Project File

Hand Gesture Recognition

Ankur Bohra (12 E), Mihir Sahu (12 E)

16th February, 2022

Certificate of Originality

This is to certify that this project documentation paper submitted by us, **Ankur Bohra** and **Mihir Sahu**, is an outcome of our independent and original work. We have duly acknowledged all the sources from which the ideas and extracts have been taken. The project is free from any plagiarism and has not been submitted elsewhere for publication.

Names of authors:

Ankur Bohra -

Mihir Sahu -

Date: 16th February, 2022

Signatures:

Name of teacher: Gulroop Kaur

Date: 16th February, 2022

Signature:

Index

Overview

This project aims to use machine learning for classification of American Sign Language (ASL) gestures from images.

Scope

The project trains a Convolutional Neural Network (CNN) model on a publicly-available dataset, and uses the trained parameters to classify images of the ASL during runtime. This program can be further customised to create various kinds of interactions between the user and the device via the sign language.

We also explore the important stages of proper retrieval, normalisation of data, and tuning a model to get the best results.

Image Classification

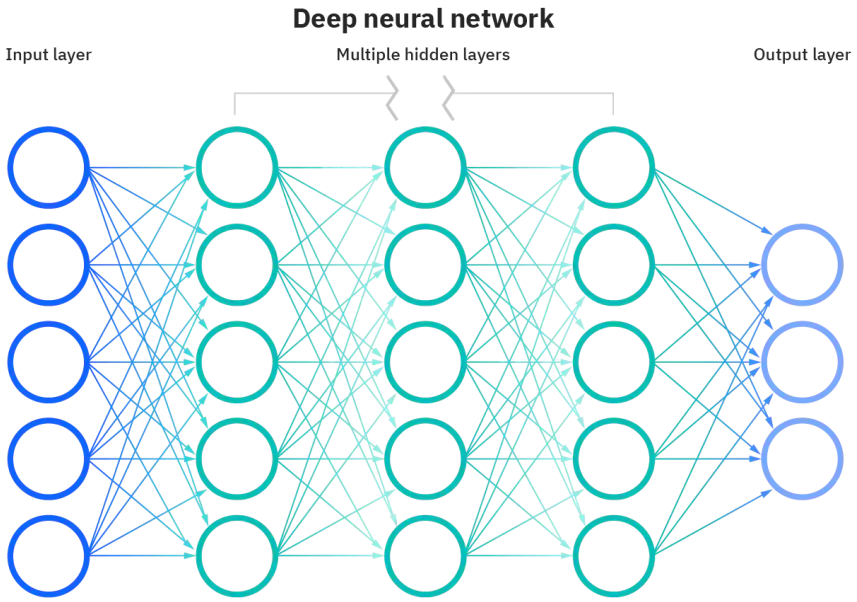
The project depends heavily on image classification. Image classification is the process of labelling and classifying images based on patterns in the pixels of the image. We use supervised learning to train our model.

ANN

An artificial neural network consists of layers of interlinked nodes called (artificial) neurons. Each neuron has input(s) and an output, each input and output is a fraction representing the *activation* of the source neuron. The output of a neuron is mathematically linked to its inputs. Each input does not have an equal effect on the output, any connection between two neurons has a *weight*, a parameter that controls the influence of a particular input node on the

connected node's output. These weights are trained to minimise a *loss function*, which evaluates how accurate the model is. This minimization is facilitated by an optimizer, which controls how loss affects the modifications of weights. In short, the loss function evaluates accuracy while an optimizer modified weights according to the evaluation.

The densely connected neurons and weights together allow the network to learn features by assigning importance to certain neurons hence allowing it to recognize patterns.



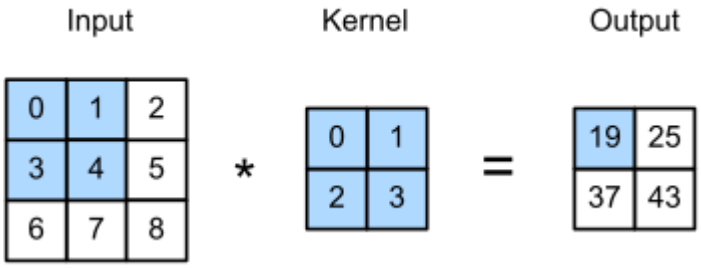
CNN

A convolutional neural network, used commonly for classifying images, functions by assigning importance to patterns in its input images. In this aspect, a CNN interprets data similar to humans, by finding important features and grouping them again and again to reliably "understand" or classify it.

For finding such features, a convolutional neural network applies a filter (just one kernel for a 2D matrix) over areas of the image, and calculates an

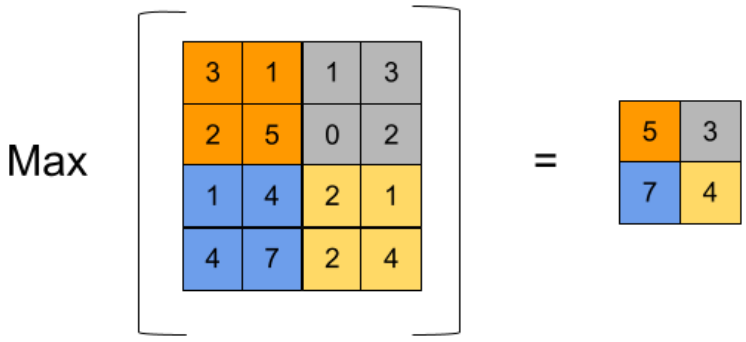
activation for each cell which indicates the position and strength of a feature. By training the parameters of this filter, a CNN is able to identify features of an image, further grouped by applying other filters.

