# Gesture Recogntion using ChaLearn IsoGD

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in
Mathematics

by

**Aron G.**
M013704

Under the guidance of
**Prof. Utpal Garain**
Indian Statistical Institute, Kolkata

Nurturing talent...promoting excellence
मौलिक विज्ञान प्रकर्ष केन्द्र

Centre for Excellence in Basic Sciences
University of Mumbai
Kalina, Mumbai, India – 400098

Autumn Semester 2017

# Acknowledgement

I would like to express my profound sense of gratitude to Prof. Utpal Garain for providing me this opportunity to learn under his invaluable guidance. I am highly indebted to Mr. Akshay Chaturvedi for his constant comments and suggestions. I would specially thank Prof. Balwant Singh for the arrangements he made for the project. I would also like to thank my classmates and parents for their prolonged support. I would like to thank Indian Statistical Institute, Kolkata for providing me with the adequate facilities. I also acknowledge the support provided by Centre for Excellence in Basic Sciences.

## Abstract

This thesis explores the implementations of 3D convolutional neural networks on ChaLearn Isolated Gesture Dataset to achieve a better state of the art. This gives an insight towards video processing techniques in general, as they are often considered one of the toughest pattern recognition problems. The limitations of the existing methods and the scopes for further improvements are also discussed. The topic was chosen as the improvements in human interactions between a computer not only aids the differently abled, but also the ones in need of a hassle free interface.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

"The secret of getting ahead is getting started."

*Mark Twain*

## 1.1 Motivation

Recognizing human actions is a big leap from mechanic inputs when it comes to human-computer interactions and what better way to execute it than to make the computer to understand hand gestures. Though this problem has been around for a while, no perfect solution has come into existence as of now. This is mainly due to the inefficiency of the current models and various other factors that we will explore in this thesis.

## 1.2 Objectives

The main goals of this thesis are the following:

- To study the Chalearn Isolated Gesture Dataset thoroughly.
- To look at various video processing techniques.
- To make a model and train the dataset to achieve a better state of the art.

# Chapter 2

# Overview of Neural Networks

## 2.1 A brief history

Artificial neural networks, as the name suggests are computing systems that are inspired from the biological neurons in a human brain. Unlike traditional algorithms, neural networks cannot be 'programmed' or 'configured' to work in the intended way. Just like human brains, they have to learn how to accomplish a task.

They started becoming popular in the late 1940's and came to a sudden halt in the early 170's even though there were many significant progress in the field. This was mainly because of the lack of computational power as the networks were too complex to be trained. From the 2000's, the Graphic Processing Units(GPUs) started retailing and were readily available to the public. These GPUs reduced the computational time drastically and neural networks started blooming.

## 2.2 Perceptrons

Perceptron is the most basic and fundamental form of an artificial neural network. They were developed in the 1950s.

A perceptron takes several binary inputs, $x_1, x_2, \ldots, x_n$ and produces a single binary output: Each input $x_i$ has a weight $w_i$ associated with it that represents the relevance of that particular input. The neuron's output, 0 or 1 is determined by whether the weighted sum, $\sum_{i=1}^{n} w_i x_i$ is less than or bigger than some *threshold value*. Just like the weights, the threshold is a

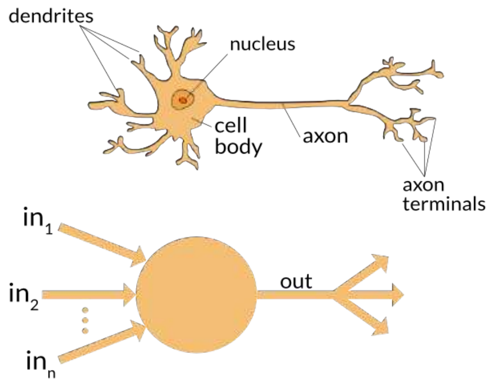

Figure 2.1: How a perceptron compares to a biological neuron.[4]

real number which is a parameter of the neuron. That is,

$$output = \begin{cases} 0 & if \ \sum_{i=1}^{n} w_i x_i \leq threshold \\ 1 & if \ \sum_{i=1}^{n} w_i x_i > threshold. \end{cases} \tag{2.1}$$

Writing $x_i$'s and $w_i$'s as vectors $x$ and $w$ enables $\sum_{i=1}^{n} w_i x_i$ to be written as the following dot product, $w \cdot x = \sum_{i=1}^{n} w_i x_i$. Now, by bringing the threshold value to the other side of the inequality and calling it bias, $b = -threshold$, the equation(1) could be re written as,

$$output = \begin{cases} 0 & if \ w \cdot x + b \leq 0 \\ 1 & if \ w \cdot x + b > 0. \end{cases} \tag{2.2}$$

Bias can be thought of as a measure of how easy it is to get the perceptron to output a 1 or biologically, it is a measure of how easy it is to get the perceptron to fire.

## 2.3 Multi-Layered Perceptrons

An MLP, as the name suggests, is a collection of perceptrons arranged as layers, just like the neural networks in human brain. Figure 2.3 shows a typical neural network, with two hidden layers and an output layer. Each perceptron in the first hidden layer will have weights associated with each input from the input layer just like a typical perceptron and the action of

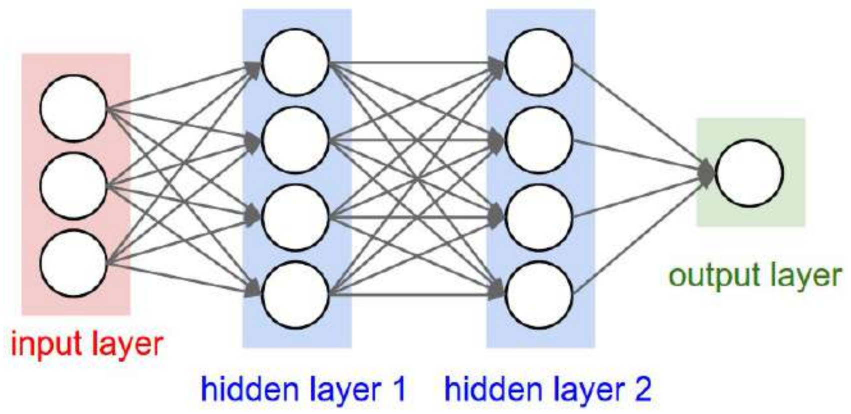

Figure 2.2: A typical neural network.[5]

these weight vectors on input vector can now be seen as an action of a weight matrix on the input vector. That is, if the input vector, $X$ is of dimension $n$, $X = (x_1, x_2, \ldots, x_n)$ and there are $m$ perceptrons in the first hidden layer, each with $n$ weights for each $x_i$, then they form a weight matrix, W, as shown,

$$W = \begin{bmatrix} w_{11} & w_{12} & \ldots & w_{1n} \\ w_{21} & \ddots & & \\ \vdots & & \ddots & \\ w_{m1} & & & w_{mn} \end{bmatrix},$$

and the bias vector be, $B = (b_1, b_2, \ldots, b_m)$, where each $b_i$ is associated with the $i^{\text{th}}$ perceptron in the first hidden layer. Now, the output after the action of the first hidden layer will be $WX + B$ and it will have a dimension $m$. The next hidden layer may have $m'$ perceptrons, with weight matrix $W'$ and bias vector $B'$. The output from that the second hidden layer will be will be $W'(WX+B)+B'$, having dimension $m'$. The output will now go to an output layer which has a length that is equal to the number of classes, in case of a classification problem. Here, the output layer will have $m'$ classes, and say if the predicted class is $k$, then the output vector will be $(0, 0, \ldots, 1, \ldots, 0)$ with 1 in the $k^{th}$ position.

