From Pixels to Sharpies: Evaluating MNIST Models Against Real Handwritten Digits

Parker Christenson, Philip Felizarta, Gabriel Emanuel Colón

University of San Diego

AAI-501: Introduction to Artificial Intelligence

Prof. A. Van Benschoten

December 11th, 2023

Table of Contents

From Pixels to Sharpies: Evaluating MNIST Models Against Real Handwritten Digits

Introduction

The MNIST dataset is a widely used benchmark dataset in the field of machine learning. It consists of a collection of 70,000 handwritten digits, each represented as a 28x28 grayscale image. This dataset is often used for training and testing various machine learning algorithms, especially in the domain of image classification. One of the key reasons why the MNIST dataset is considered good for computer vision is its simplicity and generic nature. The dataset contains a diverse set of digits ranging from 0 to 9, with different writing styles and variations in shapes. This diversity, in theory, allows machine learning models to learn and generalize well to unseen handwritten digits. With that being said, the data set can be loaded from multiple machine learning libraries. The data set is already pre-labeled which, in our case, ensures that each image is associated with the correct digit label. Thus, data labeling for the training set was not a required task for this project.

The MNIST dataset's low resource cost and availability position it to be a standard benchmark for quickly evaluating a machine learning algorithm's performance. It has played a significant role in the development and advancement of various machine learning techniques, such as deep learning, convolutional neural networks, and feature extraction methods.

Exploratory Data Analysis

Our Exploratory data analysis was simple and dry, due to the nature of not having to clean nor pre-prepare the MNIST data set. Our group presupposed that computer vision models would experience difficulty classifying similarly appearing numbers. The variation and styles found in human handwriting will not translate to the consistency of a predefined font like Times

New Roman or Consolas. During the EDA section, we wanted to make sure that some of the assumptions we had were correct, and get a general feel for how the numbers and the pixels overlapped with each other, along with getting a feel for some of the outputs.

Our first objective was to get the average shape for every single number. Given how many numbers and unique handwriting styles there are, we thought that it would be interesting to see how or what the average looks like for the digits in the data set. ***(See EDA Output A1)*** Looking at the nature of all of the average digits, we can see numbers like "1", '4','5', and '7' can be characterized by sharp edges whereas other digits may be more rounded in nature. We theorized that due to the handwriting style being unique to every person, that there would be some sort of 'rounding' of the numbers due to the fact that we are taking the mean of every pixel.

The next output we got from the MNIST dataset is the unique counts of all the digits and the subsequent distribution characterizing them. We opted to use a histogram to describe the distribution of digit counts since this method excels at presenting categorical data and their frequency. With that being said, we are able to see that the number '1' showed up in the MNIST data set the most frequently (***See EDA Output A2***). The plot was able to give us a little bit of information on the distribution of the data, but not really give us too much information that we could take action on.

After not gaining any real insight from the preliminary descriptive statistics with regard to the difficulty classifying the handwritten digits, we decided to plot the data into a U-map. U-map is short for Uniform Manifold Approximation and Projection, it serves to present a digestible view of data clustering by projecting the data onto a lower dimensional space. It may reveal some struggles that the Machine learning model might have during training by exposing large overlap between classes in latent space (McInnes et al., 2018). Looking at our U-map plots

(***See EDA Output 3 and EDA Output 4***) we are able to get an idea that the model will have a very hard time being able to decipher between the digit four and the digit nine. The similarities shared between four and nine could be from the looping structure at the top. For the most part the digits are very well spaced away from each other, though every now and then, there are some outliers that present themselves in clusters of other classes. The three dimensional Plotly U-map enabled us to zoom into some of the clusters, and we are able to see some of the example data anomalously appearing in distant class clusters. Our group theorizes that this is just from poor handwriting.

Exploratory Data Analysis has given us reasonable expectations of how an arbitrary computer vision model may perform on training examples in the MNIST dataset. Our group will proceed by designing and implementing a variety of models trained on the MNIST dataset to explore model performance on our own handwritten digits. We intend to explore the generalizability of more complicated deep learning architectures, like multilayer perceptrons and convolutional neural networks, and contrast them with more traditional models, like K-Nearest Neighbors and SVM.

Model Selection

When selecting the models we decided to take an approach that is two fold: challenging and complex. We wanted to see which of the models are able to predict the MNIST data while maintaining performance on custom handwritten digits. Our model selection therefore is limited to classification models, and algorithms to predict based off of the twenty-eight by twenty-eight numpy array. Our models should be optimized to then also be able to predict the handwritten digits we will test. The data set could be used to recognize a number on a piece of paper that was

handwritten on an application for example, before putting it into an electronically stored record system.

Our first model we selected was the K-Nearest Neighbors model from the Scikit-Learn library as we felt the model would be very good for not needing an extensive amount of prior knowledge to build, while serving as a high-performance beginner model (Ai, 2023). We intend to use this model as a baseline to compare our more advanced models to. Using Scikit-Learn's documentation, we are able to build a fully functional model and start testing fairly quickly. After completing the model, we plan to bring the trained accurate model down to a .joblib file to be able to test our own handwritten digits on. The .joblib files presented a constraint because they were massive in size. Our group does recognize that this is not specifically great for a production model, we feel that since it's only being hosted on our machines, it is sufficient for our purposes.

Our next model we opted to test was also available from Scikit-Learn as well, we opted for the Support vector machine classifier(*API Reference*, n.d.).We decided to utilize the SVM model as we felt that the model would be good at being able to handle medium to small sized data sets that should, in theory, be relatively easy to train due to Scikit-Learn's documentation. An added benefit of choosing the SVM model from Scikit-Learn was the overall speed in which it takes to train the model for deployment. We do recognize some of the drawbacks of the SVM model architecture such as, having feature engineering issues, as it is an extra step that you would have to perform when deploying the model in a real world situation (Hamilton, 2020). We also concluded that using the SVM model will be quick to code, and wise to utilize on the computation side. Since our model is using sixty-thousand training images, we understood that the computational task would not take long, and only took us five minutes from the start of

training to the end. Since the Scikit-Learn library does not allow for GPU acceleration, all of the training is accomplished within the CPU. That means, depending on the core count of the CPU, there will be a varying time that it takes for the initial model training.

The last set of model architectures our group decided we should evaluate were neural networks, as we wanted to test the complexity of the model architectures and observe the effects on the accuracy in terms of depth, and layer width selection. Our group decided on implementing small changes to network architecture to study the impacts on model accuracy on MNIST test data and custom handwritten digits. We ask, does the neuron selection in conjunction with the depth, also affect the true accuracy of the model prediction? The neural network models themselves will be compared to each other to measure the effectiveness of the accuracy based on the changes made. We want to include all of the drawbacks of building all of the models, and some of their issues when applying them in some real world examples.

