From Matrices to Models: Neural Networks Explained

Author: Jackson Eshbaugh

Institution: Lafayette College

Course: MATH 272—Linear Algebra with Applications

Instructor: Yunied Puig de Dios

December 8, 2024

From Matrices to Models: Neural Networks Explained

Jackson Eshbaugh

December 8, 2024

Abstract

This paper investigates the mathematical foundation and application of linear algebra in artificial neural networks (ANNs), a widely used model in machine learning (ML). We begin by explaining the mathematical building blocks of ML algorithms, highlighting how their structure relies on linear algebra. The discussion then transitions from theory to practice by implementing an ANN to solve the XOR problem and running a prediction through manual calculations.

1 Introduction

Artificial Intelligence (AI) has captivated the world in recent years, moving from theoretical models to practical applications that assist us daily—like virtual chatbots answering our questions [1]. While AI and Machine Learning (ML) can appear complex, they are rooted in mathematics. Because computers operate entirely on numbers, we can peel back the layers of AI and ML to reveal the mathematical principles that drive them. At the heart of these principles is linear algebra, a mathematical framework that enables the algorithms and data transformations driving AI. By understanding these fundamentals, we can better grasp how AI systems tackle complex problems like recognizing faces, translating languages, or predicting patterns. This study focuses on the application of the tools of linear algebra to tackle problems central to AI.

2 Linear Algebra and Computer Science: Complimentary Fields

Linear algebra has numerous uses throughout the field of computer science. Of course, one of the most common uses is in AI and ML—the focus of this paper—but linear algebra is also used in sectors like computer graphics, where vectors are used to build wire-frames of 3D models [2]. Indeed, the advent of the vector as a way to store and analyze data as opposed to other more traditional data structures has been revolutionary for computer science [3]. It is this idea that allows AI and ML to work. In fact, the computation of dot products makes all of the advancements of AI and ML possible. In neural networks, for example, dot products calculate the weighted sums of inputs and weights, enabling the network to predict.

Similarly, in recommendation systems, vectors represent user preferences and item features, with dot products quantifying their similarity.

3 ML Basics

Before we explore the math behind ML, we must clarify what ML is and how we define an ML model. Machine learning is "essentially a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions" [4]. Essentially, we can use computers to statistically estimate answers to complex questions that would otherwise take an huge amount of time to compute. As can be seen in Figure 1, a machine learning algorithm can be described by four items [4]:

- **A dataset**: The data that the algorithm will operate on (to start). Includes both training data and output data.
- **A model**: The mathematical representation or framework used to learn patterns or relationships from data. It is the structure that maps inputs (features) to outputs (predictions) using a set of parameters that are optimized during the training process.
- **A cost function**: An error function that will quantify how far off from the true values the predicted values are, guiding the model's learning process. We will denote the cost function as $J(\mathbf{w}, b)$, where $\mathbf{w}$ represents the weights from a linear model and b represents the biases. One example of a cost function is mean square error (MSE):

$$J(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\mathbf{w}^\top x_i + b))^2 \tag{3.1}$$

- **An optimization procedure**: A procedure that finds the best set of parameters (weights and biases) that minimize $J(\mathbf{w}, b)$. Training an ML model can be understood as minimizing the cost function. Typically, a technique called gradient descent is used, where parameters are updated in the opposite direction of $\nabla J(\mathbf{w}, b)$, the gradient of J.

Sometimes, we need more than just a simple prediction algorithm when using ML for certain tasks. This is where deep learning comes in. Deep learning is an expansion of ML where multiple layers of computation are performed. This can be modeled as a composite function. If we define our model as $f(\mathbf{x})$, and we assign it two hidden layers and an output layer, then the whole model can be described as

$$f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x}))) \tag{3.2}$$

where $f^{(1)}$ is the first layer of the neural network, $f^{(2)}$ is the second, and $f^{(3)}$ is the output layer [5].

