

# Exploring MNIST: From NumPy to Pretrained Models

A Report on Handwritten Digit Recognition Using Modern  
Machine Learning Techniques



Joel Kim

CS\_399: Artificial Intelligence

December 10, 2024

# Table of Contents

<b>Introduction</b>	<b>3</b>
<b>Data Preprocessing</b>	<b>4</b>
<b>Model Development</b>	<b>6</b>
Numpy Model	6
1. Data Preprocessing	6
3. Forward Propagation	7
4. Loss Calculation	9
5. Backpropagation	9
6. Parameter Updates	10
8. Evaluation	11
9. Visualization	11
Conclusion	11
Results:	12
Parameters:	12
PyTorch Model	13
1. Data Processing	13
2. Model Definition	13
3. Loss Calculation	14
4. Optimization and Backpropagation	14
5. Training Loop	14
Conclusion	14
Results:	15
ResNet-18 Model	17
1. Data Processing	17
2. Model Architecture	17
3. Training	18
Conclusion	18
Results:	19
<b>Evaluation and Results</b>	<b>21</b>
NumPy:	21
PyTorch:	21
ResNet18:	21
<b>Conclusion</b>	<b>22</b>

# Introduction

I had been exposed to the usage of neural networks in the past, but never really delved too deeply into it. I'd come across various projects whether it be through online YouTube videos, hackathons, or just speaking to other students. At the time, I was looking more into areas of more data science or more traditional software development. I obviously had some interest in exploring it, but had yet to do so. One video I had watched, referenced briefly the MNIST handwritten digit dataset, and how it is a strong foundation for the understanding of the workings of building a neural network to then train to greatly increase its accuracy. I had never actually worked with the dataset myself, so when presented with the choice of a final project, it made a lot of sense to me to explore this foundational dataset to really solidify my understanding of the subject.

Through this paper, I will be using the MNIST dataset to build three different models. I will lead with some data preprocessing steps that I conducted, which ended up not being that strenuous due to the well organized nature of the dataset I pulled from Kaggle. This was a massive benefit I gained from choosing such a popularized dataset as oftentimes data collection and processing can take up the vast majority of the time for a project such as this. The first model will be created solely utilizing NumPy and Pandas, meaning that I will be specifically neglecting to use preexisting libraries such as TensorFlow or PyTorch. NumPy and Pandas are extremely foundational libraries that often can even be viewed as not extensions because of how foundational both of them are, and specifically their lack of machine learning hyperspecificity allows for more of a bare manual approach to building the model. This allows for me to gain a better understanding mathematically of what is occurring inside the neural network. The second model will then be built utilizing PyTorch. This will allow me to build a stronger model utilizing commonly available libraries that likely I would be leveraging if I were to actually try to build a model for personal or work use. The third model will then be built using the ResNet18 pretrained model. This will likely be the best model, and will give me experience with utilizing a pretrained model, which is extremely useful as it is common to use pre-trained models to save heavily on training time as well as often having a higher accuracy due to the amount of complexity prebuilt into the model. Finally, I'll evaluate the results, draw conclusions, and reflect on my experiences with the project as a whole.

