Please note this is a work in progress.

Mathematical Analysis of Multilayer Perceptron Training
Independent / Self Study

Robert Davidson

BSc Mathematics and Computer Science
r.davidson1@universityofgalway.ie

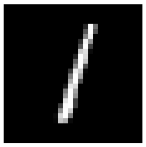
Date: March 24, 2025

Abstract

This paper provides a comprehensive mathematical derivation and implementation of a Multilayer Perceptron (MLP) for the classification of handwritten digits from the MNIST dataset, starting from fundamental principles. We begin by exploring the biological inspiration behind artificial neurons, establishing a clear mapping between biological and artificial neural components. We then develop the mathematical representation of a single artificial neuron, progressing to the construction of perceptron layers and the full MLP architecture. The necessity of hidden layers for non-linear separability is discussed, alongside the selection and application of appropriate activation functions. We detail the forward propagation process using matrix-vector notation, followed by an in-depth explanation of cost functions and the error backpropagation algorithm for gradient computation. Optimization and training techniques, including batch-based methods and strategies for improving convergence, are examined. We address the critical issue of regularization to mitigate overfitting and underfitting, including the influence of initialization strategies. Finally, we present the implementation of a fully functional MLP for MNIST classification, concluding with a discussion of extensions to deeper architectures and modern practices in neural network development. This work aims to provide a clear, step-by-step mathematical foundation for understanding and building MLPs from the ground up.

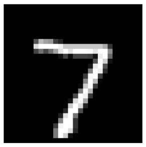
Introduction

The MNIST (Modified National Institute of Standards and Technology) dataset consists of 70,000 handwritten digit images (60,000 for training and 10,000 for testing). It is commonly used for training and evaluating machine learning models in image recognition tasks. Each image is a 28x28 pixel grayscale image, and the task is to classify the digit (0-9) represented by the image.



Sample MNIST digit 1

Sample MNIST digit 2



Sample MNIST digit 7

A Multilayer Perceptron (MLP) is a type of artificial neural network composed of multiple layers of nodes (neurons), each fully connected to the next layer. The MLP processes input data through a series of transformations, with the final layer producing an output. It is one of the fundamental types of neural networks and is used in a variety of applications, including image recognition.

In this paper, we aim to build the mathematical foundations necessary to understand and implement a Multilayer Perceptron (MLP) for the MNIST digit recognition task.

Contents

[WIP] Preliminaries

Here we will cover some basic mathematical concepts that will be used throughout the paper. These will be derived from first principles, so don't worry if you're not familiar with them.

Vectors

A vector in mathematics is an ordered list of numbers, represented as a bold lowercase letter, e.g. $\mathbf{v}$. A vector can be represented as:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

We can refer to the first element as v_1, the second as v_2, and the ith element as v_i.
Each element of $\mathbf{v}$ corresponds to a coordinate in some n-dimensional space, where n is just the number of elements in the vector. The vector points to the location of these coordinates in space. For example, take the vector in 2-dimensional space:

$$\mathbf{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

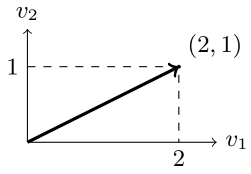


Figure 1: A 2-dimensional vector $\mathbf{v}$

The Transpose of a Vector

Consider a vector $\mathbf{v}$:

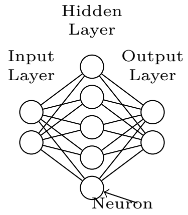
$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}.$$

The transpose of $\mathbf{v}$, denoted $\mathbf{v}^T$, is the same vector, but flipped on its side:

$$\mathbf{v}^T = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}.$$

The transpose of a vector is used to convert a row vector into a column vector, or vice versa.

1 Motivation for the Multilayer Perceptron



The perceptron is composed of multiple artificial neurons. Each of which makes a simple decision based on its inputs. These neurons are subsequently assembled into layers, with each layer functioning as a distinct 'area' of the brain. In this way, the Multilayer Perceptron serves as an abstract model inspired by the brain's structure.

