Contents

1 Introduction

Artificial Neural Networks are computational systems that mimic the way biological neural networks in the brain process information. They have seen use in a wide variety of applications including medical diagnosis, speech and handwriting recognition, object recognition, game-playing, and control systems. There are a large number of different network structures and learning approaches. The goal of this project is to gain an elementary understanding of a simple network structure and learning algorithm, the feed-forward Multi-Layer Perceptron structure and Backpropagation algorithm. Furthermore, said structure and learning algorithm will be adapted to a real-world application, collaborative filtering

The concepts and methods presented in this report were largely adapted from the following textbooks:

- Deep Learning ([1]),
- Parallel Distributed Processing, specifically the chapter named Learning Internal Representations By Error Propagation ([2]), and
- the course textbook, Networked Life ([3]).

First, a high-level overview of the network structure and learning algorithm will be presented. An implementation of a neural network based collaborative filtering model will then be examined.

2 The Neural Network Structure

For this project, a simplistic neural network structure will be considered. The neural network is represented as a directed, weighted graph organised into a number of layers. The layers consist of an input layer, an output layer and a number of hidden layers, with each layer containing a varying number of vertices. This network structure is called a feed-forward Multi-Layer Perceptron (MLP) network.

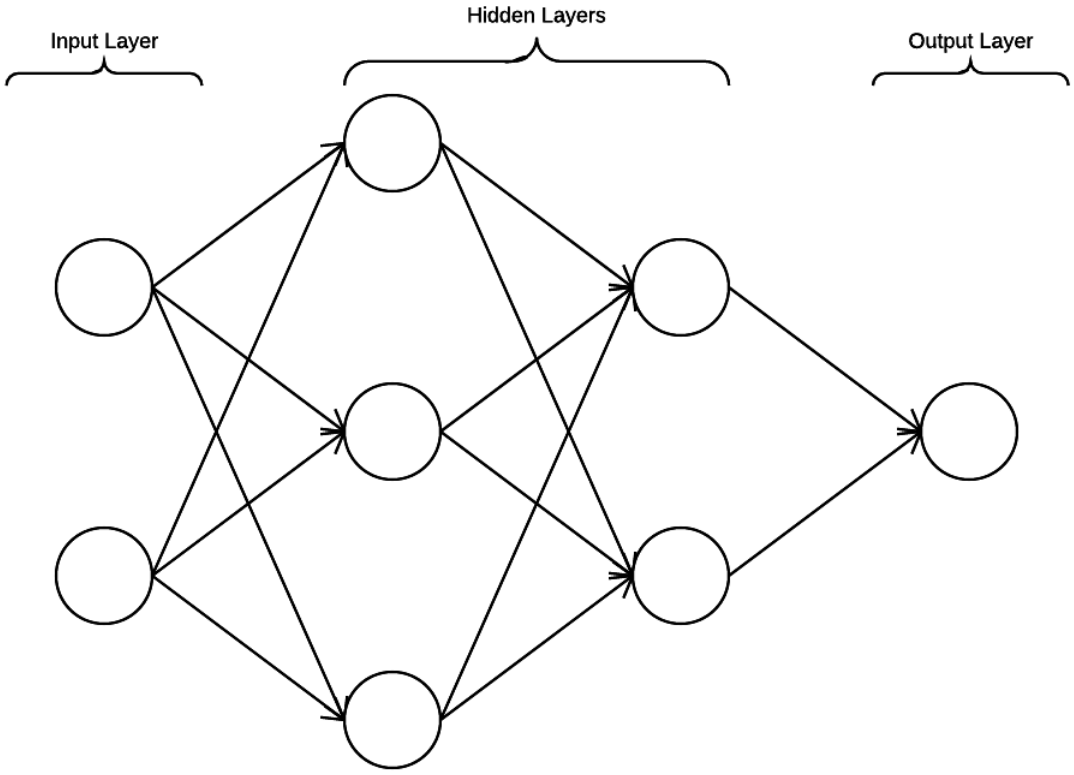

Figure 1: A feed-forward MLP, featuring an input layer with 2 nodes, 2 hidden layers with 3 and 2 nodes respectively, and an output layer with 1 node

For the sake of clearer and simpler notation, only networks with a single output value will be considered.

Layers are arranged in order and nodes in a particular layer only have edges to nodes in the immediately proceeding layer. In other words, edges are only between adjacent layers and only 'flow' in one direction, from the input layer to the output layer.

The goal of the network is to calculate an estimate for a given set of inputs. For example, one may wish to calculate the probability of a horse winning a race given it's win/loss ratio, placement in previous races, age, etc.

The set of n inputs of the network is defined as a vector of input parameters, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. For this set of inputs, the true output parameter is

defined as y and the network's estimated of the true parameter as $\hat{y}$.

Each node in the network has an input value and an output value. Let l_i^* denote the input value and l_i the output value of the i^{th} node in layer l.

Node k_i is called a parent node of l_j if there exists an edge from k_i to l_j. As one would expect, l_j is the child node of k_i in this situation. Let $w_{k_i \to l_j}$ denote the weight of the edge from node k_i to l_j.

Nodes in the input layer receive their input values from the input parameter vector, with each element corresponding to a node in the layer. The remaining nodes' input values are calculated by taking the sum of the weighted output values from parent nodes. The output value of a node is calculated by applying an activation function to its input value.

