

# Machine Learning Engineer Nanodegree

## Capstone Project

---

Darryl Ma  
January 13, 2018

## I. Definition

---

### Project Overview

American Sign Language (ASL) is the predominant sign language used amongst the deaf communities in North America. It was created in the early 19<sup>th</sup> century at the American School for the Deaf in Hartford, Connecticut. For years, due to the stigmas and beliefs in the superiority of the oral language, sign language has been largely constrained to the deaf community with a few family members embracing the language in order to communicate with their loved ones. However, with the advent of machine learning and visual classification, this barrier can easily be overcome by introducing apps within our mobile devices to translate visual cues into spoken words.

My grandfather was deaf and as such communication between family members, who did not learn sign language, was limited. Because of this handicap, he was isolated from a lot of family conversations and discussions. Enabling him to communicate with the rest of the family would have drastically altered his relationship with them and ultimately enhanced his quality of life. Therefore, I don't see this endeavor as just a way to solve a communication problem rather it is more about re-integrating a typically estranged community of people (i.e. the deaf community) back into our society.

### Problem Statement

The ASL lexicon consists of over 50,000 signs and there are many sign language dialects (e.g. French sign language, Ecuadorian sign language, etc.) used around the world, however, for this project, we will be focusing strictly on ASL and the static handshapes used to communicate the English alphabet. Furthermore, because the letters "J" and "Z" require motion, these two particular alphabets have been removed from this classification project. My goal is to use machine learning, specifically

Convolutional Neural Networks (CNN), to achieve 80% accuracy in identifying 24 of the 26 English alphabets from a given dataset of sign language cues.

The training and testing datasets for this project were obtained from Kaggle.com (<https://www.kaggle.com/datamunge/sign-language-mnist>).

## Metrics

I used classification accuracy on the test dataset as the main evaluation metric to measure the efficacy of the various classification models/algorithms used. In addition to the accuracy metric, I also plotted a confusion matrix to see if there were any particular handshapes that tended to get misclassified more often than others. This helped in identifying handshapes which required more training samples. A sample confusion matrix can be seen below:

Predicted Vs True Label						
		Predicted Labels				
True Labels		0	1	2	3	4
	0	50	0	4	2	0
	1	1	40	0	0	3
	2	0	2	45	1	0
	3	1	0	0	60	0
	4	2	4	0	0	30

The reason for using accuracy instead of other metrics such as precision, recall, or F1 score, is because, in this classification problem, there is no true negative scenario, it is simply a case of whether the predicted label is correct or incorrect. Therefore, we are only concerned with the number of correct predictions over the total number of samples. The higher the percentage of correct predictions, the more effective a model is considered.

## II. Analysis

---

### Data Exploration

The training and testing datasets consist of 27,455 and 7,172 labelled 28x28 monochrome image of a handshape, respectively. The datasets were provided in .csv files where each image was flattened to a 784x1 array.

Below are samples of the 24 input handshapes and their corresponding labels. As you can see, the images are relatively pixelated and one can already imagine that some labels will likely be misclassified on account of their likeness. For example, labels 2 and 14, labels 0 and 4, and labels 12 and 13 are all combinations of hand gestures that closely resemble each other.

