OPTIMIZING DIGIT RECOGNITION WITH THE MNIST DATASET

Paul Steller [1], Grace Ashley[2], Prosper Anyidoho[2], and Ibrahim Balogun[2]

[1] Faculty of Physical Sciences, Department of Mathematics, University of Delaware. (pstellar@udel.edu)

[2] Faculty of Engineering, Department of Civil and Environmental Engineering, University of Delaware (gashely@udel.edu, iobalo@udel.edu, panyidoh@udel.edu)

Abstract

Handwritten digit recognition is one of the most active research areas in machine learning due to its vast applications in different fields. Its popularity could be ascribed to the numerous alphabets and written digits that are peculiar to each country. In this paper, we optimized the accuracy of machine learning classifiers for digit recognition. Our models were trained on the popular MNIST dataset. The dataset contains 60,000 training and 10000 testing images. Results from the algorithms (SVC, KNN, Random forest and CNN) we adopted, suggested that CNN outperforms the other three models with a test error of only 0.55%. Finally, we deployed our CNN TensorFlow model in a user-friendly android environment.

1.0 Introduction

Image analysis is classified as a sub-field of Artificial Intelligence (AI). This area has drawn numerous attention due to the vast applications of images in today's world. Some of the applicable areas are Railway Engineering (Rail Corrugation), Highway Engineering (Intrusion Detection), and Biomedical Engineering (Brain Image processing, Breast cancer detection). Aside the active areas mentioned, one other area that has gained popularity in the last two decades is the Handwritten digits recognition. The popularity of this can be ascribed to the diversity of human spoken and written language with varying alphabets in languages such as Chinese, Arabic, French, English and many more. This sounds interesting in the sense that; machine learning models could be trained to recognize the images even when the style of writing differs for individuals. The application of digit recognition is not limited to handwritten letters but could also be applicable in office automation, check verification, postal address reading and printed postal codes, and data entry applications[1].

Since the release of the MNIST data set in 1988, several works have been done with different machine learning models. The first paper was published by LeCun et al. 1998, where a pairwise linear classifier approach was adopted with a test error of 7.9%[2]. Since then, it has been a competition among the machine learning community. The MNIST database keeps a record of top performing models trained on the MNIST data.

In this paper, we also used four (4) classification algorithms to optimize handwritten digits recognition and finally created an android app to identify user-drawn digits. The methods tested are Support Vector Classifier, K-Nearest Neighbors, Decision Trees and Convolution Neural Networks. The rest of the paper is structured as follows: Section 2 explains the reviews of some related articles. Section 3 tells about the methods used. Section 3 gives the experimental results. The last section is section 5 and it explains the conclusion as well as the areas to improve upon.

2.0 State of the Art

The MNIST database provides ranking for the different models trained on the MNIST data set, based on test errors reported for each model. The website also reports whether each model involved data preprocessing or augmentation. Early machine learning approaches adopted by LeCun et al. [3] included linear classifiers (with error rate ranges from 7.6 to 12%), K-nearest neighbors approaches (K-NN, ranging from 1.1 to 5%), non-linear classifiers (about 3.5%), support vector machines (SVM, from 0.8 to 1.4%), neural networks (NN, from 1.6 to 4.7%) and convolutional neural networks (CNN, from 0.7 to 1.7%)[2]. It is evident that models with data augmentation reported lower test error rates.

Though, models based on convolutional neural networks outperform other classifiers, some classical machine learning techniques are still able to provide competitive error rates. For example, Belongie et al. [4] achieved 0.63% and Keysets et al. [5] recorded 0.54% and 0.52% using K-NN, Kégl and Busa-Fekete [6] obtained 0.87% using boosted stumps on Haar features, LeCun et al. [3] achieved 0.8% and Decoste and Schölkopf [7] attained results from 0.56 to 0.68% using SVM.

Some recent studies have altered the architecture of the CNN architecture to optimize the performance of the model; MetaQNN [8], which used reinforcement learning for the design of CNNs , has reported an error rate of 0.44%, and 0.32% when using an ensemble of top models. Also, DEvol [9], which uses genetic programming, has obtained error rate of 0.6%. In 2018, Baldominos et al. [10] presented a work on grammatical evolution of the CNN architecture, and reported a test error rate of 0.37% without data augmentation. With neuroevolution of stacked CNNs[11] the error was narrowed down to 0.28%. Finally, the current best test error of 0.17%[12] was obtained by using a committee of 20 CNNs, coupled with a newly introduced squeeze and excitation[13] block added as layers in the network.

In the next section, we briefly describe each algorithm architecture adopted in this project.

3.0 Proposed Methods

3.1 Support Vector Machine

The Support Vector Machine is a specific type of supervised Machine learning that intends to classify data points by maximizing the margin among classes in a high-dimensional space. In a general term, the classification is done by finding the hyper-plane that differentiate the classes.

