Option 1: Coding a Convolutional Neural Network from Scratch

COSC 4P80 Project

Geoffrey Jensen
Student
Brock University
St. Catharines, Canada
al20rm@brocku.ca 7148710

Nicholas Parise
Student
Brock University
St. Catharines, Canada
np21ei@brocku.ca 7242530

Stephen Stefanidis
Student
Brock University
St. Catharines, Canada
jc20vo@brocku.ca 7140030

***Abstract*—Convolutional Neural Networks are deep learning models that excel at feature recognition in images. To learn more about them, we designed and implemented them from scratch, which allowed us to gain a much deeper understanding of how they work. To test our network's performance, we used the MNIST dataset of handwritten digits. We trained six different architectures, with varying numbers of convolutional layers, kernel sizes, pooling sizes, and hidden fully connected layers. Overall our models achieved an accuracy of 96-98% on the test set, which far exceeded our expectations, and gave us confidence in our implementation.**

I. Introduction

For our project we chose option 1, where we would design and implement a Convolutional Neural Network (CNN) from scratch (NumPy is the only non-standard dependency). We covered CNN's at a fairly high-level in class, and we wanted to learn more about how they worked at the fundamental level, which includes how the kernels are trained using backpropagation.

We designed our CNN architectures based off of the models we observed in lecture, and used the MNIST dataset to test our implementation. The results of our trained models showed high accuracy on the test set, which we were very content with.

Trying to debug our implementation was frustrating at times, the size of the training set coupled with the slow speed of the earlier implementations meant it took hours to track down hard to find bugs. At times we considered switching to option 2, however we stuck with it and managed to obtain results above our expectations.

II. Data

The dataset used in this project is the MNIST handwritten digit dataset. The MNIST dataset is one of the most influential datasets in machine learning and computer vision. It was introduced in 1998 by Yann LeCun, Corinna Cortes, and Christopher J.C. Burges as a standardized and more accessible version of the original NIST handwritten digit datasets. This original NIST data consisted of scanned handwritten digits collected from U.S. Census Bureau employees and high school students, which designated the Census digits as the training set and the student digits as the test set. Therefore, the main motivation behind the MNIST version was to mix these two datasets so that the result can be independent of the choice of training set and test among the complete set of samples [1]. In this project, the dataset was obtained from a Kaggle-hosted copy of MNIST, as the official MNIST database, previously cited, was unavailable at the time of writing.

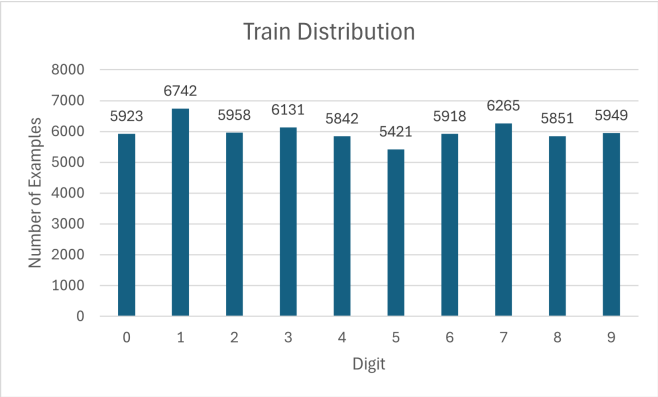
MNIST consists of 70,000 grayscale images of handwritten digits (0-9), split into 60,000 training images and 10,000 test images. Each image is 28x28 pixels, resulting in a single-channel (grayscale) matrix with pixel values in the range [0, 255]. Each image was centered to fit in the 28x28 box, meaning that the model is more suitable for centered digits when testing. Each image is paired with a single integer label representing its digit class.

The training and testing splits of the MNIST dataset maintain a balanced class distribution, with each digit represented in roughly equal proportions. While the class distribution is largely balanced, there are slight variations, with the digit 1 appearing marginally more frequently and the digit 5 appearing slightly less frequently than other classes. Overall, both subsets share similar data distributions, ensuring that evaluation on the test set reflects model generalization. The exact distribution is shown in Figure 1.