The operation of applying a filter is called a convolution, the result of which is called a feature map. This matrix is calculated by multiplying the neighbouring cells of the required cell with the corresponding cells of the filter, and summing them. This operation is repeated for each cell, resulting in the complete feature map.



Clearly a CNN involves a lot of operations to output a single feature map. The number of operations can be reduced by downsampling the image/matrix. This is done by either changing the *stride* of the convolution i.e. how many cells to skip each time the filter is applied, or by *pooling,* which allows reduction of trainable parameters by grouping regions of the feature map and considering the maximum/average/etc of their activations. As a benefit, pooling also makes the classification local-translation invariant i.e. small shifts in pixels do not throw the model off.

An example of a max pooling operation:



Modules

TensorFlow

We will use TensorFlow (tensorflow on PyPi), Google's machine learning library to build and train our model. TensorFlow allows us to shift our focus to choosing the best composition and parameters of our model to maximise its accuracy, instead of implementing complicated algorithms from scratch. TensorFlow also serves as our primary resource for the machine learning research of our project.

OpenCV

We will use OpenCV, a computer vision library, to retrieve and direct images towards our TensorFlow model for prediction, process images during training and testing, and to show the user the predicted alphabets.

tkinter

We will use tkinter to build our user interface, which allows the user to tweak the actions invoked by specific hand gestures, and modify some parameters to allow better predictions.

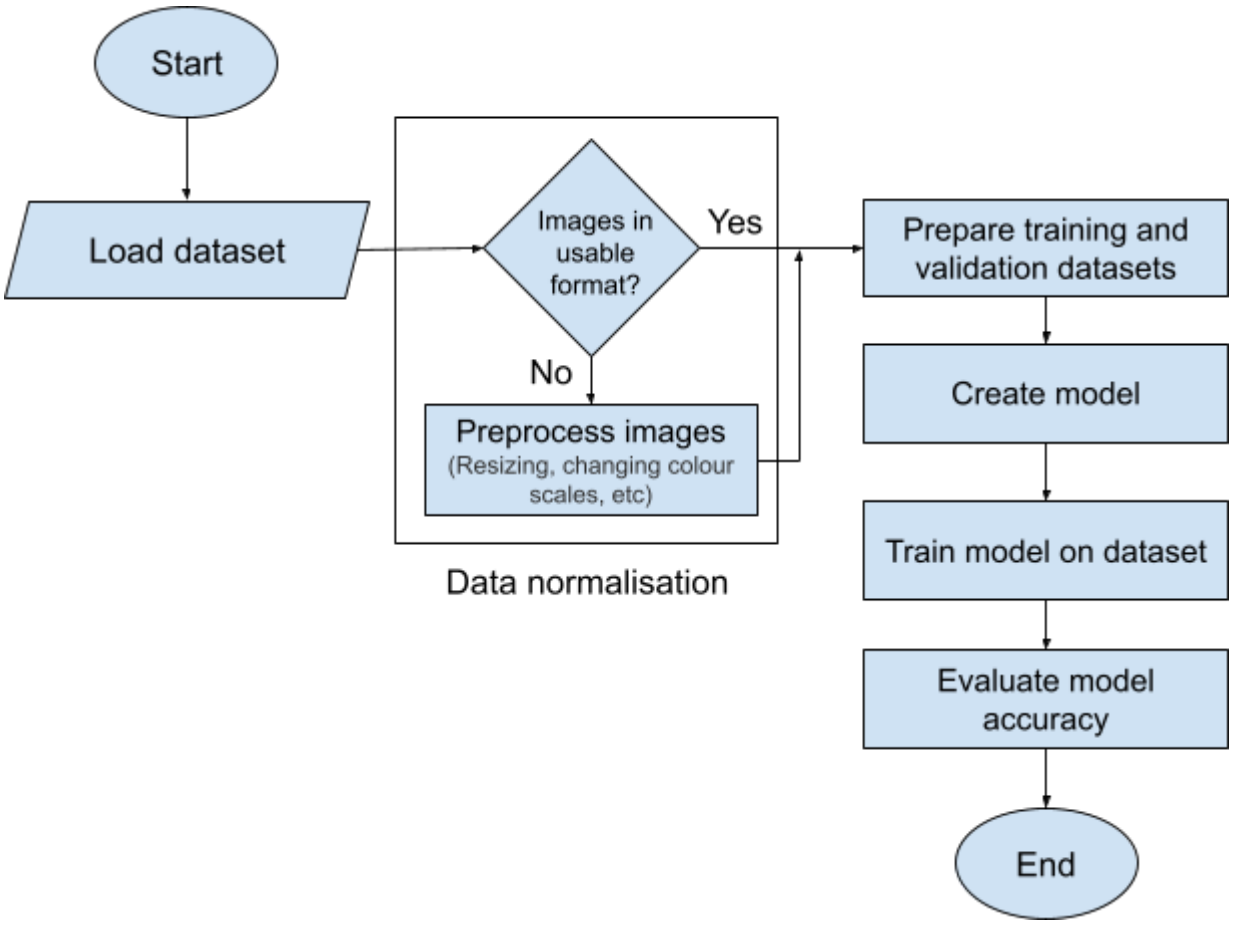
MediaPipe

We use MediaPipe, Google's framework containing ready-to-use ML solutions to locate hands from the camera feed.

