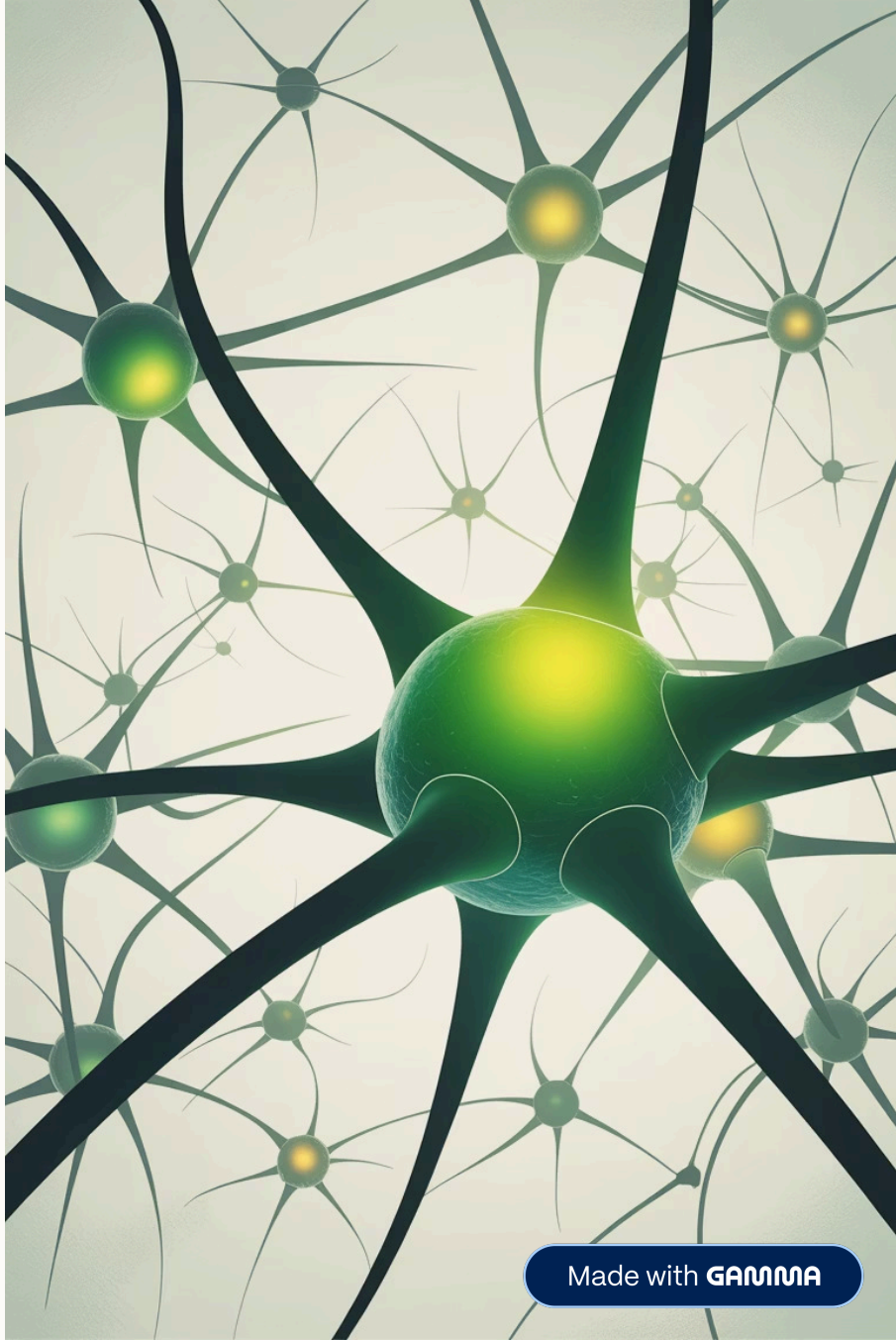
Neural Network's Learning Explained

Demystifying the technology behind artificial intelligence and machine learning



Model Representation Paradigms

Machine Learning Models Categorized by Knowledge Representation

1. Parametric Models (Weight-Based)
   A parametric model is a model that describes data using a fixed number of parameters.No matter how much data you give it, the number of parameters stays the same.
   In ML a parametric ML model assumes a specific functional form and learns a fixed number of parameters from data.
   a. Linear/Logistic Regression
   b. Neural Networks/Deep Learning
   c. Support Vector Machines
2. Tree-Based Models
   These build hierarchical splits on features, no explicit weights, knowledge is in the tree structure.
   a. Decision Trees
   b. Random Forest
   c. Gradient Boosted Trees (XGBoost, LightGBM)
3. Instance-Based Models (Non-Parametric)
   Instance-Based models Store training data and compute on-the-fly, no fixed parameters learned.
   a. K-Nearest Neighbors (KNN)
   b. Kernel Density Estimation
4. Other categories
