Tina Tran

COT 3100H Spring 2023

A Deep Dive into Neural Networks and Handwritten Digit Classification

Introduction

Handwritten digit classification is an extensively studied problem within the field of machine learning (ML). Some refer to creating an image processing model that performs on the MNIST database (Modified National Institute of Standards and Technology) as the “Hello World!” or starting point when diving into ML coding. The MNIST database is a large database of handwritten digits used in the training and testing of image processing systems. It contains 60,000 training examples and a 10,000 example test set of monochrome images sized 28x28 pixels [1]. The specific image processing model will be a neural network (NN) – named such because the structure and functions mimic a human neuron. Multiple layers contain “neurons” or functions within the computer program that process data and then pass it to the next function. When given a large database such as MNIST, a NN can extract a learned connection to be applied for the problem at hand. The problem to be solved is to accurately recognize a digit 0-9 when given a square, monochrome image of a handwritten digit using a NN trained on the MNIST database. First, the problem will be analyzed in detail. Next, the NN will be built using Python. A model will be built from scratch using only the NumPy library and a second model built using the PyTorch library. NumPy is a library including support for large arrays and matrices and is necessary for fast computation [2]. PyTorch is an ML framework based on the Torch library developed by Meta AI and is used for building deep learning models such as NNs [3]. The two models will be tested and compared to find which method will return more accurate results. The initial hypothesis is that the model implemented using PyTorch will have a higher accuracy in performance since the library was created specifically for the purpose of optimizing a model for simple tasks within ML. The results will also be analyzed to determine how each method can be improved. Finally, a procedure to implement NNs for any discrete problem like MNIST will be discussed as well as other methods within ML to explore.

Literature Review

egcode. “MNIST-To-CSV/MNIST Convert.ipynb at Master · Egcode/MNIST-To-CSV.” GitHub, 2023, github.com/egcode/MNIST-to-CSV/blob/master/MNIST%20Convert.ipynb.

This GitHub repo code was edited and used to implement a converter within the file “MNIST\_to\_CSV.py” to extract data from the raw MNIST database data and create a CSV (Comma Separated Values) file which is easier to read and work with for the NN from scratch implementation.

“How to Build a Neural Network from Scratch with PyTorch.” FreeCodeCamp.org, 15 Sept. 2020, [www.freecodecamp.org/news/how-to-build-a-neural-network-with-pytorch/](http://www.freecodecamp.org/news/how-to-build-a-neural-network-with-pytorch/).

This guide was used in coding a NN with PyTorch.

“Neural Networks from Scratch.” *Nnfs.io*, nnfs.io/.

This textbook was used in coding a NN from scratch using NumPy.

“Python - How to Read and Display MNIST Dataset?” Stack Overflow, stackoverflow.com/questions/71175143/how-to-read-and-display-mnist-dataset#:~:text=import%20numpy%20as%20np%20import%20csv%20import%20matplotlib.pyplot.

This post on Stack Overflow discussed code that was edited and used to read the MNIST dataset within the file “visualize\_data.py” for the multiple functions implemented.

Robinson, David. “Exploring Handwritten Digit Classification: A Tidy Analysis of the MNIST Dataset | R-Bloggers.” R-Bloggers, 22 Jan. 2018, [www.r-bloggers.com/2018/01/exploring-handwritten-digit-classification-a-tidy-analysis-of-the-mnist-dataset/](http://www.r-bloggers.com/2018/01/exploring-handwritten-digit-classification-a-tidy-analysis-of-the-mnist-dataset/).

This blog post was used as a guide and starting point for data analysis of the MNIST database. The functions implemented within the file “visualize\_data.py” and data gathered were inspired from the work within this post.

Tran, Tina. “Neural-Net-Practice.” *GitHub*, 3 Feb. 2023, github.com/TTrumpet/Neural-Net-Practice.

Prior work on implementing a NN in PyTorch trained on the MNIST database.

Tran, Tina. “TTrumpet/Neural-Network-From-Scratch.” *GitHub*, 18 Dec. 2022, github.com/TTrumpet/neural-network-from-scratch.

Prior work on implementing a NN from scratch trained on the MNIST database.