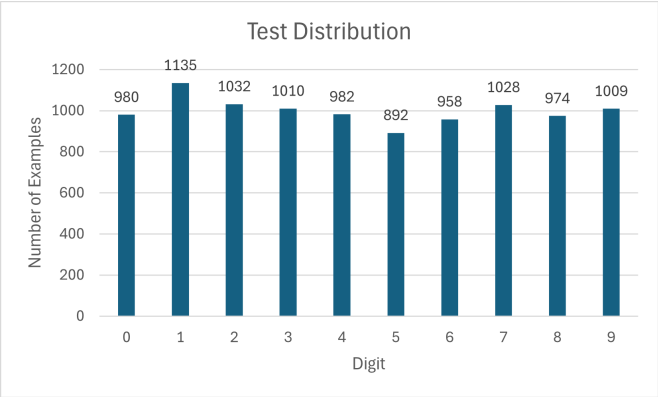
III. Design

The design of our CNN was based off the architectures we studied in class. Each network has one or more convolutional layers, which each have a collection of kernels, where the resulting kernel computation values are activated by a standard ReLU, and then downsized using a max pooling layer. It's possible for CNNs to have two layers of kernels directly connected, without ReLU or pooling in between, but we simplified our convolutional layers to always go from kernel computation, to ReLU, and then to max pooling. The result of the last convolutional layer is then flattened and fed to the fully connected layers (dense layers).

The fully connected layers consist of zero or more layers using sigmoid as their activation function, and then the final



(a) Data Distribution of Train Set



(b) Data Distribution of Test Set

Fig. 1: MNIST Data Distribution

softmax layer, where the probabilities of each class are calculated.

Figure 2 shows a generalized CNN network. Note that this network only has one convolutional layer, but more are possible to allow for more complex feature detection.

IV. Training

The network was trained using backpropagation, and categorical cross-entropy (CCE) as the loss function.

CCE combines with softmax to make a simple gradient calculation at the output layer. With,

$\hat{Y}$ being the network output.
Y being the expected output.
O being the hidden layer output.
δ_L being the error at layer L.
X being the input to the layer.
α is the learning rate.

The error in the fully connected layers are calculated by,

$\delta_L = \hat{y} - y$, at the output softmax layer.

$\delta_{L,i} = o_i \cdot (1 - o_i) \cdot \Sigma_{j=1}^{q} w_{i,j} \cdot \delta_{L+1,j}$, at the hidden sigmoid layer. (where q is the number of neurons in $L + 1$)

With the weight and bias adjustments being,

$$\Delta w_{i,j} = \alpha \cdot x_i \cdot \delta_j$$
$$\Delta b_j = \alpha \cdot \delta_j$$

Since the output of the last convolutional layer is flattened for the first fully connected layer, we have to reshape δ to be same shape as the output of that last convolutional layer.

To perform backpropagation through a convolutional layer, we will have $\frac{dL}{dP} = \delta$ passed in as the layer's error. We must then propagate this error back through the max pooling and ReLU layers before we can calculate our kernel adjustments.

First, we must find $\frac{dL}{dC}$, which is the derivative the loss with respect to the max pooling layer. To do this, we multiply our $\frac{dL}{dP}$ by the max pooling derivative $\frac{dC}{dX}$. We can think of the max pooling layer as a function $C(x)$, where $C(x) = x$ when x is the largest value in the pooling filter, else $C(x)$ = 0. Thus, $\frac{dC}{dX}$ will be 1 if x is the largest in the filter, else it will be 0. Figure 3 gives a visual representation of a max pooling filter, and it's derivative.

Now that we have $\frac{dL}{dC}$, we must calculate $\frac{dL}{dZ}$ by multiplying $\frac{dL}{dC}$ by $\frac{dZ}{dX}$. $\frac{dZ}{dX}$ is the derivative of our ReLU activation function, which is simple to calculate. $\frac{dZ}{dX} = 1$ if $x > 0$, else it's equal to 0.

Now that we have $\frac{dL}{dZ}$, we can calculate our kernel weight adjustments, our bias adjustments, and our $\frac{dL}{dX}$, which is the error we will propagate back to preceding convolutional layer.

We have,

$\frac{dL}{dZ}_n$ is the error of kernel n.
$K_{n,d}$ is kernel n at channel d.
B_n is the bias for kernel n.
X is the input to the layer.
α is the learning rate.

The number of channels (c) in X matches the depth of each kernel, and the depth of $\frac{dL}{dZ}$ is equal to the number of kernels in the layer.

Even though our input image is only one channel, if the first convolutional layer has more than one kernel, then the input to the next convolutional layer will have multiple channels. These kernels will be 3D, while each output feature map corresponds to one kernel. These 3D kernels are what allow the CNN to combine the features captured by the kernels in the previous layer. So when we calculate $\Delta K_{n,d}$, we must convolve $\frac{dL}{dZ}_n$ on every channel of the input.

