Implementation of CNN from scratch for Hand Gesture Recognition

Nishanjan Ravin[*], Parthan Olikkal[†], and Ambrose G.Tuscano[‡]

University of Maryland Baltimore County

CMSC 491/691 Final Report

Spring 2020, Group 3

Abstract

The goal of the project is to train a model, using the convolutional neural network machine learning algorithm, to be capable of recognizing different hand gestures, such as a closed fist, open palm, victory sign and others. The CNN model will 'learn' the features of each hand gesture and attempt to classify them correctly. We aim to implement this system from scratch in python, without using any external deep learning libraries. The dataset creation and pre-processing will also be done manually. We will then observe our CNN's performance, with metrics such as accuracy and F1-score, and compare this result with the performance of other CNN models in similar applications, to obtain a fair evaluation our developed CNN.

Keywords — Convolutional neural networks, Hand gesture recognition, Pre-processing, Image segmentation, Thresholding, Activation functions, Learning Rate, Weight vector initialization

[*]nishanr1@umbc.edu
[†]polikka1@umbc.edu
[‡]atuscan1@umbc.edu

1 Introduction

Hand gestures are a primitive form of human communication and one of the most natural form of expression. Evolutionary research suggests that the human language started with hand gestures and facial expressions, not sounds, illustrating the fact that hand gestures have been an integral component of human communication. Fast forward to the modern world, and we find that hand gesture recognition has a lot of potential applications, such as in sign language interpretation/translation or in interaction with machines. As hand gestures are increasingly considered to be superior in terms of convenience, many companies are trying to incorporate hand gestures as an option of providing inputs, rather than other complex forms of actions. However, the problem of hand gesture recognition has stifled progress in this field. In an attempt to solve this problem, many have turned their attention towards machine learning models.

While the neural network model is an effective form of machine learning, they suffer from two major disadvantages, particularly in the context of using images as the main dataset. Firstly, they take a lot of time to train, due to the rather large math computation required behind the neural network model. Secondly, they are limited by their capabilities to extract features, and are only able to do so at the level of each individual pixels. Thus, they do not work as well on images, in which features of the images exist on a much larger scale, i.e. a group of pixels. Hence, in an effort to improve upon these two aspects of the neural network model, the **convolutional neural network** was developed.

Convolutional neural networks (CNNs) aim to use the spatial locality of images to extract useful features, which then aid in classifying the images more accurately. In a CNN, several convolutional masks are constructed such that the output of these convolutional masks with the images of the dataset produces feature-rich layers, which greatly assist and improves the classification ability of the model. CNNs can also understand the complex and non-linear relationships amongst the images. As such, we plan on using a CNN-based approach to solve the hand gesture recognition problem.

The overall objective that we aim to complete in this project is to implement a working convolutional neural network, for the specific application of static hand gesture recognition. We aim to do this manually at all steps of the process, such as constructing our own dataset, conducting our own pre-processing, and developing the entire convolutional neural network by ourselves, without utilizing any 3rd party libraries or frameworks. By implementing the whole process manually, we will be able to gain a better understanding of the workings of the convolutional neural network model and the process of utilizing an advanced machine learning model in the context of a real-world application, e.g. the requirements that need to met when creating a dataset and the corresponding pre-processing procedures. This will also give us a greater amount of flexibility, hence providing us with more room for experimenting with the convolution neural network model to find the right set of hyper-parameters that would optimize its performance.

2 Related Work

We went through a lot of related works on the topic of Hand Gesture Recognition, thus covering major implementations and notable research done in the field. Our aim while covering these works was to increase our understanding of the working of CNNs in the context of image classification (specifically hand gesture recognition), to enable us to prepare a suitable method of approach for our project.

Alex Krizhevsky's implementation on the ImageNet Data set for Classification with Deep Convolutional Neural Networks (4), was the one we constantly fell back to. It pioneered the application of the CNN structure to classify images in a supervised data set, and was considered to be ground-breaking research during its time. The authors of (3) provide a comprehensive survey on Hand Gesture Recognition systems, and also gave insights into the structure that an ideal system needs to follow.

The authors of (1) have written on the various techniques to implement convolutional neural networks, and additionally, they have mentioned useful information related to various activation functions e.g. ReLU, max-pooling, etc. which we would be of great use in our manual implementation.

In (2), the "data augmentation" procedure increased about 4% accuracy in the traditional CNN framework. The CNN model used here utilized a train to test split of 70:30, and achieved a maximum accuracy of 98.95%. The experimentation from this research helped us in creating a better test and train split from the original dataset.

Raimundo F. Pinto, et.al. at (9) worked on how image pre-processing and subsequent segmentation to create a binary image or relevant and non relevant part (i.e. hand and background) affects the performance of the CNN. The research compares the output produced using various numbers of layers in CNNs, and how they compare with the output obtained with the pre-built model constructed using the "Keras" library. Here, a CNN of 4 layers showed the maximum precision of 96.86%.

Research at (6) worked on the process of data augmentation and utilizing dropout to reduce overfitting of the CNN model. By implementing ReLU as the primary activation function in their CNN model, they managed to attain a maximum accuracy of 88.5%. Our actual implementation of the CNN would take on a similar approach, but we will extend our focus to evaluating the performance by varying the number of layers and other hyper-parameter settings.The research paper (10) focused on building a Neural Network, concentrating on how feed-forward and back-propagation mechanisms are implemented and hence laid the foundation for implementation of a CNN.

Work by Xavier Glorot and Yoshua Bengio in (11) were referred to learn about how the weight initialization method can be optimized so that faster convergence of the model can be achieved. Additionally, we referred to Leslie Smith's research at (12) for understanding how various learning rate functions affected the performance of the model for a fixed number of epochs, and this knowledge was utilized in setting up a proper learning rate for our model.Siu Kwan Lam, et.al. at (13) shared their experience on building a JIT compiler using LLVM. We used this to enhance our knowledge on implementing parallel processing of the data, and to learn how the training process of the CNN could be significantly sped up by using a similar framework.

3 Methodology

The process which was followed when working on this project is detailed below. To facilitate our work on this project, we decided to split it up into two sections: the pre-processing section, and the implementation of the CNN section. Both sections were worked on in parallel, so that efficient progress in completing the project could be achieved.

3.1 Pre-Processing

We have subsumed the dataset creation steps in the pre-processing portion of the project. The details of the steps followed to create the dataset and conducting pre-processing of the images are discussed below.

3.1.1 Creating the Dataset

We created a dataset of different types of hand gestures, such as the victory sign, thumbs up sign, palm, etc. rather than using a dataset available online. This was done in order to gain a better understanding of how to manually create a dataset from scratch, as well as to work with a dataset that was not tampered in any way. Initially we made use of our phone camera and created a subset of the planned dataset to be used in the initial single layer fully-connected neural network. But as the size of the dataset started to increase we decided to automate this.