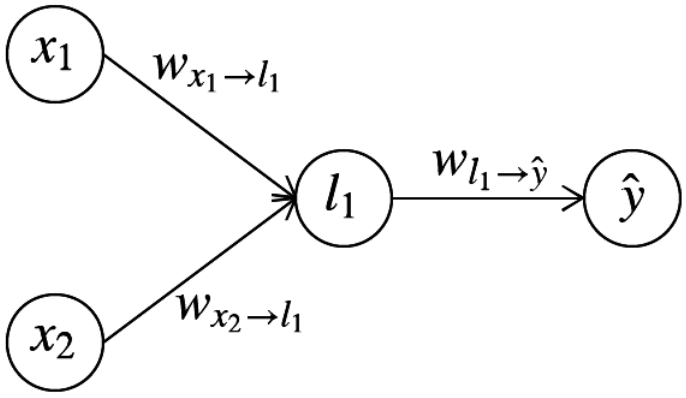

Figure 2: A simple network structure with labelled nodes and edge weights

For example, consider the network in figure 2. Let $f_{l_1}(x)$ be the activation function for node l_1 and $f_{\hat{y}}$ the activation function for $\hat{y}$. Then,

$$
\begin{aligned}
l_1^* &= (x_1 \times w_{x_1 \to l_1}) + (x_2 \times w_{x_2 \to l_1}) \\
l_1 &= f_{l_1}(l_1^*), \text{ and} \\
\hat{y}^* &= l_1 \times w_{l_1 \to \hat{y}} \\
\hat{y} &= f_{\hat{y}}(\hat{y}^*)
\end{aligned}
$$

In general, for node l_i with parent nodes $k_1, k_2, ..., k_n$ and activation function $f(x)$:

$$l_i^* = \sum_{j=1}^{n} (k_j \times w_{k_j \to l_i}), \text{ and}$$
$$l_i = f(l_i^*)$$

There are a wide variety of activation functions, each with various properties. For example, two commonly used functions are the logistic function, $f(x) = \frac{1}{1+e^{-x}}$, and the hyperbolic tan function, $f(x) = \frac{2}{1+e^{-2x}} - 1$. The choice between the two depends upon the desired strength of the activation gradient (how this affects the network will be discussed in a later section). For this report, let $f(x)$ denote an arbitrary, differentiable activation function.

3 Backpropagation

Backpropagation utilizes the error of the network's estimate, $\hat{y}$, of the true value y in order to learn. There are 3 main components of backpropagation:

- defining a cost function for the network,
- minimising the cost function with respect to each weight in the network, and
- updating the weights of the network.

3.1 The Cost Function

The cost function estimates how closely the network output approximates the true output. There are a variety of cost functions used in machine learning and the choice of cost function largely depends upon the domain of inputs and outputs and the desired properties of the learning algorithm.

For example, cross-entropy, $C = -(y \log(\hat{y}) + (1-y) \log(1-\hat{y}))$, is commonly used in classification problems, problems in which one is trying to predict the class a particular object belongs to. However, cross-entropy generally does not work for regression problems, problems in which one is trying to estimate a continuous value given a set of inputs. This is because the output domain of a classification problem is strictly a probability (the probability

of an object belonging to a class) whereas regression problems attempt to estimate a real value.

The cost function needs to be chosen with great care as it forms the crux of backpropagation. A poorly chosen function could throttle the algorithms speed or worse, prevent an acceptible solution from being found.

In general, an ideal cost function should be quickly computable and produce a smooth, differentiable, convex curve in its used context, allowing it to be easily minimised. The necessity of these properties will be explained in the following section.

For this report, the quadratic cost function, $C = \frac{1}{2}(\hat{y} - y)^2$, will be used as it is easily differentiable, commonly used for regression problems and is often convex in higher dimensions.

3.2 Minimising the Cost Function

Learning in backpropagation is essentially the act of modifying edge weights in order to minimise the cost function. Minimisation of the cost function is done with gradient descent

Gradient descent is an iterative minimisation technique that involves differentiating the cost function and moving 'down' the slope. This is why the cost function needs to be differentiable and quickly computable over the required domain.

Gradient descent is often used in machine learning as it's a general technique and is often computationally cheaper than analytical solutions. For many problems, there may not even exist a closed form analytical solution. It does, however, have serious limitations.

The goal of the algorithm is to find the global minima of the cost function. However, since it involves simply moving towards negative gradients, there is the possibility of being 'trapped' in a local minima (or even worse, starting on a maxima). This is why the cost function needs to be convex over the required domain.

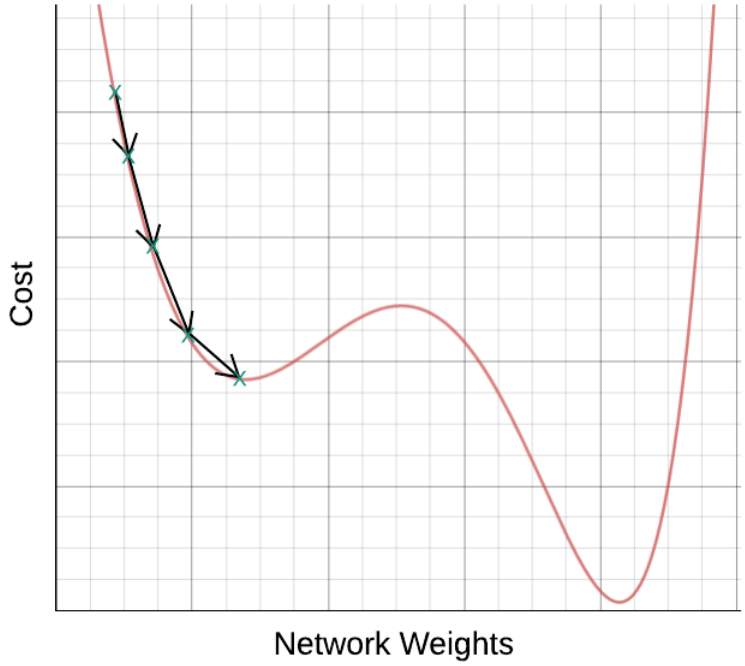

Figure 3: Getting stuck in a local minima while performing gradient descent

In order to actually minimise the cost function of the network, the partial derivative of the cost function is calculated with respect to each weight in the network.