The kernel adjustments are calculated by convolving the layer's input with $\frac{dL}{dZ}_n$.

$$\Delta K_{n,d} = \alpha \cdot convolve(X_d, \frac{dL}{dZ}_n)$$

The bias adjustments are easy to calculate, as they are simply the sum of the error for that kernel (tempered by the learning rate).

$$\Delta B_n = \alpha \cdot sum(\frac{dL}{dZ}_n)$$

To calculate the gradient to propagate backwards, $\frac{dL}{dX}$, we convolve a padded $\frac{dL}{dZ}$ matrix with each kernel, where each

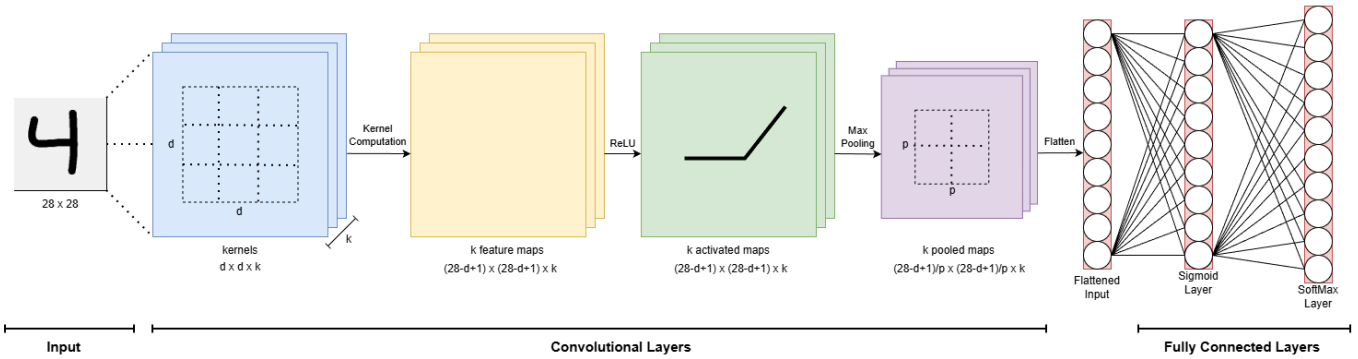


Fig. 2: Convolutional Neural Network Generalized Design

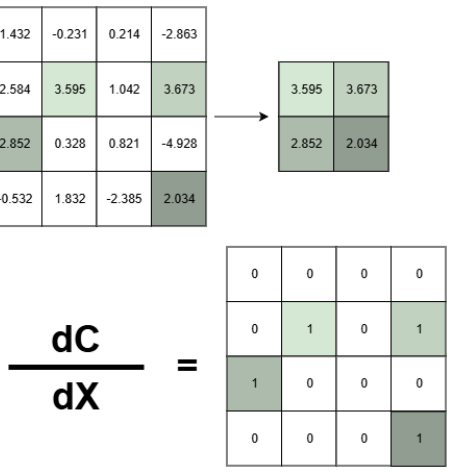


Fig. 3: Derivative of Max Pooling Layer (2x2, Stride=2)

kernel is rotated 180 degrees. For 3D kernels, we convolve the slice of every channel with every slice of $\frac{dL}{dZ}$, to calculate the error at each input channel.

$$\frac{dL}{dX}_n = \Sigma_{i=1}^{k} convolve(padded(\frac{dL}{dZ}_k), rot180(K_{n,k}))$$

When performing the training runs, we used the entire MNIST training set (60,000 images) for learning, and the entire test set (10,000 images) for performance analysis. We performed a backpropagation pass for every training example, and shuffled the order of the training set after every epoch. After only one epoch, our networks would already achieve an accuracy of 95% or more on the test set, with very minor improvements after successive epochs, before starting to diverge. The amount of time it takes for a network to train is also very large, especially for the deeper networks. For our largest networks, they were only able to complete eight epochs after 24 hours of training time on one of our machines. This will be discussed again later in the **Issues Faced** section.

V. Implementation

There are lots of different parameters that define a given network (including the number of layers of each type, kernel dimensions, number of kernels, pooling dimensions, number of hidden nodes, etc), plus the hyperparameters to train it (learning rate, max epochs, seed). To provide all of this information to our training program, we created a JSON file format to define a network's configuration.

To simplify the backpropagation implementation, the stride of kernel computations is always one, and the stride of pooling is equal to the dimension of the pooling filter.

To train a network, run train.py with a defined network configuration file. The configurations used to train our models are included in the submission for your reference.