We made use of the inbuilt camera in our laptops and developed a script to automate and simplify the process of creating the dataset. This accelerated the process of creating properly labelled images and categorized those images into different names. At the end of this process, we had created a image dataset consisting of over 300 labelled images, which will be split into test and train datasets before being sent to the CNN.

3.1.2 Basic Pre-processing Steps

With a proper dataset in hand, our next step was to work on pre-processing and normalization of the images of the dataset. This step is crucial, as providing pre-processed and normalized images to our neural networks as inputs would enhance its performance. The different steps done in order to pre-process the dataset are mentioned below.

Uniform size and aspect ratio

One of the basic steps involved in creating a dataset is to ensure that each image in the dataset has the same size and aspect ratio. As the images that we capture through our laptop cameras are of varying dimensions, e.g. 1920×1080 and 1280×720, we decided to resize the images into uniform rectangular dimensions of 112×63. Through this step, all the images we collected through the creation of the dataset are converted into uniform images of constant size and aspect ratio of 16:9.

Another advantage of resizing the image to a smaller dimension is that it reduces the input size of the images (i.e. the number of pixels), hence lowering the number of variables that is involved in our CNN model. If the dimensions of the input images are high, then the number of variables utilized in the CNN would be larger, which increases the number of calculations and hence, the total time taken for one iteration of training the CNN would be longer. Thus, decreasing the size of the image also aids in reducing the overall training time of the CNN, without losing much of the image data.

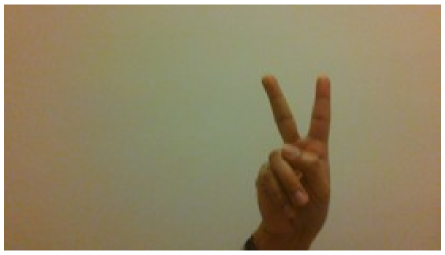

Figure 1: Victory sign image of 1920×1080 *dimension*

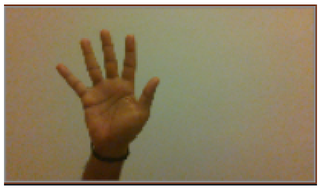

Figure 2: Palm sign image resized to 112×63 *dimension*

Dimensionality Reduction

The images captured by our laptops are of RGB-scale channel, and thus, to reduce the dimension of the images, we decided to convert them into a single gray-scale channel. As the RGB-scale channel has three layers, red, green and blue respectively, each image is represented as a three-dimensional matrix. This colored information individually is not particularly useful in identifying the important features (i.e. the shape of the hand) from the image. So with the help of some functions, we changed the RGB images to grayscale images without losing much information. Converting the images to grayscale helps us to simplify the inputs that are given to the CNN, as grayscale images utilize only a single channel of pixel intensities, rather than 3 different channels of pixel intensities in RGB images.

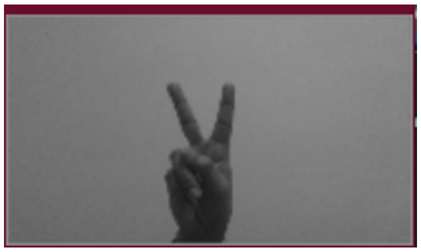

Figure 3: Grayscale version of the victory hand sign

3.1.3 Image Segmentation

As we are mainly interested in the hand signs, we should use these features only for training the neural network and try to eliminate all the other unnecessary information. The pre-processing done until this stage have resulted in a dataset consisting of images in grayscale. At this step, we attempted to segment the image, so that shape of the hand in the image could be better highlighted, and other unnecessary information can be removed. We considered several image segmentation techniques, as detailed below.

Image Segmentation using Global Thresholding

The first image segmentation algorithm that we tested was the Otsu's threshold algorithm, which is a type of global thresholding algorithm.

Otsu's threshold takes a grayscale image and returns a single intensity threshold that separates the pixels into a foreground and a background. In simpler terms this means that each pixel is checked with a global threshold value and if the value of the pixel is more than this global threshold value, it is classified as black (with pixel value 0), and otherwise, it is classified as white (with pixel value 255). This produces an image consisting of only black and white pixels, hence forming a binary image. Thus, converting the image to binary through a global threshold value removes information that is not required in identifying the hand gesture depicted in the picture.

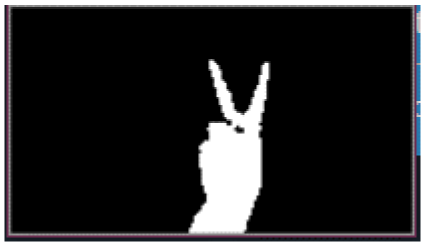

Figure 4: Binary (black & white) image of the victory hand sign

However, one of the limitations of utilizing Otsu's method for converting grayscale images to binary images is that this method only produces ideal results when the image is uniformly illuminated. As shown in the figure below, we can see this problem of illumination when utilizing a global thresholding algorithm such as Otsu's method.

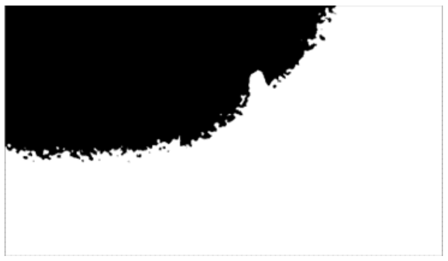

*Figure 5: **Otsu's method** output for the palm sign image under high illumination*

While we could adjust our images such that they would produce an ideal output using Otsu's method, our ultimate aim was to release a CNN model that can be adapted by anyone in a relevant real-world scenario, instead of making it specific to just our use-case. The images included in the dataset cannot always be taken in a perfect environment, and may originate from different environments with varying factors, such as amount of illumination, skin tone, etc. As such, we have looked at alternative image segmentation techniques that would work better on a wide variety of images.

Image Segmentation using Adaptive Mean and Adaptive Gaussian Thresholding

In adaptive mean thresholding, rather than using only a single value of threshold as in the global thresholding technique, this method uses a value that is the average/mean of the neighborhood area. This would provide a much better output for the same image, in comparison with using global thresholding. In the adaptive gaussian thresholding technique, instead of using the mean of the neighborhood area, this method makes use of the gaussian-weighted sum of the neighborhood pixel values. Presented below are the outputs of the adaptive mean thresholding algorithm and the adaptive gaussian thresholding algorithm on the same palm sign image shown earlier.