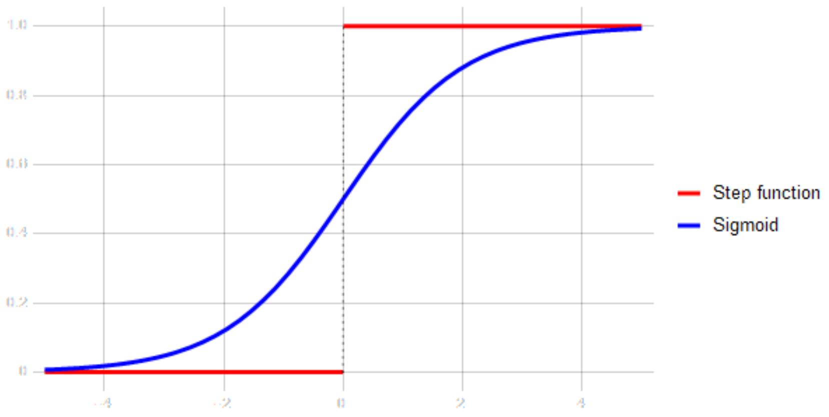

Figure 2.3: A sigmoid and a step function compared.

### 2.3.1 Sigmoid Neurons

Now we need to devise a learning algorithm that can tune these wights without direct human intervention. For that, we need to use calculus on the activation function .The perceptrons however are unable to do this because the bias is a step function, which is discontinuous at the threshold value.

Hence, we come up with activation functions of which sigmoid is the most primitive one. Activation functions are smooth and also gives the network a sense of non-linearity which is essential to make the model resemble the real world situations. The sigmoid function, $\sigma$ is defined as:

$$\sigma(z) = \frac{1}{1+e^{z}}. \tag{2.3}$$

Unlike the perceptrons, the sigmoid will have any real number between 0 and 1 as output and hence it appears as a ”smoothed” perceptron as shown in Figure 2.3. The sigmoid however does not completely abandon the perks of a perceptron, since it behaves like one when $z = w \cdot x + b$ is very positive and when it is very negative . That is, when $z$ is very positive, $e^{-z} \approx 0$, making $\sigma(z) \approx 1$ and when $z$ is very negative, $e^{-z} \approx \infty$, making $\sigma(z) \approx 0$. Now that we have a smooth activation function, let's see how small changes in weights and bias are going to cause small changes in the outputs. Let the small changes in the weights be $\Delta w_j$ and in the bias be $\Delta b$, then the small changes in the output as we expect in Figure 2.4, $\Delta$output is approximated as follows,

$$\Delta\text{output} \approx \sum_j \frac{\partial\text{output}}{\partial w_j}\Delta w_j + \frac{\partial\text{output}}{\partial b_j}\Delta b. \tag{2.4}$$

Now that we know $\Delta$output is a linear function of $\Delta w_j$'s and $\Delta b$, we'll be

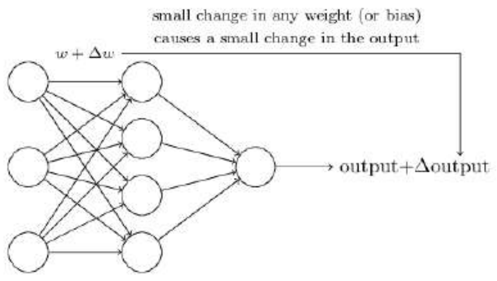
small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+$\Delta$output

Figure 2.4: How the changes in input affects the output. [7]

able to fine tune the weights and bias to get the desired output, but for that we need to first know if the further tuning is required and if so, by how much. What helps us to achieve this is something called a cost function, or a loss function, as it is commonly referred to. The loss function calculates the amount of loss happening in the training process and thereby telling us how well the model is fitting the training data. There any many kinds of loss functions of which quadratic loss function is the one we well look into for now. It is also referred to as mean squared error and is defined as follows,

$$L(w, b) = \frac{1}{2n}\sum_{i=1}^{n} \left\| p(x_i) - y(x_i)^2 \right\|, \tag{2.5}$$

where $w$ and $b$ denotes all the weights and biases respectively in the network, $n$ is the total number of training inputs, $p(x_i)$ is the predicted and and $y(x_i)$ is the expected output of the $i^{\text{th}}$ training input, $x_i$. Observe that $L(w, b)$ is never negative and $L(w, b) \approx 0$, when the predicted value is approximately equal to the expected value for all the inputs. So now the task of tuning the weights and biases becomes just a minimizing problem and we will tackle it using the gradient descent method.

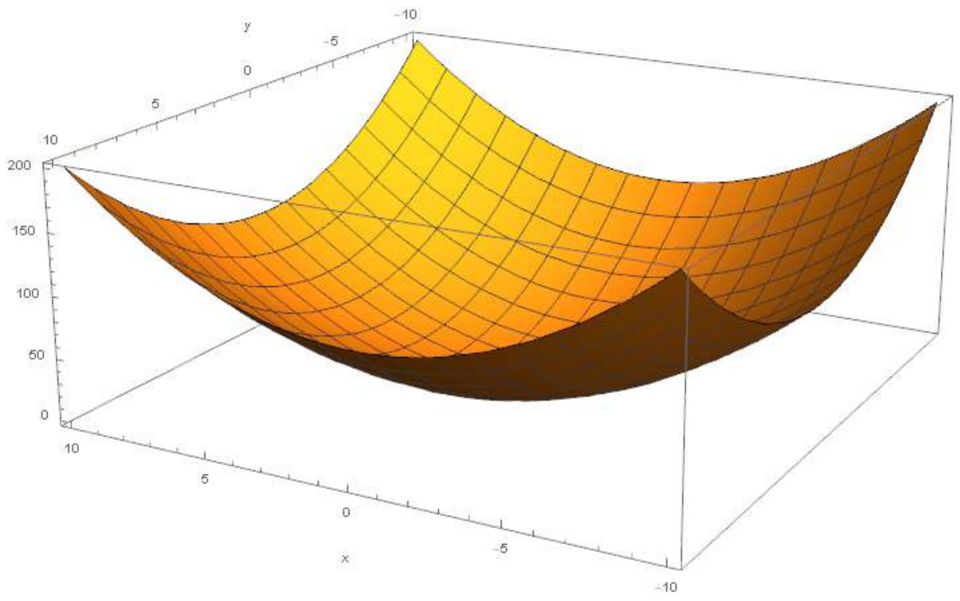

Figure 2.5: An ideal cost function with two parameters.