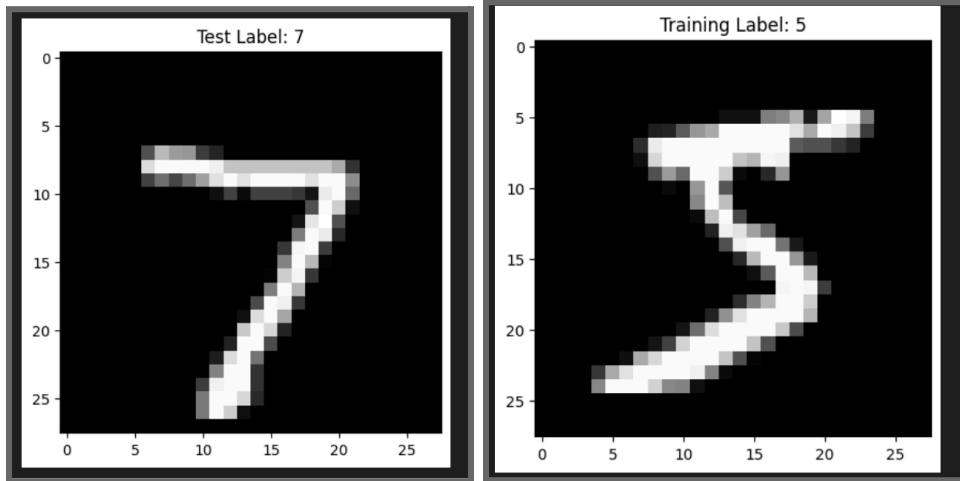
# Data Preprocessing

As stated prior, the dataset I chose was from Kaggle, so it was quite nicely organized already. There were two files, a training dataset containing 60,000 rows of data representing the images, and then a testing dataset containing 10,000 rows of data representing the images. Each of the two datasets has 785 columns of data, this corresponds to 1 column for the label of the image, and then 784 columns representing the 784 pixels of a 28x28 image (the resolution in which the images are stored as). The labels were in the digit range of 0-9, and the other columns were either marked 0 or 1 to denote whether or not the it was black or white due to the images being in grayscale for simplification. This properly allowed the data to be read as a csv. I was able to load in the datasets by utilizing the Pandas library to read the csv files, and then I began working with the data itself.

My initial steps for working with the data include the splitting of the label values versus the pixel values. This was simply as the label was always the first column for the row, with the rest of the columns being the respective pixel values. My next step included the one-hot encoding of the labels. This changes the initial labeling from being an integer 0-9 to instead being an array of size 10 to represent the digits 0-9 respectively. For example, the label "3" would be transformed into the vector [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]. This would be helpful and generally made logical sense as the predictions that the model makes would be in this form, with having a prediction for each of the possible 10 digits. These processes were identical both for the training and test datasets as they were formatted in the exact same way. Once preprocessed, the shape of the datasets was verified to confirm that the training and testing sets had the expected dimensions of [number of samples, 784] for features and [number of samples, 10] for labels.

Finally, as a sanity check, a few examples from both the training and testing datasets were visualized. By reshaping the normalized pixel values back into a 28x28 grid and plotting them, it was possible to confirm that the images still represented their corresponding digits correctly. This step served as a simple yet effective validation of the preprocessing pipeline to make sure I didn't create any errors prior to entering the actual modeling phase. Again, I think it's important to reiterate that this process was made much simpler due to the cleanliness of the dataset that I chose. I had plenty of data to work with meaning I didn't have to spend any time

on data collection, with it nearly being in the perfect form already meaning that the data wrangling was simple as well, and most importantly I was able to trust the accuracy of the data. In my past experiences, I have encountered unclean datasets that I had to spend hours working with or even worse datasets that I thought to be clean, before finding bugs when I began the modeling phases. Generally, this process was held the same for all three separate models, besides some minor variations just to get the data in the correct format.



*Above: The validation check I performed for each set*

# Model Development

I am going to detail the steps that I worked through in order to create the initial NumPy model. Though following that, the process for the next two models will not be as rigorously detailed, but instead the main differences will be highlighted between the three.

## Numpy Model

### 1. Data Preprocessing

For more in-depth details, the data preprocessing section of the report. Though giving a general review, the MNIST handwritten digit dataset comprises 28x28 grayscale images of handwritten digits from 0-9. Each image is represented by a row with 785 columns. The 785 columns represent the 784 pixels of the 28x28 image, and then one column for the label of what digit the image is. The labels were one-hot encoded to instead be a vector, though it still holds the same responsibility of representing what digit the image is. Following these preprocessing steps, the data is in a format that allows for the model to begin being built.