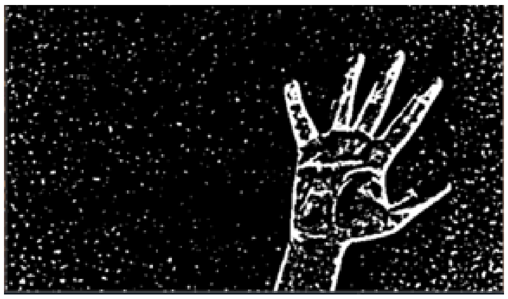

Figure 6: ***Adaptive Mean Thresholding*** *output for the palm sign image*

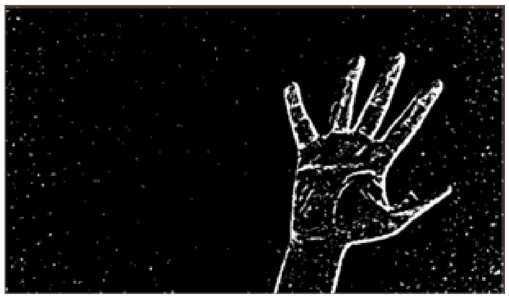

Figure 7: ***Adaptive Gaussian Thresholding*** *output for the palm sign image*

We can see from the above images that a much clearer image reproduction can be obtained with adaptive mean thresholding and adaptive gaussian thresholding, in comparison with the global thresholding technique that we had utilized earlier.

Finally, we took the adaptive gaussian algorithm as our thresholding technique of choice to pre-process the images of the dataset. The adaptive gaussian algorithm produced a much cleaner and less noisier output in comparison to the adaptive means algorithm. Pre-processing our dataset with this algorithm would enable the CNN model to perform better, as unnecessary information from the images would have been eliminated.

3.2 Implementing the CNN

For implementing the CNN, we first created a simple neural network (fully connected single-layer) in Python. The specifications of this neural network, along with the results of its performance on our dataset, are presented in the *Results* section below.

3.2.1 Single-Layer Fully Connected Neural Network

Starting off with this simple neural network model enabled us to strengthen our understanding of implementing neural network functions in code, such as creating arrays of weights and biases, differentiating the loss with respect to the output layer, backpropagating the partial derivative of the loss with respect to each variable through the neural network, and finally updating each individual weight and bias accordingly. The main difficulty that we encountered in this process was deriving the formula for the partial derivative of the loss with respect to the input of the softmax function.

However, with information that we obtained from reading relevant work, and doing further research, we managed to complete this step. We trained this version of the neural network on our training dataset, and evaluated its performance on the test dataset.

3.2.2 Multi-Layer Fully Connected Neural Network

Next, we worked on adding a single hidden layer to the current neural network structure. This model was not expected to be much of a significant improvement to the earlier version of the neural network. However, this step was crucial in enhancing our understanding of the mathematical calculations behind the neural network, particularly with regards to the backpropagation of the partial derivative of the loss through the network.

For our backpropagation step, we initialized 2 helper arrays for each layer of the network, 1 for calculating the partial derivative of the loss with respect to the weights of the layer, and 1 for calculating the partial derivative of the loss with respect to the biases of the layer. These helper arrays were reset to 0 for each iteration of neural network, so that the values of the previous iteration will not affect the backpropagation of the current iteration. In addition to the above, we also initialized helper arrays at each layer, to save the input to the layers (before the activation function) during the forward propagation step. We also implemented the forward propagation procedure as a function, as this will be particularly useful when scaling up the size of the neural network to include more hidden layers.

The actual backpropagation was done by using nested for-loops, so that the derivative of the loss with respect to neuron is precisely calculated, while still keeping the code concise. These calculations utilized the values stored in the helper arrays (which stored the input to the activation function during the forward propagation step), as well as pre-defined helper functions for calculating the derivative of the respective activation functions. These partial derivative of loss values were then stored in the corresponding helper arrays, which were used at the end of the iteration to update the weights and biases by multiplying them with a pre-defined learning rate. The performance of this CNN on our hand gesture dataset was evaluated, and the resulting observations are presented in the *Results* section below.

3.2.3 Converting the Neural Network into a Convolutional Neural Network

At the next stage, we worked on incorporating the convolution and max-pool layers into the neural network, effectively making it a convolutional neural network. Thus, we worked on separate functions to aid in the forward propagation and backpropagation steps of both these layers. While the forward propagation steps of both these layers were rather straightforward and easy to implement, the backpropagation steps required us to do some additional research and information gathering, as the backpropagation calculations in these steps vary from the backpropagation steps of a fully connected layer. After comprehending the required logic and maths, we then started to implement the code for these respective steps.

Backpropagating the partial derivative of the loss through the max-pool function is pretty straightforward to understand, but however it is slightly complicated to implement in code. In the forward propagation step, the max-pool layer works by taking the maximum pixel value of a fixed region, and setting that to be the value of the corresponding pixel in the final output. In the backpropagation step, this same logic is applied in reverse. Only the maximum pixel value of that max-pool region has contributed to the output of the max-pool function, and thus, the partial derivative of the loss should be associated with the corresponding maximum pixel value in that region, and the partial derivative of loss associated with the other pixels in that region should be set to 0. This whole process was defined in a single function with the following inputs: the input to the max-pool layer, the loss associated with the output of the max-pool layer, and the size reduction value of the max-pool layer. This whole process is better illustrated with the diagram below.

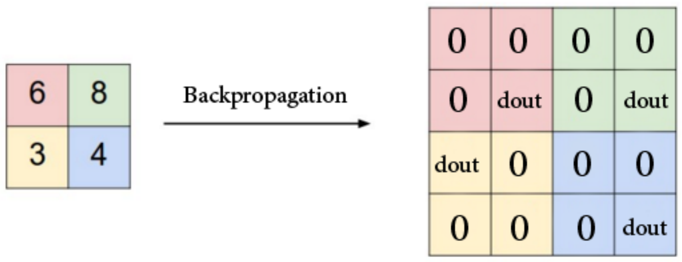

Figure 8: Backpropagation of the partial derivative of loss through the max-pool function of size & stride = 2

Backpropagating the partial derivative of the loss through the convolutional layer was much simpler. The forward propagation of the convolutional layer was done by convolving each filter with the input image to produce the output. Thus, each pixel of the convolution output is essentially affected by all corresponding values of the filters and the input image. As a result, if a pixel of the convolution output has a partial derivative of loss value associated with it, then the partial derivative of loss with respect to the corresponding filter values can be calculated by multiplying this partial derivative of loss value (with respect to the convolutional output pixel) with the corresponding pixel value of the input image. This whole process was defined in a single function with the following inputs: the input to the convolutional layer and the loss associated with the output of the convolutional layer. This partial derivative of loss value with respect to the filters is then multiplied with the learning rate to finally update the filter values at the end of each iteration.

We carried out some initial testing on a dataset of some simple 2D arrays to debug and ensure that the convolutional and max-pool layers were working correctly, before testing the CNN on our hand gesture dataset. As with the previous model, we utilized several helper arrays to aid in the calculation of the partial derivative of the loss through the network. Thus, our CNN model at this step had 1 convolutional layer, 1 max-pool layer, 1 hidden layer and 1 output layer, and its performance on our dataset was evaluated, which is presented in the *Results* section below.

