MNIST Handwriting Training on Perceptrons

Eckel, TJHSST AI2, Spring 2023

Background & Explanation

Now that we've completed several elementary exercises using Perceptron networks, it's time to train them to accomplish something that would be extremely difficult to program intentionally. Specifically, the MNIST data set is a set of 28x28 pixel images of handwritten individual digits, all centered and normalized. We want to train a network to look at each image and determine which digit has been written. I find it absolutely *incredible* how well this works!

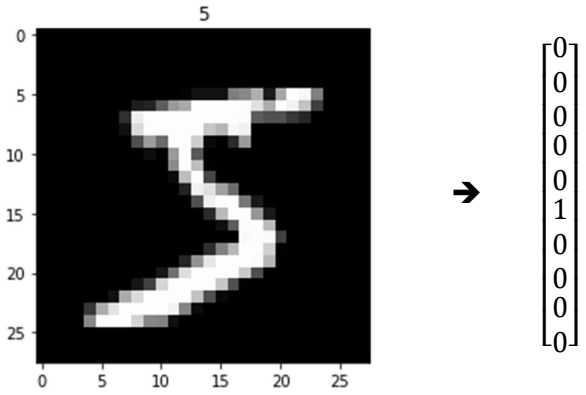
Input Layer

The input layer, while large, is at least rather simple – this is a Perceptron network where the input layer (layer zero) contains one input per pixel. This means the network has 784 (28x28) total inputs! Each pixel in the data set is a value from 0 to 255 representing a grayscale weight. These values have been reversed from reality – the background is black (0) and the handwriting is in white (255). This makes signal easier to distinguish from background; the irrelevant squares have low values instead of maximum values. **IMPORTANT:** Since sigmoid perceptron networks operate exclusively on values between 0 and 1, we'll want to take the values in the data set and divide all inputs by 255 to get input values in that same range.

Output Layer

The output layer requires more thought. We have 10 different classifications (the digits 0 to 9), so we'll need to use a slightly different strategy. We'll want an output layer of 10 Perceptrons, one for each possible digit. When we make the training set, the correct digit will be marked with a "1" and all the others will be marked with a "0". Every image in the training set will be paired with an output vector with a single "1" and a bunch of "0"s.

For example, this is one image stored in the MNIST database, an example of a handwritten "5", paired with its ideal output matrix – a single "1" at index 5 to match the classification, and all the rest are zeroes:



IMPORTANT: After the network is trained, we **will not** use the same strategy as before (round everything above 0.5 to a 1, round everything below to a 0). We will simply find **the output perceptron with the greatest raw output value (without rounding)**, and that will be the answer. So, for instance, if this is the output we get from our trained network:

[[0.6965712] [0.1813661] [0.9751193] [0.1747664] [0.5417504] [0.1870623] [0.0387748] [0.72784] [0.1338721] [0.6108751]]

…then we would say our network classifies that input vector as a "2", since the output value at index 2 is the highest, even though several other values are greater than 0.5.

THIS WAS THE MOST COMMON MISTAKE on this lab last year; many, many students simply rounded the output and compared the resulting vector of 1s and 0s directly with the desired output, causing any output with two values greater than 0.5 to appear misclassified even if the highest output value was in fact the correct classification!

Interior (Hidden Layers)

The interior of the network is up to you. Put in as many layers as you'd like, with a number of perceptrons that you want to try. Generally speaking, the most common strategy is to use layers of decreasing size – something like 784 inputs, a hidden layer of 300 perceptrons, a second hidden layer of 100 perceptrons, and a final output layer of 10 perceptrons. If you'd like more ideas for networks to try, see the links in the requirements section!

Requirements

1. Visit the links on the course website for information and to download the data sets.

2. Write code that reads in **the training set** (ignore the test set for now) and creates a properly formatted data structure to use (pairs of a 784x1 input matrix and a 10x1 output matrix with a single 1). Maybe pickle it?

3. With such a large network, we'll also need code that generates a whole lot of random weight and bias values to a certain network specification so we can train from there. So, we'll want a method that takes a network architecture as an argument and returns a list of randomly-initialized weight and bias matrices. Having a method like this will make altering your network architecture much easier – you'll just have to change what you pass to this method, and then you'll get appropriate matrices that you can directly pass into your training code!

   Specifically, the argument passed into this network generator function will be a list with the number of nodes at each layer in the network. For example, if we pass the list [2, 4, 1] into our method, it will return a list of two weight matrices – a 4x2 and a 1x4 – and a list of two bias matrices – a 4x1 and a 1x1. Or, more appropriately for this assignment, if we pass the list [784, 300, 100, 10] (which is one possible option for this assignment, just as an example) it would return a list of three weight matrices – a 300x784 and a 100x300 and a 10x100 – and a list of three bias matrices – a 300x1 and a 100x1 and a 10x1.

   Note that numpy makes it quite easy to initialize random matrices; there's an example of this in the numpy demo I posted earlier. If this seems like a hard method to write, you should reexamine that demo code!

4. Run a back propagation training process to train your network to recognize these handwriting samples.

   It may take a long time to train. You may need to leave it overnight! I **strongly** recommend writing code that **saves the weight and bias matrices to a file** after each epoch. That way, if you have to stop your code, you can re-load the weights and biases from the file and continue training from where you left off!