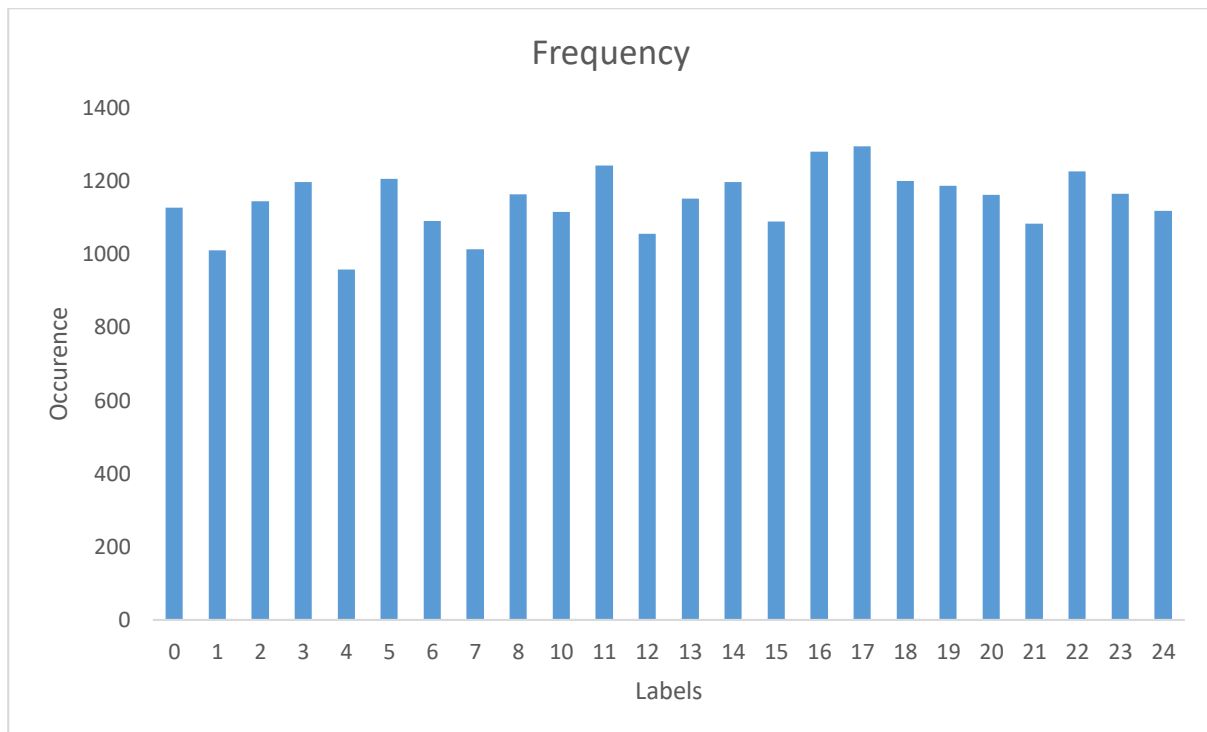


The table below is a mapping of the labels and their corresponding English alphabets. Do note that label 9 is missing as it corresponds to the alphabet, J, which, as highlighted before, is not included in this classification exercise as it is a hand gesture that requires motion:

Label	Alphabet		Label	Alphabet
0	A		13	N
1	B		14	O
2	C		15	P
3	D		16	Q
4	E		17	R
5	F		18	S
6	G		19	T
7	H		20	U
8	I		21	V
10	K		22	W
11	L		23	X
12	M		24	Y

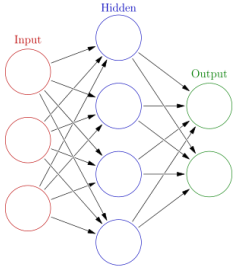
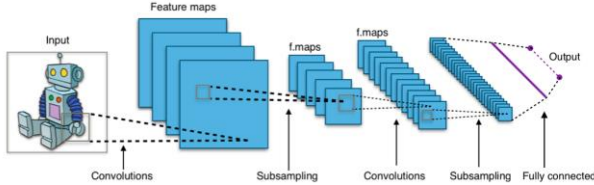
## Exploratory Visualization

The plot below shows the frequency of the different labels in the training dataset, which verifies that no particular handshake is biased, all handshakes are relatively equally represented:



## Algorithms and Techniques

A variety of algorithms were used to solve this classification problem, which have been detailed in the table below:

No	Algorithm	Description	Reasons
1	Single-Layer Perceptron	<p>A linear operation in which every input is connected to every output by a weight</p> 	This was the benchmark model against which the other algorithms were measured.
2	Multilayer CNN	<p>Analyzes images by dividing input into smaller segments and identifying specific objects/patterns within the image before combining results to obtain an output</p> 	This model consisted of 3 hidden layers. The first convolution layer helped to detect edges and colors, the second convolution layer helped to detect higher level features such as shapes and stripes, and the third convolution layer helped to detect more specific objects. With more layers to make the classification process more comprehensive, it was expected that this algorithm would perform better than the single layer perceptron.
3	AdaBoost	<p>Combines many rules of thumb/weak learners (Box 1, 2, and 3) to obtain a strong classifier (Box 4) whereby each iteration concentrates on what it did poorly in the previous iteration and doesn't concentrate much on what it got right previously</p>	These algorithms, which have real-world applications in computer vision, character recognition, and face recognition, were selected to verify if other supervised learning models could match the accuracy achieved by a CNN.