### 2.3.2 Gradient descent method

Given a function f(x,y) as shown in figure 2.5, we need to find the global minimum. One can do this analytically by calculating the partial derivatives, but when it comes to the loss function, it is going to be extremely time consuming as there are thousands of variables. Now, imagine the function as a valley and a ball is placed at any random point of it, we will try to simulate the rolling down of the ball to the bottom. Let the ball be moved a small amount $\Delta x$ in the $x$ direction and $\Delta y$ in the $y$ direction, then change in the function $f$ will be,

$$\Delta f \approx \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y, \tag{2.6}$$

which can be re written as,

$$\Delta f \approx \nabla f \cdot \Delta z, \tag{2.7}$$

where $\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$ and $\Delta z = (\partial x, \partial y)$. Now we need to make these changes, $\Delta x$ and $\Delta y$ in such a way that we proceed "downwards the hill" always, that is, to make $\Delta f$ always negative, and for that we choose $\Delta z$ such that,

$$\Delta z = -\eta \nabla f, \tag{2.8}$$

where $\eta$ is a small positive quantity called as learning rate. Now, equation 2.7 becomes,

$$\begin{aligned} \Delta f &\approx -\eta \nabla f \cdot \nabla f, \\ \Delta f &\approx -\eta \left\| \nabla f \right\|^2. \end{aligned} \tag{2.9}$$

Since $\left\| \nabla f \right\|^2 > 0$ always and $\eta > 0$, $\Delta f$ will always decrease and hence we can update $z$ as follows,

$$z \rightarrow z^{'} = z - \eta \nabla f \tag{2.10}$$

Now, we need to apply this method on Equation 2.5 to minimize it. Since the variables in a loss function are the weights and the biases, their update rule is as follows,

$$\begin{aligned} w_p \rightarrow w_p' &= w_p - \eta \frac{\partial L}{\partial w_p} \\ b_q \rightarrow b_q' &= b_q - \eta \frac{\partial L}{\partial b_q} \end{aligned} \tag{2.11}$$

This can be done over and over till the minimum of loss function is reached, but there is a small issue that we need to take care of first. That is, in the way the loss function was defined as in Equation 2.5, it looks like $L = \frac{1}{2} \sum_{i=1}^{n} L_{x_i}$, where it is an average over the loss of individual training inputs, $L_{x_i} = \frac{\left\| p(x_i) - y(x_i) \right\|^2}{2}$. If for the cost function, we consider all the inputs, then for finding the gradient $\nabla L$, we need to take the average of all the $\nabla L_{x_i}$'s computed separately, but when the number of training inputs is very large, this can slow down the learning process drastically.

To solve this issue, we use something called **stochastic gradient descent**, which estimates the gradient $\nabla L$ by computing the $\nabla L_i$'s of only a small sample of training inputs chosen randomly and then taking their average score. This randomly chosen inputs are called a mini-batch. This makes computation of $\nabla L$ very economical. To see how this works, let us say we break down the training data into $k$ such mini batches of size $m$, and denote them by $B_1, B_2, \ldots, B_k$. Note that m is chosen in such a way that, given a batch, say $B_1$,

$$\frac{1}{m}\sum_{x\in B_1}\nabla L_x \approx \frac{1}{n}\sum_{i=1}^{n}\nabla L_{x_i}, \tag{2.12}$$

where the RHS is basically $\nabla L$ and the LHS is the average of gradients of inputs in $B_1$. The update rule for weights and biases now will be,

$$\begin{aligned} w_p \to w'_p &= w_p - \frac{\eta}{m}\sum_{x\in B_1}\frac{\partial L_x}{\partial w_p} \\ b_q \to b'_q &= b_q - \frac{\eta}{m}\sum_{x\in B_1}\frac{\partial L_x}{\partial b_q}. \end{aligned} \tag{2.13}$$

After this is done, we will pick up the next batch $B_2$ and so on until $B_k$, that is when we exhaust all the training data, and it is called one **epoch** of training. Now, we start another epoch, starting with the weights and biases obtained from the first epoch.

## 2.4 Convolutional Neural Networks

The regular neural networks find it hard to take full-scale images as inputs as they account to ridiculous amount of parameters which might end up in overfitting. This is where Convolutional Neural Networks(CNN) come to picture. Though they share a similar architecture with the regular neural networks, they are tailor made to deal with images.

A typical RGB image, which appears two-dimensional, is three-dimensional according to the computer, since the colour channel adds another dimension, apart from the width and height. CNN does not flatten the image into a column vector like regular neural networks, instead it reads it as a 3D-matrix if RGB or 2D-matrix if greyscale, and the neurons in CNN are also formed likewise as shown in Figure 2.6.

There are three main types of layers for a CNN, a **convolutional layer**, a **pooling Layer**, and a **fully-connected layer**.

A convolutional layer uses something called **filters** (or **kernels**) to perform convolution on the input. Let us say we have an RGB image I, having a resolution of $m \times n$ pixels as input, then it will be read as an $m \times n \times d$ tensor, $m$ being the width, $n$ being the height and $d$ being the depth, in

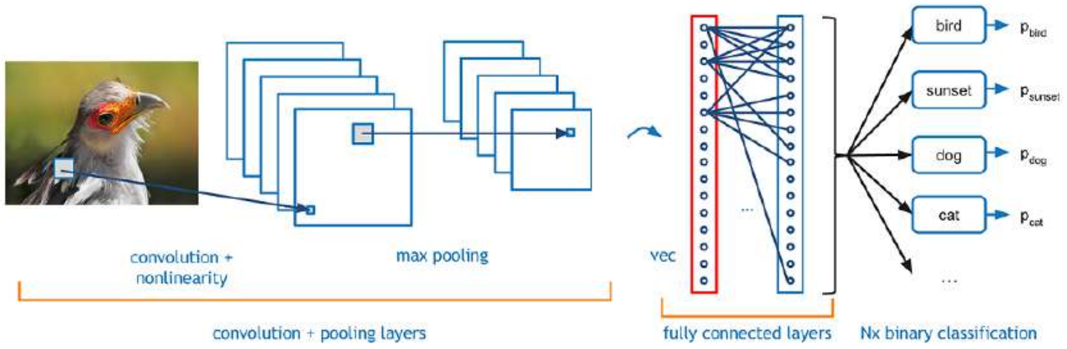

Figure 2.6: Basic architecture of CNN. [6]

case of greyscale images, $d = 1$, but in this case, $d = 3$, for the three color channels, red, green and blue. Now, there will be a kernel, which is nothing but another tensor of smaller dimension, say $k \times k \times 3, (k < m)$ and it will "scan" over the input, by taking the Hadamard product, which is computed as shown,

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \odot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \end{bmatrix},$$

with a fixed step size $s$. This step size it called a **stride** and this is the convolving process. After taking a Hadamard product a bias is added to the output. Figure 2.7 shows convolution with a greyscale image as input with shape $7 \times 7$ and a kernel with shape $3 \times 3$, taking strides of 2 and with bias 0.

