Neural Network Fundamentals: Insights via Linear Algebra and Object-Oriented Python

Michael Allen

October 24, 2023

Abstract

In today's data-centric era, neural networks (NNs) stand at the forefront of advancements across fields like computer vision, natural language processing, and autonomous systems. While these models excel at identifying patterns and predictions from vast datasets, their inner mechanics often elude many users. This lack of clarity can hinder their optimization and the potential for future innovations.

This paper aims to demystify the inner workings of NNs by delving deep into their foundational components such as weights, biases, and activation functions. It highlights the crucial role of backpropagation and gradient descent in honing a model's effectiveness. Using Python, known for its clarity and versatility, Object-Oriented Programming (OOP) is leveraged to provide a practical and structured approach to understanding NNs.

This discussion, paired with an accompanying Python implementation, offers a clear guide for those eager to grasp the core of neural networks. By combining theory, code, and hands-on insights, readers will bridge the divide between mathematical concepts and their applications in neural networks.

1 Motivation

In the modern digital era, neural networks stand as a linchpin in the rapidly evolving landscape of artificial intelligence and machine learning (AI/ML), driving innovations across a myriad of domains. The mastery of NN architecture is essential for optimizing performance, engendering innovative solutions, and ensuring effective troubleshooting. As organizations ardently embrace AI/ML, understanding these networks paves the way for tailored solutions, enhancing model transparency and fostering a competitive edge in the market. Moreover, the educational acumen garnered through delving into NN architecture not only propels career advancement amidst the soaring demand for AI/ML expertise but also contributes significantly to the vibrant community of AI practitioners. By unlocking the profound understanding of NNs, individuals and enterprises alike are better poised to harness the transformative power of AI, thereby navigating the wave of AI/ML mass adoption with informed dexterity and strategic prowess. In short, AI/ML has grown to be a widely adopted part of the modern tech stack for solving intricate and technically challenging problems.

2 What is a Deep Neural Network?

2.1 Definition

A neural network is a computational model inspired by the structure and functional aspects of biological networks in the brain. These models are designed to recognize patterns, mathematically referred to as functions, and make decisions based on new input data. As a simplification, supervised ML models (the focus of this paper) can be thought of as ’function-building machines’ that take in data and labeled answers and find some combination of transformations that best map data to answers. In doing so, AI practitioners can feed new information into a model and have the model predict the outcome. This is useful across a myriad of applications, as mentioned previously.

2.2 Components

A deep neural network consists of a surprisingly simple number of elements, with most of the complexity arising from how to modify these elements in a useful way.

Neuron
The core building block of a neural network is the neuron, a single unit that receives one or more inputs and applies a linear transformation which is passed into another function. This transformation is defined in Equation 1, where the output is commonly referred to as an activation.

$$y_j = f(\sum w_{ij}x_i + b_j) \tag{1}$$

x_i are the inputs, w_{ij} are the respective weights connecting input i to current neuron j, and b is the applied bias. f denotes some non-linear function applied to the result.

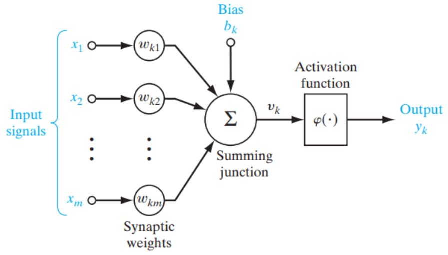

Figure 1: Visual Representation of a Neuron [1]

Layers

Deep neural networks are built in layers. A layer is an array of neurons that represent some abstract transformation of the data. Typical neural networks consist of an input layer, one or more hidden layers, and an output layer. The input layer receives the data, the hidden layers perform the neuron computations, and the output layer provides the final output. The 'deep' in 'deep learning' refers to the concept of numerous layers of a network chained together. The number of layers is often referred to as the 'depth' of a model. It is important to note that the shape of the input and output layers (how many neurons are in a layer) is dependent on the use case. Hidden layer shape selection is more arbitrary. When a previous layer's neurons are connected to all neurons in the next layer, the layer is said to be fully connected (referred to as a dense or FC layer), as depicted in the visualization.