Model Analysis

After developing our models we decided we needed a clear method to evaluate the accuracy and the way that the models functioned. For this basis we opted to use the Scikit-Learn classification report. We wanted to make sure that the models were able to have a report that we can compare and contrast to see the weaknesses and the benefits of each model. Our group feels that model comparison is a very important part when deciding which algorithm has the highest accuracy. Scikit-Learn allows for the easy implementations of key parameters when it comes to the comparisons. From this library we chose the K-Nearest Neighbors (KNN) and the Support Vector Machine (SVM) models. We knew beforehand that the KNN model was not the most optimal choice for image classification, and that it would take up a lot of memory when executing, but its simplicity in implementation provided us an easy choice as an additional model

for comparison (Hamilton, 2020). The SVM model is a more powerful model than KNN, and we thought it to be a good choice because of its capability for generalization, which would help avoid overfitting.

K-Nearest Neighbor Analysis

The Scikit-Learn accuracy report provides an insightful glimpse into the incredible effectiveness of the K-Nearest Neighbors (KNN) model across ten diverse classes. With an impressive overall accuracy of approximately 97.13%, it's clear that this model excels at correctly identifying various classes. When looking deeper into the individual class performance, the precision, recall, and F1-score values show an exceptional model proficiency. The precision, which measures the accuracy of positive predictions, is exceptional across all classes. Notably, the numbers 0, 5, 6, and 8 achieved precision rate at 98%, showcasing the model's great precision when predicting these classes.The KNN model's ability to find all actual positives, as evaluated by recall, is equally noteworthy ***(See Figure B1)***. Classes 1, 6, and 7 exhibit recall rates at 98%, indicating that the model has a strong capability in identifying instances of these particular classes. The F1-score, which is a balanced measure of precision and recall, consistently demonstrates outstanding performance across all classes. Classes 0, 1, 6, and 7 have the best F1-scores, approximately 98%, which showcases the model's ability to identify these specific numbers within the testing set. It is worth mentioning that while the lowest precision was class 9 at 96%, and the lowest recall in class 8 at 94%, these values still indicate only minor comparative weaknesses in the model's performance for these two classes. To summarize, the KNN model showcases great performance with its high accuracy, precision, recall, and F1-scores across all classes. Its consistency across these key metrics establishes it as a reliable and dependable model for effectively classifying input data to the ten distinct classes within the

MNIST data set.

Support Vector Machine Analysis

The Support Vector Machine took 5 minutes to train which we do understand will vary due to the CPU in the machine that is building the model. The model only took 10 lines of code, excluding comment lines, and print lines. The SVM model could serve as a fast implementation, for those who prioritize getting a model up and running as soon as possible.

The Support Vector Machine model has achieved an impressive overall accuracy of 96.3%, demonstrating its proficiency in accurately classifying the images across the number categories. The precision scores for all classes are high, especially for classes 0 and 1, with precision values of 99% and 98%. ***(See Figure B2)*** These high precision scores indicate the model's accuracy in making positive predictions. Additionally, the recall scores are commendable, with class 7 achieving the highest recall at 97%, showcasing the model's ability to effectively identify true instances of each class. The F1-scores, which strike a balance between precision and recall, remain consistently high across all classes, highlighting the model's robust and well-rounded performance. Notably, classes 0, 1, and 6 exhibit F1-scores exceeding 97%, underscoring the model's particular strength in identifying these classes ***(See Figure B2)***. The model maintains this high level of performance across classes with varying frequencies, as indicated by the support numbers. In summary, the SVM model demonstrates great classification capabilities, shown by its high accuracy, precision, recall, and F1-scores. This indicates its suitability for diverse classification tasks and its ability to maintain a balanced performance across different classes.

Neural Network Performance and Analysis

Our group decided on testing 5 different neural network models with 2 sets of architectures. We intended to measure the way that the model's architecture affects the accuracy score. This project did not necessarily care to measure training times, instead the study keeps training epochs constant regardless of architecture and network size. In our analysis of the neural networks, we trained every single neural network the same way over 100 epochs with a learning rate of 0.003 and L2 normalization constant of 0.00001. Each model utilized the exact same training and test splits to control for any variation in data quality. All neural network training histories can be found on **(*Figure B3)*** in the Appendix.

MLP Analysis Section

The two MLPs trained in this study share a few properties. Since MNIST data consists of 28x28 pixel images, our networks take a flat 784-dimensional vector as input. Each network has an initial hidden layer with 64 nodes. This initial hidden layer contributes 50,240 learnable weights to the model. Both networks have a second hidden layer, one network has only 16 nodes and another has 32 nodes. The size of the second hidden layer is the only difference between the two MLPs used in this study, as both networks utilize Rectified Linear Units (or ReLU) as hidden activations and a final softmax layer of 10 nodes for classification. Each network was trained for 100 epochs with a batch size of 32 on the same training data. The study is limited to just two multilayer perceptrons since we implemented another set of convolutional neural network models. We plan to analyze how MLPs, despite their larger parameter counts, perform compared against a more sophisticated model that is tailored to a computer vision task.

64-16 MLP

Initially, the training loss stood at 0.7889, steadily decreasing to a mere 0.0136 by the final epoch. This consistent decline in loss clearly indicates the model's effective learning from the training data. Concurrently, the training accuracy exhibited a substantial increase from 76.24% to 99.90%, highlighting the model's growing proficiency in accurately classifying the training data. Something equally significant to note are the validation metrics, which assess the model's ability to generalize to new and unseen data. Throughout the training process, the validation loss decreased from 0.3132 in the first epoch to 0.1022 in the 100th epoch, mirroring the trend observed in training loss. The validation accuracy began at a high 91.58% and progressively climbed to 97.75%. Despite a slight disparity between training and validation accuracies, the high validation accuracy demonstrates the model's strong generalization capabilities for the MNIST data set. Separate from the validation set, was a larger test set used to more rigorously assess model performance on unseen data. The model scored 97.12% accuracy and 0.1058853 cross entropy loss on the test set.

64-32 MLP

Notably, the slightly larger model consistently improved its performance as the epochs progressed. This improvement is evident from the decreasing loss and increasing accuracy observed on both the training and validation datasets. These results indicate that the model learned effectively and adapted well to the training data. After completing the training phase, the model underwent further evaluation using a separate test set. The evaluation showed an impressive categorical cross-entropy loss of approximately 0.1072596 and an accuracy rate of about 97.21%.