### 2. Model Architecture

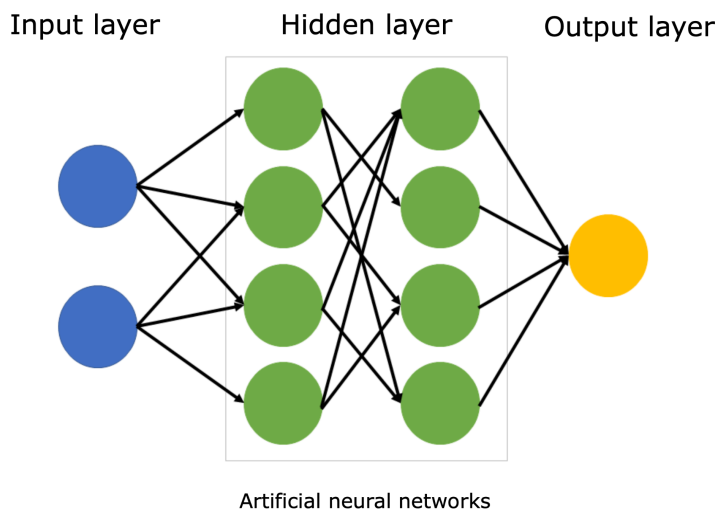
The neural network will consist of 3 main layers. The following are some key terms that will be used in the explanations.

**Weight:** A parameter that determines the strength and direction of the connection between a neuron in one layer and a neuron in the next. Weights are multiplied with input values to influence the output of a neuron, and they are adjusted during training to optimize the model's predictions.

**Bias:** A parameter added to the weighted sum of inputs in a neuron. The bias allows the activation function to shift, enabling the model to represent patterns that do not pass through the origin. It increases the flexibility and expressiveness of the model.

**Neuron:** The fundamental processing unit of a neural network. Each neuron takes in inputs, applies weights and biases, computes a weighted sum, and passes the result through an activation

function. Neurons are organized into layers and collectively learn to approximate complex functions.



*Left: A Simple Diagram of a NN*

1. **Input Layer:** Accepts a vector of length 784 corresponding to the columns representing the pixels of the image.
2. **Hidden Layer:** This hidden layer was chosen to contain 128 **neurons**. In addition to the **weights** and **biases** calculations, their hidden layer was utilizing the ReLU activation function. The purpose of an activation function is to introduce non-linearity, enabling the model to learn complex patterns. Oftentimes, there are multiple hidden layers with a model in order to create more complex models
3. **Output Layer:** Contains 10 neurons, corresponding to the 10 digit classes (0–9). The softmax activation function was utilized, which essentially converts the raw outputs of the hidden layers to then a probabilistic vector to indicate the probability of the image being each of the 10 potential digits.

The architecture was chosen for its simplicity, balancing computational efficiency with sufficient representational power for the MNIST dataset. Greater complexity could've been achieved by various factors like the number of hidden layers or the number of neurons within those hidden layers.

### 3. Forward Propagation

Forward propagation computes the output of the network layer by layer:

**Diagram simplified:**

**Input Layer** -> eq. 1 -> eq. 2 -> **Hidden Layer** -> eq. 3 -> **Output Layer** -> eq. 4

$$Z_1 = X * W_{(input-hidden)} + b_{hidden}$$

This equation (1) represents what is occurring to the values going from the input layer to then the hidden layer. In the equation above,  $Z_1$  is the weighted sum of inputs, while  $W$  and  $b$  represent the weights and biases for the hidden layer.

$$A_1 = ReLU(Z_1) = max(0, Z_1)$$

This equation (2) represents the ReLU activation function. It introduces nonlinearity by removing any negative by changing the value to 0 if it were to be negative. Again, this is to add complexity.

$$Z_2 = A_1 * W_{(hidden-output)} + b_{output}$$

This equation (3) represents what is occurring to the values between the hidden layer and the output layer. It is the same process that is occurring from the input to the hidden layer except for the respective changes in weight and bias.

$$A_2 = Softmax(Z_2)$$

This equation (4) represents the Softmax function that is occurring to put the outputs of the model into a more readable form. This is done by converting the output values to a probabilistic prediction corresponding to what the likelihood of each digit actually is.

The forward propagation step is essential for predicting outputs and is the foundation for computing the loss.



