# Machine Learning Approaches to the Recognition of Handwritten Mathematical Symbols

*COS 429 Final Project*
*January 17, 2017*

Amandeep Saini
assaini@princeton.edu

Jelani Dennis
jdenis@princeton.edu

Omar Mukadam
om@princeton.edu

## Abstract:

In this paper, we explore techniques of classifying handwritten mathematical symbols using a dataset of online (positional + temporal data) handwritten symbols. We extracted the meaningful parts of the dataset and generated downsampled images of of each symbol. For classification, we designed a simple neural network, a multi-layer convolutional neural network, and a support vector machine. Both neural networks (implemented in Tensorflow) tended to learn but overfit, with the convolutional net surpassing the simple net in performance. We think that implementing robust feature descriptors (like Histograms of Gradients) of the data would result in less overfitting. The support vector machine was implemented in scikit-learn and resulted in reasonable prediction performance, although curiously less accurate than our MLP despite much longer training times.

Our code and data are available in this GitHub repository:
https://github.com/MkdmOmar/Mathematical-Symbol-Recognition

# 1. Introduction:

## 1.1. Intro to Recognition of Handwritten Mathematical Symbols

Recognition of mathematical symbols and expressions is a tough problem in the fields of computer vision and machine learning. Part of the reason for this is due to the large variety in writing styles and enormous amount of mathematical symbols. This is also combined with the fact that mathematical expressions typically involve complex recursive structures to represent the order of operations and how the mathematical logic is organized. Such complex structures represent enormous challenges in how a machine can learn to recognize and segment the individual components of a mathematical expression then learn the logical relationships between each of them to reconstruct them in an electronic format like LaTeX or MathML.

Applications of this problem are numerous, especially in academia. For example, students in mathematically-oriented classes can take pictures of notes and convert them to appropriate electronic formats like LaTeX or MathML. Additionally, these techniques can be used to strengthen current OCR implementations to include complex mathematical expressions.

## 1.2. Background Literature

We read some papers on the recognition of handwritten mathematical symbols. Most papers took the approach of recognizing online data, where a user's input is recorded as strokes composed of point/time tuples. This is different than recognition techniques conventionally used for image datasets, like MNIST, where the entire image (flattened image vector) is fed into a classifier. Some papers took the approach of creating feature descriptors for symbol strokes and feeding them into classifiers like Multi-layer Perceptrons or techniques like Greedy Time Warping (Thoma). The Competition on Recognition of Online Handwritten Mathematical Expressions (CROHME) is a competition on recognizing handwritten mathematical expressions that has generated numerous publications in this area. Other relevant papers explored methods of partitioning an expression into individual mathematical symbols then developed algorithms complex for converting a two-dimensional arrangements of symbols into a typeset expression (Matsakis) by learning to recognize the logical relationships between each component of the expression.

### 1.3. Our Approach

We analyzed a dataset of online mathematical symbols and classified it using techniques like neural networks and support vector machines. To do this, we first had to preprocess the input data to a format amenable to our models.

---

## 2. Data Retrieval and Formatting:

### 2.1. Intro to Dataset

We used the HWRT database of handwritten symbols. This database contains on-line data of handwritten symbols such as all alphanumeric characters, arrows, greek characters and mathematical symbols like the integral symbol.
Data host page: http://www.martin-thoma.de/write-math/data/

The data is structured as a list of lists of dictionaries. Each internal list comprised a single stroke (defined as a single contiguous movement of the input pen). Each stroke is a list of dictionaries where each dictionary has the keys: "x", "y" and "time". X and Y correspond to the coordinates of a single input point whereas time is the UNIX time at which that input point was registered.

Example data of two strokes comprised of two input points each:

```
[    [ {"x":206,"y":436,"time":1400943242038}, {"x":205,"y":416,"time":1400943242054} ],
     [ {"x":204,"y":398,"time":1400943242070}, {"x":202,"y":383,"time":1400943242087} ]    ]
```

The original dataset contains 369 different symbols (described in symbols.csv).
The original training data contains different examples.
The original test data contains 17074 different examples.

The large size of this dataset is one of the main reasons why we chose it over other alternatives.

## 2.2. Extraction and Formatting Data

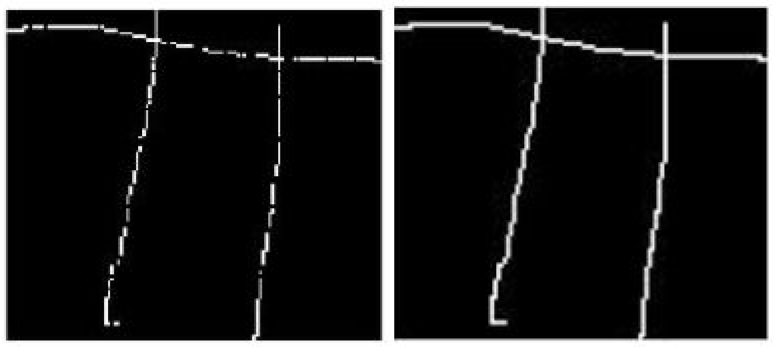
We deliberated whether or not to include timing data in our classifiers. After much thought, we decided that including time data would complicate our models and require far more computational power than was available to us. This is because users write symbol strokes in different orders. For example, in writing the the Greek letter "π", some users may draw it as one contiguous stroke, whereas others may draw the legs first as two strokes then the top bar as a last third stroke. Furthermore, other users may draw the top bar as the first stroke and then the legs as the final two strokes. Therefore, if we included time data in our models, they would have to be robust enough to account for variable stroke sizes and changing orders of strokes.