A layer of a network can have any number of nodes n, and the input vector to this layer $\mathbf{x} \in \mathbb{R}^n$. Hence, the transition from a layer of n nodes to a layer of m nodes can be described as the linear transformation

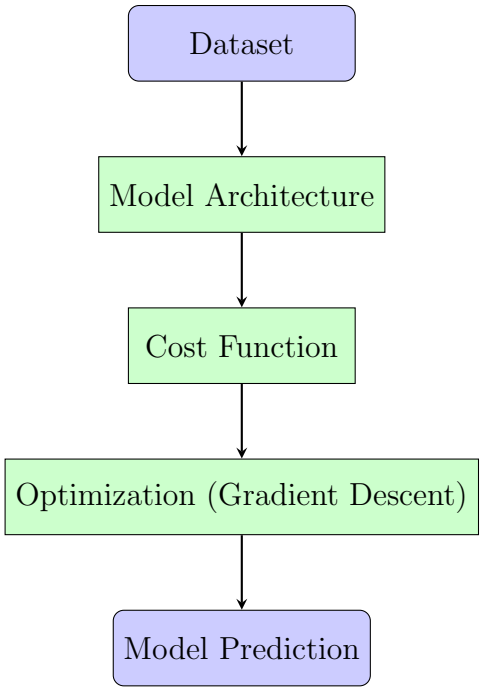

Figure 1: A machine learning algorithm makes predictions based on four objects: a dataset, a model, a cost function, and an optimization algorithm.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \longrightarrow \boxed{T} \longrightarrow \mathbf{y} = \mathbf{W}\mathbf{x} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Figure 2: A visualization of a linear transformation $T : \mathbb{R}^2 \to \mathbb{R}^2$ that might be used in a machine learning algorithm.

$$T : \mathbb{R}^n \to \mathbb{R}^m \tag{3.3}$$

Equation 3.3 implies that there exists some matrix that corresponds to this transformation. This matrix is denoted by $\mathbf{W}$ and represents the weight accrued by each component of $\mathbf{x}$ as it transitions from the layer in $\mathbb{R}^n$ to the layer in $\mathbb{R}^m$. Figure 2 gives an example of a linear transformation in an ML context.

The neural network as we've currently defined it is great at predicting one class of functions: *linear functions*. But not all data we'd like to approximate are linear. This leads to a natural question: how do we introduce non-linearity into our model? We find the solution in the other component of a layer in a neural network: the activation function. This function is run on the vector just before it is passed to the next layer. A common example of an activation function is the Rectified Linear Unit (ReLU).

$$g(z) = \max(0, z) \tag{3.4}$$

ReLU has been proven to be very effective when used as an activation function [6] and its gradient is much simpler to compute than other common activation functions, like sigmoid

(Equation 3.5) and hyperbolic tangent (Equation 3.6).

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.5}$$

$$\tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1} \tag{3.6}$$

Having discussed the key terminology and the construction of machine learning algorithms and deep learning neural networks, we will now proceed to implement a network that will solve a simple problem.

4 Creating an Artificial Neural Network: XOR

In this section, we implement a mathematical model that learns the XOR ($\oplus$) function. It is implemented in the form of an artificial neural network (ANN), specifically as a multilayer perceptron (MLP). MLPs are the most basic form of ANNs and are the simplest to understand. This discussion expounds upon chapter 6.1 of Goodfellow's *Deep Learning* [5]. XOR is a function defined in boolean algebra[1] by the following truth table. The function returns 1 if and only if one of A, B is 1 while the other is 0.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

4.1 Naïve (Linear) Solution

4.1.1 Defining the ML Algorithm

First, we need to define our ML algorithm f. We'll use the D as the dataset.

$$D = \left\{ d_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, d_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, d_3 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, d_4 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$$

We use the function $f^*(\mathbf{x}) = x_1 \oplus x_2$ to define the expected outputs for each test case in the vector $\mathbf{y}$.

$$\mathbf{y} = \begin{bmatrix} f^*(d_1) \\ f^*(d_2) \\ f^*(d_3) \\ f^*(d_4) \end{bmatrix} = \begin{bmatrix} 0 \oplus 0 \\ 0 \oplus 1 \\ 1 \oplus 0 \\ 1 \oplus 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Additionally, the matrix $\mathbf{X}$ will contain the input values for each test case.

[1] Boolean algebra, discovered by George Boole, forms the basis of computer science. It is the algebra of the base 2 number system (binary).

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Next, we select the mean square error (MSE) function as the cost function $J(\mathbf{w}, b)$ for the model. There are other cost functions that are more commonly used in ML, but MSE makes the computation more straightforward.

$$J(\mathbf{w}, b) = \frac{1}{4} \sum_{\mathbf{d} \in D} (f^*(\mathbf{d}) - f(\mathbf{d}, \mathbf{w}, b))^2 \tag{4.1}$$

Finally we define the model's linear form using the function

$$f(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^\top \mathbf{w} + b \tag{4.2}$$

Now that we have a network defined, we teach it using the training data in D.

