Implementation of CNN from scratch for Hand Gesture Recognition

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

Group 3

Department of Computer Science, University of Maryland, Baltimore County

April 2020

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 computer will have to 'learn' the features of each hand gesture and tries 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 evaluate our CNN.

Keywords— Convolutional neural networks, Hand gesture recognition, Pre-processing

[*]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. 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 utilizing each individual pixels. Thus, they do not work as well on images, in which features of the images exist in 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 aim to use the spatial locality of images to extract useful features, which then aid in classifying the images more accurately. As such, 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. As such, CNNs can also understand the complex and non-linear relationships amongst the images. Therefore, 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 convolutional neural network by ourselves, without utilizing any 3rd party libraries or frameworks. By working on the whole process manually, we will be able to gain a better understanding of the working of convolutional neural networks 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 traced back to, as it pioneered the application of the CNN structure to classify images in a supervised data set, and was considered to be groundbreaking research during its time. The authors of (3) provide a comprehensive survey on Hand Gesture Recognition systems, and also give insights on 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 such as 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 would also help us create a better test and train split from the original dataset.

Raimundo F. Pinto, et.al. at (9) works 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 also compares the output produced when using various numbers of layered 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, and utilizing convolution and max-pool layers, 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, focusing on how feed-forward and back-propogation 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.

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, and the details of the steps followed to create the dataset, and conducting pre-processing of the images are detailed 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.

3.1.2 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. Different steps done in order to preprocess the dataset are mentioned below.

Uniform size and aspect ratio

One of the basic steps involved in creating a data set is to ensure that each image in the data set 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 in a uniform rectangular form of dimensions 160×90. Through this step, all the images we collected through the creation of the data set are converted into uniform images of constant size and aspect ratio.

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 to be calculated in our CNN. If the dimensions of the input images are high, then the number of variables utilized in the CNN would be larger, and the total time taken for one iteration of the CNN would be longer as well. Thus reducing the size of the image also aids in reducing the overall training time of the CNN.

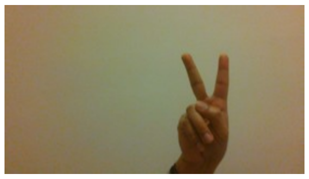
Figure 1: Victory sign image of 1920 $\times$ 1080 dimension

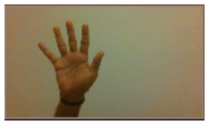
Figure 2: Palm sign image resized to 160 $\times$ 90 dimension

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.

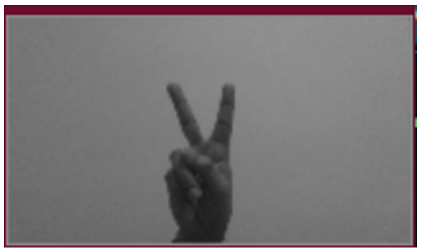
Figure 3: Grayscale version of the victory hand sign

Image Segmentation

As we are interested in only the hand signs, we should use only these features for training the neural network and try to mitigate all the other unnecessary information. The pre-processing done until this stage have resulted in a dataset consisting of images in grayscale. To further highlight the shape of the hand, and decrease the effect of the background of the image, we segment the image using an automatic image thresholding technique known as Otsu's threshold.

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 threshold value and if the value of the pixel is more than 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 threshold value removes information that is not required in identifying the hand gesture depicted in the picture. 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 image is uniformly illuminated.

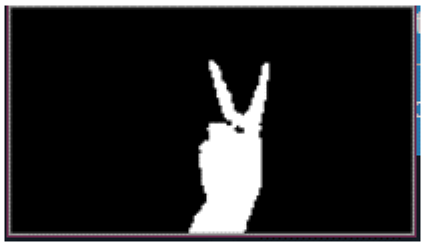

Figure 4: Binary (black & white) image of the victory hand sign

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. While this would not be much of a significant improvement to the version of the neural network developed earlier, this step was crucial in enhancing our understanding the backpropagation of the partial derivative of the loss with respect to each variable 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, as well as pre-defined helper functions, which aided in calculating the derivative of the respective activation functions. These values were then stored in the corresponding helper arrays, which were then finally used 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. 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. 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 with other CNN factors

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 CNNs that would enhance their 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. The findings that we obtained by experimenting with these factors are detailed below in the *Results* section. Another important factor that significantly affects the performance of the CNN is the learning rate. While we are currently using a constant learning rate in our CNN implementation thus far, plans to implement a learning rate function, that varies with the number of iterations/epochs, are definitely being considered. These plans are further delineated in the *Plan for Completion* section below.

3.2.5 Experimentation of various configurations of hyper-parameters

The final step in our CNN methodology would be to conduct an extensive amount of testing on our CNN model, by varying the hyper-parameters of this model, 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 will be conducting 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.

