# Machine Learning Nanodegree Capstone Project - American Sign Language Letters Recognition

Trong Canh NGUYEN

June 15, 2017

**Abstract**

In this article we investigate the recognition of ASL Letters (a-z) from an image dataset using Convolutional Neural Network.

## 1 Definition

### 1.1 Project overview

American Sign Language (ASL) recognition has long been subject to research in computer science, in particular computer vision. ASL uses mainly hand shapes, motions and sometimes facial expressions to express words. Recognizing ASL is challenging since the hand shape for each word varies from person to person and depends on the looking angle as well. While handwritten rules are very difficult to cover all the cases, machine learning, in particular deep neural network, can be more efficient since it learns directly from the data.

Another advantage of using deep neural network is that its performance can be improved with more data (which can be costly sometime but not impossible). Moreover, this model provides **probabilities** for different labels which we can use in conjunction with an Hidden Markov Model and a language model for phrase decoding in an ASL translation application for example.

Recently an architecture of deep neural network called convolutional neural network (CNN) has proven to be very efficient in recognizing objects in images. In this project we will use this network to solve our problem of recognizing ASL letters from images.

### 1.2 Problem statement

In this project we will classify only a small subset of ASL which is the alphabet letters (a-z). This is a good starting point to investigate the usability of deep learning for this ASL recognition problem and future extensions could be recognition of more general words.

In ASL, each letter corresponds to a specific hand shape as in figure 1. Most of them are static, except for J and Z which requires motion:

- J is I with a twist of the wrist, so that the little finger traces the curve of the printed form of the letter
- Z is an index finger moved back and forth, so that the finger traces the zig-zag shape of the letter Z

Since we use only static images, letters J and Z are excluded from this study.

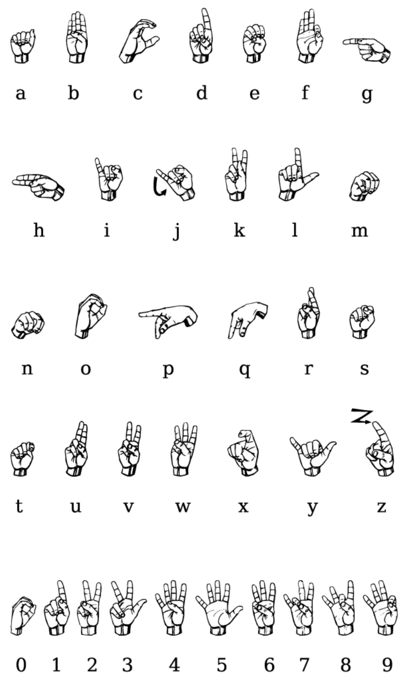


Figure 1: American Sign Language Letters. Image taken from Wikipedia: https:en.wikipedia.org/wiki/American_manual_alphabet

From images captured from a standard camera where each image contains a hand shape for a letter, our task is to recognize the letter from the image. This problem is similar to the digit recognition problem using the MNIST dataset for example. We expect that the deep learning method can be applied efficiently with this dataset containing hand shapes as well. Note that the images do not contain only the hand but some other things in the background, like part of the person's face for example, and that's why we use Convolutional Neural Network since this model works well on this kind of problem.

### 1.3 Metric

There are two evaluation metrics we can consider:

1. **Overall accuracy**:

$$accuracy = \frac{\sum_{l=a}^{z} correct(l)}{\sum_{l=a}^{z} total(l)}$$

   where $total(l)$ is the number of letter $l$ in the test set and $correct(l)$ is the number of times this letter is recognized correctly. This metric gives an overall performance of the model as a single number, and thus can be compared with other results.

2. **Confusion matrix**: The confusion matrix details the predictions on all the letters on the test set, from which we can compute the precision and recall for each letter. Whether precision or recall is more important depends on the application. What is important from this matrix is that we can identify the letters that are often confused with each other, which may help identify problems and ideas to improve the model.

