

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.2	Scope . . . . .	4
1.3	Overview . . . . .	4
1.4	Related Work . . . . .	4
<b>2</b>	<b>Sobel Filter</b>	<b>6</b>
2.1	Convolution . . . . .	6
2.1.1	Border Pixels . . . . .	7
2.1.2	Implementation . . . . .	7
2.2	Formulation . . . . .	8
2.2.1	Result . . . . .	9
<b>3</b>	<b>Neural Network</b>	<b>10</b>
3.1	Artificial Neurons . . . . .	10
3.2	Training . . . . .	12
3.2.1	Loss function . . . . .	12
3.2.2	Optimizer . . . . .	13
3.3	Implementation . . . . .	14
<b>4</b>	<b>OpenCV Application</b>	<b>17</b>
4.1	Implementation . . . . .	18
<b>5</b>	<b>Conclusion</b>	<b>20</b>
<b>6</b>	<b>Future Scope</b>	<b>21</b>

# List of Figures

1.1	Sequence Diagram of the application . . . . .	5
2.1	The three ways to handle boundaries . . . . .	7
2.2	Result of sobel operator . . . . .	9
3.1	An artificial neuron . . . . .	11
3.2	ReLU Activation Function . . . . .	12
3.3	Cross-Entropy loss function . . . . .	13
3.4	Choosing correct learning rate . . . . .	14
3.5	Accuracy of the model . . . . .	15
3.6	The confusion matrix . . . . .	16
4.1	The result of prediction . . . . .	17

# Chapter 1

## Introduction

Sign Languages are languages that use visual gestures for communication rather than spoken words. These languages are also considered natural languages, meaning they evolved over time according to the needs of the users of language. This means that they are very different from the verbal languages and have grammar and lexicon of their own. This makes communication between the users of sign language and verbal languages difficult. Thus there is a need for sign language recognition systems which can help bridge the barrier. With recent developments in computer vision there is room for models which are faster and easier to deploy on weaker hardware.

Sign Language Recognition using computers is a task with many obstacles. The high number of possible gestures means we need a recognition model which can distinguish between a myriad of classes. The gestures also look very similar which makes distinguishing them harder. Images tend to have a large number of features. But more features does not mean the model will give better predictions. To increase the accuracy of a recognition model, we need to extract features that are relevant to our predictions. Since hands are very complex, having a multitude of different skin types and tones; it is requisite to preprocess the input data to extract only necessary features.

### 1.1 Motivation

The motivation for this project stems from need for communication between users of sign language and verbal languages. This project aims to create a model which can give fast and accurate predictions. It should also be able to run on weaker hardware. Another goal was that the whole system is able to run on a single machine. The goal was to make the project more portable. This also means not using a server-client model; where client can send data to a server and a more powerful server can make predictions and send results back to client. Therefore, it is essential for the model and the user interface to be light weight. This project serves to show that a lightweight recognizer that can classify an image into one of a large number of classes and can be integrated into other applications is feasible with innovations in machine learning.

## 1.2 Scope

This project will focus specifically on the prediction of alphabets and digits of Indian Sign Language. A model that could translate the grammar and lexicon of the complete Indian Sign Language will need a recognizer, as well as a model which can translate gestures to written languages. Here, only the recognizer which can classify gestures into one of the thirty six classes (the alphabets and the digits). The project will also not focus on background removal and is trained on dataset with background removed. In a practical system, another method which could isolate the hands from rest of the image will be needed. The recognizer which is built in this project is also integrated with live video input using OpenCV for demonstrating the portability of the model. To get correct predictions a camera with high resolution and an undecorated background (ideally a single color) is recommended.

## 1.3 Overview

This project has three components. The first component is used to extract features from the input image. This is done using the Sobel–Feldman operator or Sobel filter. Edge detection allows us to effectively isolate the hands from the rest of the input image. It also helps in avoiding features that are not useful such as skin color. The second component is the model which makes the prediction. This is a simple feed-forward neural network which has takes a  $128 \times 128$  size matrix as an input and has 36 nodes (which represent the class of the gesture) as outputs. The final component is the openCV application which will take the video input using the webcam. This acts as the controller for the whole application. It will resize the input from the webcam, use the sobel filter, get the prediction using the model and finally output it to the user.

The user will interact with the OpenCV application, where they can see the video that is used as the input and it will also show the results of the prediction. The sequence diagram in Figure 1.1 shows how the application behaves during operation. A frame is taken from the video, which is resized to an  $128 \times 128$  size image. This image is taken in as an input by the Sobel Filter. The filter application will return data in form of CSV. This data can then be given to model. The prediction is then returned to the main application, which shows result to the user and takes the next frame to repeat the process.

## 1.4 Related Work

There have been multiple articles and efforts for Indian Sign Language Recognition. Some of the related works which have inspired this project and their results will be presented in this chapter.

- The first work is [6]. This paper does Indian Sign Language (ISL) recognition using Euclidean distance. The goal of this paper was to create a recognition system for Humanoid Robot Interaction (HRI), therefore it works on real-time video input. The platform was a made using JAVA software. This study managed to get a

recognition rate of 90%

- The second work is [5]. This paper used Histogram of Edge Frequency (HOEF) for feature selection. They took images as input, and used Support Vector Machine (SVM) for classification. They managed to get a recognition rate of 98.1%
- The third work is [4]. They used artificial neural network (ANN) implemented using Matlab in this paper. They worked on real-time video input. The recognition rate they got was 93%.