Example execution:

```
python train.py configs/config_A.json
```

VI. Results

We trained six different architectures to try to achieve the highest classification accuracy on the test set. Table I labels each model, summarizes their architecture, and gives their best recorded accuracy against the test set, also with the learning rate (α) used, and the number of epochs it was trained for.

We tried to create an adequate variety of architectures, where some are small with only one layer of 3x3 kernels, and others are deeper with two layers of 5x5 kernels plus a fully connected sigmoid layer.

In addition, precision and recall were calculated for each class by recording false positives, false negatives and true positives when applying each model to the test set. Table II shows the precision percentage for each digit class, which is the fraction of predicted positives that are actually correct.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Table III shows the recall percentage for each digit class, which is the fraction of actual positives that are correctly identified.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Confusion matrices were also generated with the Matplotlib and Seaborn libraries that show how the models perform with different classes at a glance. We made a matrix for each model's predictions on the test set and they are shown in Figure 4.

Model	Architecture	α	Epochs	Accuracy
A	Kernels(3x3x8, padding=0) ReLU() MaxPool(2x2) Flatten() SoftMax()	0.05	7	97.23%
B	Kernels(5x5x6, padding=0) ReLU() MaxPool(2x2) Kernels(5x5x6, padding=0) ReLU() MaxPool(2x2) Flatten() SoftMax()	0.01	3	98.16%
C	Kernels(5x5x6, padding=0) ReLU() MaxPool(2x2) Kernels(5x5x6, padding=0) ReLU() MaxPool(2x2) Flatten() FullyConnected(16) SoftMax()	0.01	2	97.52%
D	Kernels(3x3x8, padding=0) ReLU() MaxPool(2x2) Kernels(3x3x8, padding=0) ReLU() MaxPool(2x2) Flatten() SoftMax()	0.01	6	98.09%
E	Kernels(3x3x10, padding=0) ReLU() MaxPool(2x2) Kernels(3x3x6, padding=0) ReLU() MaxPool(2x2) Flatten() FullyConnected(18) SoftMax()	0.01	8	97.58%
F	Kernels(3x3x8, padding=1) ReLU() MaxPool(3x3) Kernels(3x3x8, padding=1) ReLU() MaxPool(2x2) Flatten() FullyConnected(20) SoftMax()	0.01	4	96.80%

TABLE I: Model Architecture Overview. Includes Learning Rate (α), Number of Epochs Trained for, and Test Set Accuracy.

The error of the models decreased very quickly, and with a large amount of data, we could not measure the error by epoch like we did for the feed-forward network assignment. Instead, we chose to measure the average training error over equally size blocks of training samples. A block size of 250 worked decently well, and the training graphs are shown in Figure 5. These graphs all show the error over the first 400 blocks of the training process, which is made up of only the first 10,000 training samples. As the graphs show, the models would adjust quickly in the beginning, reaching an average training error of 0.05 in the first 2,500 training samples.

We monitored the networks test set performance after every epoch, and kept the model that had the best accuracy. We may have been able to achieve better accuracy by either using a lower learning rate, or dynamic learning rate, as the error seems to jump around too much at the lower values. Finer adjustments may have allowed us to train better networks, but despite using a static learning rate, our models were still able to achieve 96-98% accuracy, which we found to be sufficient.

VII. Issues Faced

A. Dead Neurons and Weight Initialization

When debugging our implementation of the network, we would sometimes encounter an issue where multiple output vectors would be identical, despite different input images being fed to the network. Once there were a few identical outputs, the network training would quickly halt, and despite feeding more examples through, the weights would stop adjusting.

The underlying issue was all of the values in a given kernel would become negative, including the bias. This meant all resulting convolutions would produce a negative value (since all inputs are non-negative), and so the ReLU operation would rectify them all to 0. Thus, the identical outputs being created were simply the bias of the output layer. The derivative of all the ReLU operations for that kernel would then become 0 as well, causing the gradient to disappear.

This would happen near the very beginning of the training run, and we solved it by changing how we initialized the kernel weights. The weights were being initialized too high, which created massive negative gradients, causing the kernel weights to swing too far negatively. We switched to He Initialization [2], which initializes the weights to have a mean of 0, and standard deviation of $\sqrt{\frac{2}{n}}$, where n is equal to the total number of weights in the kernel, and this helped fix our issue.

B. Overconfidence

One issue observed in our classifier is the overconfidence when there are no feed-forward layers in the network (besides the output layer). In these configurations, the model frequently assigns 100% confidence to the predicted class after softmax, even when that prediction is incorrect. This behaviour is not caused by the softmax function itself, but instead by the large-magnitude logits obtained after the flatten step. This seems to be caused by the absence of sigmoid activations and how the ReLU activations in the convolutional layers may cause some values to explode. The model trained without a feedforward layer can correctly predict 3 or 5 but struggles with 1, 2, and 7, which we think is because the filters alone are enough to discriminate between distinct digits but not enough for these similar looking ones.

