Fakharyar Khan
December 4th, 2023
Professor Sable
Artificial Intelligence

Project #2: Neural Network

In this project, I implemented a feed forward neural network and created an interface that allows a user to train and evaluate the neural network on a training and test set respectively. In the training part of the program, given a training set and initial values to be used for the weights of the neural network, the model will be trained on the training set for a given number of epochs using a given step size. After which, the weights of the trained model will be exported to a given file. In the testing part of the program, the program will take in the file containing the weights of a trained model as well as the file containing the test set. It will evaluate the model on the test set and export the performance metrics into a file provided by the user.

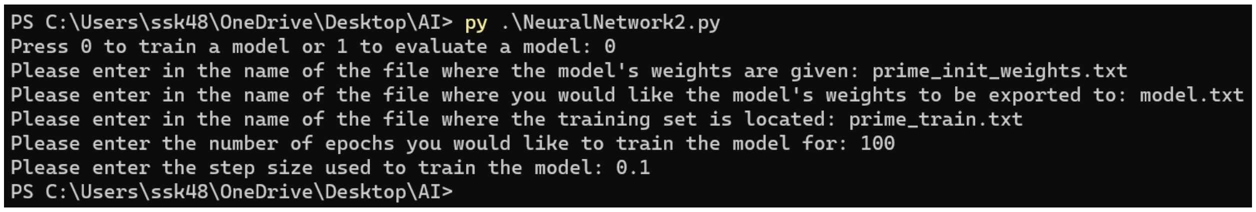
Requirements

To run this program, you should have Python 3.11.5 installed or a later version.

Instructions on Running the Program

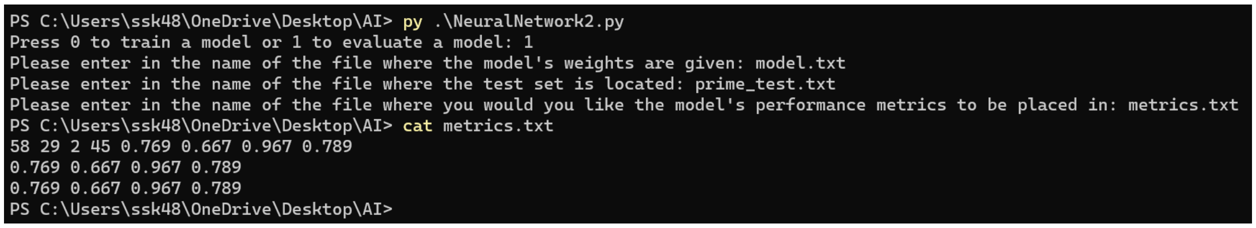
To create and train a neural network, you should have the training set as well as the model's initial weights placed in separate text files in the directory that the NeuralNetwork.py file is in. You should then open up a terminal and cd into the directory where the files are located. Once there, you should enter in the command: python NeuralNetwork.py.

The program will ask if you would like to train or evaluate a model. If you choose train, then the program will prompt you for the file that the initial weights of the model are stored in, the name of the file to export the trained model's weights to, the file containing the training set, the number of epochs the model should be trained for, and the step size to be used as can be seen below:



```
PS C:\Users\ssk48\OneDrive\Desktop\AI> py .\NeuralNetwork2.py
Press 0 to train a model or 1 to evaluate a model: 0
Please enter in the name of the file where the model's weights are given: prime_init_weights.txt
Please enter in the name of the file where you would like the model's weights to be exported to: model.txt
Please enter in the name of the file where the training set is located: prime_train.txt
Please enter the number of epochs you would like to train the model for: 100
Please enter the step size used to train the model: 0.1
PS C:\Users\ssk48\OneDrive\Desktop\AI>
```