## 2 Analysis

### 2.1 Data exploration & visualization

The dataset used in this project contains more than 65000 static color images for 24 different letters (j and z being excluded), taken from 5 different image capturing sessions under different background and lightening conditions, and is publicly available under this link: http://empslocal.ex.ac.uk/people/staff/np331/index.php?section=FingerSpellingDataset. This is the dataset used in the original article [1] where the authors used a different technique for recognition and from which we will compare our results.

Figure 2 and 3 show some examples from this dataset. We can see that the images are taken in quite different backgrounds. The image sizes are different as well, the figure 7 shows the distribution of width/height ratios in our dataset.

### 2.2 Algorithms and Techniques

We will use convolutional neural network (CNN) architecture, which normally uses 3 type of layers:

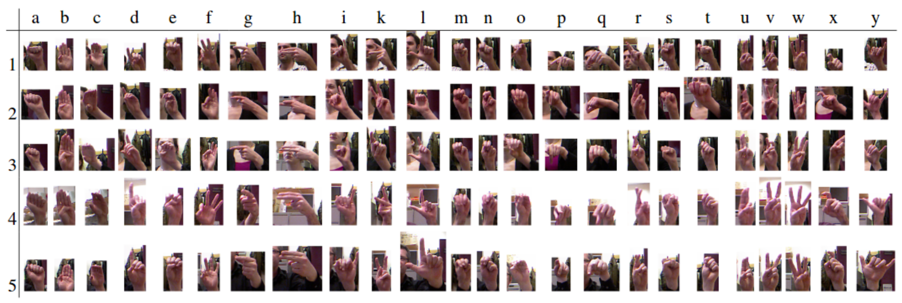


Figure 2: Example images from the dataset. Image taken from the article [1]

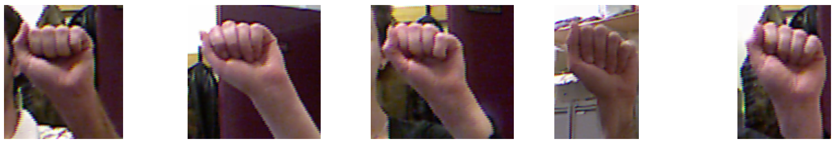


Figure 3: Example images of the same letter **a** from the 5 different sessions:

- **Convolutional Layer**
- **Pooling Layer**
- **Fully Connected Layer**.

Here is an example of a CNN with two convolutional layers:

**INPUT - [CONV-RELU-POOL] - [CONV-RELU-POOL] - FC - OUTPUT**

### 2.2.1 Convolutional Layer

A convolutional layer is produced by a list of filters. Each filter looks at a small portion of the input and scans the input. Here's how it work:

- Suppose that $X$ is an image input of shape $(W, H, 3)$ where 3 stands for the three color channels.
- A filter $f$ of size $F$ will have shape $(F, F, 3)$. Again the number 3 is equal to the number of the input channels. In practice we prefer small filter size like $F = 3, 5, 7$ for example.
- For every region of size $F \times F \times 3$ in the input we compute the cross product between the filter and that region to produce a number. The filter will scan all those possible regions in the input to produce the convolutional layer:

$$c[m,n] = \sum_{i=0}^{F-1}\sum_{j=0}^{F-1}\sum_{c=1}^{3} X[m+i, n+j, c] * f[i,j,c]) \tag{1}$$

where $(m, n)$ are indices of the top left corner of each region in X.

- Normally we apply an activation function like relu to the convolutional layer to obtain an activation map $a[m,n] = relu(c[m,n])$. From now when we speak of an convolutional layer (particularly in the illustration figures below) we suppose that an activation function is always applied after a convolutional layer.
- Normally $m \leq W - F$ and $n \leq H - F$ so the size of $c$ is $(W_c, H_c) = (W - F + 1, H - F + 1)$. However in practice we apply some padding (all zeros) of size $(F-1)/2$ to the top, bottom, left, right to the original input $X$, such that $X$ will have size $(W + F - 1, H + F - 1)$ and the activation will have the same size $(W, H)$ as the original input.
- The equation 2 above can be understood that the filter scan the input with step of 1, or each time the filter moves 1 step by increasing $m$ or $n$. However we can make the filter to moves by a larger step $S = 2$ for example. We call $S$ **stride**:

$$c[m,n] = \sum_{i=0}^{F-1} \sum_{j=0}^{F-1} \sum_{c=1}^{3} X[m*S+i, n*S+j, c] * f[i,j,c] \tag{2}$$