MLP Summary and Conclusion

Both models have achieved remarkable accuracy and made significant improvements in reducing loss. However, the larger model stands out with its slightly better performance indicated by its test loss and accuracy. Ultimately, deciding between the two models hinges on the specific needs of the application and the delicate balance between training accuracy, generalization capability, and performance on unseen test data.

Convolutional Neural Network Analysis

The second architecture our group settled on was a simple Convolutional Neural Network (or CNN). CNNs are often used in computer vision tasks because they utilize 2-dimensional learnable kernels. These convolutional kernels act as a sliding window over 2D or 3D data, thus these kernel parameters are specifically suited to 2D images in MNIST that contain positional information. Since the convolution operation is differentiable and convolutional kernels don't have to be very large, convolutional layers can be quickly trained via gradient descent. Though, this process is still slower to train than a standard MLP (LeCun, 2015).

The three CNNs used in our study all contain a base 32 filter convolutional layer with kernel sizes of 3x3. Note that the convolution operation (when no padding is used like in this study) outputs an image of smaller width and height. So, for the first convolution operation, a single kernel converts the 28x28 MNIST images into 26x26 convolved images. Since there are 32 kernels (commonly called filters), the result of the first layer of all of our CNNs is a 26x26x32 tensor. The first model settles for using this and passes the tensor to a Max Pooling layer, but the other two models (32-16f and 32-16x3f) pass the 26x26x32 tensor through 1 and 3 convolutional layers with 16 3x3 kernels, respectively. Since the 32-16f utilizes only 1 more convolutional layer, the resulting tensor is 24x24x16. 32-16x3f uses 3 convolutional layers, thus

the resulting tensor is 20x20x16. Note the diminishing width and height of tensors is usually combatted with some form of padding in deeper networks, but for the sake of simplicity our group settled on no padding.

Each network now takes the convolved tensor produced by the initial layers and passes them through a Max Pooling Layer. A Max Pooling layer is similar to a convolutional layer, but utilizes Max kernels instead of learnable ones. In this study, we used 2x2 max kernels. The max kernels slide over the convolved tensor and return the maximum value in a 2x2 square. This effectively reduces the convolved tensor's height and width by half, but keeps the third dimension intact (Boureau, Y.-L., 2010).

Finally each network flattens the final tensor into a single vector that is to be fed into a Logistic Regression classifier with Softmax activation over the 10 classes (similar to the final layer of our MLPs). All convolutional operations are followed by a ReLU activation, the same hidden activations used in the MLPs of this study. The CNNs are also trained for 100 epochs with a batch size of 32.

Though a CNN may seem to have a large number of complicated operations, the network architecture allows for many calculations to be done in parallel. Also notice how the Convolutional Layers and Max Pooling Layer in the CNNs produce a tensor that is flattened into what is essentially just a feature vector for a logistic regression. The feature vectors produced by CNNs in this study are only subjected to logistic regression, but the architecture allows for a deeper MLP. For the sake of simplicity, our group decided on the simpler logistic classifier as the final head of all the CNNs.

32F Convolutional Neural Network

The training process started with a loss of 0.5154 and an accuracy of 86.00%. As training progressed, the model steadily improved, evidenced by a decrease in loss and an increase in accuracy, both on training and validation datasets. By the 100th epoch, the training accuracy reached an impressive accuracy of 98.54%, with a corresponding loss of just 0.0204. The validation accuracy also peaked at 98.48% with a loss of 0.0567, indicating that the model not only learns well but also generalizes effectively to new data. The final test on a separate dataset yields a loss of 0.0545 and an accuracy of 98.37%, confirming the model's robust performance. Since this is our first model with only one layer, this will technically be our baseline for this model architecture. We are only going to continue expanding our model depth by adding more convolutional layers.

32-16F Convolutional Neural Network

In comparison to the first model the second model does make great improvements in accuracy. Our second model had an accuracy of 98.48% and a test loss of 0.0658. Despite half of the model parameters, the extra layer netted a strong performance. Circling back to the training data, we ended with a very high accuracy at 99.91% which shows that by adding the layers, we were able to in fact have the model become more accurate on the training data. In theory, these models should have stronger performances on new data.

32-16F x 3 Convolutional Neural Network

This brings us to our last and deepest CNN model that we will be evaluating. Our results for the final CNN were shocking, with an extremely high accuracy of 98.66% on the testing set. The Cross Entropy loss of our model was 0.0855 which is higher than the other two models. This

may highlight the problems with optimizing on loss instead of accuracy. Moreover, this last model had the smallest feature vector and smallest number of parameters. One may conclude that this model finds higher quality features and parameters.

Neural Network Conclusion

The following section will refer to ***Figure B4*** found in the appendix. The simplest 32f CNN has the largest number of parameters out of any model. This is because the network has the largest output tensor and hence feature vector. The large feature vector causes the logistic regression to have a tremendous amount of parameters. The 32f CNN scores the best test loss, but not the best test accuracy. This highlights the subtle differences between test accuracy and test loss, remember we don't optimize for accuracy because it is not differentiable. Loss is just a heuristic for accuracy.

Despite their smaller parameter counts, the two CNNs with smaller feature vectors have better accuracy on the test set. In fact, the CNN with the smallest feature vector output has the best performance. Now, with such great performance on MNIST, one would expect these models to perform relatively well on real-world data.

Testing Our Own Numbers

With our models being complete, we decided to test our own numbers. To do this we deployed two methods. First, we drew our own numbers with a Sharpie on pieces of paper, and then converted them into numpy arrays to bring them into our models. The second method we deployed was in a Jupyter notebook using HTML in which we wanted to be able to draw the number in the HTML drawing box and then bring them into the model after image processing is done using an HTML link to save the image.