3.2.1 Calculating the Derivatives by Hand

To form a general method of backpropagation, an example will first be used to demonstrate calculating the derivatives by hand.

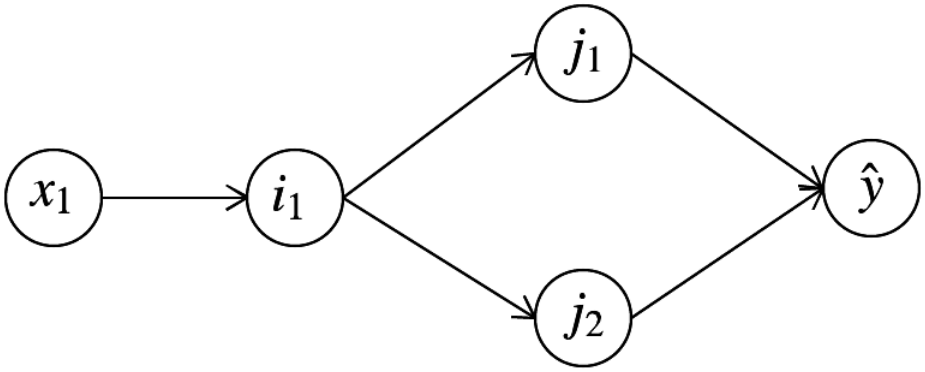

Figure 4: An example network that backpropagation will be performed on

The derivatives of the network in figure 4 will be calculated. The process begins with calculating the partial derivative of the cost function with respect to an edge going to the output layer, $w_{j_1 \to \hat{y}}$:

$$\begin{aligned}
\frac{\partial C}{\partial w_{j_1 \to \hat{y}}} &= \frac{\partial}{\partial w_{j_1 \to \hat{y}}}(\frac{1}{2}(\hat{y} - y)^2), \text{ using the cost function } C = \frac{1}{2}(\hat{y} - y)^2 \\
&= (\hat{y} - y)\frac{\partial}{\partial w_{j_1 \to \hat{y}}}(\hat{y} - y) \\
&= (\hat{y} - y)\frac{\partial \hat{y}}{\partial w_{j_1 \to \hat{y}}}, \text{ (since y is just a constant)} \\
&= (\hat{y} - y)\frac{\partial \hat{y}}{\partial \hat{y}^*}\frac{\partial \hat{y}^*}{\partial w_{j_1 \to \hat{y}}} \text{ (using chain rule on } \hat{y} = f(\hat{y}^*)) \\
&= (\hat{y} - y)f'(\hat{y}^*)\frac{\partial \hat{y}^*}{\partial w_{j_1 \to \hat{y}}} \\
&= (\hat{y} - y)f'(\hat{y}^*)\frac{\partial}{\partial w_{j_1 \to \hat{y}}}(j_1 w_{j_1 \to \hat{y}} + j_2 w_{j_2 \to \hat{y}}), \text{ (using } \hat{y}^* = j_1 w_{j_1 \to \hat{y}} + j_2 w_{j_2 \to \hat{y}}) \\
&= (\hat{y} - y)f'(\hat{y}^*)j_1, \text{ (since } w_{j_2 \to \hat{y}} \text{ is constant w.r.t } w_{j_1 \to \hat{y}})
\end{aligned}$$

The derivative with respect to $w_{j_2 \to \hat{y}}$ follows a similar calculation and yields:

$$\frac{\partial C}{\partial w_{j_2 \to \hat{y}}} = (\hat{y} - y)f'(\hat{y}^*)j_2$$

Now compute the partial derivative with respect to the next set of preceding weights in the network. The derivative with respect to $w_{i_1 \to j_1}$ is as

follows:

$$\begin{aligned}
\frac{\partial C}{\partial w_{i_1 \to j_1}} &= \frac{\partial}{\partial w_{i_1 \to j_1}}(\frac{1}{2}(\hat{y} - y)^2) \\
&= ... \\
&= (\hat{y} - y) f'(\hat{y}^*) \frac{\partial}{\partial w_{i_1 \to j_1}}(j_1 w_{j_1 \to \hat{y}} + j_2 w_{j_2 \to \hat{y}}) \\
&= (\hat{y} - y) f'(\hat{y}^*) w_{j_1 \to \hat{y}} \frac{\partial j_1}{\partial w_{i_1 \to j_1}}, \text{ (since } w_{j_1 \to \hat{y}} \text{ and } j_2 w_{j_2 \to \hat{y}} \text{ are constant w.r.t } w_{i_1 \to j_1}) \\
&= (\hat{y} - y) f'(\hat{y}^*) w_{j_1 \to \hat{y}} \frac{\partial j_1}{\partial j_1^*} \frac{\partial j_1 *}{\partial w_{i_1 \to j_1}} \\
&= (\hat{y} - y) f'(\hat{y}^*) w_{j_1 \to \hat{y}} f'(j_1^*) \frac{\partial j_1^*}{\partial w_{i_1 \to j_1}} \\
&= (\hat{y} - y) f'(\hat{y}^*) w_{j_1 \to \hat{y}} f'(j_1^*) \frac{\partial}{\partial w_{i_1 \to j_1}}(i_1 w_{i_1 \to j_1}) \\
&= (\hat{y} - y) f'(\hat{y}^*) w_{j_1 \to \hat{y}} f'(j_1^*) i_1
\end{aligned}$$