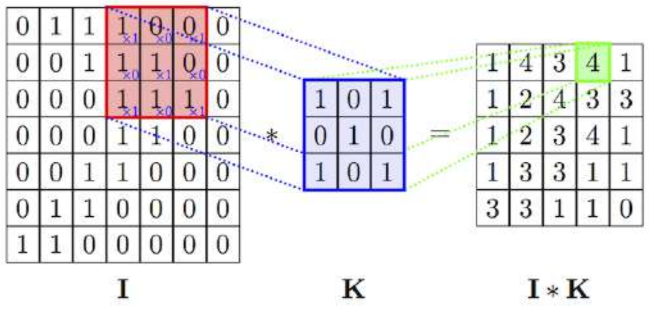

Figure 2.7: Convolution of a greyscale image.

In case of an RGB image, there will be a kernel for each colour channel, and their outputs will be added together after convolving. Also, there are are many such kernels per convolutional layer with different weights. If the strides taken by the kernel is not compatible with the input layer, meaning if the kernel overshoots the input image while taking a stride, then there is going to be a problem. This however has a solution called **zero padding**. The way this works is, it adds zeroes on the boundary of the input matrix so as to make the strides by the kernel compatible, let us say we add zero padding $z$ to the input image. Figure 2.8 shows how zero padding is done.

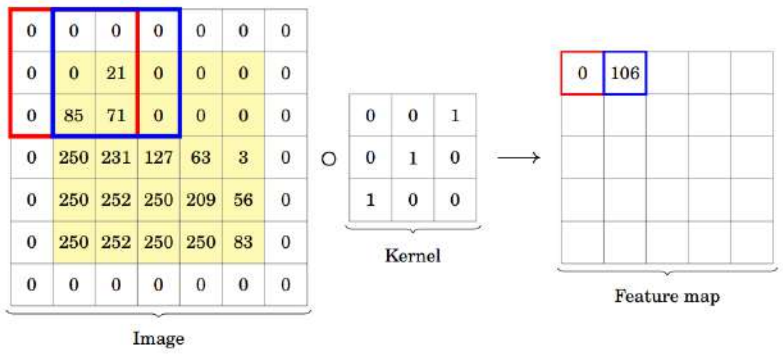

Figure 2.8: Zero padding.

In case of the RGB image I that we took, the new shape after the first convolution will be $m' \times n' \times d'$, where $m' = \dfrac{(m-k+2z)}{s} + 1$, $n' = \dfrac{(n-k+2z)}{s} + 1$ and $d'$ is the number of kernels used. Notice that the width and height is reduced now, whereas the depth is increased since $d$ is usually kept more than 3. Figure 2.9 shows convolution with two kernels of shape $3 \times 3 \times 3$ with a bias of 1 and 0 , acting on the input image of shape $5 \times 5 \times 3$ with a zero padding of 1.

Like we did in MLP, after a hidden layer, we need to add a non-linearity, but we cannot afford to use sigmoid here as it has something called a *vanishing gradient problem*, which is caused by the exponential drop in gradient of the loss function while backpropagating a through lot of hidden layers. To counter this issue, we use another activation function called **ReLU**, which stands for Rectified Linear Units, and is defined as follows,

$$f(x) = max(0, x) \tag{2.14}$$

This will act upon the output from the first convolution before being passed to the next layer.

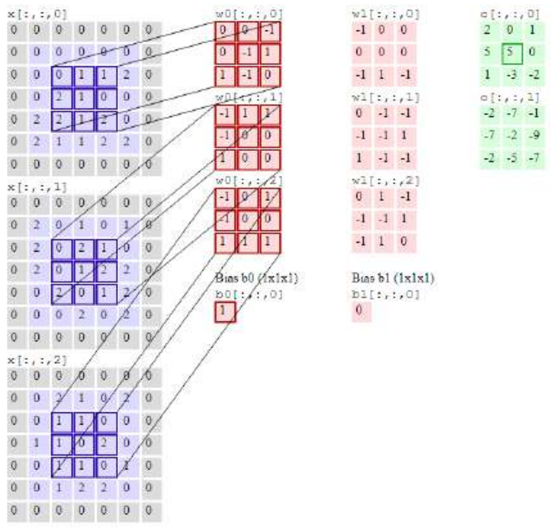

Figure 2.9: Convolution of an RGB image. [5]

After the convolution is done, comes **pooling** which are similar to convolution as pooling also has filters which ”scan” through the input with specific strides. They progressively reduce the spatial size of the representation to reduce the amount of features and the computational complexity of the network. The main reason for the pooling layer is to prevent the model from overfitting. There are two main types of pooling, **max-pooling** and **average pooling**. As the name suggests, the former computes the maximum value among the entries inside a given filter and the later computes the average of those entries, as shown in Figure 2.10. In our case, if keep the filter size $k' \times k'$ and stride $s'$, then, the output will have shape $m'' \times n'' \times d$, where $m'' = \dfrac{(m' - k')}{s'} + 1$ and $n'' = \dfrac{(n' - k')}{s'} + 1$. Notice that the width and height is further reduced now, whereas the depth remains the same.

Now, there will be a another convolutional layer with different set of kernels, after which it will go through another pooling layer and the input will get further reduced in width and height and the depth will keep on increasing. This process is continued until the width and height is 1, after which it forms the fully-connected layer. Fully connected layer is basically just MLP with the input vector as the $1 \times 1 \times d'$ shaped output from all the

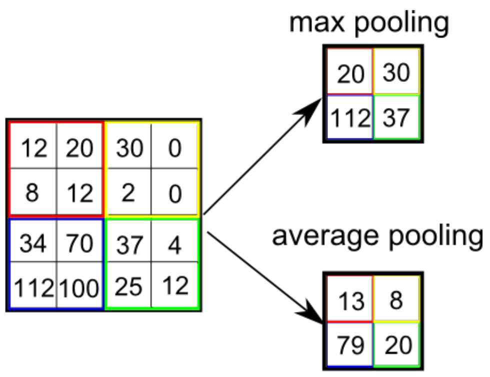

Figure 2.10: Max pooling and average pooling. [5]

convolutions, where $d'$ is the final depth. Callback how MLPs work and how the loss function is calculated and is backpropagated to update the kernels. After the desired validation accuracy is reached, the network is ready to take in test images.

## 2.4.1 3D CNN

3D CNN is an extension of CNN, meant for video processing. The inputs here will be videos, which are collection of images or frames, and hence the architecture needs to be modified likewise, as shown in Figure2.11.