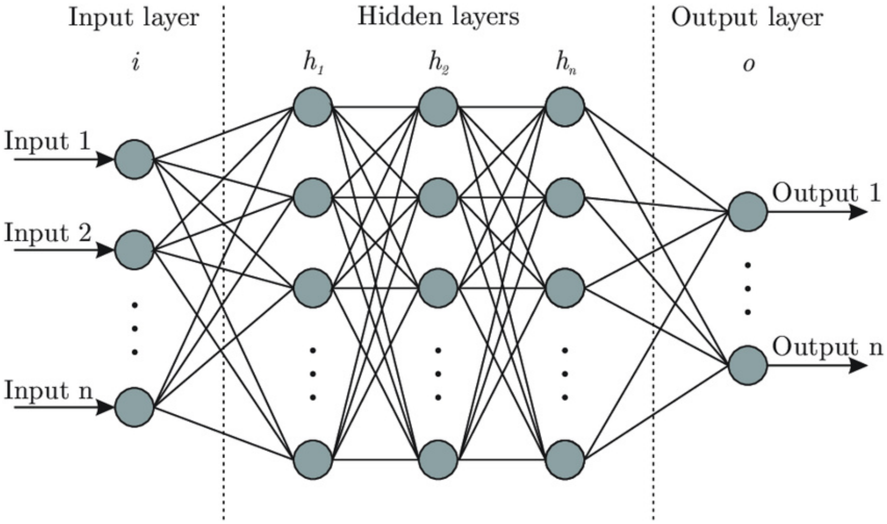

Figure 2: Visualization of Layers in a Neural Network

Weights and Biases

Dissecting the neuron equation, we can see that each connection to a neuron has an associated weight w_{ij}. The subscript i indicates which previous input element the neuron is connected to, and j represents the current neuron. Weight can be thought of as 'connective strength' where larger weights contribute more to the output of the neuron's value (more on this later). These weights are adjusted during training to minimize the error in the network's predictions. The bias is introduced to allow the activation function to be shifted to better fit the data. Together, weights and biases are considered the trainable parameters of a neural network. The number of trainable parameters impacts a network's training and prediction speed, as it takes longer to compute the resultant transformations propagating through the layers. With current hardware and optimization tooling, a practitioner can expect to feasibly run about a 7 billion parameter model on consumer-grade GPUs. This number will vary based on the use case and optimization techniques used [2]. Note that current SOTA models can have hundreds of billions of parameters, requiring an ensemble of GPUs or other hardware acceleration to serve the model. The number of parameters for a given fully connected network is as follows:

$$N = \sum_{i=1}^{L} n_i \times n_{i+1} + n_{i+1} \tag{2}$$

Where L is the number of layers, n_i is the number of neurons in the current layer, and n_{i+1} is the number of neurons in the next layer.

Computing the linear transformation for a neuron requires iterating through each previous input i, multiplying this value by the connective weight, and adding the result to a sum that is finally offset by some bias. However, there is a better way. Instead of iteration, all weights between layers can be represented in an $m \times n$ matrix, where m is the number of neurons in the current layer and n represents the input shape. Similarly, a vector $\vec{b}$ can be defined for each neuron of the current.

$$W = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1i} \\ w_{21} & w_{22} & \cdots & w_{2i} \\ \vdots & \vdots & \ddots & \vdots \\ w_{j1} & w_{j2} & \cdots & w_{ji} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix}$$

The weight matrix, each row is neuron j's connective weights.

$$\vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_j \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Bias vector

The notation of the weight matrix is important, here is an example dissecting $W_{2,1}$. w_{21} connects the 1^{st} neuron of the previous layer (input) to the 2^{nd} neuron of the current layer. Given this representation, the linear transformation for every neuron in *a single layer* can be computed using the matrix equivalent of the neuron equation.

$$\vec{y} = f(W\vec{x} + \vec{b}) = f\left(\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \right) \tag{3}$$

Where $\vec{y}$ is the output vector of size m, $\vec{x}$ is the input vector of size n, W is the weight matrix preceding the output layer, and $\vec{b}$ is the applied bias to the output. f is a vectorized version of some nonlinear function (for each element in the argument, the function is applied).

Note: The choice for notation is arbitrary, but is important when deriving further steps. The approach shown is using columns to represent the input vector x and output y. However, choosing to represent the inputs and outputs as rows is equally valid, and the respective approach is listed in Appendix B. The row approach has some nice benefits, like using matrix multiplication to simplify the derivations and differing indexing of the weight matrix. However, column representation was chosen as it is the most common method. Those interested in seeing how the row approach can be advantageous or enlightening are strongly encouraged to see the derivations for the alternate method.

Activation Function

After an intermediate value v_j has been calculated by the neuron, an activation function is applied to the value to get y_j. This activation function serves to perform some nonlinear transformation to the value. While there are many, the most common ones are *ReLU, Sigmoid, Tanh,* and *Softmax.* Without an activation function, the network remains linear, no matter how many layers are added. In fact, if only linear transformations are successively applied, then the entire network could be simplified to a single linear transformation of the input data as proven in Appendix A.

$$f(x) = max(0, x) \tag{4}$$

ReLU Activation Function

$$f(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

Sigmoid Activation Function

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{6}$$

Tanh Activation Function

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{7}$$

Softmax Activation Function

2.3 Forward Propagation

Forward propagation is a simple but fundamental concept in neural networks, referring to the process of passing the input data through the network, layer by layer, until the output layer is reached. This process represents the model making predictions, based on input data. Equation 3 encapsulates the core computation performed by a layer of neurons in the network during this process.

1. **Input Vector** ($\vec{x}$):
   - Forward propagation begins with the input vector $\vec{x}$, which represents the data being fed into the network. This vector is passed to the first layer of the network.
2. **Linear Transformation**:
   - Each layer of neurons performs a linear transformation on its input. The transformation is dictated by the layer's weight matrix W and bias vector $\vec{b}$. Specifically, the dot product of W and $\vec{x}$ is computed, and then $\vec{b}$ is added to the result. Mathematically, this operation is denoted as $W \cdot \vec{x} + \vec{b}$.
3. **Non-linear Activation**:
   - Following the linear transformation, a non-linear activation function f is applied element-wise to the result. This introduces non-linearity to the system, enabling the network to learn complex relationships within the data. The vectorized version of the activation function f ensures that the function is applied to each element of the resulting vector from the linear transformation.
4. **Output Vector** ($\vec{Y}$):
   - The output vector $\vec{Y}$ of the layer is obtained as the result of the activation function. This vector serves as the input to the next layer in the network, symbolizing the culmination of the computations performed by the current layer.
5. **Propagation to Subsequent Layers**:
   - The output vector $\vec{Y}$ of one layer is used as the input vector $\vec{x}$ to the subsequent layer, and the process of linear transformation followed by non-linear activation is repeated. This continues until the output layer is reached.
6. **Final Output**:
   - The final output of the network is obtained after the last layer has performed its computations. This output can be used for various purposes such as predictions in a supervised learning task, or further analysis in unsupervised learning tasks.

2.4 Inference

After the forward propagation process, the terminal layer of neurons executes its designated transformations, resulting in the output vector ($\vec{y}$). The values in this vector can be interpreted as the prediction. The constituents of this vector embody the network's predictions, the interpretation of which is contingent on the specific domain or task at hand. For instance, in a classification task, each element of $\vec{y}$ could represent the predicted probability of a corresponding class.