The derivative with respect to $w_{i_1 \to j_2}$ also has a similar calculation and yields:

$$\frac{\partial C}{\partial w_{i_1 \to j_2}} = (\hat{y} - y) f'(\hat{y}^*) w_{j_2 \to \hat{y}} f'(j_2^*) i_1$$

Now for the next set of weights, namely $w_{x_1 \to i_1}$. Note that the values of all

nodes except for x_1 depend upon $w_{x_1 \to i_1}$:

$$\begin{aligned}
\frac{\partial C}{\partial w_{x_1 \to i_1}} &= (\hat{y} - y) f'(\hat{y}^*) \frac{\partial}{\partial w_{x_1 \to i_1}} (j_1 w_{j_1 \to \hat{y}} + j_2 w_{j_2 \to \hat{y}}) \\
&= (\hat{y} - y) f'(\hat{y}^*) (\frac{\partial}{\partial w_{x_1 \to i_1}} (j_1 w_{j_1 \to \hat{y}}) + \frac{\partial}{\partial w_{x_1 \to i_1}} (j_2 w_{j_2 \to \hat{y}}) \\
&= (\hat{y} - y) f'(\hat{y}^*) (w_{j_1 \to \hat{y}} \frac{\partial j_1}{\partial w_{x_1 \to i_1}} + w_{j_2 \to \hat{y}} \frac{\partial j_2}{\partial w_{x_1 \to i_1}}) \\
\frac{\partial j_1}{\partial w_{x_1 \to i_1}} &= \frac{\partial j_1}{\partial j_1^*} \frac{\partial j_1^*}{\partial w_{x_1 \to i_1}} \\
&= f'(j_1^*) \frac{\partial}{\partial w_{x_1 \to i_1}} (i_1 w_{i_1 \to j_1}) \\
&= ... \\
&= f'(j_1^*) w_{i_1 \to j_1} f'(i_1^*) x_1 \\
\frac{\partial j_2}{\partial w_{x_1 \to i_1}} &= f'(j_2^*) w_{i_1 \to j_2} f'(i_1^*) x_1 \\
\therefore \frac{\partial C}{\partial w_{x_1 \to i_1}} &= (\hat{y} - y) f'(\hat{y}^*) (w_{j_1 \to \hat{y}} f'(j_1^*) w_{i_1 \to j_1} f'(i_1^*) x_1 + w_{j_2 \to \hat{y}} f'(j_2^*) w_{i_1 \to j_2} f'(i_1^*) x_1) \\
&= (\hat{y} - y) f'(\hat{y}^*) (w_{j_1 \to \hat{y}} f'(j_1^*) w_{i_1 \to j_1} + w_{j_2 \to \hat{y}} f'(j_2^*) w_{i_1 \to j_2}) f'(i_1^*) x_1
\end{aligned}$$

Despite the messiness, it's evident that as one derives with respect to weights closer to the input layer, the derivatives from proceeding layers are chained together. This is the essence of backpropagation, the error at the output layer is calculated and then propagated backwards through the network.

3.2.2 The General Form of Backpropagation

Let δ_k denote as the cost gradient with respect to the **input** value for all non-input layer nodes k. In other words:

$$\delta_k = \frac{\partial C}{\partial k^*}$$

Thus:

$$\begin{aligned}
\delta_k &= \frac{\partial}{\partial k^*}(\frac{1}{2}(\hat{y}-y)^2) \\
&= (\hat{y}-y)\frac{\partial \hat{y}}{\partial k^*} \\
&= (\hat{y}-y)\frac{\partial \hat{y}}{\partial k}\frac{\partial k}{\partial k^*}, \text{ (using chain rule on } k = f(k^*)) \\
&= (\hat{y}-y)f'(k^*)\frac{\partial \hat{y}}{\partial k}
\end{aligned}$$

The term $\frac{\partial \hat{y}}{\partial k}$ represents the change in the network's output parameter, $\hat{y}$, with respect to node k's output value. Hence for node $\hat{y}$, the term is simply equal to 1 and so $\delta_{\hat{y}} = (\hat{y}-y)f'(\hat{y}^*)$.