1.1 The Structure and Function of a Biological Neuron

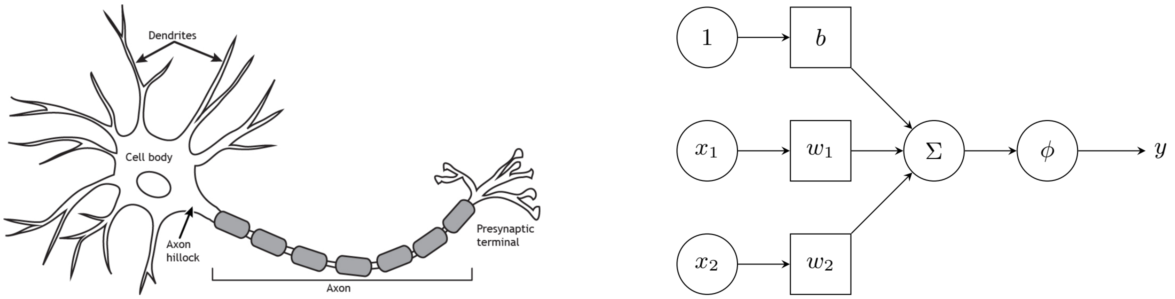
Neurons, the brain's fundamental processing units, transmit information throughout the nervous system. The human brain contains about 86 billion neurons [1], each comprising:

- **Dendrites:** Branch-like structures that receive input signals from other neurons.
- **Cell Body (Soma):** Combines incoming signals and determines whether the neuron should fire.
- **Axon:** Carries the electrical impulse (if the combined signal at the soma exceeds the threshold) to other neurons.
- **Synapse:** Junction between neurons; connects one neuron's axon to another's dendrites.
- **Firing Threshold** The neuron fires only if the total input exceeds a certain threshold.

1.2 Mapping the Biological Neuron to the Artificial Neuron

The perceptron, proposed by Frank Rosenblatt in 1958 [3], is a simplified model of the biological neuron. Its components are:

- **Inputs:** $x_1, x_2, \ldots, x_n$, analogous to dendrites.
- **Weights:** $w_1, w_2, \ldots, w_n$, indicating each input's importance.
- **Summation:** $\sum w_i x_i + b$, analogous to the soma, with b as a bias term.
- **Activation Function (ϕ):** applies a mathematical function to the weighted sum to determine the output y, analogous to the neuron's firing mechanism.
- **Output:** The final signal from the perceptron.



(a) Human neuron [2]

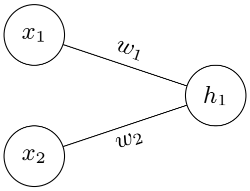
(b) Artificial neuron

Figure 2: Comparison of biological and artificial neurons.

2 Artificial Neurons

2.1 Mathematical Representation

To represent the artificial neueron, we'll start with an example, two inputs x_1 and x_2 into a neueron h_1.



Finding the total input ($\mathbf{z}$)

We could do this by adding up each input multiplied by its weight - indicating how important it is..

$$z = w_1 \cdot x_1 + w_2 \cdot x_2$$

To make this easier for us, we could represent the weights and inputs as vectors:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

We then see then see that we can write:

$$z = [w_1, w_2] \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = w_1 \cdot x_1 + w_2 \cdot x_2$$

We can see that $[w_1, w_2]$ is just out weights vector transposed. We can write this as:

$$z = \mathbf{w}^T \mathbf{x}$$

Applying the Activation Function (ϕ)

We introduce the Heaviside step activation function – which is like an on-off switch. If the total input is greater than 0, the neuron fires; otherwise it doesn't.

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

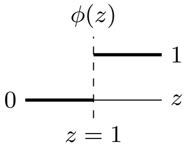


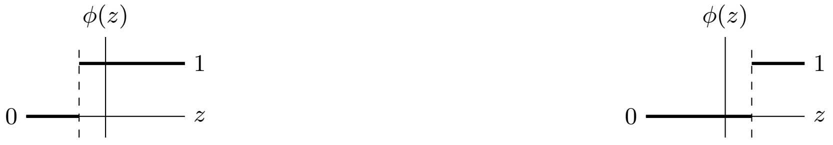
Figure 3: The Heaviside step function

Introducing the bias (b)

What if we want to change how much of a total input it takes for a neuron to fire? We can introduce a bias term b to the total input:

$$z = \mathbf{w}^T \mathbf{x} + b$$

This allows us to shift the total input by an amount b. Which makes the activation function look like:



(a) $b = 0.5$

(b) $b = -0.5$

Figure 4: The effect of the bias term on the activation function

2.2 Learning

The core idea of learning is, given an unfavorable outcome, we look back and try to figure out what we could've done differently to achieve a better outcome. We can only do this if we know how wrong our decision was, and what we should have done instead.