Handwritten Digit Classification Problem

The problem of handwritten digit classification is within the “discrete environment” of Artificial Intelligence, meaning that there is a finite number of possibilities. The handwritten digit can only be 0 through 9, with ten possible digits. The probability of a computer program to correctly recognize the digit by guessing randomly without any training is . In order to increase the accuracy of the computer program, a model will be trained on MNIST’s 60,000 training examples. Figure 1 through 5 show an image representation of the first 5 training examples within MNIST. The images and any further images were created using the Matplotlib library [4] and the code can be found in “visualize\_data.py” as the function “display\_image”.

Graphical user interface

Description automatically generated with medium confidenceChart, histogram

Description automatically generated

Figure Figure

Chart

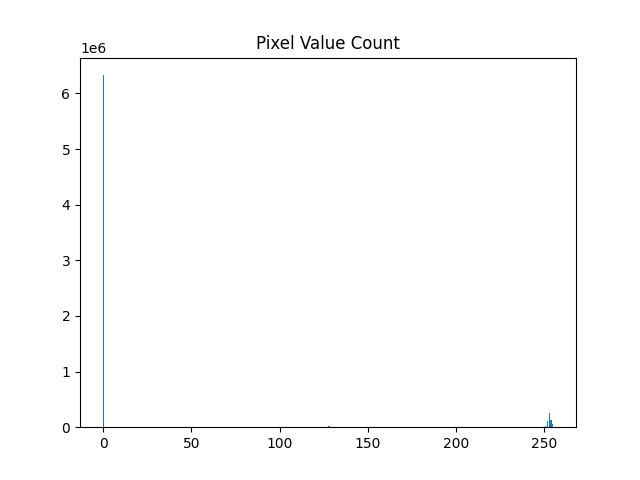
Description automatically generated Chart

Description automatically generated Chart, histogram

Description automatically generated

Figure Figure Figure

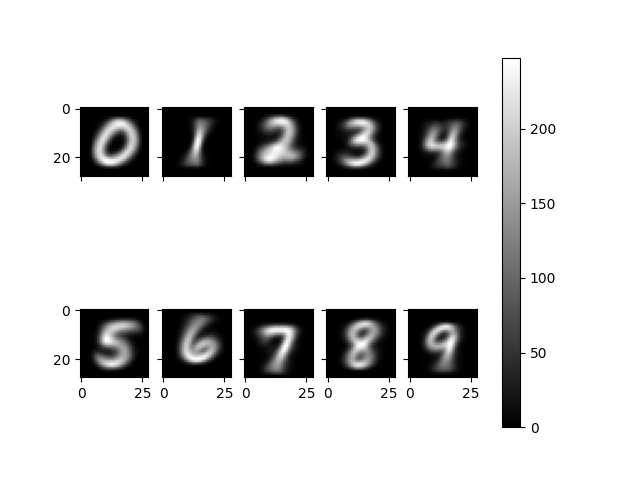
As seen from the Figures above, the images display black pixels corresponding to the “blank space” number 0 and white pixels corresponding to the number 255\*. Values in-between 0 and 255 are grey. Within all the images in MNIST, most pixels are completely white or completely black with relatively few grey pixels. Figure 6 is a visual of this fact, refer to “visualize\_data.py” within the function “display\_pixel\_data”.



Figure

Thus, we can normalize the data without losing much accuracy. Normalizing refers to representing data as either 0 or 1, like the concept of a bit within computer programming. A bit represents a logical state with one of two possibilities - either turned off (0) or on (1). This technique simplifies the process of learning since the data only has two possibilities (0 or 1) versus having 256 possibilities (0 to 255). One-hot encoding is also used to simplify each training example into a “string” to represent all combinations of the normalized data. Referring to the code within “nn\_scratch\_test.py”, when the data is loaded, the CSV file is converted using a normalize and one-hot encoding function to simplify and quicken the training process as seen in the function “load\_data”. Blank space is represented using 0 and pixels with original value of 255 are represented using 1. The positions where the value 1 is located corresponds to each digit will be what the NN will be learning.

Another consideration for the data is the variability of each digit label. In other words, how different is an image from another image that both have the same label? To visualize this, refer to the Figure 7 visualization below and the code within “visualize\_data.py” in the function “find\_centroid”. A centroid is an averaged image where each image with the same label was averaged to find which pixels were always the same on each image.