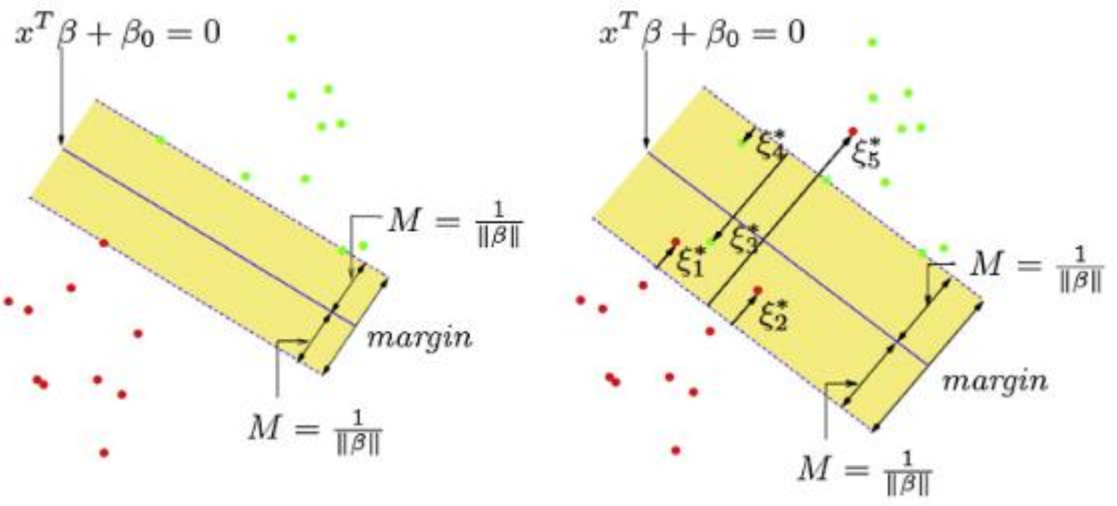


Figure1: An illustration of support vector classifier

In the figure above, the hyperplane clearly shows how the digits are classified correctly (Green) or misclassified (Red). The figure to the left could be related to the scenario where the image is obviously right. for instance, ξ_1, ξ_2, and ξ_3. However, the digits are misclassified due to orientation, positions, or sizes. An illustration to that could be seen on the right side of figure 2. The right panel shows the non-separable case in which the points labeled $\xi_1$are on the wrong side of the margin. The SVM can be expressed mathematically as follows:

$$f(x,w) = \sum_{j=1}^{n} w_j * g_j(x) + b \qquad (1)$$

Where $g_j(x)$ is a mapping function, w_j is the weight coefficient, b is the threshold. However, regression optimization constraints can be expressed as:

$$min\frac{1}{2}\|w\|^2 + c\sum_{i=}^{n}(\varepsilon_i + \varepsilon_j) \qquad (2)$$

Subject to
$$\begin{cases} y_j - f(x_i, w) \leq \varepsilon + \varepsilon_j^* \\ f(x_i, w) - y_i \leq \varepsilon + \varepsilon_j^* \\ \varepsilon i_{,}\varepsilon_j^* \gg 0, i = 1 \ldots \ldots . n \end{cases} \qquad (3)$$

Where c is the penalty parameter ε_i and ε_j^* are the slack variables for handling non separable data, ε is the insensitive loss function.

3.2 K-Nearest Neighbors

The K- nearest neighbors (KNN) algorithm is a simple, easy-to-implement supervised machine learning algorithm that can be used to solve both classification and regression problems [5]. The KNN assumes that similar things exist in proximity.

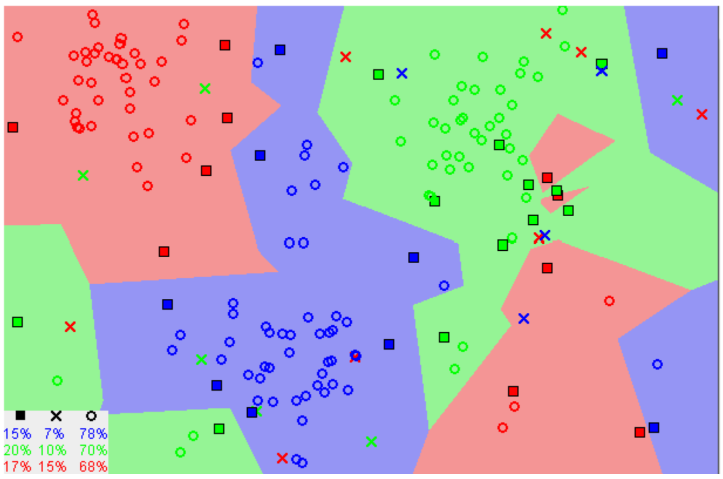


Figure 2. A sample KNN plane

Mathematically, it is expressed as follows:

For K training observations that close to x_0, represent by N_K. It then estimates f(x) using the average of all the training responses in N_o. (ISRL, 2014).

$$\hat{Y}(x) = \frac{1}{k} \sum_{x_i \in N_K(X)} y_i \tag{12}$$

Where $N_K(X)$ denotes the k-nearest neighbors of x to some metric, such as Euclidean distance, Minkowski or Manhattan.

3.3 Random Forest Classifier

The random forest is a supervised machine learning algorithm that produces accurate results when trained with datasets without any parameter tuning. Due to its simplicity and diversity, it can be used for both classification and regression problems.