Our first method was a little bit more complicated to get the digits to match the MNIST data set. We had to first draw our digits zero through nine on a piece of paper and then process the image after saving them as a PNG. We then had to perform some processing on the digits in order to get the data to match the profile of MNIST digits, which we ended up realizing is flawed. In order to get our digits to match the MNIST data set, we had to replicate the formatting. The MNIST data set is a white number with a black background, which requires our handwritten digit images to be inverted. After that process is complete, we scale the amount of pixels down from the original captured resolution to 28x28. This should finalize the formatting of our hand drawn digits and get us close to matching the MNIST data set. The MNIST data set is very specific, and the numbers are formatted in a weird manner with some kind of blur or issue with the capturing lens (***See Figure C2***). After further analysis we saw that the images are very difficult to have the models read. Our SVM model for example was predicting class 7 for almost every number that we hand drew and transformed. After we converted numbers into a Numpy array and fed it to the model, we put the hand drawn images into a SeaBorn heatmap, to get a rough idea on what the model is seeing (***See Figure C1***). After verifying the number is properly handled, we were able to draw two conclusions: the data used to train the numbers is very hard to have near identical formatting to, and the models we developed will only produce accurate predictions if we are able to get the data to look exactly like MNIST data. This conclusion is backed by the fact that the highest performing model on the handwritten digits only scored 50% accuracy (***See Figure B5***). Even after attempting to produce handwritten digits using the alternate HTML method, all the models failed to perform as promised by their MNIST results.

Summary and Closing Thoughts

Our group theorizes that the models must be struggling with the custom data due to the differences in digit sizes, digit orientations, and brushes compared to the original MNIST data. Recall how the original MNIST data used a softer brush with more gradients and had larger digits that were almost always upright. Our data is smaller, the brush is more harsh, and orientations are variable.

The availability of pre-labeled data eliminates the need for manual labeling, allowing researchers to focus on model development and refinement. The exploratory data analysis provided insights into the dataset, revealing average shapes of handwritten digits and the distribution of numbers. The analysis also highlighted potential challenges for machine learning models, such as distinguishing between similar digits (e.g., 3 and 9) and dealing with outliers caused by poor handwriting. These findings lay the foundation for further analysis and the development of machine learning models that can accurately classify handwritten digits. By leveraging the advancements in machine learning, convolutional neural networks, and feature extraction methods, the MNIST dataset will continue to contribute to the progress of computer vision algorithms.

Though every model trained in our study achieved high accuracy on MNIST test data, all models failed to generalize to the custom dataset. The custom digits already had many preprocessing measures to ensure the data resembled the profile of MNIST digits. Our team made measures to convert PNGs to grayscale and utilize the original scalar normalizations from the training data to give the models a fighting chance!

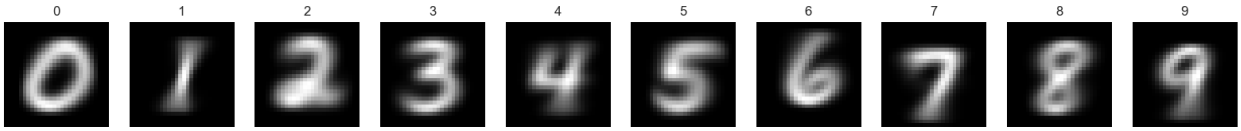
Though our study could have included a more thorough set of models and regularization techniques to ensure better generalization, it is easy to conclude that MNIST test accuracy does not reflect the generalizability of a model architecture, even for a very similar task.

Disregarding the accuracy scores and all model types, our group believes that the MNIST data set is great for studying and learning, but the data set is only good for just that. The requirement of transforming handwritten digits into the same, almost random formatting of MNIST data presents itself as a monumental challenge for applying models to the real world. The ideal handwritten digit dataset should include a far larger variety of formats, brush size, and rotations of the digits.

Our team concludes that MNIST should be seen as the bare minimum preliminary test for a more serious study of a model's performance. If the model can't learn MNIST, don't bother with more complicated data. But 98% accuracy on MNIST doesn't mean very much either!