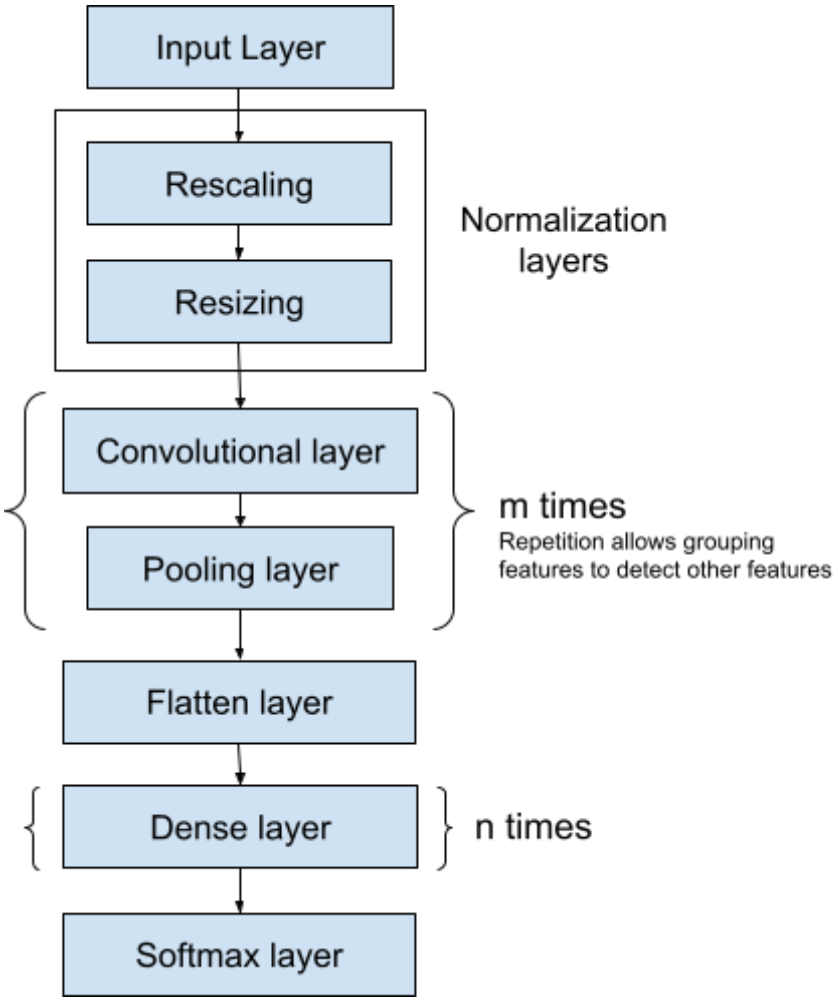
Proposed Working

We propose the following frameworks for the different aspects of our project, however we expect slight deviation from the proposal as we progress in project completion.