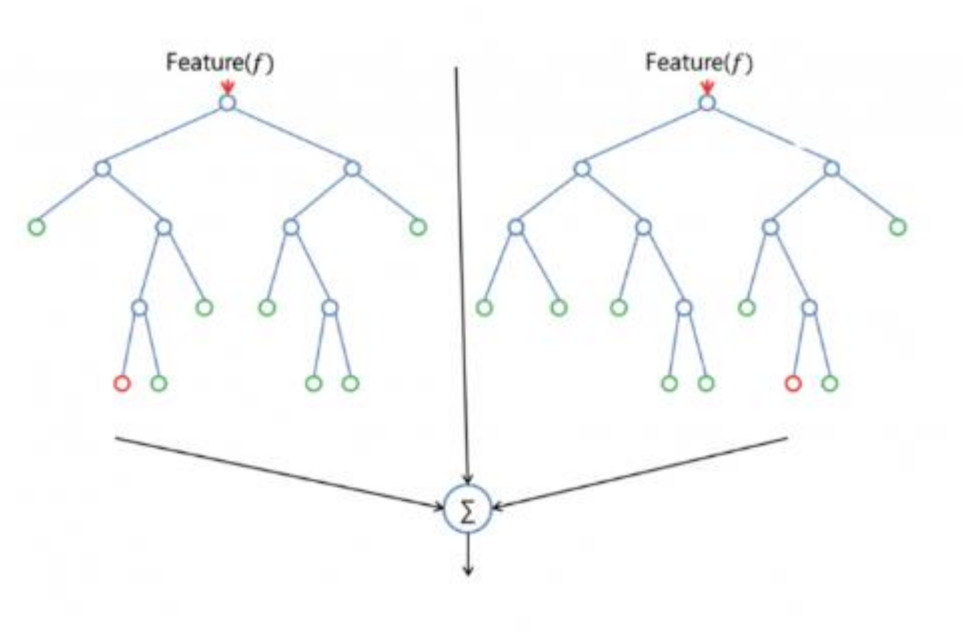


Figure 7. An illustration of a typical random forest assemblage

It is often claimed that the forest it builds, is an ensemble of decision trees, usually trained with the bagging method. Scholarly articles have been published as to the verification in the similarity and dissimilarity between random forest and decision tree. In this research, the estimator is varied to optimize its accuracy over the output predictions.

3.4 Convolution Neural Networks

A Convolutional Neural Network (ConvNet/CNN**)** is a deep learning algorithm which can take in an input image and assign weights to various sections of the image for the purpose of classification. The CNN model can learn these weights with enough training time and the right optimizers. The architecture of a ConvNet is like that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Figure 8 describes a simple CNN architecture for MNIST digit recognition. The figure shows an input which is passed through convolutional layers and later flattened for a fully connected feed forward neural layer which outputs predictions in the final layer.

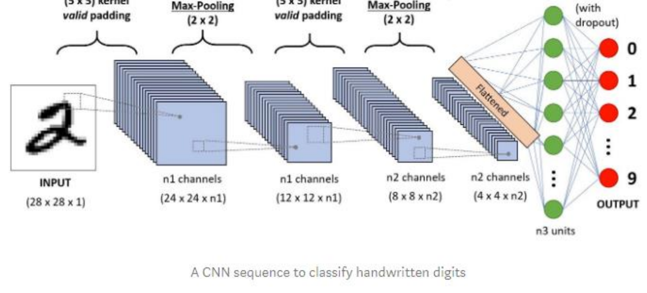


Figure 8: Simple CNN architecture for digit recognition

4.0 Results and Analysis

In this research, we used the popular MNIST dataset which contained 60,000 training example and labels. The datasets contain 10,000 test examples and labels. Each row consists of 785 values: the first value is the label (a number from 0 to 9) and the remaining 784 values are the pixel values (a number from 0 to 255). The machine learning algorithms are further trained on the datasets to determine which algorithm gives the best classification.

We consider the effect of the individual methods on the MNIST dataset holistically. The first section discusses the SVC, the second section discuss KNN, the third section discuss the Decision tree and the last section Convolution neural networks.

4.1 Support Vector Classifier Results

Support Vector Machines work by creating a (affine) hyperplane (an n - 1 subspace of an n dimensional space) to separate the data. It is worth noting that the data may not actually be separable, and so a separating hyperplane may not exist. Further, even if such a hyperplane does exist, it may not be unique. When the data is not perfectly separable, we allow for some of the points to be on the wrong side of the hyperplane, but we introduce a constant to control the error. We may also change the kernel function to better separate the data. Further, when the there are more than two classes in the data, there are a couple different approaches we may take. The "one versus all" approach fits the model to separate each class against every other class (as a whole). This differs from the "one versus one" approach, which fits the model to pairwise separate each class against every other individual class. Because the one versus one approach needs to fit the model many more times, it is more computationally expensive than the one versus all approach.
Now, looking at the MNIST data, the first thing we did was to test three different classes against each other.

Table 1: Classes of support vector classifiers

Classes of SVM		
Class	Error %	Total Time Taken
SVC	2.08	
NuSVC	7.87	
LinearSVC	14.63	
		115.74 minutes

Clearly, SVC does the best out of the box. While NuSVC and LinearSVC certainly could be improved upon (for instance, by optimizing over the kernel, degree, gamma parameter, etc.), we believed it would be best to focus on SVC. From the research done on optimizing SVC for the MNIST data, we found that one way to significantly improve the SVC performance was to deskew the data. Many of the handwritten digits were skewed and creating a more uniform dataset certainly seems like a logical way to improve the performance of the model. Following the guidance of the research, we implemented this transformation.

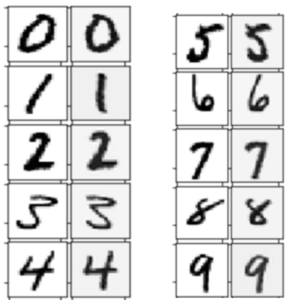


Figure 9: Skewed (Original) vs. Deskewed Digits

We can see that the deskewed digits (which are on the right in both columns) have a little bit of noise added through the deskewing process. However, as the noise seemed consistent throughout, this did not appear to hinder the effectiveness of the deskewing. As seen below, when tested on the out-of-the-box SVC, deskewing decreased the error rate by about 0.7%. Note, this improvement was achieved when both the training and test data were deskewed. When only the training set was deskewed, but the test set was left alone, the model did significantly worse. This makes sense, and implies that the model, while performing better, requires all input data to be deskewed before it can make an accurate prediction.

