D tensor
    return np.expand_dims(x, axis=0)
```

Since our dataset does not provide separate training and testing sets, we need to split the data during runtime. The load_data( ) function does this splitting job. The path to dataset containing folders of images is presented as argument to this function alongside test_split argument which specifies the percentage of data to be assigned as testing set. The images

are normalized in range 0-1 by dividing the image array values by 255. The alphabetic labels are converted to integers, i.e. A-0, B-1and C-2. To ensure that data imbalance does not occur, the data is shuffled using random.shuffle( ) function before splitting into training and testing set.

```
def load_data(container_path='datasets', folders=['A', 'B', 'C'],
              size=2000, test_split=0.2, seed=0):
    """
    Loads sign language dataset.
    """

    filenames, labels = [], []

    for label, folder in enumerate(folders):
        folder_path = join(container_path, folder)
        images = [join(folder_path, d)
                  for d in sorted(listdir(folder_path))]
        labels.extend(len(images) * [label])
        filenames.extend(images)

    random.seed(seed)
    data = list(zip(filenames, labels))
    random.shuffle(data)
    data = data[:size]
    filenames, labels = zip(*data)


    # Get the images
    x = paths_to_tensor(filenames).astype('float32')/255
    # Store the one-hot targets
    y = np.array(labels)

    x_train = np.array(x[:int(len(x) * (1 - test_split))])
    y_train = np.array(y[:int(len(x) * (1 - test_split))])
    x_test = np.array(x[int(len(x) * (1 - test_split)):])
    y_test = np.array(y[int(len(x) * (1 - test_split)):])

    return (x_train, y_train), (x_test, y_test)
```

```
from datasets import sign_language

# Load pre-shuffled training and test datasets
(x_train, y_train), (x_test, y_test) = sign_language.load_data()
```

We call the function using sign_language.load_data( ) in our working environment.

2.2.2 Visualize the training data

While loading the dataset, we change the labels to integer values. However, we need to display the results with the original label to the user. In order to do this, we make a list of actual label values which can correspond to the integer values used for the ML model. Now we visualize the image data with their actual labels to verify their integer correspondence. We use matplotlib to display images and their labels in a figure.

```
# Store labels of dataset
labels = ['A','B','C']

# Print the first several training images, along with the labels
fig = plt.figure(figsize=(20,5))
for i in range(36):
    ax = fig.add_subplot(3, 12, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_train[i]))
    ax.set_title("{}".format(labels[y_train[i]]))
plt.show()
```

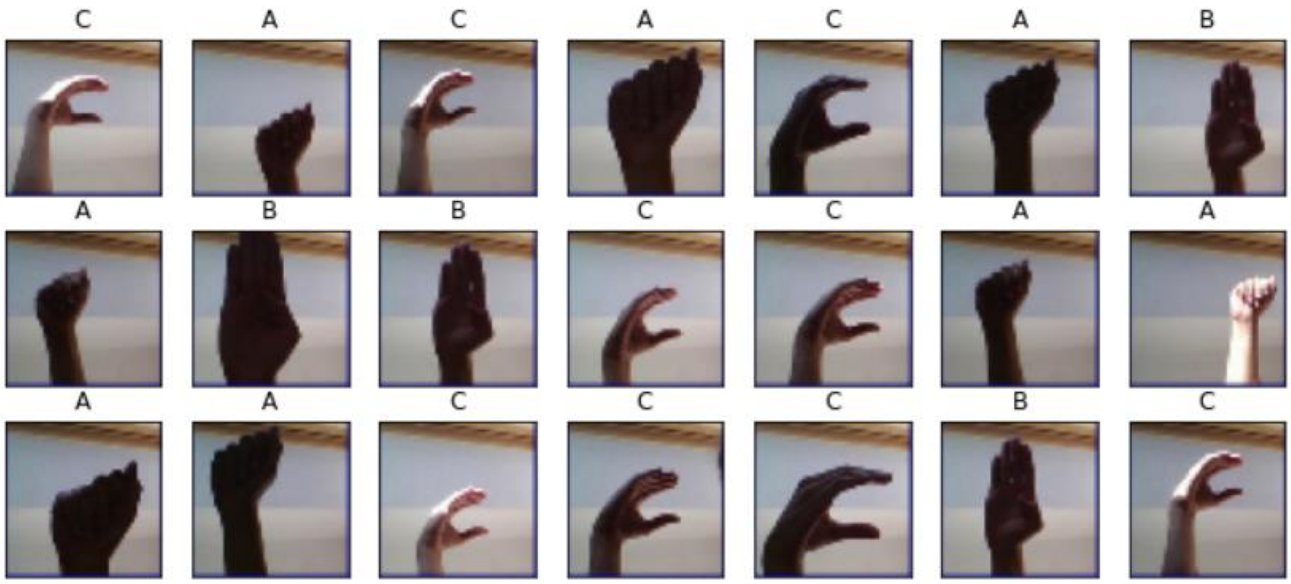

Figure 2.1: The plot shows images along with their labels on top.

2.2.3 Examining the dataset

We split the data into training and testing set by randomly shuffling the original data. We examine the generated training and testing set to check for data imbalance. The statistics about the data helps us understand how well our model will perform in case of unseen data. We count the number of samples by comparing the labels in the training and testing

set, i.e. 0 for A, 1 for B and 2 for C. We store the count of samples in num_A_train, num_B_train and num_C_train for labels in training dataset and num_A_test, num_B_test and num_C_test for test dataset.