   Probabilistic: Learns class probabilities from data counts, i.e Naive Bayes
   Ensemble: Combines multiple models (often trees and others), i.e AdaBoost, Voting Classifiers

Model Representation Paradigms cont...

Where Do Neural Networks Fit?

Neural networks, as already listed under parametric models, belong to the class of parametric learning algorithms.

They learn a fixed set of parameters, namely weights and biases, whose total number is determined by the network architecture. Once training is complete, predictions rely only on these learned parameters and do not require storing the training data.

What Are Neural Networks?

Brain-Inspired Computing

Neural networks are computing systems inspired by how biological brains process information. They consist of interconnected layers of "neurons" that work together to recognize patterns, make predictions, and solve complex problems.

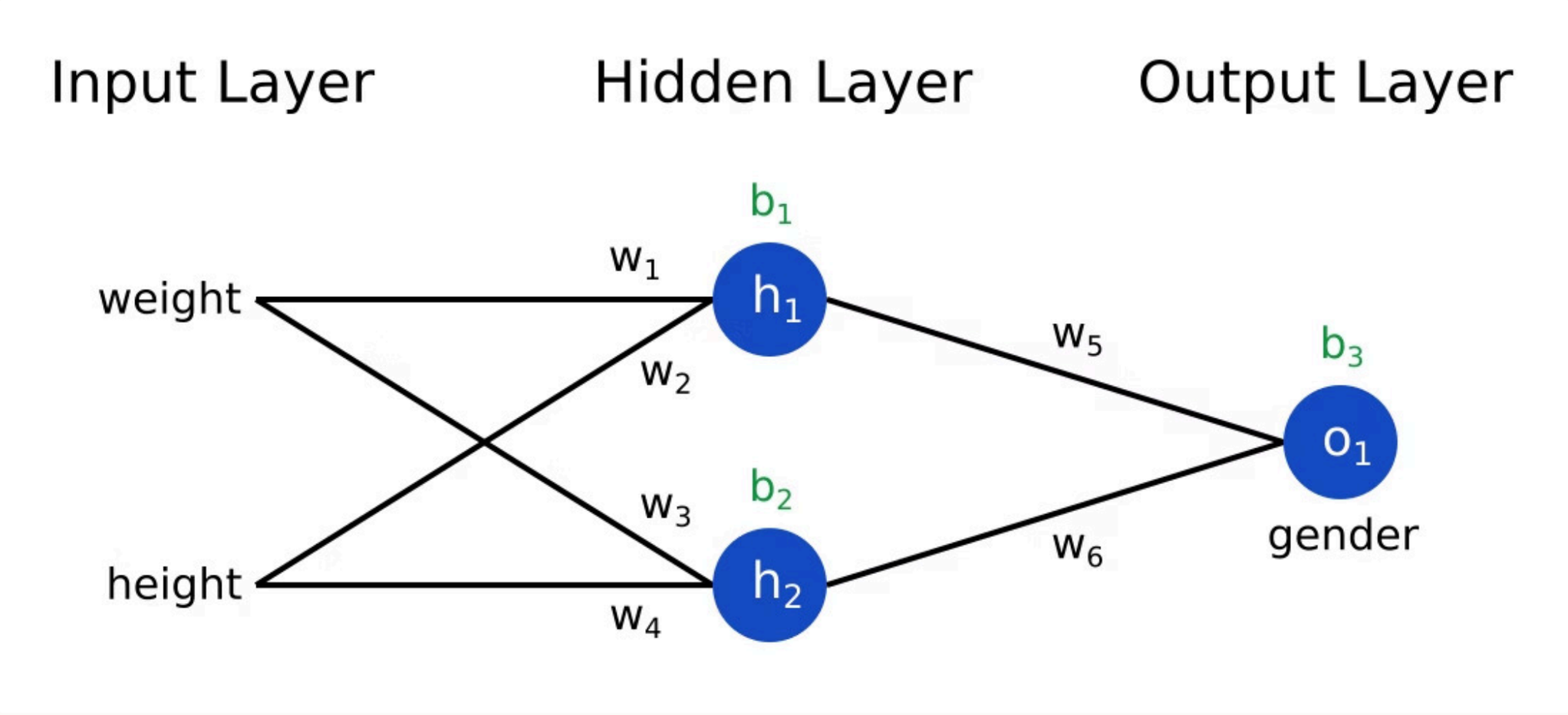
These networks learn from examples rather than following explicitly programmed instructions, making them ideal for tasks like image recognition, language processing, and predictive analytics.