A summary of results from these different works is shown in table below

Work	Input	Classification	Recognition	Platform
[6]	Video	Euclidian Distance	90%	JAVA
[5]	Images	SVM	98.1%	N/A
[4]	Video	ANN	93%	Matlab

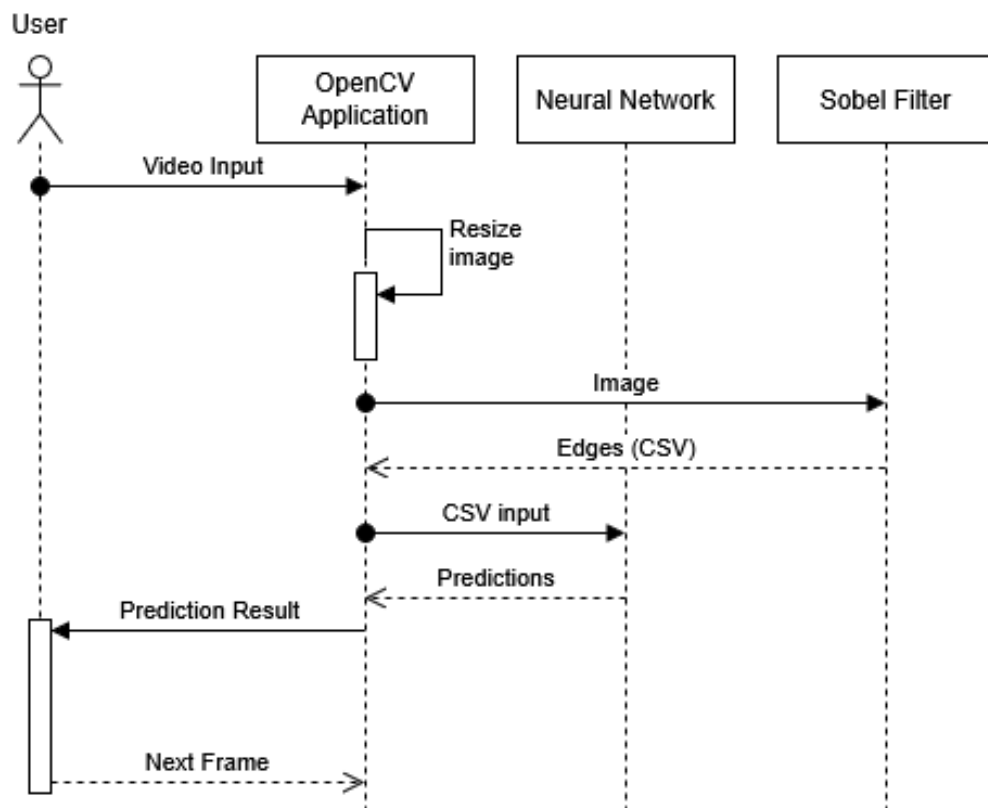


Figure 1.1: Sequence Diagram of the application

# Chapter 2

## Sobel Filter

The Sobel-Feldman operator (or simply Sobel filter) is a computer vision algorithm which can be used for edge detection. The filter takes a grayscale image as its input and create another image emphasising the edges. This project uses the sobel filter as described in [7]. The operator is named after Irwin Sobel and Gary M. Feldman, who introduced the idea in 1968 at Stanford Artificial Intelligence Laboratory (SAIL). This filter uses the convolution operator to extract the edges from image by the change in magnitude.

### 2.1 Convolution

Convolution is an operation which takes two matrices (an input and a kernel) and outputs another matrix which is of same size as the input matrix. It is the process of adding each element of the input to its local neighbors, weighted by the kernel. The convolution operation is denoted by \*. The operation usually uses a  $3 \times 3$  sized kernel, but it is not necessary. Convolution is a simple to implement and fast operation to implement. But it is also very useful allowing for multiple useful filters. Some of the most simple and useful ones are Gaussian Blur, Sharpening and Sobel Filter (which we are using).

This project only uses the  $3 \times 3$  sized kernels, this is done to make implementation simpler. This means in our case if the kernel is  $K$  and the input is  $I$ . Then the output  $O$  is given as

$$O[x,y] = \sum_{i=0}^2 \sum_{j=0}^2 I[x+i-1,y+j-1] \times K[i,j]$$

So we can have our Matrix class as follows

```
1 | class Matrix
2 | {
3 |     public:
4 |         int *data;
5 |         int width;
```

```

6 | int height;
7 | };

```

### 2.1.1 Border Pixels

But this function will not work for the pixels at the border of the input matrix. Since if we try to get the the value for border pixel  $[0, 0]$  we will need to get  $I[-1, -1]$ ,  $I[-1, 0]$ , etc. pixels which do not exist since there are no pixels to the left. There are three common ways to solve this problem

- Filling with zeros, so that pixels outside the input are zero
- Wrapping, the image is wrapped on all eight corners of the image
- Mirroring, we mirror the image along the perimeter of the image

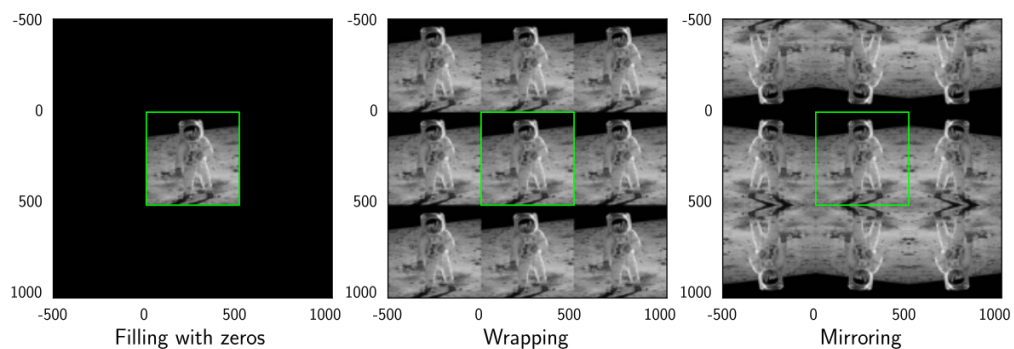


Figure 2.1: The three ways to handle boundaries

This project will fill with zeros if we need a pixel outside the boundry of the image.

```

1 | int get(int w, int h)
2 | {
3 |     if(w >= this->width || w < 0 || h >= this->height || h < 0)
4 |         return 0;
5 |     return this->data[h * this->width + w];
6 | }