Training

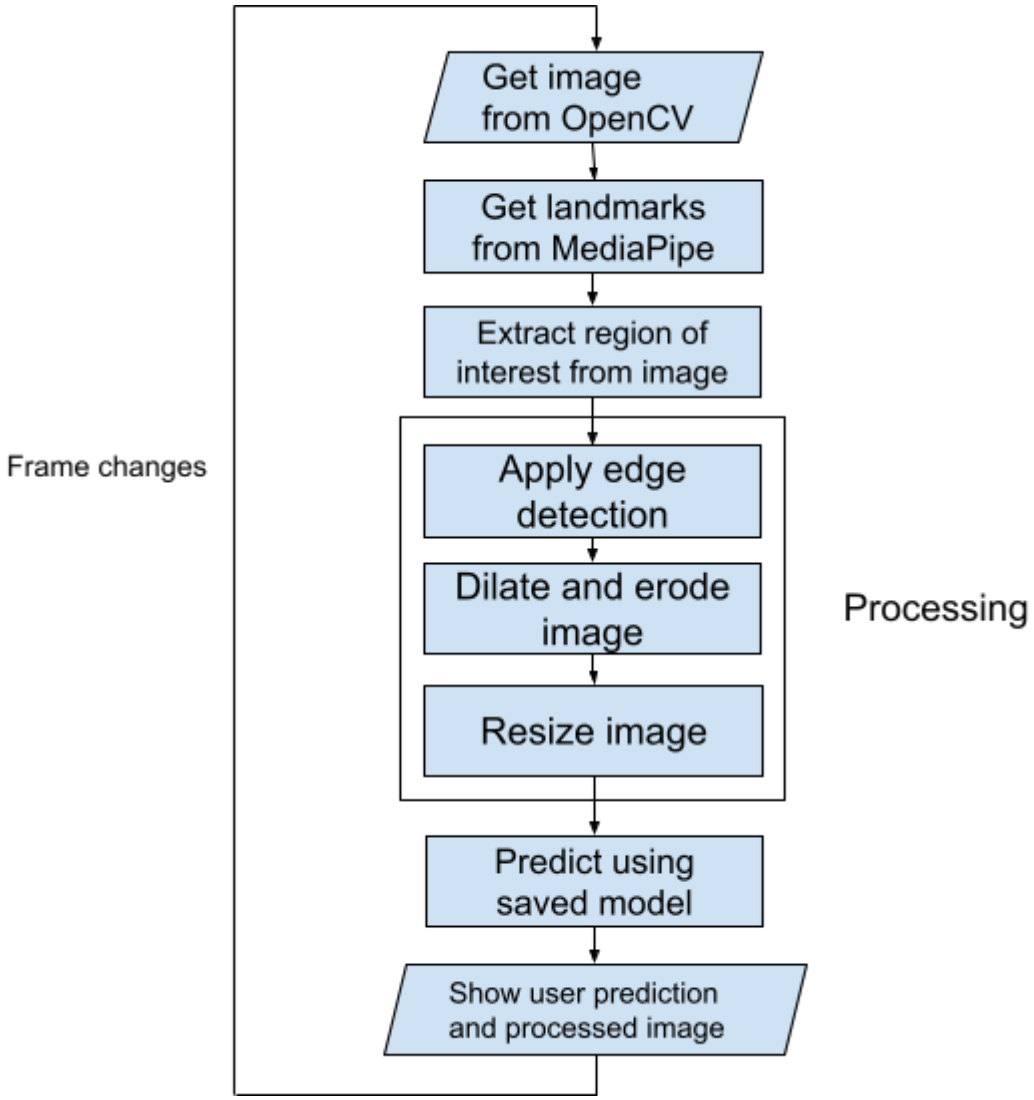


Model


Input Layer
Rescaling
Resizing
Normalization layers
Convolutional layer
Pooling layer
m times
Repetition allows grouping features to detect other features
Flatten layer
Dense layer
n times
Softmax layer

Interface

A window with the live camera feed of the user, along with the current prediction is always shown to the user. The interface also shows the user what part of the image is fed into the model, and how it looks after processing.



Source Code

Model Training

We make use of Google Colab and Amazon SageMaker Studio Lab to develop and train our CNN model in an interactive environment. Our dataset is sourced from grassknoted/asl-alphabet in Kaggle.

```
! conda install glib=2.51.0 -y
! pip install kaggle
! pip install matplotlib
! pip install tensorflow_datasets

import glob
import math
import os
import random
import subprocess

from pathlib import Path

import cv2
import matplotlib.pyplot as plt
import numpy
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPool2D,
Softmax
from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical

# Constants
IMAGE_SIZE = (100, 100)
VAL_SIZE = 0.2
BATCH_SIZE = 16
AUTOTUNE = tf.data.AUTOTUNE

# Place kaggle token
home = Path.home()
if not home.joinpath(".kaggle").exists():
  os.mkdir(str(home.joinpath(".kaggle")))  # Make /root/.kaggle
```

```
if home.joinpath("kaggle.json").exists():
  os.rename(str(home.joinpath("../content/kaggle.json")),
str(home.joinpath(".kaggle/kaggle.json")))  # Move /content/kaggle.json to
/root/.kaggle/

# Install dataset
if not 'data_dir' in globals():
  download_manager = tfds.download.DownloadManager(
    download_dir=str(home.joinpath(".tensorflow/datasets")),
    force_extraction=True  # Unzip the dataset
  )
  data_dir =
download_manager.download_kaggle_data("grassknoted/asl-alphabet").joinpath("asl_
alphabet_train/asl_alphabet_train")

# Preprocess images
def process_image(image):
  image = cv2.Canny(image, 80, 90)
  image = cv2.dilate(image, None)
  image = cv2.erode(image, None)
  image = cv2.resize(image, IMAGE_SIZE)
  return image

classes_folders = sorted(glob.glob(str(data_dir)+"/*"))
images, labels = [], []
class_to_index_map = {}
index_to_class_map = {}
for folder in classes_folders:  # /A, /del, etc
  class_name = tf.strings.split(folder, "/")[-1].ref()
  class_index = len(index_to_class_map)
  index_to_class_map[class_index] = class_name
  class_to_index_map[class_name] = class_index
  for filename in glob.glob(folder+"/*.jpg"):
    image = cv2.imread(filename, 0)
    image = process_image(image)
    image = numpy.array(image)
    image = tf.convert_to_tensor(image, dtype=tf.float32)
    label = tf.convert_to_tensor(class_index, dtype=tf.int32)
    images.append(image)
    labels.append(label)

# Prepare dataset
dataset = tf.data.Dataset.from_tensor_slices((images, labels))
dataset = dataset.shuffle(len(labels))

train_ds = dataset.skip(math.floor(VAL_SIZE * len(dataset)))
val_ds = dataset.take(math.floor(VAL_SIZE * len(dataset)))

train_ds = train_ds.batch(BATCH_SIZE).cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.batch(BATCH_SIZE).cache().prefetch(buffer_size=AUTOTUNE)
```

```
# View some images
plt.figure(figsize=(10, 10))
for _images, _labels in train_ds.take(3):
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(tf.squeeze(_images[i]).numpy().astype("uint8"), cmap="gray")
    plt.axis("off")

# Create the model
model = Sequential()
model.add(Conv2D(64, (3, 3), input_shape=IMAGE_SIZE + (1,), activation='relu'))
# 64 filters with size 3x3 over a 64x64 image of 1 channel
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPool2D())  # Defaults to (2,2)
model.add(Conv2D(128, (3,3), activation='relu'))

model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(29, activation='relu'))  # 26 alphabets + del, space, nothing
model.add(Softmax())  # Convert to probabilities

print(model.summary())

# Compile and train the model
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3,
verbose=1)
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model.compile(
       optimizer=optimizer,
       loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
       metrics=['accuracy']
)
history = model.fit(train_ds, validation_data=val_ds, epochs=10,
callbacks=[early_stopping])

# Evaluate model
plt.plot(history.history.get("accuracy"), label="accuracy")
plt.plot(history.history.get("val_accuracy"), label="val_accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.ylim([0, 1])
plt.legend(loc="lower right")
plt.show()
```