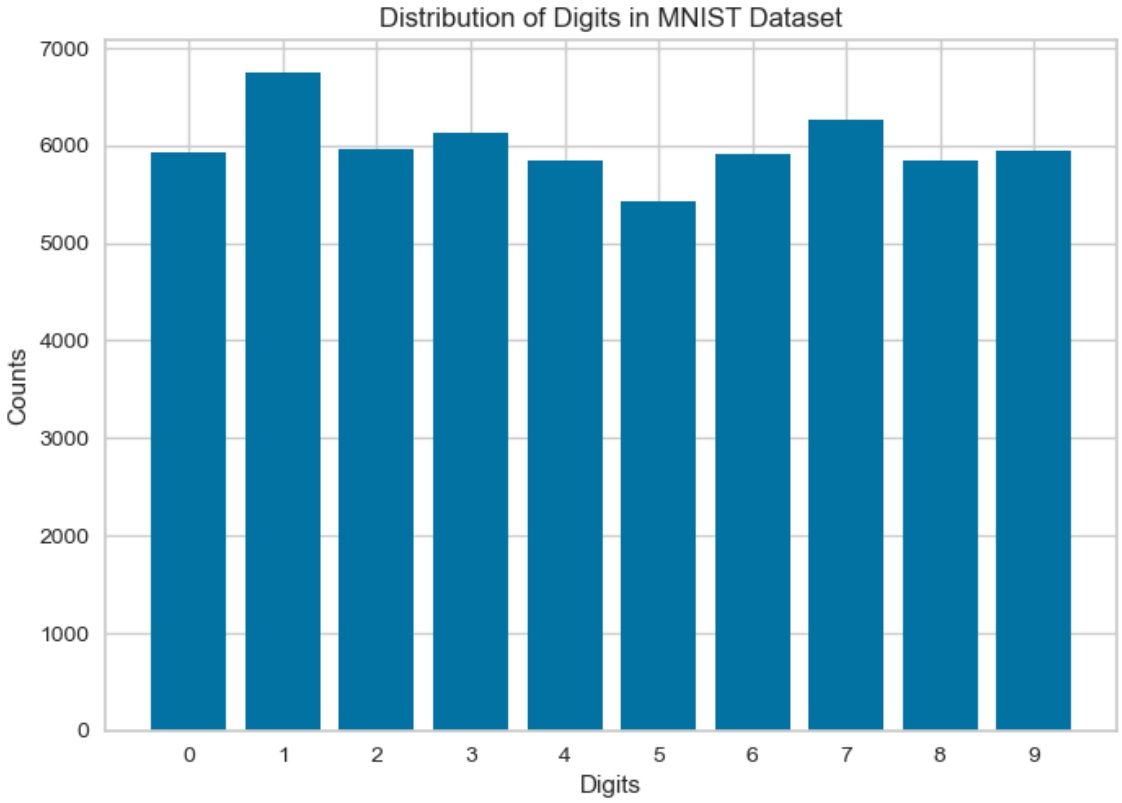
Appendix

EDA Output A1



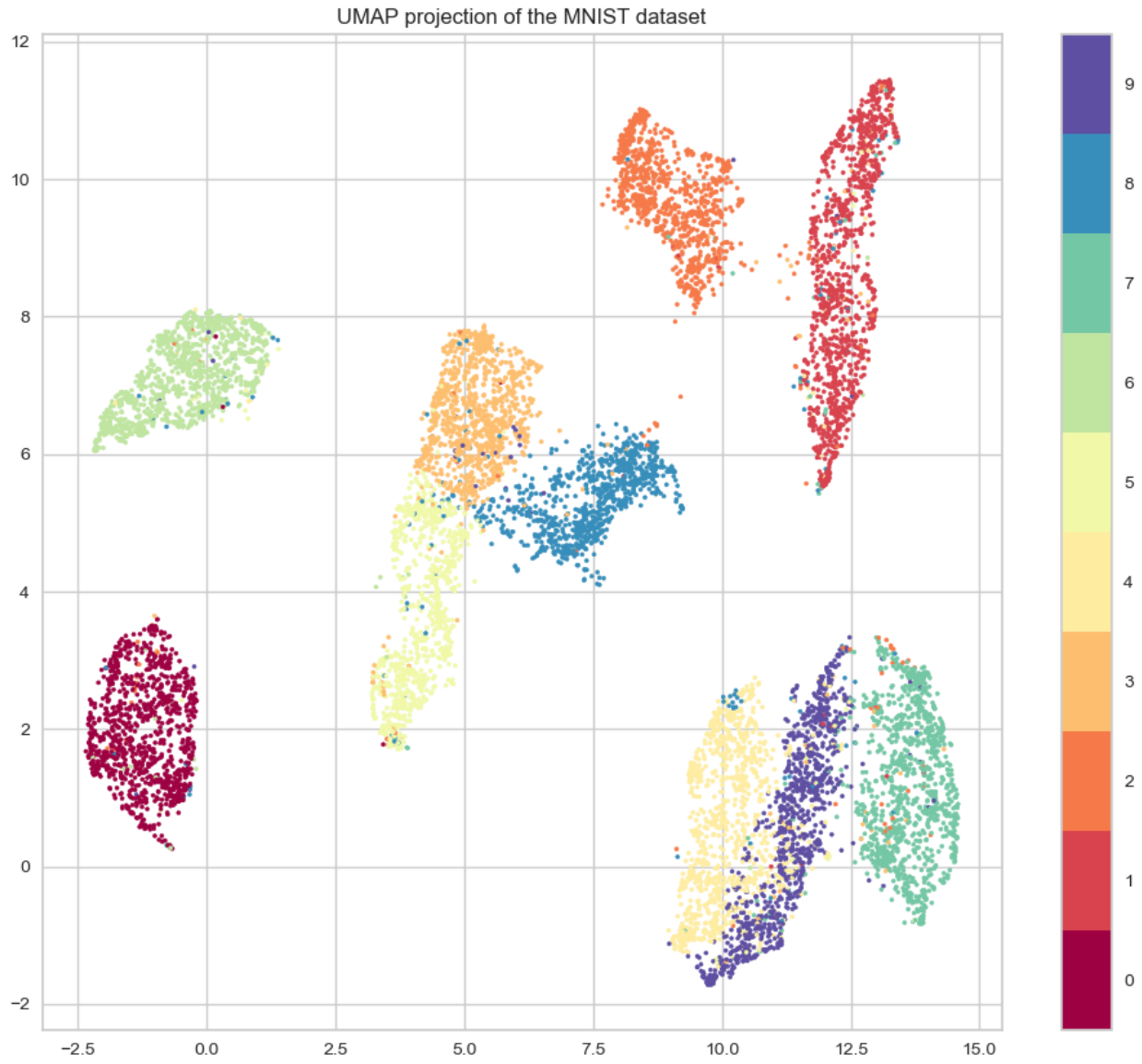
Average Shape of Every single Digit within the Data Set, this was done using the mean of the pixels across all examples, in each of the classes. This gives us the 'Average' shape of each of the images in the classes.

EDA Output A2



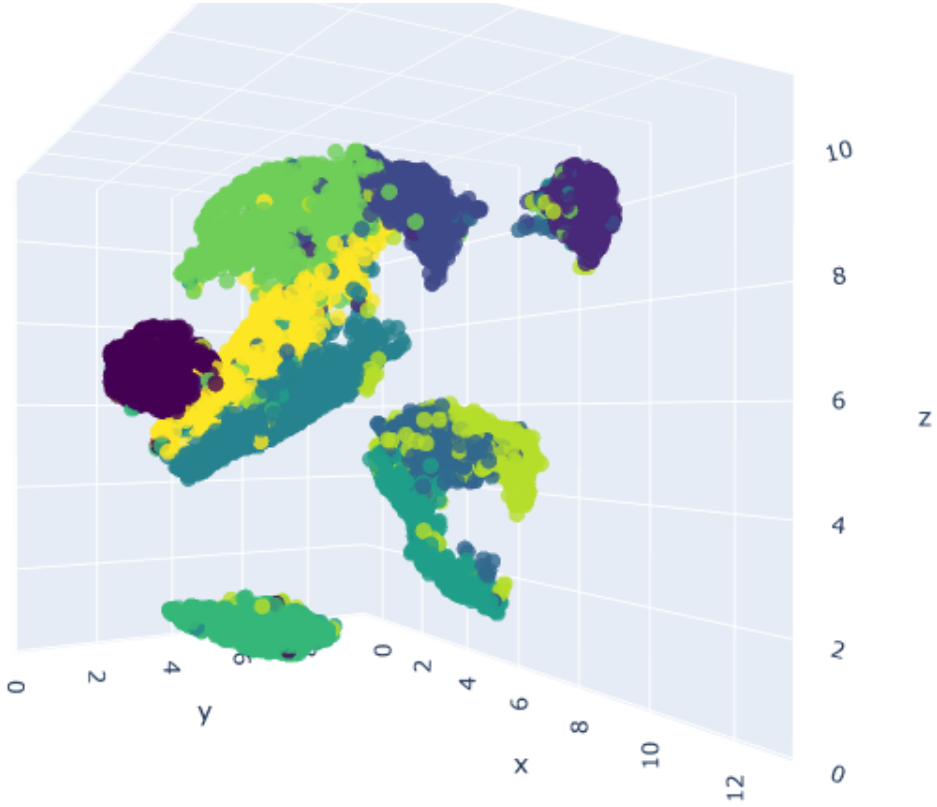
This is a Histogram of the numbers in the MNIST data set to show their frequencies. This is done so we can get a better idea of the training data that could give insight to potential biases the model may have.

