CAPSTONE PROJECT

Declan Ayres

INTRODUCTION

Optical character recognition is the process of converting text characters into machine language so that computers can process and recognize them. A brief history of OCR started in 1914, when Emmanuel Goldberg created a machine that read symbols and translated them into conventional telegraph code (https://en.wikipedia.org/wiki/Telegraph_code).[1] Early versions of OCR worked on one font at a time. A very early application of OCR was assisting the blind in hearing text from a computer. Another useful application of OCR is translating bank statements and invoices for data entry. One of the first modern optical character recognition

[1] https://en.wikipedia.org/wiki/Optical_character_recognition

algorithms was the Tessaract, created by Google in 1985. Tesseract was written in C and C++ and made as free software in 2006. It has many good features such as output text formatting and page layout analysis. Over the years Tesseract users from other parts of the world have created support for many languages. Now tesseract can recognize over 100 different languages. OCR has come a long way since Tesseract and accuracy has improved with the advent of algorithms such as convolutional neural networks.

OCR is relevant to my project the purpose of which is to recognize characters and symbols. I hope to accomplish OCR using Machine Learning techniques I have learned as part of the Udacity ML Nano degree program.

DEFINITION

For now my project’s goal is to recognize characters and symbols with a future goal of recognizing equations. In machine learning terms this is a multi-class classification problem. To this end the 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.

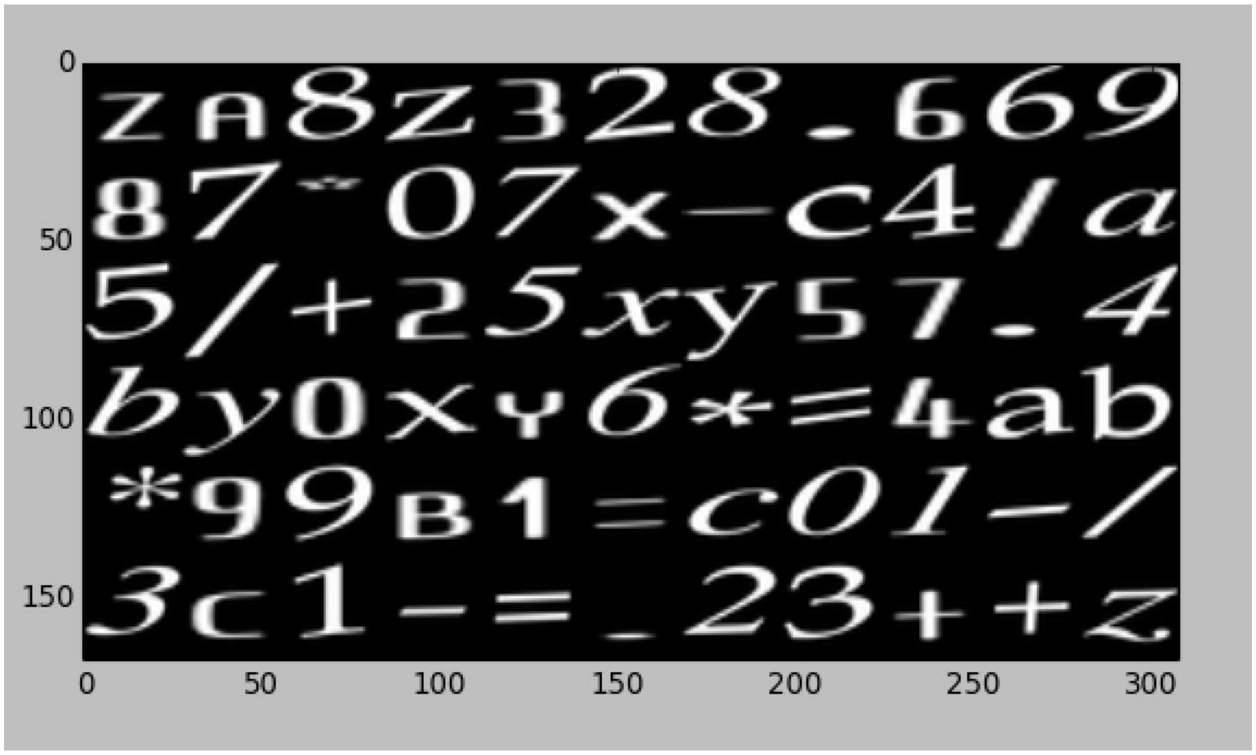
Such steps are also tedious and can be time consuming. For my method, I will be training the data through a cnn and svm and comparing the two models and results. The results I expect to see from these two algorithms are training and test accuracies greater than 90% and prediction accuracies greater than at least 60%. 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.