  Of course the larger the stride $S$, the smaller the size of the convolutional layer. Suppose that the applied padding is $P$, here is the formula for the width and height of the activation map:

$$W_c = \frac{W - F + 2P}{S} + 1 \tag{3}$$

$$H_c = \frac{H - F + 2P}{S} + 1 \tag{4}$$

- A convolution layer usually contains many filters which have the same size $F$. The number of filters, that we note $D$, is called the **depth** of this convolutional layer.
- So finally from an input of shape $(W, H, 3)$, using $D$ filters we get a convolutional layer of shape $(W_c, H_c, D)$. The number $D$ now plays the same role as the number of color channels 3 before. Figure 4 show an illustration of a convolutional layer, whose size is double the size of the input because of bigger depth. It is standard to see in a CNN that the subsequent convolutional layers have bigger and bigger depths.

#### 2.2.2 Pool layer

It is common to insert a pooling layer after a convolutional layer. Its function is to reduce the spatial size of the output to reduce the amount of parameters. A frequently used type of pooling layer is the MAX pooling, which operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. For example a MAX pooling layer of size $2 \times 2$ and stride 2 with take every non-overlapped $2 \times 2$ squares in the input (which is the result of the previous convolutional layer after activation) and only keep the max of the 4 numbers. This will reduce the width and height of the input by a factor of 2.

The figure 5 below, taken from `http://cs231n.github.io/convolutional-networks/#conv` (Stanford CS231n:Convolutional Neural Networks for Visual Recognition), illustrates clearly what a MAX pooling does. The figure 6 illustrates the combination of a convolutional layer and a pooling layer.

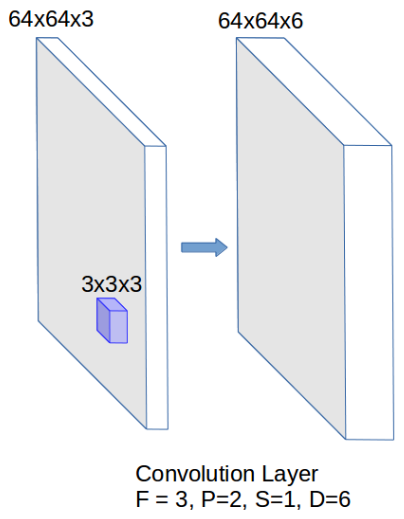


Figure 4: Convolutional layer with size F = 3, stride S = 1, padding P = 2 and depth D= 6.

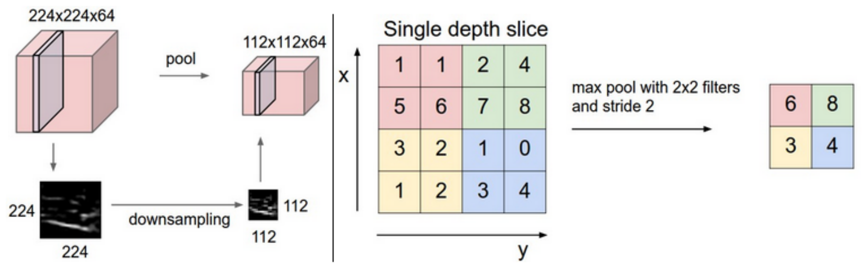


Figure 5: MAX Pooling layer with size 2 and stride 2.