For nodes other than $\hat{y}$, k's output value affects all of its children nodes. The output value of k is the only value being changed and the child nodes of k do not affect each other. Therefore, each child nodes' input value can be viewed as a function of k. Thus, the chain rule and a summation over k's children can be applied:

$$\begin{aligned}
\delta_k &= (\hat{y}-y)f'(k^*)\sum_l \frac{\partial \hat{y}}{\partial l^*}\frac{\partial l^*}{\partial k}, \text{ where nodes } l \text{ are the children of } k \\
&= (\hat{y}-y)f'(k^*)\sum_l \frac{\partial \hat{y}}{\partial l^*}w_{k\to l}, \text{ (since the change in } l^* \text{ w.r.t } k \text{ is simply the weight } w_{k\to l}) \\
&= f'(k^*)\sum_l (\hat{y}-y)\frac{\partial \hat{y}}{\partial l^*}w_{k\to l} \\
&= f'(k^*)\sum_l (\hat{y}-y)f'(l^*)\frac{\partial \hat{y}}{\partial l}w_{k\to l} \\
&= f'(k^*)\sum_l \delta_l w_{k\to l}
\end{aligned}$$

This is now a recursive relation between δ_k and k's children. Applying this general form to the earlier example yields:

$$\begin{aligned}
\delta_{\hat{y}} &= (\hat{y}-y)f'(\hat{y}^*) \\
\delta_{j_1} &= f'(j_1^*)(\delta_{\hat{y}}w_{j_1\to\hat{y}}) \\
\delta_{j_2} &= f'(j_2^*)(\delta_{\hat{y}}w_{j_2\to\hat{y}}) \\
\delta_{i_1} &= f'(i_1^*)(\delta_{j_1}w_{i_1\to j_1} + \delta_{j_2}w_{i_1\to j_2})
\end{aligned}$$

The derivative of each weight with respect to the cost function now has a much simpler form:

$$\begin{aligned}
\frac{\partial C}{\partial w_{j_1 \to \hat{y}}} &= \delta_{\hat{y}} j_1 \\
\frac{\partial C}{\partial w_{j_2 \to \hat{y}}} &= \delta_{\hat{y}} j_2 \\
\frac{\partial C}{\partial w_{i_1 \to j_1}} &= \delta_{j_1} i_1 \\
\frac{\partial C}{\partial w_{i_1 \to j_2}} &= \delta_{j_2} i_1 \\
\frac{\partial C}{\partial w_{x_1 \to i_1}} &= \delta_{i_1} x_1
\end{aligned}$$

One can reconstruct the hand-calculated derivatives above by expanding each δ term.

3.3 Updating Edge Weights

With the gradient of the cost function with respect to each weight, it's now possible to modify the weights in order to minimise the cost function. The goal is to modify the weights such that the cost function moves towards its negative gradient. Thus for weight $w_{k \to l}$:

$$\begin{aligned}
w'_{k \to l} &= w_{k \to l} - \alpha \frac{\partial C}{\partial w_{k \to l}} \\
&= w_{k \to l} - \alpha \delta_l k
\end{aligned}$$

where $w'_{k \to l}$ is the new weight and α is the learning rate. The learning rate affects how aggressively the network pursues the negative cost gradient. Some care needs to be taken when selecting the learning rate. Generally, a higher learning rate produces a faster learning network. However if it's too high, then the network will have difficulty converging to a minimum cost, slowing down the learning speed.

Earlier in the report, it was noted that the choice of activation function could depend on the desired gradient strength. As with the learning rate, the gradient strength of the activation function affects how aggressively the algorithm tries to minimise the cost function. The activation function needs

to produce a gradient that learns sufficiently quickly and does not have difficulty converging.

3.4 Learning

Once the edge weights have been updated, the new predicted value and its error can now be recomputed. The process of predicting values and updating edge weights is repeated until the error converges.

With a sufficiently trained network, one can make predictions on inputs with unknown true outputs. The network can be retrained as the true outputs are obtained. For example, with collaborative filtering the network can guess the rating a particular user will give an item. Upon receiving the users actual rating, it can be used to retrain the network, improving its accuracy.

4 Batch Processing

For the sake of clean and simple notation, a network structure that only considers a single set of inputs and outputs at a time was examined. However, the single processing approach has the obvious downside of only being able to train the network on one data set at a time. In general, networks are trained using batches of data sets (i.e. multiple sets of inputs and corresponding outputs) in order to take advantage of parallel computation. The single set approach can be trivially expanded to handle batches by using matrices.

Consider an arbitrary network with m input parameters. Then following expansions are needed to implement processing of a batch of n data sets:

- The input of the network is an $n \times m$ matrix, where rows correspond to data sets and columns correspond to input parameters.
- For layer k, the input values of all nodes are represented with an $n \times |k|$ matrix, where $|k|$ is the number of nodes in the layer. Let this matrix be K^*.
- The output value matrix of k, K, is derived by applying the activation function to all entries in the input matrix, i.e. $K = f(K^*)$.
- For layer l such that k has edges to l, the weights of such edges are represented by a $|k| \times |l|$ matrix. Each row in the matrix corresponds

to a node in k and each column corresponds to a node in l. Let this matrix be W.

- The $n \times |l|$ input value matrix of l is calculated by taking the matrix multiplication of K and W, i.e. $L^* = K \times W$.
- The output of the network will then be a $n \times o$ matrix, where o is the number of output parameters.

Backpropagation has a similar extension for batch processing. Using the same notation semantics as above:

- The cost gradient for all nodes in layer l is represented using a $n \times |l|$ matrix. As with the value matrices above, each row is the gradient of a data set and each column is a node in the layer. Let this matrix be D_l.
- The cost gradient matrix for layer k is computed as $f'(K^*) \circ (D_l \times W^T)$, where $\circ$ denotes point-wise matrix multiplication.
- The updated weight matrix, W', is calculated as $W' = W - \alpha(K^T \times D_l)$

5 Collaborative Filtering With Neural Networks

Collaborative filtering utilises patterns in the way users rate items, such as Netflix movies or Amazon products, in order to predict future ratings. The textbook provides two approaches to collaborative filtering, the neighborhood model and the latent-factor model. The neighborhood model tries to predict ratings by considering similarities in ratings across users (or, alternatively, across items). The latent-factor model tries to predict ratings by modeling low-dimensional structures in the rating data.