3.2.4 Experimenting on main CNN hyper-parameters

Before proceeding with adding more layers to the CNN and performing extensive amounts of experimentation and evaluation, we decided to look at whether we could optimize the existing version of the CNN, and hence, we investigated other factors of the CNN model that would enhance its performance. After conducting an extensive amount of research, we realized that the performance of the CNN could be improved by experimenting with two crucial factors: the activation function used at each layer, and the weight vector initialization method which was employed. This was a crucial step, as it enabled us to gain useful insights into improving the performance of the CNN, as well as providing us with more knowledge behind the mathematical workings of the CNN model.

Another important factor that significantly affects the performance of the CNN is the learning rate. Thus far, we had used a constant learning rate in the implementation of our CNNs. However, this is not very ideal in terms of performance of the CNN, as it will not allow for optimal convergence of the CNN. As a result, we researched about functional learning rates, that vary with the number of iterations/epochs of the training step, to improve on this aspect of our CNN. Based on our findings, we found the cyclic and exponential learning rates to be the best performing functional learning rates, and as a result, we have implemented these two types of learning rates in our CNN.

The results that we obtained by experimenting with these aforementioned factors are detailed below in the *Results* section.

3.2.5 Experimentation of various configurations of hyper-parameters

The final step in our CNN methodology was to conduct an extensive amount of testing on our CNN model, by varying the hyper-parameters of the CNN model that we have built thus far, which include the number of hidden layers, the types of hidden layers and the sizes of these hidden layers (i.e. the number of filters in convolutional layers, the dimensions of these filters, the size and stride of max-pool layers, and the number of neurons in fully-connected layers).

In this step, we conducted an evaluation of our CNN for each configuration, to ascertain the best set of hyper-parameters which will optimize the performance of our CNN on the hand gesture dataset. This involved a lot of experimentation with regards to the various combinations of the factors of the CNN. The findings that we obtained from this step are detailed in the *Results* section below.

3.2.6 Implementing proper evaluation of the CNN model

In order to conduct an accurate evaluation of the developed CNN model, we made use of the popularly used metrics such as recall, precision, and F1-score. These additional metrics will aid us in providing a better insight of how well our model is working, and hence will definitely be helpful in any fine-tuning of the model that will be done. This was done by evaluating the trained model of the CNN using the confusion matrix that was obtained from running it test set.

The confusion matrix was implemented by making use of numpy arrays, and then populating it according to the output of the model on the test set. This then allowed us to calculate additional metrics such as precision, recall and F1-score. While precision and recall are not particularly meaningful to us in this context, the F1-score was used an alternative measure to evaluate the performance of the CNN, apart from the regular accuracy score.

4 Results

This section discusses the results and observations that we have obtained thus far, and discusses the progress that we have made on our project. As per the organization before, we will be splitting this section into two portions: the first discusses the results pertaining to the pre-processing section of the project, while the second details the results obtained during the implementation of the CNN.

4.1 Pre-Processing

In the flowchart below, we have a visual representation of what our pre-processing stage accomplishes thus far, from the initial RGB image to the final binary image. Initial images were converted to a smaller dimensions of 112×63, converted from RGB to grayscale, and finally converted to binary using the adaptive gaussian thresholding algorithm. These final binary images are then provided as input to the CNN.

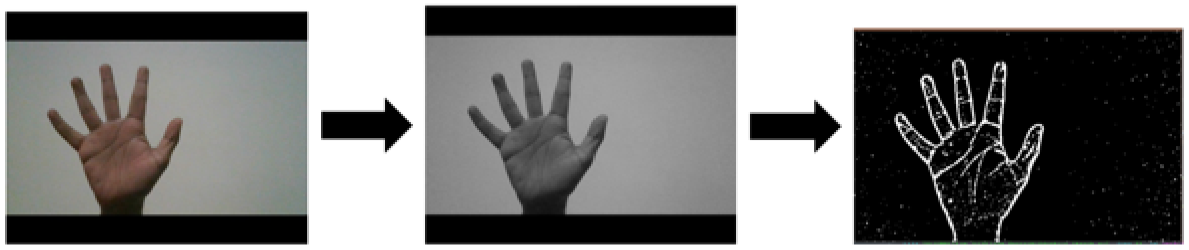

Figure 9: Process of conversion from RGB image to grayscale image to binary image

4.2 CNN Evaluation

In this section, we will be looking at the progress made on the implementation of the CNN. Each subsection below details our results pertaining to each stage of the developing CNN. In evaluating each version of the CNN, we mainly considered two factors: the accuracy of the model on our hand gesture dataset, as well as the total amount of time taken to train the model.

4.2.1 Single-Layer Fully Connected Neural Network

We first built a single-layer fully connected neural network to start off our implementation of the CNN. While this is not expected to be the best performing model on our dataset, this enabled us to start off with a simple model, and then work on improving it by adding more advanced elements to it, as discussed in the *Methodology* section. The following was the configuration of this simple single-layer fully connected neural network, and its performance on the hand gesture dataset.

Configuration

- Learning Rate $\rightarrow$ 0.05
- Train:Test Ratio $\rightarrow$ 80:20

- Output Layer → Softmax activation
- Loss Function → Cross-Entropy Loss
- Number of Epochs = 500

Performance Evaluation

- Train Accuracy = 100%
- Test Accuracy = 56%
- Training Time ≈ 1.3 hours

4.2.2 Multi-Layer Fully Connected Neural Network

Improving on the version of the single-layer fully connected neural network above, we added a single hidden layer to create a multi-layer fully connected neural network. The configurations of this neural network are listed below, along with the details of its performance on the hand gesture dataset.

Configuration

- Learning Rate → 0.05
- Train:Test Ratio → 80:20
- 1 Hidden Layer → Sigmoid activation
  - 10 hidden nodes
- Output Layer → Softmax activation
- Loss Function → Cross-Entropy Loss
- Number of Epochs = 500

Performance Evaluation

- Train Accuracy = 95%
- Test Accuracy = 63%
- Training Time ≈ 4 hours

4.2.3 Simple CNN

Following the above neural network, we decided to implement convolutional and max-pool layers, to make it a full-fledged CNN. The specifications of this CNN, as well as its performance on the hand gesture dataset, are listed below.