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 relating to 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 160×90, converted from RBG to grayscale, and finally converted to binary using Otsu's thresholding algorithm. These final binary images are then provided as input to the CNN.

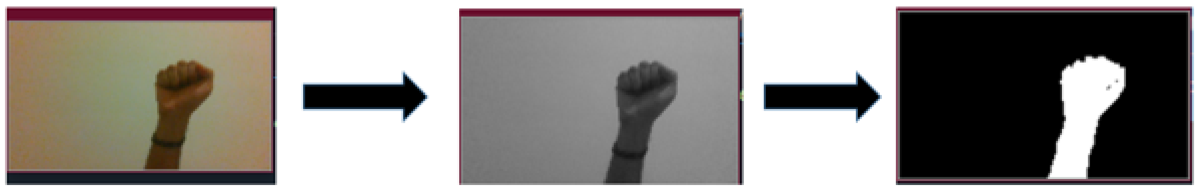

Figure 5: Process of conversion from RGB image to grayscale image to binary image

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 the videos which are 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.

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 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 = 0.05
- Train:Test Ratio $\rightarrow$ 80:20
- Output Layer $\rightarrow$ Softmax activation
- Loss Function $\rightarrow$ 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 $\rightarrow$ 0.05
- Train:Test Ratio $\rightarrow$ 80:20
- 1 Hidden Layer $\rightarrow$ Sigmoid activation
  - 10 hidden nodes
- Output Layer $\rightarrow$ Softmax activation
- Loss Function $\rightarrow$ 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 → 0.005
- Train:Test Ratio → 80:20
- 1 Convolutional Layer + Max-pool layer → Leaky ReLU activation
  - 50 filters, each of size 20×20
  - Max-pool size and stride = 2
- 1 Hidden Layer → Sigmoid activation
- Output Layer → Softmax activation
- Loss Function → Cross-Entropy Loss

Performance Evaluation

- Train Accuracy = 98%
- Test Accuracy = 66%
- Training Time = 38 hours

4.2.4 Experimentation on Other Hyper-parameters

After coming up the above model, we decided to optimize the other factors of the CNN before proceeding to experiment with other hyper-parameters. For this purpose, we mainly focused on two factors that we could play around with: the activation function, and the weight vector initialization method. 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 on 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.

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

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.

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, allowing the neural network to learn more effectively. 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.

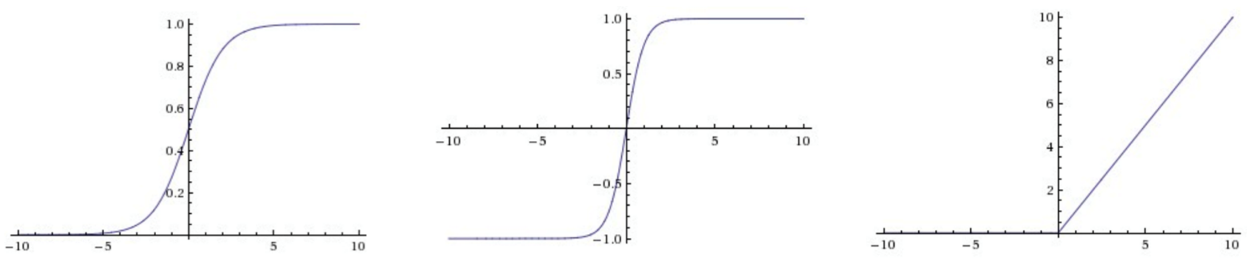

Figure 6: Graphical Representation of the Sigmoid, Tanh and ReLU activation functions (from left to right)

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 additional iterations, and hence they 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.

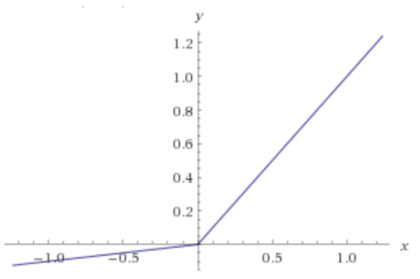

Figure 7: Graphical Representation of the Leaky ReLU activation function

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 is minimized, and the chances of reaching the global minima is 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. As a result, we assume this claim to be true, and have chosen this method of initialization for our current CNN implementation.

5 Plan for Completion

Our project is heading along in the right direction, and we have definitely made some significant progress as described above. However, there are still some aspects of the project which can be enhanced and improved further. The following lists our plan for completing each section of the project that we are currently working on.

5.1 Pre-processing

Our current focus is on minimizing and simplifying the input provided to the CNN, so that it would have less data to compute. We have tried several approaches for the pre-processing stage, due to its crucial role in improving the performance of the CNN. Listed below are some of the techniques that we have researched on, and currently considering for our final implementation.

Grab Cut

As we require only the essential features of the image (i.e. the shape of the hand), we have decided to extract only the required part from the image without losing any edge information. One technique to achieve the above result is through grab cut. This is an image segmentation technique that is based on graph cuts. This technique makes use of the bounding box rectangle specified by the user and checks for a color distribution of the target image and background. We were able to successfully extract the essential part of the image without losing the edge information. The image below provides an example of the output of the grab cut algorithm.