For this project, a neural network approach was taken to a recommender system featuring a 1-to-5 rating system. The neural networks approach relied on similar concepts as the latent-factor approach. There may be low-level relations between users, items and ratings that we cannot provide an intuitive model for. Latent-factors models use taste and item appeal as vectors and the predicted rating is computed as their inner product. Latent-factors attempts to find such tastes and appeals by using alternative projections to optimise the vectors such that the prediction error is minimised. In contrast, the neural network attempted to encode the low-level relations into

the network structure by using the prediction error and backpropagation to self-correct its model.

5.1 Network Model

Let the number of users be n and number of items m. The network structure consisted of $n-1$ input parameters and 1 output parameter. For the hidden layers, a number of setups were tested. There were two setups that generally performed better than the rest:

1. $n-1$ nodes in the first hidden layer and 200 nodes in the second hidden layer.
2. $n-1$ nodes in the first hidden layer, 200 in the second, and 2 in the third hidden layer.

The logistic activation function was used for all nodes in the network and the quadratic cost function was used for error estimation.

For every user u, a separate network was modelled. The inputs of the network were the ratings given by every user (except for u) for a particular item i. The output of the network was u's predicted rating of i. Missing ratings were given a value of 0.

The network implemented batch processing. Thus the input was actually a matrix, with rows representing each item and columns representing each user. The output was a vector containing u's predicted rating for every item.

5.2 Implementation

The network was implemented in python. The network structure was provided by the Keras package, with TensorFlow used as the back-end. The details of the implementation is as follows.

First, the input data set was parsed. Two matrices were created, one for training and the other for testing. Entries in the matrices corresponded to ratings, with rows representing users giving the rating and columns representing rated objects. Missing ratings were given a value of 0. The only difference between the training matrix and testing matrix was that the training matrix treated test values as missing ratings.

Next, the network model was created. For each user u, the testing and training matrices were spliced, removing u's ratings from the matrices. The matrices were also transposed, so that the rows corresponded to movies (data sets) and columns corresponded to users(input parameters). The network was then trained using the transposed training matrix. Since the sigmoid function was used in the final layer, the network's training matrix was transformed using the sigmoid function. The trained network was then used to predict u's ratings based on the training and test ratings. Again, the testing matrix was transformed using the sigmoid function to account for the network's output. The RMSE of both predictions were then calculated.

After the network had being trained and tested for every user, the network's overall training and testing RMSE were calculated. As the results represented predicted ratings, the inverse sigmoid function was used to transform the network output back into ratings. The ratings were then clipped between 1 and 5 and missing ratings were represented accordingly.

5.3 Experimental Results and Evaluation

For the first set of experiments, the data set provided in the course textbook was used (`set1.txt` in the code repository):

$$\begin{bmatrix} 5 & 4 & 4 & - & \mathbf{5} \\ - & 3 & 5 & \mathbf{3} & 4 \\ 5 & 2 & - & \mathbf{2} & 3 \\ - & \mathbf{2} & 3 & 1 & 2 \\ 4 & - & \mathbf{5} & 4 & 5 \\ \mathbf{5} & 3 & - & 3 & 5 \\ 3 & \mathbf{2} & 3 & 2 & - \\ 5 & \mathbf{3} & 4 & - & 5 \\ \mathbf{4} & 2 & 5 & 4 & - \\ \mathbf{5} & - & 5 & 3 & 4 \end{bmatrix}$$

A similar notation for marking missing rankings and test data is used (rows represent users, columns represent items, '-' for missing ratings, and bold for test ratings).

As described in the Network Model section, setup 1 featured $n - 1$ nodes in the first hidden layer and 200 in the second. The intuition behind the 200-node hidden layer is that each node can represent the mapping of user

latent factors onto item latent factors. The $(n-1)$-node layer was added as the added complexity appeared to improve the network's accuracy. For the setup the network produced the following set of rating predictions:

$$\begin{bmatrix} 5 & 4.06 & 4.06 & - & \mathbf{4.88} \\ - & 3.06 & 5 & \mathbf{4.19} & 4.06 \\ 4.95 & 1.95 & - & \mathbf{3.5} & 2.95 \\ - & \mathbf{1.99} & 3.05 & 1.05 & 2.05 \\ 3.96 & - & \mathbf{4.77} & 3.96 & 4.96 \\ \mathbf{4.11} & 3.05 & - & 3.04 & 5 \\ 3.03 & \mathbf{2.1} & 3.04 & 2.04 & - \\ 4.97 & \mathbf{4.48} & 3.96 & - & 4.97 \\ \mathbf{4.89} & 2.05 & 5 & 4.04 & - \\ \mathbf{4.91} & - & 4.96 & 2.96 & 3.97 \end{bmatrix}$$

The RMSE of the training data only is 0.04 and the RMSE of the test data only is 0.87. The RMSE for both training and test data is 0.44.

As evident in the results, the implementation fitted the training data much more closely than the neighbourhood model used in the textbook. Given how closely the training data was fit and the relatively large variances in prediction accuracy across users, it's likely that the network was overfitting the training data.