```
# Number of A's in the training dataset
num_A_train = sum(y_train==0)
# Number of B's in the training dataset
num_B_train = sum(y_train==1)
# Number of C's in the training dataset
num_C_train = sum(y_train==2)

# Number of A's in the test dataset
num_A_test = sum(y_test==0)
# Number of B's in the test dataset
num_B_test = sum(y_test==1)
# Number of C's in the test dataset
num_C_test = sum(y_test==2)

# Print statistics about the dataset
print("Training set:")
print("\tA: {}, B: {}, C: {}".format(num_A_train, num_B_train, num_C_train))
print("Test set:")
print("\tA: {}, B: {}, C: {}".format(num_A_test, num_B_test, num_C_test))
```

We get the following statistics for distribution of data in training and test datasets.

	Training dataset	**Test dataset**
num_A	540	118
num_B	528	144
num_C	532	138
Total	1600	400

These numbers show that the data is evenly distributed and there is no apparent data imbalance. However, looking at plain numbers is not very intuitive for understanding data distribution. So, we plot an additional bar graph in order to visualize the distribution of data across training and testing dataset. We use pyplot.bar( ) function of matplotlib library in order to make the bar graph. The full method for the same is given in the code block below.

```
num_A=[num_A_train,num_A_test]
num_B=[num_B_train,num_B_test]
num_C=[num_C_train,num_C_test]

br1 = np.arange(2)
br2 = [x + 0.25 for x in br1]
br3 = [x + 0.25 for x in br2]

plt.bar(br1, num_A, color ='r', width = 0.25,
        edgecolor ='grey', label ='num_A')
plt.bar(br2, num_B, color ='g', width = 0.25,
        edgecolor ='grey', label ='num_B')
plt.bar(br3, num_C, color ='b', width = 0.25,
        edgecolor ='grey', label ='num_C')


plt.xticks([r + 0.25 for r in range(2)],
           ['Train', 'Test'])
plt.legend(loc='upper right')