Configuration

- Learning Rate $\rightarrow$ 0.005
- Train:Test Ratio $\rightarrow$ 80:20
- 1 Convolutional Layer + Max-pool layer $\rightarrow$ Leaky ReLU activation
  - 50 filters, each of size 20×20
  - Max-pool size and stride = 2
- 1 Hidden Layer $\rightarrow$ Sigmoid activation
  - 50 hidden nodes
- Output Layer $\rightarrow$ Softmax activation
- Loss Function $\rightarrow$ Cross-Entropy Loss
- Number of Epochs = 50

Performance Evaluation

- Train Accuracy = 98%
- Test Accuracy = 66%
- Training Time $\approx$ 19 hours

4.2.4 Experimentation on Main Hyper-parameters

After coming up the above model, we decided to optimize the main factors of the CNN before proceeding to experiment with other hyper-parameters. For this purpose, we mainly focused on three factors that we could play around with: the activation function, the weight vector initialization method and the learning rate. The results that we obtained from experimenting on these 2 areas are detailed below.

Activation Function

Based on the research that we had done, we came to realize that the choice of activation function that is utilized in the model is rather crucial. The most popular choices of activation functions that are being used in the realm of CNNs right now are as follows:

- Sigmoid activation
- Tanh (Hyperbolic Tangent) activation
- ReLU (Rectified Linear Unit) activation
- Leaky ReLU activation

We tested out these activation functions in the CNN that we implemented above on a much simpler dataset (not the hand gesture dataset, as the training time on this dataset was too long), and the following are the results that we obtained.

Based on the experimentation that we conducted with the above 4 types of activation functions, the **Leaky ReLU** activation function was found to be the best. These observations can be explained by the following explanation.

Type of Activation Function	Accuracy Obtained
Sigmoid	79.8%
Tanh	80.1%
ReLU	82.6%
Leaky ReLU	83.4%

Table 1: Accuracy results of respective activation functions

The sigmoid and tanh activation functions have a higher probability of experiencing the vanishing/exploding gradients problem. This is due to the fact that as the value of the input to these activation functions tends towards ∞ or $-\infty$, the derivative value of these activation functions tends towards 0, and hence the gradient ”vanishes”. This greatly limits the capability of the neural network to backpropagate the partial derivative of the loss, and hence the updates to the weights and biases of the neural network becomes miniscule, stifling the overall learning potential of the neural network.

The ReLU activation function however precludes the vanishing/exploding gradients problem, as the derivative of the ReLU function can only either be 1 for values of input greater than zero, or 0 for values of input lesser than zero. Thus, the relevant neuron is either activated, or not activated. This greatly simplifies calculations during the backpropagation steps of training the neural network, and enables it to backpropagate the partial derivative of the loss much more effectively, hence increasing the learning capability of the neural network. However, the ReLU activation function suffers from one major drawback, known as the ”dying ReLU” problem. If the input to the neuron becomes lesser than zero, then the neuron effectively becomes dead, and it is mostly never able to recover again. As such, the ReLU function might cause a large part of the neural network to become dead, once again stifling the performance.

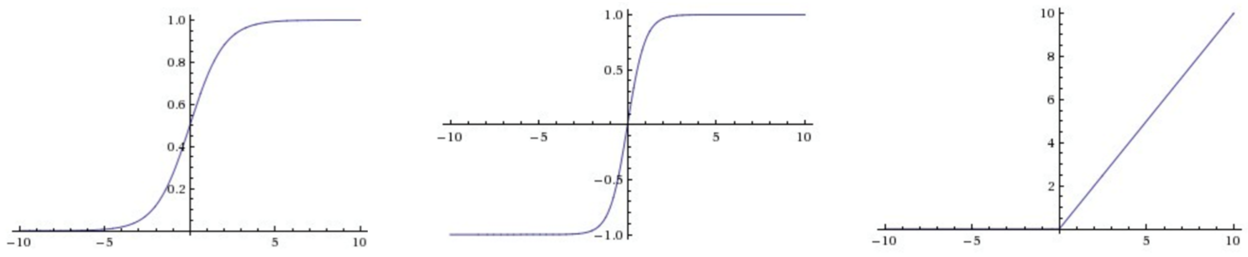

Figure 10: Graphical Representation of the Sigmoid, Tanh and ReLU activation functions (from left to right)

To mitigate the ”dying ReLU” problem, the Leaky ReLU activation function offers a good compromise. The leaky ReLU function is modified from the original ReLU function such that its derivative value is 1 for values of input greater than zero, or 0.01 for values of input lesser than zero. Due to the slight gradient given to inputs below zero, this allows dead neurons to possibly recover with further iterations, and hence, these neurons are not killed forever. As such, the neural network performs best when the Leaky ReLU activation function is used, as supported by our experimental data, and this is the activation function that we have chosen to implement in our CNN model.

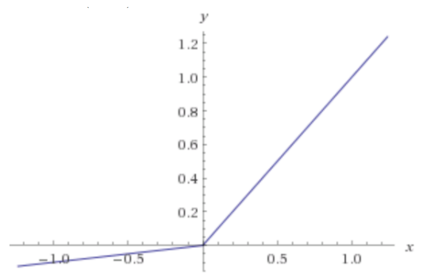

Figure 11: Graphical Representation of the Leaky ReLU activation function

Weight Vector Initialization Method

Weight vector initialization is another crucial factor to consider when initializing neural networks. If the weight and bias vectors are initialized with bad values, then the neural network may never be able to reach optimal performance, as it may end up getting stuck on one of many possible local minimas (of minimizing the loss value). Thus, it is crucial that the weight vectors are initialized such that the chances of getting stuck on a local minima are minimized, and the chances of reaching the global minima are maximized.

Up until the point of implementing the simple CNN model above, we initialized the weight and bias vectors by randomly sampling numbers between -1 and 1. While this method of initialization does not negatively impact the performance of the CNN, it does not necessarily optimize the performance of the CNN either. Thus, upon conducting further research, we learnt about two main methods of weight vector initialization, which are discussed below.

The key idea behind using custom functions to initialize weight and bias vectors, is to maintain the standard deviation of the layers' activations around 1. This will enable us to stack several more layers in the neural network, without experiencing the problem of gradients exploding or vanishing. The following methods of initialization utilize logical premises and mathematical calculations to maintain this assertion as far as possible.

1) Xavier Initialization

The Xavier initialization was devised by keeping the aforementioned concept in mind. The bias and weight values are randomly sampled from a uniform distribution that is bounded by the formula shown below.

$$Range = \pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}$$

where n_i is the number of incoming connections from the previous layer, a.k.a "fan-in" of the current layer, and n_{i+1} is the number of outgoing connections from the current layer, a.k.a "fan-out" of the current layer. For the sigmoid and tanh activation functions, on average, it enables the activation outputs of each layer to have a mean of 0 and a standard deviation of approximately 1, so that the value of the derivatives of the activation functions are never too low, and are instead kept at an ideal range. However, this would mean that the Xavier initialization method is effective only for the sigmoid and tanh activation functions, and would be ineffective for other activation functions such as the ReLU group.