Why Use Neural Networks?

Neural networks are used to solve complex, non-linear, and high-dimensional problems that traditional algorithms struggle with, such as image recognition, natural language processing, and predictive analytics

- Pattern Recognition
  Identify complex patterns in data that are difficult for traditional algorithms to detect
- Adaptive Learning
  Improve performance through experience without being explicitly reprogrammed
- Versatile Applications
  Apply to diverse fields from healthcare to finance to autonomous vehicles

Anatomy of a Neural Network



01

Input Layer

Receives raw data and passes it to the network

02

Hidden Layers

Process information through multiple transformations and feature extraction

03

Output Layer

Produces final predictions or classifications

Forward Propagation: The Forward Pass

How Information Flows

A forward pass is the initial phase in neural network training where input data travels from the input layer, through hidden layers, to the output layer to generate predictions

During forward propagation, data travels from the input layer through each hidden layer to the output layer. Each neuron receives inputs, multiplies them by learned weights, adds a bias, and applies an activation function.

This process transforms raw input into meaningful predictions. Think of it as the network making its best guess based on current knowledge.

Steps:

- **Input values**: Data enters the network
- **Weighted sum**: Values multiplied by weights
- **Activation Function**: Transformation applied
- **Output prediction**: Final result produced

Backpropagation: Learning from Mistakes

After forward propagation, the network compares its prediction to the actual result and calculates the error. Backpropagation works backward through the network, adjusting the weights to reduce this error.

It utilizes the chain rule of calculus to efficiently determine how much each parameter contributed to the error, allowing optimizers (like gradient descent) to update weights and minimize loss.

Steps:

- **Calculate Error:** Compare prediction to actual value
- **Propagate Backward:** Distribute error across layers
- **Update Weights:** Adjust connections using gradient descent
- **Iterate:** Repeat with new data to improve

This iterative process is how neural networks learn and improve their accuracy over time.

Activation Functions: Adding Non-Linearity

Activation functions are an integral building block of neural networks that enable them to learn complex patterns in data.

Why They're Essential

Activation functions introduce non-linearity into the network, enabling it to learn complex patterns and make sophisticated decisions. Without them, a neural network could only model linear relationships.

Each function transforms the weighted sum of inputs into an output that determines whether a neuron "fires" and passes information to the next layer.

Why Are Activation Functions Important?

1. **Introduce Non-Linearity:**
   Real-world data is rarely linear. Activation functions allow neural networks to model non-linear relationships.
2. **Make Neural Networks Universal Function Approximators:**
   With non-linear activation functions, neural networks can approximate any function and solve, a wide variety of problems.
3. **Enable Hierarchical Learning:**
   Activation functions help neurons learn complex patterns at deeper layers.