4	Support Vector Machines (SVM)	<p>Assigns new examples to one category or another by separating training samples into groups based on maximizing margins</p>	
5	Gaussian Naïve Bayes	<p>Classifies new examples by predicting membership probabilities for each class. The class with the highest probability is considered the most likely class.</p>	

## Benchmark

As there are 24 classes, the probability of guessing the correct label randomly is 1 out of 24 or 4.2% chance. Anything that achieves results above this percentage is an indication that there is more than just random chance at work.

For the benchmark model, I used a simple vanilla CNN, which consists of a flatten layer, one dropout layer to reduce overfitting and a fully connected layer with softmax activation to obtain probabilities for each handshake prediction. See below for vanilla model architecture:

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dropout_1 (Dropout)	(None, 784)	0
dense_1 (Dense)	(None, 25)	19625
Total params: 19,625.0		
Trainable params: 19,625.0		
Non-trainable params: 0.0		

The accuracy achieved by the benchmark model above was used as the standard to evaluate the effectiveness of the other supervised learning models.

As a secondary benchmark, I used Luis A. Estrada Jiménez and Marco E. Benalcázar's paper on "Gesture Recognition and Machine Learning Applied to Sign Language Translation", which they presented at the VII Latin American Congress of Biomedical Engineering conference in October 2016.

They claimed to achieve classification accuracy of 91.6% on 61 gestures from the Ecuadorian sign language (30 letters, 10 numbers, and 21 expressions). Their solution utilized k-nearest neighbors, decision trees, and the dynamic time warping algorithms to achieve their results and unlike this project, instead of image processing, they used flex, contact, and inertial sensors mounted on a polyester-nylon glove to build their input features.

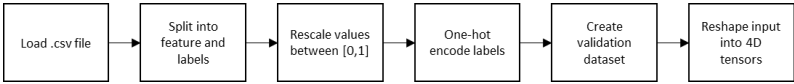

## III. Methodology

### Data Preprocessing

The input file used in this project consisted of a .csv file with samples of flattened image arrays and their corresponding labels. Refer to example below:

Label	Pixel 1	Pixel 2	Pixel 3	Pixel 4	Pixel 5	...	Pixel 784
0	125	200	0	56	8	...	9
5	10	55	88	156	147	...	12
24	53	59	78	44	255	...	111

Because each algorithm had different requirements in terms of input dimensions, I had to customize the data preprocessing procedures for each algorithm. Please find the summary below:

No	Algorithm	Data preprocessing flow
1	Single-Layer Perceptron & Multilayer CNN	 <b>Output:</b> 4D 1-channel tensors
2	AdaBoost/SVC/ GaussianNB Classifiers	 <b>Output:</b> Flattened image arrays

Below is a brief description of each preprocessing step:

No	Step	Description
1	Load .csv file	Read .csv input into panda dataframes
2	Split into features and labels	Use pandas to extract out features into one dataframe and labels into another dataframe



3	Rescale values between [0,1]	Normalized values in feature set to range between [0,1] by divided all pixel values by 255
4	One-hot encode labels	One-hot encoding creates a "dummy" variable for each possible label category
5	Create validation dataset	Reserved the first 3000 samples in the training dataset for validation
6	Reshape input into 4D tensors	For CNN, the input was converted into 4D tensors of the following dimension: (-1, 28, 28, 1)

The code snippet below is the code use to pre-process data for the single-layer perceptron and multilayer CNN:

```
# Rescales [0,255] --> [0,1]
x_train = train_features.astype('float32')/255
x_test = test_features.astype('float32')/255

# One-hot encode the labels
num_classes = len(unique_labels)+1
y_train = keras.utils.to_categorical(train_labels.values, num_classes)
y_test = keras.utils.to_categorical(test_labels.values, num_classes)

# Breaks training dataset into training and validation datasets
(x_train, x_valid) = x_train[3000:], x_train[:3000]
(y_train, y_valid) = y_train[3000:], y_train[:3000]

# Reshapes training, validation and testing features into 4D tensors
x_train = x_train.values.reshape(-1, 28, 28, 1)
x_valid = x_valid.values.reshape(-1, 28, 28, 1)
x_test = x_test.values.reshape(-1, 28, 28, 1)

print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_valid.shape[0], 'validation samples')
```

The datasets were extremely sanitized and no abnormalities were identified so I did not need to clean the datasets any further.