Setup 2 simply augments setup 1 by adding an extra hidden layer that features 2 nodes. The intuition behind this added layer is that the nodes represent the particular user and item biases. The setup produced the following ratings:

$$\begin{bmatrix} 4.42 & 4.42 & 4.42 & - & \mathbf{4.42} \\ - & 3.7 & 3.7 & \mathbf{3.7} & 3.7 \\ 2.79 & 2.79 & - & \mathbf{2.79} & 2.79 \\ - & \mathbf{2.99} & 3 & 1 & 2.05 \\ 4.38 & - & \mathbf{4.38} & 4.03 & 4.38 \\ \mathbf{4.85} & 3.05 & - & 3.06 & 4.9 \\ 3.05 & \mathbf{2.03} & 3.05 & 2.0 & - \\ 4.99 & \mathbf{4.07} & 3.96 & - & 5 \\ \mathbf{4.79} & 2 & 4.99 & 3.95 & - \\ \mathbf{4.72} & - & 4.99 & 2.95 & 3.95 \end{bmatrix}$$

The RMSE of the training data only is 0.55 and the RMSE of the test data only is 0.7. The RMSE for both training and test data is 0.59.

A notable difference between setup 2 and setup 1 are the predicted ratings for the first three users. The network appears to have computed some sort of average across user ratings. Remarkably, this average is quite close to the test ratings. Furthermore, this only occurred for the first three users, the remaining users have ratings that are close to the training data.

Due to the presence of these averaged ratings, the RMSE of the training data was significantly worse. Whilst the RMSE of the test data is better than that of setup 1, the prediction of the test data does not appear to be significantly more accurate.

Its unclear why such behaviour arose. One suspicion is that the two nodes in the last hidden layer along with the sigmoid activation function acted as a bottleneck. A property of the sigmoid function is that negative values approach 0 and positive values approach 1 very quickly. Since a large number of nodes were feeding into the two nodes in the last layer its possible that the two nodes were consistently outputting values close to 0 or 1, regardless of the input parameters. In such a case, the optimisation of the network would then degenerate to a linear optimisation problem, thereby producing the average-like values.

The network was also tested on a somewhat larger data set (`set3.txt`). This data set featured 20 users giving ratings for 20 items. Due to the limitations of the implementation, this data set had to be hand-crafted. As a result, the data set may not represent any realistic rating patterns. The ratings are largely random, however, there is a slight degree of similarity between some users. As the matrices are quite large only the RMSE results will be examined:

- Setup 1: training only RMSE = 0.04, test only RMSE = 1.41, combined RMSE = 0.67
- Setup 2: training only RMSE = 0.13, test only RMSE = 1.47, combined RMSE = 0.7

The test RMSE results were significantly worse than the previous data set's results. This is likely due to the unrealistic nature of the data set. Arguably, the results of this experiment do not provide any significant conclusions, however, there are some interesting observations that can be made. In the data set, there were some groups of users that had very similar rating patterns. Some users also exhibited simulated biases (some gave low ratings in general and some gave higher ratings). For such users, the network was able predict test ratings reasonably well. Furthermore, despite the large er-

ror in the test predictions many of the predictions were reasonably close to the true vale. The error appears to be skewed by large deviations in a few predictions.

There were a number of other setups also tested besides the ones provided here. These other setups generally performed worse though. The details of these setups and performance can be found in the implementation's code.

5.4 Model Limitations and Possible Improvements

5.4.1 Missing Data

The representation of missing data is a significant issue inherent to neural networks. Ideally, one would minimise the effect that incorrect/inaccurate conclusions drawn as a result of missing data has on the network. In the implementation, missing ratings were given a value of zero. This of course creates a misrepresentation of data. The ideal outcome is that the network 'learnt' to ignore values of zero, but it could have just as likely treated 0 as another rating value.

Approaches to handle missing data generally requires careful engineering of the network structure. For example, the network could be designed such that nodes and weights affected **only** by missing data are ignored (nodes and weights affected by other existing values still need to be considered). Another approach involves predicting the values of missing data and feeding them back through the network. Unfortunately, the use of a package in the implementation made such fine tuning of the network inherently difficult.

5.4.2 Memory Costs

As the number of users increased, the memory cost of the network increased sharply. This is largely due to the use of dense networks, wherein each node has an edge to every node in the proceeding layer. The implementation was not able to handle just 25 movies from Netflix's data-set; it ran out of memory when attempting to create the network.

As with the missing data issue, the memory overhead can be reduced by carefully engineering a custom network structure. For example, a network

structure featuring clusters rather than a densely connected network would allow one to store the unused parts of the network on disk whilst processing. It would also allow parts of the network to be distributed across several machines and processed in parallel.

5.4.3 Time Complexity

It's certainly likely that the time taken to train the network for all users increases significantly with the number of users. With the data sets used in the experiment, it takes about 10 seconds to train the network for a single user. As with the memory cost, the denseness of the network is a large factor in the increase of computation time.

The need to train the network for every user also plays significant part in the computation time. Ideally, one would train the network for all users in a single training session. This can be done trivially by having an output parameter for each user. However, the network's inability to handle missing data prevents this solution from being viable. When training user u, the current implementation can only train using items that u has rated. Thus the network needs to be trained individually for each user (as different users may not rate the same set of items). Addressing the issue of missing data could, by proxy, remove the necessity of individually trained networks, thereby increasing performance significantly.

5.4.4 Data Set Density

As a result of the support data set size being so small, the provided dataset needs to be extremely dense in order to provide meaningful low-level relations. This made the use of any realistic data sets for testing impossible. Thus, it was difficult to accurately evaluate the performance of the implementation as such evaluations were performed using hypothetical data sets.

