Hand Sign Detection

Project Report

SC - 6611

(Neural Network and Deep Learning)

Team Members

(Naing Linn Aung + Min Ko Ko Linn + Min Hein Khant)

Introduction

Hand sign detection is basically a computer vision project. The basic concept is to recognize specific hand gestures from images, video streams or live webcam using machine learning and image processing techniques. Nowadays, this technology is commonly applied in sign language translation, human-computer interaction and virtual reality application.

In terms of sign languages, there are over 200 signs for alphabets, numbers and common words. The number of sign languages may vary depending on the language or system being used. With Regards to types, there are especially two types such as alphabet hand signs and compound word hand signs.

This system is highly beneficial for the deaf and hard-of-hearing community. It mainly aids in bridge communication gaps, allowing individuals to interact with others through automated detection systems. This improves accessibility and inclusivity in various social settings.

Project Overview

Our project emphasizes on alphabets detection rather than words. For hand sign gestures, we reference ASL which is (American Sign Language). The examples of hand gestures are mentioned as follows.

American Sign Language Gestures Sample

As for the methodology, although this hand sign project is basically a computer vision project, we combined it with neural network technology to generate accurate predictions. In order to test this proposed model, we determined to apply two main types such as image testing and live web cam testing.

Main Components of Project

(1) **Data Collection & Preprocessing** : This is the very first phase of our project. We are determined to build our own datasets by taking photos on our own. Since we are going to use the dataset for CNN model, we have to convert it into applicable data format (such as image dimension, colour format, resolution and so on)

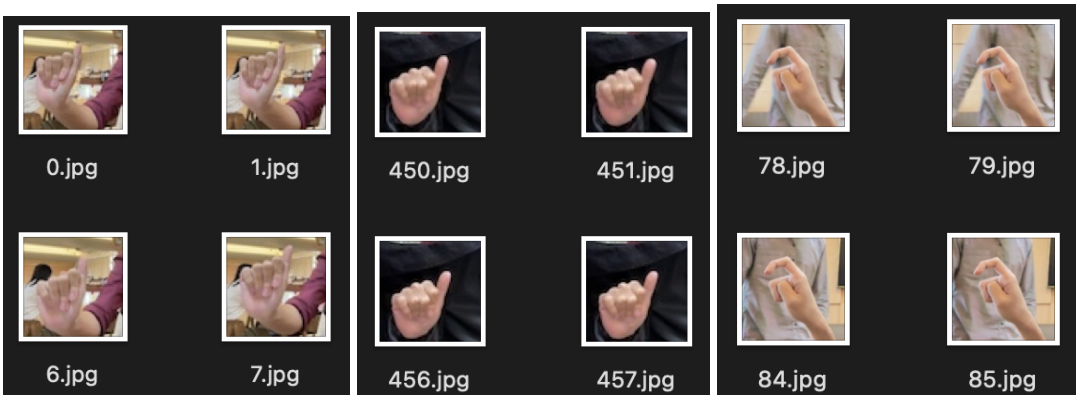
(2) **CNN Model** - Since we are going to incorporate the computer vision project with neural network, we are determined to apply CNN model. This is highly beneficial deep learning technology as they excel at automatically learning and extracting features from images, allowing for accurate recognition of complex hand gestures and variations in hand shape, positions and background.

(3) **Testing and Prediction** - For this phase, we considered testing our model in two main different domains such as images and live webcam. For these two cases, we have to use image processing technology called OpenCV for loading images, putting results on images, loading webcam, media pipelines integration and so on.

Data Collection and Preprocessing

In order to build our own datasets, we applied the online tool called Teachable Machine (https://teachablemachine.withgoogle.com/). We chose this tool for data collection because it allows us to collect hundreds of images easily and quickly in a short amount of time. Additionally, we can also organize into different classes.

Throughout the project, we experimented with different datasets and various configurations. We also tested the model with different dataset sizes (e.g., 50, 100, 150, 500 images, etc.). Ultimately, we found that using 500 images per alphabet provided the best predictions. To further improve accuracy, we added additional images with varying color settings, backgrounds, exposure, contrast, and distances. These are example images of our datasets.



Data Preprocessing

We cannot put all images taken directly into the model because they still need some modification for being able to process in CNN model .Here's a breakdown for the data preprocessing.