## Implementation

The final implementations of the 4 different algorithms used in the classification problem are described below:

1. **Single Layer Perceptron** – The keras module was used to create the single layer perceptron (see code snippet below). The first layer flattens the input 4D 1-channel tensor into a 1-D array, the dropout layer is used to reduce overfitting

and the last layer is a fully-connected layer to assign weights for each of the 784 features to arrive at a 25 value output vector.

```
model = Sequential()
model.add(Flatten(input_shape=x_train.shape[1:]))
model.add(Dropout(0.4))
model.add(Dense(25, activation='softmax'))
model.summary()
```

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dropout_1 (Dropout)	(None, 784)	0
dense_1 (Dense)	(None, 25)	19625
Total params: 19,625.0		
Trainable params: 19,625.0		
Non-trainable params: 0.0		

- Multilayer CNN** – The keras module was used to create the multilayer CNN (see code snippet below). Essentially this algorithm builds upon the single layer perceptron, adding 3 hidden layers prior to, what is in fact, the single layer perceptron described above. Each hidden layer consists of a 2d convolution that uses 'relu' activation followed by a pooling function to reduce dimensionality. By doing this, we increased the number of parameters from 19,625 to 24,841.

```
model = Sequential()
model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu',
input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Flatten())
model.add(Dropout(0.4))
model.add(Dense(25, activation='softmax'))
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 16)	80
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 16)	0
conv2d_2 (Conv2D)	(None, 14, 14, 32)	2080
max_pooling2d_2 (MaxPooling2D)	(None, 7, 7, 32)	0
conv2d_3 (Conv2D)	(None, 7, 7, 64)	8256
max_pooling2d_3 (MaxPooling2D)	(None, 3, 3, 64)	0
flatten_2 (Flatten)	(None, 576)	0

dropout_2 (Dropout)	(None, 576)	0
dense_2 (Dense)	(None, 25)	14425
=====		
Total params: 24,841.0		
Trainable params: 24,841.0		
Non-trainable params: 0.0		

The sklearn module and corresponding libraries were used to create the AdaBoost, SVC and Gaussian Naïve Bayes classifiers. Initially, I used default parameters so I could compare the effectiveness of each algorithm against one another. Only after determining that out of the three, SVC performed the best, I started to tweak the C and gamma parameters to obtain the optimized SVC classifier.

3. **AdaBoost** – As shown in the code snippet below, the AdaBoost implementation was minimal. All parameters were kept as default to standardize the comparison between AdaBoost, SVC and Gaussian Naïve Bayes.

```
# Rescales [0,255] --> [0,1]
x_train = train_features.astype('float32')/255
x_test = test_features.astype('float32')/255
y_train = train_labels.values
y_test = test_labels.values

# Defines classifier
clf = AdaBoostClassifier()
clf = clf.fit(x_train, y_train)
end = time()

# Predicts and prints accuracy of predictions
y_hat_train = clf.predict(x_train)
y_hat_test = clf.predict(x_test)
training_accuracy = accuracy_score(y_train[0:10000], y_hat_train[0:10000])*100
print('Training accuracy (AdaBoost): %.4f%%' % training_accuracy)
testing_accuracy = accuracy_score(y_test, y_hat_test)*100
print('Test accuracy (AdaBoost): %.4f%%' % testing_accuracy)
print('Time to train: %.2fs' % (end - start))
```

4. **SVC** – the snippet below shows the default SVC implementation and the subsequent GridSearchCV code to find the optimal parameters. For the GridSearchCV exercise, I only chose to vary the C and gamma parameters mainly because the processing time involved was very time-consuming. The other parameters I could have changed were kernel and degree but based on past experience, the default kernel, rbf, tends to produce better results than the other kernels (i.e. linear, sigmoid, and poly) and since degree is only applicable if the kernel is 'poly', there was no point in varying the degree parameter.

## Default SVC

```
# Rescales [0,255] --> [0,1]
x_train = train_features.astype('float32')/255
x_test = test_features.astype('float32')/255
y_train = train_labels.values
y_test = test_labels.values