2) Kaiming Initialization

To extrapolate the method of optimal weight vector initialization to ReLU functions, the Kaiming initialization method was developed. It follows the methodology mentioned below:

1. All the weight values are initially set to values sampled from a standard normal distribution (mean of 0, standard deviation of 1).
2. All the weight values are multiplied by a factor of $\sqrt{\frac{2}{n_i}}$, where n_i is the number of incoming connections from the previous layer, a.k.a "fan-in" of the current layer.
3. The bias values are initialized to 0.

Hence, the Kaiming initialization method has been proven to be the most effective method of weight vector initialization for the ReLU group of activation functions. However, due to the random nature of weight vector initializations, this claim will require a lot of experimenting and testing to be validated with evidence. Thus, we decided not to prove this claim by ourselves, and we instead assume this claim to be true, and have chosen this method of weight initialization for our current CNN implementation.

Learning Rate

The learning rate is one of the most crucial hyper-parameters of the neural network model, as it could be the factor that could make or break the entire model. If the learning rate is set to be too high, the CNN will never be able to converge, and instead will oscillate further away from the optimal configuration; on the other hand, if the learning rate is set to be too low, a large number of iterations will be required for the CNN to converge, hence increasing the overall training time. As such, the learning rate should be set at an optimal value that precludes the former case, and yet mitigating the possibility of the latter.

Up till now, all our CNN models had employed a constant learning rate. While this might be a very simplistic approach at setting the learning rate, it is definitely not the most optimal. Upon doing further research, we realized that functional learning rates should give us a significant improvement over using a constant learning rate. A functional learning rate will allow for faster convergence of the neural network by basing the learning rate value on a suitable function that changes according to the number of iterations/epochs that have passed. We studied various types of learning rate functions, which are graphically represented in the figure below:

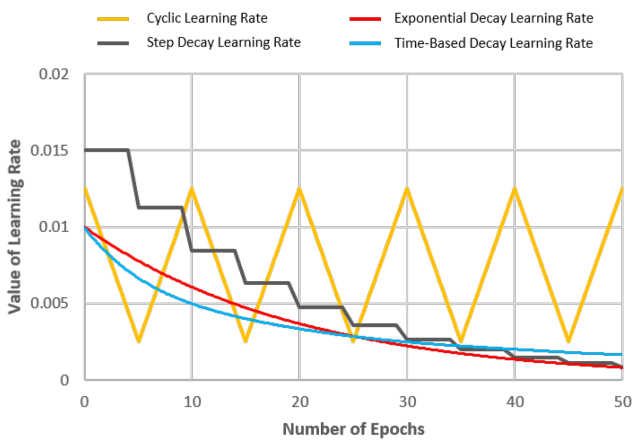

Figure 12: Graphical representation of the different types of learning rate functions

On doing further research, we realized that the exponential decay learning rate, and the cyclic learning rate were found to be the most effective types of learning rate in simple CNN models. We implemented these learning rates in our CNN model, and observed its corresponding performance. The results are plotted graphically below.

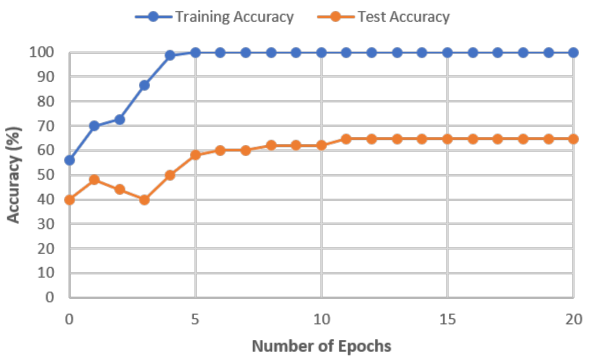

*Figure 13: Graph of test and train accuracy vs. number of epochs when using **exponential learning rate***

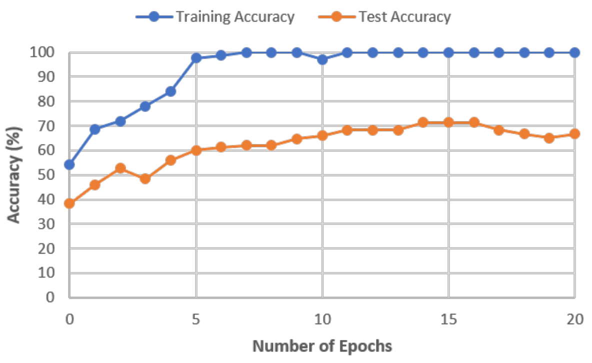

*Figure 14: Graph of test and train accuracy vs. number of epochs when using **cyclic learning rate***

With the exponential learning rate, we observed that the test accuracy approached a asymptotic limit, and was able to get a maximum test accuracy of 64.67% from the 11th epoch. The cyclic learning rate implementation achieved a maximum test accuracy of 71.33% at the 15th epoch, but this appeared to decrease as the number of epochs increased. This observation is explained as follows.

The exponential decay learning rate will set the learning rate to relatively larger values during the initial stages of training, allowing the updates to the weights and biases of the CNN to be more significant and larger in magnitude, hence accelerating the process of reaching the optimal values. As training progresses and the number of epochs increases, the learning rate value decreases exponentially, causing the magnitude of the updates to the weights and biases to decrease in a similar manner.

This effect has its pros and cons. On one hand, this might be considered ideal, as the weights and biases would only need minor updates as they get closer to the optimal values, i.e. convergence of the CNN. However, if the learning rate value becomes too small in the early stages of the training, then the CNN may never be able to converge to optimality, as the updates to the weights and biases would become so small that they do not improve the performance of the CNN. This is observed in our experimental data, as we see that the test accuracy does not increase/decrease after the 11th epoch.

The cyclic learning rate is able to mitigate the problem described above. This comes from the fact that cyclic learning rate function never causes the learning rate to become too small, and instead oscillates between a larger than average and smaller than average learning rate value. As such, a cyclic rate allows the CNN to improve significantly even at a larger number of epochs.

However, the cyclic learning rate has one major disadvantage. While it does allow significant learning to happen at a larger number of epochs, this comes with the tradeoff that it also increases the chances of overfitting occurring in the CNN model, due to the larger than average learning rate values at certain points of the training. This can be observed in our experimental data as well. While a maximum test accuracy of 71.33% was achieved at the 15th epoch, further epochs caused the test accuracy value to decrease slightly, indicating the possibility of overfitting.

As such, we decided to go for a best-of-both-worlds approach, and combine the positive aspects of both learning rate functions to create an optimal learning rate function that we expect to perform best. This learning rate function - termed as **exponential cyclic learning rate function** - is shown below, as well as the performance of the CNN when utilizing this learning rate function.