EDA Output A3



Two Dimensional U-Map of the Data set, used to initially determine what numbers in the model will cause problems by evaluating overlap. Observe the overlap between classes '4' and '9.' Also notice the random disbursement of outliers within each of the classes. These factors may prevent any model from reaching 100% accuracy on the training set.

EDA Output A4



Three Dimensional U-map of the Data set, made using Plotly, which allowed for us to be able to examine some of the more complex similarities between the numbers within the dataset.

Figure B1

Accuracy: 0.9712857142857143

	precision	recall	f1-score	support
0	0.98	0.99	0.98	1343
1	0.96	0.99	0.98	1600
2	0.97	0.97	0.97	1380
3	0.97	0.96	0.97	1433
4	0.97	0.96	0.97	1295
5	0.98	0.97	0.97	1273
6	0.98	0.99	0.99	1396
7	0.97	0.98	0.97	1503
8	0.99	0.94	0.96	1357
9	0.96	0.95	0.96	1420
accuracy			0.97	14000
macro avg	0.97	0.97	0.97	14000
weighted avg	0.97	0.97	0.97	14000

This is the print out from the JuPyter notebook which shows the KNN's report, which shows precision, recall, F-1 score, and the Support. The Report also shows the Accuracy, macro average and weighted average.

Figure B2

Accuracy: 0.963

	precision	recall	f1-score	support
0	0.99	0.98	0.98	1343
1	0.98	0.99	0.98	1600
2	0.95	0.96	0.95	1380
3	0.96	0.95	0.96	1433
4	0.96	0.96	0.96	1295
5	0.97	0.96	0.96	1273
6	0.97	0.98	0.97	1396
7	0.92	0.97	0.95	1503
8	0.97	0.95	0.96	1357
9	0.96	0.93	0.95	1420
accuracy			0.96	14000
macro avg	0.96	0.96	0.96	14000
weighted avg	0.96	0.96	0.96	14000

This is the SVM models Classification report from Sci-Kit Learn. Identical to the KNN report in measures.

Figure B3

This set of graphs represents the training and accuracy histories of all neural networks trained in this study. The yellow lines always represent validation sets. Validation sets are sets that are not trained on, but are smaller than test sets. We use validation sets to get an idea of how well training is going, without optimizing on the test set! Notice how each neural network performs very well on the training data and also has very high performance on the validation sets. Traditionally, this would indicate great generalization. However, we'll see with the custom data, that the networks do not perform so well.

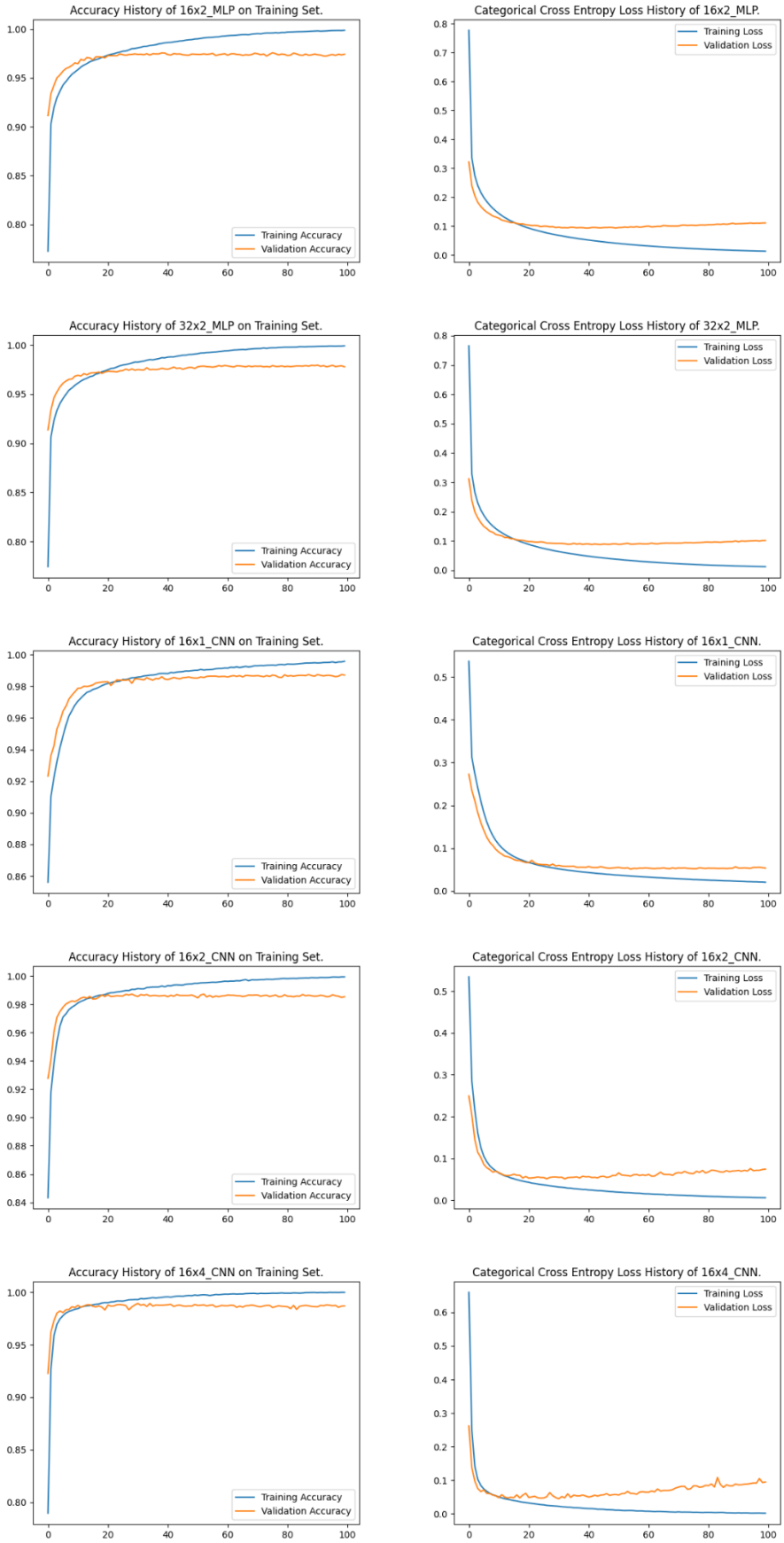


Figure B4

Final Results on MNIST

Model Name	Number of Parameters	Test Accuracy	Test Loss
64-16 MLP	51450	97.12%	0.1058853418
64-32 MLP	52650	97.21%	0.1072596163
32f CNN	**54410**	98.37%	**0.05450668931**
32-16f CNN	27994	98.48%	0.06588789076
32-16fx3 CNN	25594	**98.67%**	0.08556618541

As a whole, the Convolutional Neural Networks (CNNs) performed better than the simpler Multilayer Perceptrons (MLPs) in terms of test loss and test accuracy. Within the Convolutional Neural Networks, however, there is a curious set of relationships between model depth, test accuracy, and test loss. Notice how increasingly deeper CNNs produce greater test accuracy, but have higher test losses! Though loss is what the neural network optimizes for, accuracy is the more important metric for which loss acts are a heuristic for.