Figure

The centroid images already lead to some suspicions on which digits may be easier to distinguish, the 0 and 1. Digits that look similar to each other such as 4 and 9 or 3 and 8 will have more overlap and thus will be more challenging to distinguish. The variability per digit can further be visualized using Euclidean distance, an algorithm to find how different each image is to its corresponding centroid. Figure 8 is a box and whisker plot representing the Euclidean distance to each digit’s centroid, or the variability within each digit. The code can be found within “visualize\_data.py” as the function “display\_centroid\_euclidean\_distance”.

Chart, box and whisker chart

Description automatically generated

Figure

With Figure 8, the Euclidean distances of the images of label 1 are considerably less than the other digits. This makes sense, since the ways that a person can write a 1 is less than the other digits – a simple straight line or the fancy way with a bar on the bottom seem to be the only two ways that come to mind. On the other hand, the other digits such as 7 have multiple ways to be written such as with a line through the middle, a hook at the upper left corner, and sometimes with both or neither of the prior two.

With the handwritten digit problem, the MNIST database was discussed and visualized. The code utilizes normalization and one-hot encoding to simplify the data. Finally, the problem of variability in the handwritten digits within each label was acknowledged. Next, an in-depth discussion on NNs and the learning process will further connect these concepts together.

Neural Network from Scratch

A NN works like the human brain in creating a connection between data. The goal of a single neuron within an NN is to compute and learn a weight number and a bias number where the input data multiplied by the weight and added by the bias will lead to the desired output. The dot product of the outputs of each neuron and the matrix product of each layer of neurons lead to the full output of the NN. A neuron computes these weights and biases through training, a process with 3 steps: a forward pass, a backward pass (backpropagation), and gradient descent.

A forward pass is simply passing the data through a predetermined function known as an activation function. The most used activation functions are the step, linear (linear forward), sigmoid, Rectified Linear Activation Function (ReLU), and Softmax. The activation functions utilized in the NN from scratch can be found in “nn\_scratch\_test.py” in the “forward” function. Linear forward, ReLU, and Softmax are used, thus those will be defined. Linear forward or just linear is defined by , thus the data input will be the same as the data output. ReLU is defined by , or in code: np.maximum(0, inputs). The data that is inputted will be the same output as long as the input data is greater than zero, else the data returned will be zero. The dReLU activation function seen defined under ReLU within the code will later be seen in the next step of backpropagation. The “d” represents the derivative of ReLU and is defined by , or in code: 1 \* (x > 0). Finally, the Softmax function is defined as , or in code:

exp\_values = np.exp(inputs – np.max(inputs, axix=1, keepdims=True)   
probabilities = exp\_values / np.sum(exp\_values, axis=1, keepdims=True)

Softmax takes a vector of raw outputs and returns a vector of probability scores. In other words, if Softmax returns a vector [0.0, 0.0, 0.2, 0.4, 0.2, 0.1, 0.1, 0.0, 0.0, 0.0], there is a 0% chance the image passed to the NN is a handwritten digit of 0, 0% the image label is 1, 20% the label is 2, etc. The greatest possibility for the made-up example is that the digit label is 3, so the NN will label the image as a 3 and compare it with the actual label to see if the guess is correct. This process leads to the second step in the training process: a backward pass or backpropagation.

A backward pass starts with the comparison of the output found in the forward pass. A function called loss or cost is calculated to see if the “guesses” made in the forward pass are leading to a more accurate result. The loss function is defined by or , where k is an index of “true” probability or in code: -np.log(output). The partial derivatives of the outputs are calculated and compared with the loss to see each part’s role in the goal of minimizing the loss (and increasing accuracy). Refer to “nn\_scratch\_test.py” in the “backprop” function for the full code implementation. This step corresponds with the logical step of finding connections in the data. If the NN finds a connection that accurately represents the data, the function that leads to this prediction will not be altered, while an incorrect prediction will be severely altered to reduce the loss.

Finally, gradient descent is the last step. This step uses a predetermined learning rate to find the local minimum of a function by taking repeated “steps” (subtraction) in the opposite direction of the gradient at the current point. Referring to “nn\_scratch\_test.py” at the end of the “backprop” function, the subtraction of the learning rate here is the gradient descent step, the “steps” being formally defined as the learning rate, a constant small number (0.001) that was kept the same for the NN from scratch and the NN using PyTorch. The steps towards a “local minimum” is essentially finding the path towards minimizing loss and maximizing accuracy.