5. You'll notice that this lab has a training set and a test set. As with decision trees, the idea is to judge your network based on how well it classifies data points that weren't used in its training. So, once you've trained your network, generate a new set of input/output pairs from the test set file. Run your saved network on the test set without altering it, reporting the percentage of images that are incorrectly classified.

   One possible implementation here is to write the training and testing processes as separate python files. The training saved the weights and biases to a file (pickled) after each epoch. Then, *while the training process was happening for the next epoch*, I was able to run my testing script which loaded the most recent saved weights and biases and ran the test set to get an accuracy. I found this to be pretty efficient, and let me monitor my results easily while training occurred!

Get Your Code & Document Ready to Submit

I won't be running your code with a grading script, but I still want you to submit it to me so that I can see a couple of implementation details. Submit your results via an image or document with copy/pasted console output; I want to see three things:

1) Your network architecture (eg, [784, 300, 100, 10])
2) The percentage of misclassified items **in the training set**
3) The percentage of misclassified items **in the test set** (which you **have not used** for training)
4) How many epochs of training you did

If you're curious about good levels of accuracy to target, the first link on the previous page, the basic information link about the MNIST data set, has some example results. If you scroll down to the section titled "Neural Nets", you can see various architectures and what the error rates were. In that section, the number of layers does not include the input layer, so a 2-layer NN with 300 hidden units represents a [784, 300, 10] network. Lower down, you can see ridiculous networks, like [784, 2500, 2000, 1500, 500, 10] – these take so long to process without the benefit of additional acceleration (which we don't have through vanilla Python, regardless of your computer's capabilities) that I do not recommend using them. I recommend trying one of the 3-layer NNs without any additional features like distortions and see how close you can get to the error shown.

If you're in the ballpark, like your error is less than twice as big as the error shown, you're ready to turn in. If the error shown is 2.95% and your error is 20%, then you need to rethink or debug or run more epochs.

Once you're happy with your error and you've formatted your output as given above, you're ready to turn in this lab!

(And, as a sidenote, try finding a page where you can see a bunch of the images in this data set – like, I think Wikipedia might have some – and take a look at how varied the different handwritten digits are. I find it absolutely mindboggling that this Perceptron network method is so effective. My code for this is less than 100 lines, one of the shortest final versions of any assignment all year, and there's *so much variation* in the input, but our code can get it right something like 97% of the time! This is *crazytown*, and I hope you find this as cool as I do!)

Specification

Submit **a single python script** and a **document file** to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code and document match the specifications above.

Specification for RED credit: Distortions

A strategy for improving the outcomes of a network is to "jitter" the training set when it's input into the network. As the training process progresses, at each input, randomly select whether that input should be put in normally, shifted up one pixel, shifted left one pixel, shifted down one pixel, shifted right one pixel, rotated right 15 degrees, or rotated left 15 degrees. (You'll probably want to look up what numpy's roll command does and the rotate command from scipy.ndimage.) Do not distort the testing set when judging the effectiveness.

Implement this and write a short document indicating how you did it (data structures, what algorithm you used for rotation, etc) and your results **specifically as compared to a non-distorted single network** like you did for the main assignment. Was this better? Worse?

I won't run your code, but I still want to see it!

Submit **a single python script** and a **document file** to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code and document match the specifications above.

There are two ways to get BLACK credit for this assignment; choose one.

Option for BLACK credit: Pairwise Comparison

Another strategy for classification when you have lots of possible categories is to make separate pairwise comparison networks. For example, we could take every example of a "5" and every example of a "7" from the training set, and using only those, train a perceptron network with a single output perceptron (perhaps it would output "0" to represent a "5" and output "1" to represent a "7"). We would do this for every possible pair of digits, all 45 of them, generating 45 separate networks.

To test, we would take a new input vector and run it through **all 45 networks**, keeping track of the results of each network. At the end, whichever classification was selected most frequently would be our choice. (In other words, as one example, if our input was an "8" then all of the pairwise networks that included "8" would output a classifier of "8". All the other pairwise networks wouldn't, but nine separate classifications of "8" would be recorded, which would be more than any other number could occur. Thus: our code would decide it was an "8"!)

Implement this and write a short document indicating how you did it (data structures, etc) and your results **specifically as compared to a single network** like you did for the main assignment. Was this better? Worse?

Once again, I won't run your code, but I still want to see it!

Submit **a single python script** and a **document file** to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code and document match the specifications above.

Option for BLACK credit: Visualization

Something that I'd love to add to this lab is a clear set of instructions for reading in an image from the MNIST data set and displaying it. If you look back to the first page of this assignment, you can see an image of a particular handwritten 5. What's the easiest way to get Python to do this? Ideally, I'd want instructions that take no more than ~30 minutes or so to implement that I could add to this assignment. Specifically, this is what I want students to be able to add to their code:

- Use your image display code to output all of the input images that you classified wrong and the classifications they should have had.

The goal here is to demonstrate how good the code really is – if you get a chance to actually *see* the misclassified images, it becomes clear that they'd be confusing to humans too!

To get credit for this, I'd like you to talk to me about your approach, and then I'd like you to take a crack at writing instructions for another student to duplicate your work. This would include a Python demo file that contains all the commands you used, but does not use them to construct an image, then instructions for how the student would use the commands you demoed to make the images themselves. We'll discuss how to turn it in when we talk!