ANALYSIS

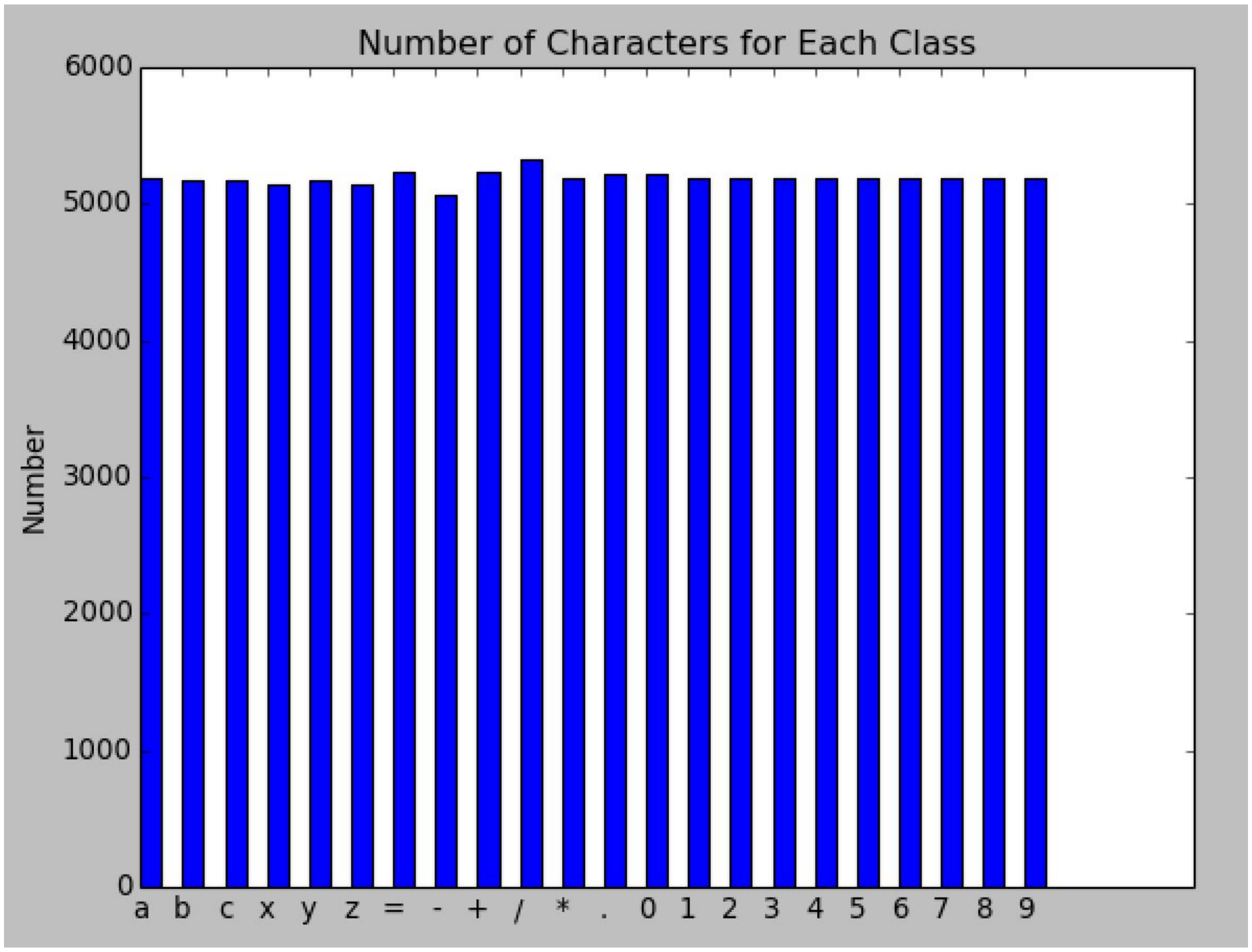
The basis of my dataset are the otf and ttf files. Otf and ttf stand for open type font and true type font. The otf and ttf files reside within the usr shared and local directories of the unix server. The font files contain glyphs and labels in the form of unicode. The glyphs are the symbols and characters which form the data set. For example, the letter a in a specific font is one glyph. The unicode representing the glyph is its label. This obviates the need for clustering since the glyphs are already labeled. I describe in the preprocessing section how the otf and ttf files are processed into the glyph files before being converted to the mnist files. A sample composite of the images I gather from the fonts files is shown below:

Figure 1.



The above image shows a composite of some of the glyphs that appear in my dataset. An interesting aspect about the data is that it has both the glyphs and the unicode number labels so that I do not have to do further tedious preprocessing steps such as clustering and other unsupervised learning techniques. Obtaining the font files and extracting the glyphs is not too hard other than finding the suitable python libraries like fontforge for the extractions and transformations and the learning curve thereof. I had to learn how to use fontforge for font processing and openCV for contour extraction to produce the final input data set which are the Mnist formatted files. The typical image size varies depending on the font but during conversion to Mnist, the images get resized to become 28x28 numpy matrices. There are a total of 114136 images in the data set.

Figure 2.

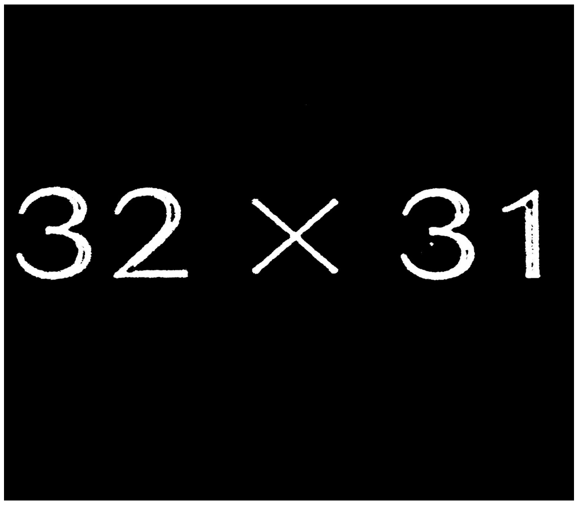


The above graph shows the number of glyphs per class. As can be seen, the distribution is very uniform and each class have slightly over 5000 character symbols.

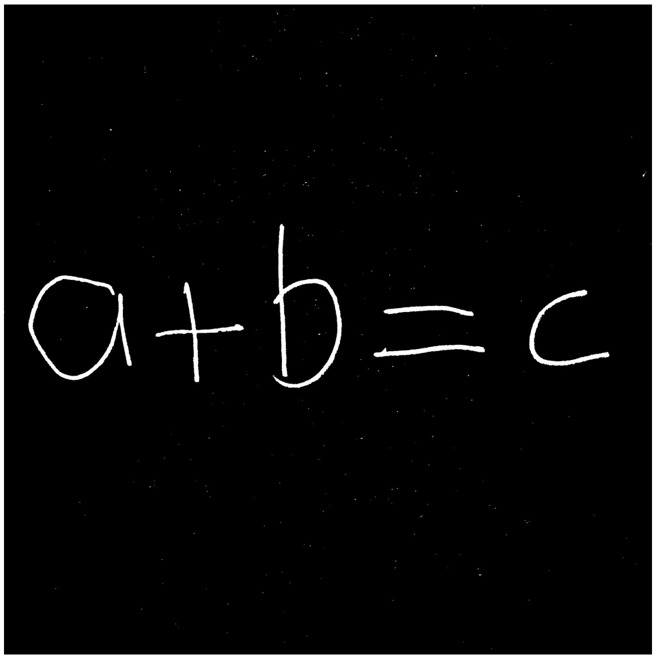
To predict new data, I use my camera app to take pictures of equations.To create the MNIST dataset for prediction purposes, I took a picture of an equation from a math textbook and another of a handwritten equation. Before the images were sent to the server, my app processes the image. These preprocessing steps include Gaussian blurring to smooth the image, adaptive thresholding, and turning the image into a black and white image, i.e. a binary image. After the images are processed, they get sent to the server for contour extraction.The phone app was therefore only to the extent of preparing the image for contour extraction. The extraction and MNIST all happen on the server. Following are some samples of these images and their contours.