Step 1 : Dataset Path and Categories

```
dataset_path = 'TeachableDataset'
```

The location where our dataset is stored is specified. This folder contains subfolders representing different categories (e.g letters , numbers)

```
categories = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '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']
```

The categories list represents the possible labels for each image. Folder names are also specified as the same name as categories so that the dataset folder can be iterated through categories.

Step 2 : Initialize Data and Labels

```
data = []
labels = []
```

Empty lists are created to store the image data and their corresponding labels

Step 3 : Image Loading and Preprocessing

```
for category in categories:
    category_path = os.path.join(dataset_path, category)
    label = categories.index(category)  # index of the ca

    for img_file in os.listdir(category_path):
        img_path = os.path.join(category_path, img_file)

        # Open the image, resize, and convert to array
        img = Image.open(img_path).convert('L')  # Conver
        img = img.resize(image_size)
```

The path to the folder for each is constructed, looping through each category (A,B,etc). Each label is assigned to each image based on the index of the category in the list. Then, the full path for each image file in the category folder is prepared.

Image is opened and **converted into a grayscale ('L' model)**, which simplifies the input data (**from RGB to a single channel**). This is useful for **reducing model complexity**, especially when color is not essential for classification. Then, the image is **resized to (64x64) pixels**. Standardizing the size makes sure that the model receives the input data in a consistent dimension.

Step 4 : Array Conversion

```
img_array = np.array(img)

data.append(img_array)
labels.append(label)
```

The resized grayscale image is converted into a NumPy array to be suitable for processing in machine learning models. The image data is appended to a specified data list and the corresponding label to the labels list.

Step 5 : Normalisation and Reshaping data

```
data = data / 255.0 # normalize image data from range(0 t

data = data.reshape(-1, image_size[0], image_size[1], 1)
```

The pixel values are normalized from a range of [0,255] to [0,1]. Normalization helps speed up the training process and improve model performance by presenting large gradients.
Then, the image data is reshaped into a 3D array format that is expected by CNN model. The last parameter - 1 indicates that the image are in grayscale (single channel)

Step 6 : Label Binarization

```
labels = to_categorical(labels, num_classes=len(categories))
```

The labels are converted from integer indices to one-hot encoded vectors. This is needed for multi-class classification where each class is represented as a binary vector.

Step 7 : Train-Test Split

```
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2,shuffle = True, random_state=42)
```

The dataset is split into training and test sets, with 80% of the data used for training and 20% for testing. Shuffling the data ensures that the training and test sets are representative for the entire dataset. The random state ensures that the results are the same each time we run the code.

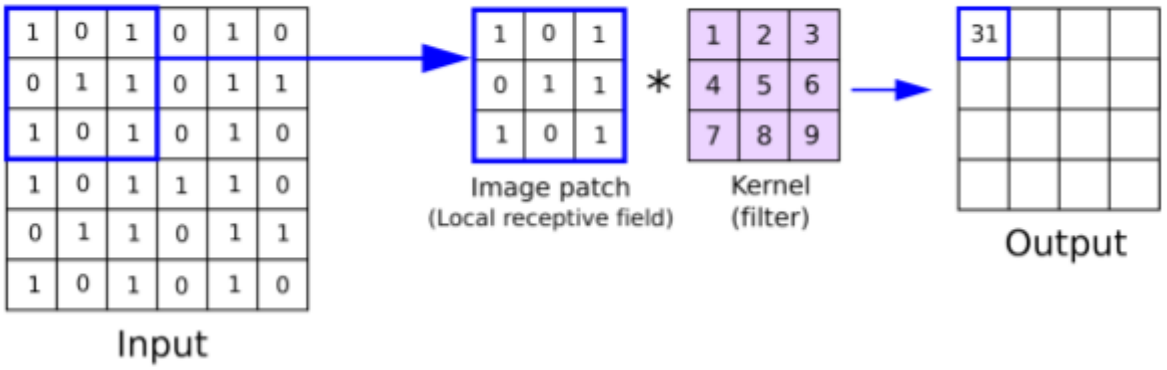
Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN) is a type of neural network specially designed to process structured grid data, like images. CNNs are widely used in computer vision tasks, such as image classification, object detection, and facial recognition.