Activation Functions: Adding Non-Linearity

Common activation functions

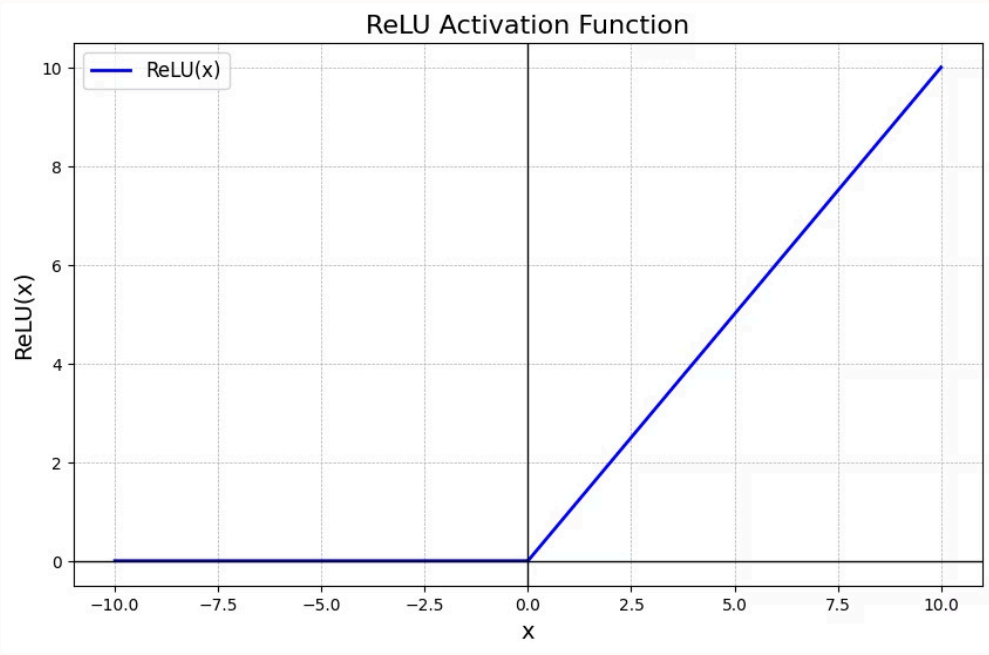
a. ReLU Activation (Rectified Linear Unit)

It thresholds the input at zero, returning 0 for negative values and the input itself for positive values.

Most popular for hidden layers. Fast, efficient, and works well in practice

ReLU formula:-

$$f(x) = \max(0, x)$$



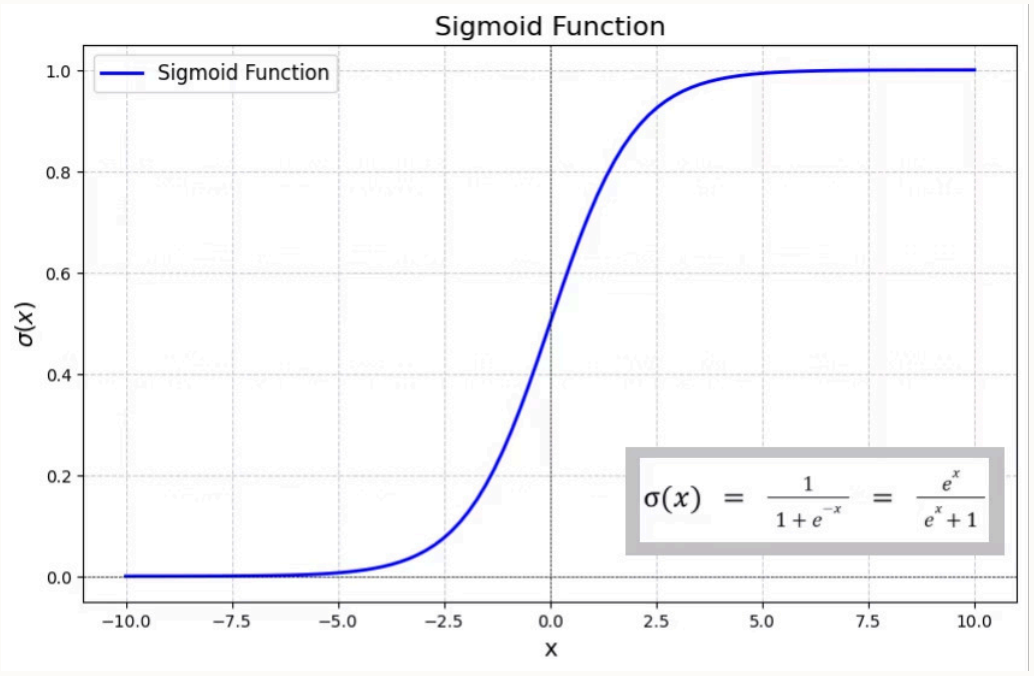
b. Sigmoid activation

Maps inputs to a range between 0 and 1; commonly used for binary classification in output layers.

Smooth curve from 0 to 1. Good for binary classification

Sigmoid function:-

$$f(x) = 1/(1 + e^{-x})$$



Activation Functions: Adding Non-Linearity

Common activation functions cont...