4.1.2 Training the ANN

Typically, training is done using gradient descent and back-propagation, where the network structure is updated backwards. The most brute-force way to accomplish this is by using batch gradient descent, modeled by the equation

$$\mathbf{w} = \mathbf{w} - \eta \cdot \nabla J(\mathbf{w}) \tag{4.3}$$

where $\mathbf{w}$ is the vector of weights, η is the learning rate, and $J(\mathbf{w})$ is the cost function for the model [7].

There are ways that this process can be optimized which are out of the scope of this paper, but the curious reader can read Ruder's overview of gradient descent optimization techniques [2].

This is typically how training is done—since minimizing a function with thousands of inputs directly is computationally heavy—but due to the simplicity of the network we define (namely, since it has only two input layers and one output layer), we simply minimize the function J directly, with respect to $\mathbf{w}$ and b.

Before we begin this process, it will be helpful to redefine $\mathbf{X}$ and $\mathbf{w}$ such that the first entry of $\mathbf{w}$ is b. Hence,

$$\mathbf{X} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

and

$$\mathbf{w} = \begin{bmatrix} b \\ w_1 \\ w_2 \end{bmatrix}$$

[2] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: ArXivabs/1609.04747 (2016). url: https://api.semanticscholar.org/CorpusID:17485266.

Now, consider the matrix form of the MSE equation. We simplify before we compute the gradient.

$$\begin{aligned} J(\mathbf{w}) &= \frac{1}{4}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w}) \\ &= \frac{1}{4}(\mathbf{y}^\top - \mathbf{X}^\top\mathbf{w}^\top)(\mathbf{y} - \mathbf{X}\mathbf{w}) \\ &= \frac{1}{4}(\mathbf{y}^\top\mathbf{y} - \mathbf{y}\mathbf{X}^\top\mathbf{w}^\top + \mathbf{X}^\top\mathbf{w}^\top\mathbf{X}\mathbf{w} - \mathbf{y}^\top\mathbf{X}\mathbf{w}) \end{aligned}$$

We simply further by observing that $(\mathbf{X}\mathbf{w}^\top\mathbf{y}) = (\mathbf{y}^\top\mathbf{X}\mathbf{w})$ since the dot product is scalar and symmetric.

Hence,

$$J(\mathbf{w}) = (\mathbf{y}^\top\mathbf{y} - 2\mathbf{y}^\top\mathbf{X}\mathbf{w} + \mathbf{X}^\top\mathbf{w}^\top\mathbf{X}\mathbf{w}) \tag{4.4}$$

We now use calculus to obtain the minimum of J.

$$\begin{aligned} \frac{\delta J(\mathbf{w})}{\delta \mathbf{w}} = \nabla J(\mathbf{w}) &= \frac{1}{4}(-2\mathbf{X}^\top\mathbf{y} + 2\mathbf{X}^\top\mathbf{X}\mathbf{w}) \\ &= \frac{1}{2}(\mathbf{X}^\top\mathbf{X}\mathbf{w} - \mathbf{X}^\top\mathbf{y}) \end{aligned}$$

To find the minimum, set $\nabla J(\mathbf{w}) = \mathbf{0}$.

$$\begin{aligned} \mathbf{0} &= \frac{1}{2}(\mathbf{X}^\top\mathbf{X}\mathbf{w} - \mathbf{X}^\top\mathbf{y}) \\ \implies \mathbf{0} &= \mathbf{X}^\top\mathbf{X}\mathbf{w} - \mathbf{X}^\top\mathbf{y} \\ \implies \mathbf{X}^\top\mathbf{y} &= \mathbf{X}^\top\mathbf{X}\mathbf{w} \end{aligned}$$

In order to isolate $\mathbf{w}$, multiply both sides by $(\mathbf{X}^\top\mathbf{X})^{-1}$.

$$\begin{aligned} (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y} &= (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{X}\mathbf{w} \\ \implies (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y} &= \mathbf{w} \end{aligned} \tag{4.5}$$