After determining that deskewing the data improved the SVC model, we optimized over the kernel. The default SVC kernel is rbf, which we already knew had a 1.38% error rate on the deskewed data. The additional kernels tested were linear, polynomial, and sigmoid. Below are the results:

Table 2: Results for SVC after augmentation

Preprocessing	Error %	Total Time Taken
Skewed	2.08	
Deskewed test and train data	1.38	
Deskewed only train data	89.68	
		14.4 minutes

Table 3: Results using different SVC kernels

Optimizing over Kernel on Deskewed Data		
Kernel	Error %	Total Time Taken
Linear	3.62	
Poly	1.35	
Sigmoid	30.44	
		22.82 minutes

As we can see, the polynomial kernel (with default degree 3) performed the best. We wanted to make sure that deskewing the data did not actually make the polynomial kernel worse, so we optimized over the degree for both the skewed and deskewed data. As expected, the deskewed data performed significantly better for all degrees tested (from 1 to 12). Surprisingly, optimizing over the original, skewed data took roughly three times longer (234.53 minutes) versus the deskewed, preprocessed data (77.16 minutes). We can see from the table below that the lowest error achieved for the skewed (original) data is 1.56% when the degree is 2. The lowest error achieved for the deskewed (preprocessed) data is 1.35% when the degree is 3. In other words, optimizing over the degree did not actually improve our model at all, as the default degree was already 3 when we tested the polynomial kernel earlier. It is worth nothing, however, that for both the skewed and deskewed data, the lowest error percentages occurred around degrees 2 through 5. Although the error steadily increased as the degree increased passed 5, the error increase was much more drastic for the skewed data. This can be seen in the scatter plot below.

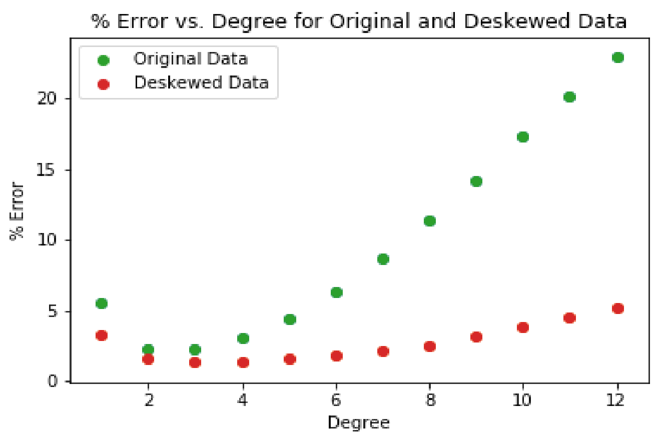


Figure 10: Results for varying polynomial degrees

Table 4: Results for varying polynomial degrees

Optimizing over Degree, Skewed vs. Deskewed		
Degree	Skewed/Deskewed	Error %
1	Skewed	5.55
1	Deskewed	3.24
2	Skewed	2.26
2	Deskewed	1.56
3	Skewed	2.29
3	Deskewed	1.35
4	Skewed	3.02
4	Deskewed	1.38
5	Skewed	4.39
5	Deskewed	1.56
6	Skewed	6.26
6	Deskewed	1.83
7	Skewed	8.68
7	Deskewed	2.10
8	Skewed	11.35
8	Deskewed	2.54
9	Skewed	14.19
9	Deskewed	3.12
10	Skewed	17.27
10	Deskewed	3.83
11	Skewed	20.14
11	Deskewed	4.51
12	Skewed	22.89
12	Deskewed	5.20

The other form of preprocessing that the literature indicates is useful for optimizing SVC for the MNIST data is “pixel jittering.” This involves translating each image by a very small amount (1 or 2 pixels) in each direction to create new data points. In this way, for each data point, we create four additional data points. This increases the size of the training dataset from 60,000 to 300,000.

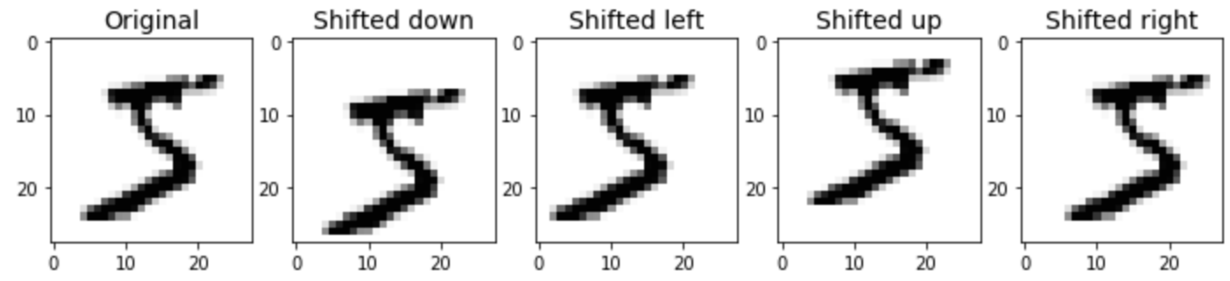


Figure 11: 2-pixel jittering on the first element of the dataset

When testing this augmented dataset, we chose to use the parameter C = 2 (the default is C = 1). This C value is what the literature suggested. In an ideal situation, the C parameter, along with the gamma parameter, would be optimized using GridSearchCV. However, given the size of the dataset, this was not realistic. Below is the result of optimizing over the degree for the deskewed and translated data (1-pixel jitter). Note, this optimization took just short of 17 hours to complete.