Therefore, we decided to take all of our data and extract the x,y coordinates for each symbol and convert it to a square image. Our methodology was as follows:

1. **Extraction of x,y stroke data**
2. **Interpolation of data between successive points**
3. **Converting data to images & downsampling**
4. **Converting to proper input format**

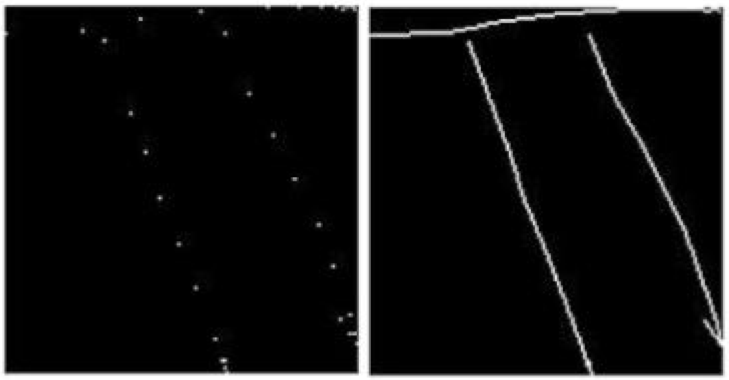
**Extraction of x,y stroke data:**
We decided to work on and classify 24 mathematical symbols. We extracted the x,y data in all the testing/training examples for those symbols. This would leave out all timing data (as explained in the section above).

**Interpolation of data between successive points:**
The output of the previous step would give us a list of sparse points. To make the data less sparse, we extrapolated missing points by solving for the line between successive points and then discretizing the resulting line into pixel values using a 2d-array of 0s and 1s. To do this, we'd find the slope between the previous and current point, and interpolate points assuming a straight line between each point, and rounding to the nearest integer. Since the points were generally drawn in quick succession under a single stroke, this linear assumption was more than adequate for our purposes, especially since we'd be downsampling later on. This step was performed in java, and an example result can be seen below for the symbol pi:



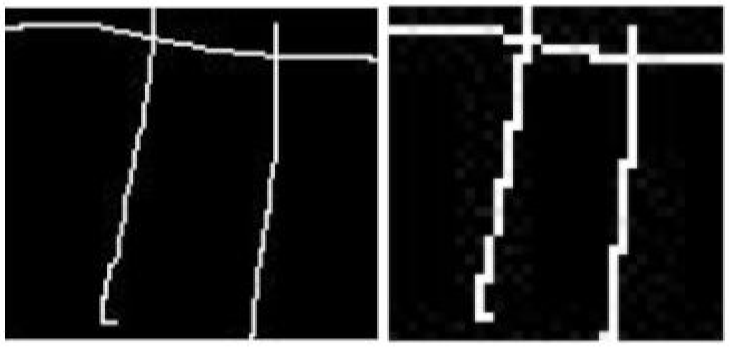
In the example shown to the right for pi, the input image (left image) had a decent number of points, so even if we had skipped this interpolation step, it would not have been very detrimental. The output image (right image) has fully connected strokes.

There were many samples that did not have an abundance of points (like the pi to the right). The need for interpolation is more apparent in the example below for pi, where the original data (left image) had a very sparse amount of points, but through which, we were still able to recreate a very good 2-d image representative of the original image (right image).



**Converting data to images & downsampling:**
To convert each symbol's data to an image, we printed out the 0s and 1s from the previous step to a csv file that could be easily read into matlab, which could interpret the matrix as an image. However, before writing out to the file, we'd crop the image so that we would not have excess and unnecessary rows or columns of 0s. The image above was with the cropping already having been done.

After reading in the matrix from the CSV files into matlab, we converted the matrix down to a 35x35 binary matrix, and this was done by going through each pixel and projecting it onto a 35x35 grid if it was a 1 and ignoring it if it was a 0. To project, we simply factored it down by the proper scale to contain it with 0-34, and added 1 to the value. This way, we had a minimal amount of loss of information. The results of this can be seen in figure below, where the image on the left is the original large-size image and the image on the right is the resulting downsampled 35x35 image.



**Converting to proper input format:**

To convert this data to a proper format to feed into our training algorithm, we again used Matlab. We had saved each sample(image) as a separate matrix, and would go through each image, flatten the matrix, and output the result. We would, at the same time, write the label corresponding to each flattened matrix in another file that would serve as the ground truth input for our training. Lastly, we split up our data set so that 80% would be saved as training data, and 20% would be saved as test data, and the samples were chosen uniformly at random within each label.