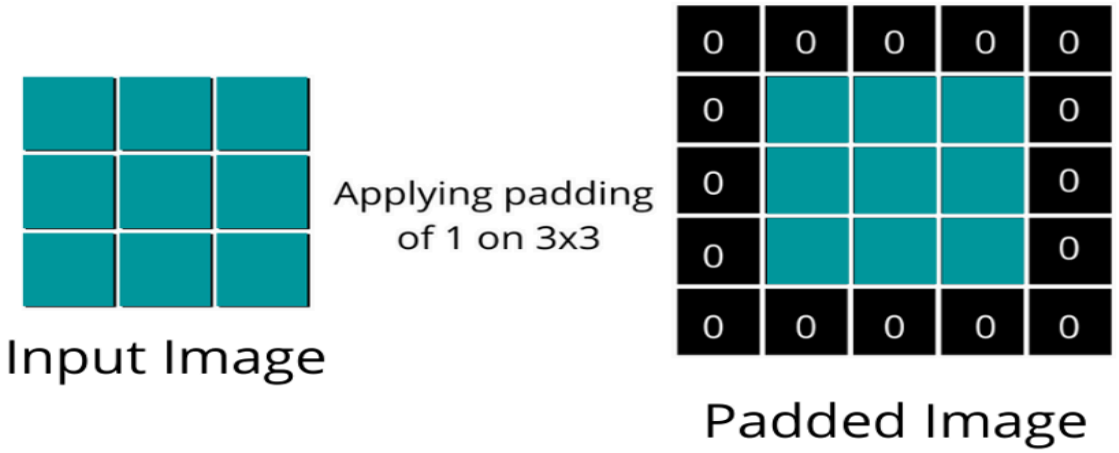
Convolution Layer

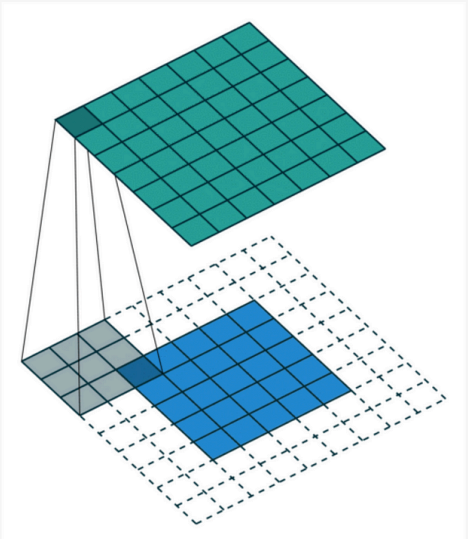
- Convolution is the core operation in CNN. In this layer, small matrices called filters (or kernels) aer slid over the input image to extract features like edges, textures, or objects. Each filter is designed to detect a specific feature.
- The result of this convolutional operation is a feature map, which highlights where certain features occur in the image. Multiple filters are used to generate several feature maps.
- The convolutional operation involves sliding the filter over the image using a step called the stride, which determines how far the filter moves in each step. A smaller stride keeps the output feature map large, while a larger slide reduces its size.
- Padding is used to control the output size. Adding padding around the image ensures that the convolutional preserves the size of the input feature map, especially at the edges.

```
# First Convolutional Block
model.add(Conv2D(32, (3, 3), strides=1, padding='same', activation='relu', input_shape=(64, 64, 1)))
```