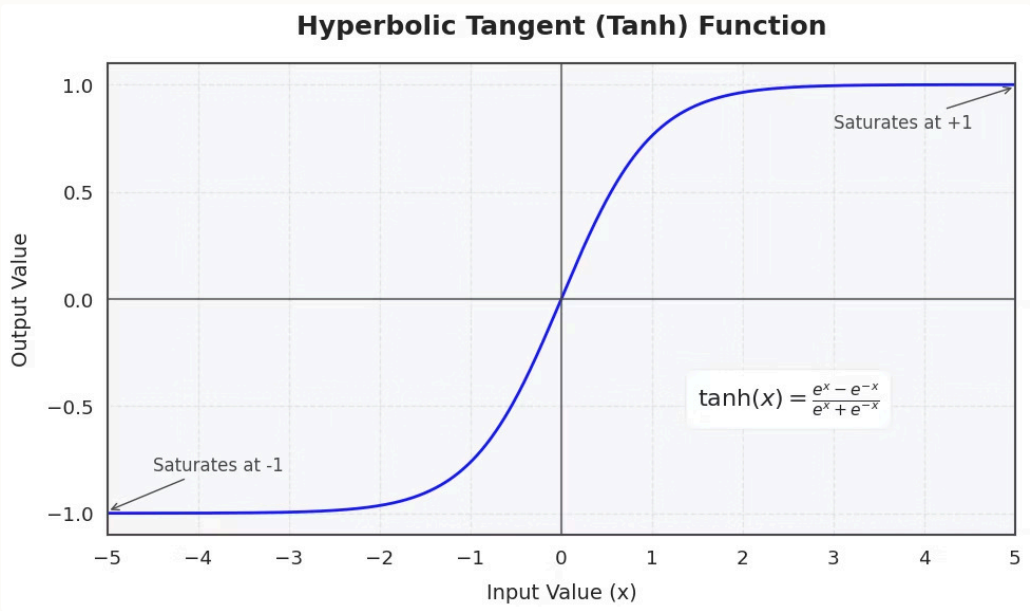
c. Tanh (Hyperbolic Tangent)

Similar to sigmoid but maps inputs between -1 and 1, often leading to better convergence for hidden layers.

S-shaped from -1 to 1. Often used in hidden layers

Tanh formula

$$f(x) = (e^x - e^(-x))/(e^x + e^(-x))$$



b. Softmax activation

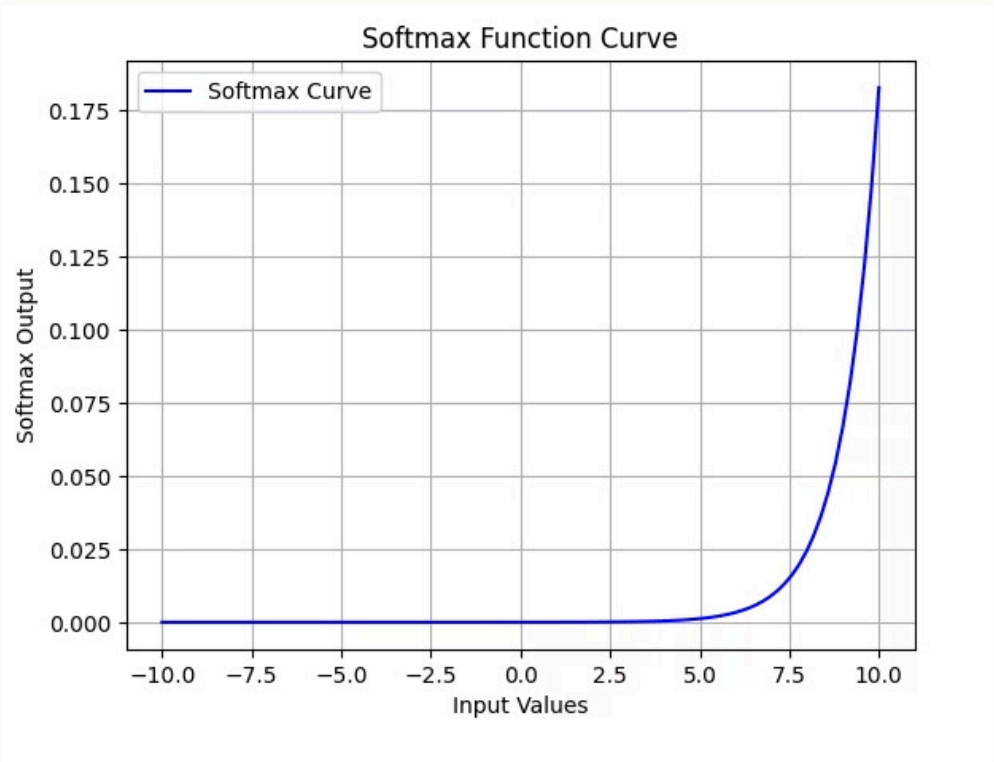
Designed to handle multi-class classification problems.