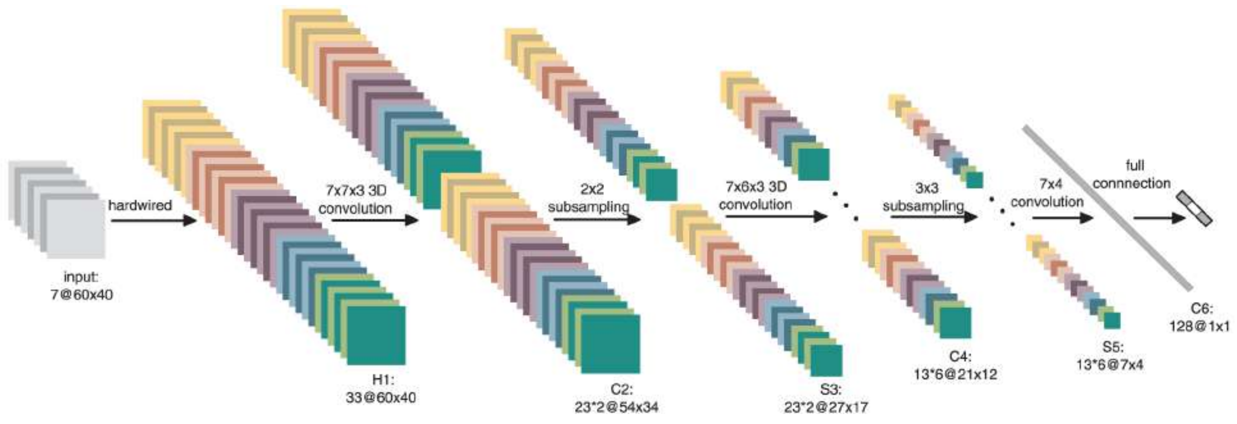

Figure 2.11: Architecture of a 3D-CNN.

# Chapter 3

# Dataset

## 3.1 The Chalearn Isolated Gesture Dataset

The dataset is a modified version of the ChaLearn Gesture Data released as part of the ChaLearn looking at people challenge 2011 [3], which had about 50,000 thousand videos with one video containing one or more gestures. This is however a harder problem compared to the modified one which has only one gesture per video, and hence the later one is taken for the study.

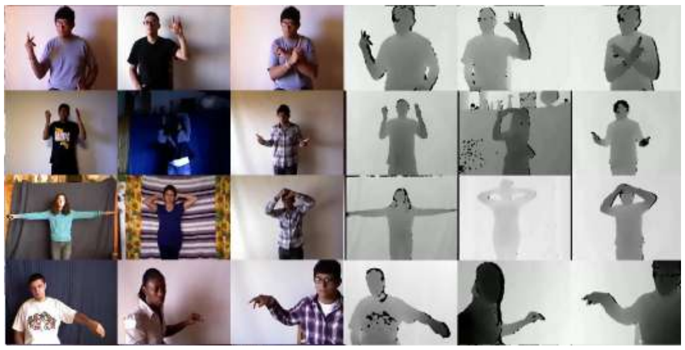

Figure 3.1: RGB videos and their corresponding depth videos. [1]

The dataset has 47933 RGB videos and corresponding to each of them, there is also a depth video recorded using a Kinect$^{TM}$ camera as shown in Figure 3.1. Each video has a frame rate of 10 frames per second and frame size of 240 x 320 pixels. There are a total of 249 distinct gestures and performed

| Sets | No. of Labels | No. of RGB videos | No. of performers |
|---|---|---|---|
| Training | 249 | 35878 | 17 |
| Validation | 249 | 5784 | 2 |
| Testing | 249 | 6271 | 2 |

Table 3.1: The subsets of the dataset

by 21 different individuals. The dataset is divided into three subsets as shown in Table 3.1 for the convenience of using, and are mutually exclusive, meaning, a performer appearing in any one of the subsets will not be seen in the other two subsets. This way, the training could be made more general and not be biased by the performers. The baseline method for the dataset is BoVW+MFSK, which stands for bag of Visual Words and Multi Frequency Shift Keying respectively and it has achieved an accuracy of 24.19%. The confusion matrix for the same is shown in Figure 3.2.

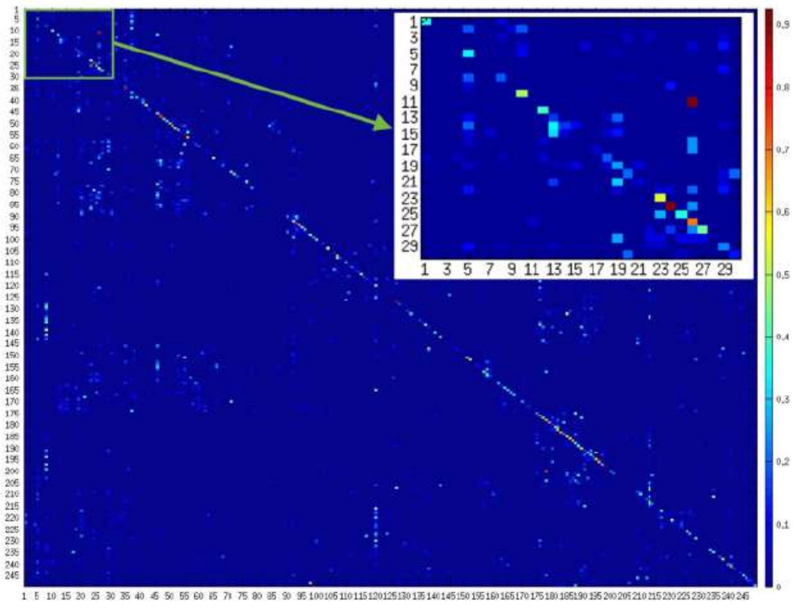

Figure 3.2: Confusion matrix of the baseline method. [1]

The dataset however has many limitations, which makes the classification hard.They are,

- The low frame rate makes it impossible to use optical flow and hence

| Team name | Validation accuracy | Testing accuracy |
|---|---|---|
| ASU | 64.4 | 67.71 |
| SYSU_IEEE | 59.7 | 67.02 |
| Lostoy | 62.02 | 65.97 |
| AMRL | 60.81 | 65.59 |
| XDETVP | 58.0 | 60.47 |

Table 3.2: The top five results.

making tracking of hand very difficult, also the low resolution of the videos make features extraction cumbersome.

- The training set is very small, that is there are only an average of 128 samples per class. This makes the clustering very difficult since there are more chances of intersections between classes, making the inter-class variance low.
- The intra-class variance is very high as well, meaning the way in which individuals have done the same gesture is very different from each other.

It is these limitations that make the dataset more challenging and hence more demanding, which is a good thing as it pushes the limits of whoever attends to make a better model. Table 3.2 shows the top five results at the ICPR 2017 Challenge.

# Chapter 4

# Methodology

"Whether you think you can, or that you can't, you are usually right."

*Henry Ford*

## 4.1 Previous works

The roughness of the dataset has challenged video processing techniques worldwide and is still being used in many challenges including the annual CVPR workshops. The state of the art is 67.7%, and the method that was used is so nad so. Though this includes the usage of both RGB and depth videos, this thesis focuses on only RGB videos and hence the accuracy achieved cannot be compared with the state of the art.