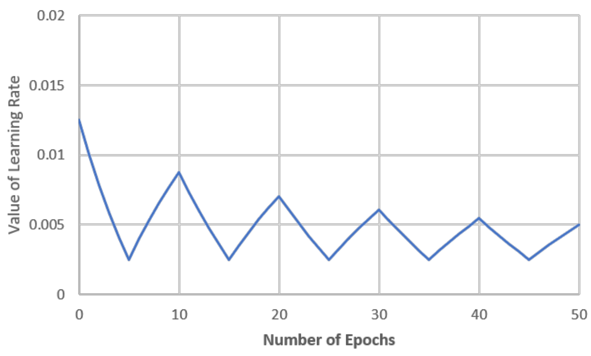

Figure 15: Graphical representation of the exponential cyclic learning rate function

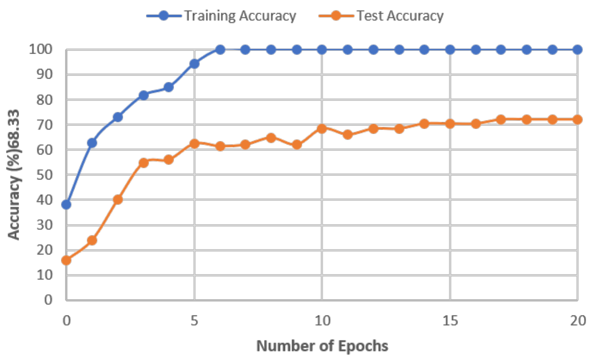

Figure 16: Graph of test and train accuracy vs. number of epochs when using ***exponential cyclic learning rate***

As observed, we were able to mitigate the problem of overfitting, that was present in the initial cyclic learning rate implementation, while keeping the benefits of the exponential decay learning rate function. As such, we were able to achieve a test accuracy of 72% from the 17th epoch, and the CNN was able to maintain this test accuracy at larger numbers of epochs. These positive results led us to finalizing this learning rate function for the final CNN model.

4.2.5 Experimentation of various configurations of hyper-parameters

As the last step in finalizing our developed CNN, we conducted an extensive amount of experimentation on the various possible configurations of the hyper-parameters of the model. This included, the number of filters in the convolutional layer, the height and width of the filters, the max-pool layer size and stride, and the size and number of the fully connected layers.

We experimented with various values for the above factors, and evaluated them accordingly. The details below lists the best configuration of our CNN, such that the highest test accuracy value was obtained.

Configuration

- Learning Rate $\rightarrow$ Exponential Cyclic Learning Rate
- Train:Test Ratio $\rightarrow$ 80:20
- 1 Convolutional Layer + Max-pool layer $\rightarrow$ Leaky ReLU activation
  - 50 filters, each of size 50×50
  - Max-pool size and stride = 2
- 1 Hidden Layer $\rightarrow$ Leaky ReLU activation
  - 35 hidden nodes
- Output Layer $\rightarrow$ Softmax activation
- Loss Function $\rightarrow$ Cross-Entropy Loss
- Number of Epochs = 20

Performance Evaluation

- Train Accuracy = 100%
- Test Accuracy = 76.33%
- F1 Score = 0.775928
- Training Time $\approx$ 23 hours

We came to realize that adding additional layers to the CNN did not necessarily increase its accuracy on the test set, but however significantly increased the training time of the model. This could be attributed to the fact that efficient pre-processing was done on the dataset, and as such, the CNN did not require many convolutional layers to extract useful features from the dataset. As such, the final CNN model followed the configuration listed above.

4.2.6 Comparative Evaluation of the Final CNN

To better evaluate the performance of our final CNN, we developed a equivalent CNN using the **Keras** library, and ran it on our pre-processed hand gesture dataset. The specifications of this CNN are detailed below.

Configuration

- Train:Test Ratio $\rightarrow$ 80:20
- 1 Convolutional Layer + Max-pool layer $\rightarrow$ ReLU activation
  - 50 filters, each of size 50×50
  - Max-pool size and stride = 2
- 1 Hidden Layer $\rightarrow$ ReLU activation
  - 35 hidden nodes
- Output Layer $\rightarrow$ Softmax activation
- Loss Function $\rightarrow$ Cross-Entropy Loss
- Number of Epochs = 20

Performance Evaluation

- Train Accuracy = 100%
- Test Accuracy = 80.67%
- Training Time $\approx$ 15 minutes

As we can see, the equivalent CNN implemented in the **Keras** library had a test accuracy of 80.67%. While this may be slightly better than our implementation of the CNN, the improvement in accuracy is only marginal. As such, we can conclude that our version of the CNN is able to achieve an almost equivalent performance as compared to the **Keras** version, proving that we have managed to implement an effective CNN model.

5 Conclusion

Through this project, we learnt about a myriad of topics, ranging from dealing with raw, non pre-processed data to developing a fully functional convolutional neural network. Starting off, we created our own dataset rather than using a pre-processed and ready-to-go dataset, to get an idea about what we should keep in mind when creating a dataset from scratch. We had a huge learning curve in this phase of pre-processing.

We looked at different possible steps for pre-processing, such as dimensionality reduction, where we reduced the dimensions of the image from a 3-channel RGB image to a single channel grayscale image. Additionally, maintaining a uniform aspect ratio for the entire image dataset was crucial. We also reduced the size of the images in the dataset, as it will enable the CNN to perform faster without much loss in accuracy.

We also learnt to use image segmentation techniques like Otsu's method, global thresholding and adaptive thresholding. Upon experimentation, we understood that Otsu's method works best for uniformly illuminated images that are captured and processed. We realized this would be a major issue and went with the method adaptive gaussian thresholding for image segmentation, which eliminated the need for uniform illumination in our dataset images. Through the above process, we recognized the effort that is required to clean the dataset, as well the importance of the pre-processing step in creating an ideal dataset, as it will eventually enhance the performance of our CNN.

In the CNN implementation portion of our project, we have gained an immense understanding of the inner workings of the convolutional neural network model, and what are the methods that are employed to optimize the CNN model. We learnt that the leaky ReLU function works best in the CNN model, while the Kaiming initialization method is the preferred technique which is utilized for initializing weight vectors and biases. We also understood that the learning rate plays a crucial role in the CNN model, and based on our experimentation, observed that the exponential cyclic learning rate function gave us the best results.

Finally, through adjusting all the possible hyper-parameters on the CNN model, we obtained the best configuration possible, and were able to achieve a maximum test accuracy of 76.33%. This was compared to an equivalent **Keras** CNN model, which was able to attain a test accuracy of 80.67%. The difference in test accuracies was observed to be marginal, and hence, we have concluded that we have managed to successfully achieve our goal of implementing a fully-fledged CNN for the purpose of hand gesture recognition.