Table 5: Results after 1-pixe jitter transform

Optimizing over Degree, Deskewed with 1-Pixel Jitter	
Degree	Error %
1	3.22
2	0.90
3	0.84
4	0.97
5	1.06
6	1.19
7	1.41
8	1.66
9	1.93
10	2.26

It was our intention to additionally optimize over the degree with the deskewed and 2-pixel jittered data. However, training this model for a single degree (degree 9) took almost 4 hours, and it resulted in a worse model (2.33% error). Thus, this was not completed. And so, out of all the models we tested, the best was the SVC with degree 3 polynomial kernel, C = 2, and with deskewing and 1-pixel jittering preprocessing (0.84% error). After training this model, we analyzed the confusion matrix in order to take a better look at how the model performed.

```
[[ 977    0    1    0    0    0    1    1    0    0]
 [    0 1131    1    0    0    1    2    0    0    0]
 [    2    4 1019    1    1    0    1    1    3    0]
 [    0    0    0 1005    0    2    0    2    0    1]
 [    0    0    0    0  975    0    4    0    1    2]
 [    1    0    0    4    0  882    2    1    1    1]
 [    2    1    0    0    3    3  948    0    1    0]
 [    2    0    3    1    0    0    0 1020    0    2]
 [    2    0    1    0    1    2    0    1  967    0]
 [    0    1    0    2    7    3    0    4    0  992]]
```

Figure 12: Confusion Matrix for the SCV model with the lowest error %

As we can see, the model did very well. We can also look at some of the false negatives and false positives that the model predicted on the test set. For instance, here are all the examples in the test set where the actual digit was a 9, but the model predicted a different value.

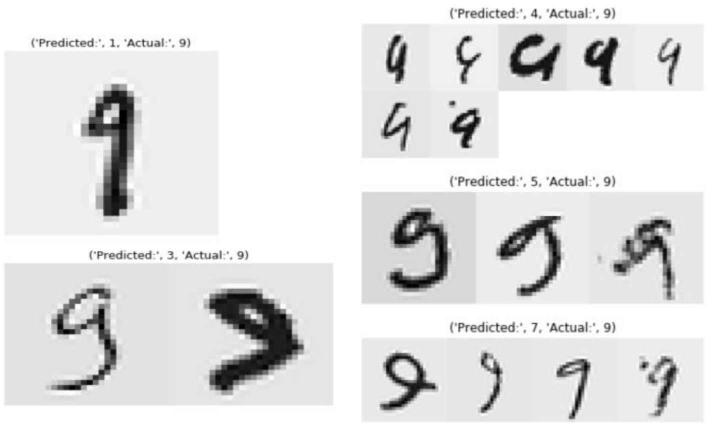

('Predicted:', 1, 'Actual:', 9)
('Predicted:', 4, 'Actual:', 9)
('Predicted:', 3, 'Actual:', 9)
('Predicted:', 5, 'Actual:', 9)
('Predicted:', 7, 'Actual:', 9)

Figure 13: Handwritten ‘9’s that were incorrectly labeled as a different number.

Some of these errors are entirely reasonable and might even confuse the human eye. The following are all of the digits that the model predicted as a 9 but were actually a different digit.

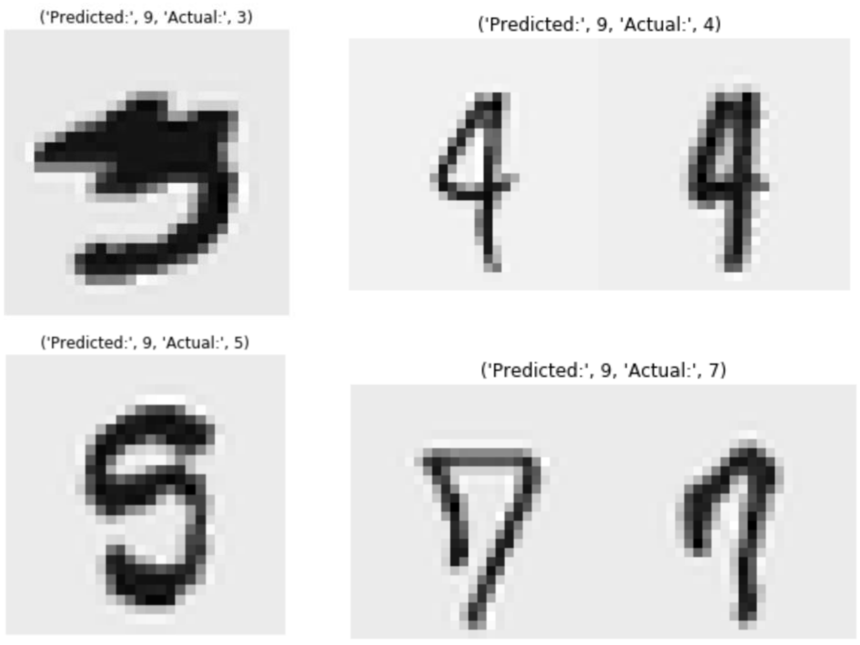

('Predicted:', 9, 'Actual:', 3)
('Predicted:', 9, 'Actual:', 4)
('Predicted:', 9, 'Actual:', 5)
('Predicted:', 9, 'Actual:', 7)

Figure 14: Numbers that were incorrectly labeled as a ‘9’

It is worth pointing out that some of the digits are written so comically bad, that surely most models would label them incorrectly. A sample is given below.

