University of Waterloo
Faculty of Engineering

PaperlessAI: Product Recommendation Engine Prototype using Feed-forward Artificial Neural Network

A report prepared for

Paperless Post
New York City, NY

By Si Te Feng
20474492
4A Mechatronics Engineering
September 13, 2016

Si Te Feng
39 Mentor Blvd.
Toronto, ON
M2H 2M9

William W. Melek
Director of Mechatronics Engineering
University of Waterloo
Waterloo, Ontario
N2L 3G1

Dear Professor Melek,

This work term report was prepared for my 4A co-op company, Paperless Post. This report is named "PaperlessAI: Product Recommendation Engine Prototype using Feed-forward Artificial Neural Network", and is the third work term report. The purpose of the report is to show how I used a feed-forward neural network to create a product recommendation engine.

Previous coop term, I was an iOS developer within Paperless Post iOS team. My role was to develop new features and make bug fixes for iOS application. Paperless Post is an online invitation e-commerce company that focuses on developing great contents and technologies for making beautiful gift cards. The mobile app on iOS plays an important role in helping the company to deliver a smooth user experience. Lots of people use Paperless Post to send online birthday cards or wedding invites.

I would like to thank Sal Randazzo, David Cilia, Daniel Rodriguez, Iris Zhang, and Corey Goodermote for providing valuable guidance throughout the term.

This report was entirely written by me and has not received credit at any other academic institutions. I have gotten help from the "Work Term Report Guideline" document along with other sources. All sources are listed in the Reference section of this report. I'd like to thank classmate Wesley Fisher for proof-reading the document.

Sincerely,

Si Te Feng,

4A Mechatronics Engineering
ID: 20474492

Table of Contents

List of Figures iv

List of Tables v

Summary vi

1.0 Introduction 1

2.0 Technique selection 6

2.1 Preset Categories 7

2.2 Singular Value Decomposition (SVD) 8

2.3 Artificial Neural Networks (ANN) 9

2.4 Support Vector Machines (SVM) 11

3.0 Implementation of Artificial Neural Network 14

3.1 Designing the Neural Network 16

3.2 Data Acquisition and Formatting 19

3.3 Constructing and training the neural network 22

4.0 Validating the Neural Network 26

4.1 Writing validation code 26

4.2 Result Summary and Experimentation 27

5.0 Conclusion 30

6.0 Recommendation 31

7.0 References 33

Appendices 35

Appendix A: Data parsing code, DataParser.py 35

Appendix B: Main training code, train.py 40

List of Figures

Figure 1: Human Brain Compared with Perceptron …… 2

Figure 2: Sigmoid Threshold Function for an Artificial Neuron …….. 3

Figure 3: SVM Classification in 2-D ……..……..…….. 11

Figure 4: PyCharm IDE User Interface …….. 15

Figure 5: A Simple Trained Feed-forward ANN …… 16

Figure 6: Using Backpropagation to train an Artificial Neural Network … 18

Figure 7: Number of Items Trained vs. Validation Accuracy ……... 28

List of Tables

Table 1: Decision Matrix for Recommendation Engine Techniques …….. 6

Summary

This report describes the entire engineering process for creating PaperlessAI, a product recommendation engine prototype that leverages the power of feed-forward neural network. The purpose of PaperlessAI is to investigate the feasibility of implementing a production grade recommendation engine. During the brainstorming period, several alternatives were considered, which includes 4 major techniques: Preset Categories, Singular Value Decomposition, Artificial Neural Networks, and Support Vector Machine. The selection process for selecting from these 4 alternatives is explained in detail.

The report fully discusses the details behind designing the neural network so that it can understand the input given and correctly output the expected values. Furthermore, the report documents the process of implementing, training, and finally validating the neural network.

Experiments were performed after the neural network is fully implemented to test the accuracy with various number of input items. The network accuracy results from these experiments showed that using neural network for the recommendation engine is not ideal in the short term. More research is needed to improve the quality of the neural network. Alternative methods of implementing the recommendation engine should be considered as well. The reasons for the poor performance are explained.

1.0 Introduction

It was in the year 1943 that a pair of American scientists, Warren McCullough and Walter Pitts, first published the basic mathematical model of the neuron in the Bulletin of Mathematical Biophysics [1]. It was called the Threshold Logic Unit (TLU). The idea of this artificial neural is simple and closely resembles a biological neuron. When a given input electrical stimulation to the neuron exceeds a certain threshold called the Activation Threshold, the artificial neuron outputs a one. Similarly, it outputs a zero if there isn't enough stimulation. The activation function in such a binary fashion is called the Heaviside function, which is a real yet non-differentiable. This fundamental invention marked the beginning of a tumultuous history of Artificial Neural Networks (ANN). It is important to note that the world's first digital computer – ENIAC – would not have been created until 3 year later, in 1946 [2]. People at the time thought that a capable computer should closely mimic the human brain instead of the structured and independent computer submodules that are widely used today.

In 1957, Frank Rosenblatt published the concept of a perceptron, which is a single layered feed-forward neural network with many limitations [3]. A feed-forward neural network is a chain of connected nodes that only passes signals from the input to the output and never transmit signals in cyclic form. The term "node" is used in favor of neuron in the context of within a neural network. By chaining a collection of nodes together, Rosenblatt was able to adjust input going into each node differently to produce desired output. This variability in the input is called adaptive

weights. A visual illustration of the human brain compared with the Perceptron is shown in Figure 1.

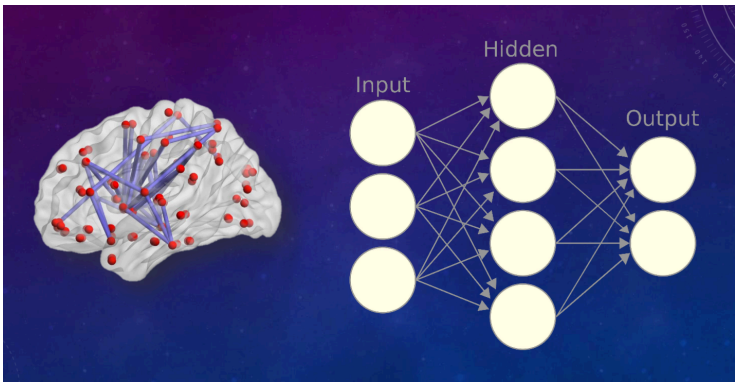

Figure 1: Human Brain Compared with Perceptron

People also had to hard code feature detectors with the perceptron because the network wasn't able to learn on its own. Despite of these limitations, the invention set off a wave of optimism that such a network can match the intelligence of humans within a decade.

Neural networks research came to a grinding halt when Minsky and Pappert published the book Perceptrons in 1969 that presented severe limitations to the perceptron model [4], setting off "The Dark Ages" of AI in the 1970s. The problem is that a single layered perceptron can only solve linearly separable problems, meaning that a perceptron can never mimic the function of something as simple as an XOR gate.

In the late 1980s, after more than a decade of virtually no research in Neural Networks, AI researchers rediscovered the Backpropagation Algorithm by Paul

Werbos [5]. It was found that by combining this new method to train neural networks with the advancements in parallel computing pioneered by Rummelhart, a Multilayered Perceptron (MLP) can be useful to be used as a general function approximator. The key discovery is that the activation function doesn't need to be binary. By using other activation functions such as the sigmoid function or hyperbolic tangent function, researchers can differentiate the equations, which is a requirement for the Backpropagation Algorithm. A graph of the Sigmoid function is shown in Figure 2. The "tanh" function looks almost identical the Sigmoid function.