Now, we calculate some products that we will use when solving for $\mathbf{w}$. Recall

$$\mathbf{X} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

and

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$$\implies \mathbf{X}^\top\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 2 & 2 \\ 2 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix} \tag{4.6}$$

Notice that the product $\mathbf{X}^\top\mathbf{X}$ is invertible by the IMT, since $\det(\mathbf{X}^\top\mathbf{X}) = 4 \neq 0$. Furthermore, its inverse is given by

$$(\mathbf{X}^\top\mathbf{X})^{-1} = \begin{bmatrix} \frac{3}{4} & -\frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{bmatrix} \tag{4.7}$$

Finally,

$$\mathbf{X}^\top\mathbf{y} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} \tag{4.8}$$

Now, to solve for $\mathbf{w}$, we use the results from equations 4.6, 4.7, and 4.8.

$$\begin{aligned} \mathbf{w} &= (\mathbf{X}^\top\mathbf{X})^{-1}(\mathbf{X}^\top\mathbf{y}) \\ &= \begin{bmatrix} \frac{3}{4} & -\frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \frac{1}{2} \\ 0 \\ 0 \end{bmatrix} \\ \implies b = \frac{1}{2}, \mathbf{w} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned} \tag{4.9}$$

Our naïve attempt simply outputs $\frac{1}{2}$ for all inputs. This is problematic for multiple reasons. For one, XOR is a function in boolean algebra, so we should obtain a binary output. Additionally, different inputs to XOR should produce different outputs. This attempt fails to solve XOR because XOR is inherently non-linear (see Figure 3).

4.2 Deep Learning Solution

We solve this problem by transforming the naïve algorithm into a neural network—we add a hidden layer. This layer allows us to transition to another space where the problem is represented linearly. Then, a linear model can be used (see Figure 3).

We define our hidden layer as follows.

$$\mathbf{h} = g(\mathbf{W}^\top\mathbf{x} + \mathbf{c}) \tag{4.10}$$

where $\mathbf{W}$ is the matrix of weights of the linear transformation, $\mathbf{c}$ is the vector of biases, and $g(z)$ is the ReLU (Rectified Linear Unit) function given in 3.4. $g(z)$ is applied *element-wise*,

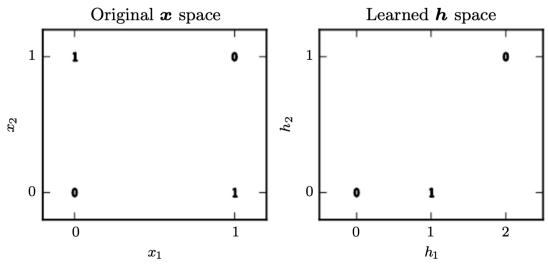

Figure 3: The solutions to non-linear problems are found in converting them to linear problems, which linear models can then analyze (figure courtesy of Goodfellow [5]).

so for each element in the resulting vector, g will be evaluated with that element and the element will be updated as the resulting value. The vector $\mathbf{x} \in \mathbb{R}^2$ in this layer; that is, $\mathbf{h}$ consists of 2 neurons.

The output of this layer is then fed through the same linear model we developed before, which produces a prediction. The graphical form of this new model is represented in Figure 4. The function representing our entire model is

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \sigma(\mathbf{w}^\top \cdot \max(0, \mathbf{W}^\top \mathbf{x} + \mathbf{c}) + b) \tag{4.11}$$

where $\mathbf{x}$ represents the input vector, $\mathbf{W}$ is the matrix of weights for the transformation to the hidden layer, $\mathbf{c}$ is the vector of biases for each node in the hidden layer, $\mathbf{w}$ is the vector of weights for the given linear transformation, and b is the bias for the one output node.

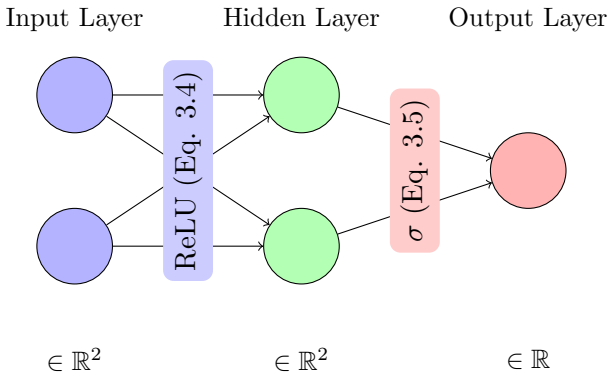

Figure 4: A map of the proposed neural network. Each layer can be thought of as a linear transformation whose matrix corresponds to the weights between each neuron. The activation function chosen for each layer is applied to each neuron on that layer after the weights from the previous layer are applied. Layers can utilize different activation functions.