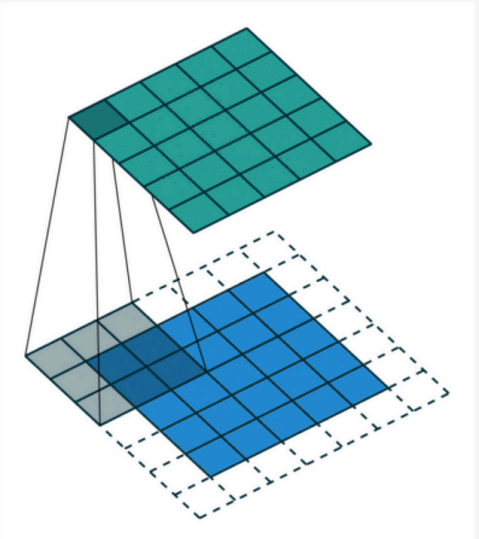


Convolution Operation in a CNN





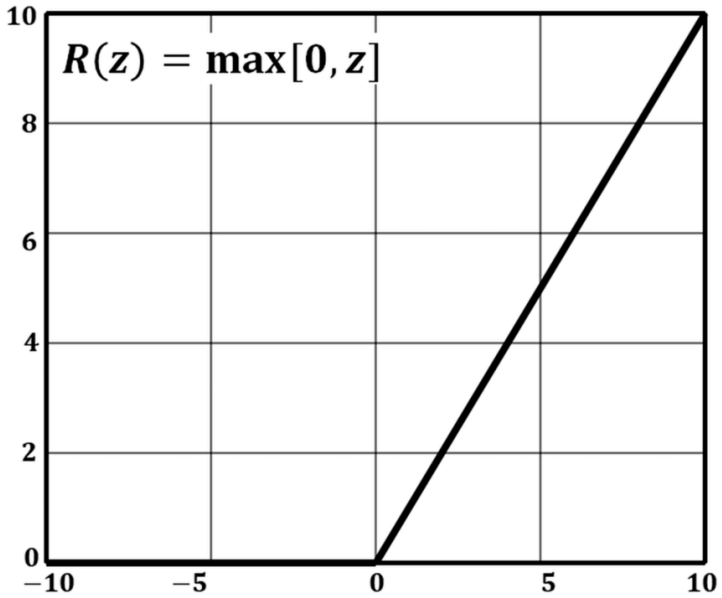
Full padding. Introduces zeros such that all pixels are visited the same amount of times by the filter. Increases size of output.



Same padding. Ensures that the output has the same size as the input.

Activation Function(ReLU)

After applying convolution, the feature map is passed through an activation function, typically ReLU (Rectified Linear Unit).ReLU is computationally efficient because it involves only a simple thresholding at zero. This leads to faster training times as compared to other activation functions like sigmoid or tanh, which involve more complex calculations (exponentials).
ReLU speeds up the convergence of the model during training, allowing it to learn faster.

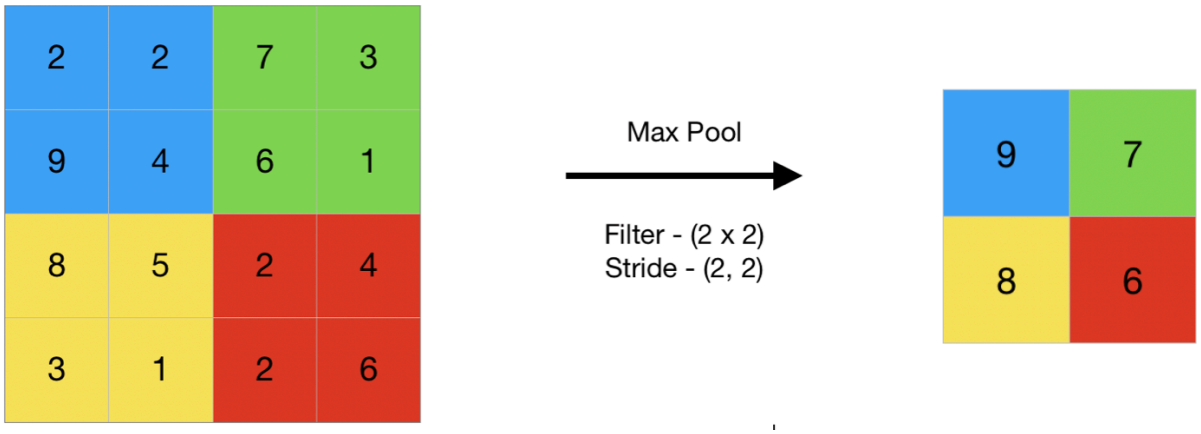


ReLU Activation Function

Pooling Layer

- A pooling layer reduces the spatial dimensions (width and height) of the feature map, but retains the most important information. This helps make the model more computationally efficient and reduces overfitting.
- The most common type of pooling is **Max Pooling**, which takes the maximum value from a region (like a 2x2 grid) of the feature map.

```
model.add(MaxPooling2D((2, 2)))
```