These flattened-out vector representation of the 35x35 images had a length of 1225 each and were fed into our neural nets and SVMs.

---

# 3. Neural Network Classifier:

## 3.1. Motivation for Using Neural Nets

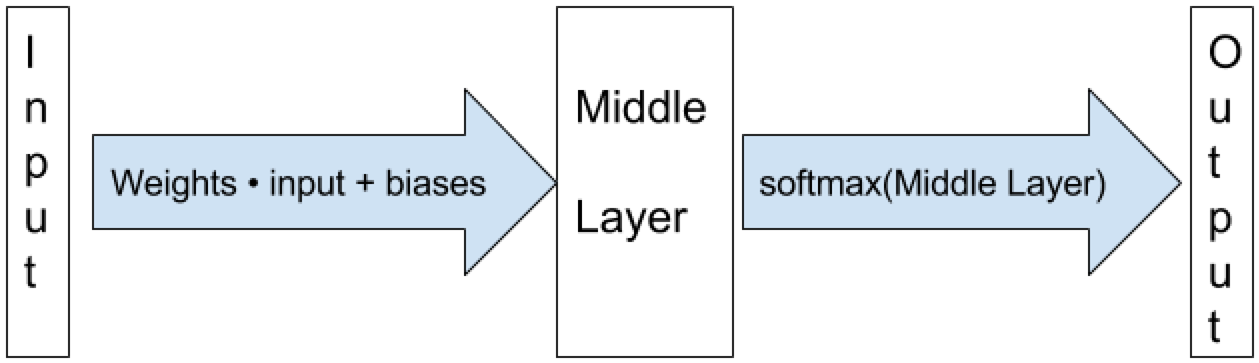
Since this was a multi-label classification problem and we had access to a relatively large dataset, neural networks were our first intuitive guess on how to robustly classify the data. Neural networks have been used very successfully in industry and academia to classify a myriad of datasets. It has especially exploded in growth and popularity in the past few years due to deep learning (neural networks with many hidden layers) and convolution.

## 3.2. TensorFlow

We decided to use Google's TensorFlow machine learning library to implement, train, and run our neural networks. This is because TensorFlow enables us to build data flow graphs to implement complex neural networks in a relatively intuitive way. Furthermore, it is built on top of a fast C++ backend, which allows for highly optimized computation that takes advantage of parallelization and available GPUs when training and running a neural network.

## 3.3. Simple Neural Net

We decided to first build a very simple single-layer neural network (as a proof-of-concept) using Tensorflow. The topology of the network was as follows:



The input to the neural network is a flattened out vector of the 35x35 pixel images, amount to 1125-length vectors. Therefore, the input has a size of 1125.

The middle layer takes the inner product of the inputs with their associated weights and then adds the biases to generate a vector of length 24 (number of symbols we are classifying).

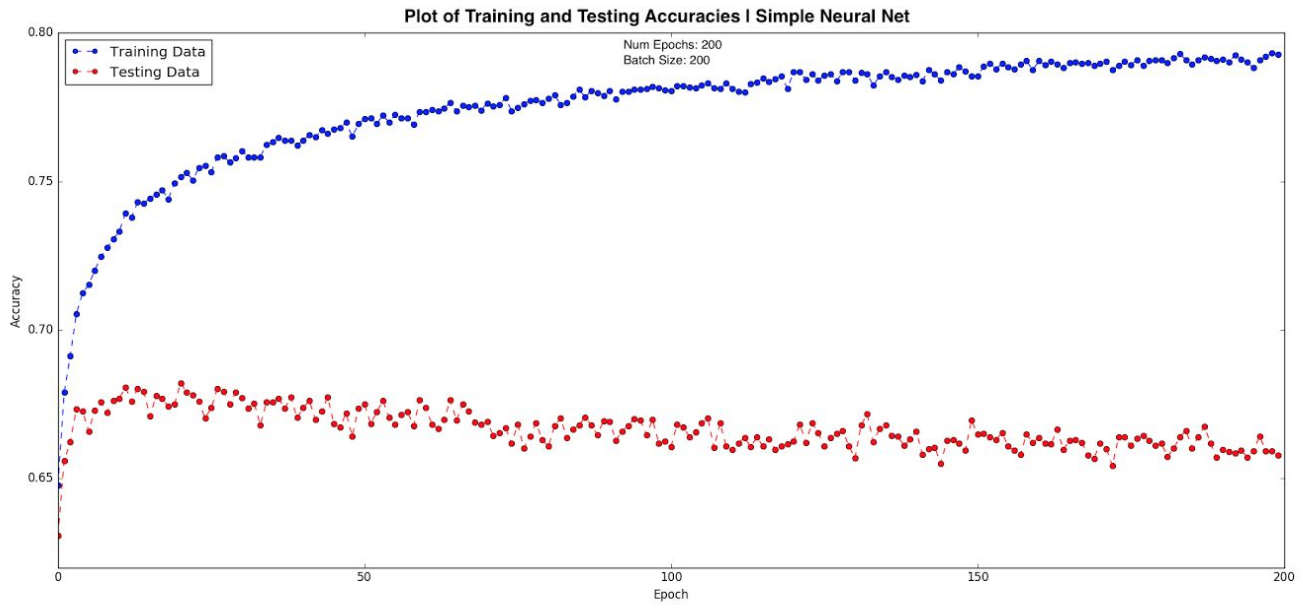
The output then is calculated by simply applying the softmax function to the 24-length vector to convert the values to an array of probabilities (that sum up to one). The interpretation of the network will then be an assignment of probabilities to each possible symbol classification, with the highest probability corresponding to the symbol that the network has the most confidence in.