plt.show()
```

Figure 2.2 shows the distribution of data of 3 classes for training and testing dataset. The bar graph makes it simple to understand that the data is evenly distributed without looking at the actual number of individual class samples.

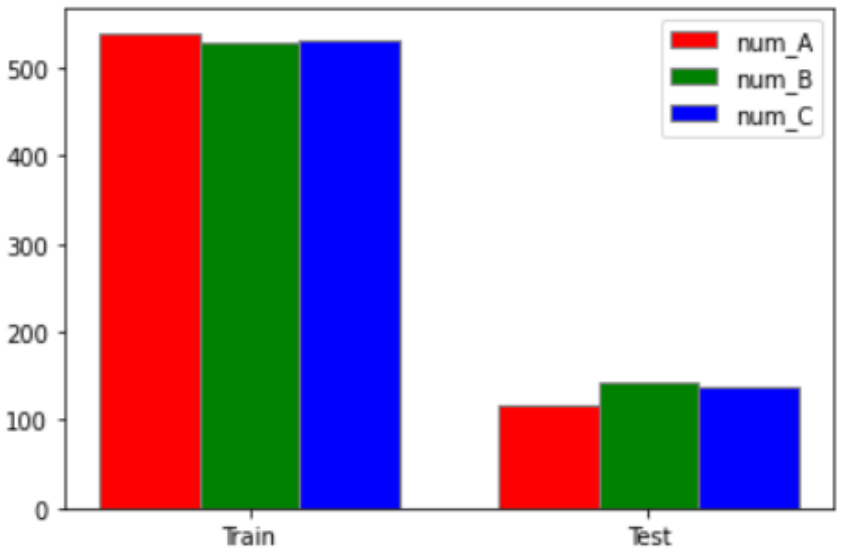

Figure 2.2: The bar plot describing the distribution of categorical data across training and testing set..

2.2.4 One-hot encoding of labels

Currently, our labels for each of the letters are encoded as categorical integers, where 'A', 'B' and 'C' are encoded as 0, 1, and 2, respectively. However, Keras models do not accept labels in this format, and we must first one-hot encode the labels before supplying them to a Keras model.
This conversion will turn the one-dimensional array of labels into a two-dimensional array. Each row in the two-dimensional array of one-hot encoded labels corresponds to a different image. The row has a 1 in the column that corresponds to the correct label, and 0 elsewhere.
The one-hot encoding for our labels will be as follows:
0 is encoded as [1, 0, 0]
1 is encoded as [0, 1, 0]
2 is encoded as [0, 0, 1]

We will use keras function to_categorical( ) to change the labels into one-hot encodings. The encoded labels are stored as y_train_OH and y_test_OH.

```
from keras.utils import np_utils

# One-hot encode the training labels
y_train_OH = np_utils.to_categorical(y_train)

# One-hot encode the test labels
y_test_OH = np_utils.to_categorical(y_test)
```

2.3 Machine Learning Model

2.3.1 Defining the model

This network accepts an image of an American Sign Language letter as input. The output layer returns the network's predicted probabilities that the image belongs in each category. We specify the model using the models.Sequential( ) method of keras. Since our model has only one input and one output, the sequential model is best suited for such conditions.

We use 4 different kinds of layer in our model:

1) 2D Convolution layers (x2) : This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs.

Hyperparameters: filters = 5 and 15, kernel_size =5*5, padding = ‘same’, activation = ‘relu’

2) 2D Maxpooling layers (x2) : Downsamples the input representation by taking the maximum value over the window defined by pool_size for each dimension along the features axis.
Hyperparameters: pool_size = 4*4

3) Flatten layer : Flattens the input data to 1 dimensional array. Required for input data of a fully connected layer.

4) Dense layer (Output) : Dense implements the operation: output = activation(dot(input, kernel) + bias) where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer.
Hyperparameters: units = 3 (# of output classes), activation = ‘softmax’ (gives probability across all output labels)

```
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Flatten, Dense
from keras.models import Sequential

model = Sequential()
# First convolutional layer accepts image input
model.add(Conv2D(filters=5, kernel_size=5, padding='same', activation='relu',
                 input_shape=(50, 50, 3)))
# Add a max pooling layer
model.add(MaxPooling2D(4))
# Add a convolutional layer
model.add(Conv2D(filters=15, kernel_size=5, padding='same', activation='relu'
))
# Add another max pooling layer
model.add(MaxPooling2D(4))
# Flatten and feed to output layer
model.add(Flatten())
model.add(Dense(3, activation='softmax'))