```

### 2.1.2 Implementation

Now that we know the definition of convolution and how to handle the edge cases, we can implement the this operation in code.

```

1 | Matrix convolute(Matrix kernel)
2 | {
3 |     uint8_t *opt = new uint8_t[this->width * this->height];
4 |     Matrix output(opt, this->width, this->height);
5 |

```

```

6   for(int w = 0; w < this->width; w++)
7       {
8           for(int h = 0; h < this->height; h++)
9               {
10                  int value =
11                      kernel.get(0, 0) * this->get(w - 1, h - 1) +
12                      kernel.get(0, 1) * this->get(w - 1, h - 0) +
13                      kernel.get(1, 0) * this->get(w - 0, h - 1) +
14                      kernel.get(1, 1) * this->get(w - 0, h - 0) +
15                      kernel.get(0, 2) * this->get(w - 1, h + 1) +
16                      kernel.get(1, 2) * this->get(w - 0, h + 1) +
17                      kernel.get(2, 2) * this->get(w + 1, h + 1) +
18                      kernel.get(2, 0) * this->get(w + 1, h - 1) +
19                      kernel.get(2, 1) * this->get(w + 1, h - 0);
20                  output.set(w, h, value);
21              }
22          }
23
24      return output;
25  }

```

## 2.2 Formulation

The sobel operator needs two convolution operations to detect the edges. The sobel operator works by calculating the change in image intensity, that is it works by calculating the difference in value of the value compared to its neighbors. The two convolution operations are to get the vertical changes and horizontal changes

1. Horizontal changes: This is represented as  $G_x$  and is calculated as

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$

2. Vertical changes: This is represented as  $G_y$  and is calculated as

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

We don't need the direction of the edge, we only need the magnitude. To calculate the magnitude, we will combine both the above results

$$G = \sqrt{G_x^2 + G_y^2}$$

Using the Matrix class we implemented in previous section, we can implement this algorithm as



```

1 | Matrix sobel_x({1, 0, -1,
2 |                 2, 0, -2,
3 |                 1, 0, -1}, 3, 3);
4 | Matrix Gx = image.convolute(sobel_x);
5 |
6 | Matrix sobel_y({1, 2, 1,
7 |                 0, 0, 0,
8 |                 -1, -2, -1}, 3, 3);
9 | Matrix Gy = grayscale.convolute(sobel_y);

```

The code to calculate magnitude is as follows

```

1 | double *magnitude = new double[width * height];
2 | double max_magnitude = 0;
3 | for(int i = 0; i < (width * height); i++)
4 | {
5 |     double value = std::sqrt(Gx.data[i] * Gx.data[i] +
6 |                               Gy.data[i] * Gy.data[i]);
7 |     if (value > max_magnitude)
8 |         max_magnitude = value;
9 |     magnitude[i] = value;
10 | }

```

Finally, we will output the image by converting the magnitude to an integer value between 0 and 255 by using the maximum magnitude.

```

1 | uint8_t *output = new uint8_t[width * height];
2 | for(int i = 0; i < (width * height); i++)
3 | {
4 |     output[i] = (magnitude[i] / max_magnitude) * 255;
5 | }

```

### 2.2.1 Result

The result of the sobel filter on one of the image from our dataset is shown in Figure 2.2. Here, we have shown the output in form of a image by converting magnitude to an integer between 0 and 255. In the project, we work using magnitudes directly.



Figure 2.2: Result of sobel operator

## Chapter 3

# Neural Network

A neural network is a type of machine learning model. It is inspired by the human brain, which is composed of neurons that form different pathways to process data and learn patterns. To mimic this behaviour, neural networks in computers are made of artificial neurons. An artificial neuron is a function which is conceived as a model of biological neuron in a neural network. An artificial neuron is the smallest unit of the Artificial Neural Network (ANN). The artificial neuron is based on how biological neurons work in human brain.

The artificial neurons are placed in layers to form our neural network. The input is passed to the first layer which is called the input layer, after which it moves forwards where transformations are done on it by each neuron. This process is called the forward propagation. Eventually, the signal reaches the last layer called the output layer. The layers between input layer and output layers are called hidden layers. The number of artificial neurons in the input layer will depend on the size of our input. In our project, we have a  $128 \times 128$  sized image, thus we will have 16,384 neurons in our input layer. We are making a recognizer, so number of neurons in output layer is number of classes. Thus our model will have 36 neurons in the output layer.

### 3.1 Artificial Neurons

The design of artificial neurons is inspired by how biological neurons work. More precisely, it uses two types of behaviour from the biological neuron. There is excitatory potential and inhibitory potential for activation. So for every incoming signal, the neuron either increases it with excitatory potential or decreases it with inhibitory potential. This is copied in our artificial neuron as weights. These weights are multiplied with our inputs and can be used to increase or decrease the intensity of the different inputs. The appropriate value for these weights is calculated during training using a process called backpropagation.