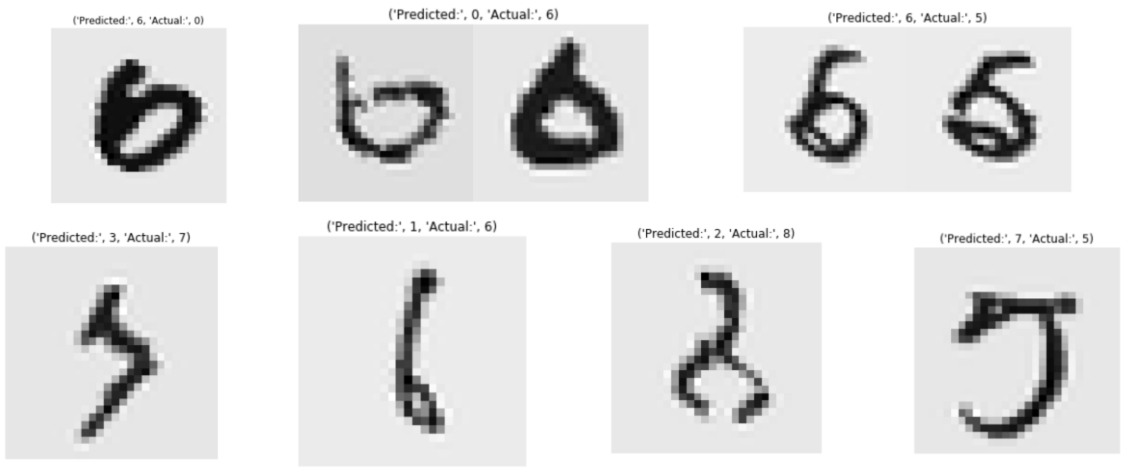

('Predicted:', 6, 'Actual:', 0)
('Predicted:', 0, 'Actual:', 6)
('Predicted:', 6, 'Actual:', 5)
('Predicted:', 3, 'Actual:', 7)
('Predicted:', 1, 'Actual:', 6)
('Predicted:', 2, 'Actual:', 8)
('Predicted:', 7, 'Actual:', 5)

Figure 15: Comically written digits

4.2 K-Nearest Neighbors

We assessed how well the K-Nearest Neighbor (KNN) Classifier could perform predictions on the MNIST dataset. We selected and tested nearest neighbors 2, 5 and 7 to perform this analysis.

Table 6: Results for KNN with k=,2,5,7

KNN on Original Data			
Nearest Neighbor	Error %	Model Ranking	Total Time Taken
2	3.73	3	
5	3.12	1	
10	3.35	2	
			52 minutes

First, this analysis was performed on the dataset which was free of any form of augmentation. The model created with nearest neighbors=5 outperformed the other models although it was by a slight margin. The model with nearest neighbors=2 produced the highest error rate. Although the errors were not extremely high, we believed that we could improve the results by applying a data augmentation method that had been implemented by other researchers trying to achieve lower errors with this kind of data.

In the data augmentation process, we decided to apply was deskewing. We then performed the same analysis on the deskewed data to observe any changes on model performance. The results proved that deskewing the images in the dataset was a very effective way to improve model performance. For nearest neighbor=2, the model improved my 1.8%. An improvement of 1.43% was observed for nearest neighbor=5 and the error for nearest neighbor=10 reduced by 1.48%. Although nearest neighbor=10 saw the best model improvement, nearest neighbor=5 still outperformed the other models.

Table 6: Results for KNN after data augmentation with k=,2,5,7

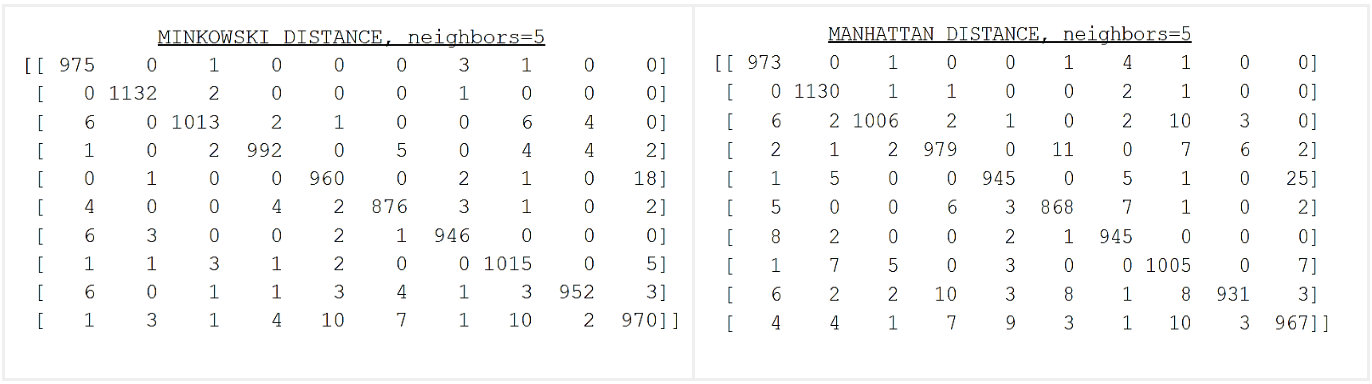
KNN on Deskewed Data			
Nearest Neighbor	Error %	Model Ranking	Total Time Taken
2	1.93	3	
5	1.69	1	
10	1.87	2	
			57 minutes

The distance metric used for obtaining models using the deskewed and skewed datasets was the Minkowski Distance. We decided to change the distance metric to observe the effect on model performance. This analysis was performed on the best model obtained from the deskewed data which was the one with nearest neighbor=5.

