# CAPSTONE PROJECT

Declan Ayres

## Introduction

The purpose of my project is to train a data set of fonts to be able to recognize characters and symbols. The end goal is to be able to recognize equations and solve them. To that end, my project will use the font files found on Unix servers, specifically the ‘otf’ and ‘ttf’ fonts. Font files contain glyphs and meta data such as the font type, the ascii or unicode value of the font etc. The ascii or unicode is the class (label) of the glyph (labels and class are used interchangeably in this report). Thus the glyphs and labels within the fonts become the basis for creating a labeled dataset necessary for training my model for purposes of svm or CNN. Using this approach, clustering and other intermediate steps to successfully label a dataset is avoided. Such steps are also tedious and can be time consuming. If a

machine learning model can be trained to recognize characters and symbols then the model can recognize math equations . By recognizing equations, the model could be further programmed to solve an equation and provide the solution to the student with an end goal of helping student learn. I am also hoping to showcase my Udacity based ML learning in designing this project.

# DESIGN

To keep the project manageable I have selected only 22 classifiers a-c,x-z,0-9,*,-,/, ., and +. I have also avoided the need for clustering and labeling of data by using pre labeled fonts. The following steps (pre processing, algorithms) describe the design in detail.

## PREPROCESSING

The preprocessing steps include

- A simple shell script to ‘find’ and collate all the font files on the server (to this end i rented an AWS server)

- Using the absolute path to each of the approximately 1200 font files found on the server, a python script does a few transformations of each font ( rotation, skewing and translation) to increase the number of samples used to train. The script then uses OpenCV to extract the contours from each of the glyphs from each font file for a total of roughly 5K samples of each class. I studied MNIST prior to deciding on these preprocessing steps and it was clear I needed a large number of samples to train my CNN and SVM models.
- To extract the contours from each of the 5000 font files I use a python script with OpenCV applying a Gaussian filter to smooth the image. Following this I use adaptive thresholding of the image to highlight the intensity points. Lastly the image is inverted to aid in better detection of contours.
- Extracted contours are put together as training and test datasets for TF and SVM. They are formatted and written out as MNIST files with both images and labels.

## Algorithms

Convolutional Neural Network (CNN) is the first of the two algorithms I used to train my model. CNN takes the input data and puts it through many different layers. The first layer is the convolution, which takes the data and applies convolution to it with biases and weight matrices. After that the pooling layer takes the input and divides it into sub matrices. The max value of every submatrix is put into the new matrix. The next layer is the ReLu layer. This stands for Rectified Linear Units and uses the function $f(x)=max(0,x)$ where x is the input data. This layer is necessary because it normalizes the negative input data to 0 and the rest to itself. Following RELU, the Softmax layer takes the input into a softmax function and outputs the vector with probabilities, the sum of those equals 1. This function says which class in the vector has the highest probability of being the true label after being run through the previous layers. Also there is the cross entropy, which takes two probability distributions and computes the entropy to find how to identify an event from the two. In order to regularize the data and prevent overfitting, there is the dropout layer, which drops out

certain nodes from the neural net with a certain probability. This minimizes the complexity of the neural network model and makes it faster.

The CNN is coded with the TensorFlow libraries and framework. First I create all the initialize weight and bias variable with dimensions of 784 by the number of classes there are which is 22. After that I softmax the linear equation of the matrices which will output the vector of probabilities of each class. Next I run the convolutions and reshape the input image into a 28 by 28 array.The algorithm goes through two convolutional layers. In the first layer, the weight matrix and image are convoluted and added to the bias. Then it goes through the ReLu and Max pooling layers. The output of this becomes the input for the second convolution which goes through the same layers. After that it goes through two fully connected layers with initialized weight and bias to get the final probability distribution over the classes in a vector with the softmax function. In order to reduce complexity and increase efficiency, the distribution vector goes through some loss functions. The 28x28 image matrices are binary images. This means they have intensity points ‘1’ and no intensity (black) points ‘0’. To mimic this binary input vectors, the labels are converted into one-hot vectors. One-hot as the name suggests is a distributed vector, where all elements are 0

except one which is 1. We have normalized both the input image vectors and its labels and makes it easier for matrix multiplication and such other operations required by the cross entropy step.  All elements are zero except one which is 1. These two vectors are inputted into the cross entropy and this cross entropy variable is used for the train steps when you run the algorithm. The train step is an Adam optimization algorithm set with a learning rate of 1*10^-4. The Adam optimizer is a stochastic gradient descent algorithm that uses time steps to continuously creates locally stochastic functions and updates the parameter to the function until the convergence is achieved and returns the final parameter. The Adam optimizer is used to minimize the cross entropy. After this the correct prediction and accuracy are computed. Finally the session is run and the variables are initialized for running the algorithm.