C. Slow Training Times

Despite our efforts to optimize our python code, the training runs take a large amount of time. We converted as many calculations into NumPy operations as we could, but we are

Model	Classes									
	0	1	2	3	4	5	6	7	8	9
A	97.39%	98.25%	98.23%	96.49%	97.73%	97.49%	98.32%	96.05%	96.35%	95.98%
B	96.92%	98.86%	96.85%	98.61%	99.08%	98.00%	98.53%	98.13%	99.26%	97.43%
C	96.81%	98.60%	97.17%	97.24%	97.11%	97.51%	98.22%	96.78%	97.64%	98.05%
D	96.82%	99.30%	97.70%	98.41%	98.98%	96.29%	99.36%	98.51%	98.95%	96.49%
E	97.86%	98.85%	96.84%	98.48%	98.27%	96.65%	97.82%	97.09%	96.59%	97.14%
F	97.26%	99.11%	97.47%	98.09%	93.17%	97.09%	97.69%	96.24%	95.07%	96.71%

TABLE II: Precision per Class for Models A-F

Model	Classes									
	0	1	2	3	4	5	6	7	8	9
A	98.98%	98.94%	96.61%	98.12%	96.44%	95.96%	97.81%	97.08%	97.43%	94.65%
B	99.39%	99.38%	98.26%	98.22%	98.17%	98.88%	97.91%	96.79%	97.02%	97.52%
C	99.08%	99.03%	96.41%	97.82%	99.29%	96.75%	97.70%	96.60%	97.74%	94.65%
D	99.49%	99.56%	98.64%	98.02%	98.37%	98.88%	96.76%	96.69%	96.63%	98.02%
E	98.16%	98.24%	97.87%	96.34%	98.47%	96.97%	98.33%	97.47%	96.10%	97.72%
F	97.76%	98.06%	96.90%	96.73%	98.68%	97.20%	97.29%	96.98%	94.97%	93.36%

TABLE III: Recall per Class for Models A-F

still running everything sequentially on a single thread. With deep learning models, we can see why parallelization and using GPU's is incredibly important.

There are only 60,000 training examples in the dataset, which was enough for this application of deep learning, but our implementation would not scale well for larger amounts of data, and larger individual samples at that. We can see how using a deep learning library that can run the model on a GPU could be much more powerful and practical than what we've created in this project.

D. Understanding Backpropagation in Convolutional Layers

In class, we covered CNN's at a fairly high level. We discussed the architecture, and how the kernels could be trained using backpropagation similarly to the fully connected layers. We did not however, go into the exact math for the kernel updates like we did for feed-forward networks, so we had to do some extra research on how backpropagation worked with kernels.

It was surprisingly hard to find good resources on coding CNN's from scratch, and how the learning was performed. Many explanations either used some form of library to aid them, or described the network at too high of a level. The crux of many explanations was they would only use one kernel per layer, which meant their output was always 2D. When you have more kernels however, you have output in 3D, which makes the kernel adjustment and gradient computations much more complicated. If you are interested in learning more about the math behind backpropagation in the convolutional layers, we recommend this two video series, which gave the best explanation we could find (Part 1, Part 2).

VIII. Hands On

Along with our submission are the trained models listed in this report, which can be found in the models/ folder. To run any model against the test set, you can use test.py. This program will run the test set against the model you provide it, and it will calculate the accuracy, precision, and recall of the model.

Example execution:

```
python test.py models/CNN_A.json
```

We've also included a simple drawing program (draw.py), where you can draw your own digits, and have a trained model classify them. This method of testing the models is not scientific, but can give you a feel for which digits each model classifies well, and which digits they do not.

To use the program, use the mouse to draw in the white sketchpad, and click the green check mark to send the sketch to the model, and click the red X to reset the sketch.

The training examples are all centered, and are sized to fill most of the grid. When drawing your own digits, you will see better results when the digit fills most of the box, and is relatively centered. This is just a simple drawing application we made to test the models first hand, and does not do any pre-processing, besides down scaling the image to 28x28.

This drawing program uses the tkinter GUI toolkit. It comes standard with most python installations, but you may have to install it if yours did not.