The structure of an artificial neuron is shown in Figure 3.1. A single neuron has multiple inputs it can take. This is shown in figure as variables  $x_1, x_2, x_3 \dots x_n$ . Every input to the

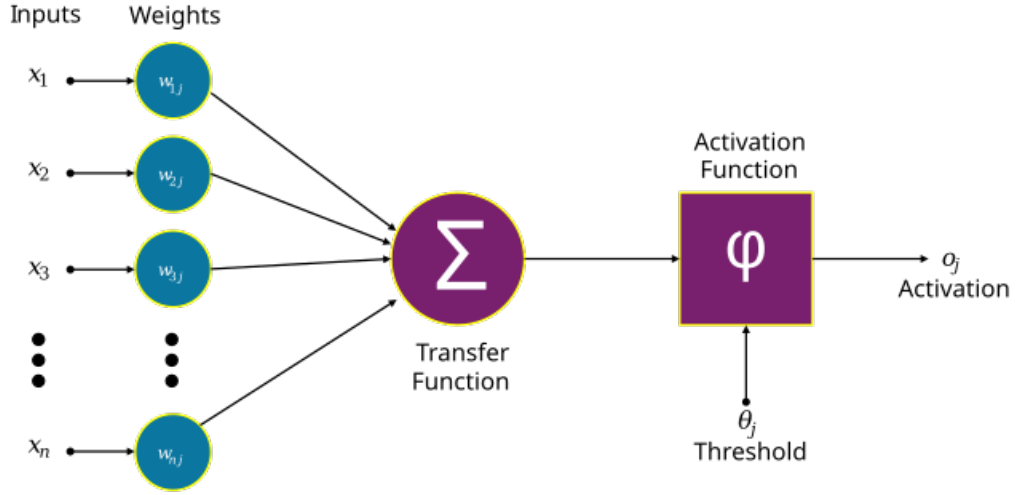


Figure 3.1: An artificial neuron

neuron will have an associated weight. If we suppose the neuron number is  $j$  in the layer. We will represent the weights as  $w_{1j}, w_{2j}, w_{3j} \dots w_{nj}$ . Now we need to combine all the inputs. This is done by a transfer function, which is summation in most models. Therefore, we will combine all the inputs as,

$$x_1 w_{1j} + x_2 w_{2j} + x_3 w_{3j} + \dots + x_n w_{nj} = \sum_{i=1}^n x_i w_{ij}$$

The next part of the neuron is the activation function. This function will take the result of the transfer function and produce the final output of the neuron. The need for activation function is to provide non-linearity. The transfer function only applies linear operations on the inputs. Having non-linear output allows our model to store relationships and patterns more efficiently. Some activation functions such as the ReLU function also help preventing signal saturation, a phenomenon where gradients become too small for learning.

Our model is only using a single activation function. It is the ReLU activation function. This function is very useful, specially in our project. This function helps to avoid the problem of vanishing gradients, which occurs in other activation functions. It is also simple to implement and computationally inexpensive. The one limitation of ReLU is that if inputs are consistently negative, output will always be zero. In our project, we are using images with intensity of each pixel as input, this problem won't occur in our model. The graph of the ReLU function is shown in Figure 3.2.

The function is defined mathematically as

$$ReLU(x) = \frac{x + |x|}{2}$$

But it is easier to implement in code by using the definition

$$ReLU(x) = \max(0, x)$$

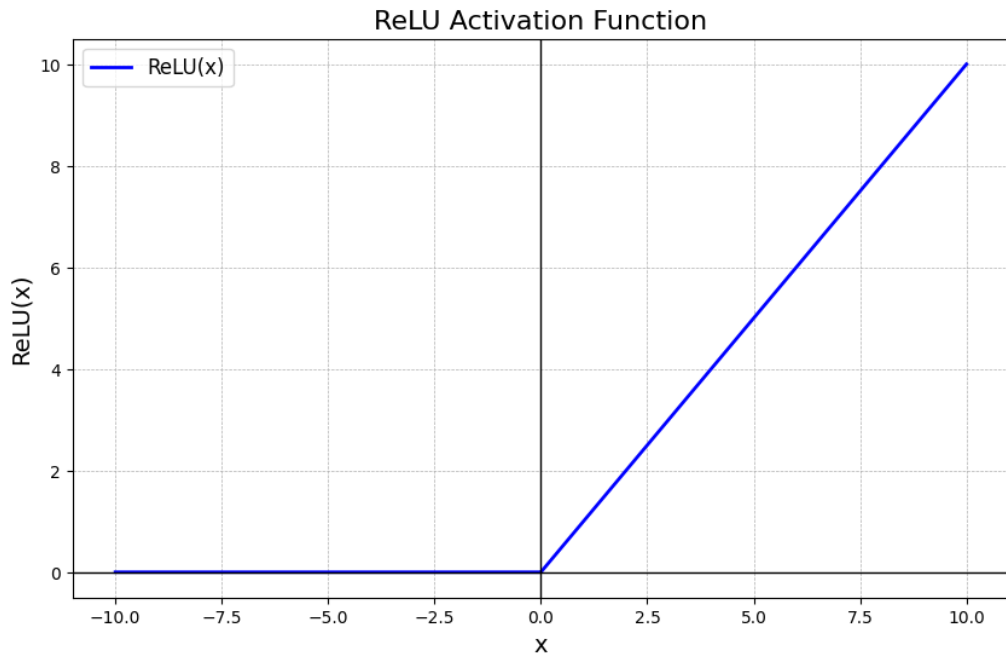


Figure 3.2: ReLU Activation Function

## 3.2 Training

The process of training in a neural network involves tweaking the weights associated with inputs of the neurons until we get expected results. In training and tweaking the weights, the model will learn the patterns in our input data. Thus, we will split our dataset into training data and testing data. The training data is used in the training process and the testing data is for testing our model. In our model, we have an 80 split for training and 20 for testing. The first step in training the model is choosing the loss function.