It works by squashing the output values of each class into the range of 0 to 1 while ensuring that the sum of all probabilities equals 1.

Converts outputs to probabilities. Perfect for multi-class classification

Softmax function

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$



How a Neural Network Works

A neural network is a system that learns to make predictions or decisions by studying examples.

1. Layers (Structure)
   The network is made of layers:
   a. **Input layer**: Takes in the data (for example: numbers, images, or text).
   b. **Hidden layers**: Process the data by doing calculations and finding patterns.
   c. **Output layer**: Gives the final answer or prediction.
2. Neurons (Small Workers)
   Each layer contains neurons, which are small units that:
   a. Receive numbers
   b. Do simple math
   c. Pass the result forward to the next layer
3. Weights and Biases
   Weights decide how important each piece of information is.
   Biases help adjust the result so the network can make better decisions.
4. Learning
   The network learns by:
   a. Making a guess
   b. Comparing it with the correct answer
   c. Adjusting its weights and biases to reduce mistakes.

By repeating this process many times the network become more accurate.

Before We Dive In: Building Your Foundatio

The next sections introduce calculus and computational graphs. These might sound intimidating, but they're just practical tools that we'll break down with simple examples.

We're going to cover two key foundations:

- **Calculus Basics:** Gradient descent, power rule, chain rule, partial derivatives
- **Computational Graphs:** How data flows through calculations

Prerequisites: Calculus Basics for Neural Networks

Neural networks learn by adjusting weights to reduce errors. To know which direction to adjust weights, we need calculus specifically, we need to understand how small changes in weights affect the final error. That's what calculus does.

The Big Picture: Gradient Descent

Imagine you're standing on a dark mountain and want to reach the bottom. You can't see the whole mountain, but you can feel the slope under your feet. Gradient descent is exactly this: you feel which way is downhill (the gradient), take a step in that direction, and repeat until you reach the bottom. In neural networks, the "mountain" is the loss function, and we're trying to find the lowest point (minimum loss).

The Gradient: What Is It?

The gradient is simply the slope of a curve at a specific point. It tells you:

- Which direction to move (uphill or downhill)
- How steep the slope is (how fast to move)

If the gradient is positive, the function is going up. If it's negative, it's going down. We move in the opposite direction of the gradient to reach the minimum.

Calculus Rules You Need to Know

1. Power Rule

What it does: Helps you find the slope of simple polynomial functions.

Simple explanation: If you have a function like f(x) = x², the power rule tells you how fast it's changing at any point.

The rule: If $f(x) = x^n$, then the derivative is $f'(x) = n \times x^{n-1}$

Simple example:

- f(x) = x²
- Using power rule: $f'(x) = 2 \times x^{2-1}$ = 2x
- At x = 3: the slope is 2 × 3 = 6 (the function is going up steeply)
- At x = 0: the slope is 2 × 0 = 0 (flat, this is the minimum)

2. Chain Rule

What it does: Helps you find the slope when functions are nested inside each other (like Russian dolls).

Simple explanation: When you have a function inside another function, you can't just apply the power rule directly. The chain rule breaks it down into smaller pieces.

The rule: If you have $f(g(x))$, then the derivative is $f'(g(x)) \times g'(x)$

(Derivative of outer function × Derivative of inner function)

Simple example:

- f(x) = (x + 2)²
- This is a function inside a function: outer is "square it", inner is "add 2"
- Using chain rule:
  - Derivative of outer: 2(x + 2)
  - Derivative of inner: 1
  - Combined: 2(x + 2) × 1 = 2(x + 2)
- At x = 1: the slope is 2(1 + 2) = 6

3. Partial Derivative

What it does: Finds the slope when you have multiple variables, but you only care about how one of them affects the output.

Simple explanation: Imagine a function that depends on both weight and bias. A partial derivative asks: "If I only change the weight (keeping bias fixed), how does the output change?" It's like asking about one ingredient's effect on a recipe while ignoring all others.