Consider, for a moment , getting a college test result back. If we wanted to get 90% on the test and only got 40%, we know we need to give a much higher weight to studying next time. Whereas, if we got 88% we know we're on the right track, and only need to make minor adjustments.

In an artificial neuron, we can compare the difference between the correct output and the output produced by the neuron. The larger the error, the greater need for adjustment. The learning rate η controls how sensitive we are to these errors. A high learning rate means we make large adjustments, while a low learning rate means we make small adjustments.

The question then becomes, why not always use a high learning rate? Let's revisit the college test scenario. Imagine you aimed for 90% but only scored 40%. A large error signal indicates a significant adjustment is needed. With a high learning rate, you would react very strongly to this error. You might drastically increase your study time, perhaps even overcompensating and studying excessively. This could lead to burnout or neglecting other important aspects of life. Conversely, with a low learning rate, even with a 40% score, your adjustment would be small. You might only slightly increase your study time, potentially not enough to significantly improve your score on the next test. The learning rate, therefore, is a delicate balance. It needs to be high enough to enable learning in a reasonable time, but low enough to prevent wild oscillations or overreactions to errors. The learning rate, is something we can adjust to find the right balance between making large adjustments when we are far off, and making small adjustments when we are close to the correct answer.

Mathematically, we can represent the learning process as follows. We define the error as the difference between the predicted result and the actual result. This error indicates how far off our prediction was from the correct answer, and can be positive (overprediction) or negative (underprediction)

$$\text{Error} = y_{\text{actual}} - y_{\text{predicted}}$$

We then define how much we should change the weight of each input (x_i) as:

$$\Delta w_i = \eta \times \text{Error} \times x_i$$

Here, η is the learning rate, controlling our sensitivity to errors. The multiplication by $x_i$is important because it scales the weight adjustment by the magnitude of the input x_i. Inputs with larger values have a proportionally larger influence on the output, and therefore their weights should be adjusted more significantly for a given error. We also calculate the change for the bias term as: We also calculate how much we should change the bias term as:

$$\Delta b = \eta \times \text{Error}$$

The intuition behind these adjustments is that if the prediction is consistently too low, we increase weights and bias to make firing easier (like studying more). If consistently too high, we decrease them to make firing harder (like studying less). We then update the weights and bias as follows:

$$w_i = w_i + \Delta w_i$$

$$b = b + \Delta b$$

2.3 What does the Artificial Neuron Actually do? - Example: Mushroom Classification

Imagine we're in a field filled with all kinds of mushrooms. Some have dozens of little white spots, and tall stems. While others have very few spots and short terms. You know from past experience that the mushrooms with many spots are poisonous, while the ones with few spots are safe to eat.

But there's a catch. Sometimes we find mushrooms that are hard to classify - they have medium length stems, or only a few spots. Where exactly do we draw the line between poisonous and safe mushrooms? You may not know from a glance, and you certainly don't want to eat a poisonous mushroom to find out.

This is exactly the problem the artificial neuron helps solve. It learns to draw a line in the mushroom field that separates the poisonous mushrooms from the safe ones.
Lets define the two inputs to our neueron as:

$$x_1 = \text{number of spots} \quad \text{and} \quad x_2 = \text{length of the stem}$$

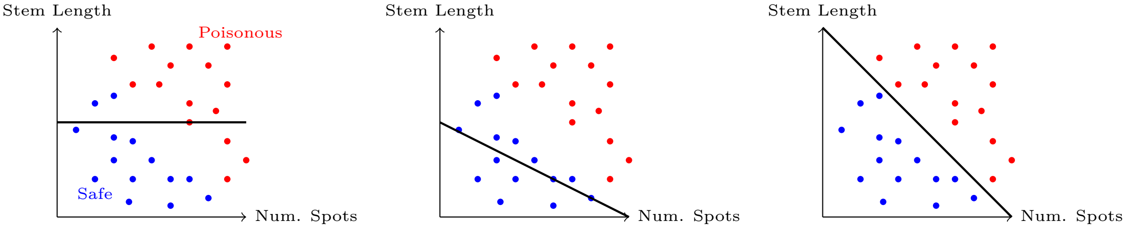
The artificial neueron then forms a linear combination of these inputs:

$$z = \sum_{i=1}^{2} w_i x_i + b = w_1 x_1 + w_2 x_2 + b \Rightarrow z = \mathbf{w}^T \mathbf{x} +$$

where our weights scale the importance of each input. The neueron then decides whether the mushroom is poisonous or not:

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

where, $y = 1$ means the mushroom is poisonous, and $y = 0$ means it is safe to eat. The perceptron learns the weights w_1 and w_2 to draw the best line that separates the two classes of mushrooms.