### 2.2.3 Fully Connected layer

Finally before the output layer we apply one or more fully connected layers as in normal multilayer neural network. Each neuron in this layer connects to every points of the previous layer. Suppose that the previous layer output has shape $(8, 8, 48)$ after some covolutional and pooling layers, and that the fully connected layer has 100 neurons, this fully connected layer will have $8 * 8 * 48 * 100 \approx 300000$ weights. Normally most of the weights are in this fully connected layer (convolutional layers have much less weights and pooling layers have no weight). This big number of weights can contribute to over-fitting. One technique to prevent over-fitting is **Dropout** ([2]). The idea is that during training we randomly turn off a proportion of the neurons. By this we force the network to learn multiple independent representations of the same data and this can help the network to generalize better. Dropout is supported in TensorFlow

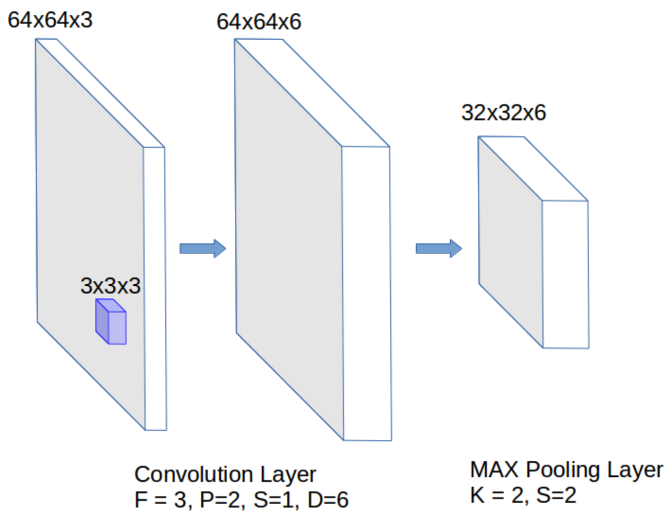


Figure 6: Convolutional layer and pooling layer

so we will just use it in our implementation later.

## 3 Methodology

### 3.1 Data Preprocessing

The images in our dataset have different image size, with average width of 103 and height of 143. Figure 7 show the distribution of the width/height ratios. As standard, we use square images as input in a convolutional neural network, so we resize the images to be of equal height and width (instead of making random crops that may cut critical parts of the image). Since the average ratio width/height is 0.7, the resizing will not distort the original too much, as shown in the figure 8.

We also rescale the image to 64x64, by this we loose some information but the images are still very well distinguished and we can reduce the number of features of the problem which make learning faster. Moreover we want to be able to load all the training images in RAM (8Gb) so the training can be done in reasonable amount of time since loading images from disk takes a lot of time and is the bottleneck of the performance.

Initially we only rescaled the images to 120x120 but for each batch we had to load the images from disk, and the training time become much longer. Other advanced solutions consist of loading and training in parallel but this is not what we want to go into in the scope of this small project.

We will keep all the 3 color channels, without converting to black and white for

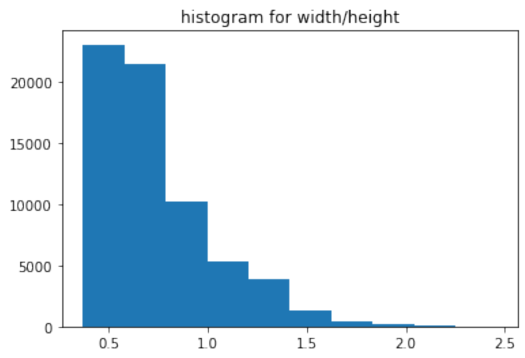


Figure 7: Image width/height ratios distribution. Avarage ratio is 0.7.

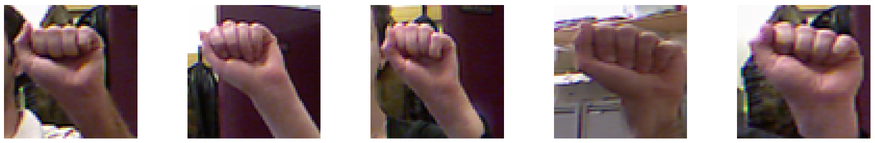


Figure 8: The same 5 images as in figure 3 but resized to (64x64).

example. First we want to learn directly from the original data. Second we think that the color may help in recognizing the hand part in the images. Other researches as well as the CNN architectures being used in the ImageNet competition all keep the 3 color channels so we just follow. Moreover in the future we might want to use transfer learning by using a pre-trained CNN which normally operates on all the 3 color channels.

