Applications of Linear Algebra in Neural Networks

## Introduction

Neural networks are the backbone of artificial intelligence (AI), a growing computational field. AI has taken the world by storm in the past couple of years, being implemented anywhere and everywhere, from well-established tech companies to business start-ups. Behind the fancy branding of AI, making consumers believe it is some magic human-like entity, lies the neural network. Chakrabarti (1995) comments on how the beginnings of the neural network can be found in the human brain. The idea behind them is that they should mimic networks of neurons, with each node in a neural network acting similarly in function to neurons in the brain. However, neural networks are constantly evolving and being improved, as detailed in Fan et al. (2021).

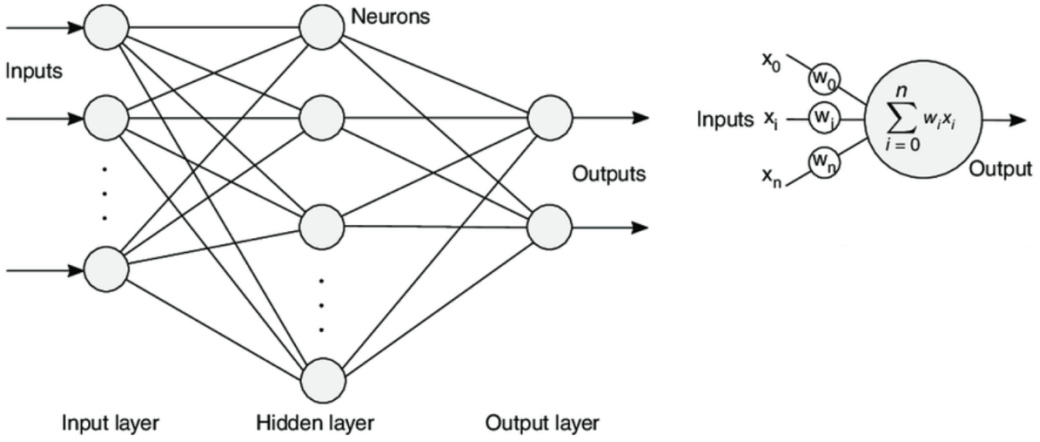


*Figure 1. A diagram of the workings of a neural network. Adapted from "A survey on graphic processing unit computing for large-scale data mining" by Cano, A. (2017).*

The idea is that each node in a layer receives a linear combination of all the nodes, or inputs, via weights, coefficients multiplying the input values, and biases, constants added to the product of the weights and inputs. The idea of these weights and biases are detailed in Higham & Higham (2019). This basic layer is known as a dense layer. In a neural network there is the input layer, which takes in the input values to be trained on, hidden layers, with values of each node

either determined by these linear combinations and output to the next layer or activation functions, and the output layer, which acts similarly to hidden layers but instead of outputting to the next layer it outputs the value of the network for the specified values. One common output layer for classification is the softmax layer, also used in Fan et al. (2021), that calculates the probability of all classifications that add up to one. Some hidden layers may also be called activation layers, which apply a non-linear activation function to allow the network to go beyond outputting a linear combination of inputs and solve more complex real-world problems that are non-linear, as detailed in West et al. (1997). The typical format is to follow every normal hidden layer with an activation layer to apply non-linearity at every step. In the hidden layers, all the weights and biases are randomized to start. The forward propagation consists of this passing of values forward through the network and is fairly simple. Both Higham & Higham (2019) and Fan et al. (2021) give an overview of the neurons' roles in this feed-forward approach.

The complex part, and the part that allows the network to "learn," is the backward propagation, utilizing a method called gradient descent that Zhang (2007) explores. The idea is that the error gradient is utilized to calculate how much the weights and biases of the neural network should change in order to achieve minimum error. This series of steps essentially calculates the loss, or error, of the output values for a certain input. In training, both the input values and the expected output value must be passed into the model so that the network can compare its output to the expected output, calculating said error value, also called an error gradient. This error gradient is then passed back through the network, using partial derivatives to calculate how much the weights and biases of each layer impacted the error of the output as well as a new error gradient to pass farther back, as both Fan et al. (2021) and Higham & Higham (2019) explain. The weights and biases of a specific layer are corrected by adding the error

gradient, with respect to the weights or biases respectively, multiplied by a learning rate. The learning rate is a hyperparameter of the model, seen in Higham & Higham (2019), set by the user during initialization of the model, and is generally a small value below one to assure the model converges and does not jump around.