Camera Connectivity

We make use of the openCV module and Google's MediaPipe to make a camera connectivity model to integrate the above model and run the programme in real time.

```
import cv2
import mediapipe as mp
import numpy as np
import tensorflow as tf
from tensorflow import keras

mp_hands = mp.solutions.hands

IMAGE_SIZE = (100, 100)

model = keras.models.load_model('Hand_Gesture_Model.h5')
classes = [chr(num) for num in range(ord('A'), ord('Z')+1)]  # List of all
(capital) alphabets
classes.extend(("d", "n", "s"))  # d -> delete; n -> nothing; s -> space

def process_image(image):
    '''Processes the image to be fed into the model.

    Args:
        image: The image to be processed.

    Returns:
        The processed image.
    '''
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    image = cv2.Canny(image, 80, 90)  # Edge detection
    image = cv2.dilate(image, None)  # Dilation and erosion
    image = cv2.erode(image, None)   # clears up noise
    image = cv2.resize(image, IMAGE_SIZE)  # Resize to the trained image size
    return image

def get_roi_coords(landmarks, padding=25):
    '''Finds the square region-of-interest coordinates from the hand landmarks.

    Args:
        landmarks: The hand landmarks.
        padding: The padding to be added to the bounding box before translation
to roi.

    Returns:
        The roi coordinates.
    '''
```

```
    for hand_landmarks in results.multi_hand_landmarks:
        x_values = [landmark.x for landmark in hand_landmarks.landmark]
        y_values = [landmark.y for landmark in hand_landmarks.landmark]
        x_min = min(x_values)
        x_max = max(x_values)
        y_min = min(y_values)
        y_max = max(y_values)

    # Scale to actual dimensions
    x_min = int(x_min * width)
    x_max = int(x_max * width)
    y_min = int(y_min * height)
    y_max = int(y_max * height)

    # Apply padding while respecting bounds
    # Minimum values should be decreased during padding, but >= 0
    # Maximum values should be increased but <= image width/height
    x_min = max(x_min - padding, 0)
    x_max = min(x_max + padding, width)
    y_min = max(y_min - padding, 0)
    y_max = min(y_max + padding, height)

    # Expand bounding box to square roi
    difference = abs((x_max  - x_min) - (y_max - y_min))
    if x_max - x_min > y_max - y_min:
        # Need to expand the height of the bounding box.
        # Expand the bounding box equally on both sides.
        y_min -= difference // 2
        y_max += difference // 2
        if y_min < 0:
               # The expansion has caused the box to cross the top edge
               # Shift bounding box downwards
               delta = -y_min
               y_min += delta
               y_max += delta
        elif y_max > height:
               # Box has crossed bottom edge, shift bounding box upwards
               delta = y_max-height
               y_min -= delta
               y_max -= delta
    elif x_max - x_min < y_max - y_min:
        # Need to expand the width of the bounding box.
        x_min -= difference // 2
        x_max += difference // 2
        if x_min < 0:
               delta = -x_min
               x_min += delta
               x_max += delta
        elif x_max > width:
               delta = x_max-width
```

```
                x_min -= delta
                x_max -= delta

    return (x_min, y_min, x_max, y_max)

def predict_class(roi):
    '''Predicts the class of the hand gesture in the image roi.

    Args:
        roi: The region-of-interest image.

    Returns:
        The class of the hand gesture.
    '''
    roi = np.expand_dims(roi, axis=0)  # Add the channel dimension as is in the
model
    probabilities = model(roi).numpy()[0]  # Convert tensor result to python
object
    prediction = classes[np.argmax(probabilities)]
    return prediction, probabilities

cap = cv2.VideoCapture(0)
_, frame = cap.read()
height, width, channels = frame.shape
with mp_hands.Hands(
        model_complexity=0,  # Less latency with lower complexity
        max_num_hands=1,
        min_detection_confidence=0.5,
        min_tracking_confidence=0.5) as hands:
    while cap.isOpened():
        success, image = cap.read()

        if not success:
                continue

        # To improve performance, optionally mark the image as not writeable to
        # pass by reference.
        image.flags.writeable = False
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # cv2 operates in BGR,
mediapipe in RGB
        results = hands.process(image)
        image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
        if results.multi_hand_landmarks:
                # Process roi and predict
                x_min, y_min, x_max, y_max =
get_roi_coords(results.multi_hand_landmarks)
                roi = image[y_min:y_max, x_min:x_max]  # Get roi from image
                roi = process_image(roi)
                prediction, probabilities = predict_class(roi)
```

```
                # Display prediction and roi
                roi = cv2.cvtColor(roi, cv2.COLOR_GRAY2BGR)
                image[y_min:y_max, x_min:x_max] = cv2.resize(roi, (x_max - x_min,
y_max - y_min))  # Paste the roi back into the image
                cv2.rectangle(image, (x_min, y_min), (x_max, y_max), (0, 0, 0), 2)
                image = cv2.flip(image, 1)  # Flip *before* the text is applied
                cv2.putText(image, prediction, (width-x_max, y_max),
cv2.FONT_HERSHEY_DUPLEX, 0.5, (0, 255, 0), 1)
        else:
                image = cv2.flip(image, 1)  # Flip for a selfie-view display

        cv2.imshow('Hand Gesture Recognition', image)
        if cv2.waitKey(5)== ord('q'):
                break
cap.release()
```

Output and Methodology

Model Training

We conclude training with the following observations on some core parameters:

1. **Optimizer:** We find that our model's accuracy stagnates when we use the ADAM optimizer. Although SGD too stagnates for the most part, it is found that the fine tuning of SGD's learning rate solves stagnation of accuracy. We attribute this benefit of SGD over ADAM to SGD's better generalisation, the explanation of which is out of the domain of our project.