Table 7: Results with distance metric

Effect of Changing Distance Metric on Model			
Distance	Neighbors	Error %	Model Rank
Minkowski	5	1.69	1
Manhattan	5	2.51	2

We chose to observe the effect of changing the distance metric from Minkowski to Manhattan for nearest neighbor=5. The results showed that there was a 0.82% increase in the error rate when the distance metric was changed. We decided that if this model was going to be used, we would have to maintain the Minkowski distance metric. We went ahead to produce classification reports and confusion matrices with visualizations for nearest neighbor=5 with the different distances.



```
MINKOWSKI DISTANCE, neighbors=5
[[ 975    0    1    0    0    0    3    1    0    0]
 [   0 1132    2    0    0    0    1    0    0    0]
 [   6    0 1013    2    1    0    0    6    4    0]
 [   1    0    2  992    0    5    0    4    4    2]
 [   0    1    0    0  960    0    2    1    0   18]
 [   4    0    0    4    2  876    3    1    0    2]
 [   6    3    0    0    2    1  946    0    0    0]
 [   1    1    3    1    2    0    0 1015    0    5]
 [   6    0    1    1    3    4    1    3  952    3]
 [   1    3    1    4   10    7    1   10    2  970]]
```

```
MANHATTAN DISTANCE, neighbors=5
[[ 973    0    1    0    0    1    4    1    0    0]
 [   0 1130    1    1    0    0    2    1    0    0]
 [   6    2 1006    2    1    0    2   10    3    0]
 [   2    1    2  979    0   11    0    7    6    2]
 [   1    5    0    0  945    0    5    1    0   25]
 [   5    0    0    6    3  868    7    1    0    2]
 [   8    2    0    0    2    1  945    0    0    0]
 [   1    7    5    0    3    0    0 1005    0    7]
 [   6    2    2   10    3    8    1    8  931    3]
 [   4    4    1    7    9    3    1   10    3  967]]
```

Figure 16: Confusion matrix for different distance metrics

We took a closer look at each label prediction by creating confusion matrices for predictions of both models. Both models did well with the predictions although as stated earlier, the model with the Minkowski distance metric performed slightly better. The images labelled as “1” were predicted most accurately by both models. The least accurately predicted label was “9” by both models. Most of the misclassified “9” labels were predicted to be “4” and “7”. Many “2” labels were also predicted to be “7” by the model with the Manhattan metric. Precision, Recall, F1-score, and Support for each label have been reported in the table below.

Table 8: Evaluation metrics across individual digits

	Precision		Recall		F1-score		Support	
Label	Manhattan	Minkowski	Manhattan	Minkowski	Manhattan	Minkowski	Manhattan	Minkowski
0	0.97	0.97	0.99	0.99	0.98	0.98	980	980
1	0.98	0.99	1.00	1.00	0.99	1.00	1135	1135
2	0.99	0.99	0.97	0.98	0.98	0.99	1032	1032
3	0.97	0.99	0.97	0.98	0.97	0.99	1010	1010
4	0.98	0.98	0.96	0.98	0.97	0.98	982	982
5	0.97	0.98	0.97	0.98	0.97	0.98	892	892
6	0.98	0.99	0.99	0.99	0.98	0.99	958	958
7	0.96	0.98	0.98	0.99	0.97	0.98	1028	1028
8	0.99	0.99	0.96	0.98	0.97	0.98	974	974
9	0.96	0.97	0.96	0.96	0.96	0.97	1009	1009

4.3 Random Forest Classifier

Random Forest is a very easy model to build and can produce quite impressive results. Of course, the data used in this model is not as complicated and does not require as much pre-processing as a bigger dataset would. The most important parameter of the Random Forest Classifier is the number of estimators. We run the model for 20, 50, 75 and 100 estimators

The accuracy obtained by varying the number of estimators does not have much effect on larger trees e.g 50,75 and 100. However, the least estimator gave the accuracy of 96% and as such ranked 4th. In general, the digits are well classified using the Random Forest except for the digit “9” which was weakly classify across the estimators. Table 9 below shows different evaluation metrics and their performance on each digit for the optimal number of estimators (100)

Table 9: Evaluation metrics for optimal number of estimators

N-Estimator	Label	Precision	Recall	F1 score
100	0	0.97	0.99	0.98
	1	0.99	0.99	0.99
	2	0.96	0.97	0.96
	3	0.96	0.94	0.95
	4	0.97	0.97	0.97
	5	0.97	0.97	0.97
	6	0.98	0.99	0.98
	7	0.98	0.97	0.97
	8	0.96	0.96	0.96
	9	0.95	0.95	0.95

4.4 Convolution Neural Networks (CNNs)

Similarly, we trained a multi-layer CNN to classify our digits. The model architecture adopted is similar to traditional CNN layers. The only introduction here is the use of a squeeze and excitation block. This layer ensures channel wise and spatial patterns across image channels are correctly captured. The optimizer adopted is Adam which ensures weights can reach optimal points after a considerable number of epochs. The detailed architecture of the CNN used is presented in our notebooks. Before applying the model to the test data, we carried out validation to see how well the model performs. As shown in the figure below, the training and validation loss decreases with increase in number of epochs. It is also evident that, the validation loss at each epoch is less than that of the training loss which is a good sign our model is not overfitting.