3D-CNN could achieve an accuracy of only 4% when it was applied directly on the raw dataset, which suggested that there should be some preprocessing required to "clean" the background. Various methods were tried to make this happen, like segmentation of skin and histogram-oriented-features(HOG) but the one which proved the most effective was heatmaps. The lab in which the model was trained have previously achieved an accuracy of 27% by using heatmaps. The way this was done was as follows, The frames in the video are read as matrices. For all the frames except the first frame, the element wise difference from the first frame is taken, then the matrices corresponding to those frames are normalized and then plotted using a specific colourmap.

This however had an issue as given in Figure 4.1. because as the difference was always taken from the first frame, the initial position of the hand left a trace whenever the hand moved from there and hence it had a "second hand" whenever a hand moved.

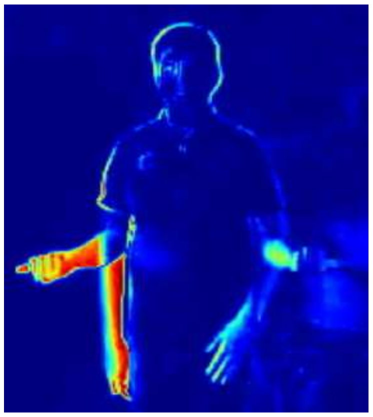

Figure 4.1: The issue with the heat map.

## 4.2 Pre-processing

The heatmap method seemed so good that it could not be abandoned altogether to enter a new method. The issue that was arising was mainly because the location of the hand was unknown, but if there is way to know it, then we are through. So we make use of *Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields*[3], which won the 2016 MSCOCO Keypoints Challenge, 2016 ECCV Best Demo Award, and 2017 CVPR Oral paper. Figure 4.2 shows how good the algorithm is in detecting the limbs.

Now that we know where the limbs are exactly, we can form a mask over each frame on the heatmaps such that only the real time position of the hands are visible. For doing this, we make small rectangles along the arms and a relatively bigger square on the fist to get the maximum detail from the fingers. Using these rectangles, a mask is made as shown in Figure 4.3 and placed on top of the frames used in heatmaps so that the "second hand" is

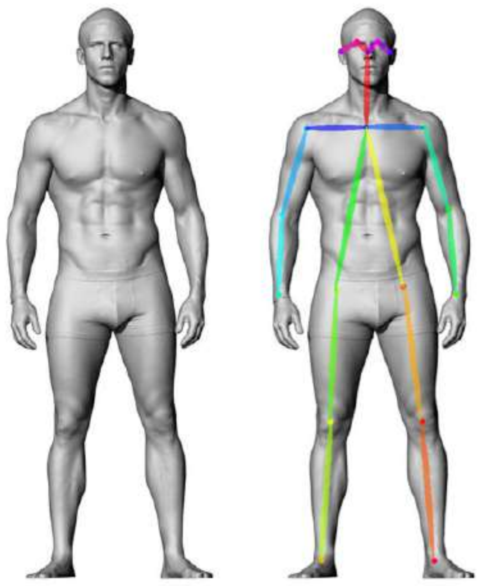

Figure 4.2: Pose estimation of a sample image.

blacked out including all the other background noises that was prevalent in the previous method.

The program for finding the pose however was not able to perform well on all the frames due to lack of good resolution images, and hence the rule was given as whenever the pose of both the arms was present, do the masking, otherwise skip it but this proved inefficient as in most frames, only one arm was detected from shoulder to the wrist, that is all the three nodes. Therefore the rule was updated as to mask the frame whenever there is at least one arm fully detected. Figure 4.4 shows how well the masking works when both the arms are detected, here the top left is the original frame, top right is the heatmap, bottom left uses first strategy and bottom right uses second strategy.

Figure 4.5 shows how the first masking strategy fails when there is only one arm detected, the conventions are the same as in Figure 4.4.

This masking was done over the entire dataset using the PyTorch module in Python on a system with Core-i7 processor and an Nvidia GeForce 1060 graphics card, consuming 6GB video memory over 28 days in a stretch.

There is still one issue that is not taken care of, that is, the low samples per class. To solve this to an extend, we extend the dataset by flipping the

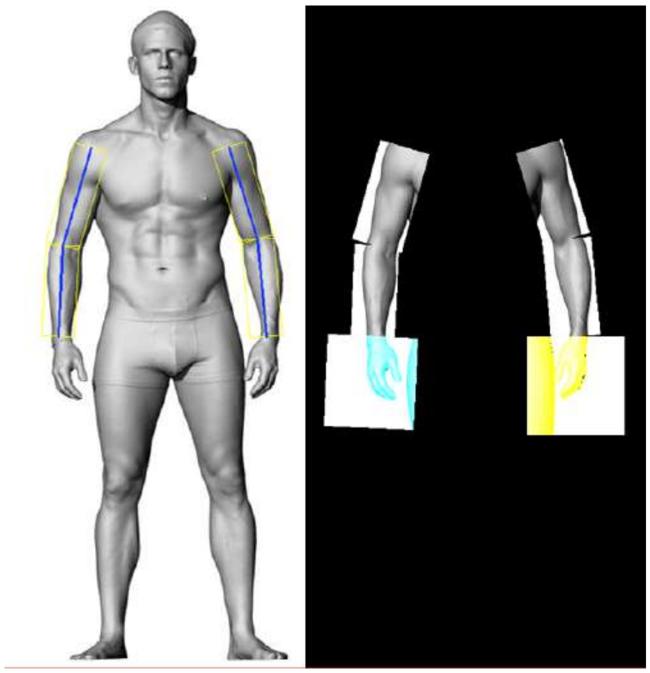

Figure 4.3: Masking out the background.

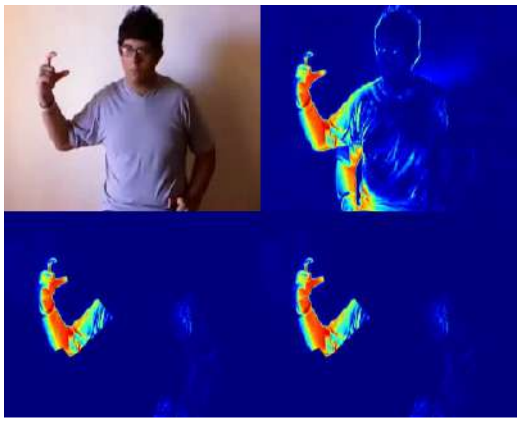

Figure 4.4: The first masking strategy.

frames on each video horizontally as shown in Figure 4.6 and hence making a mirror video for each of the video in the dataset. Now, there are twice the number of videos from before.

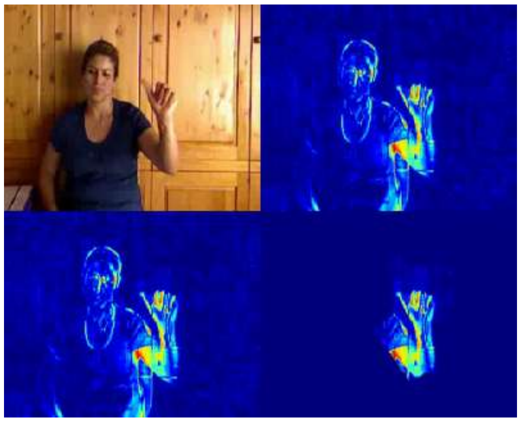

Figure 4.5: How the second strategy betters the first.