The NN from scratch utilizes these three steps with a three-layered model, taking the input of 784 nodes (one node per pixel in the 28x28 image) and reducing it to 256, second layer reducing to 128, and final layer reducing it to 10 corresponding with the 10 digits 0-9.

Other key terminology includes batch size, the number of pictures per batch (64); and epochs, the number of runs through the data (80). These two variables like learning rate were kept the same in the two NN implementations. The constant numbers of batch size, epoch number, and learning rate are part of a group known as hyperparameters.

Neural Network using PyTorch

The NN using PyTorch utilizes a simpler two-layered model, reducing 784 to 50 and then to 10 in the second layer. Without the increased layer in the NN from scratch, the results would be more disparate between the two due to the easy implementation of optimizing functions.

The optimizing functions are defined though the use of criterion and optimizer. These two variables which can be found in the “pytorch\_test.py” are functions defined in PyTorch. Criterion is defined by cross entropy loss, a loss function that can calculate the cross-entropy between two probability vectors to get a more accurate loss value to be minimized. Optimizer is defined by the Adaptive Momentum (Adam) optimizer, an addition to gradient descent that calculates the first order and second order derivatives to optimize step movements. These differences lead to a disparity in the accuracy between the two models, which will be explored in the results section.

Running\*\* and Testing

The initial test consisted of the same conditions stated above, with a change of 100 epochs. A standard practice in ML is to initially test with many epochs since a phenomenon known as “false minimum” can occur where loss is minimized to a local minimum rather than the global minimum. The following graphs were made using the Weights & Biases library wandb tracking system [5]. Figures 9 and 10 show the accuracy and loss for the NN from scratch after 50 epochs and Figures 11 and 12 show the accuracy and loss for the NN using PyTorch after 50 epochs. Although these Figures may seem like the accuracy and loss are reaching a plateau, when compared with the 100-epoch iteration (Figure 13, 14: NN from scratch; Figure 15, 16: NN using PyTorch) there was a drastic increase in accuracy in the NN from scratch around epoch 50 and a dip around epoch 65 for the NN using PyTorch. This led to a change in the epochs from 50 to 80 across the two implementations to account for a possible “false minimum” run. This occurs when the gradient descent algorithm falls into a local minimum when minimizing the loss – believing the loss has been minimized to the lowest possible value – and leading to a stall in the overall accuracy as the global minimum is not reached. In addition, the 80 epochs avoid two phenomena known as “underfitting” and “overfitting”. Underfitting is when the data creates weights and balances that do not accurately fit the data as much as they are possibly able to do. This phenomenon is easier to avoid, since usually adding more epochs will lead to increased accuracy. However, overfitting must be avoided, where the data creates weights and balances that accurately fit the training data but are not applicable to the testing data. If the epoch number is increased to an astounding number such as 200, overfitting could occur. The NN will essentially establish a connection between the images within the 60,000-training set only. For example, there could be many instances of the digit 0 written as circles within the training set. The NN could falsely assume that all 0 must be written as a circle instead of an oblong shape, causing the accuracy to decrease on the testing set. Thus, an epoch number must be chosen that is not too low or too high to avoid these two phenomena.