The following are the images after I process them with the app (includes Guassian Blur, Adaptive Threshold and binarization) .

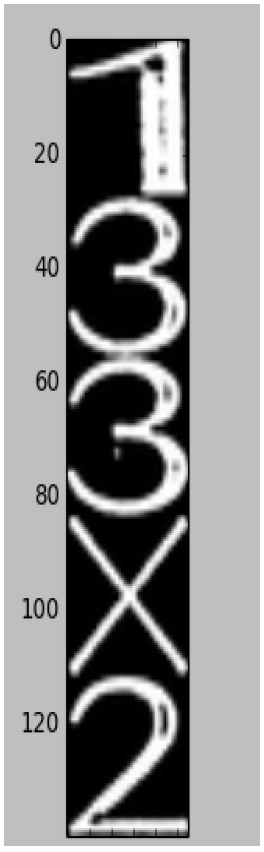
Figure 3.

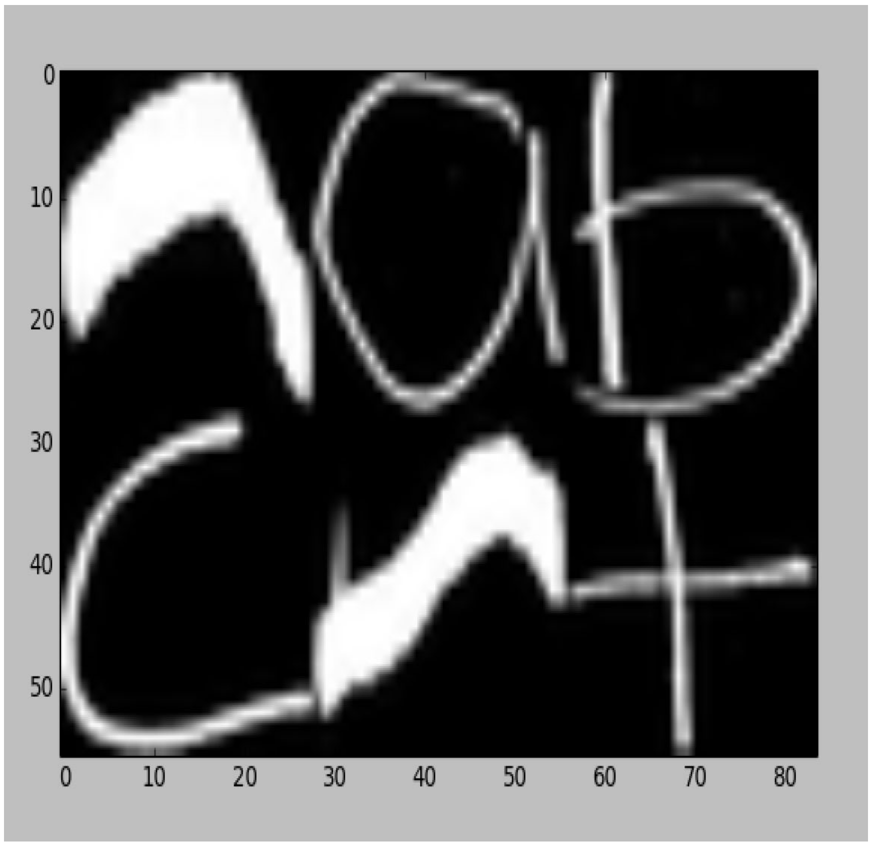


32×31



$a+b=c$





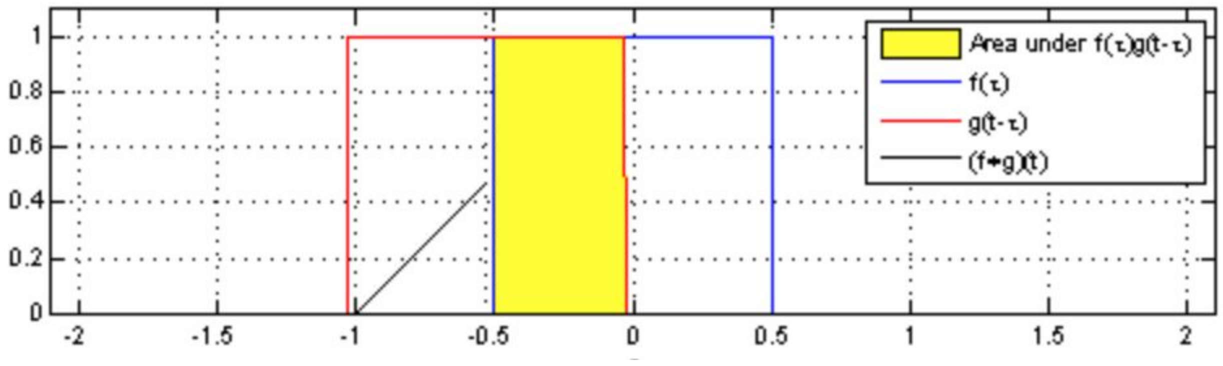
The above image are the composites for the contours of the two equations. Once I have the contour files, I convert them into the MNIST files which becomes the data sets that I input into the cnn and svm for prediction.

ALGORITHMS

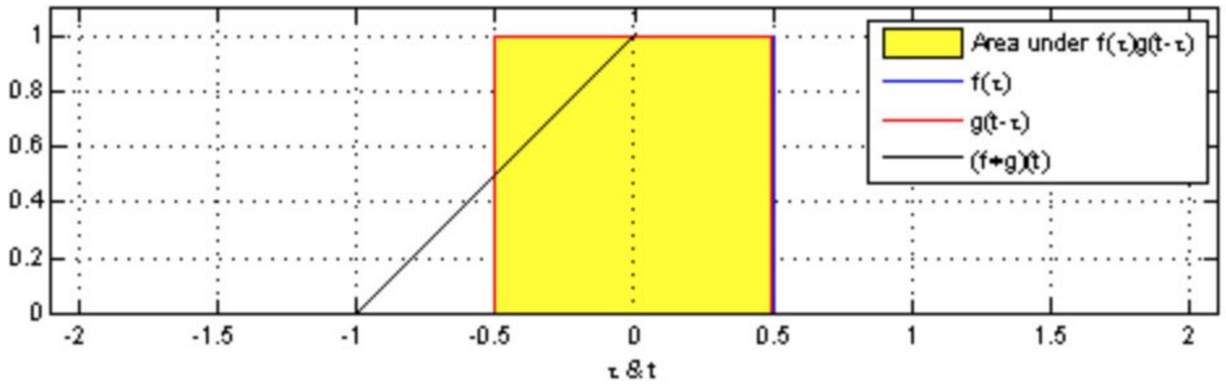
CNN

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. A convolution is a commutative operation on two functions. It is the total likelihood of all the possible probabilities of reaching a number. Whatever the first functions takes as the input has a direct influence on the second function’s input. For example, the picture below shows two functions convolving around each other.