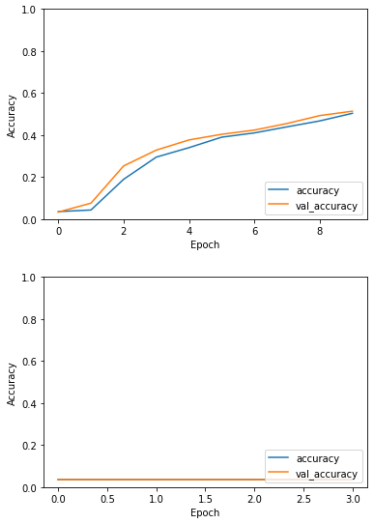
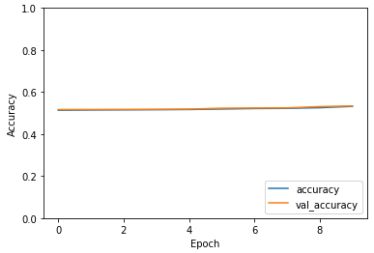
   We also come across the effects of the stochasticity of SGD (Stochastic Gradient Descent), where certain runs result in abnormally high accuracies (see S.No. 1 and 3 below) that are not reproducible. Interestingly these abnormalities tend to have dissimilar accuracies when applied to actual testing data.

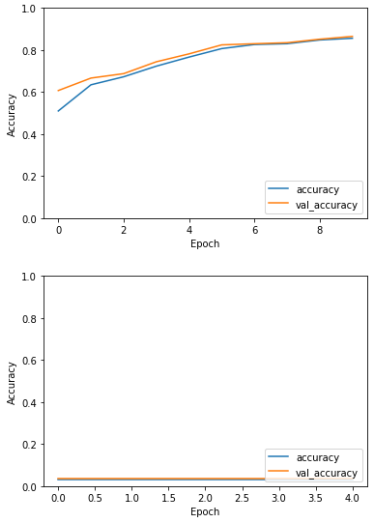
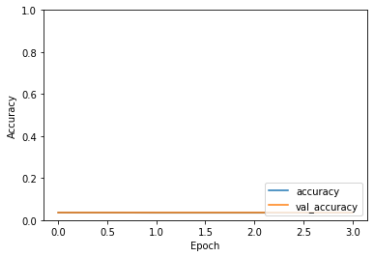
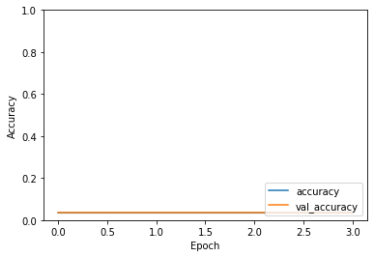
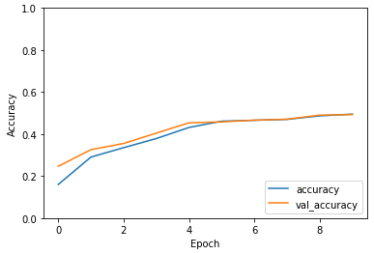
2. **Learning rate:** Learning rate seems to be the most impactful parameters of the training procedure. A learning rate that is too low or too high causes the model to be as accurate as a random guess (~0.034 accuracy), while a moderate learning rate sees noticeable improvement as epochs pass. This is expected as the learning rate controls how quickly loss is minimised by scaling changes in weights. If the loss is minimised too slowly, we may not be able to reach an optimal solution. If the loss is minimised too fast, we may reach a suboptimal solution.

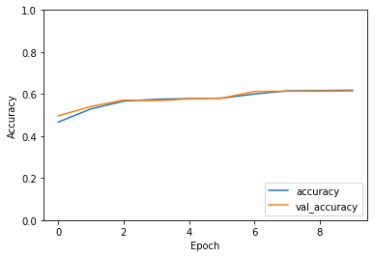
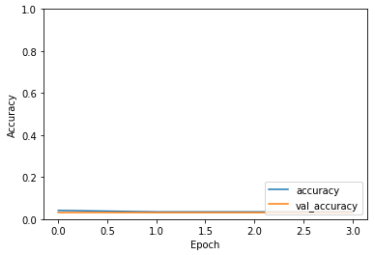
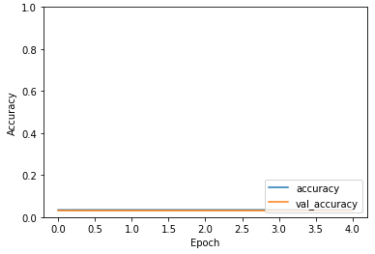
3. **Model complexity:** We observe that adding even a single pair of convolutional-pooling layers causes accuracy to stagnate. The large increase in the amount of parameters to train can deem the model unable to find a suitable solution with the limited amount of data it is given.

4. **Processing:** We observe that the removal of all processing from the dataset causes accuracy to be nearly equal to that of a random guess. This is expected as the dataset's images are grainy and the background is not especially plain.