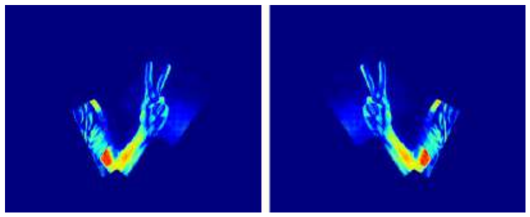

Figure 4.6: Flipping the videos horizontally.

## 4.3 Training the processed dataset

The processed dataset was trained using the 3D-CNN package from the Keras library in Python on a system with Core-i7 processor and an Nvidia GeForce 1050-Ti graphics card consuming 3GB video memory over 3 days. Not all videos have the same duration, and hence they should be made to have the same duration for the network to read them as files of the same shape. 80 is found to be the average number of frames in all the videos in the dataset, therefore those videos that does not have 80 frames are added with blank frames at either ends till they reach 80 and for those videos which have more than 80 frames, continuous 80 frames are chosen at random. A batch of 200 videos is taken as input and 4 videos from them is chosen. Each frame is re-sized to $112 \times 112$ pixels. The program has the following architecture,

- The first convolution layer with 16 kernels of shape (3,3,3) taking strides of 1 in each direction, which takes inputs of shape (x, 3, 8, 112, 112), where x is flexible. The input then undergoes max pooling with pool size (2,2) taking strides of 2 on each frame, that is, keeping, the depth unchanged.

- The second convolution layer also has 16 kernels of shape (3,3,3) having stride 1, but the second max pooling layer has pool size (2,2,2) taking strides of 2 in all the three directions.
- The third convolution layer has 32 kernels of shape (3,3,3) taking strides of 1 and the max pooling is the same as second layer.
- The last convolution layer has 64 kernels with same shape and strides as the first layer, and the max pooling has shape (2,2,2) with stride 2.
- A zero-padding on all sides of each frame and another max pooling with a pool shape (1,2,2), with stride 2. The "flatteing" of the input happens after that, giving it a shape of (332681,1,1).
- An MLP architecture now follows with a output coming through a softmax layer.
- This output is used to compute the cost function, which is then back-propogated and the weights in the kernel updated.

# Chapter 5

# Results and analysis

"I have not failed. I have just found 10,000 ways that won't work."

*Thomas Alva Edison*

The model however with all the background removal and hand extraction could only achieve an accuracy of 20%, before the horizontal flipping. However, even after the flipping was done, the accuracy only increased by 2%, this poor performance could be because of the inability of masking the relevant hand even when it was fully present on the frame.

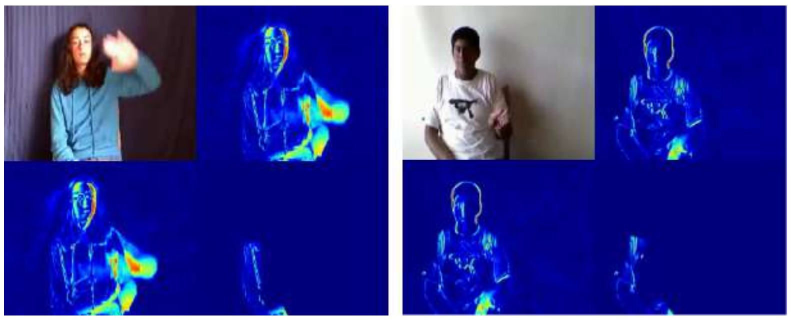

Figure 5.1: Error caused by motion blur.

Look at Figure 5.1, the fast movement of the hand used for gesture made a motion blur, which is due to the poor frame rate. A normal "smooth" video has at least 30 frames per second where the videos in the dataset, as mentioned earlier has only 10.

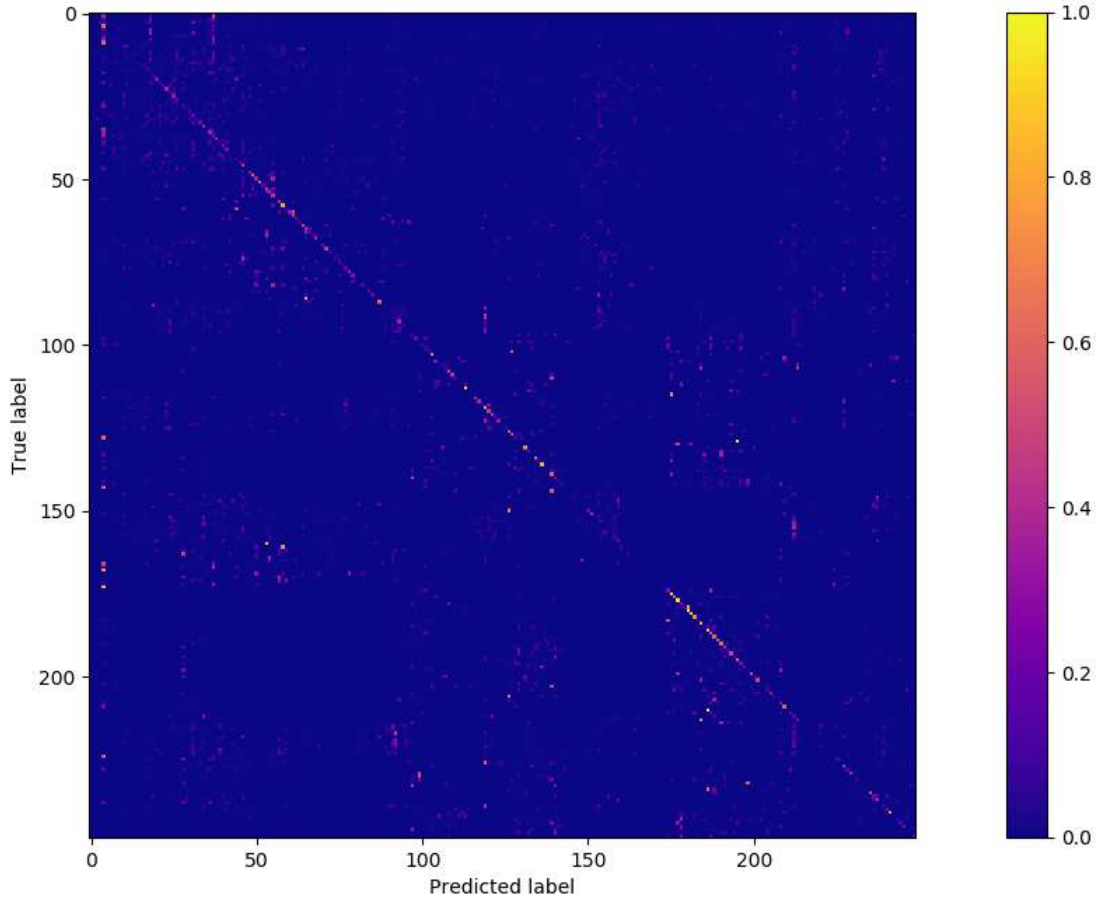

Figure 5.2: Confusion matrix for the used method.