The loss function we used was the cross-entropy cost function. To train the network, we used stochastic gradient descent with a learning rate of 0.5.

The code behind this network can be viewed in simpleNet.py on the GitHub repository.

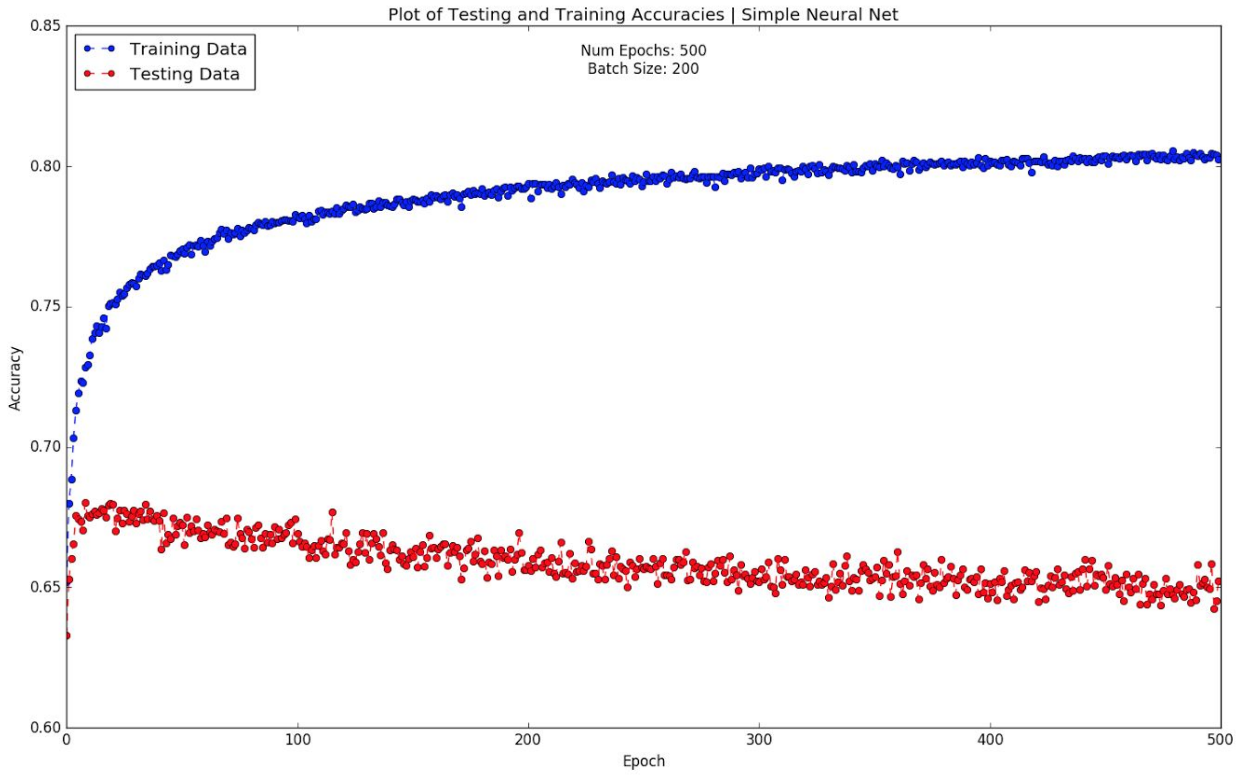
We trained this network on a Macbook Pro Mid 2015 model with a 2.2 GHz Intel Core i7, 16 GB RAM, and an Intel Iris Pro 1536 MB.

We trained the network for 200 epochs using a batch size of 200. We plotted the training and testing accuracies for each epoch:



Both accuracies start off at around 63%. The training accuracies continues to rise. However, the testing accuracy rises slightly then begins a slight downward trend. This is probably indicative that some overfitting is occurring and the neural net is not generalizing well enough.