Figure 4.

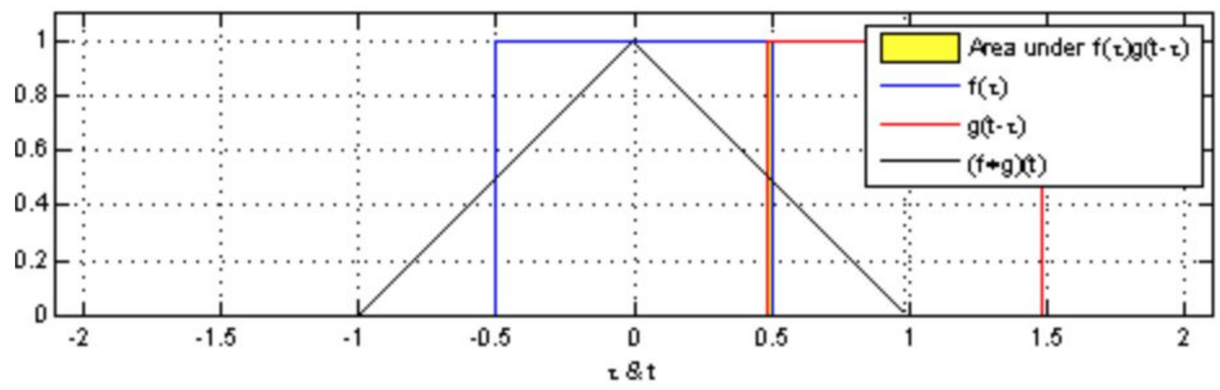


As the g function moves from across f, the convolution of f and g is being created, which is the black line.[2]



Now when the g function overlaps with the f function and the area under f and g is maximized, the convolution of f and g is also at its maximum point.

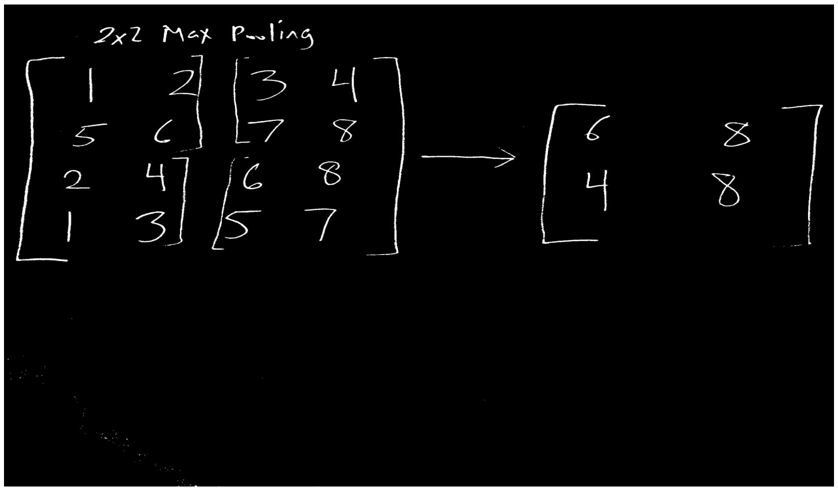
[2] Image taken from Wikipedia



As the area between f and g decreases the convolution of f and g also decreases.

The pooling layer takes the output of the convolution layer as its input and divides it into sub matrices. Max pooling means that the maximum value of every sub matrix is taken and put into the new matrix. Max pooling reduces the dimensionality and overfitting.

Figure 5. Example of a 2x2 max pooling



$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 2 & 4 & 6 & 8 \\ 1 & 3 & 5 & 7 \end{bmatrix} \longrightarrow \begin{bmatrix} 6 & 8 \\ 4 & 8 \end{bmatrix}$$

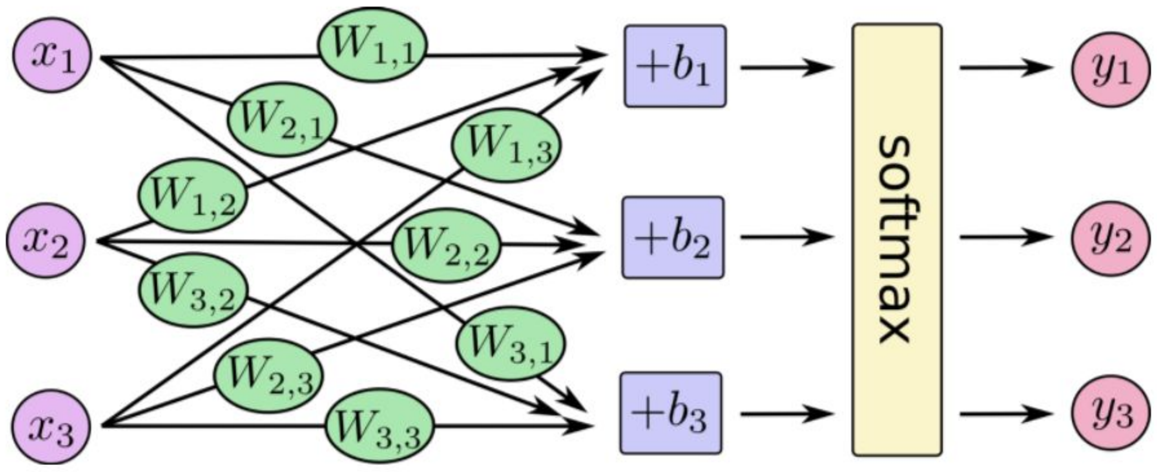
In the above image using 2x2 max pooling on a 4 by 4 matrix with strides of 2, the matrix is divided into 4 2x2 submatrices and the max value of each is transferred to the new matrix. The result is a dimensionality reduced matrix.

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 output of the max pooling layer becoming 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 output of ReLu 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.