3 How do Neural Networks Learn?

3.1 Supervised Learning

The focal point of this investigation is to delve into the behavior and fundamental principles of supervised learning models. Analogous to human learning, supervised neural networks augment their knowledge through experiential learning. The capability of individuals to identify entities such as automobiles in the real world is fostered by the recurrent observation of distinctive patterns associated with cars — the wheels, silhouette, and contextual environment. This iterative pattern recognition fortifies our cognitive association with the concept of a car. Neural networks fundamentally do the same thing, observing many cases of different, known instances of cars and non-cars. These known instances are referred to as training data.

3.2 Loss Function

Recall that neural networks can be conceptualized as function-building machines. This notion naturally segues into the introduction of a loss function, a pivotal mechanism that assesses the adequacy of the function crafted by the model. The loss function acts as a compass for the training process, quantifying how well the model's predictions align with the true labels. It operates by comparing the output vector, obtained after the forward propagation process, to an objectively correct datum vector (the true labels). The magnitude of the differences between these vectors is evaluated and scored based on the particular loss function chosen. This quantified disparity is often referred to as the 'cost' of a given training example, hence the loss function is also commonly termed as a 'cost function.'

$$L(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2 \tag{8}$$

Mean Squared Error (MSE) Loss Function

$$L(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y_i}| \tag{9}$$

Mean Absolute Error (MAE) Loss Function

$$L(y, \hat{y}) = -\sum_{i=1}^{N} y_i \log(\hat{y_i}) \tag{10}$$

Cross-Entropy Loss Function

$$L(y, \hat{y}) = -\sum_{i=1}^{N}(y_i \log(\hat{y_i}) + (1 - y_i)\log(1 - \hat{y_i})) \tag{11}$$

Binary Cross-Entropy Loss Function

$$L(y, \hat{y}) = -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic} \log(\hat{y_{ic}}) \tag{12}$$

Categorical Cross-Entropy Loss Function

$$L(y, \hat{y}) = \sum_{i=1}^{N} \max(0, 1 - y_i \cdot \hat{y_i}) \tag{13}$$

Hinge Loss Function

3.3 Backward Propagation

Given the quality of the output from a model can be measured using loss functions, there is a quantifiable metric for correctness that can be optimized. Backward propagation is the mechanism by which the error signal is propagated back through the network to update the trainable model parameters (weights and biases) in a way that minimizes the loss. The application of this process is the most important part of deep neural networks, and also the most complex as alluded to earlier. For this reason, the process will be meticulously broken down to provide both a technical definition of what is happening and more importantly an intuition for the process.

Objective

In order to adjust the trainable parameters as efficiently as possible, a process called gradient descent is applied. [3]

> For those unfamiliar, the gradient of a multi-variable function is effectively the function's 'slope' and is notated using ∇. Taking the gradient at a specified point space results in a vector that points in the steepest slope upwards. So naturally, $-\nabla(p)$ is a vector pointing to the local minima given the input point p. To understand gradient descent, consider the analogy of being in a mountainous terrain blindfolded, with the goal of reaching the lowest valley. At each step, you would feel the ground around you and take a step in the direction where the ground is descending most steeply.

As the name suggests, neural networks employ this optimization technique to "descend" the gradient of a cost or loss function. The gradient of this loss function indicates the direction and rate of change of the function with respect to its parameters. In sum, gradient descent adjusts the parameters of the neural network in the direction that most rapidly reduces the loss. Gradient descent can be implemented using the following partial derivatives [4]:

$\frac{\partial L}{\partial y}$	$\frac{\partial L}{\partial W}$	$\frac{\partial L}{\partial b}$	$\frac{\partial L}{\partial x}$
Change in loss function with respect to output y (known).	Change in loss function with respect to weights (unknown).	Change in loss function with respect to bias (unknown).	Change in loss function with respect to input x (unknown).

By solving for these components, backpropagation can effectively be implemented thus enabling the model to learn. Note that $\frac{\partial L}{\partial \vec{x}}$ is not inherently a part of the gradient descent process, but it will be used for calculating $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$ for previous layers. Recall the matrix generalization of the linear transformation in the neuron equation $\vec{y} = W\vec{x} + \vec{b}$ which can be written out in matrix form as:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{14}$$

Validating $\frac{\partial L}{\partial y}$ as Known

Consider the final layer of a neural network with output y derived from the forward propagation process. Note that the expected output $\hat{y}$ is also known for correctly labeled training data. Given these values are known, the loss function can be evaluated. For the sake of example, *Mean Squared Error (MSE)* will be used: $L(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$. In this example, $L(y, \hat{y})$ is differentiable with respect to the model output layer $\hat{y}$ such that:

$$\frac{\partial L}{\partial y} = 2(y - \hat{y}) \tag{15}$$

Derivative of *MSE Loss* with respect to layer y output.

The batch generalization for this equation is $\frac{\partial L}{\partial Y} = 2(Y - \hat{Y})$. Given this equation is defined, and $\frac{\partial L}{\partial x}$ can be solved for, then the previous layer's $\frac{\partial L}{\partial y}$ can also be solved for. Through induction, this process is repeatable for all layers in a network until the input layer is reached.

Note: In practice, non-differentiable functions may be used to calculate the loss of a model. As such, approximation techniques can be used like Subgradient Descent, Discretization, or Smoothing Techniques. While these are beyond the scope of this paper, it is important to understand that more complex processes may be required to compute or approximate $\frac{\partial L}{\partial y}$.