Following is a table showing some recorded training sessions:

S. No.	Model Layers	Notes	Accuracy Plot
1.	Conv2D(64, (3, 3), 'relu') Conv2D(64, (3, 3), 'relu') MaxPool2D() Conv2D(128, (3,3), 'relu') Dropout(0.2) Flatten() Dense(64, 'relu') Dense(29, 'relu') Softmax()	Image size: (100, 100) Batch size: 16 Processing: Canny, Dilate, Erode, Resize Optimizer: SGD **(LR=0.001)**	
2.	Conv2D(64, (3, 3), 'relu') Conv2D(64, (3, 3), 'relu') MaxPool2D() Conv2D(128, (3,3), 'relu') Dropout(0.2) Flatten() Dense(64, 'relu') Dense(29, 'relu') Softmax()	Image size: (100, 100) Batch size: 16 Processing: Canny, Dilate, Erode, Resize Optimizer: SGD **(LR=0.0001)**	

3.	Same as above	Image size: (100, 100) Batch size: 16 Processing: Canny, Dilate, Erode, Resize Optimizer: SGD **(LR=0.01)**	
4.	Same as above	Image size: (100, 100) Batch size: 16 Processing: Canny, Dilate, Erode, Resize Optimizer: SGD **(LR=0.1)**	
5.	Same as above	Image size: (100, 100) Batch size: 16 Processing: Canny, Dilate, Erode, Resize Optimizer: **ADAM**	
6.*	Conv2D(64, (3, 3), 'relu') Conv2D(64, (3, 3), 'relu') MaxPool2D() Conv2D(128, (3,3), 'relu') Dropout(**0.5**) Flatten() Dense(64, 'relu') Dense(29, 'relu') Softmax()	Image size: (100, 100) Batch size: 16 Processing: Canny, Dilate, Erode, Resize Optimizer: SGD (LR=0.001)	

7.	Same as above	Image size: (100, 100) Batch size: 16 Processing: None Optimizer: SGD (**LR=0.01**)	
8.	Same as above	Image size: (100, 100) Batch size: 16 Processing: None Optimizer: SGD (**LR=0.1**)	
9.	Conv2D(64, (3, 3), 'relu') Conv2D(64, (3, 3), 'relu') MaxPool2D() Conv2D(128, (3,3), 'relu') **MaxPool2D()** **Conv2D(256, (3,3), 'relu')** Dropout(0.5) Flatten() Dense(64, 'relu') Dense(29, 'relu') Softmax()	Image size: (100, 100) Batch size: 16 Processing: Canny, Dilate, Erode, Resize Optimizer: SGD (LR=0.01)	

*Our final model has come from this training session.

Camera Connectivity

The processing of raw images to resemble the format of the training data is done through the following steps:

1. Recognizing the hand:

   As cameras capture a large part of the background and having just the hand visible in the camera is an impractical solution, we use MediaPipe to locate the landmarks of any hand in the camera. This gives us the positions of certain joints in the image (see Image 1).

   These landmarks also present an alternative method of recognizing gestures possibly without the use of images, by training for relative positions of landmarks. This is however out of the scope of this project.

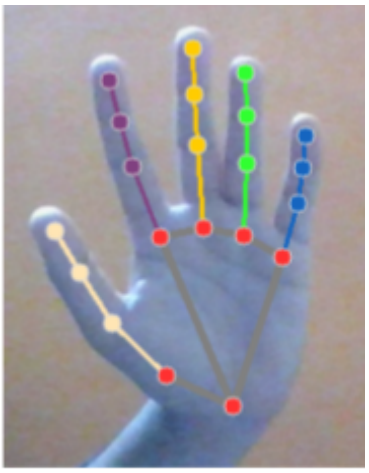


Image 1: Hand landmarks using the MediaPipe hands tutorial code

2. Creating a bounding box and padding:

   Using the landmarks, we can create a bounding box that contains all landmarks by finding the minimum and maximum values of their coordinates. The bounding box so created does not contain complete fingers (see Image 2 red box). To solve this problem, we apply some padding to the bounding box and expand it from all sides (see Image 2 green box). The padding is arbitrarily chosen and may need to be modified for varying distances from the camera.

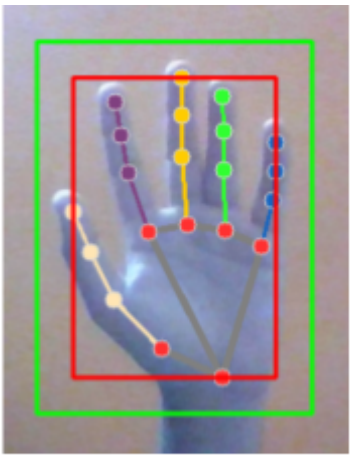


Image 2: Inner bounding box (red) and padded bounding box (green)

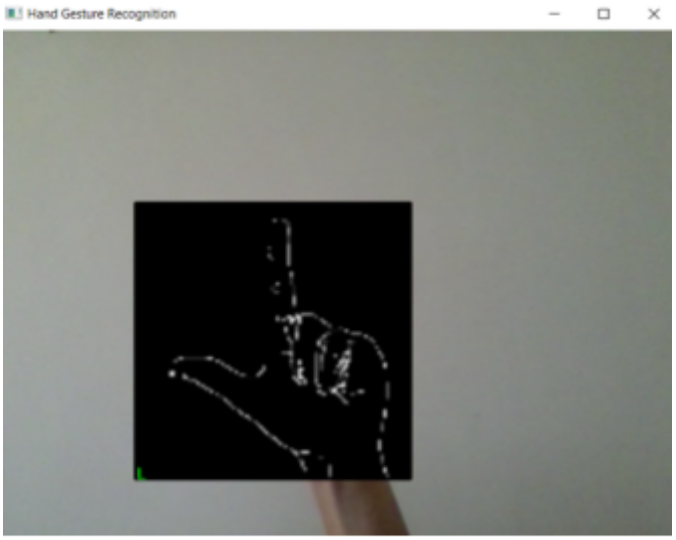
3. Creating the region of interest / Squaring the padded box:

As our model takes a square image as an input, we must expand the bounding box along the required axis to create a square region we call the region of interest (see Image 3 blue box). It is this region that we finally extract from the image to apply edge detection etc. and feed into the model for predictions.