Figure B5

Final Results on Custom Handwritten Digits

Model	Handwritten Digit Accuracy
KNN	0.00%
SVM	25.00%
64-16 MLP	**50.00%**
64-32 MLP	45.00%
32f CNN	45.00%
32-16f CNN	45.00%
32-16fx3 CNN	45.00%

Every single model produced in our study, be it the traditional K-Nearest Neighbors and Support Vector Machine or a more advanced Neural Network scored pathetically low accuracy. Despite the large amount of preprocessing on the custom digits created by Gabe and myself. None of the models generalized well to the new data. Though the

neural networks performed much better than the more traditional models, they still scored 50% and below. Notice how the simplest MLP model outperformed all other neural networks. This suggests to us that though CNNs and larger MLPs have a higher capacity to learn the training set, they may be overfitting to the unique characteristics of the MNIST digits.

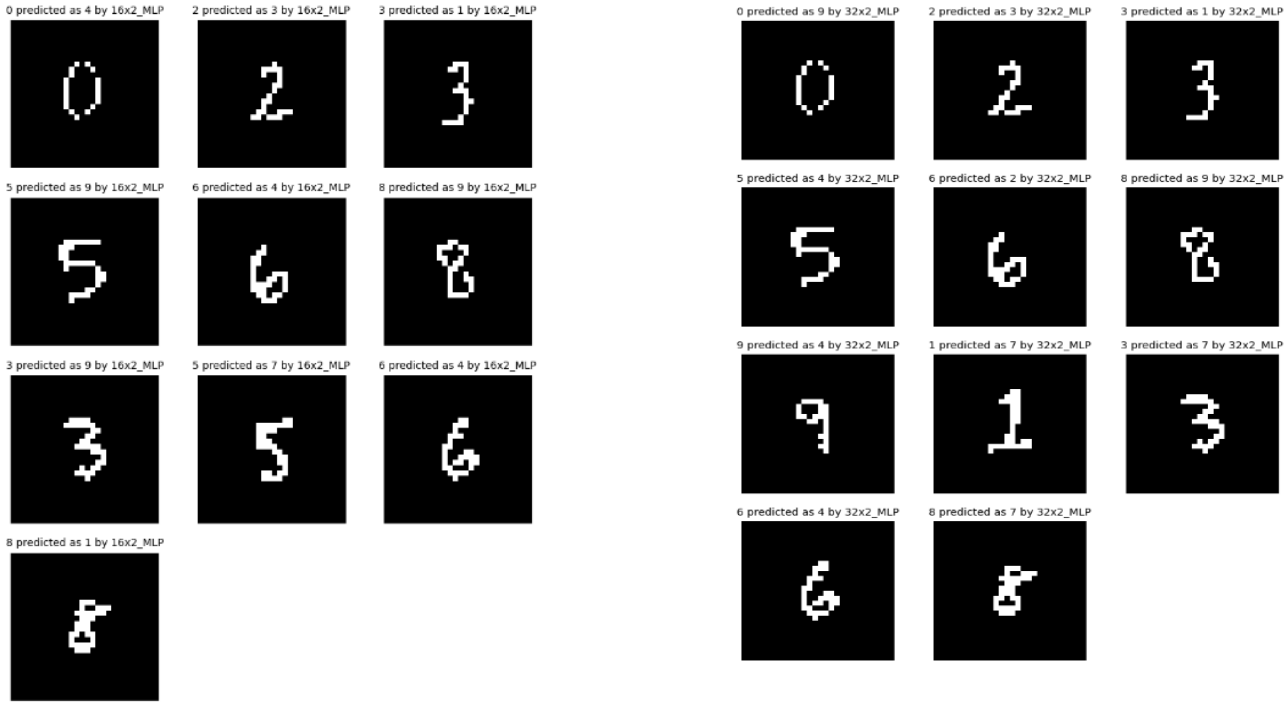
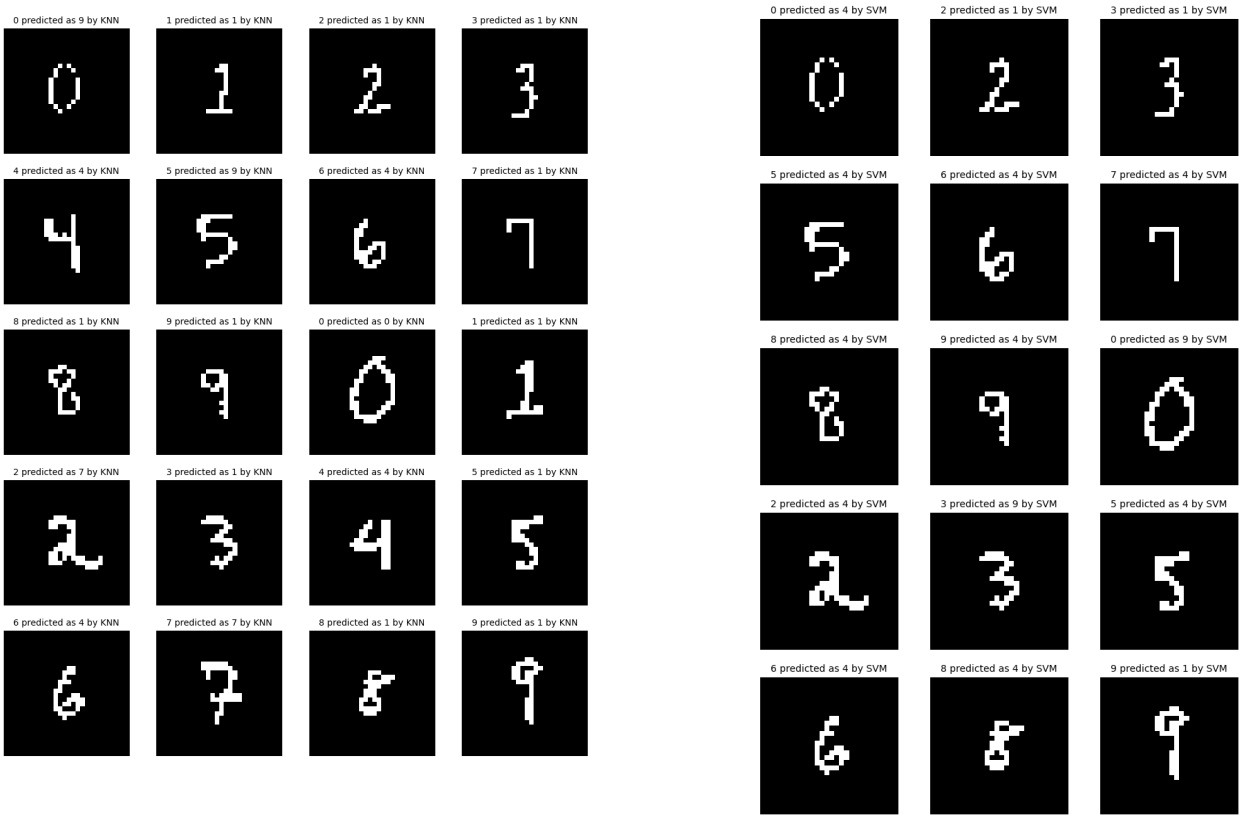