After that there is the fully connected layer, which connects all the pieces of the cnn together. The fully connected layer takes the output of the previous layers as input and performs matrix multiplication and addition with the weights and biases. The output of the fully connected layer is a vector with a length of however many classes there are. This vector goes through loss functions such as 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.

Figure 6.



The above is an architectural diagram of cnn. The input xs go through the convolution with the weights and biases and then goes through the softmax to output the ys.[3]

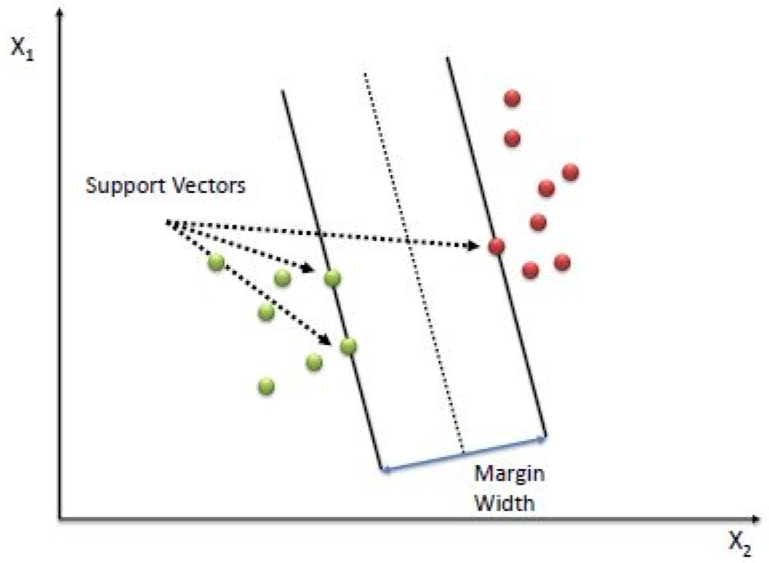
SVM

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 kernel function takes inputs and maps them to higher dimensions so that a hyperplane can linearly separate the classes that couldn't be linearly separated in the original feature space. The loss function that svm tries to minimize is

[3] Image from tensorflow.org

the hinge loss, which is defined as the max of 0 and the classifier's decision function multiplied by the correct class. Since the decision function is dependent on the weights, then having small weights will make the loss function output smaller so minimizing the weights will maximize the margin. If the predicted label is correctly classified then the weights get smaller and if the label is misclassified the weights get bigger.

Figure 7.



The above image is an architectural diagram of svm that shows the support vectors lying on the margins and the hyperplane as the dotted line separating the data classes.[4]

BENCHMARK

METRICS

The metrics that I will use to test the success of my model are accuracy and the cross entropy loss function. Entropy by definition in the context of information theory is the uncertainty of the measurement. The lower the uncertainty the higher the predictability. My model requires higher predictability/accuracy. Therefore, I want to minimize the loss so that the cross entropy approaches zero. The other overarching metric for my project is the accuracy of its predictions.

[4] http://www.saedsayad.com/images/SVM_2.png

Accuracy is the number of total correct predictions over the total number of predictions. Finally the F1 score, which is defined as 2*precision*recall/precision+recall, does not appear to be relevant since the project uses a uniform distribution of the classes (i.e. the F1 will yield the same score for all the classes used). Since there are 22 classes, calculating precision or recall for every class is repetitive. The balanced distributions of labels makes it even stronger argument in favor of accuracy as the benchmark over other measurements.

For my benchmark I used the <u>k-nearest</u> neighbors algorithm. The KNN is a simple algorithm that looks at the k nearest neighbors of each data point and assigns the class of the data whatever is the majority of the neighbors classes. When I ran the algorithm, it got a very high test accuracy of approximately 98%. This would be a very high benchmark but the prediction accuracy is also very important. The prediction accuracy for the printed equation image was 80% but the prediction accuracy for the handwritten image was 20% and only got one character right which is much lower. Thus I found knn algorithm to be inconsistent. To reach this benchmark I want to have both my cnn and svm algorithms have at least as good test accuracy and at least one of them have better prediction accuracy for the printed and handwritten images. My goal for my model is for accuracy to be as close as possible to 100%.

METHODOLOGY

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.

CNN using TENSORFLOW

The CNN is coded with the TensorFlow libraries and framework.[5] 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.

[5] https://www.tensorflow.org

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 20000 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 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.

Since the MNIST dataset labels are one hot arrays, I convert the arrays to the index of the 1 in the array. For example, if the array is [0,0,1,0], then this would be converted to 2.

Hyperparameters

I tried many different combinations of hyperparameters for my model to test how it changed the results. First I set the keep probability for the dropout of the cnn to .25 which means there is a 25% chance of keeping a node in the cnn and something very interesting happened. The training accuracy was .9 or better until the 11000th step when the it suddenly dropped to below .1

and stayed that way for the rest of the steps. The final test accuracy was 0.0470 so this is equivalent to random guessing. I do not know why this happened but my hypothesis is that so many of the nodes were being dropped out that eventually the cnn miscalculated and got wrong values but thought these were the correct values. I also set the keep probability to 1 so that none of the nodes were being dropped out. This resulted in a test accuracy of .9923 so reducing the dropout rate below .5 does not have a significant effect on the accuracy. The dropout rate is equal to 1 - keep probability. For svm, I tried C = 10 and gamma =.001 and got test accuracy of 0.9778.

CNN:

keep probability	steps	accuracy
0.5	3000	0.9687
0.5	20000	0.9922

1	20000	0.9923
0	20000	0.0470
0.5	1000	0.9555
0.1	3000	0.0470
0.3	3000	0.9753

SVM: (C=Regularization parameter for penalties for misclassifications,

gamma= effects the kernel function between two data points

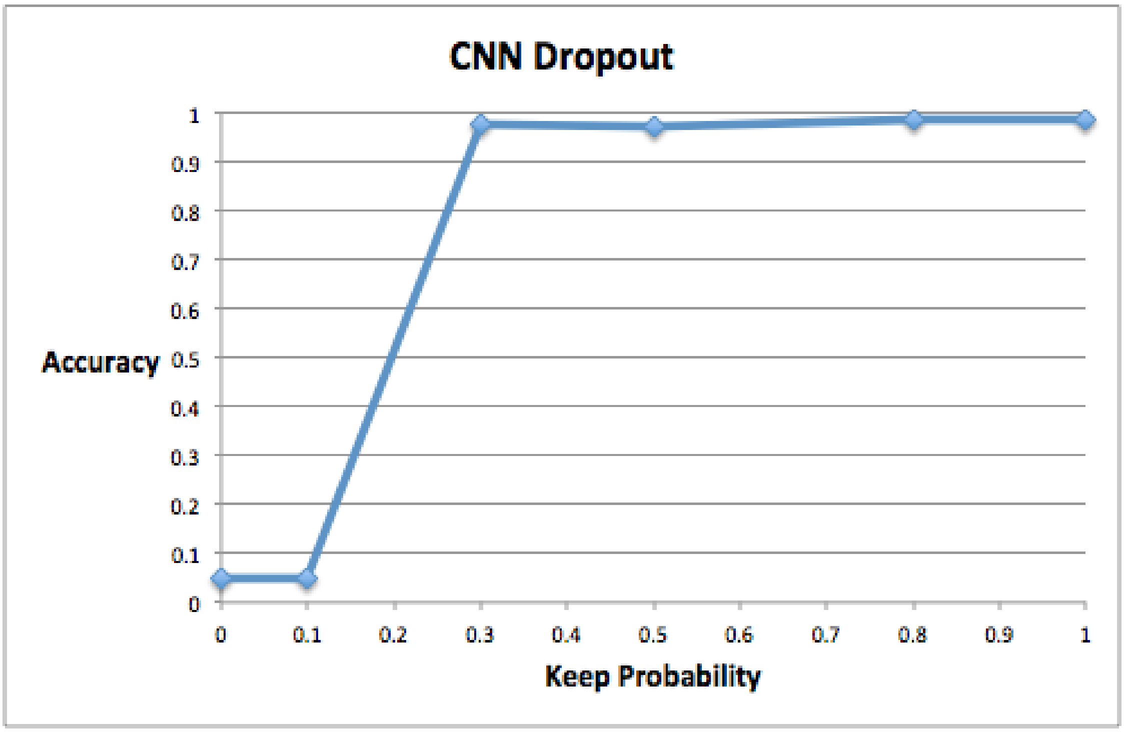
C	Gamma	Accuracy
1	1/784	0.9555
10	0.001	0.9778
100	0.1	0.9869

POST PROCESSING

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.

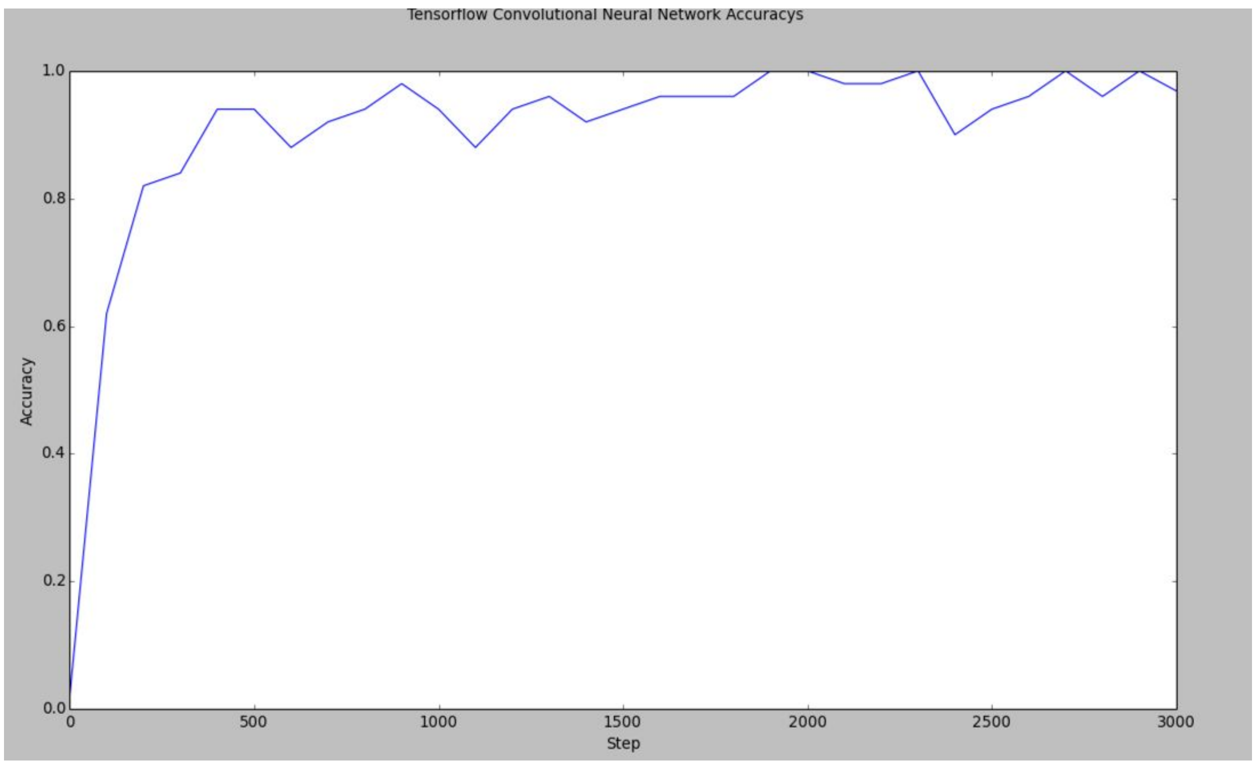
RESULTS

The results showed that both the cnn and svm have very high test accuracies. The final test accuracy for the cnn is 0.9687. The final test accuracy for the svm is 0.9555. To improve the test accuracy of the algorithms, I changed the hyperparameters. For the cnn, the main hyperparameter is the number of running steps which I originally set to 3000. But as the number of steps increases, so does the accuracy. So changing this hyperparameter to 20000 steps got a test accuracy of 0.9922. The model is very robust because the test accuracy is similar to the training accuracy. Changing the hyperparameters of the cnn, such as the dropout rate and the number of steps will not dramatically change the results as long as the dropout is not 0 and the steps is enough to train the entire data set. To test this, I conducted an experiment where I set the dropout keep probability to 0 and the result was that the training accuracy was below .1 every time and not improving because every node was being dropped from the cnn so there was nothing left. I also set the number of running steps to 1000 so only half the dataset was being trained and it still got a very high test accuracy of 0.95549.