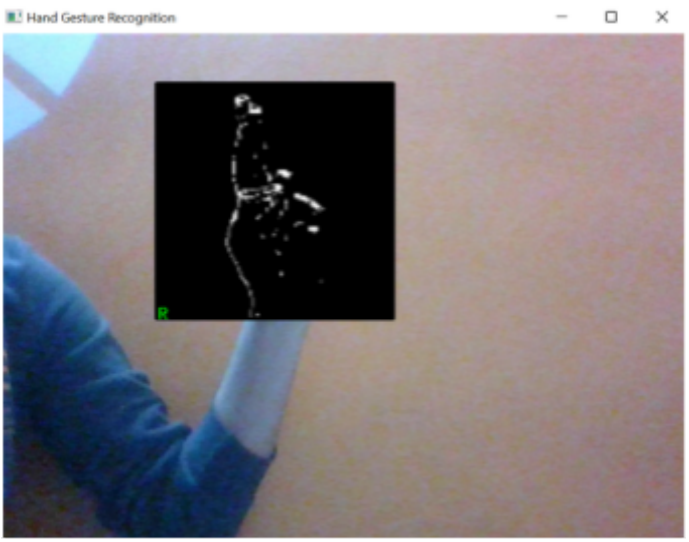
Image 3: Bounding box (green) and region of interest (blue)

A recording of the application is available using the QR code or the following link: https://bit.ly/36a774O

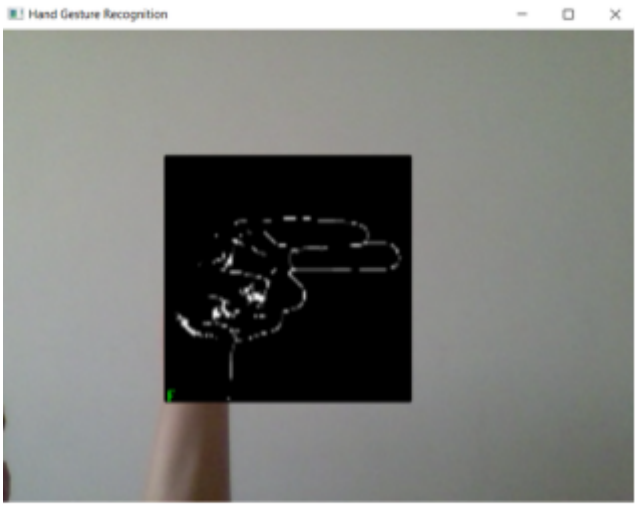
Following are some labelled screenshots of the program as seen by the user in real-time.



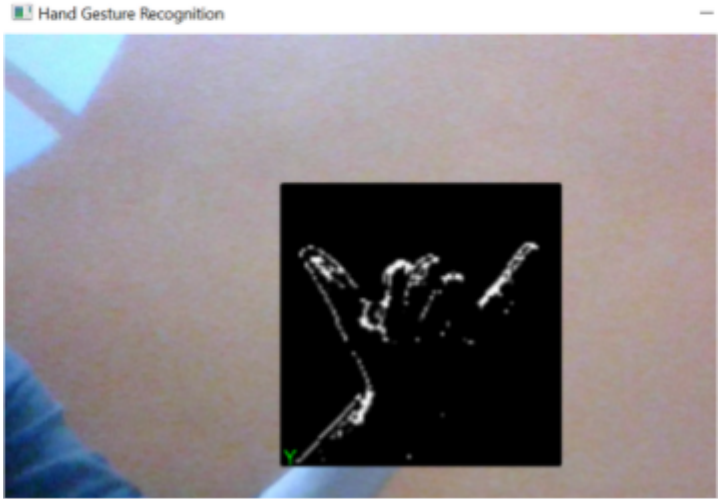
L correctly classified as L



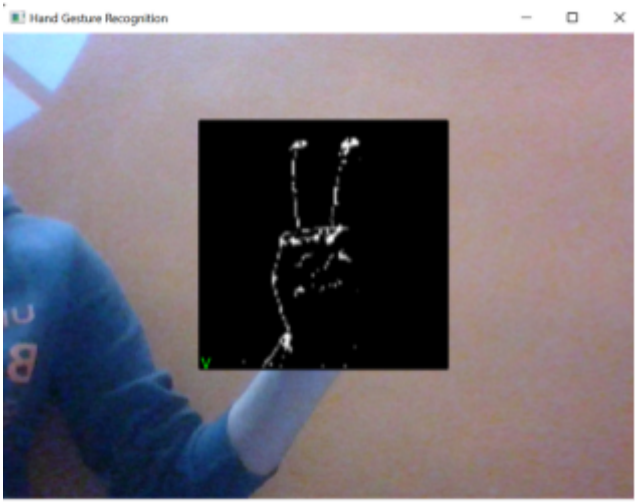
R correctly classified as R



H incorrectly classified as F



Y correctly classified as Y



V correctly classified as V

Bibliography

We have used the following websites while researching and programming for the project:

- https://www.tensorflow.org/
- https://towardsdatascience.com
- https://www.pyimagesearch.com
- https://google.github.io/mediapipe/
- https://opencv.org/
- https://stackoverflow.com/
- https://www.kaggle.com/grassknoted/asl-alphabet
- https://github.com/Mohamedyasserhelmy/Sign-Language-Translator-ASL
- https://github.com/Gogul09/gesture-recognition/