A process like gradient descent or minimization will lead to the following parameters for the neural network.

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$

We substitute these parameters into the network function f.

$$f(\mathbf{x}) = \sigma(\begin{bmatrix}1 & -2\end{bmatrix} \cdot \max\left(0, \begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}\mathbf{x} + \begin{bmatrix}0\\ -1\end{bmatrix}\right) + 0) \tag{4.12}$$

Using various values for the input vector $\mathbf{x}$ and the network's trained model f given in Equation 4.12 we can predict the XOR function. Let $\mathbf{x} = \begin{bmatrix}0\\ 1\end{bmatrix}$. Then, we have

$$f\left(\begin{bmatrix}0\\ 1\end{bmatrix}\right) = \sigma\left(\begin{bmatrix}1 & -2\end{bmatrix} \cdot \max\left(0, \begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}0\\ 1\end{bmatrix} + \begin{bmatrix}0\\ -1\end{bmatrix}\right) + 0\right)$$

Simplifying the inside of the ReLU function, we gain

$$\begin{aligned} f\left(\begin{bmatrix}0\\ 1\end{bmatrix}\right) &= \sigma\left(\begin{bmatrix}1 & -2\end{bmatrix} \cdot \max\left(0, \begin{bmatrix}1\\ 1\end{bmatrix} + \begin{bmatrix}0\\ -1\end{bmatrix}\right) + 0\right) \\ &= \sigma\left(\begin{bmatrix}1 & -2\end{bmatrix} \cdot \max\left(0, \begin{bmatrix}1\\ 0\end{bmatrix}\right) + 0\right) \end{aligned}$$

Now, ReLU is applied to each element in $\begin{bmatrix}1\\ 0\end{bmatrix}$, which yields the same vector. We proceed by evaluating the transformation to the output layer, which is just sigmoid applied to the dot product of $\mathbf{w}^\top$ and the output from ReLU.

$$\begin{aligned} f\left(\begin{bmatrix}0\\ 1\end{bmatrix}\right) &= \sigma\left(\begin{bmatrix}1 & -2\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} + 0\right) \\ &= \sigma\left(\begin{bmatrix}1 & -2\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix}\right) \\ &= \sigma(1) \\ &= 0.7310586 \end{aligned} \tag{4.13}$$

To convert to binary, we round our output, that is,

$$\left\lfloor f\left(\begin{bmatrix}1\\ 0\end{bmatrix}\right)\right\rceil = 1$$

Indeed, $0 \oplus 1 = 1$, so our network functions as expected!

5 Conclusion

All types of AI and ML dissolve to linear algebra and gradient descent, forming the foundation of systems as simple as the network described above and as sophisticated as state-of-the-art models. While this network operates on two inputs, most neural networks process thousands—if not millions—of inputs, enabling applications from image recognition to language understanding. For instance, a neural network trained to classify handwritten digits[3] can achieve accuracy rates exceeding 99% [8], showcasing the remarkable potential of these

[3]The curious reader can read the appendix for an example of this network.

systems. The ability to estimate minima using gradient descent and backpropagation has made neural networks indispensable in modern AI. While we continue to push the boundaries of what these models can achieve, we are reminded that the complexities of their behavior ultimately stems from the simplicity and elegance of mathematical principles and structures, primarily those given in linear algebra.

Supplementary Appendix: Implementation in Python 3

In the appendix, we discuss the implementation of the neural network we defined to solve the XOR problem (presented in Section 4) both with and without ML libraries. We then discuss the implementation of the digit classification ANN mentioned in Section 5. THe code is available in Jupyter notebook file format on GitHub at `https://github.com/jacksoneshbaugh/From-Matrices-to-Models`.

A.A: XOR Network Implemented with Pure Python

This ANN is based on the discussion in section 4. In this section, you'll find the code needed to manually implement the ANN that solves XOR , along with annotations and comments that reference parts of the paper.