**Methods**

This project utilizes the MNIST data set, a standard beginning project for neural networks (AstroDave & Cukierski, 2012). It consists of the data for written numbers in a 28 by 28 grid, with each pixel in the grid having a value. These 784 values serve as the input to the network, and the class of the entry, the expected output, is simply what written number it is. Data is split into two sets: the training set to train the model and help it learn, and the testing set to test the model on data it has not seen before. The neural network class was implemented from scratch in python, only using the numpy library to compute the matrix multiplication. The sample network consisted of a Dense layer, serving as the input layer, a Relu layer, an activation layer with the ReLU activation function that makes a value 0 if it is less than 0, another Dense layer, acting as a hidden layer, and the Softmax layer, that applies the softmax function and acts as the output layer. It also passed in the hyperparameters of 1000 iterations and a learning rate of 0.5. While the dataset serves as a good problem to solve, it is much simpler than many other real-world applications of neural networks.

In python, classes had to be constructed for the overall network as well as each custom layer. The Network class (Appendix A) initialization takes in 3 arguments: a list of the layers, the learning rate hyperparameter, and the epochs (number of times the model iterates through the training data). It has 2 methods. The train method consists of the previously described forward and backward propagation, and it does this process as many times as specified by the epoch. It

forward propagates by iterating through the list, passing in either the input or last output to the forward method of the next layer, until the output of the last layer is reached. The loss is then calculated, using a simple method of taking the average of the difference between the one-hot encoded n by 1 matrix, where n is the number of classes, of the real results and the outputted n by 1 matrix, where each row corresponds to the probability of a certain class. This loss is then fed as the error gradient back through the list in a reverse manner, where each layer's backward method updates anything that may need to be updated with respect to the error gradient and outputs the new calculated error gradient that will be fed to the previous layer. Once the error gradient is fed all the way to the input layer, all updates are made, and the network is ready for the next epoch. The predict method consists of forward propagation, passing the input or previous output to the next layer, eventually returning the final output of the model.

Moving onto the individual layer classes, there is a Layer parent class (Appendix B) that initializes its input and output variables to None in its construction. It passes the job of defining the forward and backward methods to its children classes.

The Dense class (Appendix C) is a child of the Layer class. Its constructor initializes the weights m by n matrix, where m is the size of the output size and n is the input size, specified by the constructor for the layer respectively, with randomized values between -0.5 and 0.5. A similar weight randomization was also done in West et al. (1997). Its forward method consists of setting the input to the new input, setting the output to the product of the weights and input plus the biases, and returning the output. The backward method consists of finding the error gradient for the previous layer by doing the product of the transposed weight matrix and the output gradient (Appendix D), subtracting from the weights the learning rate times the product of the output gradient and the transposed input matrix (Appendix E), subtracting from the biases the

learning rate times the sum of the output gradient for each class (Appendix F), and returning the new error gradient.

The Activation class (Appendix G) is also a child of the Layer class, but it is also the parent class to any activation layers. Its constructor takes in an activation function and an activation prime function, the derivative of the activation function, and sets them equal to its activation and activation prime instance variables, respectively. The forward method consists of setting the input to the new input of the layer, setting the output to result of the activation function being applied to the input, and returning the output. The backward method consists of setting the error gradient for the previous layer equal to the multiplication of each value in the output gradient with its corresponding value in the result of the activation prime function applied to the input and returning that error gradient (Appendix H).

The Relu class (Appendix I) is a child class of the Activation Class. It has the same forward and backward methods as the Activation class, it just specifies that the activation function is the ReLU function, also used in Fan et al. (2021), and the activation prime function is the ReLU prime function, or derivative of ReLU.

The Softmax class (Appendix J) is a child class of the Activation Class. It has the same forward and backward methods as the Activation class, it just specifies that the activation function is the softmax function and the activation prime function is the softmax prime function, or derivative of softmax.

It was previously mentioned that the network used had a list of layers, 0.5 learning rate, and 1000 epochs. The list of layers consisted of a Dense layer with 784 input size and 10 output size, a Relu layer, a Dense layer with 10 input size and 10 output size, and the Softmax layer (Appendix K). The original dataset is size 784 by 33600. Since each training instance has 784

variables, this means that each column is one training instance out of the 33600. In the first layer the product of the 10 by 784 weights matrix and the 784 by 33600 input matrix is taken to produce a 10 by 33600 matrix. The 10 by 1 biases matrix will be added on, essentially to each column or instance. The result of this will be the output of the first dense layer. For the Relu layer, the relu function is applied to the output from the first Dense layer, keeping the size 10 by 33600. This matrix is then fed into the second Dense layer where the product of the 10 by 10 weights matrix and the 10 by 33600 input matrix is taken to produce a 10 by 33600 matrix. For the Softmax layer, the softmax is applied to the output from the first Dense layer, keeping the size 10 by 33600.