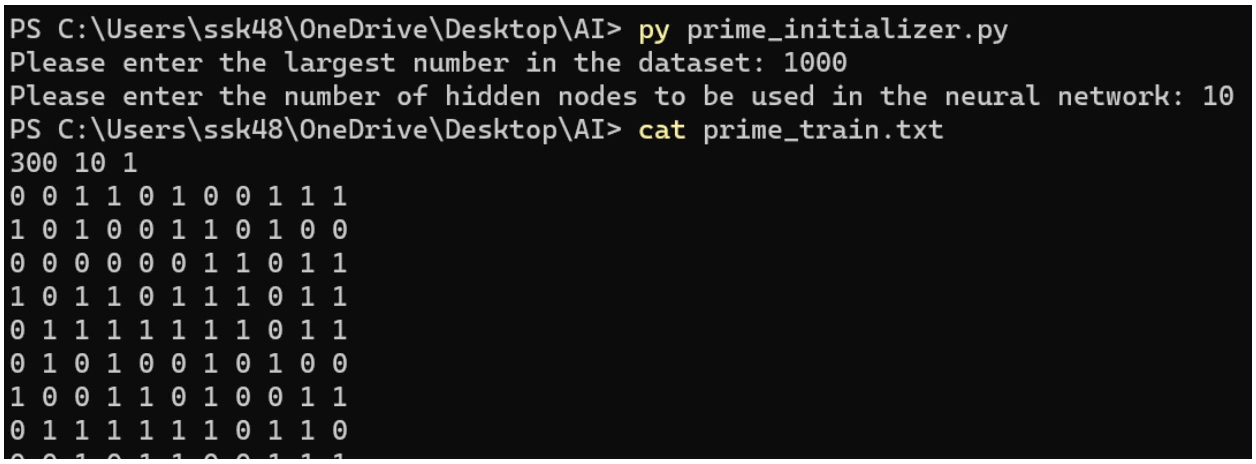
To test a model, you should have the file containing the test set as well as the file containing the trained model's weights in the same directory as the NeuralNetwork.py file. If you choose to evaluate the model, the program will ask you for the file where the trained model's weights are located, the name of the file where the test is located, and the file where the performance metrics of the model should be exported as shown below.



```
PS C:\Users\ssk48\OneDrive\Desktop\AI> py .\NeuralNetwork2.py
Press 0 to train a model or 1 to evaluate a model: 1
Please enter in the name of the file where the model's weights are given: model.txt
Please enter in the name of the file where the test set is located: prime_test.txt
Please enter in the name of the file where you would you like the model's performance metrics to be placed in: metrics.txt
PS C:\Users\ssk48\OneDrive\Desktop\AI> cat metrics.txt
58 29 2 45 0.769 0.667 0.967 0.789
0.769 0.667 0.967 0.789
0.769 0.667 0.967 0.789
PS C:\Users\ssk48\OneDrive\Desktop\AI>
```

<u>Prime Numbers Dataset</u>

For my dataset, I wanted to see how well my neural network could do at determining whether or not a given number is prime or not. The neural network takes in the binary representation of a number as inputs and classifies the number as 1 (prime) or not prime (0). To create this dataset as well as the initialized neural network, I created a python file called prime_initializer.py. This program allows you to easily change the range (maximum number) of the dataset as well as the number of hidden nodes used in the neural network. The number of input nodes is dependent on the range of the dataset. So for example, if the user wanted the largest number in their dataset to be 1000 then we would only need 10 bits to represent each of the numbers in the dataset. As a result, all of the numbers in the dataset would have their binary string padded to the left with 0s so that they are all 10 bits long each (so 3 which in binary is 11 would become 0000000011). A sample run of the prime_initializer program would look something like this:



```
PS C:\Users\ssk48\OneDrive\Desktop\AI> py prime_initializer.py
Please enter the largest number in the dataset: 1000
Please enter the number of hidden nodes to be used in the neural network: 10
PS C:\Users\ssk48\OneDrive\Desktop\AI> cat prime_train.txt
300 10 1
0 0 1 1 0 1 0 0 1 1 1
1 0 1 0 0 1 1 0 1 0 0
0 0 0 0 0 0 1 1 0 1 1
1 0 1 1 0 1 1 1 0 1 1
0 1 1 1 1 1 1 1 0 1 1
0 1 0 1 0 0 1 0 1 0 0
1 0 0 1 1 0 1 0 0 1 1
0 1 1 1 1 1 1 0 1 1 0
```

The program will output the training and test set to files called prime_train.txt and prime_test.txt respectively. And the model's initial weights will be exported to a file called prime_init_weights.txt.