#### 4. Loss Calculation

Following the forward propagation, a loss calculation is then made in order to get a value representing the accuracy and strength of the model. Loss and accuracy are both values often used to determine the strength of a model, but loss gets a value that more strictly investigates how “confident” the model was in its prediction, whereas accuracy is more of a binary in terms of correctness. For example if given an image of a “4”, the model can say that it was 51% sure it was a “4” while only being 49% sure it was a “9”, and despite this resulting in 100% accuracy for that row, the cross-entropy loss function would penalize the model for not being very confident in it being a “4”. This is more optimal for training and building a correct model, as these stricter regulations will create a better model overall. The model employs the cross-entropy loss function, which is well-suited for multi-class classification tasks. The exact details are not necessarily relevant to this report, but the main idea is that it is checking how correct the predictions are, and the goal is to minimize the loss amount.

$$L = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^{10} y_{ij} \log(\hat{y}_{ij})$$

*Above: Cross-Entropy Loss Function for the model*

#### 5. Backpropagation

Backpropagation computes the gradients of the loss with respect to the model’s parameters. This essentially means that by working backwards from the output layer, the neurons (weights and biases of them) are investigated by calculating the gradient of the loss as a result of those weights and biases. Following this, these values can then be utilized to then modify the parameters to limit the overall loss. The process involves:

1. **Output Layer Gradients:** The gradients are propagated backward to update the weights and biases of the output layer.

$$dZ_2 = A_2 - Y$$

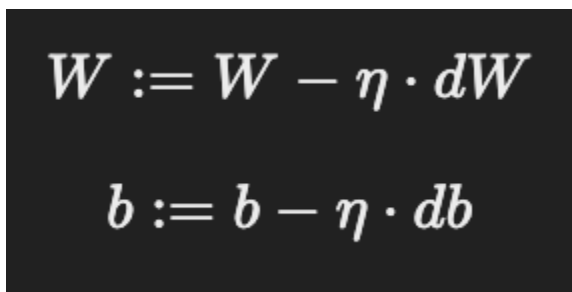
2. **Hidden Layer Gradients:** The derivative of ReLU is used to calculate  $dZ_1$  which contributes to the gradients of the hidden layer weights and biases.

$$dA_1 = dZ_2 * W_{hidden-output}^T$$

$$dZ_1 = dA_1 * ReLU'(Z_1)$$

## 6. Parameter Updates

The weights and biases are updated using gradient descent, which is an optimization algorithm that aims to minimize a function, specifically in this case the loss function. Essentially this is aiming to adjust the model parameters in order to result in a model that has predictions that are more accurate. The entire process of the calculator of the loss and gradient of the loss is part of the gradient descent process.


$$W := W - \eta \cdot dW$$
$$b := b - \eta \cdot db$$

*Left: Equation used to update weights and biases*

The  $\eta$  symbol represents the learning rate. This essentially means how quickly the model will work towards the congruence of the minimum loss. A higher value will have much more drastic changes to the weights and biases value, which sometimes will cause issues.

## 7. Training Loop

This training loop is essentially the steps that are iteratively taken in order to refine the model.

1. Shuffles the training data to ensure randomness in mini-batches.

2. Processes each mini-batch through forward propagation.
3. Computes the loss and backpropagation to calculate gradients.
4. Updates the weights and biases through gradient descent
5. Print results

This loop was performed 20 times, epochs, for the NumPy model. This value could be increased in order to try to refine the model more, though this does run the risk of overfitting due to overtraining on the train data.

## **8. Evaluation**

Following the 20 epochs, the model's performance is evaluated on the test set. The forward pass is used to generate predictions, and the accuracy is computed by comparing the predicted labels to the true labels. This is generally what is used to evaluate how accurate and strong the model is. Additionally, the loss is calculated on the test set to quantify the model's generalization capability. Due to the large amount of testing images, this should generally be enough to have a good idea on how the model would perform on any other unseen data.

## **9. Visualization**

To assess the training process, the loss is plotted against the epochs. This allows me to see how the model is working in terms of its gradual improvement by the reduction of the loss. A downward trend in the loss indicates effective learning. Additionally, being able to see how the model is leveling out at the bottom can indicate whether it is necessary to increase the epoch size if sizable improvements are still being made, or if the epoch count needs to be reduced due to marginal improvements being made.