# Defines classifier
clf = SVC()
clf = clf.fit(x_train, y_train)
end = time()

# Predicts and prints accuracy of predictions
y_hat_train = clf.predict(x_train)
y_hat_test = clf.predict(x_test)
training_accuracy = accuracy_score(y_train[0:10000], y_hat_train[0:10000])*100
print('Training accuracy (SVC): %.4f%%' % training_accuracy)
testing_accuracy = accuracy_score(y_test, y_hat_test)*100
print('Test accuracy (SVC): %.4f%%' % testing_accuracy)
print('Time to train: %.2fs' % (end - start))
```

## Optimized SVC

```
# Initializes the classifier
clf = SVC()

# Defines parameter list to tune
parameters = {'C': [0.5, 0.75, 1.25, 1.5], 'gamma': [0.0001, 0.001, 0.01, 0.1]}

# Defines scoring method
scorer = make_scorer(accuracy_score)

# Performs grid search on the classifier using 'scorer' as the scoring method using GridSearchCV()
grid_obj = GridSearchCV(clf, param_grid = parameters, scoring=scorer)
grid_fit = grid_obj.fit(x_train, y_train)
end = time()

# Get the best estimator
best_clf = grid_fit.best_estimator_

# Predicts and prints accuracy of predictions
y_hat_test = best_clf.predict(x_test)
testing_accuracy = accuracy_score(y_test, y_hat_test)*100
print('Test accuracy (SVC Optimized): %.4f%%' % testing_accuracy)
print('Time to train: %.2fs' % (end - start))
```

5. **Gaussian Naïve Bayes** – As shown in the code snippet below, the AdaBoost implementation was minimal. All parameters were kept as default to standardize the comparison between AdaBoost, SVC and Gaussian Naïve Bayes.

```
# Rescales [0,255] --> [0,1]
x_train = train_features.astype('float32')/255
x_test = test_features.astype('float32')/255
y_train = train_labels.values
y_test = test_labels.values