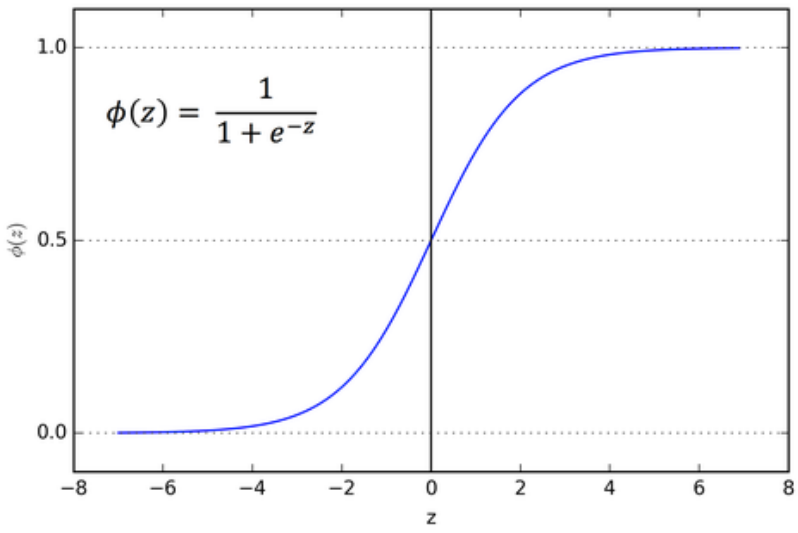

Figure 2: Sigmoid Threshold Function for an Artificial Neuron

Starting from the mid-2000s, the second renaissance of AI was enabled by the popularization of cloud computing, big data, and the increase in CPU and GPU power. These advancements allowed ANNs with many hidden layers to be created

and trained very fast. These ANNs are considered to be a part of Deep Learning, because the layers are deep. When a deep network's hyper-parameters are accurately tuned, it could produce drastically better accuracy than the shallow networks.

Fast forward to 2016, AI is everywhere. At Paperless Post, a company specialized in online invitations and e-cards, it was surprising that it did not yet have a personalized card recommendation system for its users. This was the noticeable problem that PaperlessAI attempted to solve. The objective of this project is to investigate the feasibility of implementing a production grade recommendation engine for Paperless Post, measure the performance of the potential solution, and to offer recommendations on the project direction in the future. The project began with a technique selection period where the canonical ways of implementing a recommendation algorithm were considered. The method of ANN was ultimate selected. The implementation of the project includes four parts: data acquisition, implementation, training, and validation. Implementing such a personalized feature based on past customer behavior will allow higher product engagement, customer satisfaction, and better content discovery. Users will be able to find new cards befitting their tastes without using the visual card browser or the search bar.

The method that was used to train PaperlessAI in this report is a feed-forward multilayered neural network, not too different from the version developed in 90s. The project takes advantage of a newly open-sourced framework TensorFlow for

the implementation. The TensorFlow framework originated from Google's Brain team and was used by many internal teams before and after its public release [6]. The framework was developed in response to the increasing demand for a Machine Learning framework that incorporates all the modern features such as parallel computing. This is a direct result from the second renaissance of AI. With TensorFlow PaperlessAI only needed to focus on the curating the input data and shaping the overall structure of the neural network.

2.0 Technique selection

The project is focused on exploring the potential methods of implementing a prototype recommendation system. This is why deciding the preferred method involves both the technical aspects as well as the actual value to the company. There are ways in which a primitive recommendation system can be built, but it will provide little value to the company since that will never be implemented in a production environment. The selection process relies heavily on the initial brainstorming. Many ideas were written down and considered.

After similar methods are combined, there are four main ways in which a card recommendation engine can be implemented. These characteristics of these methods are summarized in Table 1. Each method is described in detail in sections below.

Table 1: Decision Matrix for Recommendation Engine Techniques

Technique	Preset Categories	Singular Value Decomposition	Artificial Neural Networks	Support Vector Machines
Accuracy and Performance	3	5	3	4
Flexibility	1	3	5	3
Ease of Implementation	1	3	5	1
Personal Concept Understanding	5	1	3	1
Speed	5	5	2	3
Value	1	3	5	5
Total	**16**	**20**	**23**	**17**

2.1 Preset Categories

Present Categories is a method that came up first during the brainstorming session. The process is largely manual but nonetheless effective to a certain degree. The idea is that cards can be separated into a few major categories, such as birthday, national holiday, or wedding. Then, the backend software would create a simple counter that increments the count on a category whenever the user sends a card in that category. This category count represents the category inclination for that specific user. For instance, if the user likes to send a lot of birthday cards, the system would recommend more popular birthday cards for the user.

One major drawback is that there needs to be a lot of manual adjustments for the system to work as expected. For instance, event invites for weddings or funerals are sometimes emotionally sensitive for the user. Therefore, the recommendation engine should refrain from recommending cards in those categories. The other consideration is that this method offers little technical value for the company. There wouldn't be anything new that can be learned from this.

It was with these considerations in mind that Preset Categories was not chosen. It has far less flexibility value when compare to the other methods involving Machine Learning (ML). In terms of accuracy, it always recommends the best cards, which is a good thing, but not enough personalization. This system relies heavily on web backend technologies that the author has little experience in, so it scores very low

in ease of implementation, but the underlying concept is very easy to understand. Overall, this method provides little value to the company in the long run.

2.2 Singular Value Decomposition (SVD)

Singular Value Decomposition is method that emerged around the same period as when the first Perceptrons were first developed. It is a method in the field of Linear Algebra that involves factorization of a (m x n) matrix **C**. The desired format for the decomposed matrices are as follows in Equation 1:

$$\mathbf{C} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathbf{T}}$$

Equation 1: SVD Equation

where **U** is left singular vectors, a (m x r) matrix

Σ is singular values, a (r x r) matrix

V is right singular vectors, a (n x r) matrix

r is the rank of matrix **C**

The interesting aspect to this approach is that each decomposed matrix actually corresponds to meanings in a physical sense, in terms of general concepts. The initial matrix **C** could contain "m" users sending "n" types of cards. If the value of "n" is large in the case of Paperless Post, which contains a repository of 20,000 types of cards. The transformed **U** matrix is able to condense that value to "r",

which -- depending on the patterns in the data -- could be much less than the value "n".

At the time of decision. It was found that there were other people at Paperless Post already implementing the SVD method. In terms of expertise, lots of learning would be required which makes a successful implementation difficult to achieve in the short timeframe given.

2.3 Artificial Neural Networks (ANN)

Artificial Neural Networks was another promising candidate. It offers the most flexibility of all due to its resemblance to the human brain, and is theoretically capable of learning any kind of tasks. The drawback is that the computational resources required to train the neural network surpasses all other methods. If the ANN is designed well, it can not only become a recommendation engine, but also be used for other classifying tasks as well.

There are many different kinds of ANN to select from. The feed-forward neural network is the most basic option that offers classification or regression capabilities. It is best suited for dealing with purely informational data and tasks that doesn't involve memorization. The general data flow of such a network always goes from a set of inputs to a different set of outputs.