# Summarize the model
model.summary()
```

Figure 2.3 shows the summary of our convolutional machine learning model. Total number of trainable parameters are 2678.

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 50, 50, 5)         380
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 12, 12, 5)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 12, 12, 15)        1890
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 3, 3, 15)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 135)               0
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 408
=================================================================
Total params: 2,678
Trainable params: 2,678
Non-trainable params: 0
_________________________________________________________________
```

Figure 2.3: The summary of Convolutional Machine Learning Model.

2.3.2 Compiling the model

The model needs to be compiled in order to enable training of the parameters. We need to define the loss and optimizer for model to learn the parameter values. We use the following functions to compile our model:

Optimizer : 'rmsprop'. The optimizer is responsible for improving the training by adjusting the loss gradient. ‘rmsprop’ does this by using momentum of rms of gradients.

Loss : 'categorical_crossentropy'. The loss is the error function used for measuring the difference between actual and predicted values. The gradient of loss is propagated backwards in model for training the parameters.

Metric : ‘accuracy’. The metric is used to identify how well our model is performing. We use accuracy as our metric.

```
# Compile the model
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

2.3.3 Model Training

We use the model.fit() method to train the model.We pass features as x_train and the label data as y_train_OH to train the model for 2 epochs. We use validation split of 0.2, i.e. during each epoch 20% of the training data is left for validation. We use batch size of 32. As the batch size increase, the training time reduces but also the rate of improvement of model decreases. We store our model's training history in hist variable.

```
# Train the model
hist = model.fit(x_train,y_train_OH,epochs=2,validation_split=0.2,batch_size=32)
```

```
Epoch 1/2
40/40 [==============================] - 1s 8ms/step - loss: 1.0934 - accuracy: 0.4487 - val_loss: 0.7859 - val_accuracy: 0.7031
Epoch 2/2
40/40 [==============================] - 0s 5ms/step - loss: 0.7466 - accuracy: 0.7201 - val_loss: 0.5379 - val_accuracy: 0.8687
```

Figure 2.4: The metrics and loss of model during 2 epochs of training.

As shown in figure 2.4, at end of 2 epochs, the training loss is 0.7466 and the training accuracy is 0.72, and the validation loss is 0.54 and the validation accuracy is 0.87. Thus, our model achieves >80% validation accuracy and satisfies the criteria for acceptable model. However, if the model trained for more epochs, the training and validation accuracy tend to reach 100% as our training dataset is relatively small. This can be seen in figure 2.5.