## **Conclusion**

Ultimately, this model is quite bare because of how everything had to be manually done, so it was aimed to be kept very simple. Though, all in all it performed quite well. Because of the specific exclusion of frameworks like PyTorch or TensorFlow, it resulted in a greater understanding of what was really going on behind the model. It would likely not be smart to

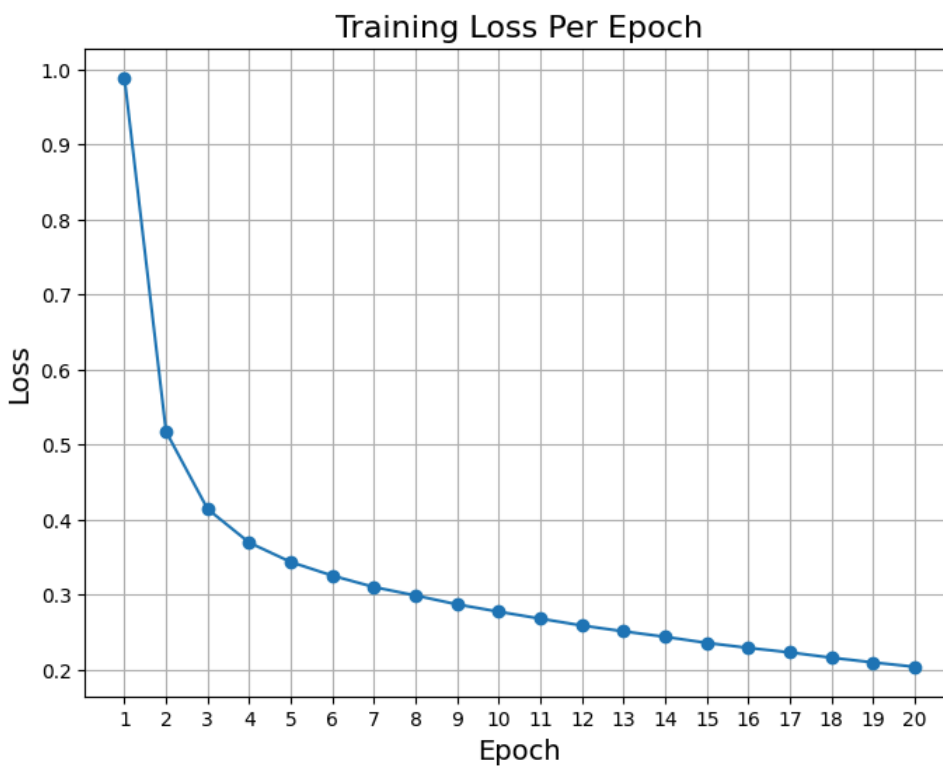
repeat this process for other models if performance is valued more, but for learning this method is quite strong.

**Results:**

Training Loss: 0.2035

Test Loss: 0.2038

Test Accuracy: 94.06%



The model had a solid accuracy of 94%, and it was not overfit as seen by the near equal train and test loss.

**Parameters:**

Weights = Random values between around -0.01 and 0.01

Biases = 0s

Input layer size = 784

Hidden layer size = 128

Output layer size = 10

Epochs = 20

Learning Rate = 0.01

Batch Size = 64

## **PyTorch Model**

This second model was built using PyTorch, a powerful deep learning library. PyTorch simplifies several aspects of model design, training, and evaluation compared to a NumPy-based approach. Utilizing just NumPy forced a lot of the calculations to be done manually, but PyTorch has prebuilt functions that automates a lot of the processes.

### **1. Data Processing**

1. Conversion to tensors: PyTorch required the conversion of the inputs to be in tensors
2. TensorDataset: Pairs features and labels in PyTorch
3. DataLoader: Automates batching and shuffling

These steps allow for more efficient computation, more organized data handling, and parallelized data loading. Specifically, DataLoader helps with the parallelization calculations.

### **2. Model Definition**

In the NumPy model, I had to manually code forward passes, activation functions, and backpropagation. This is all streamlined by using PyTorch. I had to define multiple different functions and declare different variables to set up the model when creating the NumPy model.

Beginning with the forward propagation functions and general architecture, the PyTorch streamlined code is as such:

```
model = nn.Sequential(  
    nn.Linear(28*28, 128),  
    nn.ReLU(),
```

```
nn.Linear(128, 10)
)
```

### 3. Loss Calculation

Instead of the manual computation that occurred in the NumPy model, PyTorch simplifies this by already having a pre-built loss function. Specifically, it is the exact same as the loss function I utilized in the NumPy model:

```
criterion = nn.CrossEntropyLoss()
loss = criterion(outputs, labels)
```

### 4. Optimization and Backpropagation

As opposed to the manual computation from before, PyTorch again automatized this by the *loss.backward()* and the *optimizer.step()* functions. These compute the gradients for the parameters and apply them respectively. I chose to use the same batch-size of 64, and still use gradient descent to try to keep a level of continuity. Though, I also explored using a different optimization technique like Adam, which is stronger typically.