To confirm these results, we ran the net again for 500 epochs using the same batch size of 200:

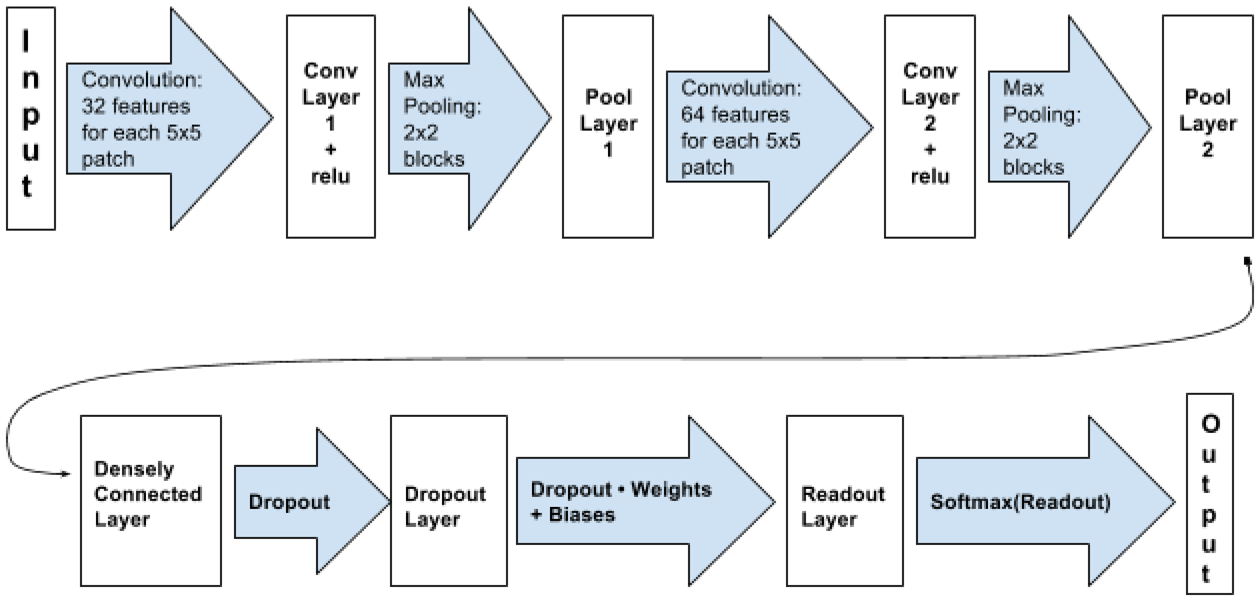


Here the general trend observed in the previous run continues. We can see that the training accuracy continues to rise, reaching 80% accuracy at around the 300th epoch. However, the testing accuracy rises in the beginning from 63% to 77%, then goes on a downward trend eventually reaching 65% at the 500th epoch.

This confirms our previous observation that overfitting is probably occurring, since the testing accuracy goes down while the training accuracy continues to rise.

## 3.4 Multi-layer convolutional neural net

We next embarked on building a multi-layer convolutional neural network. We hoped that this would increase the training/testing accuracies as well as generalize better on the testing set. The topology of the network is as follows:



The input to the neural network is a flattened out vector of the 35x35 pixel images, amount to 1125-length vectors. Therefore, the input has a size of 1125.

The first convolutional layer computes 32 features for each 5x5 patch of the input layer. It then applies the ReLU activation function to each feature.

The first max pooling layer executes a simple 2x2 max pooling of the output from the first convolutional layer, reducing the size of the image to 18x18.

The second convolutional layer computes 64 features for each 5x5 patch of the first max pooling layer. It then applies the ReLU activation function to each feature.

The second max pooling layer executes a simple 2x2 max pooling of the output from the second convolutional layer, reducing the size of the image to 9x9.

The densely connected layer processes the entire output of the second max pooling layer (dot product with weights then adds biases) using 1024 neurons.

The dropout layer keeps some neuron activations and leaves others out, depending on a certain dropout probability parameter. This is intended to reduce overfitting.

The readout layer takes the dropout layer and process it one last time (dot product with weights then adds biases). It produces a 24-length vector of values, one for each possible output (24 outputs).