Now for backward propagation. The one-hot encoded matrix (Appendix L) of the real values starts by creating a matrix of the size of the expected outputs by the number of possible classifications, in this case 33600 by 10. Then the matrix is traversed, setting the nth index of each row to 1, where n is the class of expected output for that instance, and transposing the matrix, making the size 10 by 33600. This will result in a matrix with a row for each instance with all 0's and a 1 to signify the class of the instance. Now the loss for the network can be calculated by taking the difference between the output and the one-hot encoded values for the expected output. This error gradient will start with a size of 10 by 33600 for this case. It will apply the softmax prime function to the 10 y 33600 inputs of the softmax layer and multiply each value with its corresponding value in the 10 by 33600 output gradient data for the layer, which will result in a 10 by 33600 sized error gradient passed to the previous layer.

The first dense layer will calculate the error gradient of the previous layer doing the product of the transposed 10 by 10 weights, remaining 10 by 10, and the 10 by 33600 error gradient, making the dimension for the error gradient passed to the previous layer 10 by 33600

also. The sum of the output gradient can be used to update the biases. The product of the output gradients, 10 by 33600, and the transpose of the inputs, 10 by 33600 to 33600 by 10, allows the 10 by 10 weights to be updated.

Again, the derivative for the activation function, this time relu, is taken for each value of the 10 by 33600 input and multiplied by its corresponding value in the 10 by 33600 output gradient matrix, keeping the 10 by 33600 shape. The calculated output gradient is passed along to the previous Dense layer.

The second dense layer is the end, so no error gradient calculation is needed. The sum of the output gradient can be used to update the biases. The product of the output gradients, 10 by 33600, and the transpose of the inputs, 784 by 33600 to 33600 by 784, allows the 10 by 784 weights to be updated.

**Results**

The model is trained on the training set, about 80% of the data, and has 92.6% accuracy. The more important number is the accuracy on the test set because it makes sure the model can generalize to data it has not seen (been trained) on. The model performed well with 91.4% accuracy on the test set. Fairly high accuracy, which makes sense considering the simplicity of the classification problem compared to other potential real-world implementations.

**Conclusions**

While neural networks have been trained before, this project was undertaken to explore the application of linear algebra, matrices, and even some partial differentials. It was done with the aim to understand the inner workings of neural networks, as they play an important role in the growing wave of artificial intelligence in the real world. Understanding the math behind the artificial intelligence can help extrapolate the methods to more complex problems while also

understanding the limitations. While the method of gradient descent applied by neural networks is the most commonly utilized method, it is one of many computer learning algorithms out there. Some methods may be better for certain situations, and understanding the math behind the methods can aid in applying the best algorithms for each situation and potentially discovering new ones in the future.

## Appendix

### Appendix A (Network Class)

```
class Network:

    def __init__(self, layers: list[Layer], learning_rate: float, epochs: int):

        self.layers = layers
        self.learning_rate = learning_rate
        self.epochs = epochs

    def train(self, X: np.ndarray, Y: np.ndarray):

        for epoch in range(self.epochs + 1):

            output = X

            for layer in self.layers:

                output = layer.forward(output)

            error_gradient = error(Y, output)
            # error_gradient = mse_prime(y, output)

            print(error_gradient.shape)

            for layer in reversed(self.layers):

                error_gradient = layer.backward(error_gradient, self.learning_rate)

            if epoch % 50 == 0:

                preds = get_preds(output)

                print(f"Epoch {epoch}")
```

```
                print(f"Accuracy: {np.sum(preds == Y) / Y.size}")
                print(f"Preds:\t {preds[:20]}")
                print(f"Real:\t {Y[:20]}")
                print()

    def predict(self, X):

        output = X

        for layer in self.layers:

            output = layer.forward(output)

        preds = get_preds(output)

        return preds
```

Appendix B (Layer Class)

```
class Layer:

    def __init__(self):

        self.input: np.ndarray = None
        self.output: np.ndarray = None

    def forward(self, input):

        pass

    def backward(self, output_gradient, learning_rate):

        pass
```