The confusion matrix, Figure 5.2 shows the that many videos were wrongly labelled as the gesture with label 4. Once we take a look at the gesture labelled 4 itself as in Figure 5.3, it is clearly visible what the issue is.

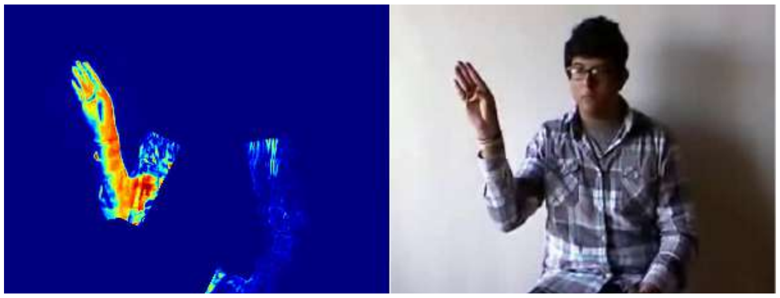

Figure 5.3: The gesture with label 4.

Interestingly, most gestures that got confused with the $4^{th}$ one, have similar pose, as shown in Figure 5.4 with varying finger positions.

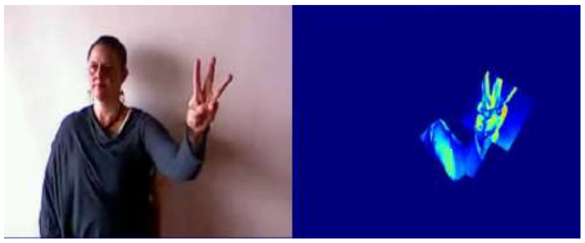

Figure 5.4: The gesture which got confused with 4.

This says that the feature extraction was not precise enough as to differentiate visually similar poses. This could have be made better by making the kernels rectangular, but at the cost of the training becoming slower.

# Chapter 6

# Conclusion and future prospects

”What doesn’t kill you, only makes you stronger.”

*Friedrich Nietzsche*

The dataset still remains one of the toughest challenges in video processing. The limitations are in a way also the perks of the dataset, which made it widely popular. Since the background and illumination in the dataset is not biased, it makes the training very general and hence aiding the devices irrelevant of their video capturing ability. Though the model developed in this thesis could isolate the background, it was not able to clean the noises made during the pose of the signers very well.

In future, the depth videos, which is also a part of the dataset could be taken into account to improve the accuracy of the existing model. Also a better convolution technique needs to be developed to extract the finer details of the dataset without overfitting.

# Bibliography

[1] Jun Wan, Stan Z. Li,Yibing Zhao, Shuai Zhou, Isabelle Guyon, Sergio Escalera. *ChaLearn Looking at People RGB-D Isolated and Continuous Datasets for Gesture Recognition* National Laboratory of Pattern Recognition, Institute of Automation, Chinese Academy of Sciences, China.

[2] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[3] Zhe Cao, Tomas Simon, Shih-En Wei and Yaser Sheikh *Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields*. CVPR 2017

[4] Christoph Berger. `https://appliedgo.net/perceptron/`. 2017

[5] Andrej Karpathy and Justin Johnson. *Convolutional Neural Networks for Visual Recognition*. Stanford CS class CS231n, 2017.

[6] Adit Deshpande.
`https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner\%27s-Guide-To-Understanding-Convolutional-Neural-Networks/`
UCLA, 2016

[7] Michael Nelson. `http://neuralnetworksanddeeplearning.com/chap1.html` August, 2017.

[8] Shuiwang Ji, Wei Xu, Ming Yang, Kai Yu. *3D Convolutional Neural Networks for Human Action Recognition*. IEEE Transactions on Pattern Analysis & Machine Intelligence, vol. 35, no. , pp. 221-231, Jan. 2013, doi:10.1109/TPAMI.2012.59

# Appendix

## Code listing for the model's architecture.

```python
import numpy as np
np.random.seed(100)
import keras

def model_main(rgb):
  import keras.backend as K
  from keras.models import Sequential
  from keras.layers.core import Dense, Activation
  from keras.layers import Convolution3D, MaxPooling3D,
    TimeDistributed, Flatten, ZeroPadding3D, Dropout, Lambda
  nb_classes = 249
  sum_dim1 = Lambda(lambda xin: K.sum(xin, axis = 1),
    output_shape=(1024,))
  model = Sequential()
  # 1st layer group
  model.add(TimeDistributed(Convolution3D(16, 3, 3, 3,
    activation='relu',
                            border_mode='same', name='conv1',
                            subsample=(1, 1, 1)),
                            input_shape=(None,3,8, 112, 112)))
  model.add(TimeDistributed(MaxPooling3D(pool_size=(1, 2, 2),
    strides=(1, 2, 2),
                          border_mode='valid', name='pool1')))
  # 2nd layer group
  model.add(TimeDistributed(Convolution3D(16, 3, 3, 3,
    activation='relu',
                            border_mode='same', name='conv2',
                            subsample=(1, 1, 1))))

  model.add(TimeDistributed(MaxPooling3D(pool_size=(2, 2, 2),
    strides=(2, 2, 2),
                              border_mode='valid', name='pool2')))
  # 3rd layer group
  model.add(TimeDistributed(Convolution3D(32, 3, 3, 3,
    activation='relu',
                               border_mode='same', name='conv3b',
                               subsample=(1, 1, 1))))

  model.add(TimeDistributed(MaxPooling3D(pool_size=(2, 2, 2),
    strides=(2, 2, 2),
                              border_mode='valid', name='pool3')))
  # 4th layer group
  model.add(TimeDistributed(Convolution3D(64, 3, 3, 3,
    activation='relu',
                               border_mode='same', name='conv4b',
                               subsample=(1, 1, 1))))

  model.add(TimeDistributed(MaxPooling3D(pool_size=(2, 2, 2),
    strides=(2, 2, 2),
                              border_mode='valid', name='pool4')))

  model.add(TimeDistributed(ZeroPadding3D(padding=(0, 1, 1),
    name='zeropadding')))
  model.add(TimeDistributed(MaxPooling3D(pool_size=(1, 2, 2),
    strides=(2, 2, 2),
                              border_mode='valid', name='pool5')))
  model.add(TimeDistributed(Flatten(name='flatten')))
  # FC layers group
  model.add(sum_dim1)
  model.add(Dropout(0.5))
  model.add(Dense(nb_classes))
  model.add(Activation('softmax'))
  return model

def weight_sharing(rgb):
  from keras.optimizers import SGD
  model = model_main(rgb)
  sgd = SGD(lr=0.0025, momentum=0.0, decay=0.0, nesterov=False)
  model.compile(loss='sparse_categorical_crossentropy',
                optimizer=sgd,
                metrics=['accuracy'])
  return model
```