A similar network to the feed-forward type is called an Auto Encoder, which is a feed-forward neural network trained to reconstruct the input. Mathematically speaking, such a network with linear activation function is the same as SVD. Since Auto Encoders are harder to debug than SVD, it provides little added value to the project.

Convolutional neural networks are widely used, but are almost always used for machine vision tasks, thus not directly applicable for PaperlessAI.

Long Short Term Memory (LSTM) neural networks are used only when the ANN needs to remember certain details about the input. For example, when the network is trying to read a string of characters, it must be able to remember what it read in before.

The feed-forward neural network was finally chosen for its applicability to PaperlessAI. Thanks to the flexibility and proclaimed performance of ANNs in general. This is combined with the fact that the general decision matrix shown in Table 1 produced the highest value for the feed-forward ANN. Thus, ANN seems to be the most suitable for implementing PaperlessAI.

2.4 Support Vector Machines (SVM)

The foundation for Support Vector Machines lies in its Decision Rule, shown in Equation 2 and illustrated in Figure 3. In the figure, blue circles represent positive data items, while the red squares represent negative data items. The words positive or negative here does not implicate good or bad, they simply mean whether the data item should be accepted or rejected in a given scenario based on the SVM Decision Rule.

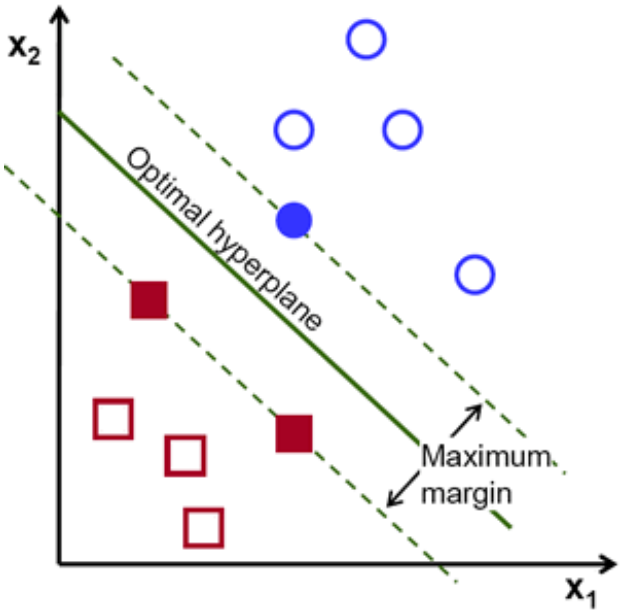

Figure 3: SVM Classification in 2-D

$$y_i(x_i w + b) - 1 = 0$$

Equation 2: SVM Decision Rule

where y_i is the output

x_i is the current sample

w is the direction vector orthogonal to the optimal hyperplane

b is the maximum margin between the closest data sample to the optimal hyperplane

Although a 2-D classification is illustrated in figure 2, SVM can transform the data into a higher dimension as necessary, and can therefore handle any types of classification tasks, not just the linearly separable ones. Linear separability refers to scenario when items on a 2-D graph can be completely separated into two groups by drawing an infinitely long straight line though the graph.

The SVM method involves solving many complex differential equations, which was thought to be a huge challenge when implementing in Python. Implementing solutions for differential equations with a Python math framework is very challenging, especially when the values are in the form of matrices. Additionally, the SVM method is not as flexible as ANN for the types of problems it can solve. Modifying it for another task would be more difficult to achieve.

The difficulty of SVM combined with its lack of performance resulted in a low

score on the decision matrix. Therefore, the method of ANN with the highest score is used instead of SVM.

3.0 Implementation of Artificial Neural Network

The task of creating a card recommendation engine is very open-ended. At Paperless Post, there are many kinds of real life anonymized customer data and product data that could potentially work with ANN. Different types of ANNs are available depending on the data that is collected. Therefore, designing the neural network was probably the most important step that came before the programming part. Data that is queried from the database was then restructured before it could be used in the designed ANN.

Programming the ANN itself actually took less time than data structuring thanks to the TensorFlow framework [6]. Both data structuring and ANN code was written in Python v2.7 because of two reasons. One is that Python is a high level scripting language that is extremely easy to learn and use. In the case where one simply wants to transform input data from one form to another, writing a Python script is one of the fastest ways to do so. This, however, comes at a small cost that the speed of execution is considerably slower than other languages. Second reason is that Python is the only language supported by the TensorFlow and Numpy frameworks. Without the TensorFlow framework, all the backpropagation code must be implemented manually, which is a time consuming process. Python v2.7 was used because wat is better supported than v3.4.

To run the Python code, I've used both Sublime Text 2 text editor and PyCharm IDE (Integrated Development Environment). Figure 4 shows the user interface for

the fully-featured PyCharm IDE. Sublime Text 2 is easy to use and provides instant python code execution within the editor, which made things fast and efficient. PyCharm on the other hand is a fully featured IDE that has console query and debugging capabilities.

Both applications used to run the code were running on Mac OS X El Capitan, on a MacBook Pro machine with 16GB of memory. The large amount of memory helped the TensorFlow framework to cut down on training time dramatically.

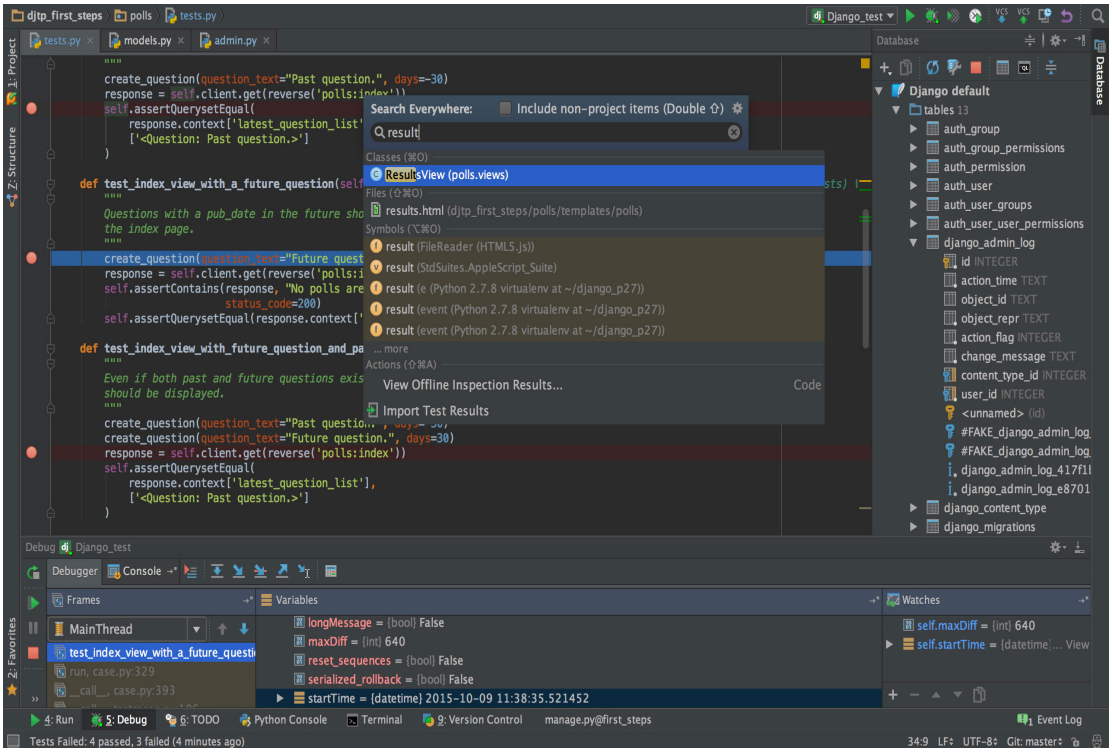

Figure 4: PyCharm IDE User Interface

3.1 Designing the Neural Network

The first idea is the ANN design is that when recommending new products to a user, the software system should be able to first retrieve the historical record for that specific user. Only then could the ANN be able to use its judgement to decide. However, in order for the ANN to be able to have the correct judgement, it must be trained prior to deployment using the famous Backpropagation Algorithm and thousands of card sending records made by other users. The idea is that some users will be very similar to the current user, so by learning the past users' behavior, the ANN will be able to predict the future actions of the current user.