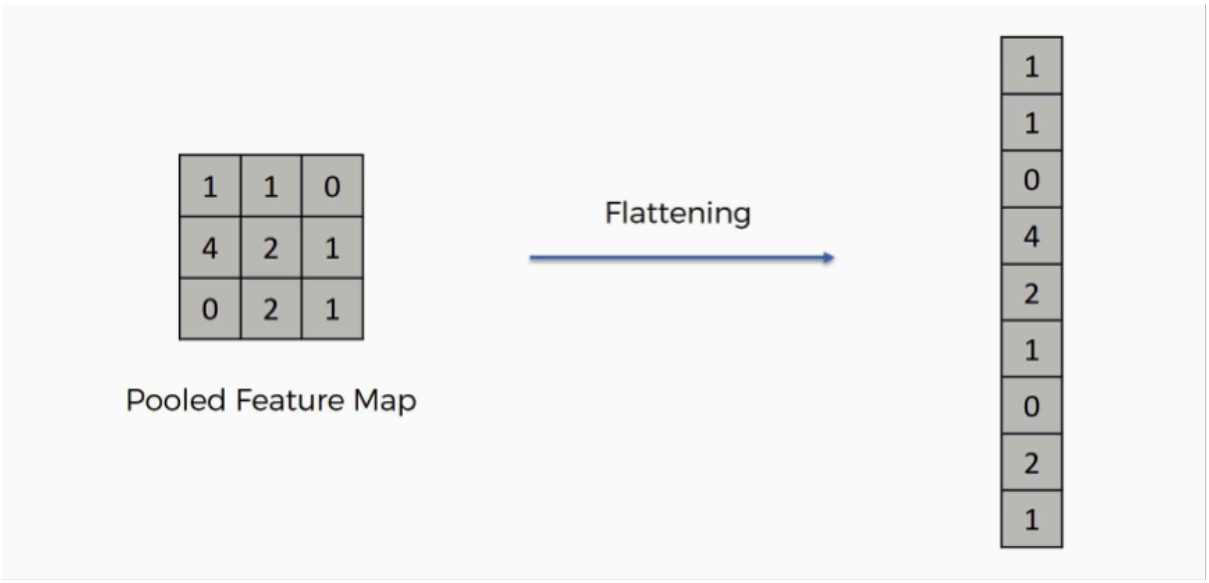


Max Pooling Operation with a 2x2 Filter and Stride (2, 2)

Flattening Layer

After several convolutional and pooling layers, the output feature maps are flattened into a 1D vector. This vector is then fed into fully connected layers, which are responsible for making the final classification or prediction.

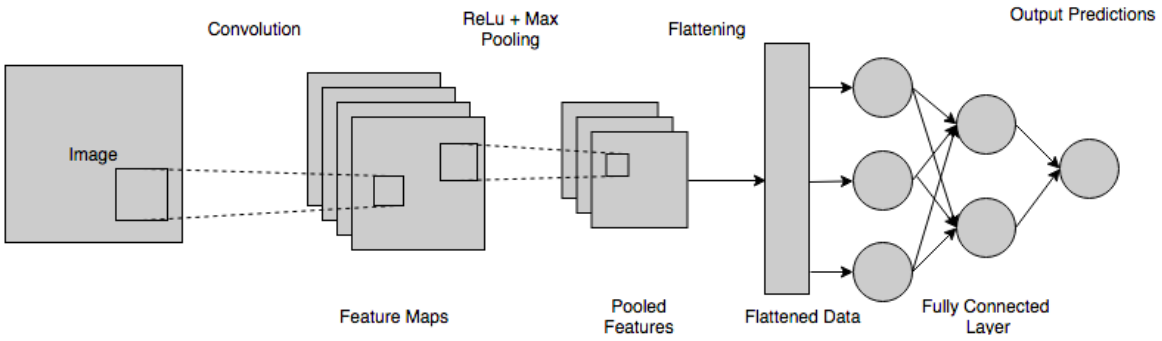
```
model.add(Flatten())
```



Flattening Layer

Fully Connected Layer

The fully connected layer is similar to a traditional neural network, where each neuron is connected to every other neuron in the next layer. These layers combine the features learned by previous layers to predict the class of the image.