(a) Before training (b) After 100 training passes (epochs) (c) Final decision boundary

Figure 5: Artificial Neuron learning to separate safe and poisonous mushrooms.

We refer to this line as the decision boundary, as it separates the two classes of mushrooms. We now have a simple model, that given the number of spots and the length of the stem, can predict whether a mushroom is poisonous or safe to eat. This is the essence of what an artificial neuron does - it learns to draw decision boundaries in the input space to separate different classes of data.

In essence, an artificial neuron integrates multiple inputs by scaling them with weights, adds a bias to adjust the activation threshold, and passes the resulting sum through a non-linear activation function. Its learning process—driven by error correction—enables it to modify its parameters to better approximate the desired output. This rigorous yet intuitive framework is the cornerstone of more complex neural network architectures.

3 A Layer of Neurons

3.1 Recap of the Single Neuron and Intro to Layers

The artificial neuron is a simple unit, like a tiny decision maker. It takes inputs representing different features (think stems and spots of mushrooms), it gives these features different weights (importance), adds them up, applies a little tweak (a bias) and runs it through an activation function (like a yes/no switch) to decide what to output. It learns by tweaking these weights and biases to make better decisions on examples its given.

Now, instead of one neuron working alone, we introduce a layer of neurons, each of them looking at the same inputs but looking at them in different ways, for example one might focus on the color of the mushroom, while another might focus on its size. Together, these neurons can work together to make more complex decisions, where we can consider multiple features at once.

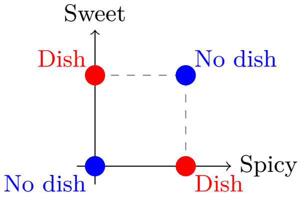
3.2 The XOR Problem

A single artificial neuron can separate data with only one straight line—great if your data is linearly separable. But some datasets require more complex boundaries. One classic example is the XOR (exclusive OR) problem: how do we classify data points that should be labeled “1” if they satisfy exactly one condition (but not both), and “0” otherwise?

Analogy: A Chef and Two Flavors
Imagine a chef wants to create a dish that is appealing only if it is *either* sweet *or* spicy, but not both (that combination clashes) and not neither (too boring). This is precisely the XOR condition: the dish is appealing if it is sweet *or* spicy, but not both. The chef needs to find the right balance of flavors to create a dish that satisfies this condition.

Sweet	**Spicy**	**Dish**
No	No	No (boring dish)
No	Yes	Yes (sweet dish)
Yes	No	Yes (spicy dish)
Yes	Yes	No (flavours clash)

(a) XOR truth table

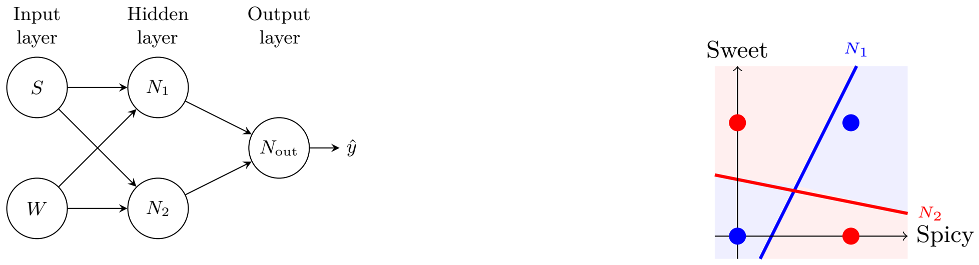


(b) Visualization of XOR: Red = 1, Blue = 0

Figure 6: The XOR problem

As we can see, we can’t split this data with a single straight line so that all the red points are one side and all the blue points are on the other side. This is where a single neuron fails. We need a more complex decision boundary. To solve this problem we can use multiple neurons in the same layer, where each neueron draws its own boundry ad the final decision is made by combining the outputs of these neurons.