Learning through Backpropagation is a general mathematical way to adjust the weights of the node connections. In biological analogy, it's changing the overall strength of electrical stimulation delivered to the dendritic tree for each neuron. Figure 5 illustrates a simple neural network with varying weights for each node for a pre-trained ANN.

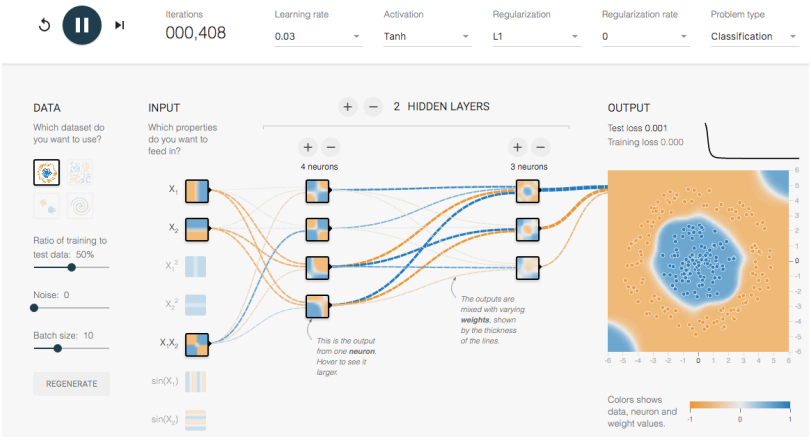

Figure 5: A Simple Trained Feed-forward ANN

There are many kinds of Backpropagation algorithms such as Gradient Descent, Momentum, Adagrad, etc. These techniques affect the training speed, but are unlikely to change the final result produced by the ANN. This will be revisited in sections below.

The next question that was considered was how to input the card into the neural network. For the ANN to know which user, the naïve approach would be directly inputting the numerical ID of a user and hope that it outputs a list of product IDs to recommend. That approach will not work at all because the ANN can only recognize data that creates a pattern over time. A user's anonymized ID is completely random, and thus will not teach any pattern to the ANN. This is probably the trickiest part when designing the structure of an ANN.

After some planning, it was decided that a feed-forward Neural Network is best for dealing with tasks that doesn't involve recalling memory or images. The input is going to be a 40,000 x N sparse matrix, where N is the number of training samples in a mini-batch typically set to around 50. The first 20,000 input nodes out of 40,000 represents the current card that needs to be validated for the current user. If the neural network thinks it's probable that the current user will send this card next, it will output a possibility of X of greater than zero, but less than one. Process would be repeated for many cards, and later sorted by possibility, thus generating a list of recommendations based on send likelihood. The last 20,000 out of 40,000 nodes represents the previous cards that the current user has sent before, which allows the

ANN to fully know the current user that it's dealing with. The number 20,000 is the approximate total number of cards in Paperless Post's database, and is only for technical purposes.

Such a uni-directional ANN can also be trained in a very similar way that it operates. One could use the send historical record to train the ANN. If it is known that a user would send card "A" next for sure, set the output node value of the ANN for card "A" to 1 and Backpropagate the results from the output layer to the input layer to adjust the weight of the nodes. Likewise, if the historical record shows that card D was never sent by the user, set the output node value to zero. Hopefully, the ANN will learn which type of users send what kinds of cards. Figure 6 shows the process of Backpropagation visually.

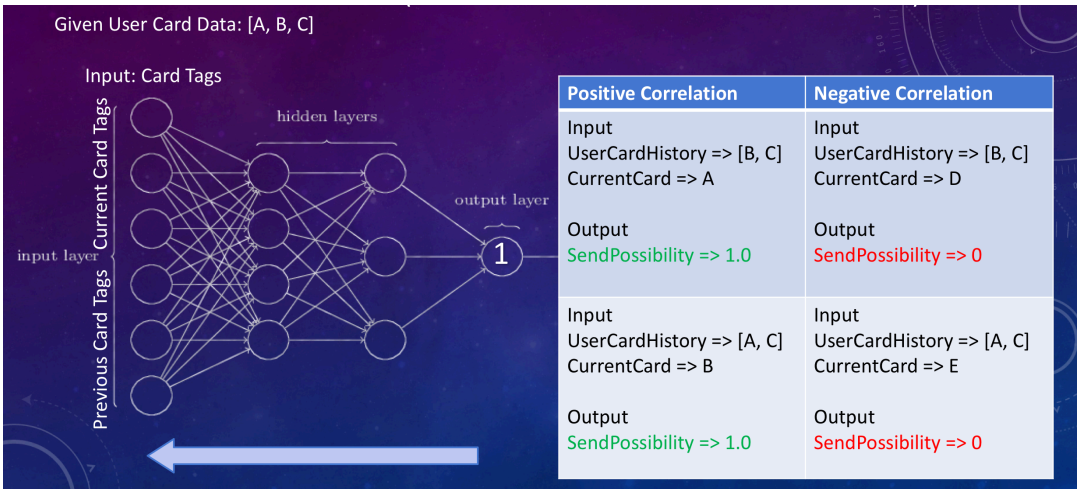

Figure 6: Using Backpropagation to train an Artificial Neural Network

As illustrated in Figure 6, a feed-forward neural network always contains one or more hidden layers. It's called hidden because the value of the nodes along with

their weights cannot be known. In that respect, an ANN is a black-box that outputs some values when given an input. Considerations were made about how many layers to use, and how many nodes per layer to use. The most optimal hyper-parameters are often derived empirically, so the most sensible values were given based on personal judgement. Three hidden layers were chosen, with 1000 x N, 150 x N, and 10 x N nodes for each hidden layer. These are eventually connected with 40,000 x N input nodes and N output nodes to form the complete neural network.

Prior to the design decision, another approach was considered but was not used. It was thought that card IDs can be correlated to the card images, and using a convolutional neural network to preprocess the card images before feeding into the fully connected network. The benefit of such approach is that the ANN will make decision not just based on the card ID, but also the color and shape within each card. It was determined that this approach would use too much computing power, as each image would have at least 300x700 pixels. Which is equivalent to a massive 210,000 nodes input layer. The MacBook Pro that was used in the project will not able to handle that much data.