Solving for $\frac{\partial L}{\partial W}$

1. $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial W}$ (Chain Rule)

2. $\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \vdots \\ \frac{\partial L}{\partial y_m} \end{bmatrix} \frac{\partial y}{\partial W}$ (Expansion of $\frac{\partial L}{\partial y}$)

3. Note that at this point, multiplying by $\frac{\partial y}{\partial W}$ is confusing without context. In order to paint a full picture, consider the following examples of solving for the change in loss with respect to only one weight at a time:

Part 1 (w_{11}):

$\frac{\partial L}{\partial w_{11}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \vdots \\ \frac{\partial L}{\partial y_m} \end{bmatrix} \frac{\partial y}{\partial w_{11}}$ (Expressing the partial change in L for only a single weight w_{11})

$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial w_{11}} + \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial w_{11}} + ... + \frac{\partial L}{\partial y_m}\frac{\partial y_m}{\partial w_{11}}$ (Expansion, considering $\frac{\partial}{\partial w_{11}}$ for all y)

$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial y_1}x_1 + \cancelto{0}{\frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial w_{11}}} + ... + \cancelto{0}{\frac{\partial L}{\partial y_m}\frac{\partial y_m}{\partial w_{11}}}$ (Evaluating the expression)

$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial y_1}x_1$ (Simplifying). ✓

Part 2 (w_{12}):

$\frac{\partial L}{\partial w_{12}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \vdots \\ \frac{\partial L}{\partial y_m} \end{bmatrix} \frac{\partial y}{\partial w_{12}}$ (Expressing the partial change in L for only a single weight w_{12})

$\frac{\partial L}{\partial w_{12}} = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial w_{12}} + \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial w_{12}} + ... + \frac{\partial L}{\partial y_m}\frac{\partial y_m}{\partial w_{12}}$ (Expansion, considering $\frac{\partial}{\partial w_{12}}$ for all y)

$\frac{\partial L}{\partial w_{12}} = \frac{\partial L}{\partial y_1}x_2 + \cancelto{0}{\frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial w_{12}}} + ... + \cancelto{0}{\frac{\partial L}{\partial y_m}\frac{\partial y_m}{\partial w_{12}}}$ (Evaluating the expression)

$\frac{\partial L}{\partial w_{12}} = \frac{\partial L}{\partial y_1}x_2$ (Simplifying). ✓

Part 3 (w_{21}):

$\frac{\partial L}{\partial w_{21}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \vdots \\ \frac{\partial L}{\partial y_m} \end{bmatrix} \frac{\partial y}{\partial w_{21}}$ (Expressing the partial change in L for only a single weight w_{21})

$\frac{\partial L}{\partial w_{21}} = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial w_{21}} + \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial w_{21}} + ... + \frac{\partial L}{\partial y_m}\frac{\partial y_m}{\partial w_{21}}$ (Expansion, considering $\frac{\partial}{\partial w_{21}}$ for all y)

$\frac{\partial L}{\partial w_{11}} = \cancelto{0}{\frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial w_2 1}} + \frac{\partial L}{\partial y_2}x_1 + ... + \cancelto{0}{\frac{\partial L}{\partial y_m}\frac{\partial y_m}{\partial w_{21}}}$ (Evaluating the expression)

$\frac{\partial L}{\partial w_{21}} = \frac{\partial L}{\partial y_2}x_1$ (Simplifying). ✓

Notice the general pattern, that the ∂L for some weight w can be represented as $\frac{\partial L}{\partial y_i}x_j$ for any given row i and column j. This can be used to express the derived matrix as the product of $\frac{\partial L}{\partial y}$ and x^T:

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial y_1}x_1 & \frac{\partial L}{\partial y_1}x_2 & \cdots & \frac{\partial L}{\partial y_1}x_n \\ \frac{\partial L}{\partial y_2}x_1 & \frac{\partial L}{\partial y_2}x_2 & \cdots & \frac{\partial L}{\partial y_2}x_n \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial y_m}x_1 & \frac{\partial L}{\partial y_m}x_2 & \cdots & \frac{\partial L}{\partial y_m}x_n \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \vdots \\ \frac{\partial L}{\partial y_m} \end{bmatrix} \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix} \equiv \frac{\partial L}{\partial y} \cdot x^T \tag{16}$$

The batch implementation of this equation is $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}X^T$ for some matrix of numerous training inputs X. Notice that the effective result of this is the sum of the change in L with respect to w for each training example in X.

Solving for $\frac{\partial L}{\partial b}$

1. $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial b}$ (Chain Rule)
2. $\frac{\partial L}{\partial b} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \vdots \\ \frac{\partial L}{\partial y_m} \end{bmatrix} \frac{\partial y}{\partial b}$ (Expansion of $\frac{\partial L}{\partial y}$)
3. $\frac{\partial L}{\partial b} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \vdots \\ \frac{\partial L}{\partial y_m} \end{bmatrix} \cancelto{I}{\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}}$ (Evaluation of $\frac{\partial y}{\partial b}$)
4. $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}$ (Simplification)

The batch implementation of this equation when expanded to $\mathbb{R}^{B\times n}$ is $\frac{\partial L}{\partial Y}$ which requires the summation of losses from each batch element which can be expressed as:

$$\frac{\partial L}{\partial b} = \begin{bmatrix} \sum_B \frac{\partial L}{\partial y_1} \\ \vdots \\ \sum_B \frac{\partial L}{\partial y_m} \end{bmatrix} = \frac{\partial L}{\partial Y} \tag{17}$$

Solving for $\frac{\partial L}{\partial x}$

1. $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$ (Chain Rule)

Similar to proving $\frac{\partial L}{\partial W}$, $\frac{\partial L}{\partial x}$ can be approached first with a case and then a generalization.