## 3.2 Implementation

### 3.2.1 Train, Validation and Test set split

We use around 10% of the images for hold on test set, 10% for validation set (to check performance during training and to adjust hyper-parameters). The remaining (more than 50K images) are used for training. There are **2 ways** that we can define the validation and test set:

1. The validation and test set contains all the random images from all the 5 sessions.
2. The validation and test set are chosen from one session and the other 4 sessions are used for training

With (1) we had much better score than (2) and this is reasonable since with (1), the images in the training set are representative for the test set (with similar background conditions). However (2) shows the generalization strength of the prediction model, or how the model can recognize the hand posture under new environments. One way improve the model is to gather images from as many different backgrounds or environments as possible.

*In this report we show the result only with the second split strategy.*

#### 3.2.2 Convolutional Neural Network

We use TensorFlow to build a Convolutional Neural Network with 3 conv layers followed by max pooling, and finally a fully connected hidden layer with 100 neurons. The output layer has 24 neurons corresponding to 24 letters. We use softmax cross entropy loss.

**conv 3x3x6 – max pooling – conv 3x3x12 – max pooling – conv 3x3x24– max pooling – fc 100 – dropout – output 24**

Dropout is used after the fully connected layer where we have most of the weights.

We used Adam Optimization as implemented in TensorFlow for loss optimization. For each epoch we run the optimization step on every batches of 128 training images (thus each epoch has around 400 batches). Before each epoch we shuffle the training images so that the batches are different from epoch to epoch.

### 3.3 Refinement

Besides the number of convolutional layers, their sizes and depths, the other hyper parameters we need to optimize are the learning rate, optimization algorithm, and the dropout probability.

Initially we tried a standard multilayers neural network without convolution as a starting point. The idea is to make the whole workflow working (from data processing, model construction, training...). With this model we got an overall accuracy on test set of 55% which was much lower than with the convolutional architechture as we will report later. So the convolutional neural network did indeed perform better and improve the results as expected.

The main technical point we encountered is the training time, since initially we loaded each batch from the disk since we did not have enough RAM to hold all the training images. This made learning rate tuning and network architecture optimizing more difficult. We solved this problem by:

- Adding more RAM
- Reducing the image size to 64x64 (from 120x120)

We did some research on how people solve this problem in practice (with more data), however this must need more advanced scheme of loading and training which we did not want to go further in.

Once the training works much faster, we tuned the learning rate by factor of 10 as well as the learning rate decay until we observed a good converging of the loss function.

Adam optimization provided much faster convergence than the standard Stochastical Gradient Descent, and worked quite stable, so we chose it.

Finally we tried to put more depths and convolutional layers, however this did not really improve the results with the architecture described above, so we decided to just stick with that architecture.

## 4 Results

The figures (9) and (10) are the loss function and train/validation accuracies. While we have good optimization on the loss function and obtain training accuracy nearly 100%, the predictive power of the model is much lower (around 78%) on the validation set.

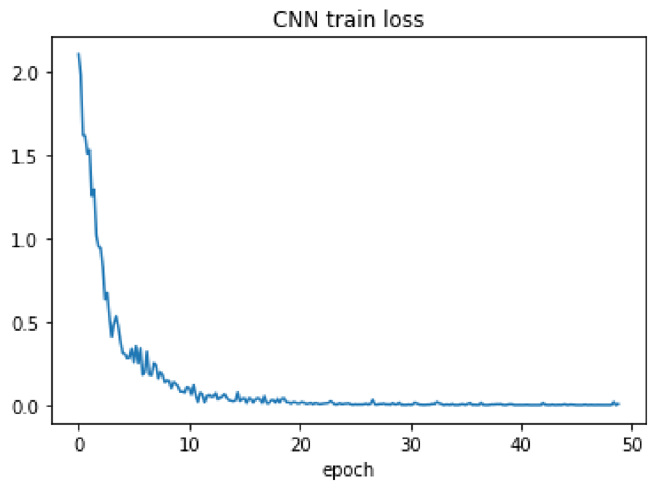


Figure 9: Loss is computed on batch at a regular interval (every 100 batches)

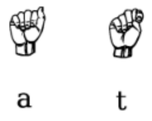
### 4.1 Accuracy

The overall accuracy on hold out test set is 78% which is consistent with the validation accuracy curve.