### 5. Training Loop

The training loop is essentially the same. The same steps are taken in terms of the forward pass, computing loss, backward passing, and the updating of the parameters. Though, the main difference is that this code is just a lot cleaner and more readable due to the cleanliness that PyTorch provides.

### Conclusion

The PyTorch implementation is more extensible and scalable compared to the manual NumPy version. The framework's tools allow for ease of prototyping and experimentation. This includes all the prebuilt functions. While the NumPy implementation provides valuable insights into the foundational mechanics of neural networks, the PyTorch implementation demonstrates the power

of leveraging a high-level framework for machine learning. It simplifies much of the complexity associated with manual computations, and this allows for the focus more on the general design for the user. This combination of simplicity, flexibility, and efficiency makes PyTorch an ideal choice for building and deploying machine learning models.

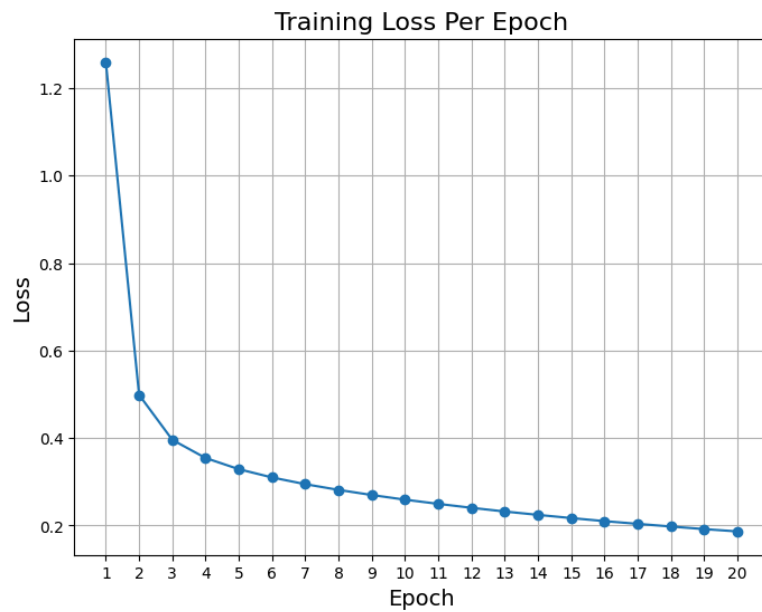
## Results:

### Gradient Descent Version

Train Loss: 0.1865

Test Loss: 0.1829,

Test Accuracy: 94.71%

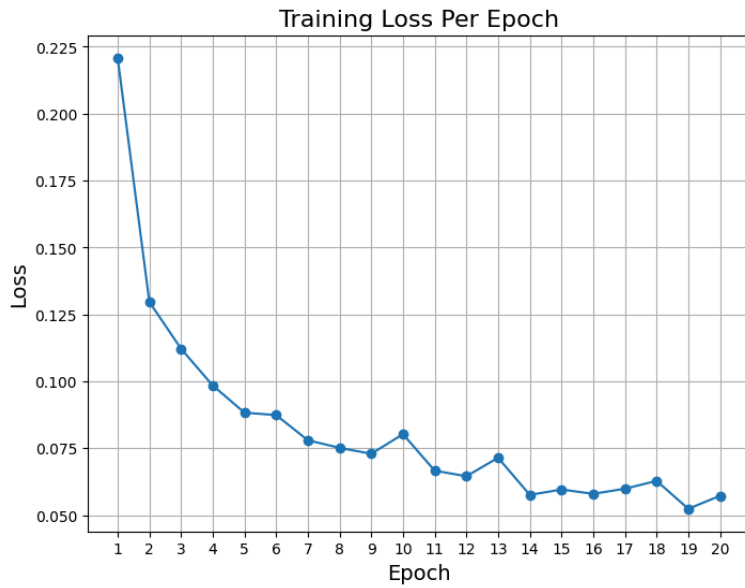


### Adam Version (v1)

Train Loss: 0.0573

Test Loss: 0.3649

Test Accuracy: 96.84%



**Adam Version (v2)**

Train Loss: 0.0682

Test Loss: 0.0918

Test Accuracy: 97.32%



The gradient descent model resulted in extremely similar results to that of the NumPy model, which makes a lot of sense as those models should nearly be identical as the parameters were