Returning to the chef analogy: the chef might hire two sous-chefs. One sous-chef checks sweetness (N_1); the other checks spiciness (N_2). Their “votes” are then combined in a final decision. In neural network terms, each neuron in the hidden layer focuses on a specific aspect of the input; then the outputs are aggregated by the next neuron. This layering of simpler decisions leads to richer, non-linear decision boundaries—enabling the network to classify the XOR data correctly.



(a) Architecture of the chef analogy

(b) Decision boundaries

Figure 7: A layer of neurons for the XOR problem

4 Why stop at one layer?

The XOR problem (chef anology) is a small illustration on non-linearly seperable problems. Most real word tasks have layers of complextity that drawf the XOR example. For example:

- **Image Recognition**: Deciding whethera cat or dog involves thousdands or millions of pixels. A single layer of neueron cannot capture the patterns of edges, shapes, and textures that define a cat or dog.
- **Natural Language Processing**: Understanding a sentence involves understanding the meaning of each word, the context in which they are used, and the relationships between them. A single layer of neurons cannot capture the complexity of language.

4.1 The Power of Chaining Layers

Just having one layer of neurons is like having a single sous-chef in the kitchen. They can only focus on one aspect of the dish. To create a complex dish, the chef needs multiple sous-chefs, each focusing on different aspects of the dish. Similarly, to solve complex problems, we need multiple layers of neurons, each layer focusing on different aspects of the input data. This is the essence of deep learning: using multiple layers of neurons to learn complex patterns in data.

One way to overcome these limitations is to chain mutliple layers of neuerons so that the output of one layer becomes the input of the next. For example, in image recoginition, one layer might learn to respond to simple edges or colors, while the next layer might take these edges as inputs and learn to detect more complex shapes, like corners and textures, a third layer might combine these shapes into an understanding of objects like a cats ear, or a dogs nose. This chaining of layers is what gives deep learning its name. When we talk about hidden layers, we refer to the layers of nuerons between the input and output layers. These layers are called hidden because we don't directly observe their ouputs, but rather they are intermediate processing steps. Networks with one more more hidden layers are often called multi-layer perceptrons (MLPs). The more layers we add, the deeper the network becomes, and the more complex patterns it can learn - at a cost of more computational resources and longer training times.

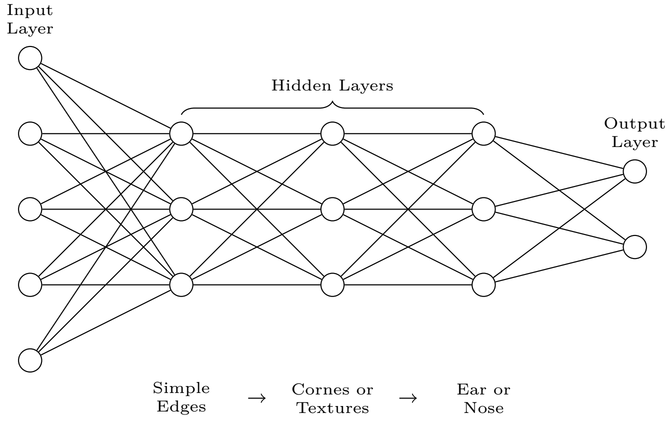
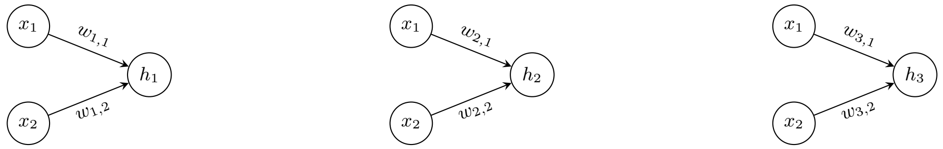


Figure 8: A deep neural network for image recognition

4.2 The Forward Pass : The journey from input to output

Lets define a network with three neuerons, h_1, h_2, h_3, each of which get the same inputs, x_1, x_2