### 3.2.1 Loss function

The loss function (also called the cost function) is a function which shows how far our current predictions are from the actual answer. This allows us to automate the task of tweaking our weights in a way such that gets us better predictions, since we now only need to worry about minimizing the cost function and not worry about all of the neurons individually. There are multiple loss functions to choose from based on different use cases. A frequently used loss function is the Mean Square Error (MSE), but that function is better suited for regression. The loss function which is suitable for classification task is the Cross-Entropy loss function.

The Cross-Entropy loss function is used from [1]. The cross-entropy loss increases as the predicted probability diverges from actual label. This function minimizes the cost when the signal of the predicted label is correct. It also increases the cost for when signals of other labels are high but this effect is weaker than previous. This is shown in Figure 3.3 with the value of loss in y-axis and signal strength of predicted label is in x-axis.

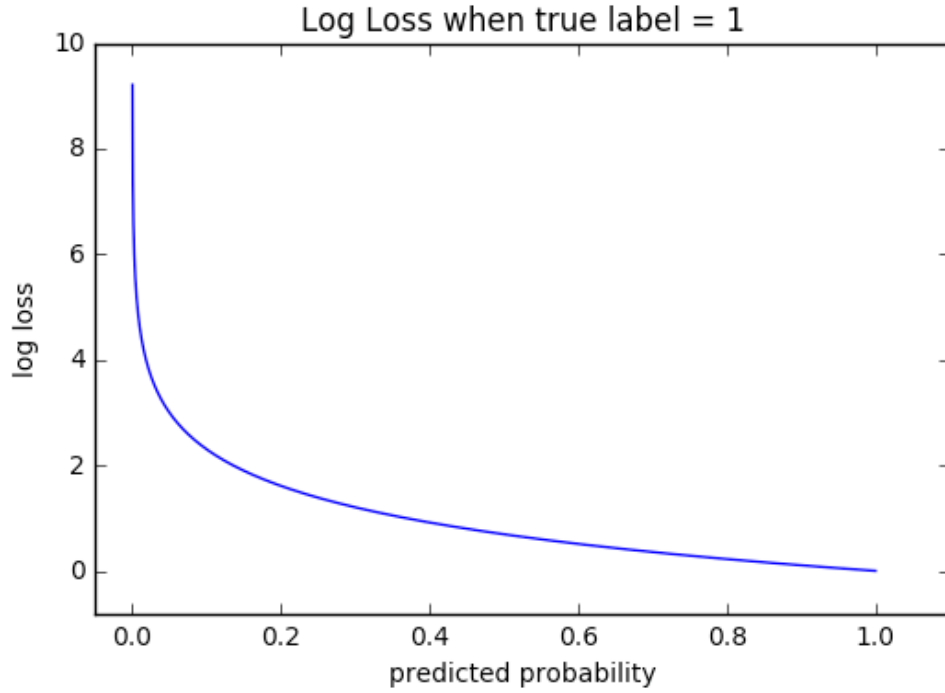


Figure 3.3: Cross-Entropy loss function

The Cross-Entropy loss is calculated by the following equation. Here,  $M$  is the number of classes,  $y_i$  is the expected output of label  $i$  and  $p_i$  is the prediction.

$$-\sum_{i=1}^M y_i \ln(p_i)$$

### 3.2.2 Optimizer

We now have a way to check the performance of the model, so we can use an optimizer which will find the appropriate weights to minimize the loss function. The most well known optimizer is the stochastic gradient descent (SDE). It is a variant of the gradient descent method, which is a general purpose algorithm which can be used to minimize any function. It uses gradient (slope) to calculate the minima of the given loss function. According to [2], the formula for gradient descent is

$$W_{new} = W_{old} - \alpha \times \frac{\delta(Loss)}{\delta(W_{old})}$$

Where,  $W$  is the current value of the weight, and  $\alpha$  is the learning rate. Learning rate is the amount in which direction we move the weight. Choosing the correct learning rate is necessary for getting the minimum cost efficiently and correctly. If it is too high the optimization will take a very long time, having to do this for every weight will get very computationally expensive. Having it too high will cause the weights to jump around a lot, this causes it to never settle for the minima. This is show in Figure 3.4

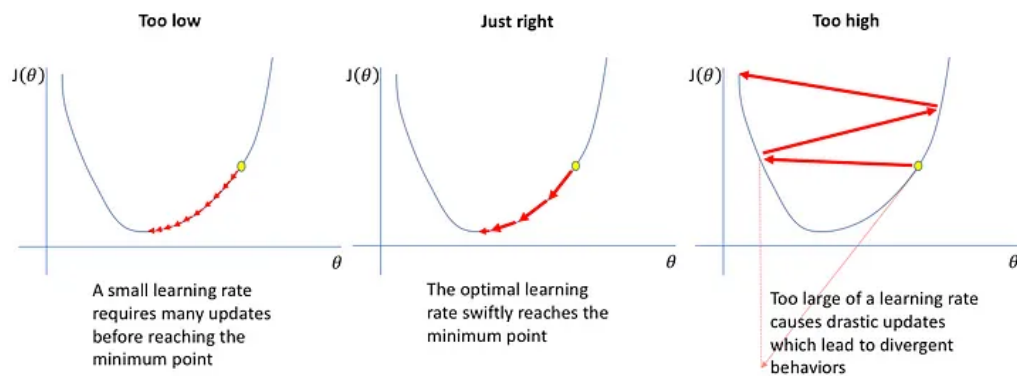


Figure 3.4: Choosing correct learning rate

This project uses Adaptive Gradient Descent (AdaGrad) which adaptively changes learning rate. This means that rather than being constant like we saw previously they are updated based sum of previous gradients squared.