Before running the algorithm, a saver is created so that the model and variables can be saved for future prediction of more data. If one wishes to run the algorithm to generate training data for a new model, then train is set to true so that the algorithm can train on the data. The data is inputted in batches of 50 to be run to the train step and at every interval of 100 the training accuracy is computed and the variables are saved into the

checkpoint file. After 3000 iterations, the final accuracy is computed with the test data set. If you want to predict new data on an existing model, then no_train is set so that the old model can be restored and the new data predicted on that model.

The second algorithm I will discuss is the svm, also known support vector machine. The svm will take the input data and separate it into different classes based on the features. It will maximize the margin between different classes of data points. The support vectors are the data vector points that lie closest to the hyperplane and the other vectors go to zero. The support vectors determine these margins for separating and classifying the data. Sometimes the data cannot be linearly separated in the current dimensional space.  A kernel trick is used to put the data into a higher dimensional space that can be separated into its different classes.

The svm is coded with the scikit learn tools. I create the svm classifier model and then fit it with the batch and labels. Once it is fitted, I store the model into a file using joblib and predict the accuracy of the test data. Then I can predict new data with the saved model.

## PostProcessing

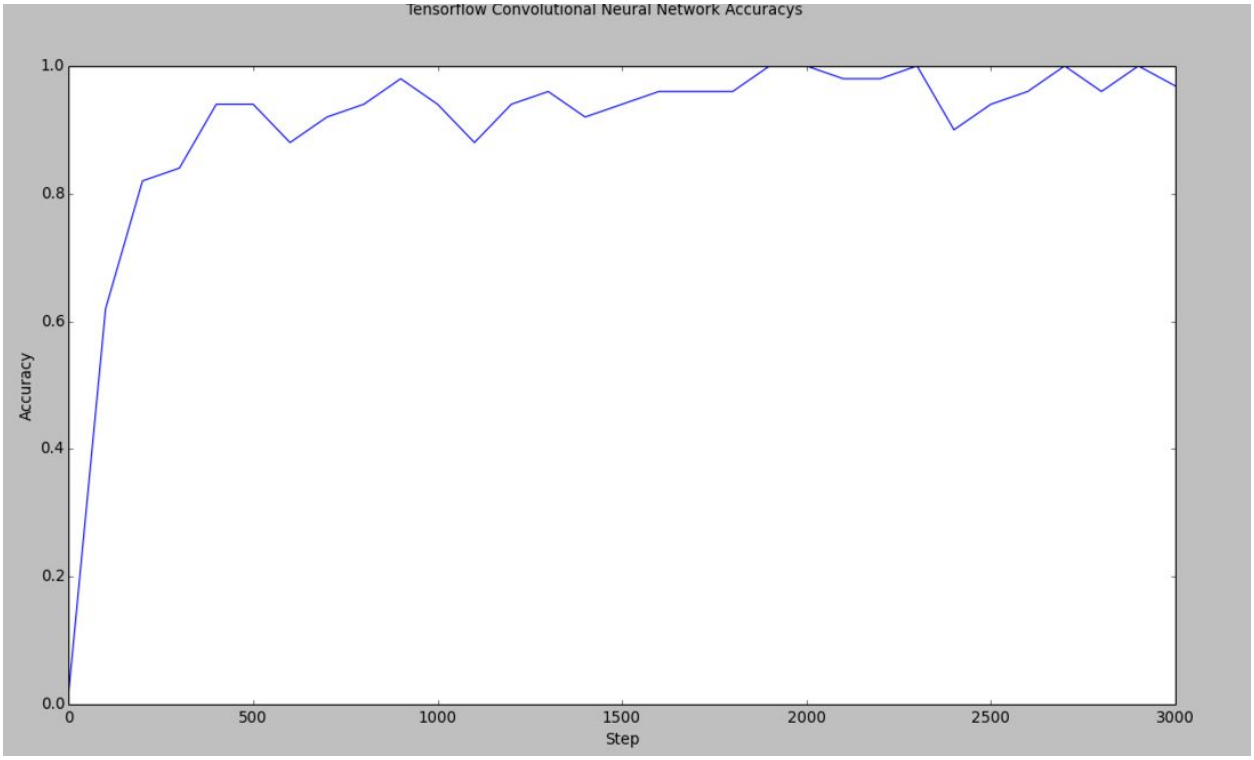
Post processing includes the steps to create the data to be predicted. To post process new data, which are going to be images of equations, I created an app on my iphone that will do this for me. This saves a number of steps on the server side otherwise needed to ‘filter’ an image for contour extraction and such steps. The app will first use the camera to take a photo of an image, for example an equation from a textbook. The camera image will then be processed into a black and white image. During the processing the photo will go through filters such as a Gaussian blur filter that smooths the image. Then it will go through an adaptive threshold filter which accentuates the intensity points in the image while degrading other points so that a contour detection can clearly detect the edges. After this the image will be inverted to help in the contour detection process. There is also a slider to adjust the intensity of the image. Once the image is fully processed, it gets sent to the remote server for contour detection and extraction. The contour extraction program will extract the contours from the image and store the files as 28 by 28 numpy arrays in a contours directory. These 28 by 28 numpy arrays are processed further to create

MNIST formatted compressed file to be used as input to both the Tensorflow and SVM models. The difference between creating the mnist files for the original data set and the new data set is that only one data set is created without labels. The program creates the mnist file and this becomes the input to the algorithms to predict its classes.