Example execution:

```
python draw.py models/CNN_C.json
```

If you try models without any sigmoid layers (A, B, D), the confidence of the network will be saturated close to 100%, even when incorrect. If there is at least one sigmoid layer, then the confidence values will vary as you would expect them to. This happens because without any sigmoid activations, there are only ReLU activations, which allow high positive values to "runaway", causing the winning class to dominate the others in the softmax layer.

IX. Conclusion

Coding a CNN from scratch was a frustrating, yet rewarding experience. We gained a deeper understanding of how CNN's work, from how kernel's combine features from multiple channels, down to how the kernel weight adjustments are calculated using backpropagation. The MNIST dataset can

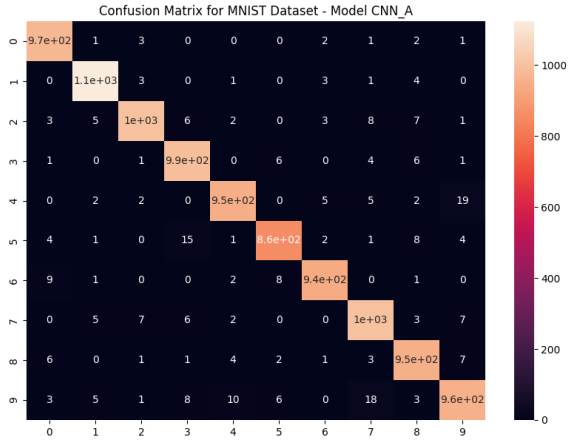
be considered an easier one to work with, but we are very satisfied with the results we were able to achieve from our CNN implementation.

If we were to apply CNN's to another dataset, we would definitely try using available deep learning libraries instead of coding it from scratch, because those libraries have been tried and proven to work. They also include lots of optimizations, and are ready to run your models on GPU's, decreasing training times drastically. For the purpose of learning however, it was worth making a CNN from scratch at least this once.

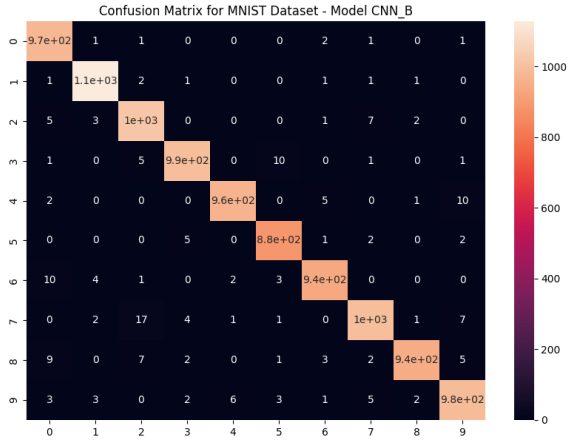
REFERENCES

[1] Y. LeCun, C. Cortes, and C. J. C. Burges, "The MNIST handwritten digit database," 1998, archived: https://web.archive.org/web/20200430193701/http://yann.lecun.com/exdb/mnist/. [Online]. Available: http://yann.lecun.com/exdb/mnist/

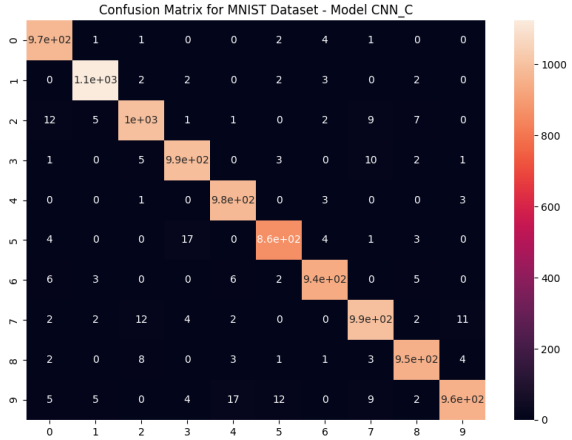
[2] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034.