2. $\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial y_1}w_{11} + \frac{\partial L}{\partial y_2}w_{21} + ... + \frac{\partial L}{\partial y_m}w_{m1}$ (Evaluating ∂L for x_1)

3. $\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial y_1}w_{12} + \frac{\partial L}{\partial y_2}w_{22} + ... + \frac{\partial L}{\partial y_m}w_{m2}$ (Evaluating ∂L for x_2)

$$\frac{\partial L}{\partial x} = W^T \cdot \frac{\partial L}{\partial y} \tag{18}$$

Generalizing for all x of $\frac{\partial L}{\partial x}$

Updating Trainable Parameters

Given the partial derivatives of the weights and biases between input layer x and layer y have been found, they can now be updated in the most efficient direction to reduce the loss. This update can be represented using the following expressions:

$$w = w - \alpha \frac{\partial L}{\partial w} \tag{19}$$

Weight update rule

$$b = b - \alpha \frac{\partial L}{\partial b} \tag{20}$$

Bias update rule

Notice the inclusion of the term α, referred to as the learning rate. This coefficient determines how large of a step to take in the direction of the gradient during each iteration of optimization. While a frequently used starting value for the learning rate is 0.001, it can vary significantly, often spanning two orders of magnitude or more in either direction. To strike a balance between fast convergence and model stability, many practitioners adopt learning rate schedules or adaptive techniques. These approaches adjust the learning rate during training, starting with a relatively larger rate for faster initial convergence and then reducing it to ensure stability and prevent oscillations around the minima.

4 Building a Simple Neural Network in Python

The inspiration for this project is from Omar Aflak's project on building networks from scratch using Python [5]. Most of the code included is a modified version of their original work here: [6]. The completed code for this project is published to GitHub.

Having explored the foundational concepts and mathematical formulations of neural networks, the subsequent section offers a practical application of these principles. The previously discussed computations, while complex, are integral to the functionality of neural networks. With a comprehensive understanding of these calculations established, the next logical step is to implement a neural network from scratch. The ensuing pages provide an in-depth look into the coding and implementation of these foundational equations.

In the subsequent implementation, an Object-Oriented Programming (OOP) approach will be employed using Python. This methodology allows for the creation of distinct classes, each representing different components of the neural network. By instantiating these classes, individual layers of the network can be formed. Notably, the network itself will be represented as a list of these instantiated layers, facilitating modularity and ease of expansion. A key design decision in this implementation is the separation of the activation function from the linear transformation layers. Treating the activation function as an independent layer simplifies the backpropagation process, ensuring clarity in the flow of computations and enhancing the efficiency of gradient calculations.

As a final note, this neural network will attempt to define weights and biases that most accurately represent the *AND* function. While a trivial implementation of neural networks (given there is already a model that determines the output of *AND* given two input features), the goal is to show how these networks learn underlying trends in data and build approximations based on the defined loss functions.

4.1 Abstract Layer

Building an abstract framework for all layers to follow will simplify the structure, enabling overriding classes to not have to declare redundancies. As discussed, each layer has an input that returns some output. Furthermore, the direction of this transformation will be referred to as 'forward' and 'backward' depending on the direction of propagation through the network.

```
class Layer:
  def __init__(self) -> None:
      self.input = None
```

```
        self.output = None

    def forward(self, input):
        pass

    def backward(self, gradient, learning_rate):
        pass
```

4.2 Dense (FC) Layer

The dense layer inherits the input tensor from the layer class, as well as output shape m and input shape n. From this, the $m \times n$ weights matrix and bias can be initialized with random values. Propagation steps are the simple implementations of the process described in the prior sections.

```
class Dense(Layer):
    def __init__(self, x_size, y_size) -> None:
        self.weights = np.random.randn(y_size, x_size)
        self.bias = np.random.randn(y_size, 1)

    def forward(self, input):
        self.input = input
        return  np.dot(self.weights, self.input) + self.bias

    def backward(self, y_gradient, learning_rate):
        weight_gradient = np.dot(y_gradient, self.input.T)
        self.weights -= learning_rate * weight_gradient #update weights
        self.bias -= learning_rate * y_gradient
        return np.dot(self.weights.T, y_gradient) #Return dL/dx for next layer
```

4.3 Activation Layer

Provided the activation layer is considered a separate layer, the definition of $\frac{\partial L}{\partial x}$ takes on a different meaning. The math for the activation function's backpropagation can be written as:

$$(\frac{\partial L}{\partial x})_a = \frac{\partial L}{\partial y} \odot \frac{\partial y}{\partial x} \equiv \frac{\partial L}{\partial y} \odot f'(x) \tag{21}$$

This equation is simply implemented in the backpropagation step provided the derivative of the activation function is defined and passed. The forward propagation step is simply the application of the nonlinear function specified.

```
class Activation(Layer):
    def __init__(self, activation, activation_prime):
        self.activation = activation
        self.activation_prime = activation_prime

    def forward(self, input):
        self.input = input
        return self.activation(self.input)

    def backward(self, y_gradient):
        return y_gradient * self.activation_prime(self.input)
```

This example will use the hyperbolic-tangent activation defined as $tanh(x)$. Note that any activation function can be chosen, but it is preferable to have one that is differentiable for simplicity.

```
from activation import Activation
class tanh(Activation):
def __init__(self):
    def tanh(x):
        return np.tanh(x)

    def tanh_prime(x):
        return 1 - np.tanh(x) ** 2

    super().__init__(tanh, tanh_prime)
```

4.4 Loss Function

The loss function used in this example will be *MSE*. See the code below for the definition of the functions in Python:

```
def mse(y, y_hat):
     return np.mean(np.power(y - y_hat, 2))

def mse_prime(y, y_hat):
     return 2 * (y - y_hat) / np.size(y)
```