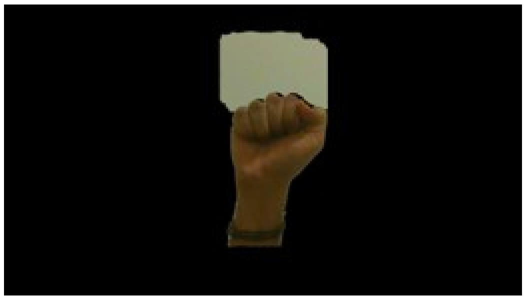

Figure 8: Grab cut technique on the fist sign image

We are currently working on automating the user specified rectangle to reduce manual work, so that we can automate this process on the dataset, hence filtering the dataset to contain only the essential features and ignore the noise in other areas of background.

Region Growing

This is a simple region based image segmentation technique. It is more of a pixel based segmentation as it depends on an initial seed point. The idea of this approach is that it checks the nearby points and looks at its properties, and if they are considered to be similar then that pixel is added to the region of highest similarity. As a result, this technique is regarded to be more of a clustering method. We are currently looking at its implementation, and other use cases that could contribute meaningfully to our project.

5.2 Improving the CNN

In terms of plans for finishing the CNN portion of the project, the following is a list of all the tasks that we are aiming to complete.

1. The bulk of the work that remains would be experiment and evaluate different hyper-parameter settings of the implemented CNN. The following are a list of hyper-parameters that we can tweak to achieve optimal performance on the hand gesture dataset:

   - The number of layers used of each type – convolution & max-pool, fully-connected
   - The number of filters in the convolutional layers
   - The dimensions of these filters
   - The size and stride of max-pool layers
   - The number of neurons in the fully-connected layers

2. Another task that we are yet to implement is the incorporation of learning rate functions, instead of a constant learning rate that is currently being used. A learning rate function will allow for faster convergence of the neural network by basing the learning rate function on a suitable function that changes the value of the learning rate according to the number of iterations/epochs that have passed.

   For example, 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 iterations/epochs increases, the learning rate value exponential decreases, causing the magnitude of the updates to the weights and biases to decrease in a similar manner, which would be ideal, as the weights and biases would only need minor updates as they reach values close to the optimal values, i.e. convergence of the CNN.

   We are currently considering a variety of adaptive learning rate functions, which include:

   - Time-based decay
   - Step decay
   - Exponential decay
   - Cyclic learning rates

   As with the other factors of the CNN that we experimented with, we will be implementing and evaluating the effect of each of these learning rate functions on the performance of the CNN. After conducting appropriate analysis of the above, we will then choose the best learning rate to be implemented for the final version of our CNN.

3. We are also researching on possibly changing the framework of our code, so that it could be trained on the GPU instead of on the CPU. As seen in the results section of our project, one of the limiting factors in our aim of determining the best CNN configuration is the extremely large amount of time it takes to train the model. This is attributed to the fact that currently, the training of the model is being done on the CPU, and hence the process takes a considerable amount of time due to the extensive

amount of calculations being done at this stage.

However, the training process could be significantly sped up if it were to be done on the GPU instead of the CPU. The architecture of the GPU is such that it allows for tasks to be highly parallelized (in comparison to the CPU), which reduces the overall time taken to complete the calculations by a significant factor. One of the possible implementations of the aforementioned would be to use 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 execution time. However, this might require some alterations to be made to the code, which we are currently working on.

4. Finally, we are working on a better evaluation of the different versions of the CNN we have implemented thus far. Currently, we are only taking into account 2 performance metrics; the accuracy of the model on the hand gesture dataset, and the total amount of time taken to train the model. While these two metrics give us a good basic overview of the performance of each model, in order to achieve a better in-depth analysis of the models, we would need a few more additional metrics. For this purpose, we are considering the incorporation of two additional performance metrics, as part of our final stage of the project, which are detailed below:

   (i) Implementation of the confusion matrix during the evaluation stage of the model, so that additional helpful metrics can be calculated, such as:

      - Recall
      - Precision
      - 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 we do.

   (ii) Implementation of an equivalent model using a deep-learning library in Python, such as **Keras**. This will provide us with useful information as to whether the performance of the models that we have manually implemented is on par with the respective equivalent models implemented in **Keras**. We can also obtain a side-by-side comparison between how well our CNN model performs against an equivalent model implemented in **Keras**, allowing us to better evaluate the model that we have created.

   Working on the above two additional modes of evaluation will altogether provide us with more information about the performance of our CNN model, and hence allowing us to better evaluate our CNN implementation. This will in turn aid us in fine-tuning our model with more precision, enabling us to produce a more optimized CNN for the final version.

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