```
Epoch 1/10
40/40 [==============================] - 1s 7ms/step - loss: 0.9671 - accuracy: 0.5691 - val_loss: 0.5993 - val_accuracy: 0.7969
Epoch 2/10
40/40 [==============================] - 0s 5ms/step - loss: 0.5777 - accuracy: 0.7971 - val_loss: 0.3779 - val_accuracy: 0.8906
Epoch 3/10
40/40 [==============================] - 0s 5ms/step - loss: 0.3726 - accuracy: 0.8772 - val_loss: 0.2190 - val_accuracy: 0.9594
Epoch 4/10
40/40 [==============================] - 0s 5ms/step - loss: 0.2447 - accuracy: 0.9278 - val_loss: 0.1406 - val_accuracy: 0.9719
Epoch 5/10
40/40 [==============================] - 0s 4ms/step - loss: 0.1577 - accuracy: 0.9600 - val_loss: 0.0971 - val_accuracy: 0.9812
Epoch 6/10
40/40 [==============================] - 0s 5ms/step - loss: 0.1294 - accuracy: 0.9674 - val_loss: 0.1083 - val_accuracy: 0.9719
Epoch 7/10
40/40 [==============================] - 0s 4ms/step - loss: 0.0951 - accuracy: 0.9712 - val_loss: 0.0718 - val_accuracy: 0.9750
Epoch 8/10
40/40 [==============================] - 0s 4ms/step - loss: 0.0748 - accuracy: 0.9859 - val_loss: 0.0424 - val_accuracy: 0.9906
Epoch 9/10
40/40 [==============================] - 0s 5ms/step - loss: 0.0789 - accuracy: 0.9788 - val_loss: 0.0335 - val_accuracy: 0.9906
Epoch 10/10
40/40 [==============================] - 0s 4ms/step - loss: 0.0440 - accuracy: 0.9927 - val_loss: 0.0301 - val_accuracy: 0.9906
```

Figure 2.5: The metrics and loss of model during 10 epochs of training.

3. RESULTS AND CONCLUSION

3.1 Test Results

After training our model for 2 epochs, we get final training accuracy 72% and validation accuracy 87%. However, these metrics are based on training data and thus cannot account for unseen or new data outside training set. In order to better evaluate the model performance, we use the test dataset which we separated while loading the data. This test dataset contains images which are not used in training the model and thus are completely new for the model. We use model.evaluate() function of keras sequential model to measure the model performance on test data. We pass x_test as features and y_test_OH as labels while evaluation.

```
# Obtain accuracy on test set
score = model.evaluate(x=x_test,
                       y=y_test_OH,
                       verbose=0)
print('Test accuracy:', score[1])
```

As shown in figure 3.1, we get test accuracy of 84%. The classification accuracy on the test dataset is similar to the training dataset, this is a good sign that the model did not overfit to the training data.