# Defines classifier
clf = GaussianNB()
clf = clf.fit(x_train, y_train)
end = time()

# Predicts and prints accuracy of predictions
y_hat_train = clf.predict(x_train)
y_hat_test = clf.predict(x_test)
training_accuracy = accuracy_score(y_train[0:10000], y_hat_train[0:10000])*100
print('Training accuracy (GaussianNB): %.4f%%' % training_accuracy)
testing_accuracy = accuracy_score(y_test, y_hat_test)*100
print('Test accuracy (GaussianNB): %.4f%%' % testing_accuracy)
print('Time to train: %.2fs' % (end - start))
```

For the three supervised learning models, AdaBoost, SVC and Gaussian, I didn't encounter many implementation issues as for the most part, I was only implementing the default classifiers. However, for the multilayer CNN, I went through some iterations to improve the accuracy. Firstly, I played around with the activation functions for each hidden layer, trying sigmoid, tanh, linear and relu. Next I tried varying the number of layers, starting from 1 to 3 layers and lastly, I tried removing the dropout layer at the end. The final 3-layer CNN with relu activation functions for each hidden layer and a dropout layer at the end seemed to produce the best accuracy scores.

## Refinement

The improvements made for each algorithm are documented below:

No	Algorithm	Improvements
1	Single-Layer Perceptron	<ul style="list-style-type: none"><li>- Reduced the number of epochs from 100 to 25 to reduce the processing time and it was observed that accuracy did not improve much after 25 epochs</li><li>- Added dropout layer to reduce overfitting</li></ul>
2	Multilayer CNN	<ul style="list-style-type: none"><li>- Changed activation function at hidden layers from sigmoid to relu</li></ul>
3	AdaBoost	<ul style="list-style-type: none"><li>- After it was discovered that the default SVC outperformed the other supervised learning models in terms of accuracy, a GridSearchCV was performed to determine the optimum parameters for the SVC classifier. The optimum parameters for the SVC classifier were identified to be:  C: 1.25 Gamma: 0.1</li></ul>
4	SVC	
5	GaussianNB	

Originally, I had planned to create a CNN using transfer learning. However, as I progressed through the project, I decided to focus solely on comparing the multilayer CNN with other supervised learning models. I felt adding another CNN model would unnecessarily complicate and deflect the focus away from this comparison.

## IV. Results

---

### Model Evaluation and Validation

Below is a summary of the results achieved by the various algorithms used to classify the 24 ASL hand gestures in the testing set provided by Kaggle:

No	Algorithm	Training Time (s)	Accuracy score (Kaggle testing set)
1	Single-Layer Perceptron	39.7	55.4%
2	Multilayer CNN	250.7	87.5%
3	Default AdaBoost	48.9	31.6%
4	Default SVC	478.9	69.7%
5	Optimized SVC	450.1	84.5%
6	Default GaussianNB	0.6	39.0%

Results show that the multilayer CNN is able to achieve better accuracy than the default supervised learning models (e.g. AdaBoost, SVC, GaussianNB). But after applying a GridSearchCV on the best performing supervised learning model, SVC, we see that the SVC classifier was able to achieved similar accuracy scores to the multilayer CNN, 84.5% Vs 87.5%.

However, in terms of training time, the multilayer CNN was able to train and solve the classification problem 2x times faster than the optimized SVC classifier. Since both the multilayer and optimized classifier achieved similar accuracies, both were used to predict the labels for a different set of test images (refer to *Justification* section for results).

### Justification

All the classification algorithms explored in this project managed to outperform random chance (i.e. 4.2%). Even the benchmark single layer perceptron achieved an accuracy of 55.4%. I was able to further improve on this accuracy by +29.1% to 87.5% through the

use of a multilayer CNN. Similar accuracy results of 84.5% were achieved by optimizing a SVC classifier.

Unfortunately, compared to the secondary benchmark, Luis A. Estrada Jiménez and Marco E. Benalcázar's paper on "Gesture Recognition and Machine Learning Applied to Sign Language Translation", which achieved an accuracy of 91.6%, my final model fell a little short, though given the limited resources available, I would say my results are respectable.

To verify the robustness of the models created, I tested the models against a group of hand gesture images provided by Kaggle and group of hand gesture images sourced from the internet. The results were very interesting to say the least. For the group of images provided by Kaggle, both algorithms performed very well.

Algorithm	Number of test images	% of Correct Predictions	Errors
Multilayer CNN	24	100%	
Optimized SVC	24	92%	Predicted: N[13] & Actual: A[0] Predicted: N[13] & Actual: M[12]

The two errors produced by the Optimized SVC may suggest that all the A, M and N hand gestures have been classified as N, which is not hard to imagine as they look very similar:



A[0]



M[12]





N[13]

When given images of hand gestures sourced from the internet, however, both algorithms failed to correctly predict the labels for any of the test images despite the images having very little interference in the background (i.e. background is white for most test images).

Algorithm	Number of images	% of Correct Predictions
Multilayer CNN	8	0%
Optimized SVC	8	0%