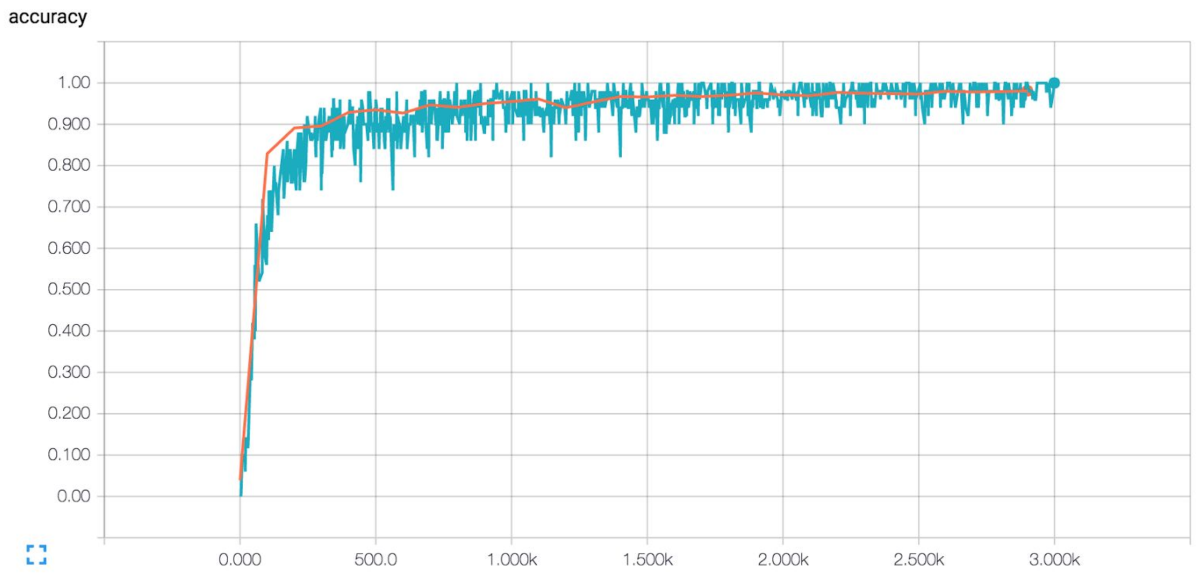
The above graph shows how the accuracy changes as the dropout rate is changed. When the keep probability is .3 or higher the accuracy will be high but if it is 0.1 or lower it will not be able to learn.

Figure 8.



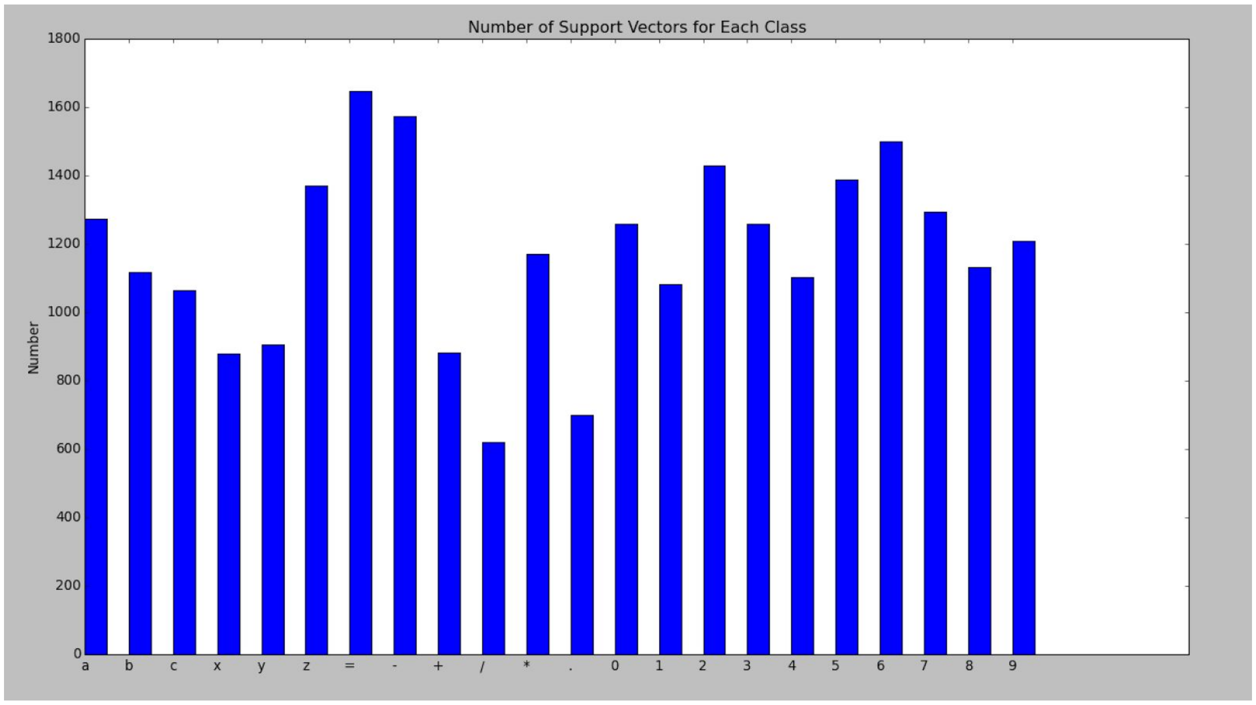
The above graph shows the training accuracy at every 100 steps as a line graph. Analyzing the graph can see that the accuracy is high after the first 200 steps and the last point is the final test accuracy. The results are consistent with the data and the svm and cnn work equally well but the cnn has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

Figure 9.