aimed to be kept nearly the same. Though the Adam version did suffer from overfitting as the test accuracy did increase to nearly 97%, the test loss was much greater than the train loss. Some sort of regularization, or a lower learning rate could be useful to get the overfitting down. Adam optimization has an adaptive learning rate, so it can start smaller with a lower initial learning rate than a SGD model. Therefore, I then lowered the learning rate to 0.0001 from 0.01, and then compensated by increasing the epoch amount to 30. This resulted in much less overfitting, while still having the 97% accuracy rate. This was a decent way to get rid of the overfitting.

## **ResNet-18 Model**

The third model was built by utilizing a pre-trained model known as ResNet-18. This is a deep convolutional neural network that is pre-trained on more than a million images from the ImageNet database. It is 18 layers deep, and is more traditionally used for object classification, but can be modified to work for digit recognition as well. There only needs to be some small changes made to apply the pretrained model for our situation, and the results were a great improvement.

### **1. Data Processing**

There were no real changes needing to be made to the data.

### **2. Model Architecture**

While the PyTorch model I created was a simple 3 layer neural network with an input, hidden, and output layer, with total parameters being quite small due to the size not being that big, the ResNet18 model is a deep convolutional neural network that includes many different layers. This includes convolutional layers that work to extract certain features, pooling layers that combine surrounding sections of simplification, and a fully connected output layer. In total there are over 11 million parameters as opposed to the 100k or so parameters in the PyTorch model. One key aspect is that the ResNet18 model is actually considering the spatial awareness of the image structure, since the flattening kind of goes against that idea.

The main changes in the code are as follows:

```
resnet = models.resnet18(pretrained=True)
resnet.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
resnet.fc = nn.Linear(resnet.fc.in_features, 10)
```

These lines are responsible for loading in the dataset, adjusting the input layer, and then adjusting the output layer. The adjustment of the input and output layers are how the pretrained model is able to be implemented.

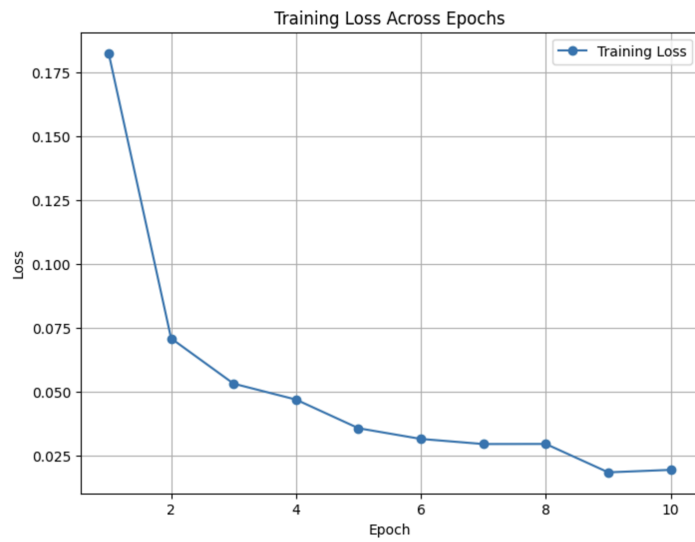
### **3. Training**

The big difference in training is that the PyTorch model was starting from scratch, meaning that the parameters were generally randomly initialized. It had to train from nothing in order to reach its peak. Furthermore, the training was simpler as it was a forward pass through only one layer. This heavily differs from the Pretrained ResNet18 model as that comes in already trained on a massive dataset, so it already knows how to properly handle images. It only needs to train slightly to perfect it for the specific usage of the MNIST dataset. Though the general epochs are slower because it is more complex, because it is already pretrained, it requires less epochs than the PyTorch model.