4.5 Network Object

With the layers and components now defined, we can proceed to instantiate the network object. This network aims to approximate the *AND* logic gate, as outlined in the truth table below:

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: Truth table for the *AND* function

The corresponding Python representation of the inputs (X) and outputs (Y) for the *AND* function, as well as the definition of the network object, are presented below.

```
#'and' inputs (X) and outputs (Y)
X = np.reshape([[0, 0], [0, 1], [1, 0], [1, 1]], (4, 2, 1))
Y = np.reshape([[0], [0], [0], [1]], (4, 1, 1))

#define the network object
network = [
    Dense(2, 3),
    tanh(),
    Dense(3, 1),
    tanh()
]
```

4.6 Training the Network

To enable the neural network to approximate the AND function, it is imperative to iteratively adjust the model's weights based on its predictions and the corresponding true outputs. This process is denoted as training the network.

First, a function, `predict`, is defined. This function processes the input through each layer of the network to yield the output. This output, or prediction, signifies the network's inferred answer based on its current state, characterized by the present values of its weights and biases.

Subsequently, the training function, `train`, is delineated. This function updates the weights and biases of the network over multiple iterations, termed epochs. During each epoch, the network formulates a prediction for every input in the training dataset. The discrepancy between this prediction and the true output, as quantified by a specified loss function, guides the modifications to the network's weights and biases. The following steps elucidate the training process:

- A forward pass is computed using the `predict` function for each input.
- The error (loss) for each prediction is ascertained utilizing the provided loss function.
- Leveraging the derivative (gradient) of the loss, a backward pass is executed to amend the weights in the network. This is accomplished in reverse sequence, commencing from the final layer to the initial one.
- Upon processing all inputs, the mean error for the epoch is determined and, if `verbose` is set to `True`, it is printed to monitor the training progression.

```
def predict(network, input):
    output = input
    for layer in network:
        output = layer.forward(output)
    return output

def train(network, loss, loss_prime, x_train, y_train,
          epochs=1000, learning_rate=0.001, verbose=True):

    for e in range(epochs):
        error = 0
        for x, y in zip(x_train, y_train):
            output = predict(network, x)
            error += loss(y, output)
            grad = -loss_prime(y, output)
            for layer in reversed(network):
                grad = layer.backward(grad, learning_rate)

        error /= len(x_train)
        if verbose:
            print(f"{e + 1}/{epochs}, error={error}")
```

4.7 Visualizing Results

Visualizing the results of the trained model is important for several reasons. Primarily, these visualizations provide an intuitive and comprehensive perspective on the model's behavior over the input space. By plotting the model's predictions across a grid of input values, it becomes feasible to discern patterns, trends, and potential anomalies in its responses.

The ”3D Scatter Plot” showcases how the model's output varies over the two-dimensional input space. The ”Contour Plot” offers a 2D perspective, using color intensity to signify predicted outputs. This plot effectively highlights the decision boundaries, helping to quickly evaluate whether the model has captured the essence of the AND operation.

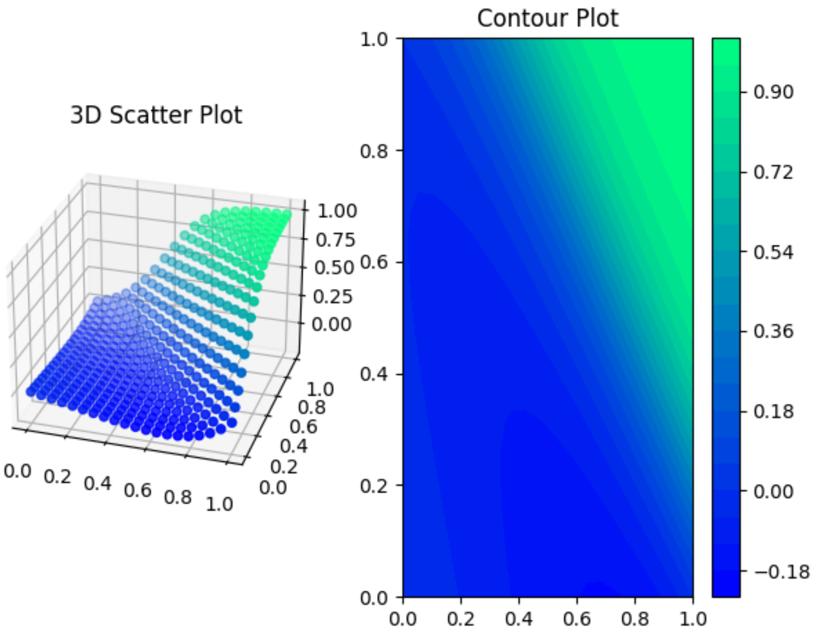

Figure 3: Neural Network prediction of *AND* function given varying input features. Created with the following code.

```
# decision boundary plot
points = []
z_matrix = np.zeros((20, 20))
for i, x in enumerate(np.linspace(0, 1, 20)):
    for j, y in enumerate(np.linspace(0, 1, 20)):
        z = predict(network, [[x], [y]])
        points.append([x, y, z[0, 0]])
        z_matrix[i, j] = z[0, 0]

points = np.array(points)

# 3D scatter plot
fig = plt.figure()
ax = fig.add_subplot(121, projection=”3d”)
ax.scatter(points[:, 0], points[:, 1], points[:, 2], c=points[:, 2], cmap=”winter”)
ax.set_title(”3D Scatter Plot”)

# 2D contour plot
ax2 = fig.add_subplot(122)
X_grid, Y_grid = np.meshgrid(np.linspace(0, 1, 20), np.linspace(0, 1, 20))
contour = ax2.contourf(X_grid, Y_grid, z_matrix.T, levels=20, cmap=”winter”)
ax2.set_title(”Contour Plot”)
plt.colorbar(contour)

plt.tight_layout()
plt.show()
```