### 3.3 Implementation

The project uses TensorFlow to create and train the neural network. The model is trained on the Google Colab T4 (Tesla T4) compute unit. The model is trained on cloud and is then downloaded to be used in offline project. The following imports are used to train the model

```
1 import pandas as pd
2 import numpy as np
3 import tensorflow as tf
4 from sklearn.model_selection import train_test_split
5 from google.colab import drive
```

We process the dataset using sobel filter and save the results in a csv file in the our drive, this is then loaded into the cloud. We also split the dataset into training and testing data.

```
1 data = pd.read_csv('/content/drive/MyDrive/is1.csv', header=None)
2 X, Y = data.iloc[:,0:16384].values, data[16384].values
3 X_train, X_test, Y_train, Y_test =
   ↪ train_test_split(X,Y,test_size=0.2)
```

After the dataset is loaded, we will make our model. This model has three layers, input layer has 16,384 neurons and uses the ReLU activation function. The second layer has 8,174 neurons and also uses the ReLU activation function. This hidden layer has the number of neurons chosen to be between neurons in input layer and those in output layer. The output layer is a layer with 36 neurons. This is built in tensorflow as

```
1 model = tf.keras.Sequential([
2     tf.keras.layers.Dense(128*128, activation='relu'),
```

```

3 |     tf.keras.layers.Dense(8174, activation='relu'),
4 |     tf.keras.layers.Dense(36)
5 | ])

```

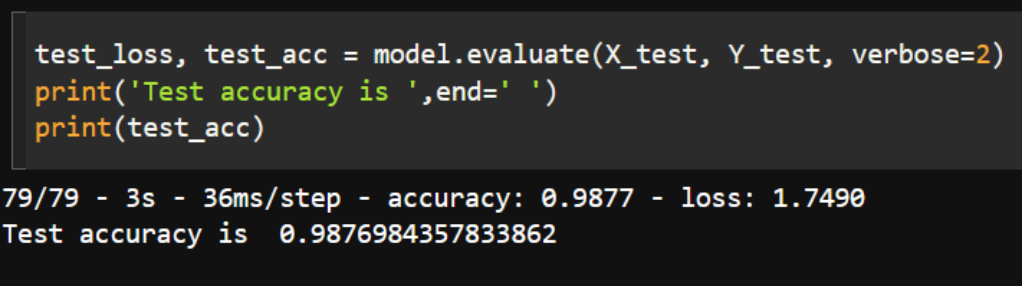
To train this model we need to call the compile and fit methods as shown

```

1 | model.compile(optimizer='adagrad', metrics=['accuracy'],
2 |               loss=tf.keras.losses.SparseCategoricalCrossentropy(fr
   |               ↪ om_logits=True))
3 | model.fit(X_train, Y_train, epochs=5)

```

Now, we can check the accuracy of our model using the testing data. The output for this is shown in Figure 3.5.



```

test_loss, test_acc = model.evaluate(X_test, Y_test, verbose=2)
print('Test accuracy is ',end=' ')
print(test_acc)

79/79 - 3s - 36ms/step - accuracy: 0.9877 - loss: 1.7490
Test accuracy is  0.9876984357833862

```

Figure 3.5: Accuracy of the model

The common way to show the accuracy of a model is to use a confusion matrix. It has the true labels in the Y-axis and predicted label in X-axis and data is shown in form of a matrix. Each cell shows the percent of time the model makes the prediction for the given label corresponding true label. Thus if model is has high accuracy, the diagonal will have should be high. The confusion matrix of our model is shown in Figure 3.6. We can see that our model shows confusion for 'U', '7', '6' and '4' but otherwise has very high accuracy.