Firstly, we setup our imports, and we define the two activation functions we'll use later, which are ReLU (Equation 3.4) and sigmoid (Equation 3.5). Notice that we define both to operate on *each entry in a vector or matrix*—this is key.

```
import numpy as np
from numpy import floating

# Equation (3.4)
def relu(z) -> np.ndarray:
rn np.maximum(0, z)

# The derivative of Equation (3.4)
def relu_derivative(z: np.ndarray) -> np.ndarray:
    return (z > 0).astype(float)

# Equation (3.5)
def sigmoid(z: np.ndarray) -> np.ndarray:
    return 1 / (1 + np.exp(-z))

# The derivative of Equation (3.5)
def sigmoid_derivative(z: np.ndarray) -> np.ndarray:
    return sigmoid(z) * (1 - sigmoid(z))
```

Next, we define our:

- input data (`X`), a matrix;
- expected outputs (`y`), a vector;
- and our hyperparameters.

The hyperparameters describe the structure of the network (how many neurons in each layer) and learning algorithm parameters (the learning rate and number of epochs to train for). Specifically, `learning_rate` is η in Equation 4.3, and `training_epochs` represents how many times we should run the training algorithm. These numbers both must be tuned on

a network-by-network basis—trial and error is almost guaranteed. Specifically, training too much can lead to an overfit algorithm (and too little leads to an underfit algorithm) [9].

```
# Setup input/output data for xor
X: np.ndarray = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y: np.ndarray = np.array([[0], [1], [1], [0]])

# Hyperparameters
input_size: int = 2              # 2 neurons in the input layer
hidden_layer_size: int = 2       # 2 neurons in the hidden layer
output_size: int = 1             # 1 neuron in the output layer

learning_rate: float = 0.1      # eta in equation 4.3

training_epochs: int = 10000     # how many times to run the training
    algorithm (fine tune
                                 # this number, since the network can
                                     over- or under-fit if
                                 # this number is too large or small
                                    --see [9] for more info)
```

Now, we initialize the weights and biases for the network to random values. `np.random.randn(rows, columns)` generates a random `ndarray` (matrix or vector) of numbers. Then, as described in Equation 3.3, transitions between layers can be thought of as linear transformations, such that:

- $T_1 : \mathbb{R}^2 \to \mathbb{R}^2$ (input layer $\to$ hidden layer) has weight matrix W_1 and bias vector b_1
- $T_2 : \mathbb{R}^2 \to \mathbb{R}$ (hidden layer $\to$ output layer) has weight matrix W_2 and bias vector b_2

We scale the random numbers—in matrices W_1 and W_2, we use what is called Xavier initialization. This scales the weights so that the variance of the activations is the same across layers, helping to ensure that the network converges to a global minimum during training [10].

```
# Initialize weights and biases

np.random.seed(38)  # for reproducibility

# T_1 : input layer -> hidden layer (R^2 -> R^2)
W1: np.ndarray = np.random.randn(input_size, hidden_layer_size) * np
   .sqrt(2. / input_size) # Scale the weights
                    # so that the variance of activations is
                    # the same across layers (Xavier Initialization)
b1: np.ndarray = np.random.randn(1, hidden_layer_size) * 0.1

# T_2 : hidden layer -> output layer (R^2 -> R)
```

```
W2: np.ndarray = np.random.randn(hidden_layer_size, output_size) *
   np.sqrt(2. / input_size)
b2: np.ndarray = np.random.randn(1, output_size) * 0.1
```

Next, it's time to train the network. This is done in a two-step process:

1. We do a forward pass, computing the current network estimation. We calculate the cost function—in the case of this network, MSE: $J(\mathbf{w}, b) = \frac{1}{4}\sum_{\mathbf{d}\in D}(f^*(\mathbf{d}) - f(\mathbf{d}, \mathbf{w}, b))$ (Equation 4.1)