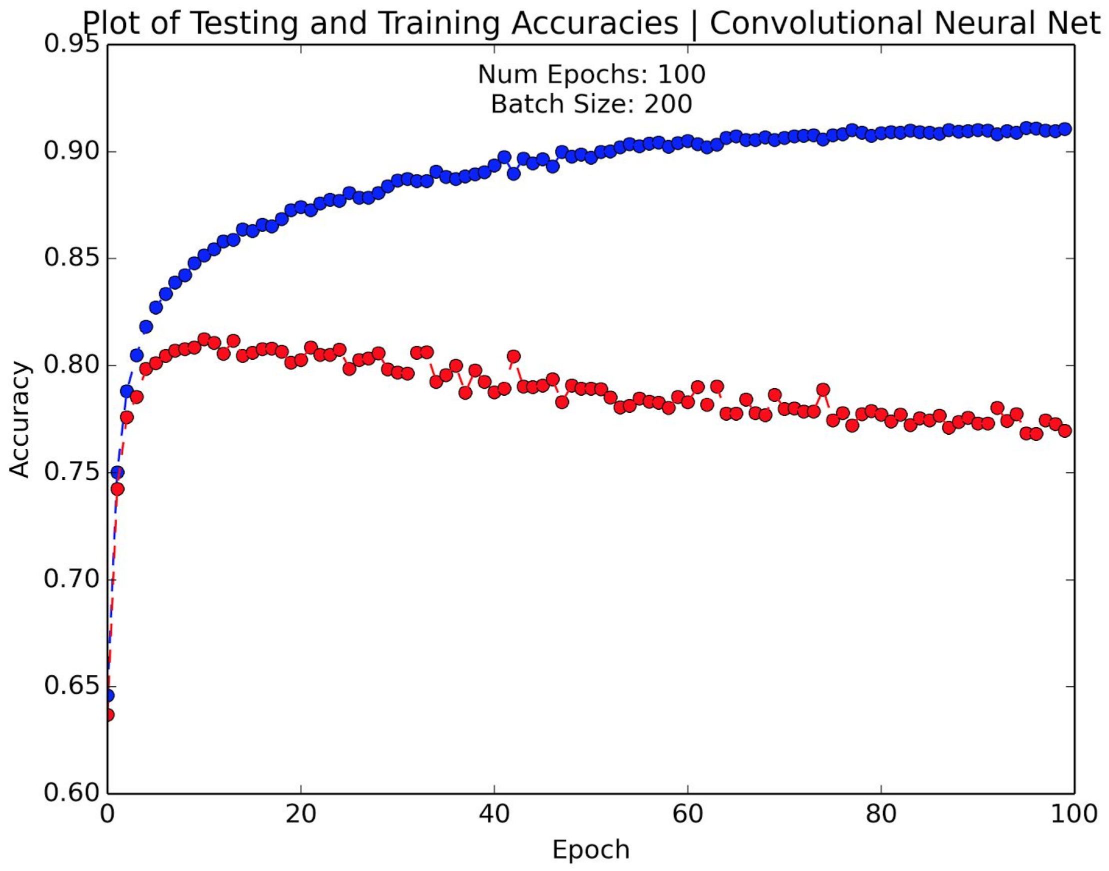
Finally, the output is calculated by simply applying the softmax function to the 24-length vector to convert the values to an array of probabilities (that sum up to one). The interpretation of the network will then be an assignment of probabilities to each possible symbol classification, with the highest probability corresponding to the symbol that the network has the most confidence in.

The loss function we used was the cross-entropy cost function. To train the network, we used the ADAM optimizer (Adaptive Moment Estimation) (Kingma), which is considerably faster than gradient descent.

The code behind this network can be viewed in convNet.py on the GitHub repository.

Initially, we trained this network on a Macbook Pro Mid 2015 model with a 2.2 GHz Intel Core i7, 16GB RAM, and an Intel Iris Pro 1536 MB. However, it took an extremely long time to train, around 13 minutes per epoch. So we were provided with access to a CS Vision GPU server that provided us with much more computing power. On this server, it took an average of 3 minutes and 26 seconds for each epoch.

We trained the network for 100 epochs using a batch size of 200. It took around 5 hours and 45 minutes to complete training using the server. We plotted the training and testing accuracies for each epoch:



Both accuracies start off at around 64%. The training accuracies continues to rise, quickly at first. By the 100th epoch, the training accuracy is at around 91%. However, the testing accuracy rises for the first 10 epochs then begins a slight downward trend, just as it did in the simple neural network. It reaches an accuracy of around 81% at the 10th epoch then continues to decline to 77% accuracy at the 100th epoch. This is probably indicative that some overfitting is occurring and the neural net is not generalizing well enough, just as in the simple net.

## 3.5 Observations

In both the simple net and the conv net, the training accuracies start off at around 63-64%. The training accuracies rise in both nets, but the convNet's training accuracy rises much more quickly. By the 20th epoch, the convNet has a training accuracy of 87% while the simpleNet has a training accuracy of 75%. This illustrates the power of convolution and how it can learn to model complex datasets much better than non-convolutional nets.

In both the simple net and the conv net, the testing accuracies start off at around 63-64%. The testing accuracies rise in both nets for a few epochs then plateau and begin to slowly decline. In both nets, the testing accuracy plateaus at around the 10th epoch, with simpleNet having a max testing accuracy of 67% and convNet having a max testing accuracy of 81%. Thus the convNet's testing accuracy rose much quicker than simpleNet's accuracy

We think that providing the nets with a more robust representation of the data would probably lead to better generalization capability. If we had more time, we would probably have gone with a Histogram of Gradients (HoG) or a Pyramid of Histograms of Gradients (PHoG) descriptor for the input data. This could both increase the accuracies of our models and result in a decrease of the amount of overfitting that we observed.

---

## 4. Support Vector Machine (SVM) Classifier:

### 4.1. Motivation for Using SVM

SVMs are designed to do binary classification to identify data as being part of either a "positive" or a "negative" group. There are ways to adapt an SVM to do multi-class prediction, namely the "one vs one" and "one vs rest" techniques. We choose to evaluate an SVM on our symbol-classification problem to provide a comparison for our MLP.