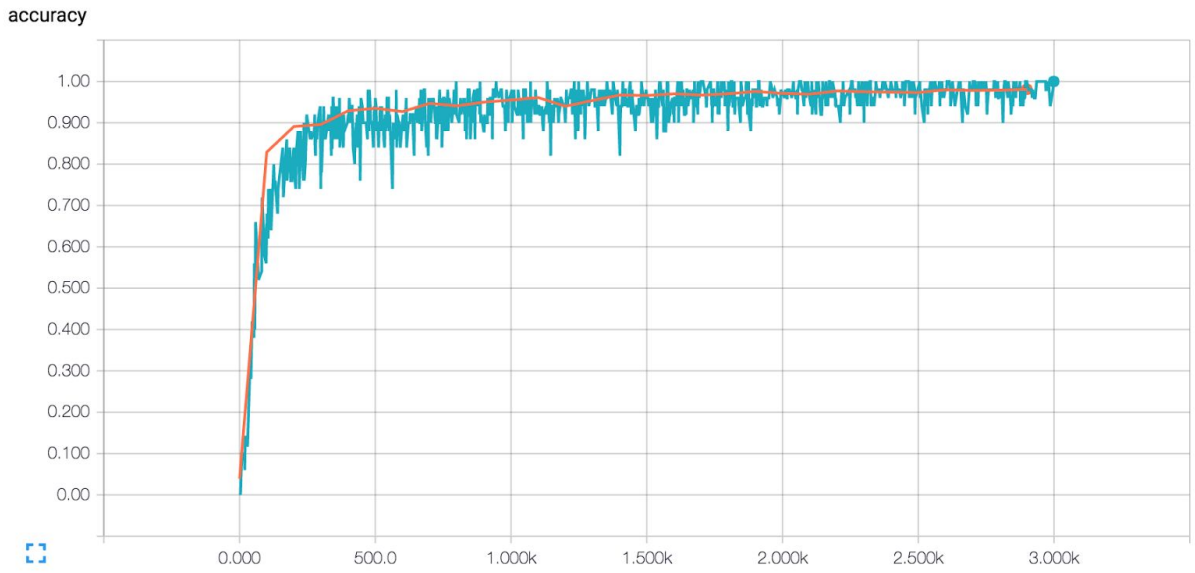
## Results

The results showed that both the cnn and svm have very high test accuracies. The final test accuracy for the cnn is 0.968723. The final test accuracy for the svm is 0.955537059751.

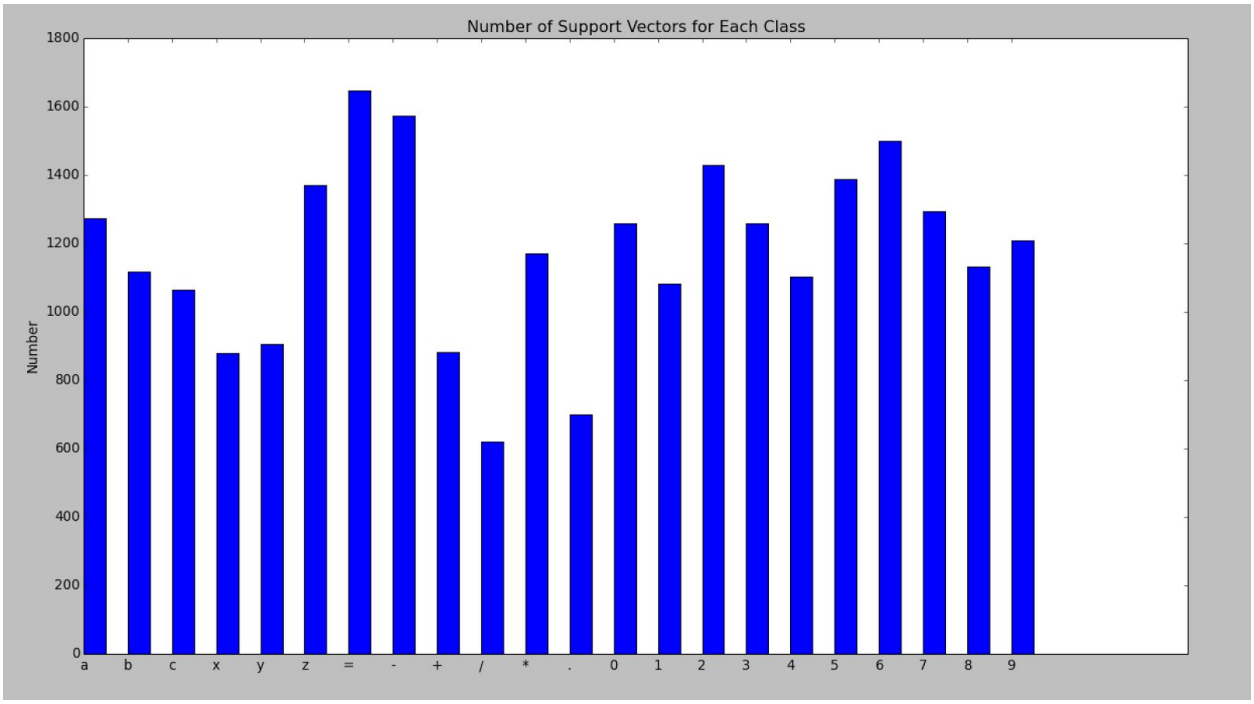
SVM performed better when it came to predictions giving an accuracy of 50% or more.