Appendix C (Dense Layer Class)

```
class Dense(Layer):

    def __init__(self, input_size, output_size):

        self.weights: np.ndarray = np.random.rand(output_size, input_size) - 0.5
        self.biases: np.ndarray = np.random.rand(output_size, 1) - 0.5

    def forward(self, input):

        self.input = input
        self.output = self.weights.dot(self.input) + self.biases
```

```
        return self.output

    def backward(self, output_gradient: np.ndarray, learning_rate: float):

        error_gradient = self.weights.T.dot(output_gradient)
        self.weights -= learning_rate * output_gradient.dot(self.input.T)
        self.biases -= learning_rate * np.sum(output_gradient)

        return error_gradient
```

Appendix D (Input Error Gradient)

$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix}$$

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} \\ \frac{\partial E}{\partial x_2} \\ \vdots \\ \frac{\partial E}{\partial x_j} \end{bmatrix}$$

$$Y \begin{cases} y_1 = x_1 w_{11} + x_2 w_{12} + \cdots + x_i w_{1i} + b_1 \\ y_2 = x_1 w_{21} + x_2 w_{22} + \cdots + x_i w_{2i} + b_2 \\ \vdots \\ y_j = x_1 w_{j1} + x_2 w_{j2} + \cdots + x_i w_{ji} + b_j \end{cases}$$

$$\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_1} + \frac{\partial E}{\partial y_2}\frac{\partial y_2}{\partial x_1} + \cdots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_1} = \frac{\partial E}{\partial y_1} w_{11} + \frac{\partial E}{\partial y_2} w_{21} + \cdots + \frac{\partial E}{\partial y_j} w_{j1}$$

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_1} w_{1i} + \frac{\partial E}{\partial y_2} w_{2i} + \cdots + \frac{\partial E}{\partial y_j} w_{ji}$$

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial y_1} w_{11} + \frac{\partial E}{\partial y_2} w_{21} + \cdots + \frac{\partial E}{\partial y_j} w_{j1} \\ \frac{\partial E}{\partial y_1} w_{12} + \frac{\partial E}{\partial y_2} w_{22} + \cdots + \frac{\partial E}{\partial y_j} w_{j2} \\ \vdots \\ \frac{\partial E}{\partial y_1} w_{1i} + \frac{\partial E}{\partial y_2} w_{1i} + \cdots + \frac{\partial E}{\partial y_j} w_{ji} \end{bmatrix}$$

$$\frac{\partial E}{\partial X} = \begin{bmatrix} w_{11} & w_{21} & \dots & w_{j1} \\ w_{12} & w_{22} & \dots & w_{j2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & \dots & w_{ij} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix}$$

$$\frac{\partial E}{\partial X} = W^T \cdot \frac{\partial E}{\partial Y}$$

Appendix E (Weights Error Gradient)

$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix}$$

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \frac{\partial E}{\partial w_{12}} & \dots & \frac{\partial E}{\partial w_{1i}} \\ \frac{\partial E}{\partial w_{21}} & \frac{\partial E}{\partial w_{22}} & \dots & \frac{\partial E}{\partial w_{2i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{j1}} & \frac{\partial E}{\partial w_{j2}} & \dots & \frac{\partial E}{\partial w_{ji}} \end{bmatrix}$$

$$Y \begin{cases} y_1 = x_1 w_{11} + x_2 w_{12} + \cdots + x_i w_{1i} + b_1 \\ y_2 = x_1 w_{21} + x_2 w_{22} + \cdots + x_i w_{2i} + b_2 \\ \vdots \\ y_j = x_1 w_{j1} + x_2 w_{j2} + \cdots + x_i w_{ji} + b_j \end{cases}$$

$$\frac{\partial E}{\partial w_{12}} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial w_{12}} + \frac{\partial E}{\partial y_2}\frac{\partial y_2}{\partial w_{12}} + \cdots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{12}} = \frac{\partial E}{\partial y_1} x_2$$

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} x_i$$

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial y_1} x_1 & \frac{\partial E}{\partial y_1} x_2 & \dots & \frac{\partial E}{\partial y_1} x_i \\ \frac{\partial E}{\partial y_2} x_1 & \frac{\partial E}{\partial y_2} x_2 & \dots & \frac{\partial E}{\partial y_2} x_i \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial y_j} x_1 & \frac{\partial E}{\partial y_j} x_2 & \dots & \frac{\partial E}{\partial y_j} x_i \end{bmatrix}$$

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix} \cdot \begin{bmatrix} x_1 & x_2 & \dots & x_3 \end{bmatrix}$$

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial Y} \cdot X^T$$

Appendix F (Biases Error Gradient)