3.2 Data Acquisition and Formatting

Acquiring data and later transforming it into usable format was not a trivial task for PaperlessAI. There were difficulties in learning how to use the Redshift database at Paperless Post. Help was provided by a fellow co-worker to extract user sent

records into a local file by querying the database. SQL (Structured Query Language) commands were used to query the online database for all the card sending transactions from January 2014 to July 2016. With the help of a simple shell command, the queried data is directly transferred to a CSV (Comma Separated Values) file in raw format. The data in the CSV file contains approximately 3 million rows, with 2 numbers in each row separated by the "|" symbol. The first number of each row represents the user identifier for that particular transaction. The second number represents the card package ID that the user sent. This format is drastically different from the input that is needed by the ANN previously designed. Out of the 3 million records, only about 200,000 end up being used.

Transforming data from CSV to the input matrix was initially thought to be a one-step process. Later, it was found that this would make the procedure unnecessarily complicated. Data was instead transformed in a four-step process by a Python class called "DataParser", where each step is clearly handled by one Python function. The Python code for all four steps of data formatting is added for reference in Appendix A.

The first function is called "readCSV". The function takes in the name of the CSV file to read into the Python program and the number of rows to be read. Then the function does several things, and attempt is made to describe the overall procedure without going into the programming details. The "readCSV" function parses each row in the raw CSV and add the card package identifier into a set of known package

identifiers. Meanwhile, it looks for the current user identifier in a Python 2-D array, where each row represents a user. If the user exists already, add the package identifier to that user. If the user doesn't exist, append a new row to the 2-D array. The output of this function is the entire user sending history 2-D array, as well as a 1-D set of unique package identifiers. Both of these are important in helping the Python program to keep track of the historical records.

The second function "shuffle" was actually created after the ANN has been built. It was later found that training a neural network with data sorted chronologically was a bad idea. The backpropagation algorithm relies on training by using mini-batches of 50. The average error rate from the mini-batch is assumed to be the global error rate so that computations can be sped up. Therefore, the input data must be ensured to be completely random for the mini-batch speed optimization to well in practice. As the name of the function indicates, "shuffle" simply shuffles the user sending history 2D array. Although simple, it is an important part of data transformation.

The third function "getTrainingMatrixFromRawDataset" is somewhat a misnomer. It takes in the user sending history 2D array and turns it into a more redundant form to be used for training the network as described in the network design section. For each user, it generates all the positive correlation training cases possible, and create an equal number negative training correlations. For example, if a user has sent the cards: "A", "B", and "C", it will create a matrix where "A" and "B" is the historical

record, “C” is the current card, and the output possibility is set to 1. Then, it will also create a matrix where “A” and “B” is the historical record, “G” is the current card, and the output possibility is set to 0, because the user has never sent the card “G” before. By reiterating the procedure to all combinations of cards for all users, the function is able to create the full training dataset ready for ANN to consume.

The fourth step is separating the full dataset from the previous function into 2 parts. Out of the 200,000 entries that was fed into the program, 92% of the data will be used for training the ANN, called “trainDataset” and its corresponding labels “trainLabels”. The rest 8% is stored in two other matrices for validation purposes. Validation is the process of verifying the accuracy of an ANN and will be discussed later.

With data fully prepared, the project was ready to move onto the next stage of constructing the actual structure of the ANN through Python.

3.3 Constructing and training the neural network

The feed-forward ANN is programed in Python with TensorFlow framework. The full neural network code is in Appendix B. The key characteristic of the framework is that commands are not executed right away at the time of constructing the data flow graph, but rather, they are executed together at the end with the run command. This provides the benefit of concurrent execution of the commands on multiple

cores, which are all handled by the framework behind the scenes. TensorFlow calls any type of connected computational nodes as a data flow graph, which is synonymous with the word network.

The only major challenge when constructing the ANN was the syntax for inputting the dimension of the layers correctly. There were a lot of trial and errors to get it right. For example, there should be three dimensions for the input layer: batch size (of size 25), number of desired input nodes (of size 40,000), and placeholder dimension (of size 1). The last dimension is only used for mathematical compatibility when doing matrix multiplication. Additionally, the order of these dimensions were also initially unknown. Ordering the dimensions according to linear algebra rule actually did not work because the TensorFlow API were designed a little differently. The problem was eventually fixed after carefully reading through the documentation [7] along with trial and error.

The network output from nodes and weights are calculated using the conventional method, namely with the formula: $\mathbf{y} = \mathbf{W} * \mathbf{X} + \mathbf{b}$, where **y** is the output value, **W** is the weight of the connection, **X** is the input value, and **b** is the bias. Bias is simply a value from 0 to 1 in this case, and is important for the ANN to function property. The output of the node is then transformed into actual output signal using the Sigmoid function. Thus, one can expect the output value to be analog, between 0 to 1.

Error value for training purposes is calculated using the Mean Squared Error (MSE) function as shown in Equation 3. In this figure, "J" represents the error value; "m" represents the total number of training cases in the mini-batch; "h" represents the ideal target value; "y" represents the actual output value of the network; "i" represents the current training case index. Using MSE is much better than taking the difference between the output and target, or also known as Mean Absolute Error (MAE). The result of MSE will always be a positive value, which ensures positive errors will not cancel out with negative errors. Moreover, large errors in MSE will be penalized harshly, giving the ANN a boost in learning rate in the beginning.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

Equation 3: Mean Squared Error Function

To actually use Backpropagation to train the weights of the network, Adagrad optimizer is used. Adagrad is a relatively sophisticated learning algorithm that does well with sparse inputs, which fits this project rather well.

After the structure of the network has been set up, the program specifies the number of steps to train the network. This value is only used if early stopping is desired. For this project, the maximum number of training steps was specified because there are already other ways to control the number of training cases. The "run" function is repeatedly executed inside a loop to reiterate through each training step.

When the number of records given to the ANN is set to 200,000, it took the machine more than 8 hours to train through all the records. Other training sessions ranging from a few minutes to few hours have also been conducted with the same network. However, the overall accuracy during and after complete training were not satisfactory. This issue is discussed in detail in the next section.

4.0 Validating the Neural Network

Training the neural network was passing data backwards from the output to the input, as indicated by the method name “Backpropagation”. To check the accuracy of the neural network at any given time, one can simply input a known example item, and check if the output of the ANN matches the expected target.

4.1 Writing validation code

There were two kinds of accuracy checking for PaperlessAI: test and validation.

Testing was done during the training process to give the software developer an idea of the training progress. When the error rate wasn’t going down, he/she could quickly stop the script. The testing code for PaperlessAI was embedded after Adagrad Backpropagation calculation was done for each mini-batch. As a result, the programmer was able to receive a test feedback every 5 seconds.

Validation was performed after the ANN has completed the training, and it used a different set of historical records than the tests. For PaperlessAI, this also meant that the validation was run after all the given inputs were processed by the ANN. For the purpose of validation, another loop was constructed in the Python script, similar to the training loop. This time around, no training was done to the network—only the classification accuracies for validation error were computed and averaged. Unlike the error function MSE that was used for training, validation error function used binary determination. The result for each sample item is either

labelled as guessed correctly or incorrectly. To get the error rate, PaperlessAI used a substantial number of testing item and averaged the validation results. Correct items were represented as floating point value of 1.0, and incorrect items were represented as floating point value of 0. When hundreds of these items were averaged, the script produced a value between 0 to 1 that indicated the real performance of the ANN.

In order to notify the programmer of the final result, PaperlessAI printed the test and validation values on screen as percentage accuracies.