```
13/13 [==============================] - 0s 8ms/step - loss: 0.5906 - accuracy: 0.8400
Test accuracy: 0.8399999737739563
```

Figure 3.1: The metrics and loss of model during testing.

In order to better understand the how loss and accuracy of model change over time, we plot the Accuracy vs Epoch and Loss vs Epoch graph. We use the training history stored in hist variable to plot these graphs for 2 and 10 epochs. As it can be seen in figure 3.2, the plots on the right shows the accuracy and loss over 10 epochs of training. As the number of epochs increases the plot of accuracy asymptotically reaches 1. The same is true for loss, as the number of training steps increases, the loss asymptotically reaches 0. The plots also indicate that the model is not overfitting as the validation loss follows the same trend as training loss.

2 epochs 10 epochs

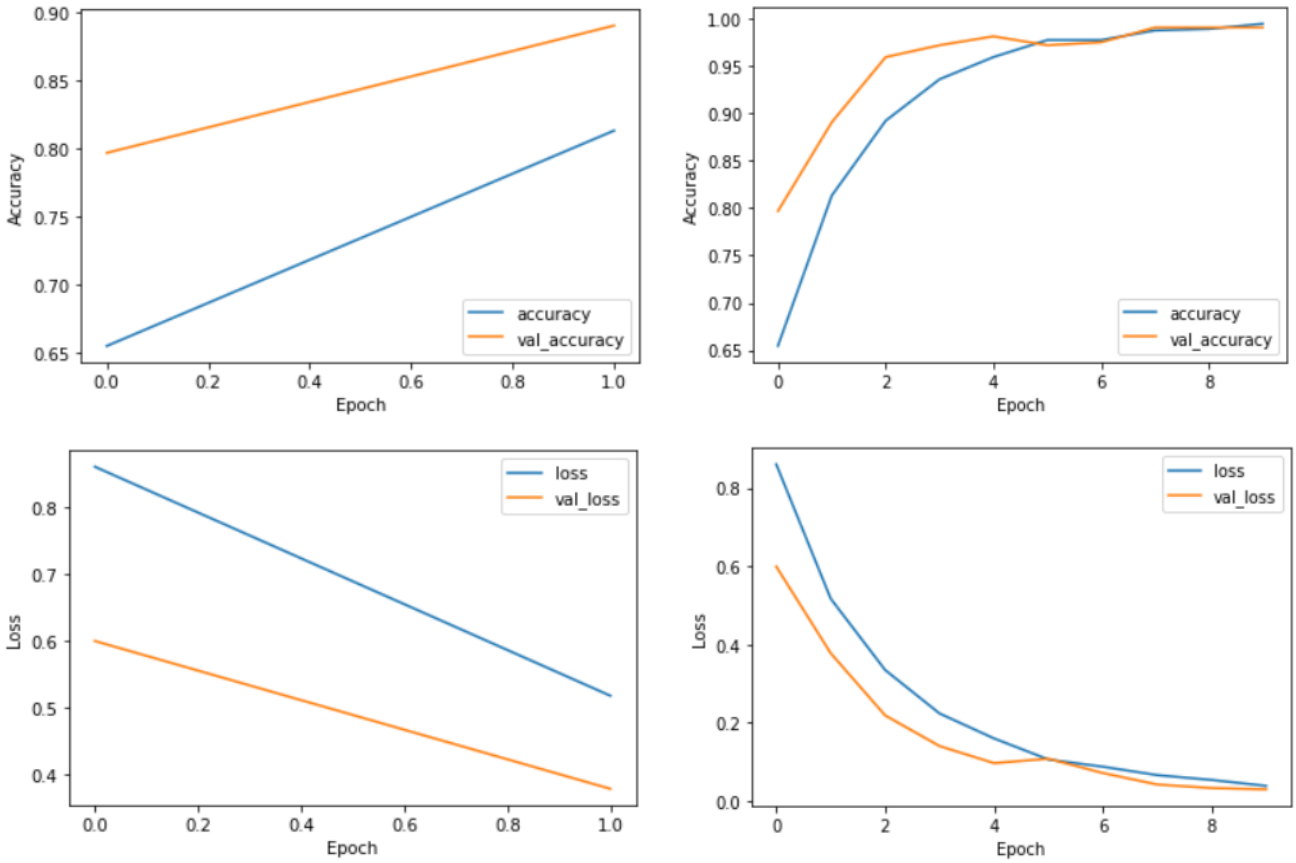

Figure 3.2: The plot of accuracy (above) and loss (below) for different number of training steps: 2 epochs (left) and 10 epochs (right).

3.2 Visualizing Prediction Errors

We take a look at the images that were incorrectly classified by the model. Sometimes, it's possible to review the images to discover special characteristics that are confusing to the model.

We use the model.predict() method to assign y_probs to a numpy array containing the model's predicted probabilities that each image belongs to each image class.We then assign y_preds to the model's predicted labels for each image in the testing dataset. y_preds should be a numpy array where each entry is one of 0, 1, or 2, corresponding to 'A', 'B', and 'C', respectively. We then use the ground truth labels for the testing dataset y_test and the model's predicted labels y_preds to determine which images were misclassified. We assign the variable bad_test_idxs to a one-dimensional numpy array containing all indices corresponding to images that were incorrectly classified by the model. We then use matplotlib to display the misclassifies images with their true label as well as their predicted label.

```
# Get predicted probabilities for test dataset
y_probs = model.predict(x_test)
# Get predicted labels for test dataset
y_preds = np.array([np.argmax(i) for i in y_probs])
# Indices corresponding to test images which were mislabeled
bad_test_idxs = [i for i in range(len(y_test)) if y_preds[i]!=y_test[i]]