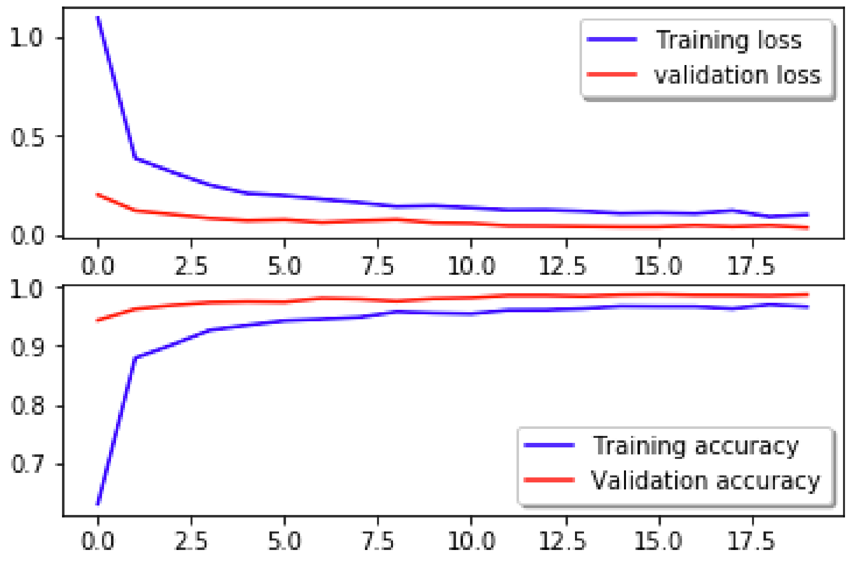
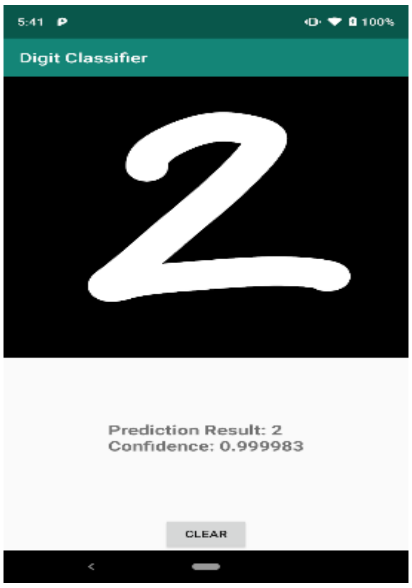


Figure 16: Training and validation loss

Applying the model to the test set gave an error of 0.55% which still happens to be less than the current best error of 0.17%. To deploy the model on an android environment, we converted the final model into a TensorFlow Lite model. However, to run such a model on an android environment with limited resources, we had to perform quantization on the weights. Quantization decrease precision of model weight and as a result decreases the test performance of the model. The figure below shows how the model is can be used to recognize user drawn digits. For instance, the user drawn digit of 2 has been classified correctly by the app with confidence of approximately 1.



Prediction Result: 2
Confidence: 0.999983

Figure 17: Android app demo

5.0 Discussion and Conclusion

In this paper, we optimized four classification algorithms on the MNIST data set for digit recognition. The results suggested, the performance of models increases when data augmentation is used during data processing. Data augmentation provides new training instances which helps to solve the problem of high variance. That is, we assist models to have high generalization which improves performance when deployed. The CNN model turned out to be the best model among the 4 models trained. This makes sense since, CNN model architecture is perfectly designed for images. We also went ahead and deployed our final CNN model in android environment, where a user draws a digit in a canvas for the app to classify.
In the future, we might consider integrating a camera component in our app, where a user could classify digits in real time. We can also extend this application to other types of texts and alphabets.

References

[1] A. Baldominos, Y. Saez, and P. Isasi, "A survey of handwritten character recognition with MNIST and EMNIST," *Appl. Sci.*, vol. 9, no. 15, 2019.

[2] C. J. C. LeCun, Y.; Cortes, C.; Burges, "The MNIST Database of Handwritten Digits. 2012," 2012. [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed: 23-May-2020].

[3] P. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, "Gradient-based learning applied to document recognition," *IEEE*, vol. 86, 1998.

[4] J. Belongie, S.; Malik, J.; Puzicha, "Shape matching and object recognition using shape contexts," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, pp. 509–522, 2002.

[5] D. Keysers, T. Deselaers, C. Gollan, and H. Ney, "Deformation models for image recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 8, pp. 1422–1435, 2007.

[6] R. Kégl, B.; Busa-Fekete, "Boosting products of base classifiers," in *In Proceedings of the 26th Annual International Conference on Machine Learning*, 2009, pp. 497–504.

[7] B. Decoste, D.; Schölkopf, "Training invariant support vector machines.," *Mach. Learn.*, vol. 46, pp. 161–190, 2002.

[8] R. Baker, B.; Gupta, O.; Naik, N.; Raskar, "Designing Neural Network Architectures using Reinforcement Learning.," in *In Proceedings of the 5th International Conference on Learning Representations*, 2017.

[9] J. De. Davison, "Automated Deep Neural Network Design," *Genetic Programming*, 2017. [Online]. Available: https://github.com/joeddav/devol. [Accessed: 21-May-2020].

[10] P. Baldominos, A.; Saez, Y.; Isasi, "Evolutionary Convolutional Neural Networks: An Application to Handwriting Recognition. Neurocomputing.," vol. 283, pp. 38–52, 2018.

[11] P. Baldominos, A.; Saez, Y.; Isasi, "Hybridizing Evolutionary Computation and Deep Neural Networks: An Approach to Handwriting Recognition Using Committees and Transfer Learning.," 2019.

[12] Matuzas, "MNIST classifier with average 0.17% error," 2019. [Online]. Available: https://github.com/Matuzas77/MNIST-0.17.

[13] E. Hu, Jie; Shen, Li; Albanie, Samuel; Sun, Gang; Wu, "Squeeze-and-Excitation Networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, 2019.