SVM for Image Classification

| Pros | Cons |
|---|---|
| ❏ Higher expected prediction accuracy/performance<br>❏ Simple Decision Function based on hyperplane<br>❏ Effective in high dimensions<br>❏ Versatile - different kernel functions<br>❏ Memory efficient - only support | ❏ Very slow training time. Complexity determined by number of samples. Quadratic optimization problem.<br>❏ Hard to parallelize<br>❏ Intended for Binary Classification<br>❏ Many parameters, choices for tuning |

|  |  |
|---|---|
| vectors needed for decisions | ❏ No direct probability estimates |

## 4.2. Scikit Learn

SciKit Learn provides the SVC, NuSVC, and LinearSVC classes to perform multiple-class and binary classification on datasets. The implementations are actually handled by “libsvm” and “liblinear” wrapped in C and Cython. SciKit learn provides easy to use methods for each of these classes, and the implementations are fast.

In our experiments, we choose to use both the SVC and LinearSVC classes. The SVC class can fit SVM models with different kernels, and uses a “one-versus-one” approach that is computationally much more intensive, but performs better in certain situations, as we discuss next. The LinearSVC class assumes a linear kernel, and runs much faster since it implements a “one-versus-rest” technique.

For the SVC class, models are created using svm.SVC() and the following parameters can be defined in the constructor:

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

The most important of these is C, which determines the smoothness of the decision surface (the hyperplane) and thereby also the accuracy of the fit. The higher C is, the more complex the decision function and the more features selected for classification.

Once the SVM has been fit to the data, one simply calls the .predict() method on an array of samples. Attributes of interest for analyzing the resulting SVM include: support_vectors_, support_, and n_support which display the support vectors, their indices, and the number of support vectors for each class (in multi-class scenario) respectively.

In addition, the coef_ attribute has shape [n_class * (n_class - 1) / 2, n_features],. Since SVC implements “one-vs-one” this attribute is a matrix of feature coefficients, where each row is a classifier and the number of classifiers is roughly N^2 / 2. Each classifier corresponds to the pair of classes that it was trained to distinguish between.

For the LinearSVC class, the default parameters are as follows:

```
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
```

Attributes of interest include coef_ and intercept_ which have the shape [n_class, n_features] and [n_class]. Since LinearSVC implements “one-vs-rest”, the coef_ attribute is a matrix of feature coefficients, where each row corresponds to one of the classifiers used by the SVM.

## 3.4 Experimental Results

We trained our SVM on the same test and training data that was fed to the MLP. SciKit Learn’s SVM classes take two matrices as input: one containing all samples and one containing all labels. We initially used only 20% of the data in a mini-experiment to test the SVM’s performance. This amounted to ~8,000 training samples and ~2,000 testing samples. The data was randomized so as to provide a broad distribution on which to fit, and then fed into the SciLearn classes. We used the SVM class to perform a “one-vs-one” training using an RBF (radial basis function) kernel which is quite computational intensive but provides good classification. The observed performance was disappointing, however, with little more than 56% correct accuracy on the test samples. Likewise, the “one-versus-rest” LinearSVM class did not fair much better, resulting in about 52% correct prediction accuracy.

**Mini Experiment for OVO SVM with RBF Kernel**

Training Matrix Size: ~8,000 x 1225
Testing Matrix Size: ~2,000 x 1225
**Test Classification Error: 43%**
**Percent Error Rate: 0.439598997494**
**Time to Train: 201.408341169**

number of support vectors per class
[220 366 453 277 373 221 858 505 206 210 202 226 211 652 346 241 215 238
 303 301 210 383 269 313]

## Mini Experiment for OVR SVM with Linear Kernel

Training Matrix Size: ~8,000 x 1225
Testing Matrix Size: ~2,000 x 1225
**Test Classification Error: 48%**
**Time to Train:  12.0392069817**

```
get coefficients, (0,1) (0,2) ... (1,2) (1,3) ...
[[-0.03036866 -0.26578024  0.58125911 ...,  0.          0.          0.        ]
 [-0.62628486 -0.29995263 -0.08946905 ...,  0.          0.          0.        ]
 [-0.23645626  0.10832417  0.04079525 ..., -0.15058886 -0.03499845  0.        ]]
```

We postulate that for the number of classes for our data (24) and the number of samples we provided (8,000) there was simply a dearth of data which caused the SVM to perform so poorly.  The mini-experiment was performed on a Mid-2010 MacBook with 4GB of 1067 MHz DDR3 RAM, and a 2.4 GHz Intel Core 2 Duo processor.  We definitely needed to increase the amount of data we fed into the SVM, however, so we got access to and ran our code on one of Princeton University's high speed clusters. We used 20% of our data (~10,000 samples) to test the SVM that we fit on 80% of the data (~40,000 samples). The results are shown below.