2. We perform batch gradient descent and backpropagation, described by Equation 4.3: $\mathbf{w} = \mathbf{w} - \eta \cdot \nabla J(\mathbf{w})$

```
# Train the network
for epoch in range(training_epochs):
# Forward pass
    z1: np.ndarray = np.dot(X, W1) + b1 # Evaluate the inside of the
        ReLU function
    a1: np.ndarray = relu(z1) # Apply ReLU

    z2: np.ndarray = np.dot(a1, W2) + b2
    a2: np.ndarray = sigmoid(z2)

    error: np.ndarray = y - a2
    cost: floating = np.mean(error**2)

    # Backpropagation (Equation 4.3)
    d_a2: np.ndarray = error * sigmoid_derivative(a2)            #
       Estimate gradient at output
    d_W2: np.ndarray = np.dot(a1.T, d_a2)                        #
       Get the weight update for W2
    d_b2: np.ndarray = np.sum(d_a2, axis=0, keepdims=True)       #
       Get the bias update for b2

    d_a1: np.ndarray = np.dot(d_a2, W2.T) * relu_derivative(a1)  #
       Estimate gradient at hidden layer
    d_W1: np.ndarray = np.dot(X.T, d_a1)                         #
       Get the weight update for W1
    d_b1: np.ndarray = np.sum(d_a1, axis=0, keepdims=True)       #
       Get the bias update for b1

    # Update the weights and biases
    W1 += learning_rate * d_W1
    b1 += learning_rate * d_b1
    W2 += learning_rate * d_W2
    b2 += learning_rate * d_b2

    # Print the cost occasionally
```

```
    if epoch % 1000 == 0:
        print(f'Epoch: {epoch}, Cost: {cost}')
```

Our network is trained and good to go! Now, we test it with all the values we trained it on (Equations 4.12 and 4.13).

```
# Testing the network
print("Predictions after training:")
for i in range(len(X)):
    z1: np.ndarray = np.dot(X[i], W1) + b1
    a1: np.ndarray = relu(z1)
    z2: np.ndarray = np.dot(a1, W2) + b2
    a2: np.ndarray = sigmoid(z2)
    print(f"Input: {X[i]} -> Predicted Output: {a2.round()} (Raw: {
        a2})")
```

A.B: XOR Network Implemented with `tensorflow`

This ANN is based on the discussion in section 4. In this section, you'll find the code needed to implement the ANN that solves XOR using `tensorflow`, along with annotations and comments.

First, we setup our imports.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, Sequential
```

Next, we set up some parameters for the model, and we create the model.

```
# Set random seed for reproducibility
tf.random.set_seed(38)

# Set input values and expected output values
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)
y = np.array([[0], [1], [1], [0]], dtype=np.float32)

# Define input, hidden, and output layer sizes
input_size = 2
hidden_layer_size = 2
output_size = 1
```

Now, we add each layer to the model.

```
# Create the model
model = Sequential([
    layers.Input(shape=(2,)),  # Input layer (2 features for XOR)
    layers.Dense(8, activation='relu', kernel_initializer='he_normal
        '),  # Hidden layer
    layers.Dense(1, activation='sigmoid')  # Output layer (sigmoid
        for binary classification)
```

```
])
```

Finally, we compile the model, training it with the Adam gradient descent algorithm. We also display the summary once the model is complete.

```
# Compile the model
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss='binary_crossentropy',  # Loss function
    metrics=['accuracy']  # Metrics to monitor
)

# train the model
history = model.fit(
    X,        # Input data
    y,        # Target data
    batch_size=4,  # Number of samples per gradient update
    epochs=1000,    # Number of complete passes through the training
        dataset
    validation_data=(X, y)
)

# Display the model summary
model.summary()

loss, accuracy = model.evaluate(X, y)
print(f'Loss: {loss}, Accuracy: {accuracy}')
```

It's plain to see that packages like `tensorflow` make it much simpler to create an ANN.

A.C: Handwritten Digit Classification

Now, we apply the math discussed in this paper to build a handwritten digit classification ANN using `tensorflow`. This was alluded to in Section 5. The network is trained to recognize handwritten digits. Each image of a handwritten digit (grayscale) is converted into a list of values between 0 and 1—each pixel of the image (an entry) holds its color.

We begin by importing all the libraries we'll need for this network.

```
import numpy as np
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
```

Next, we load and reshape the data, and convert the labels to one-hot encoding (there will be 10 nodes on the output layer, and this tells the network that the most active (value closest to one) node is the prediction).