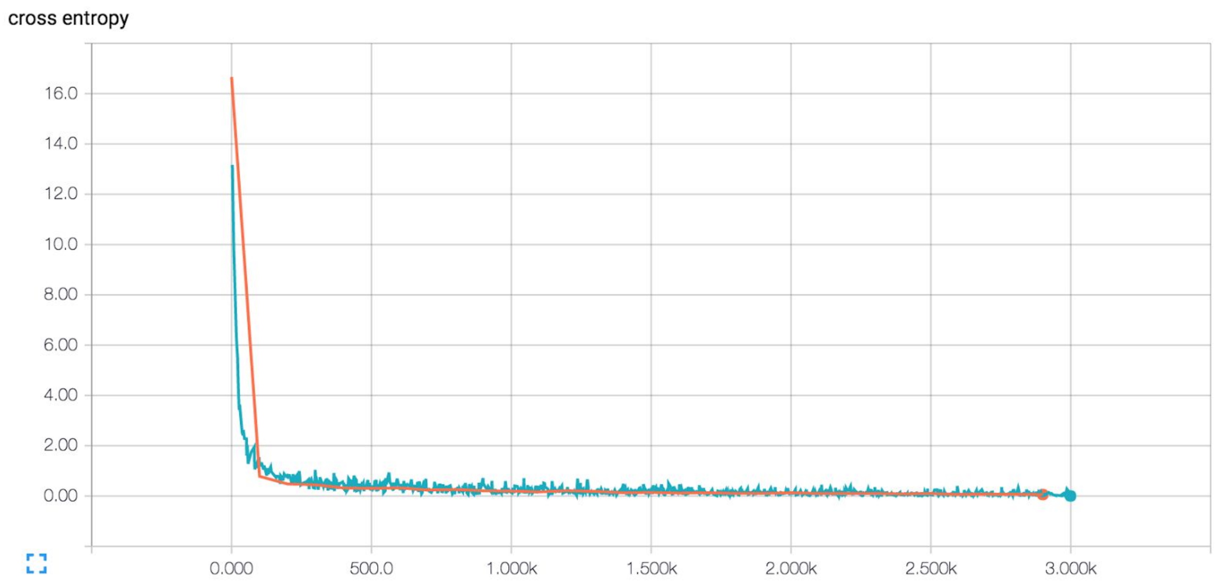
The above graph shows the training and test accuracy over the 3000 steps. As can be seen the training and test accuracies have a very similar shape which means the model is robust.

Figure 10.



The above graph shows the number of support vectors used in the SVM classifier. As can be seen some of the classes have more support vectors than the others, especially = and -. Since the support vectors are what create the margin of the hyperplanes, having more support vectors means that it is harder so separate the data and create the hyperplane. The reason that = and - have the most support vectors is that their glyphs look similar to each other and have similar features so the svm needs more of their support vectors to separate them.

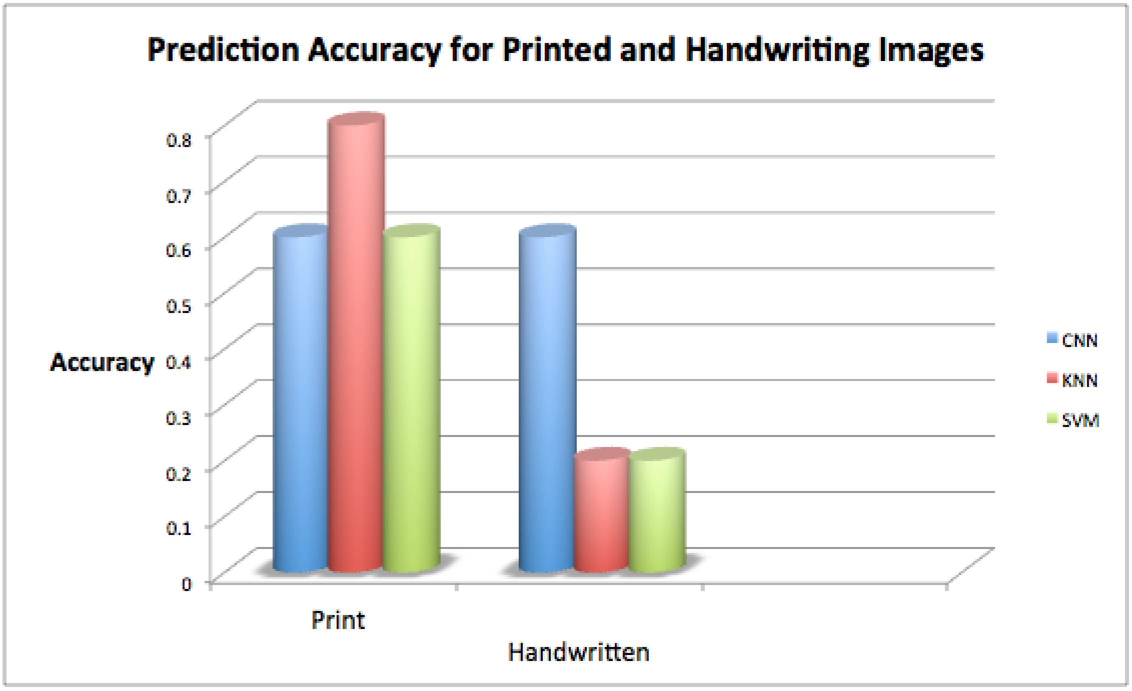
Figure 11.



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.

Comparing my benchmark knn model to the cnn and svm model, they all have very similar test accuracies, within .03.

Figure 12.



This graph shows the prediction accuracies with the different models using the processed images of equations. As can be seen, the KNN has the highest accuracy for the printed image at 80% but cnn and svm also have good accuracies for the printed image at 60%. However, for the handwritten image, the cnn has the best accuracy at 60% but the knn and svm have very low accuracies and were able to correctly predict one character.

Conclusion

My explanation for why the knn succeeded with the printed dataset is that the knn does well when it remembers the data. The image that I used was from a textbook so the font was probably one that was in my training dataset. This book image fell in the memorized font set that knn trained on. Thus it was able to predict with greater accuracy. Also to clarify, I wrote the iPhone App to save myself some post processing steps on the server. Using the app I am able to apply all the filters to the image before sending it to the server for contour extraction. The app itself does not do any prediction.

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.

Improvements

First and foremost I would like to add more samples for each of the 22 classes. Knowing how well MNIST performs with 6000 samples for each of the 10 classes, if I could add more handwritten samples coupled with the samples from font files I believe my CNN and SVM models will perform predictions with better accuracy.

I would like to experiment with the various SVM kernels particularly the HOG kernel because I would like to see if using HOG to detect features causes SVM to perform even better.

I would like to distribute the SVM and CNN models across multiple servers to speed up the training step. Currently for 20000 CNN steps the Tensorflow model takes roughly an hour and a half. Using the parallel Tensorflow support I would like to speed up the process. More so in light of the fact my data set will only increase in size distributed processing is very appealing.

I would also like to refactor and compact the code so that some of the duplicate steps are avoided.

As mentioned earlier the end goal of this work is to recognize and solve equations. I realize it is a lofty goal and is beyond the scope of the current project. However it is one i'd like to continue to pursue in future.