## Large Test Results for OVO SVM with RBF Kernel

Training Matrix Size: 40,000 x 1225
Testing Matrix Size: ~10,000 x 1225
**Test Classification Error: 28%**
**Percent Error Rate:  0.282534589934**
**Time to Train:  1669.48762202**

```
number of support vectors per class
[1084 1713 2106 1373 1779 1007 4041 2529  789  901  986  968  891 3071 1755
 1098 1055 1092 1124 1023 1008 1476  895  988]
```

## Large Test Results for OVR SVM with Linear Kernel
## C parameter set to 0.5

Training Matrix Size: 40,000 x 1225
Testing Matrix Size: ~10,000 x 1225
**Test Classification Error: 34.2%**
**Percent Error Rate:  0.34228995388**
**Time to Train:  107.855374098**

```
get coefficients, (0,1) (0,2) ... (1,2) (1,3) ...
[[ -1.17901485e-02   1.20478720e-01   9.09936590e-02 ...,   3.48993785e-01
    8.10969905e-02   0.00000000e+00]
 [ -5.83776528e-01  -6.05157874e-02  -2.19437138e-01 ...,   3.50532734e-01
    2.18828557e-01   1.71329417e-01]
 [ -7.02000844e-02   1.08238125e-01  -7.51824305e-02 ...,  -5.97426256e-02
   -6.19148876e-01   8.00054637e-01]]
```

**Large Test Results for OVO SVM with Linear Kernel**

Training Matrix Size: 40,000 x 1225
Testing Matrix Size: ~10,000 x 1225
**Test Classification Error: 32%**
**Percent Error Rate:  0.324243031883**
**Time to Train:  827.121375799**

```
number of support vectors per class
[ 820 1146 1776 1324 1411  692 3537 2492  399  593  727  557  713 2902 1719
  810  815  871  771  556  599  929  515  602]
```

**Large Test Results for OVR SVM with Linear Kernel**
**C parameter set to 1.0**

Training Matrix Size: 40,000 x 1225
Testing Matrix Size: ~10,000 x 1225
**Test Classification Error: 34.7%**
**Time to Train:  121.146058083**
**Percent Error Rate:  0.347603769801**

```
get coefficients, (0,1) (0,2) ... (1,2) (1,3) ...
```

```
[[ -1.31977044e-02   1.23217313e-01   9.28289070e-02 ...,   4.55189706e-01
    9.38298895e-02   2.16840434e-19]
 [ -5.97185359e-01  -6.22689607e-02  -2.25200667e-01 ...,   6.41674553e-01
    1.65939961e-01   1.36068928e-01]
 [ -7.11524243e-02   1.09123974e-01  -7.63455912e-02 ...,   2.42569578e-02
   -8.78063493e-01   1.02653827e+00]
```

The prediction accuracy for most of the models we fit was around 66%, which was a reasonable improvement but also suggestive of a lack of adequate data, especially since we expected the SVM to outperform our MLP significantly.

The best result we had was a 70% prediction accuracy on the test set for an "one-vs-one" SVM with RBF kernel.  As we mentioned previously, the RBF kernel is the most computationally intensive choice, but yields the best results.  Training time took 1669.5 seconds, or about 30 minutes, in comparison to OVO SVM run with a Linear Kernel which only took 14 minutes (roughly half the time), although at a cost of about 4% test prediction accuracy.

## Bibliography

Kingma and Ba,  ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION
arXiv:1412.6980v8 [cs.LG] 23 Jul 2015

Jimenez, Nicolas D., and Lan Nguyen. "Recognition of Handwritten Mathematical Symbols with PHOG Features." (n.d.): n. pag. Recognition of Handwritten Mathematical Symbols with PHOG Features. Stanford University, Fall 2013. Web. 17 Jan. 2017. <http://cs229.stanford.edu/proj2013/JimenezNguyen_MathFormulas_final_paper.pdf>.

Matsakis, Nicholas E. "Recognition of Handwritten Mathematical Expressions." (1999,): n. pag. Recognition of Handwritten Mathematical Expressions. MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 21 May 1999. Web. 17 Jan. 2017. <http://www.ai.mit.edu/projects/natural-log/papers/matsakis-MEng-99.pdf>.

Thoma, On-line Recognition of Handwritten Mathematical Symbols
arXiv:1511.09030v1 [cs.CV] 29 Nov 2015

Jonathan Milgram, Mohamed Cheriet, Robert Sabourin. ”One Against One” or ”One Against All”: Which One is Better for Handwriting Recognition with SVMs?. Guy Lorette. Tenth International Workshop on Frontiers in Handwriting Recognition, Oct 2006, La Baule (France), Suvisoft, 2006.

---

## Other Resources

HWRT database of handwritten symbols:
http://www.martin-thoma.de/write-math/data/

Competition on Recognition of Online Handwritten Mathematical Expressions:
http://www.isical.ac.in/~crohme/

TensorFlow Tutorial on Convolutional Neural Networks:
https://www.tensorflow.org/tutorials/mnist/pros/

SciKit Learn SVM Tutorial:
http://scikit-learn.org/stable/modules/svm.html