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

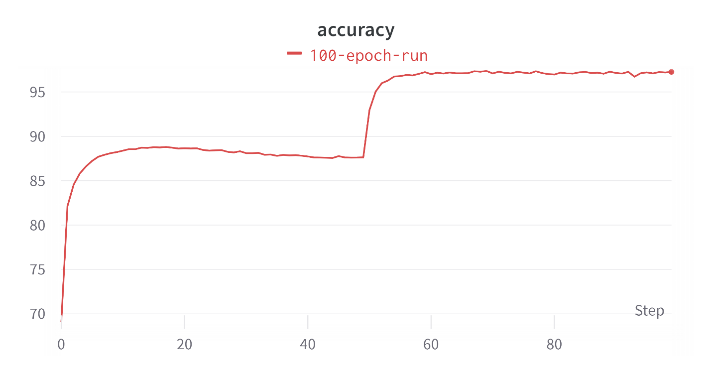
Figure Figure

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

Figure Figure

Chart, line chart

Description automatically generated

Figure Figure

Chart

Description automatically generatedChart

Description automatically generated

Figure Figure

Initial Results

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

Figure Figure

Chart

Description automatically generatedChart, line chart

Description automatically generated

Figure Figure

Figures 17 and 18 depict the accuracy and loss of the NN from scratch. Figures 19 and 20 depict the accuracy and loss of the NN using PyTorch. Each NN was trained a total of 10 times each with 80 epochs. The NN from scratch achieved an average of 91.6587% accuracy and 0.014057 loss. The NN using PyTorch achieved an average of 97.224% accuracy and 0.00026966 loss. Findings show that there is a possibility of the NN from scratch achieving a result comparable to the NN using PyTorch (See Figure 13). An explanation on the reasoning behind the large range of cases for the NN from scratch (Between 88.157 on Figure 17 and 97.27 on Figure 13) is the “false minimum” phenomenon mentioned previously. In rare cases, the NN from scratch can avoid a false minimum such as in Figure 13 and obtain a greater accuracy comparable with the NN using PyTorch. The NN using PyTorch is more likely to climb out of or even avoid a false minimum due to the optimizers. The hypothesis that the NN using PyTorch would have a greater accuracy was proven to be true in most cases, even with the NN from scratch having more layers than the NN using PyTorch because of the easy to implement optimizers to avoid a false minimum result. An NN from scratch could likely reach 97% accuracy if the Adam optimizer were implemented, however time and space are concerns in this situation since optimizers can be complex to implement from scratch.

Further Testing

To increase the accuracy of the NN from scratch, one attempt was made by the constant learning rate was changed from 0.001 to 0.05. The reasoning behind this decision is that the steps taken by the original NN from scratch were much too small and increased the likelihood it converged towards a false minimum. A greater learning rate would increase the chances of skipping over these false minimums towards the global minimum closer to the 97% accuracy shown in the NN using PyTorch. Figures 21 and 22 show that fine tuning constants such as the learning rate can have an impact on the accuracy and loss, with an average of 95.2856% accuracy and 0.0071157 loss over 10 runs. Even with this improvement, the NN using PyTorch still has achieved greater accuracy overall.

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

Figure Figure

Conclusion

The MNIST problem has been solved with the highest accuracy of about 97% in this paper. Other models have been used to further push this accuracy higher such as Deep NNs which are NNs with many more layers and neurons. For example, Cires¸an et al. used a Deep NN with 6-10 layers and many more neurons trained on MNIST with a 99.8% accuracy [6]. Another method is a Convolutional Neural Network (CNN), a NN used mainly in image processing due to the use of a specific convolution layer to extract features from the image itself. CNNs have produced a high accuracy of 99.79% on MNIST.

To conclude, general procedure tips for solving any discrete problem like MNIST using NNs will be discussed. First, researching the data is paramount. When creating the NN layers, the first layer should have an input of the number of initial input data (e.g., 784 for the number of pixels in each picture) and the last layer should have an output of the number of possibilities (e.g., 10 digits). Next, fine tune the hyperparameters such as epochs. Try testing with a large epoch number and fine tuning of the epoch number as desired. Finally, if greater accuracy is desired, implement optimizers. Optimizers will help in avoiding a “false minimum” with gradient descent and lead to training of correlations to solve the desired problem.

Citations

[1] “MNIST Handwritten Digit Database, Yann LeCun, Corinna Cortes and Chris Burges.” *Lecun.com*, 2009, yann.lecun.com/exdb/mnist/.

[2] Numpy. “NumPy.” *Numpy.org*, 2009, numpy.org/.

[3] PyTorch. “PyTorch.” *Pytorch.org*, 2019, pytorch.org/.

[4] Matplotlib. “Matplotlib: Python Plotting — Matplotlib 3.1.1 Documentation.” *Matplotlib.org*, 2012, matplotlib.org/.

[5] “Weights & Biases – Developer Tools for ML.” *Wandb.ai*, wandb.ai/site.

[6] Cireşan, Dan, et al. “Multi-Column Deep Neural Network for Traffic Sign Classification.” *Neural Networks*, vol. 32, Aug. 2012, pp. 333–338, https://doi.org/10.1016/j.neunet.2012.02.023. Accessed 25 Mar. 2019.