5 Conclusion

As the digital age propels us into an era of data-driven science and decision-making, neural networks stand at the forefront of this revolution, driving breakthroughs in diverse sectors from computer vision to bioinformatics. The intricate design of these networks, mirroring the human brain's complex synaptic connections, has demonstrated unparalleled capabilities in pattern recognition and predictions. However, their enigmatic internal operations have often left both seasoned professionals and newcomers in the field seeking clarity. This paper embarked on a journey to explore NNs, employing linear algebra, Python, and object-oriented programming as a framework. As we navigate the vast expanse of artificial intelligence and machine learning, it becomes imperative to understand, adapt, and optimize these neural architectures. Only by peeling back the layers of complexity and fostering a deeper comprehension can we truly harness the potential of NNs and pave the way for further innovations and advancements in the realm of computational intelligence.

This exploration dives into one of many powerful tools at the modern practitioner's disposal, with a toy example to best show what a neural network is doing at its core. There are many augmentations to the traditional neural network not covered in this paper. Furthermore, a plethora of models are available that may be better suited to a given task. A key challenge for data scientists is to determine what approach best suits a given use case. While neural networks are a valuable tool, avoid the naive approach that they are a catch-all solution to any data challenge - if a problem can be solved without a NN then don't use one. Nonetheless, by intimately understanding these models practitioners are better suited to develop impactful solutions in the digital age.

References

[1] Simon S. Haykin. Neural networks and learning machines. *Pearson Education, Upper Saddle River, NJ*, 2009.

[2] HuggingFace. Methods and tools for efficient training on a single gpu. *HuggingFace*, 2023.

[3] Grant Sanderson and Josh Pullen. What is backpropagation really doing?, 2017. Updated on Oct 18, 2023.

[4] alex ai7517. Deriving matrix equations for backpropagation on a linear layer, 2022.

[5] Omar Aflak. Neural network from scratch in python, 2018.

[6] Omar Aflak. Neural-network, 2021. git repository of Omar Aflak's work.

A Proving Necessity of Activation Functions

Proof. Suppose there is an input vector x, with k successive layers. Each of these layers applies a linear transformation with weight matrix W_k and bias vector b_k. The first layer will apply a transformation such that:

$$z_1 = W_1 \cdot x + b_1$$

Now the intermediate output z_1 is fed as input to the second layer, which applies another linear transformation:

$$z_2 = W_2 \cdot z_1 + b_2$$

We know that we can represent z_1 as the transformation of the previous layer, so we can substitute and manipulate as follows:

$$\begin{aligned} z_2 &= W_2 \cdot (W_1 \cdot x + b_1) + b_2 && \text{Substitution} \\ z_2 &= W_2 \cdot W_1 \cdot x + W_2 \cdot b_1 + b_2 && \text{Distribution of } W_2 \end{aligned}$$

This new weight matrix $W_2 \cdot W_1$ can be generalized as W, and the bias term $W_2 \cdot b_1 + b_2$ can be generalized as b for a final expression of:

$$z_2 = W \cdot x + b$$

This remains true for any z_k, meaning any sequence of linear transformations across layers can be represented as a single linear transformation. Following the same pattern of substitutions and simplifications, the output of the k-th layer can be expressed as:

$$z_k = W \cdot x + b$$

where W is the product of all weight matrices from layer 1 to layer k, and b is the sum of the transformed bias vectors from layer 1 to layer k. Therefore, without a non-linear activation function between these layers, the entire network can be reduced to a single linear transformation. □

B Row Approach and Proofs for Input Vectors

Consider that the neuron equation is written as $y_j = f(b_j + \sum x_i w_{ij})$. The matrix equivalent can be written using row vectors:

$$W = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1j} \\ w_{21} & w_{22} & \cdots & w_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & \cdots & w_{ij} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} b_1 & b_2 & \cdots & b_i \end{bmatrix}$$

The weight matrix, each column is neuron j's connective weights.

Bias vector

The notation of the weight matrix is important, here is an example dissecting $W_{2,1}$. w_{21} connects the 2^{nd} neuron of layer $(n-1)$ to the 1^{st} neuron of layer (n). Given this representation, the linear transformation for every neuron in *a single layer* can be computed using the matrix equivalent of the neuron equation.

$$\vec{y} = f(\vec{x}W + \vec{b}) \tag{22}$$

$$\begin{bmatrix} y_1 \cdots y_n \end{bmatrix} = \begin{bmatrix} x_1 \cdots x_m \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix} + \begin{bmatrix} b_1 \cdots b_n \end{bmatrix} \tag{23}$$

Solving for $\frac{\partial L}{\partial W}$

1. $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial W}$ (Chain Rule)
2. $\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n} \end{bmatrix} \frac{\partial y}{\partial W}$ (Expansion of $\frac{\partial L}{\partial y}$)

3. Note that at this point, multiplying by $\frac{\partial y}{\partial W}$ is confusing without context. In order to paint a full picture, consider the following examples of solving for the change in loss with respect to only one weight at a time:

Part 1 (w_{11}):

$\frac{\partial L}{\partial w_{11}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n} \end{bmatrix} \frac{\partial y}{\partial w_{11}}$ (Expressing the partial change in L for only a single weight w_{11})

$\frac{\partial y}{\partial w_{11}} = \begin{bmatrix} \frac{\partial y_1}{\partial w_{11}} \\ \vdots \\ \frac{\partial y_n}{\partial w_{11}} \end{bmatrix} \implies \frac{\partial L}{\partial w_{11}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n} \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial w_{11}} \\ \vdots \\ \frac{\partial y_n}{\partial w_{11}} \end{bmatrix}$ (Expand to $\frac{\partial y}{\partial w_{11}}$ column form)