### **Conclusion**

Overall, the ResNet18 model was quite simple to implement. The main issue was transferring my code into Google Collab in order to have access to a GPU to run the more complex model. There exists less flexibility with the model since a lot of the layers I don't have any control over, but the ResNet18 model is already amazing at 2D image recognition. If I were to implement a model for any type of image recognition, I would very likely start by using the ResNet18 model as that is likely the best way to get a high accuracy. I would be trading off much understanding and customization, but if the goal is an accurate model then it is a fair trade.

## Results:

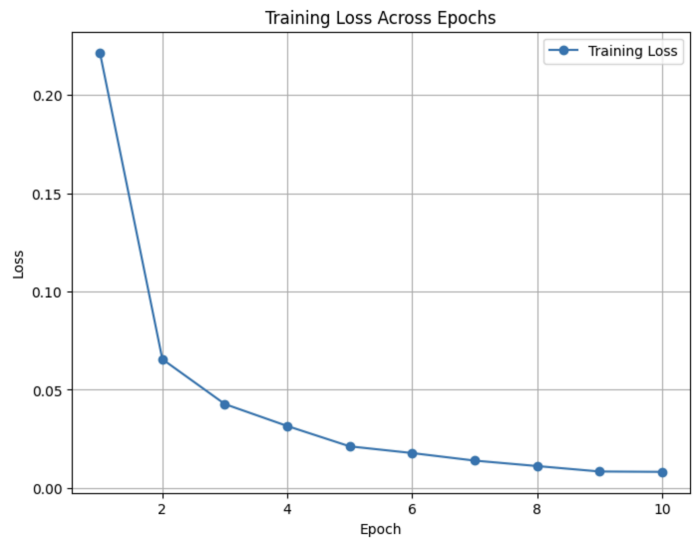


ResNet-18 (Adam)

Train Loss: 0.0193

Test Loss: 0.0302

Test Accuracy: 99.16%



ResNet-18 (SGD)

Train Loss: 0.0082

Test Loss: 0.0322

Test Accuracy: 99.12%

There wasn't much difference between the SGD and Adam optimization for the ResNet-18 model. Both models had very similar accuracy at 99%, and quite low test loss. Though the SGD model was slightly more overfit compared to the Adam model. It is key to note the lower amount of epochs needed for these models being at only 10. It is likely I could've also just done 5 epochs judging off of the graphs, and that shows the strength. Although these epochs did take longer than that of the other models, it required less because of all the prior training that it benefits from.

# Evaluation and Results

Comparing the “best” version for each model that I created gets these results.

## **NumPy:**

Training Loss: 0.2035

Test Loss: 0.2038

Test Accuracy: 94.06%

## **PyTorch:**

Train Loss: 0.0682

Test Loss: 0.0918

Test Accuracy: 97.32%

## **ResNet18:**

Train Loss: 0.0193

Test Loss: 0.0302

Test Accuracy: 99.16%

A main point I would like to make is that the goal was to not really adjust the hyperparameters to get the best model for each of these types. The main focus was to analyze the general structure of each of the models and then compare them. It can be understood that improvements can be made to each model in order to enhance their performances.

# Conclusion

Overall, I am extremely happy with the project as a whole. I accomplished everything I had set out to do in terms of building three different models in three different ways to learn three different things. I think I was able to draw valid conclusions about the pros and cons of each of the models making learning all three a useful tool. If I were to suggest a model to build for a beginner CS student, I would recommend building it using NumPy to understand the math of what's going on. Past that, I think students should genuinely be using a framework like PyTorch to experiment with different types of models with different layers and hyperparameters to see how that affects results. Though, I do see value in learning how to use pretrained models as if a job or role really needed a strong model for a task, utilizing a well-built and trusted model by professionals such as ResNet18 makes a lot of sense in terms of time and value. The MNIST dataset is a great one to work with as it is simple to understand and use, but you can do some great things with it. Some next steps would be to explore different datasets to see how these results compare. This might include sourcing my own data, which would require a lot more work on the preprocessing side. Though in conclusion, my project was able to effectively explore three different ways to build a neural network, and thoroughly report the differences between them to communicate the value of understanding all three.