Originally, I had the dataset include all of the numbers from 1 to the maximum number given by the user, N. However, this created a really large class imbalance because there are much more prime than composite numbers. As a result, when I trained a model on that dataset, the model would learn to just predict everything as not prime and it would be able to get a pretty high accuracy.

To fix this, I had my dataset contain all of the prime numbers from 1 to N and then randomly sampled from the remaining composite numbers until the dataset had an equal number of composite and prime numbers. This removed the class imbalance issue and forced the model to learn how the features are related to the output instead of just guessing the same output. The one downside to this was that my dataset became much smaller and if I wanted more data, I would need to increase the size of my model (the number of input nodes) which would make training somewhat slow and increase the risk of overfitting. So for example, there are only 1028 prime numbers less than 8192 which means that I would need 13 input nodes for my network to be able to train on a dataset of that size. It scales pretty well however. For a dataset with a million primes, I would only need 24 input nodes which isn't too bad.

For initializing my neural network, I decided to use xavier weight initialization. This initializes the weights by drawing them from a normal distribution with standard deviation sqrt(6/(num_inputs + num_outputs)) where num_inputs and num_outputs are the number of inputs and outputs respectively into the layer that the weight belongs to. The outputs from the nodes in this neural network can be treated like random variables and because we take a weighted average of the outputs from these nodes, the variance of that sum could end up being really large leading to a very large sum. And if we're using a sigmoid activation function, this could result in the output saturating and giving us a value very close to 1. At this value, the derivative of the sigmoid is close to 0 which means that our gradient ends up being very small which leads to very slow training. To avoid this, we can scale the variance of the initial weights so that the variance of the weighted sum doesn't become incredibly large.

I decided to give the dataset a range of 2047 which gave me a training set with 529 elements in it and a test set with 133 elements. While trying out different parameters I found that my model would overfit if the number of epochs was too large. This makes since my dataset is pretty small so it would be pretty easy to overfit on it. In the end I found that I achieved the best performance when I used 15 hidden nodes, 85 epochs, and a step size of 0.1. With this, I was able to achieve an accuracy of 79.0% on the training set and an accuracy of 78.2% on the test set. The figure below shows the confusion matrices for the model's performance on the training and test set.

Test Set			
TARGET / OUTPUT	Prime	Composite	SUM
Prime	62 46.6%	25 18.8%	87 71.3% 28.7%
Composite	4 3.0%	42 31.6%	46 91.3% 8.7%
SUM	66 93.9% 6.1%	67 62.7% 37.3%	104 / 133 78.2% 21.8%

Training Set			
TARGET / OUTPUT	Prime	Composite	SUM
Prime	251 47.4%	97 18.3%	348 72.1% 27.9%
Composite	14 2.6%	167 31.6%	181 92.3% 7.7%
SUM	265 94.7% 5.3%	264 63.3% 36.7%	418 / 529 79.0% 21.0%

One thing that’s good is that with balancing the dataset, the bias of the model has been significantly reduced. In fact, it looks like it now has a bias towards predicting that a number is prime which is the opposite of the problem that it had before the dataset was balanced. The number of false negatives is also very rare as the recall score is 93.9%. I tried to see if I could get the model to reach 80% accuracy but no matter what I tried, the best I could get on the test set was 78.2%. But I think that it is reasonably well given that it only used one hidden layer. I think the reason why it was able to train and generalize on such a small dataset was because it was starting to learn common divisibility rules. For example, if it learned that the least significant bit of an input number being 0 meant that the number was even and therefore not prime, that would already give it a 50% accuracy. I think with more layers, it would be able to learn the more complex divisibility rules (and the divisibility rules might be harder to express in binary than in decimal) like the one for being divisible by 7.