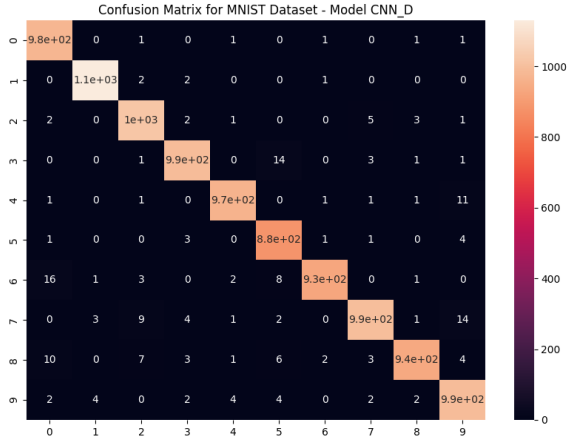
(a) Model A



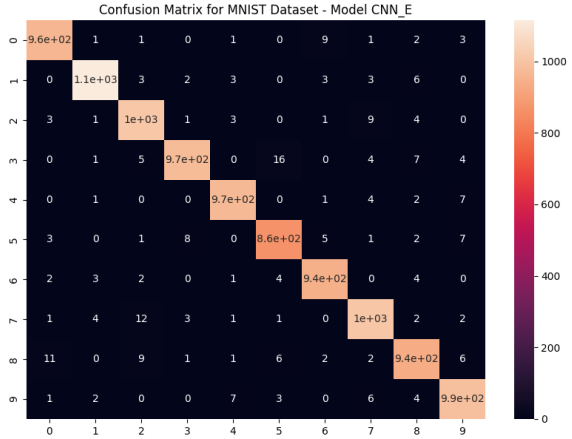
(b) Model B



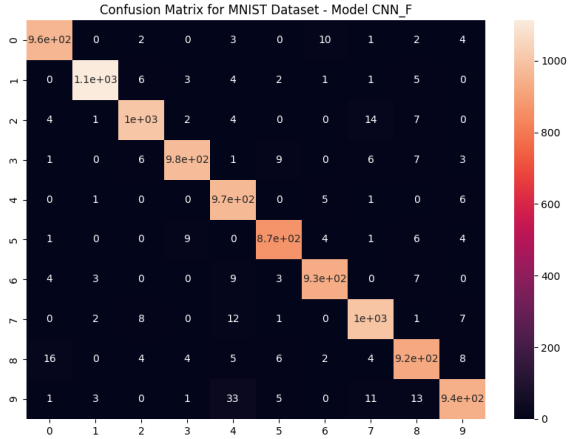
(c) Model C



(d) Model D

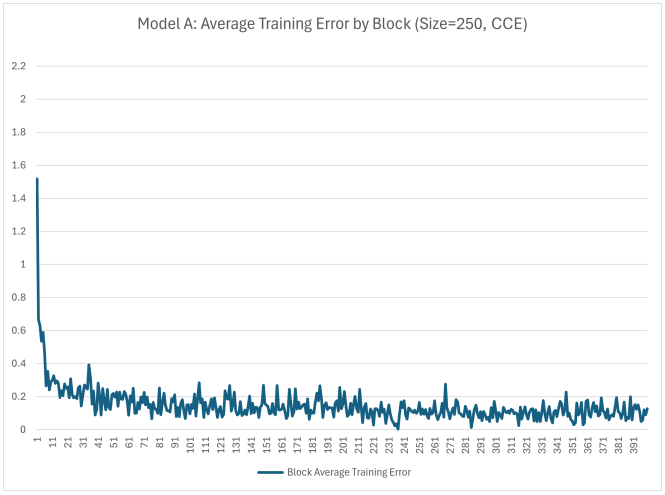


(e) Model E

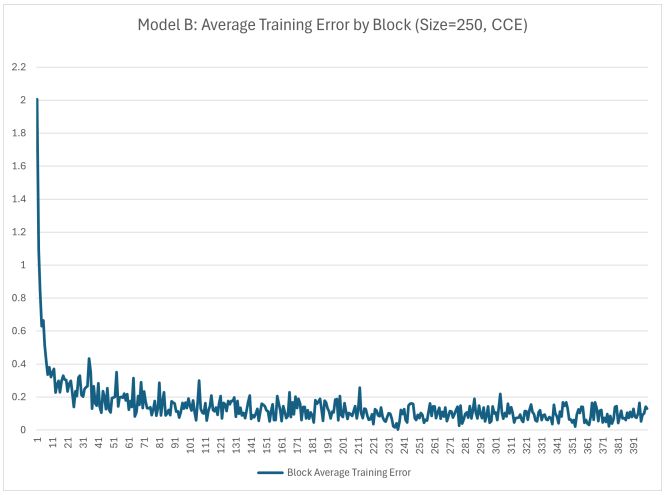


(f) Model F

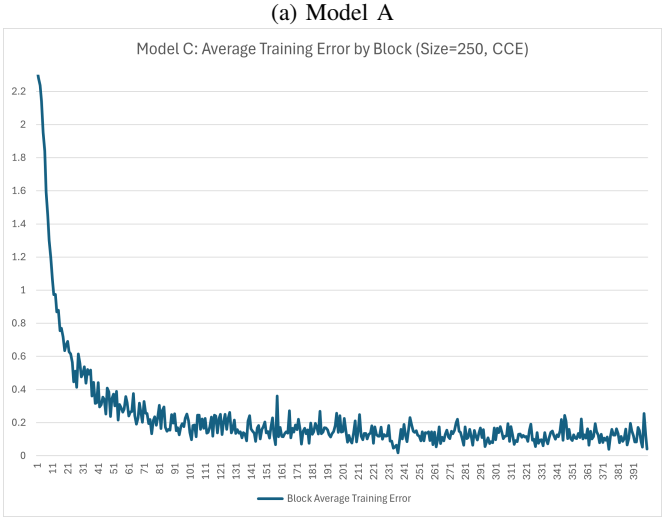