Fully Connected Layer

Softmax Layer

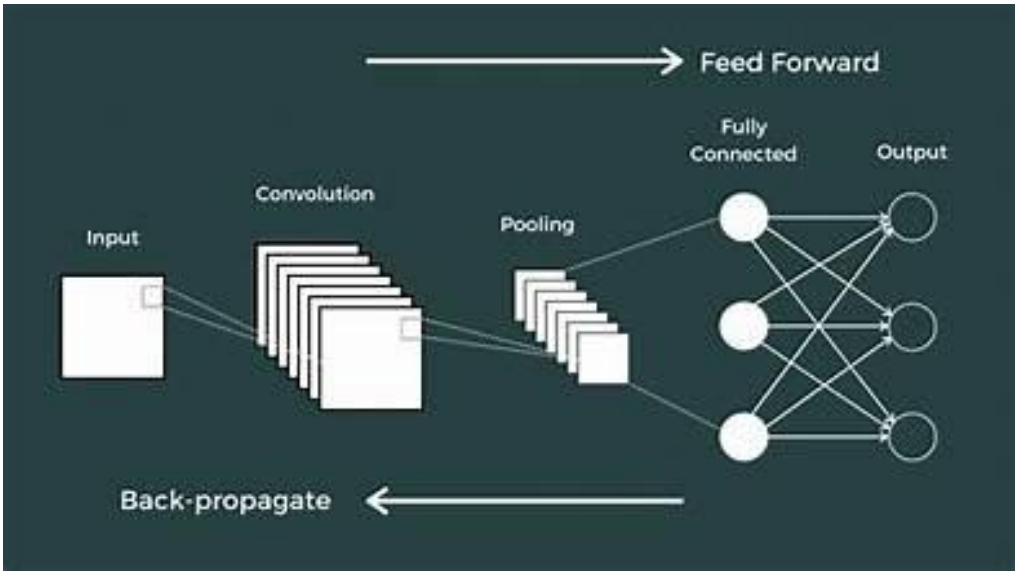
In classification tasks, the final output layer is usually a **softmax** layer, which converts the output from the fully connected layer into probabilities for each class.

```
num_classes = len(categories)
model.add(Dense(num_classes, activation='softmax'))
```

Training CNNs

- CNNs are trained using **backpropagation** and **gradient descent**. The network learns by adjusting the weights of the filters based on the errors between the predicted output and the actual output.
- A loss function, like **categorical cross-entropy**, measures how far off the prediction is, and an optimizer, like **Adam**, updates the weights to minimize this loss.

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



Convolutional Neural Network (CNN)

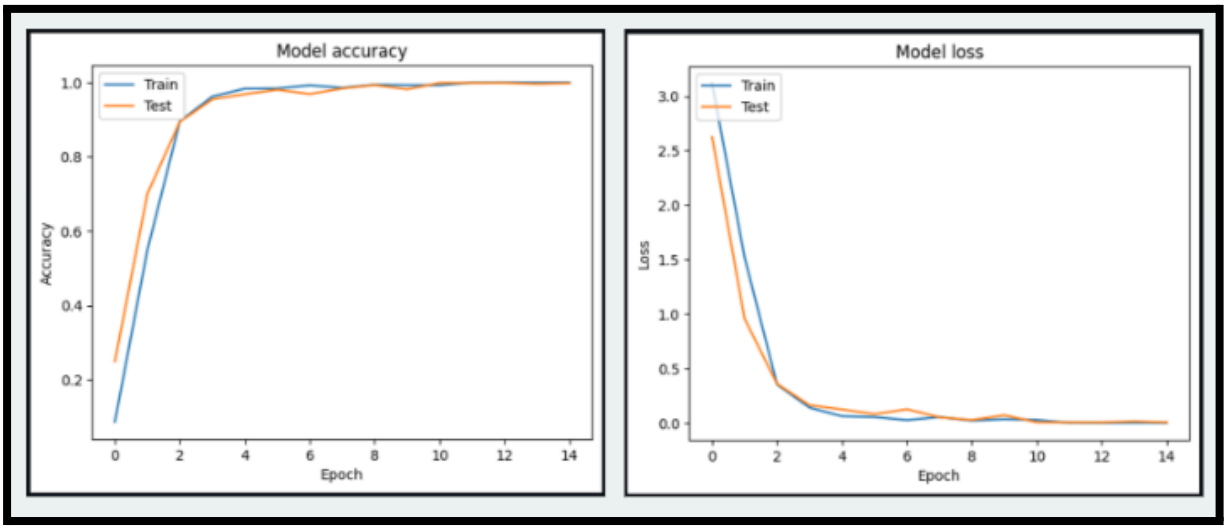
Testing

As for the testing we go with the following three methods:

1.Test The Model With Test-set(20%)
2.Test Detection On Images
3.Test Real Time Detection

1.Test The Model With Test-set(20%)

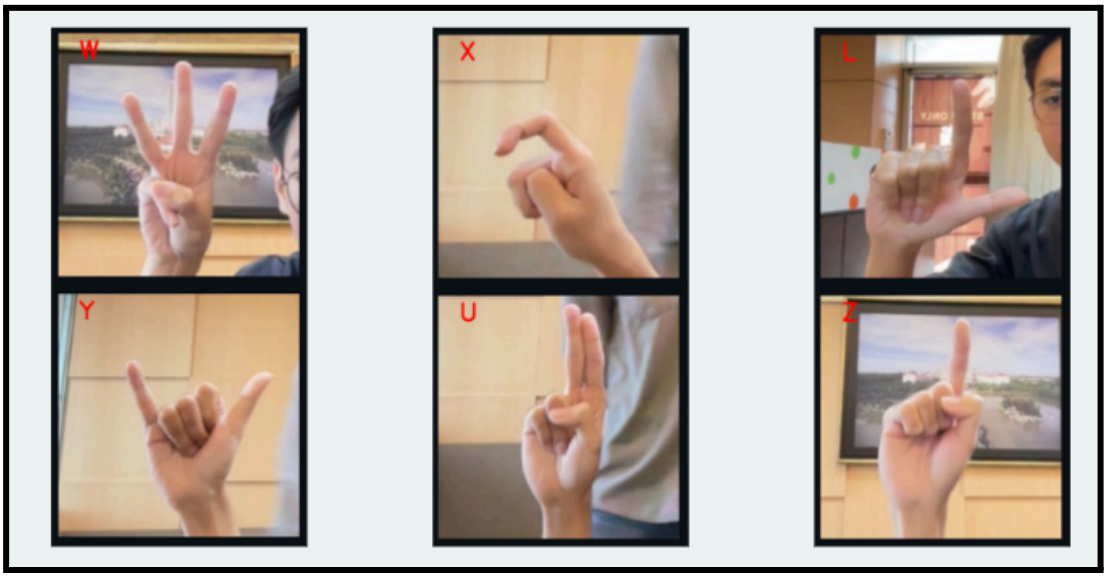
We evaluated the model's performance by splitting the dataset into 80% for training and 20% for testing. After training, we used the evaluate() function to calculate the test accuracy and loss on the 20% test set (x_test, y_test). We then used the predict() function to generate predictions on the test data and compared the predicted classes with the true labels. A classification report, including precision, recall, and F1-score, was generated using classification_report(), and a confusion matrix was printed to visualize prediction errors. Finally, the trained model was saved as asl.h5 for future use.



Result Of Test The Model With Test-set(20%)

2.Test Detection On Images

As for the images detection we prepared a folder which contained images of each hand sign and created the opencv program which linked with the train ASL model (asl.h5) to make detection on each image which contained in that test file.We created the program to detect and show output of the predicted image with the predicted alphabet at the corner of each image.



Result Of Test detection on images

3.Test Real Time Detection

For real-time detection, we used Mediapipe to track hand landmarks and OpenCV to capture video. The program detects the hand, crops the region, processes it to match the input size for the trained ASL model (asl.h5), and predicts the corresponding sign. The predicted letter is displayed in real-time at the bottom of the frame, making the process simple and user-friendly for beginners. Also the accuracy of the prediction is shown for evaluation.

Result Of Real Time Detection-1

Result Of Real Time Detection-2

Conclusion

In conclusion, we successfully implemented a deep learning model for ASL hand sign detection and evaluated its performance using an 80-20 train-test split. The model achieved a test accuracy that reflects its ability to generalize to unseen data. By analyzing the classification report and confusion matrix, we gained insight into the model's strengths and weaknesses in predicting specific hand signs. The trained model was saved for real-time applications, enabling continuous improvements and adjustments. Overall, the model demonstrates promising results for ASL detection, providing a solid foundation for further development.

References

https://teachablemachine.withgoogle.com

https://www.researchgate.net/publication/364185120_Real-Time_Sign_Language_Detection_Using_CNN

https://opencv.org/