4.2 Result Summary and Experimentation

Experiments were conducted on the ANN with various number of input ranging from 291 to 14,453. The validation accuracies of these experiments are summarized in Figure 7. It is interesting to note that the general accuracy is going downwards as the ANN is being trained.

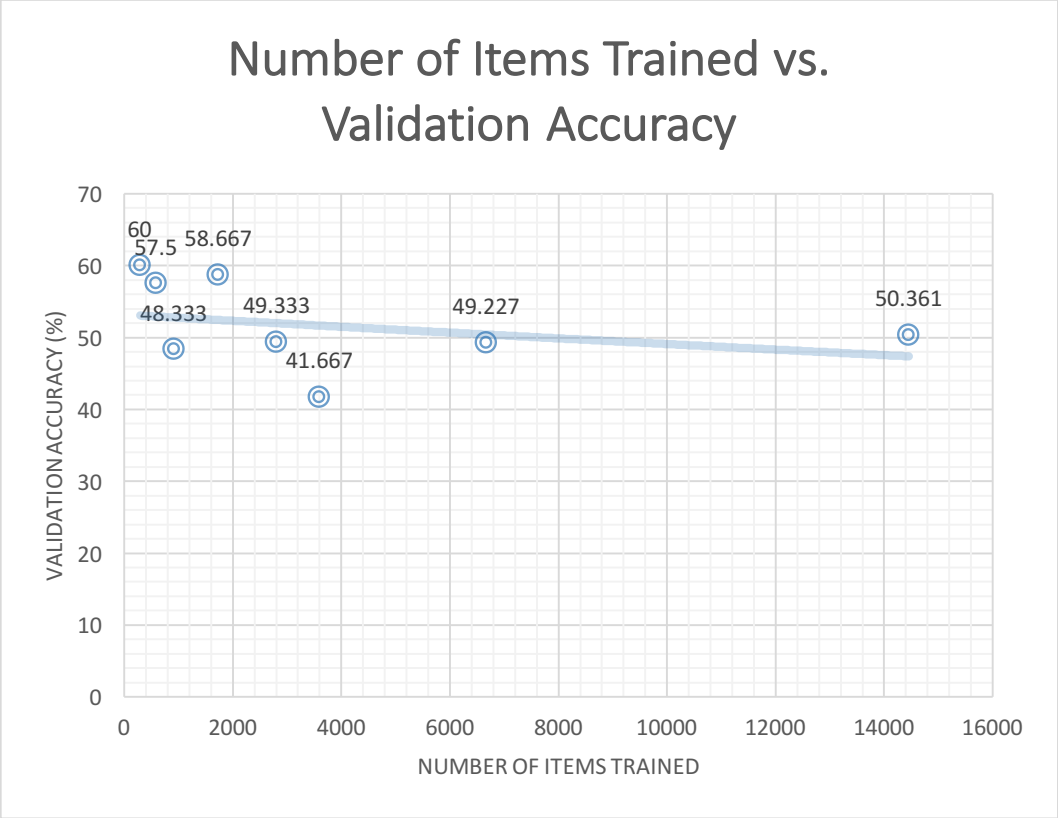

Figure 7: Number of Items Trained vs. Validation Accuracy

There are not enough data points to indicate a sure cause of the issue, but it is almost certain that the cause is the experiments with smaller number of training cases were more prone to give outlier results. If there were 291 training items in total, and only 8% of that number is used for validation, it is clearly conceivable that the 60% accuracy was the result of pure chance. The actual accuracy seems to be around 50% as more training items are used.

Initially, attempts were made to change the hyper-parameters of the ANN, such as number of nodes per hidden layer. These change did alter the output of the network, but did not improve the accuracy of the network at all.

The sparsity of the input and the inherent randomness in the input data were ultimately found to be the two main causes that have led to the poor performance of the ANN. There are 40,000 nodes in the input layer, and only a handful of those contain any value at all besides zero. This would have likely caused the vanishing gradient problem where the values in the neural network get smaller and smaller until every node in the hidden layer becomes zero. When investigating the ANN, it was indeed found that all the output values of the ANN were zero. Due to time constraints, a solution for this problem was investigated but never implemented.

It is also questionable whether the historical data has any patterns at all that can be extracted by the ANN. There are 20,000 packages in total, but the PaperlessAI script was only able to cover 14,453 training cases in the longest training session of around 8 hours. There might be so little user behavior overlap that it's simply impossible to conclude with any usage patterns. In addition, the real world is full of messy inconsistencies and seasonal shifts. For example, a valentine's gift card might be popular in general, but will likely not be sent during Christmas season. It is this realization that have led to some extensive reflection on the effectiveness of such a project only power by historical records without any cultural context.

5.0 Conclusion

The objective of PaperlessAI is to investigate on the feasibility of implementing a production grade recommendation engine. In terms of programming, it was found that implementing an ANN is relatively trivial in Python v2.7 thanks to the TensorFlow open-source framework. However, manipulating the structure and hyper-parameters are difficult tasks that takes time and skill to learn.

Several machine learning alternatives were explored before the conceptual design. The benefits and drawbacks between Preset Categories, SVD, ANN, and SVM were compared, and ANN was ultimately chosen based on the decision matrix shown in Table 1.

During the project, it was found that the important part of a neural network is that it gets meaningful input data in the first place. Information should be formatted such that the input matrix into the ANN should not be sparse.

After the implementation and testing, it was discovered that the benefits of neural network proclaimed by many experts in machine learning largely applies to the development of deep learning. Migrating into deep learning requires a lot of knowledge in new problems such as vanishing gradients in a deep neural network. Research into this area is still far from complete and often pales in comparison to more robust approaches such as SVD.

6.0 Recommendation

The use of ANN for Paperless Post production recommendation engine is not recommend in the near term due to its lack of performance. However, there should be an ongoing investigation into improving PaperlessAI so that it gets better over time as more optimization are applied to the project. For the near term, it is recommended that another recommendation engine prototype should be built using the SVD method.

There are several potential paths that could lead to a better PaperlessAI performance. Perhaps there is a way to separately categorize user send history with different time periods of the year. That way, the ANN will have a context to which time of the year it's processing the cards for, which could greatly reduce the number of items that needs to be learned and increase the validation accuracy.

Additionally, all the Paperless Post cards should be grouped into major categories of no more than 30 kinds. This would allow the number of inputs to decrease from the astronomical 40,000 nodes to the reasonable 30 nodes. The corresponding input data should also be reformatted to fit the new structure. This dramatic change in the number of inputs will not only reduce the computational time, but also largely eliminate the danger having a sparse input. However, it is noted that categorizing all 20,000 cards will not be a simple feat to accomplish.

Furthermore, future improvements to PaperlessAI should include better separation

of concerns. For the purpose of demonstrating the use of ANN for Paperless Post, the main python script was written without a class. However, the script should have been separated into different classes and executed individually by the main function instead. This will no doubt make the code easier to understand.