Figure B6

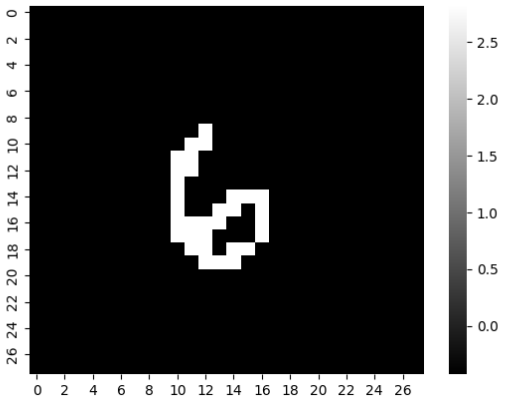
These two images showcase the results of both trained MLPs on the custom handwritten dataset. The 16x2 MLP scores only a 50% accuracy on the data, and the 32x2 MLP scores even worse at 45%. Figure 4 shows how both these models perform with greater than 97% accuracy on MNIST test sets. We hypothesize brush type, digit size, and slight variations in handwriting cause a lack of generalization in these models.

Figure B7



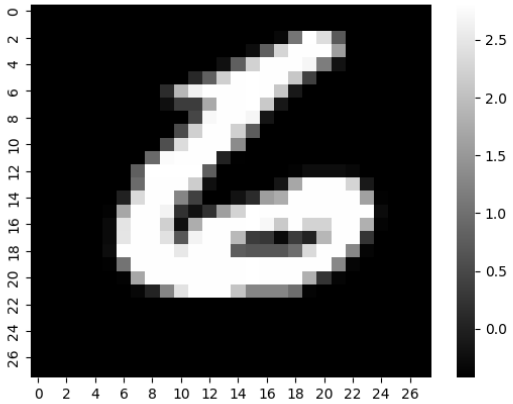
The traditional models performed much worse on custom handwritten digits. The KNN model (depicted on the left) did not predict a single class correctly. The SVM had a measly 25% accuracy. Seemingly only able to predict 1's, 4's and the occasional 7.

Figure C1



This is a hand drawn number by Gabriel, the number started off as a black hand drawn number on a white page, before applying the pre-processing steps. This is what the numpy array looks visualized, if it was plotted. Notice how there are only sharp edges, and there is no "soft pixelation" around the core shape of the number.

Figure C2



This is the number 6 from the MNIST data set. Which does look extremely similar to the 'Figure C1'. When observing the image, you can see that there is a softness around the harsh white inside base of the number. Note that the edge colors do end up being further down the scale, which is actually a gray color, which the model was trained on.

References

1.6. Nearest neighbors. (n.d.-a). Scikit-learn. https://scikit-learn.org/stable/modules/neighbors.html#classification

3.3. Metrics and scoring: quantifying the quality of predictions. (n.d.). Scikit-learn. https://scikit-learn.org/stable/modules/model_evaluation.html#model-evaluation

Ai, L. (2023, January 10). K-Nearest Neighbors (KNN) - Learn AI - Medium. *Medium*. https://medium.com/@divakar1591/k-nearest-neighbors-knn-f1677d989049

Anderson, A. (n.d.). *The difference between computer vision and machine vision*. https://www.clearview-imaging.com/en/blog/the-difference-between-computer-vision-and-machine-vision#:~:text=A%20machine%20vision%20system%20uses,of%20a%20larger%20machine%20system

Boureau, Y.-L., Ponce, J., & LeCun, Y. (2010, June 25). A Theoretical Analysis of Feature Pooling in Visual Recognition - ENS. NYU Scholars. https://www.di.ens.fr/willow/pdfs/icml2010b.pdf

Guide. (n.d.). TensorFlow. https://www.tensorflow.org/guide

Hamilton, D. (2020). kNN vs. SVM: A comparison of algorithms. US Forest Service Research and Development. https://www.fs.usda.gov/research/treesearch/62328

joblib.load — joblib 1.4.dev0 documentation. (n.d.). https://joblib.readthedocs.io/en/latest/generated/joblib.load.html

LeCun, Y., Bengio, Y., & Hinton, G. (2015, May 28). Review - Department of Computer Science, University of Toronto. Nature Deep Review. doi:10.1038/nature14539

McInnes, L., Healy, J. J., Saul, N., & Großberger, L. (2018). UMAP: Uniform manifold Approximation and Projection. *Journal of Open Source Software*, *3*(29), 861. https://doi.org/10.21105/joss.00861

Practices of Science: False Positives and False Negatives | manoa.hawaii.edu/ExploringOurFluidEarth. (n.d.). https://manoa.hawaii.edu/exploringourfluidearth/chemical/matter/properties-matter/practices-science-false-positives-and-false-negatives

Pupale, R. (2023, November 9). Support Vector Machines(SVM) — an overview - towards data science. *Medium*. https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989

Statquest with Josh Starmer. (n.d.). Home [Channel]. YouTube. Retrieved October 11, 2020, from https://www.youtube.com/channel/UCtYLUTtgS3k1Fg4y5tAhLbw

Team, K. (n.d.). *Keras: Deep Learning for humans*. https://keras.io/

Towards Data Science. (2001, December 9). *Towards data science*. https://towardsdatascience.com/

T-SNE. (n.d.). https://plotly.com/python/t-sne-and-umap-projections/