Dakota Blackwell
300083663

December 11th, 2020
COMP 455 ON1

Identifying Credit Card Fraud via a Neural Network

For the final project, I decided to implement a neural network that would read the variables in the data set and make a prediction whether or not a given transaction was fraudulent. Initially, I had simply decided to use the implementation provided in the lab and apply it to the data found. However, I disliked how the lab implementation was simply using a library and decided that I would better understand the material if the neural network was implemented myself.

The data used for this project is found here: https://www.kaggle.com/mlg-ulb/creditcardfraud

The provided .csv file contains 284807 rows with 31 columns. One of the columns is labelled "Class", and this is a 1 if the transaction is fraudulent and 0 otherwise. There is also have a "Time" and "Amount" column holding their respective values, and finally 28 unspecified variable values. To protect user anonymity, these variables were transformed and unlabelled, but they relate to the transaction in some way.

The code is done in a Jupyter notebook, and is divided into three sections: Imports, set creation, and execution.

The importing of the data is straightforward. I made a single change to the data at this point: I dropped the "Time" column as it appeared to be nearly irrelevant to the data. This column simply has an integer with the number of seconds passed since the beginning of the data – meaning it has an integer between 0 and 172799 (48 hours). This code only needs to be run once, just after opening the file, although there is a slight delay to import the file.

During set creation, the training and test sets are constructed using the provided data. The training set consists of 800 negative samples and 100 positives samples, and the test set consists of 3200 negatives and 400 positive samples. I purposefully use a set number of positives and negatives due to the original sparseness of the dataset – of the 284807 samples, there are only 492 positives, and datasets with only a few positives don't seem to train well. The data is also scaled using sklearn so that the neural network can better handle the data. This code should be run once to get the training and test sets but can be run again to obtain different sets.

For the execution, I decided on using the standard sigmoid (logistic) function, mostly because the derivative was easier to implement. I also decided on having 1 hidden layer, also because the math was easier to implement. That said, this particular experience has convinced me to add linear algebra to my course list, as I can see how useful it is to this field.

When starting, the initial weights are randomized – this is because with equal starting weights and every node having the same input values, the neural network calculates the same backprop values everywhere, and the neural network fails to learn unique values for the weights. I admit that this particular issue took me a few hours to discover.

Once initialized, the inputs are passed through the layers. Since the weights are randomized, the first few passes are going to return nothing more than wild guesses. Once passed through, we look at the derivative of the loss function – Mean Squared Error, again because the derivative is easier to implement – and apply the different to the weights. The actual math here is complex and most of my

time was spent making this work. Appendix 1 has a list of all the sites used in the making of the project – nearly all of them were for the math involved.

Once the training function has looped enough times, the weights are provided to the test function, where they are applied to the test set and the values compared. This code can be run multiple times as needed, although if the second block isn't run, the training and tests will be performed on the same sets.

Because there is a stochastic element to the neural network, the results vary slightly between execution. However, typically there are between 70-80 errors in the set of 3600, which is a success rate of approximately 98%. This is "okay" for some fields, but honestly it was a little lower than I was hoping for, and certainly not good enough for the important job of identifying fraud. More importantly, there are more false negatives than false positives, which means many prospective fraud cases are slipping through.

This could be improved by adding more hidden layers to the network, or potentially by identifying related variables and using a convoluted neural network (tying only certain nodes in a layer to certain nodes in the next layer). Both of these seemed too challenging to do as part of this course.

Altering the ratio of positives to negatives helped more than I expected. I initially ran my tests with a ratio of 1:4 (1 positive for every 4 negatives) and had a success rate of 96%. Through trial and error, I've discovered that drastically raising or lowering the ratio is detrimental, but a ratio of 1:8 actually resulted in less errors, giving a success rate of just under 98%. Going beyond this ratio caused the success rate to begin dropping again.

A better way to improve this is to have more data. We simply don't have the data in this exercise, but in practice the sample size should be much larger. Increasing the training size increases the execution time, but in practice this would be done offline, and only the tests would be done online. However, the sparseness of the positives might still prevent this approach from working.

Overall, I believe my implementation of a neural network was a success, but the actual success rate of credit card fraud detection is not high enough to recommend using it in practice, particularly when looking at the false negatives.

Appendix 1

Websites referenced during the creation of the neural network.

http://neuralnetworksanddeeplearning.com/chap2.html

https://www.youtube.com/watch?v=tIeHLnjs5U8

https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f

https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6

https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks-26c649eb3b78

https://machinelearningmastery.com/why-initialize-a-neural-network-with-random-weights/