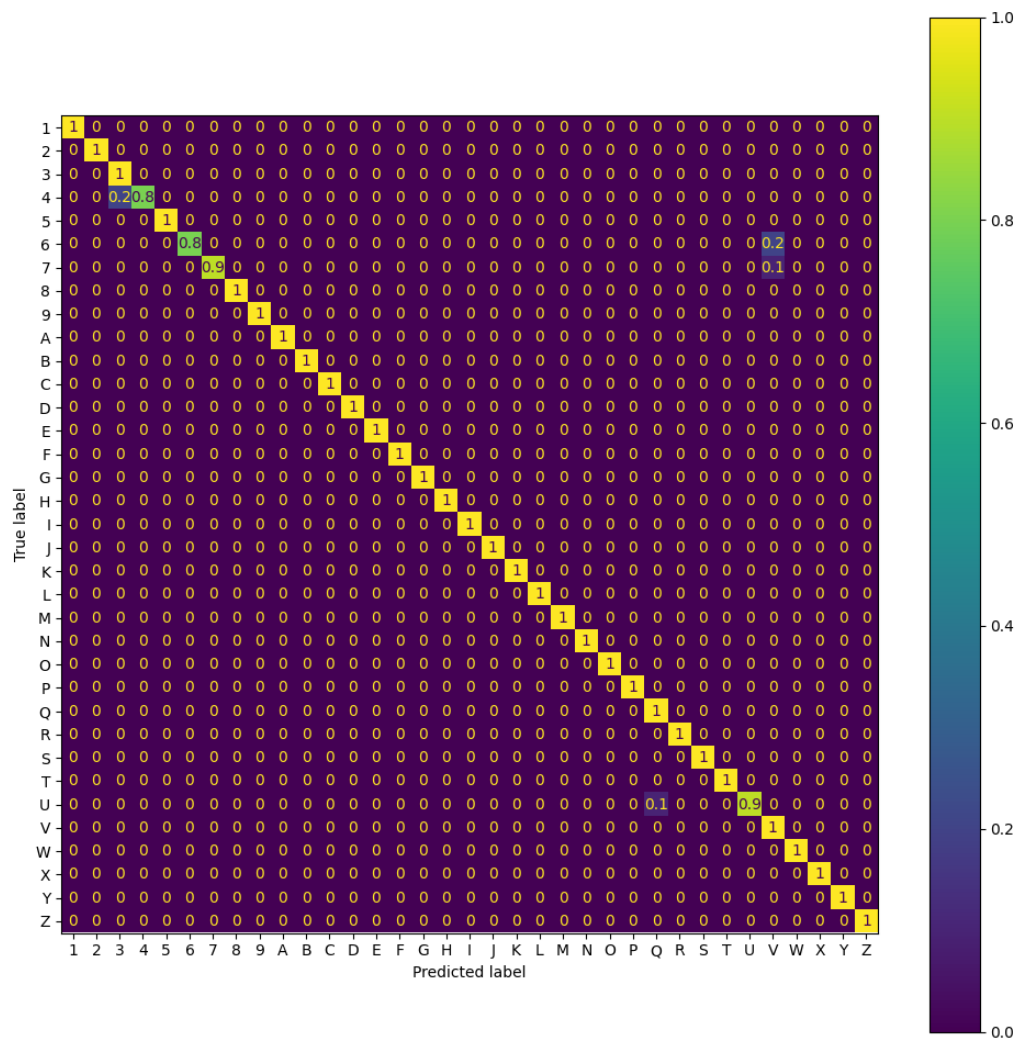


Figure 3.6: The confusion matrix



## Chapter 4

### OpenCV Application

The final component of this project is the OpenCV application which will take the input from the user. It takes the video from webcam as the input. It will then pass the input through our sobel filter. It will finally use the model which we trained to give a prediction. We can predict using the *model.predict* method. The result of this function is shown in Figure 4.1. This returns an array of predictions, where every index represents a different label. The index with the largest signal is the label which our model predicted.

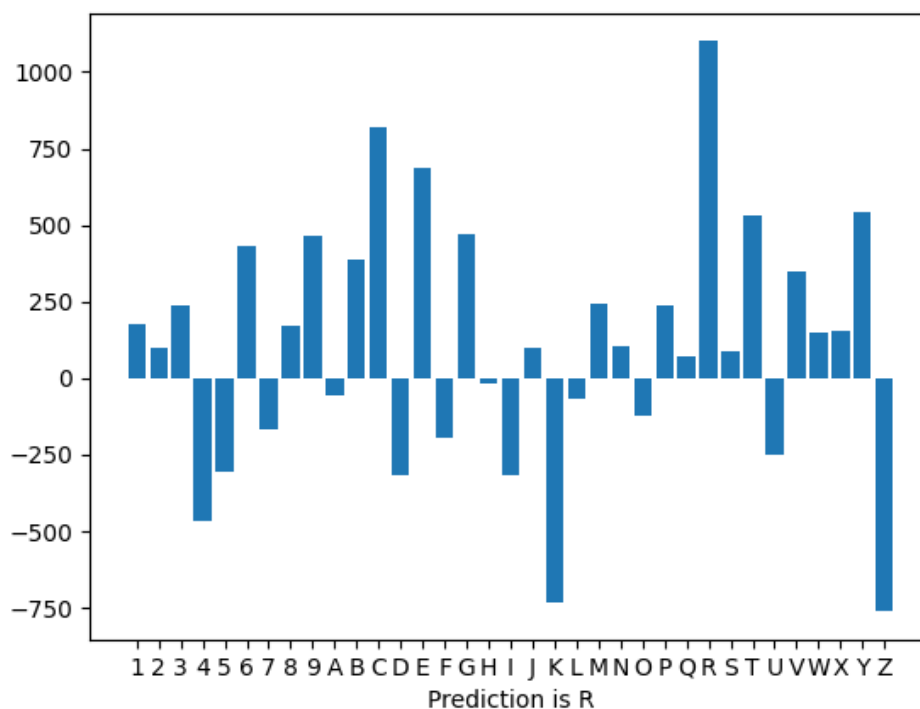


Figure 4.1: The result of prediction

## 4.1 Implementation

The first step is to setup the openCV loop. This loop is called every frame of the video

```
1 import pandas as pd
2 import tensorflow as tf
3 import numpy as np
4 import cv2
5
6 import subprocess
7 from io import StringIO
8
9 # Main loop
10 capture = cv2.VideoCapture(0)
11 frame_number = 0
12 current_prediction = "Waiting..."
13 while(True):
14     ret, frame = capture.read()
15     cv2.putText(frame, current_prediction,
16                 (10,20), cv2.FONT_HERSHEY_COMPLEX,
17                 1,( 255, 0, 0),2,cv2.LINE_AA)
18     cv2.imshow("Press Q to close", frame)
19
20     if (cv2.waitKey(1) == ord('q')):
21         break
22
23 capture.release()
24 cv2.destroyAllWindows()
```

Now, we need to get a square region in the middle of the frame where we can get the input. Thus we will need some global variables that will define the region

```
1 # Region properties
2 REGION_TOP_LEFT = (272, 272)
3 REGION_SIZE = (512, 512)
4 REGION_BOUNDARY_COLOR = (255, 0, 0)
5 REGION_BOUNDARY_SIZE = 3
6
7 region_bottom_right = (REGION_TOP_LEFT[0] + REGION_SIZE[0],
8                        REGION_TOP_LEFT[1] + REGION_SIZE[1])
```