These results suggest that both algorithms are overfitted to the training and testing datasets provided by Kaggle. It further indicates that the training and testing datasets used in this classification problem are quite similar, which is why the algorithms were able to superficially perform well when tested against the testing datasets. Hence, when given outside testing samples, these same algorithms were not able to reproduce the same accuracies. A sample of the images provided in the Kaggle dataset and images sourced from the internet along with their respective multilayer CNN and optimized SVC predictions are shown below:

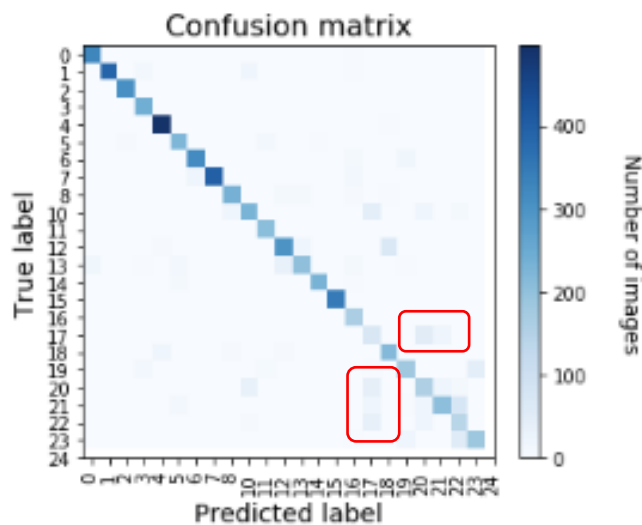
Image included in Kaggle dataset	Image sourced from internet
 <p>True Label: B[1] CNN Multilayer: B[1] Optimized Label: B[1]</p>	 <p>True Label: C[3] CNN Multilayer: O[14] Optimized Label: Q[16]</p>



## V. Conclusion

### Free-Form Visualization

The plot below shows the confusion matrix generated by the multilayer CNN model. The diagonal line represents labels that were correctly identified. Anything outside the diagonal line represents scenarios where the predicted label did not match the true label. I have highlighted in red the types of hand gestures that seemed to get misclassified more frequently than others:



The table below tabulates the hand gestures that were misclassified more than 30% of the time.

No	True Label	Percentage Error	Incorrect Predicted Labels
1	R[17]	51%	U[20] – 36% V[21] – 15%
2	U[20]	40%	R[17] – 15% K[10] – 13% V[21] – 8% W[22] – 3%
3	V[21]	40%	W[22] – 22% R[17] – 7% U[20] – 6% F[5] – 4%

4	W[22]	31%	R[17] – 19% U[20] – 10% K[10] – 2%
---	-------	-----	--

It seems from the table above that the multilayer CNN seems to confuse R, U, V and W hand gestures the most often. Looking at their respective images below, it is not hard to see why some of them might have been misclassified:



R[17]



U[20]



V[22]



W[23]

The results of this confusion matrix are useful in driving future classification problems dealing with ASL. It demonstrates that the specific hand gestures identified above may require more training samples.

## Reflection

To summarize what I have accomplished in this project – I first started out with the problem statement, which was to use machine learning to classify 24 ASL hand gestures with >80% accuracy. Initially, I started with a single-layer perceptron benchmark algorithm and then compared its accuracy to the accuracy achieved by a multilayer CNN, AdaBoost, SVC and Gaussian Naïve Bayes classifier. From there, I observed that the SVC classifier was able to outperform the AdaBoost and Gaussian Naïve Bayes classifiers for this particular problem so I optimized the SVC classifier and achieved accuracies similar to the multilayer CNN, ~87% accuracy. Strangely, I discovered that after testing both the multilayer CNN and optimized SVC algorithms on a few test images, I was able to correctly identify hand gestures provided in the Kaggle dataset but when images sourced from the internet were processed through the same algorithms, the predictions were not very accurate. This led me to conclude that both algorithms were over-fitted to the Kaggle dataset.

All in all, I felt this was a challenging project, which helped me, firstly, appreciate the difficulty of pre-processing data. In the previous Udacity projects, most of the pre-processing code was provided to the students so very little thought was given into understanding what was truly being performed on the input data. A lot of my time and

effort was spent manipulating the input data into forms that could be processed by the various algorithms.

Another aspect of machine learning that I gained a better appreciation for is how accuracies can be manipulated if the training dataset is not diverse enough. Originally, after I had completed implementing the algorithms and had achieved >80% accuracy on the testing datasets provided, I thought I would be able to simply grab any random image of a hand gesture off the internet and my algorithms would be able to correctly identify them 80% of the time. Sadly, this was not the case, in fact, the algorithms achieved 0% accuracy for images sourced off the internet. The fact that the algorithms could not identify hand gesture images sourced from the internet indicates that the training dataset may not have been diverse enough.

Lastly, pulling together the supervised learning and deep learning portions of the Machine Learning Udacity course helped me see the results of both types of algorithms side by side and appreciate that just because the AdaBoost algorithm was good at solving the donor classification problem, did not necessarily mean that it would always outperform other supervised learning models in every problem. For this particular classification problem, the default SVC classifier was 2 times more accurate than the default AdaBoost classifier.

## Improvement

The following are a list of improvements and how they would help address some of the shortcomings highlighted in this report:

No	Improvement	Reason
1	Train more samples of the letters R, U, V and Won the multilayer CNN	The corresponding hand gestures for these alphabets were the most frequently misclassified by the multilayer CNN. More training samples would help teach the model to better distinguish the differences between these hand gestures.
2	Use more diverse training samples	This will help the model deal with overfitting and enable it to correctly predict more images sourced from the internet.
3	Use augmented images	This will add noise to the training dataset and help the model to deal with hand gestures of various orientations and overfitting.