6 Discussion

Looking back the progress that we have achieved over the course of our project, we feel that we have covered a significant amount of ground, and gained a considerable amount of knowledge in topics relating to convolutional neural networks, i.e. creating a proper dataset, pre-processing the data, and actual implementation of the CNN. However, given the huge domain of our project, there are still some minor improvements that could be incorporated. We have discussed the additional work that could be implemented in the future, as well as ideas that did not make the cut, in the section below.

6.1 Pre-processing

The following is a discussion of the ideas that we considered for the pre-processing stage of our project.

Background Subtraction

Initially we considered the technique of Background Subtraction where the algorithm extracts the image's foreground for processing. The foreground mask is calculated by subtracting the background information from the original image. This method is usually used to detect a moving object in a video which is captured by a stationary camera. The limitation of this technique is that it assumes that the camera does not move, and hence we are able to subtract the background (which is constant) and acquire the foreground.

But while implementing this we came to a standstill, as the algorithm performs best for objects which are moving, which would not be ideal for our dataset. Thus we have decided to move on with other algorithms, keeping this at bay.

Grab-Cut

One of the main ideas that we considered for the pre-processing stage was the grab-cut algorithm. As the shape of the hand was our main region of interest, we theorized that we could use this algorithm to detect the contour of the hand in the image, and use this contour to significantly define the shape of the hand. This would enable us to get a clearly segmented image, with the shape of the hand being outlined perfectly, while also eliminating any noise present in the image.

The main difficulty we encountered with this algorithm was that it requires user interference to point to the region of interest in the image. This made it troublesome to specify the region of interest for each image of the dataset. In order to eliminate the user interference to specify the interested region, we created a function that would automate this process. This function creates a bounding box on the largest area in the image and then extracts that from the image. As our learning model has a training dataset of only hand signs, the background information gets eliminated.

However, we had some issues with the grab-cut algorithm itself, as it was unable to perfectly detect the contour of the hand in certain images. This limited the effectiveness of this grab-cut as a valid pre-process algorithm, and hence, this idea was not utilized in our final implementation. But, this algorithm did show some promising results on a few images, and given some additional time, we could have possibly ironed out the issues present in this algorithm. The image below shows an example of an image that we obtained by using the grab-cut algorithm.

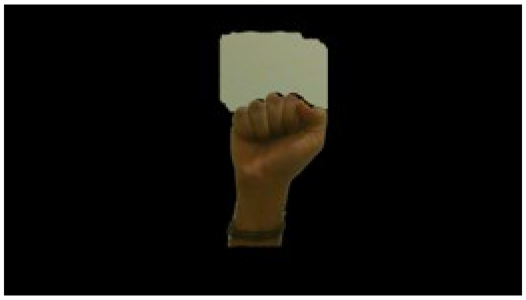

Figure 17: Grab-cut technique on the fist sign image

Region Growing

This is a simple region based image segmentation technique, that we considered earlier on in the project. It is more of a pixel based segmentation algorithm, as it depends on an initial seed point. The idea of this approach is that it checks the nearby points and looks at their properties, and the pixel is then added to the region with which it is considered to share the highest similarity. As a result, this technique is regarded to be more of a clustering method.

We managed to do some simple testing based on this algorithm, but due to the slight complexity of the algorithm in comparison to other segmentation algorithms, we did not have sufficient time to perfect the implementation of this algorithm. However, this is one algorithm that could definitely be considered in the process of segmenting the hand from the background.

6.2 Improving the CNN

The main issue that we had with our CNN was that the training time was too long, especially when convolution and max-pool layers were introduced into our model. This comes from the fact that we were using our CPU to train the CNN model, which is highly limited in parallelizing this process. To improve our model in this aspect, we were planning to train our model on the GPU, as the architecture of the GPU is such that it allows for tasks to be highly parallelized (in comparison to the CPU).

One possible method to achieve the above would be to utilize the **numba** library available in python, which utilizes CUDA-enabled Nvidia graphics cards to parallelize the execution of the code, resulting in a much lesser training time. This required significant alterations to be made to the python code, and considering the large size of the code, we were unable to complete this procedure for the final implementation. However, this is definitely something that we could incorporate in the future.

References

[1] Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B. et al. (2017). Recent Advances in Convolutional Neural Networks. Retrieved 10 April 2020, from https://arxiv.org/pdf/1512.07108.pdf.

[2] Islam, M., Hossain, M., Islam, R., Andersson, K. (2019). Static Hand Gesture Recognition using Convolutional Neural Network with Data Augmentation. Retrieved 10 April 2020, from https://ieeexplore.ieee.org/document/8858563.

[3] Khan, R., Ibraheem, N. (2012). Survey on Gesture Recognition for Hand Image Postures. Retrieved 10 April 2020, from https://pdfs.semanticscholar.org/085d/4a026eb425ff87094857e3a0ad4324419468.pdf.

[4] Krizhevsky, A., Sutskever, I., Hinton, G. (2012). ImageNet Classification with Deep Convolutional Neural Networks. Retrieved 10 April 2020, from https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[5] Lin, H., Hsu, M., Chen, W. (2020). Human Hand Gesture Recognition Using a Convolution Neural Network. Retrieved 10 April 2020, from https://ieeexplore.ieee.org/document/6899454.

[6] Mohanty, A., Rambhatla, S., Sahay, R. (2016). Deep Gesture: Static Hand Gesture Recognition Using CNN. Retrieved 10 April 2020, from https://link.springer.com/chapter/10.1007/978-981-10-2107-7_41.

[7] Molchanov, P., Gupta, S., Kim, K., Kautz, J. (2015). Hand Gesture Recognition with 3D Convolutional Neural Networks. Retrieved 10 April 2020, from https://ieeexplore.ieee.org/document/7301342.

[8] Murthy, G., Jadon, R. (2010). Hand Gesture Recognition Using Neural Networks. Retrieved 10 April 2020, from https://www.researchgate.net/publication/224120227_Hand_Gesture_Recognition_using_Neural_Networks.

[9] Pinto Jr., R., Borges, C., Almeida, A., Paula Jr., I. (2019). Static Hand Gesture Recognition Based on Convolutional Neural Networks. Retrieved 10 April 2020, from http://downloads.hindawi.com/journals/jece/2019/4167890.pdf.

[10] Chen Wang and Yang Xi.(2015) Convolutional Neural Network for Image Classification Retrueved 20 April 2020, from http://www.cs.jhu.edu/ cwang107/files/cnn.pdf

[11] Xavier Glorot, Yoshua Bengio.(2010).Understanding the difficulty of training deep feedforward neural networks. Retrieved 23 April 2020, from http://proceedings.mlr.press/v9/glorot10a.html .

[12] Leslie N. Smith. (2015). Cyclical Learning Rates for Training Neural Networks. Retrieved 23 April 2020, from https://arxiv.org/abs/1506.01186

[13] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. (2015). Numba: a LLVM-based Python JIT compiler. Retrieved 23 April 2020 from https://doi.org/10.1145/2833157.2833162