Furthermore, due to the density of the data set, the implementation does not feature any methods of handling sparse data sets. In practical applications, an encoding technique is usually used to handle the extremely sparse data sets inherent of recommender systems. Handling sparsity of data sets is another way of reducing the memory costs of the network.

5.4.5 Non-convex Cost Function

Another issue with the model relates to the minimisation of the cost function. By experimenting with the epoch value of Keras's fit function, it was found that the error of the network does not converge after a large number of iterations. Furthermore, the required number of iterations to get acceptable predictions was abnormally large. This could be caused by the lack of convergence or the edge learning rate and activation function may be producing an extremely slowly learning network.

This was somewhat expected as the used cost function is, of course, very general and does not provide any guarantees of convexity for all problems. It was chosen because it is very simple and 'good enough' for the prototype implementation. However for a real system, one would design a cost function that satisfies the needs of the system (i.e. convex, differentiable, and relatively fast).

5.4.6 Overfitting

As evident in the experiment results, the network appeared to overfit training data. Given the other problems with the network, particularly the missing data issue and lack of convergence, it's difficult to determine if this is really the case.

If overfitting is indeed an issue, then adding more complexity to the network may be a solution. Through testing, it appeared as though adding more complexity to the network (more layers) reduced the networks ability to fit the training data. This is likely due to the predictions of ratings being influenced by more low-level relations between users and items. In a realistic data set, this could improve the networks prediction accuracy. However, it could also introduce more noise into the predictions. It's difficult to make judgements regarding overfitting and the effects of possible solutions whilst the other issues are present.

6 Related Work

There have been a large number of applications of neural networks for collaborative filtering ([4]). Some of these approaches use neural networks to

supplement traditional recommender systems whereas others attempt recommendation using only neural networks. Some notable approaches that feature 'pure' neural network implementations will be examined.

Of the approaches examined here, Neural Collaborative Filtering (NCF, [5]) is the most conceptually similar. NCF uses MLP's to encode the low-level relations between user tastes and item properties. However, unlike the simplistic model, NCF learns using implicit feedback (watched movies, purchases, viewed items) rather than explicit feedback (user provided ratings). Furthermore, NCF combines user tastes and item properties into a single input vector and embeds the input to handle the sparseness inherent to the data.

AutoRec ([6]) is another similar model that uses a form of MLP, autoencoders. An autoencoder is, in essence, a 3-layer feed-forward MLP network, with an equal number of input and output parameters. The autoencoder is a specialisation of a MLP in the sense that it attempts to predict its input values, rather than some arbitrary target value. In doing this, it attempts to find an encoding of a data set that minimises the error on decoding. AutoRec takes the ranking vector of users (or items), encodes it into latent space, and reconstructs it back into the ranking space. AutoRec's method of input data handling could be applied to the simplistic implementation to solve the missing data issue.

Recurrent Recommender Networks (RRN, [7]) is an approach to collaborative filtering that uses a different kind of neural networks, recurrent neural networks. Unlike feed-forward networks, recurrent networks feature node links that can travel to arbitrary nodes in the network (regardless of the layer), allowing for loops in the network. Rather than explicitly predict the latent state of ratings for a given data set, RRN predicts the changes in ratings over time.

7 Conclusion and Future Work

In this report, the feed-forward MLP neural network and backpropagation learning algorithm were examined. An approach to collaborative filtering using such network structure and learning algorithm was implemented. Given the simplicity of the implementation, it managed to predict ratings users might give to items with reasonable accuracy.

Addressing the various issues with the implementation is certainly an area of future work. It would be interesting to see how much the performance improves if a custom, purpose-built network data structure is used. It would also be interesting to observe the performance of different network structures and learning algorithms. Of course, getting the implementation to run with realistic data sets, such as Netflix's data set, is a major necessity for properly evaluating and improving upon the implementation.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[3] Mung Chiang. *Networked Life: 20 Questions and Answers*. Cambridge University Press, 2012.

[4] Shuai Zhang, Lina Yao, and Aixin Sun. Deep learning based recommender system: A survey and new perspectives. *CoRR*, abs/1707.07435, 2017.

[5] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 173–182, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[6] Suvash Sedhain, Aditya Krishna Menon, Scott Sanner, and Lexing Xie. Autorec: Autoencoders meet collaborative filtering. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, pages 111–112, New York, NY, USA, 2015. ACM.

[7] Chao-Yuan Wu, Amr Ahmed, Alex Beutel, Alexander J. Smola, and How Jing. Recurrent recommender networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, pages 495–503, New York, NY, USA, 2017. ACM.

A Code and Libraries

The implementation and utilised libraries can be found at the following links:

- The implementation code: https://github.com/Orion-L/NetflixBackprop
- Keras: https://keras.io
- TensorFlow: https://www.tensorflow.org

The python scripts expects a data set text file to be passed as argument 1, e.g. `./backprop_filter.py data.txt`. The ratings are represented in the text file as a space separated matrix. Each row corresponds a user giving a rating and each column corresponds to the rated object. Missing ratings are represented with a '-' and test data is identified by placing a '?' after the rating. For example, the data set provided in the course text book is encoded as follows:

5	4	4	–	5?
–	3	5	3?	4
5	2	–	2?	3
–	2?	3	1	2
4	–	5?	4	5
5?	3	–	3	5
3	2?	3	2	–
5	3?	4	–	5
4?	2	5	4	–
5?	–	5	3	4