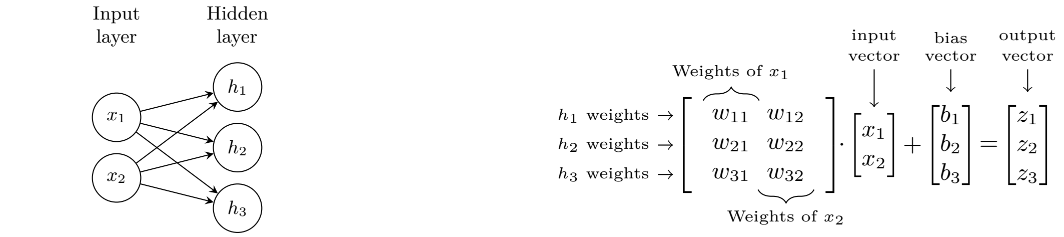
$$z_1 = w_{1,1}x_1 + w_{1,2}x_2 + b_1 \qquad z_2 = w_{2,1}x_1 + w_{2,2}x_2 + b_2 \qquad z_3 = w_{3,1}x_1 + w_{3,2}x_2 + b_3$$

$$z_1 = \begin{bmatrix} w_{1,1} & w_{1,2} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_1 \qquad z_2 = \begin{bmatrix} w_{2,1} & w_{2,2} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_2 \qquad z_3 = \begin{bmatrix} w_{3,1} & w_{3,2} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_3$$

Lets let the vector **z** represent the outputs of these neuerons:

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

This is the essence of a forward pass, we've desribed a network of two input neuerons and three hidden neuerons.



(a) Architecture of the chef analogy

(b) Matrix representation of the forward pass

Figure 9: Representation of the forward pass

5 References

References

[1] Azevedo et al. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain, 2009.

[2] Physio-pedia.com. hhttps://www.physio-pedia.com/neurone.

[3] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain, 1958.

TO DO

- Current Section
  - Finish the forward pass sub section
  - Subsection on how each layers output is compted $z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$ and $a^{(l)} = \phi(z^{(l)})$
  - emphasis on vectorization for efficiency
  - Subsection on cost functions and error metrics
- New sub section on learning
  - Summary of how we adjust weights
  - Backprop
  - Why we use smooth activation functions
  - discuss sigmoid, tanh, relu, leaky relu (pros cons - sigmoid staturation, tanh zero centered, relu simple, leaky relu to prevent dying neurons)
  - When to use each (typical guidlines - relu for hidden layers, tanh for output, leaky relu for deeper networks, softmax for multiclass classification)
  - derivaties of common activation functions
  - chain rule application from output layer back to input layer
  - how to update weights and biases using the computed gradients
  - the role of the learning rate
- Choice of cost function
  - Mean Squared Error, Cross Entropy
  - explain the difference between the two, i.e. MSE for regression, cross entropy for classification
- Error Backpropogation
  - how the network distributes the error back through the network
  - deriving partial derivaties for weights and biases at each layer
- Optimization and Training
  - Batch Based Methods
  - Stochastic Gradient Descent, Mini Batch Gradient Descent
  - Explain the difference between the two, i.e. SGD for noisy data, Mini Batch for more stable convergence
  - Improving convergence
  - Momentum, RMSProp, Adam
  - Explain the role of each, i.e. momentum to prevent oscillations, RMSProp to adjust learning rates, Adam as a combination of the two
  - Explanation of the hyperparameters for each optimization method
- Regularization
  - Overfitting and Underfitting
  - Techniques to prevent overfitting, i.e. dropout, L1/L2 regularization, early stopping
- Initalization
  - Why naive (all zeros) initialization fails
  - Recommended schemes for initialization, i.e. Xavier, He
- Building the Multilayer Perceptron for MNIST
  - Data input: Organize your data and labels
  - Network structure: define layers, number of
  - Forward Propogation: Implement the sequence of matrix multiplications, biases and activations
  - Loss calculation: e.g. cross entropy for classification tasks
  - Backpropogation: Compute gradients layer by layer
  - Weight updates: Incorporate an optimization method (SGD, Adam, etc)
  - Training loop: Shufflee data, iterate over epochs, track training and validation loss
- Extensions and Next steps
  - Deeper architectures: convoulition nueral networks, recurrent neural networks, transformers
  - Modern practices: Batch normalization, advanced regularization, learning rate schedules
- Modern Practices
  - Batch Normalization
  - Advanced Regularization
  - Learning Rate Schedules