Now, in our main loop we will add the following line to put the region in our frame after the cv2.putText

```
1 cv2.rectangle(frame, REGION_TOP_LEFT, region_bottom_right,
2               REGION_BOUNDARY_COLOR, REGION_BOUNDARY_SIZE)
```

Finally, in our we can get the region after the cv2.imshow call to get the region and write

it to an output.jpg

```
1 | if frame_number % 120 == 0:
2 |     # Getting region
3 |     region = frame[REGION_TOP_LEFT[0]:region_bottom_right[0],
4 |                   REGION_TOP_LEFT[1]:region_bottom_right[1]]
5 |     region = cv2.cvtColor(region, cv2.COLOR_BGR2GRAY)
6 |     region = cv2.resize(region, (128, 128))
7 |     region = cv2.flip(region, 1)
8 |     cv2.imwrite("output.jpg", region)
```

After we have written our output image, we can then give it to the sobel filter. To run other processes, we use the python subprocess module.

```
1 | opt = subprocess.check_output("./sobel.exe " + "output.jpg")
2 | inpt = pd.read_csv(StringIO(opt.decode('utf-8')[0:-1]), header=None)
```

Since we now have data in form of numpy array, we can pass the values to the model to get our predictions and change the current predictions accordingly.

```
1 | predictions = model.predict(inpt.values)
2 | current_prediction = "Prediction is " +
3 |                     characters_label[np.argmax(predictions)]
```

## Chapter 5

### Conclusion

In this chapter, we will present the results we got in this project in brief. The sobel filter is the first major component of this project. Since this is written in C++, this is very fast and computationally efficient. It is also very portable because the main logic is written in pure C++ and library to read and write images are the stb single file libraries [3] which are very small and cross platform. The filter being implemented in a system programming language also means it is easy to integrate with other languages.

To classify our images after applying the sobel operator, we have use an Artificial Neural Network (ANN) with three layers. The model supports inputs of size  $128 \times 128$ . The first layer has 16,384 neurons for input and is using ReLU activation function. The second layer which is the hidden layer of our model, has 8,174 neurons and also uses the ReLU activation function. The final layer which is the output layer has 36 neurons for the number of classes we need to recognize. We use the cross-entropy loss function because it works well with classification tasks, especially with high number of classes. We use the AdaGrad algorithm for optimization of the loss function. This is a variant of the Gradient Descent in which learning rates are adjusted during training. In our testing data, this gave us an accuracy of 98.77% as seen in Figure 3.5 this may be lower for other more general datasets and needs more testig with varied inputs.

The final component of our project is the openCV application which ties the other two parts together, allowing us to take input from real-time video and pass it through other two components. It will also tell the prediction to the user. This application has the most scope for work in the future. This component currently rescales the input, applies grayscale (though this part is not necessary since the sobel filter application can do it) and then saves the image. Then it calls sobel filter application to convert the image to csv output, this is data is given to the model which produces the final prediction.

## Chapter 6

### Future Scope

There are many future avenues for exploration, several compelling directions emerge, aligning seamlessly with the trajectory of this project and its uses for use by general public. Firstly, we can delve deeper into the edge detection where we can upgrade from a simple sobel filter to canny edge detection. This algorithm works after sobel filter to further select only the most prominent edges. This will help in getting better accuracy when working on real-time input like video, because most modern cameras usually focus on the foreground effectively making edges of the focus (hands) more prominent. We can go further and add a way to remove the background from our input. This would require another suitable algorithm or model which can detect the background and remove it.

The neural network can be upgraded to a Convolution Neural Network (CNN). These networks are better suited for computer vision tasks than traditional Neural Network which we have used in this project. Increasing the quality of the dataset, with more varied images which the network can train on. It would also be ideal if these images could have more noise in the background so that we could train the model to only isolate the hands.

The frontend application needs the most work in the future, if this project is to be used for gesture detection in true-to-life setting. The current implementation relies on OpenCV, but we are not using all of the capabilities of the library. It was chosen since it was appropriate for the scope of this project, but switching to another library which is lighter and can allow us to work with camera directly will significantly improve performance. Alternatively, we could also use OpenCV to do more difficult tasks such as the canny edge detection mentioned before. This would make the use of such a large library appropriate. The frontend also needs to be more user friendly and intuitive to use.

In essence, the future holds boundless opportunities for advancing the field of sign language recognition through interdisciplinary collaboration, innovative technologies, and a steadfast commitment to excellence. By embracing these opportunities and staying attuned to emerging trends and challenges, we can continue to push the boundaries of what's achievable, driving innovation and shaping the future of computer vision.

# Bibliography

- [1] [https://ml-cheatsheet.readthedocs.io/en/latest/loss\\_functions.html](https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html).
- [2] <https://musstafa0804.medium.com/optimizers-in-deep-learning-7bf81fed78a0>.
- [3] <https://github.com/nothings/stb>.
- [4] P. Kishore, P. R. Kumar, E. K. Kumar, and S. Kishore. Video audio interface for recognizing gestures of indian sign. *International Journal of Image Processing (IJIP)*, 5(4):479, 2011.
- [5] H. Lilha and D. Shivmurthy. Evaluation of features for automated transcription of dual-handed sign language alphabets. In *2011 international conference on image information processing*, pages 1–5. IEEE, 2011.
- [6] A. Nandy, S. Mondal, J. S. Prasad, P. Chakraborty, and G. Nandi. Recognizing & interpreting indian sign language gesture for human robot interaction. In *2010 international conference on computer and communication technology (ICCCT)*, pages 712–717. IEEE, 2010.
- [7] I. Sobel. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, 02 2014.