$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix}$$

$$\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial b_1} \\ \frac{\partial E}{\partial b_2} \\ \vdots \\ \frac{\partial E}{\partial b_j} \end{bmatrix}$$

$$Y \begin{cases} y_1 = x_1 w_{11} + x_2 w_{12} + \cdots + x_i w_{1i} + b_1 \\ y_2 = x_1 w_{21} + x_2 w_{22} + \cdots + x_i w_{2i} + b_2 \\ \vdots \\ y_j = x_1 w_{j1} + x_2 w_{j2} + \cdots + x_i w_{ji} + b_j \end{cases}$$

$$\frac{\partial E}{\partial b_1} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial b_1} + \frac{\partial E}{\partial y_2}\frac{\partial y_2}{\partial b_1} + \cdots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial b_1} = \frac{\partial E}{\partial y_1}$$

$$\frac{\partial E}{\partial b_1} = \frac{\partial E}{\partial y_1} \rightarrow \frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial Y} = \frac{\partial E}{\partial B}$$

Appendix G (Activation Layer Class)

```
class Activation(Layer):

    def __init__(self, activation, activation_prime):

        self.activation: function = activation
        self.activation_prime: function = activation_prime

    def forward(self, input):

        self.input = input
        self.output = self.activation(self.input)

        return self.output

    def backward(self, output_gradient, learning_weight):

        error_gradient = output_gradient * self.activation_prime(self.input)

        return error_gradient
```

Appendix H (Activation Error Gradient)

$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix}$$

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} \\ \frac{\partial E}{\partial x_2} \\ \vdots \\ \frac{\partial E}{\partial x_j} \end{bmatrix}$$

$$Y \begin{cases} y_1 = f(x_1) \\ y_2 = f(x_2) \\ \vdots \\ y_i = f(x_i) \end{cases}$$

$$\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_1} + \frac{\partial E}{\partial y_2}\frac{\partial y_2}{\partial x_1} + \cdots + \frac{\partial E}{\partial y_i}\frac{\partial y_i}{x_1} = \frac{\partial E}{\partial y_1} f'(x_1)$$

$$\frac{\partial E}{\partial x_i} = = \frac{\partial E}{\partial y_i} f'(x_i)$$

$$\frac{\partial E}{\partial X} = = \frac{\partial E}{\partial Y} \odot f'(X)$$

Appendix I (Relu Activation Layer Class)

```
class Relu(Activation):

    def __init__(self):

        relu: function = ReLU
        relu_prime: function = deriv_ReLU
        super().__init__(relu, relu_prime)
```

Appendix J (Softmax Activation Layer Class)

```
class Softmax(Activation):

    def __init__(self):

        softmax: function = SoftMax
        softmax_prime: function = deriv_SoftMax
        super().__init__(softmax, softmax_prime)
```

Appendix K (Test Network)

```
net = Network(
```

```
    [
        Dense(784, 10),
        Relu(),
        Dense(10, 10),
        Softmax()
    ],
    0.5,
    1000
)
```

Appendix L (One-Hot Encoding Method)

```
def one_hot(Y: np.ndarray):

    one_hot_Y = np.zeros((Y.size, Y.max() + 1))

    one_hot_Y[np.arange(Y.size), Y] = 1

    # Transpose matrix
    one_hot_Y = one_hot_Y.T

    return one_hot_Y
```

## References


AstroDave, & Cukierski, W. (2012). *Digit recognizer* [Data set]. Kaggle. https://www.kaggle.com/competitions/digit-recognizer/data

Cano, A. (2017). A survey on graphic processing unit computing for large-scale data mining. *WIREs Data Mining and Knowledge Discovery*, *8*(1). https://doi.org/10.1002/widm.1232

Chakrabarti, B. K. (1995). Neural networks. *Current Science*, *68*(2), 153-155.

Dolson, M. (1989). Machine tongues xii: Neural networks. *Computer Music Journal*, *13*(3), 28-40. https://doi.org/10.2307/3680009

Fan, J., Ma, C., & Zhong, Y. (2021). A selective overview of deep learning. *Statistical Science*, *36*(2), 264-290.

Higham, C. F., & Higham, D. J. (2019). Deep learning: An introduction for applied mathematicians. *SIAM Review*, *61*(4), 860-891.

West, P. M., Brockett, P. L., & Golden, L. L. (1997). A comparative analysis of neural networks and statistical methods for predicting consumer choice. *Marketing Science*, *16*(4), 370-391.

Zhang, C.-H. (2007). Continuous generalized gradient descent. *Journal of Computational and Graphical Statistics*, *16*(4), 761-781.