### 4.2 Confusion matrix

The figure (11) shows the confusion matrix with some big confusions marked in orange. Let's analyze some of them:

- **A** and **T** confusion: There are 59 T mistaken as A and 37 A mistaken as T. This can be explained by the similarity between the hand shapes corresponding to these two letters:



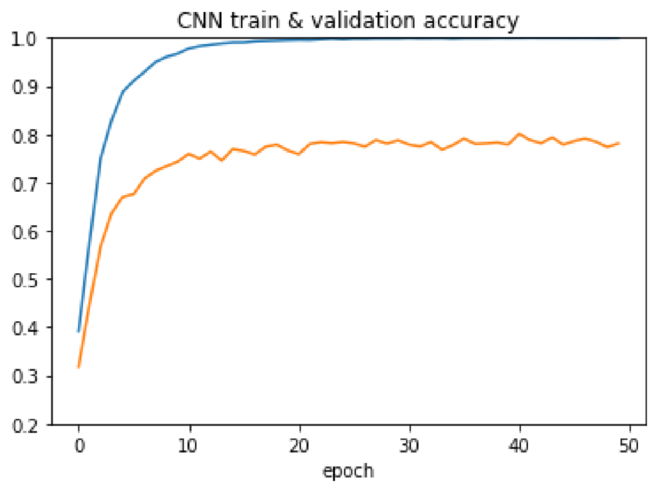
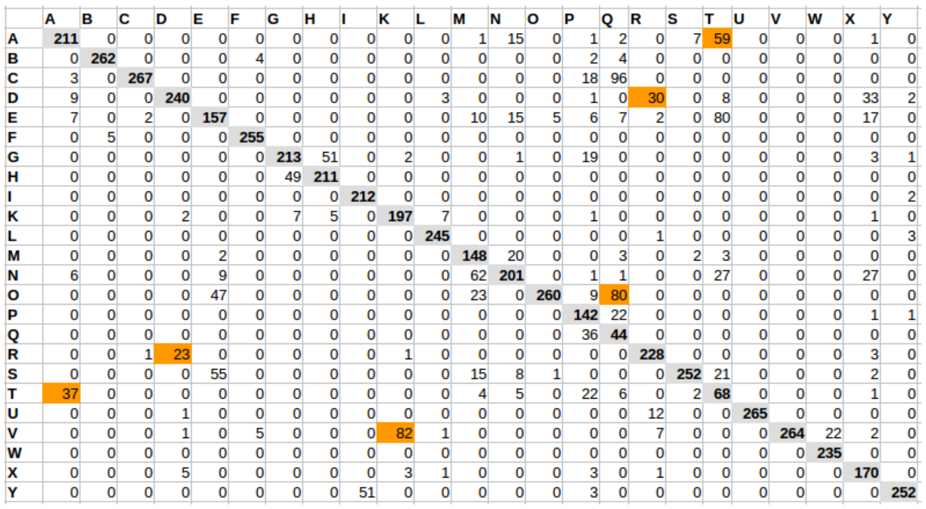


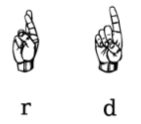
Figure 10: Train/validation accuracy is computed after each epoch. To save computing time, we compute the train accuracy only on a subset of 10000 (random) training images rather than full training data.)