$\frac{\partial L}{\partial w_{11}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ 0 \end{bmatrix}$ (Evaluating $\frac{\partial y}{\partial w_{11}}$ results in only x_1 in the first entry)

$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial y_1} x_1$ (Evaluating the expression). ✓

Part 2 (w_{12}):

$\frac{\partial L}{\partial w_{12}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n} \end{bmatrix} \frac{\partial y}{\partial w_{12}}$ (Expressing the partial change in L for only w_{12})

$\frac{\partial y}{\partial w_{12}} = \begin{bmatrix} \frac{\partial y_1}{\partial w_{12}} \\ \vdots \\ \frac{\partial y_n}{\partial w_{12}} \end{bmatrix} \implies \frac{\partial L}{\partial w_{12}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n} \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial w_{12}} \\ \vdots \\ \frac{\partial y_n}{\partial w_{12}} \end{bmatrix}$ (Expand to $\frac{\partial y}{\partial w_{12}}$ column form)

$\frac{\partial L}{\partial w_{12}} = \left[\frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n}\right] \begin{bmatrix} 0 \\ x_1 \\ \vdots \\ 0 \end{bmatrix}$ (Evaluating $\frac{\partial y}{\partial w_{12}}$ results in only x_1 in the first entry)

$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial y_2} x_1$ (Evaluating the expression). ✓

Part 3 (w_{21}):
$\frac{\partial L}{\partial w_{21}} = \left[\frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n}\right] \frac{\partial y}{\partial w_{21}}$ (Expressing the partial change in L for only w_{21})
$\frac{\partial y}{\partial w_{21}} = \begin{bmatrix} \frac{\partial y_1}{\partial w_{21}} \\ \vdots \\ \frac{\partial y_n}{\partial w_{21}} \end{bmatrix} \implies \frac{\partial L}{\partial w_{21}} = \left[\frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n}\right] \begin{bmatrix} \frac{\partial y_1}{\partial w_{21}} \\ \vdots \\ \frac{\partial y_n}{\partial w_{21}} \end{bmatrix}$ (Expand to $\frac{\partial y}{\partial w_{21}}$ column form)
$\frac{\partial L}{\partial w_{21}} = \left[\frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n}\right] \begin{bmatrix} x_2 \\ \vdots \\ 0 \end{bmatrix}$ (Evaluating $\frac{\partial y}{\partial w_{12}}$ results in only x_1 in the first entry)
$\frac{\partial L}{\partial w_{21}} = \frac{\partial L}{\partial y_1} x_2$ (Evaluating the expression). ✓

Notice the general pattern, that the ∂L for some weight w can be represented as $\frac{\partial L}{\partial y_j} x_i$ for any given row i and column j. This can be used to express the derived matrix as the product of x^T and the $\frac{\partial L}{\partial y}$ vector:

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial y_1} x_1 & \frac{\partial L}{\partial y_2} x_1 & \cdots & \frac{\partial L}{\partial y_n} x_1 \\ \frac{\partial L}{\partial y_1} x_2 & \frac{\partial L}{\partial y_2} x_2 & \cdots & \frac{\partial L}{\partial y_n} x_2 \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial y_1} x_m & \frac{\partial L}{\partial y_2} x_m & \cdots & \frac{\partial L}{\partial y_n} x_m \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \cdot \left[\frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n}\right] \equiv x^T \frac{\partial L}{\partial y} \tag{24}$$

The batch implementation of this equation is $\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Y}$ for some matrix of numerous training inputs X. Notice that the effective result of this is the sum of the change in L with respect to w for each training example as a column in X.

Solving for $\frac{\partial L}{\partial b}$
1. $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial b}$ (Chain Rule)
2. $\frac{\partial L}{\partial b} = \left[\frac{\partial L}{\partial y_1} \cdots \frac{\partial L}{\partial y_n}\right] \frac{\partial y}{\partial b}$ (Expansion of $\frac{\partial L}{\partial y}$)
3. $\frac{\partial L}{\partial b} = \left[\frac{\partial L}{\partial y_1} \quad \cdots \quad \frac{\partial L}{\partial y_n}\right] \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}^{\nearrow I}$ (Evaluation of $\frac{\partial y}{\partial b}$)
4. $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}$ (Simplification)

The batch implementation of this equation when expanded to $\mathbb{R}^{B \times n}$ is $\frac{\partial L}{\partial Y}$ which requires the summation of losses from each batch element which can be expressed as $\frac{\partial L}{\partial b} = \left[\sum_B \frac{\partial L}{\partial y_1} \sum_B \frac{\partial L}{\partial y_2} \cdots \sum_B \frac{\partial L}{\partial y_n}\right]$.

Solving for $\frac{\partial L}{\partial x}$
1. $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial x}$ (Chain Rule)

2. $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$ (Expanding $\frac{\partial y}{\partial x}$)

3. $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \begin{bmatrix} w_{11} & w_{21} & \cdots & w_{m1} \\ w_{12} & w_{22} & \cdots & w_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1n} & w_{2n} & \cdots & w_{mn} \end{bmatrix} = \frac{\partial L}{\partial y} W^T$ (Evaluating $\frac{\partial y}{\partial x}$ matrix)

Generalizing to batches results in the same equation: $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W^T$

Note: Notice how the generalized solutions take on a different shape, therefore resulting in a fundamentally different way of performing matrix math and calculations during layer propagation. For this reason, it is imperative to decide on an approach and stick with the convention in terms of data shape and transformations during development.