7.0 References

1. McCulloch, W., & Pitts, W. (1943). A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY. Bulletin of Mathematical Biophysics, 5, 115-133. Retrieved from http://www.minicomplexity.org/pubs/1943-mcculloch-pitts-bmb.pdf

2. CHM Revolution - ENIAC. (2016, September 16). Retrieved September 16, 2016, from http://www.computerhistory.org/revolution/birth-of-the-computer/4/78

3. Perceptrons. (2016, September 16). Retrieved September 16, 2016, from https://www.britannica.com/technology/perceptrons#ref1009913

4. Minsky, M., & Papert, S. (1987). Perceptrons: An Introduction to Computational Geometry. The Science Press.

5. Werbos, O. J. (1994). The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting. John Wiley & Sons.

6. About TensorFlow. (2015, October 1). Retrieved September 16, 2016, from https://www.tensorflow.org/about.html

7. API Documentation. (2015, October 1). Retrieved September 16, 2016, from https://www.tensorflow.org/versions/r0.10/api_docs/index.html

Appendices

This section presents the Python code that was written for PaperlessAI.

Appendix A: Data parsing code, DataParser.py

```
#
#  Created by Si Te Feng on Stampy Day, Jul/13/16.
#  Data queried by Dean Hillan
#  Copyright c 2016 Paperless Post. All rights reserved.
#
#

import csv
import numpy as np
from sets import Set
from random import randint


class DataParser:
    def __init__(self):
        pass

    # private
    def num(self, s):
        try:
            return int(s)
        except ValueError:
          return int(0)

    # Returns tuple (trainingDataset, validDataset) which are
split from the original dataset
    # based on the validation data size specified in the
parameter
    def splitDatasetIntroTrainingAndValidation(self, dataset,
validDataSize):

        trainIndexCutoff = len(dataset) - validDataSize
        trainDataset = dataset[:trainIndexCutoff]
        validDataset = dataset[trainIndexCutoff:]

        return (trainDataset, validDataset)

    '''
    Param fileName specify the csv file to read from in the
format of
                    accountId|packageId, assuming sorted by
accountId!
    Returns a tuple with the following in order:
```

```
    - Matrix with each row representing send history of one
user
      eg row1: 0 2 0 0 0 1 0 1 0 ... <3000>
    - An unordered unique set of packageIds that has been
sentduring the dataset period

    * CSV parsing steps:
    1) Since some packages are not active, first store all the
sent packageIds
    in an unordered set. Then get the length of the set.

    2) Create a new 2D matrix of width {numActivePackages}

    3) For each row from csv, add the package to the 2D matrix
for the users row,
       since csv is sorted by accountId. If the accountId
changed from last iteration,
       move down one row and add new package

    *Note:* Eliminating users who only sent 1 card, which is
not useful for training.
            Thus, the returned array will not contain users
with single card send history
    '''

    def readCSV(self, fileName="accounts_packages.csv",
maxReadRows=200000):

        # listOfAccounts = []
        uniquePackages = Set([])

        f = open(fileName, 'rb')
        reader = csv.reader(f)

        # 1) Get packageIds
        print("Getting Unique PackageIds...")
        for row in reader:

            rowString = row[0]
            separatedElems = rowString.split("|")
            packageNumber = self.num(separatedElems[1])
            if packageNumber == 0:
                continue

            uniquePackages.add(packageNumber)

        uniquePackageCount = len(uniquePackages)
        print("Unique PackageId Count: %d" %
uniquePackageCount)
        uniquePackageList = list(uniquePackages)

        # 2) Create new 2D matrix
        userSentHistory = []
        print("Parsing User Sent History...")
```

```
        # 3) Add user sent data to new matrix
        f = open(fileName, 'rb')
        reader = csv.reader(f)

        i = 0
        currAccountId = 0
        currPackages = np.empty(0)  # In sparse format
        userSentCardCount = 0
        for row in reader:
            rowString = row[0]
            separatedElems = rowString.split("|")
            accountId = self.num(separatedElems[0])
            packageId = self.num(separatedElems[1])
            if packageId == 0 or accountId == 0:
                continue
            # print("Parsing: AccId[%d], PackId[%d]" %
(accountId, packageId))

            if accountId != currAccountId:
                # Current user must have sent 2 card or more
to be considered for training data
                if userSentCardCount > 1:
                    userSentHistory.append(currPackages)
                # reset variables for next row
                currPackages = np.zeros(uniquePackageCount,
dtype=np.uint8)
                currAccountId = accountId
                userSentCardCount = 0

            columnToAdd = uniquePackageList.index(packageId)
            currPackages[columnToAdd] =
currPackages[columnToAdd] + 1
            userSentCardCount += 1

            i += 1
            if i % 3000 == 0:
                print(
                    "Processing data... [%.02fk out of max
of %.02fk](%.0f%%)" % (
                        i / 1000.0, maxReadRows / 1000.0, 100
* i / maxReadRows))
            if i > maxReadRows:
                break

        # Adding the last user to history list as well
        if userSentCardCount > 1:
            userSentHistory.append(currPackages)

        f.close()
        print("User Sent History gathered: rows[%d],
cols[%d]" % (len(userSentHistory), uniquePackageCount))
        return (userSentHistory, uniquePackageList)
```

```
    # Returns transformed dataset and labels ready for neural
network consumption
    # Width of the return matrix is twice that of the input
raw dataset
    def getTrainingMatrixFromRawDataset(self, rawFullDataset,
uniquePackageCount, matrixMagnificationFactor = 10):
        rawFullDatasetCount = len(rawFullDataset)

        fullDataset = []
        fullLabels = []
        expectedLabel = np.array([1.0], dtype=np.float)
        notExpectedLabel = np.array([0.0], dtype=np.float)

        userIndex = 0
        for currUserHistory in rawFullDataset:

            for i, cardSentCount in
np.ndenumerate(currUserHistory):

                currUserCard = np.zeros(uniquePackageCount,
dtype=np.uint8)
                notCurrUserCard = np.zeros(uniquePackageCount,
dtype=np.uint8)
                restOfUserHistory = np.array(currUserHistory,
dtype=np.uint8)

                if cardSentCount != 0:
                    # Generating positive correlation data:
ANN is expected to output favorably to
                    # these input data because it's what
happened. See powerpoint for details.
                    currUserCard[i] = 1 *
matrixMagnificationFactor
                    restOfUserHistory[i] = (cardSentCount - 1)
* matrixMagnificationFactor

                    # For each positive correlation data, we
generate one negative correlation
                    # data: ANN is expected to output 0 to
these input data because the current
                    # card was not sent by the user in
reality.
                    randomIndex = randint(0,
uniquePackageCount - 1)
                    while randomIndex == i:
                        randomIndex = randint(0,
uniquePackageCount - 1)
                    notCurrUserCard[randomIndex] = 1

                    # Add for positive correlation
                    combinedTrainItem =
np.append(currUserCard, currUserHistory)
                    fullDataset.append(combinedTrainItem)
                    fullLabels.append(expectedLabel)
```

```
                    # Add for negative correlation
                    combinedNegativeTrainItem =
np.append(notCurrUserCard, currUserHistory)

fullDataset.append(combinedNegativeTrainItem)
                    fullLabels.append(notExpectedLabel)

            userIndex += 1
            if userIndex % 50 == 0:
                print(
                "Transforming user history...
[%.03fk/%.03fk]" % (userIndex / 1000.0, rawFullDatasetCount /
1000.0))

        return (fullDataset, fullLabels)
```