```
# Load MNIST dataset
```

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Preprocess the data
# Normalize the images to be between 0 and 1
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255

# Reshape data to (num_samples, 28, 28, 1) for compatibility with
   Conv2D layer (even though we are using Dense layers)
X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

# Convert labels to one-hot encoding
y_train = to_categorical(y_train, 10)  # 10 classes (digits 0-9)
y_test = to_categorical(y_test, 10)
```

Now, we define the model itself, made up of 5 layers:

- Input Layer: $\mathbb{R}^{28\times 28}$
- Second Layer (Flatten Layer): $T_1 : \mathbb{R}^{28\times 28} \to \mathbb{R}^{784}$ [This layer converts the 2D image into a 1D vector].
- Third Layer (Dense Layer): $T_2 : \mathbb{R}^{784} \to \mathbb{R}^{128}$ [This layer narrows down the number of neurons to move toward an output].
- Fourth Layer (Dropout Layer): $T_3 : \mathbb{R}^{128} \to \mathbb{R}^{128}$ [This layer sets a fraction of the inputs to zero and helps to prevent overfitting].
- Fifth Layer (Output Layer) $T_4 : \mathbb{R}^{128} \to \mathbb{R}^{10}$ [The node with the greatest value is the network's prediction (nodes labeled 0–9)].

```
# Define the model
model = models.Sequential([
    layers.Input(shape=(28, 28, 1)),
    layers.Flatten(input_shape=(28, 28, 1)),  # Flatten the image to
        a 1D vector
    layers.Dense(128, activation='relu', kernel_initializer='
       he_normal'),  # Hidden layer
    layers.Dropout(0.2),  # Dropout layer to avoid overfitting
    layers.Dense(10, activation='softmax')  # Output layer for 10
       classes
])
```

Next, we compile, train, and test the model.

```
# Compile the model
model.compile(
    optimizer='adam',               # Adam optimizer
```

```
    loss='categorical_crossentropy',  # Loss function for multi-
        class classification
    metrics=['accuracy']              # Monitor accuracy during
        training
)

# Train the model
history = model.fit(
    X_train,         # Training data
    y_train,         # Training labels
    epochs=10,       # Number of epochs
    batch_size=32,   # Batch size
    validation_split=0.2  # Split off 20% of training data for
        validation
)

# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=2)
print(f'Test accuracy: {test_acc * 100:.2f}%')

# Make predictions on the test set
predictions = model.predict(X_test)

# Show the first test image and its predicted label

plt.imshow(X_test[0].reshape(28, 28), cmap='gray')
plt.title(f"Predicted label: {np.argmax(predictions[0])}")
plt.show()
```

We have successfully created an artificial neural network that can identify handwritten digits.

References

[1] Juergen Schmidhuber. "Annotated History of Modern AI and Deep Learning". In: *ArXiv* abs/2212.11279 (2022). URL: https://api.semanticscholar.org/CorpusID:254974067.

[2] Mark Crovella. *Computer Graphics: Linear Algebra, Geometry, and Computation*. https://www.cs.bu.edu/fac/snyder/cs132-book/L13ComputerGraphics-Spring2021.html. Accessed: November 18, 2024. 2021.

[3] Yikun Han, Chunjiang Liu, and Pengfei Wang. *A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge*. 2023. arXiv: 2310.11703 [cs.DB]. URL: https://arxiv.org/abs/2310.11703.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "Deep Learning". In: http://www.deeplearningbook.org. MIT Press, 2016. Chap. 5.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "Deep Learning". In: http://www.deeplearningbook.org. MIT Press, 2016. Chap. 6.

[6] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Nov. 2011, pp. 315–323. URL: https://proceedings.mlr.press/v15/glorot11a.html.

[7] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *ArXiv* abs/1609.04747 (2016). URL: https://api.semanticscholar.org/CorpusID:17485266.

[8] Sanghyeon An et al. "An Ensemble of Simple Convolutional Neural Network Models for MNIST Digit Recognition". In: *ArXiv* abs/2008.10400 (2020). URL: https://api.semanticscholar.org/CorpusID:221266038.

[9] Ismoilov Nusrat and Sung-Bong Jang. "A Comparison of Regularization Techniques in Deep Neural Networks". In: *Symmetry* 10 (2018), p. 648. URL: https://api.semanticscholar.org/CorpusID:56482833.

[10] Justin A. Sirignano and Konstantinos V. Spiliopoulos. "Scaling Limit of Neural Networks with the Xavier Initialization and Convergence to a Global Minimum". In: *ArXiv* abs/1907.04108 (2019). URL: https://api.semanticscholar.org/CorpusID:195848363.