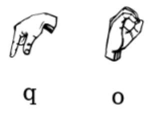
| | A | B | C | D | E | F | G | H | I | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 211 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 15 | 0 | 1 | 2 | 0 | 7 | 59 | 0 | 0 | 0 | 1 | 0 |
| B | 0 | 262 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C | 3 | 0 | 267 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 18 | 96 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D | 9 | 0 | 0 | 240 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 1 | 0 | 30 | 0 | 8 | 0 | 0 | 0 | 33 | 2 |
| E | 7 | 0 | 2 | 0 | 157 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 15 | 5 | 6 | 7 | 2 | 0 | 80 | 0 | 0 | 0 | 17 | 0 |
| F | 0 | 5 | 0 | 0 | 0 | 255 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | 0 | 0 | 0 | 0 | 0 | 213 | 51 | 0 | 2 | 0 | 0 | 1 | 0 | 19 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 1 |
| H | 0 | 0 | 0 | 0 | 0 | 0 | 49 | 211 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 212 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| K | 0 | 0 | 0 | 2 | 0 | 0 | 7 | 5 | 0 | 197 | 7 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| L | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 245 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |
| M | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 148 | 20 | 0 | 0 | 3 | 0 | 2 | 3 | 0 | 0 | 0 | 0 | 0 |
| N | 6 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 62 | 201 | 0 | 1 | 1 | 0 | 0 | 27 | 0 | 0 | 0 | 27 | 0 |
| O | 0 | 0 | 0 | 0 | 47 | 0 | 0 | 0 | 0 | 0 | 0 | 23 | 0 | 260 | 9 | 80 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 142 | 22 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| Q | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 36 | 44 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R | 0 | 0 | 1 | 23 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 228 | 0 | 0 | 0 | 0 | 0 | 3 | 0 |
| S | 0 | 0 | 0 | 0 | 55 | 0 | 0 | 0 | 0 | 0 | 0 | 15 | 8 | 1 | 0 | 0 | 0 | 252 | 21 | 0 | 0 | 0 | 2 | 0 |
| T | 37 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 5 | 0 | 22 | 6 | 0 | 2 | 68 | 0 | 0 | 0 | 1 | 0 |
| U | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 265 | 0 | 0 | 0 | 0 |
| V | 0 | 0 | 0 | 1 | 0 | 5 | 0 | 0 | 0 | 82 | 1 | 0 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 264 | 22 | 2 | 0 |
| W | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 235 | 0 | 0 |
| X | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 3 | 1 | 0 | 0 | 0 | 3 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 170 | 0 |
| Y | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 51 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 252 |

Figure 11: Confusion matrix with some confusions marked in orange.

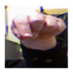
- **R** and **D**: there are 30 R are mistaken as D, and 23 D mistaken as R. This again can be explained by the similarity between the hand shapes corresponding to these two letters:

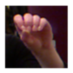


- **Q** and **O**: there are 80 Q are mistaken as O and this is very strange since the two signs are not similar:



  We decided to get some examples of Q that were mistaken as O, and to look at the O images in the training data, then we saw that for some image capturing angles this may create confusion as in the picture below:


Q mistaken as O


O from train data

  We think that with more data this confusion would become less important.

Compare with the result in the article [1], we have quite different confusion matrix (probably due to different techniques) but the overall precision is better(78% compare to 73% in the article). This is promising since our model learns directly from raw images without applying any advance computer vision techniques, like hand extraction in [1]

## 5 Conclusions

In this project, using machine learning technique with convolutional neural network, we could recognize the ASL letters from images taken from a standard camera with an overall accuracy more than 75%. Some letters can be recognized with high accuracy but some others not due to the similarities of the hand shapes corresponding to them.

Convolutional neural network normally takes time and computer resource for training, and in this project we overcame this problem by preprocessing our data to fit our computing resource. In the future if we have more data then we should need more advance training scheme as well as more resource like GPU for example.

The results obtained are not good enough to be used in production but there are still many opportunities for improvements:

- Collecting more data and more importantly with images taken under different backgrounds or environments. This will help the model to generalize better and recognize the important part in the image which is the hand shape.
- Data augmentation by changing the background of the images or making some crops. We have not done this in this project lack of time, since changing the background or cropping (without taking out the hand position) will take time.

The model used in this project can be extended easily to recognize more words in the vocabulary provided that we have enough data.

The most interesting aspect of the project is that this is the first time we build a deep neural network from start to end and apply it to a new dataset. This require many works from data exploration, data preprocessing, optimization, result interpretation, etc. By the end we had a much better understanding of deep learning and particularly convolutional neural network .

## References

[1] Nicolas Pugeault & Richard Bowden: *Spelling It Out: RealTime ASL Fingerspelling Recognition*

[2] Nitish Srivastava, Geoffrey Hinton et al: *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*