Fig. 4: Confusion Matrix of the Model's Predictions (x-axis) Versus the True Class Labels (y-axis).



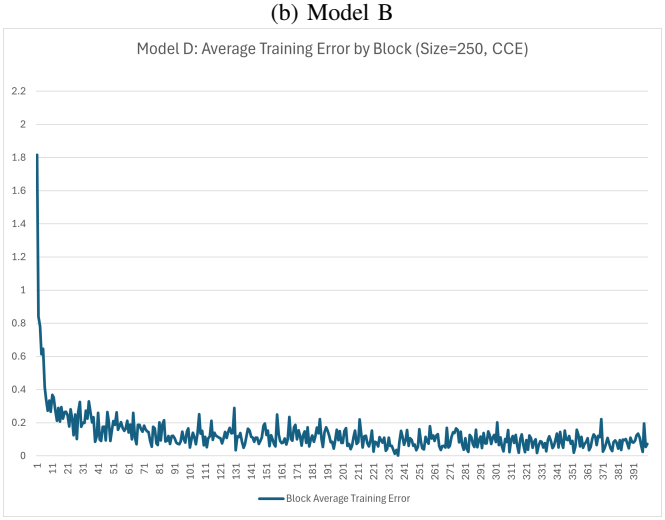
(a) Model A



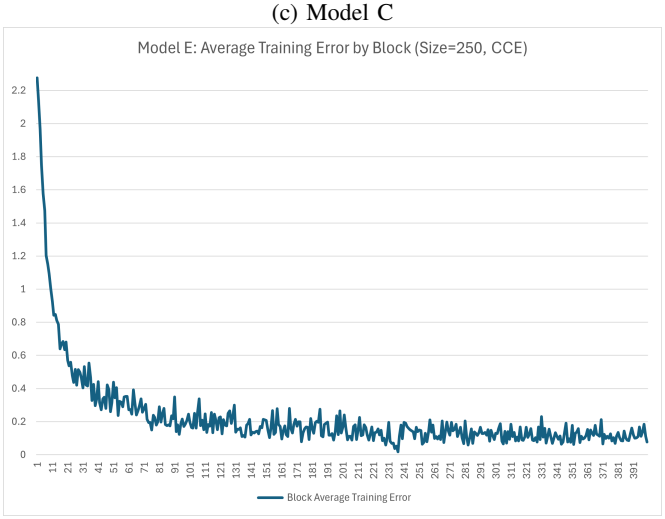
(b) Model B



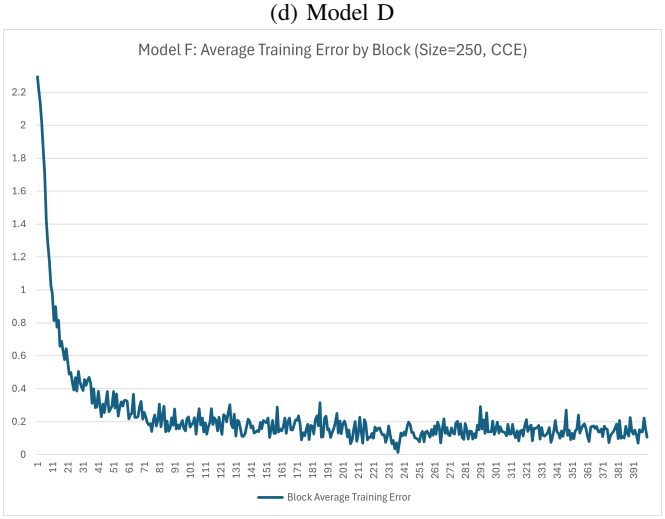
(c) Model C



(d) Model D



(e) Model E



(f) Model F

Fig. 5: Model Training Error by Block (250 Training Samples in Each Block, Categorical Cross-Entropy Loss)