# Print mislabeled examples
fig = plt.figure(figsize=(25,4))
for i, idx in enumerate(bad_test_idxs):
    ax = fig.add_subplot(2, np.ceil(len(bad_test_idxs)/2), i + 1, xticks=[], y
ticks=[])
    ax.imshow(np.squeeze(x_test[idx]))
    ax.set_title("{} (pred: {})".format(labels[y_test[idx]], labels[y_preds[id
x]]))
```

Figure 3.3 shows the misclassified images along with their true and predicted labels. We have only shown a small number of misclassified images for 2 epochs of training. As number of epochs increase, the number of misclassified images decreases.

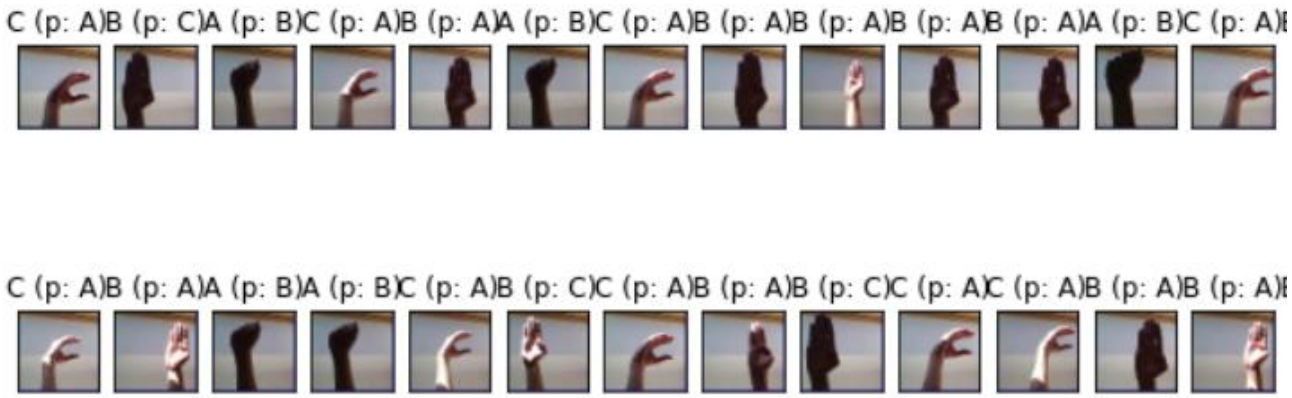

Figure 3.3: The images which are incorrectly labeled are shown with their true label shown as A,B or C and their predicted label is shown inside parenthesis i.e. (p: _).

3.3 Conclusion

As we have seen during training and testing our model, when the number of classes is less and the training data is small with less internal variation, a simple model with few training steps is able to perform well. As ASL is well defined, the images of its hand signs can be used to train a convolutional machine learning model to predict the intended alphabet and can be deployed to bridge communication gap between people with speaking or listening disabilities and those who are not familiar with ASL.

4. FUTURE WORK

In this project, we limited our scope by reducing the number of classes and small number of training samples and using only static images. However, ASL is very expressive and is capable of conveying as much information as English language. To be able to fully translate ASL to English, we also need to take into consideration other variables like hand gestures and facial expressions which contribute to convey more complex information and sentiments. To achieve this, the future work needs to take into account the above mentioned ASL components. Such model may need to be trained on video feed instead of static images and can be optimized to work on live video feed for in frame translation of ASL with all bells and whistles.

5. LINKS AND REFERENCES

Implementation : https://github.com/abhiwolf13/ASL/blob/main/ASL_notebook.ipynb

Dataset : American_Sign_Language

Certification : Datacamp/Machine_Learning_For_Everyone

Technologies used :

- **Python :** https://www.python.org/

Python is an interpreted, high-level and general-purpose programming language. Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural), object-oriented and functional programming.

- **Tensorflow :** https://www.tensorflow.org/

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications.

- **Keras :** https://keras.io/

Keras is the high-level API of TensorFlow 2: an approachable, highly-productive interface for solving machine learning problems, with a focus on modern deep learning. It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity.

- **Numpy :** https://numpy.org/

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

- **Matplotlib :** https://matplotlib.org/

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Matplotlib produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, web application servers, and various graphical user interface toolkits.