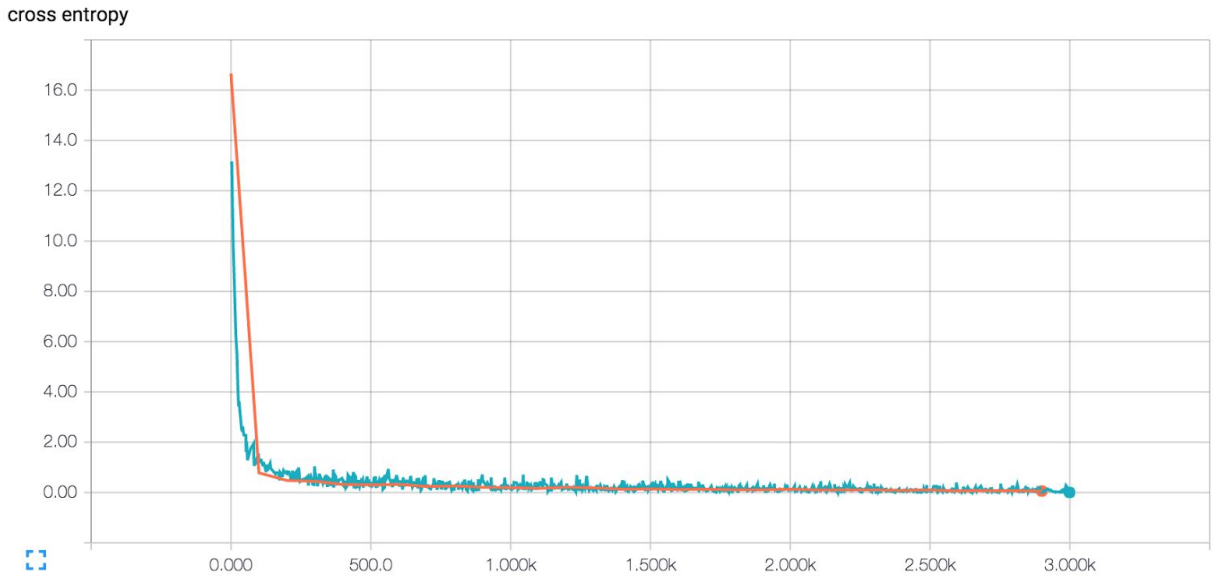

The above graph shows the training accuracy at every 100 steps as a line graph. Analyzing the graph can see that the accuracy is high after the first 200 steps and the last point it the final test accuracy. The results are consistent with the data and the svm and cnn work equally well but the cnn has a little higher accuracy.




The above graph shows the training and test accuracy over the 3000 steps.




The above graph shows the number of support vectors used in the SVM classifier.




The graph above shows the cross entropy values across the time step range of the runtime of the CNN. The blue line represents the cross entropy on the training data and the orange line represents the cross entropy on the test. Analyzing this graph one can see that the cross entropy starts high which means there is a lot of uncertainty between the data and becomes very low fast approaching zero when the algorithm has learned the data.

## Conclusion

The project demonstrated the skills I learned through the Udacity ML Nano degree program. I have applied this learning to create, compare and contrast two different algorithms for predicting character from 22 different

classes. This has been a journey to use these machine learning algorithms to create an app that can use these concepts to solve math equations and help students learn better. I hope that this project has the potential to grow to become a teaching assistant for students.