Appendix B: Main training code, train.py

```
#
#  Created by Si Te Feng on Stampy Day, Jul/13/16.
#  Data queried by Dean Hillan
#  Copyright c 2016 Paperless Post. All rights reserved.
#
#

'''
This file constructs a fully connected neural network that
can predict the chance of the user selecting certain gift
card based on previous send history. The ANN is trained on
Paperless Post customer card sending data with Stochastic
Gradient Descent method with mini-batches.

The training data is SQL queried and parsed from red shift
and contains all the sent cards from all users from
 2015/01/01 to 2016-07-13
'''

import tensorflow as tf
import numpy as np
from random import randint
from DataParser import DataParser
from ArrayFormatter import ArrayFormatter


# Convenience Functions for main code
# Get the mini-batch from the full dataset and label matrices
# BatchNum starts at 0
def getBatch(batchNum, dataset, label, batchSize):
    batchStart = batchNum * batchSize
    batchEnd = batchStart + batchSize
    return (dataset[batchStart: batchEnd], label[batchStart:
batchEnd])


# Cost function
def squaredErrorCost(output, target):
    diffTensor = output - target
    cost = tf.square(diffTensor)
    return cost


# Importing and parsing data
print("Reading input CSV into raw dataset...")
rowsToRead = 200000
print("Reading rows: %d" % rowsToRead)

parser = DataParser()
rawFullDataset, packageIds =
parser.readCSV(fileName="accounts_packages.csv",
maxReadRows=rowsToRead)
```

```
# Shuffling the raw dataset
print("Shuffling raw input dataset...")
arrayFormatter = ArrayFormatter()
rawFullDataset = arrayFormatter.shuffle(rawFullDataset)

# Separating full datasets into processed tensors required by
TensorFlow
print("Transforming parsed data into training format...")

uniquePackageCount = len(packageIds)
fullDataset, fullLabels =
parser.getTrainingMatrixFromRawDataset(rawFullDataset,
uniquePackageCount, 10)
# Shuffle again
fullDataset = arrayFormatter.shuffle(fullDataset)
fullLabels = arrayFormatter.shuffle(fullLabels)

print("Finished transforming data: fullDataset[%d],
fullLabels[%d]" % (len(fullDataset), len(fullLabels)))


# Further separate into training and validation datasets
print("Further separating data into training and validation
datasets...")

fullDatasetSize = len(fullDataset)
validationSize = int(fullDatasetSize * 0.08)

trainDataset, validDataset =
parser.splitDatasetIntroTrainingAndValidation(fullDataset,
validationSize)
trainLabels, validLabels =
parser.splitDatasetIntroTrainingAndValidation(fullLabels,
validationSize)

print("Dataset separated into training and validation
portions.")
print("trainDataset[%d], trainLabels[%d] | validDataset[%d],
validLabels[%d]" %
      (len(trainDataset), len(trainLabels), len(validDataset),
len(validLabels)))

# Main training code with TensorFlow
print("Setting up Neural Network...")
# Constants
numTrain = len(trainDataset)
numValidation = len(validDataset)

numInput = uniquePackageCount * 2
numNodesL1 = 1000
numNodesL2 = 150
numNodesL3 = 10
numOutput = 1
```

```
batchSize = 25
stdDeviation = 0.3
learningRate = 0.15


# Start Training
inputImgs = tf.placeholder(tf.float32, shape=[batchSize,
numInput])
inputLabels = tf.placeholder(tf.float32, shape=[batchSize,
numOutput])

# x.get_shape() => [batchSize, numInput, 1]
x = tf.expand_dims(inputImgs, 2)

W01 = tf.Variable(tf.random_normal([batchSize, numNodesL1,
numInput], stddev=stdDeviation, dtype=tf.float32),
name="weight01")
b1 = tf.Variable(tf.constant(0.01, shape=[batchSize,
numNodesL1, 1]), name="bias1")

# Wx => [], b => []
z1 = tf.batch_matmul(W01, x) + b1

hidden1 = tf.nn.sigmoid(z1, name="output1")

# Second hidden layer
W12 = tf.Variable(tf.random_normal([batchSize, numNodesL2,
numNodesL1], stddev=stdDeviation), name="weight12")
b2 = tf.Variable(tf.constant(0.01, shape=[batchSize,
numNodesL2, 1]), name="bias2")
z2 = tf.batch_matmul(W12, hidden1) + b2

hidden2 = tf.nn.sigmoid(z2, name="output2")

# Hidden Layer 2
W23 = tf.Variable(tf.random_normal([batchSize, numNodesL3,
numNodesL2], stddev=stdDeviation), name="weight23")
# z3.get_shape() => [batchSize, outputSize, 1]
z3 = tf.batch_matmul(W23, hidden2)
hidden3 = tf.nn.sigmoid(z3, name="output3")

# Output layer (Layer 3)
W34 = tf.Variable(tf.random_normal([batchSize, numOutput,
numNodesL3], stddev=stdDeviation), name="weight34")
temp_z4 = tf.batch_matmul(W34, hidden3)

z4 = tf.reshape(temp_z4, [batchSize, numOutput])
y4 = tf.nn.sigmoid(z4, name="output4")

y_ = tf.reshape(inputLabels, [batchSize, numOutput])

outputErrors = squaredErrorCost(y4, y_)
avgError = tf.reduce_mean(outputErrors)

# Minimize error though backpropagation
```

```
global_step = tf.Variable(0, name='global_step')
optimizer = tf.train.AdagradOptimizer(learningRate)
trainOp = optimizer.minimize(avgError,
global_step=global_step)

# Initialize Session and Variables
sess = tf.Session()
sess.run(tf.initialize_all_variables())

print("Training Neural Network...")
stepsToTrain = numTrain // batchSize
# Due to limited computing power, steps have been limited to
1000
stepsToTrain = stepsToTrain

batchNum = 0
for step in xrange(stepsToTrain):
    (batch_x, batch_t) = getBatch(batchNum, trainDataset,
trainLabels, batchSize)

    _, currError, output, target = sess.run([trainOp,
avgError, y4, y_], feed_dict={inputImgs: batch_x, inputLabels:
batch_t})
    print("TrainStep[%d/%d], Error[%f]" % (step, stepsToTrain,
currError))
    batchNum += 1


# Post training validation, Classification Accuracy
outputToTargetDiff = tf.abs(tf.round(y4) - tf.round(y_))
outputToTargetNeg = tf.sub(outputToTargetDiff, 1)
outputToTargetEquality = tf.neg(outputToTargetNeg)
classificationAccuracy =
tf.reduce_mean(tf.cast(outputToTargetEquality, tf.float32))

print("Validating Neural Network Accuracy...")

stepsToValidate = int(numValidation / batchSize)
accuracySum = 0
batchNum = 0
for step in xrange(stepsToValidate):
    (valid_batch_x, valid_batch_t) = getBatch(batchNum,
validDataset, validLabels, batchSize)

    currAccuracy, diff, neg =
sess.run([classificationAccuracy, outputToTargetDiff,
outputToTargetNeg], feed_dict={inputImgs: valid_batch_x,
inputLabels: valid_batch_t})

    accuracySum += currAccuracy
    batchNum += 1

percentAccuracy = 100 * accuracySum / stepsToValidate
print("Steps to Validate: %d" % stepsToValidate)
print("Validation Accuracy: [%f%%]" % percentAccuracy)
```

```
# Saving neural network
# saver = tf.train.Saver()
# saver.save(sess, "graph1")
# saver.export_meta_graph("/graphs/graph1.meta")

sess.close()
```