The notation: $\partial f/\partial w$ means "the partial derivative of f with respect to w"

Simple example:

- $f(w, b) = w \times 2 + b$
- Partial derivative with respect to w: $\partial f/\partial w = 2$ (changing w by 1 changes output by 2)
- Partial derivative with respect to b: $\partial f/\partial b = 1$ (changing b by 1 changes output by 1)
- This tells us: weight changes have twice the impact of bias changes

Why This Matters for Neural Networks

In neural networks, we have a loss function that depends on many weights and biases. We use:

- Power rule and chain rule to compute how the loss changes with respect to each weight
- Partial derivatives to isolate the effect of each individual weight
- The gradient (slope) to know which direction to adjust each weight

This is exactly what backpropagation does automatically—it uses these calculus rules to compute gradients for every weight in the network.

Weight-Based Training: How It Really Works

Weight-based training is the process of finding the perfect weights and biases that make your network's predictions match the true answers. We measure how wrong we are using a **cost function**, then use **gradient descent** to adjust weights step-by-step until we find the best values.

Key Concepts

Definitions:

Cost Function (Loss): Measures the difference between predicted and actual values. For regression, we use Mean Squared Error (MSE): $L = 1/m(y - \hat{y})^2$

Gradient: The slope of the loss curve. It tells us which direction and how fast to move the weight to reduce loss.

Gradient Descent: The algorithm that updates weights using: $w_{\text{new}} = w_{\text{old}} - \alpha \times \frac{dL}{dw}$, where `α` is the learning rate.

Learning Rate (`α`): Controls step size. Too large = overshooting, too small = slow learning.

A Simple Training Example:

Setup:

- Input: `x = 1`
- Target: `y = 2`
- Initial weight: $w_0 = 0$
- Initial bias: $b_0 = 0$
- Learning rate: `α = 0.5`
- Network equation: $\hat{y} = w \times x + b$

Iteration 1:

Forward Pass: $\hat{y}_0 = 0$

Loss: $L_0 = ½(0 - 2)^2 = 2$

Gradients: $\partial L/\partial w = -2$, $\partial L/\partial b = -2$

Weight Update: $w_1 = 0 - 0.5 \times (-2) = 1$, $b_1 = 0 - 0.5 \times (-2) = 1$

Iteration 2:

Forward Pass: $\hat{y}_1 = 1 \times 1 + 1 = 2$

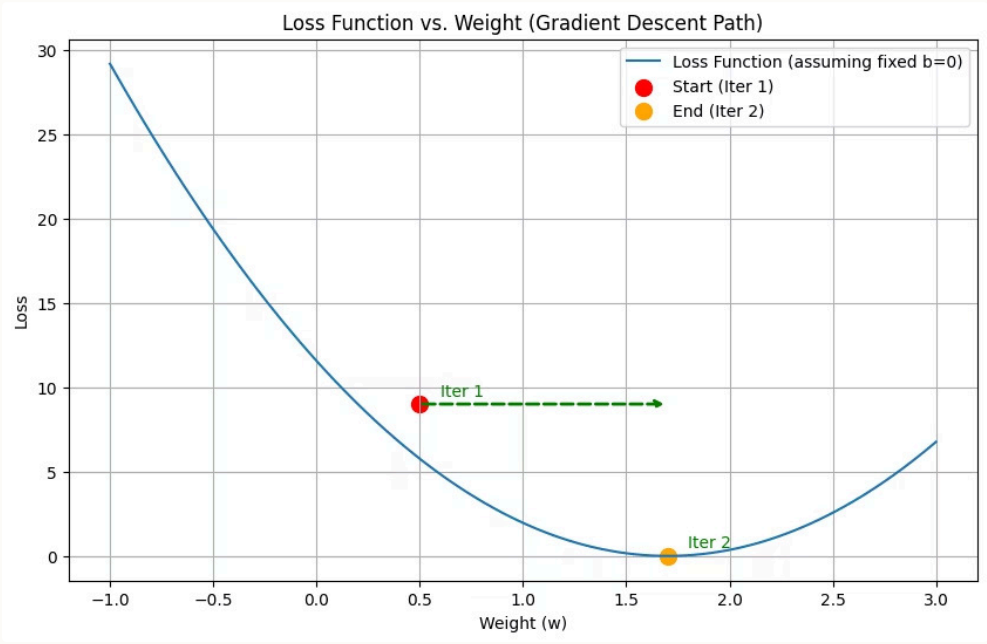
Loss: $L_1 = ½(2 - 2)^2 = 0$

✓ **Network has converged! Perfect prediction achieved.**

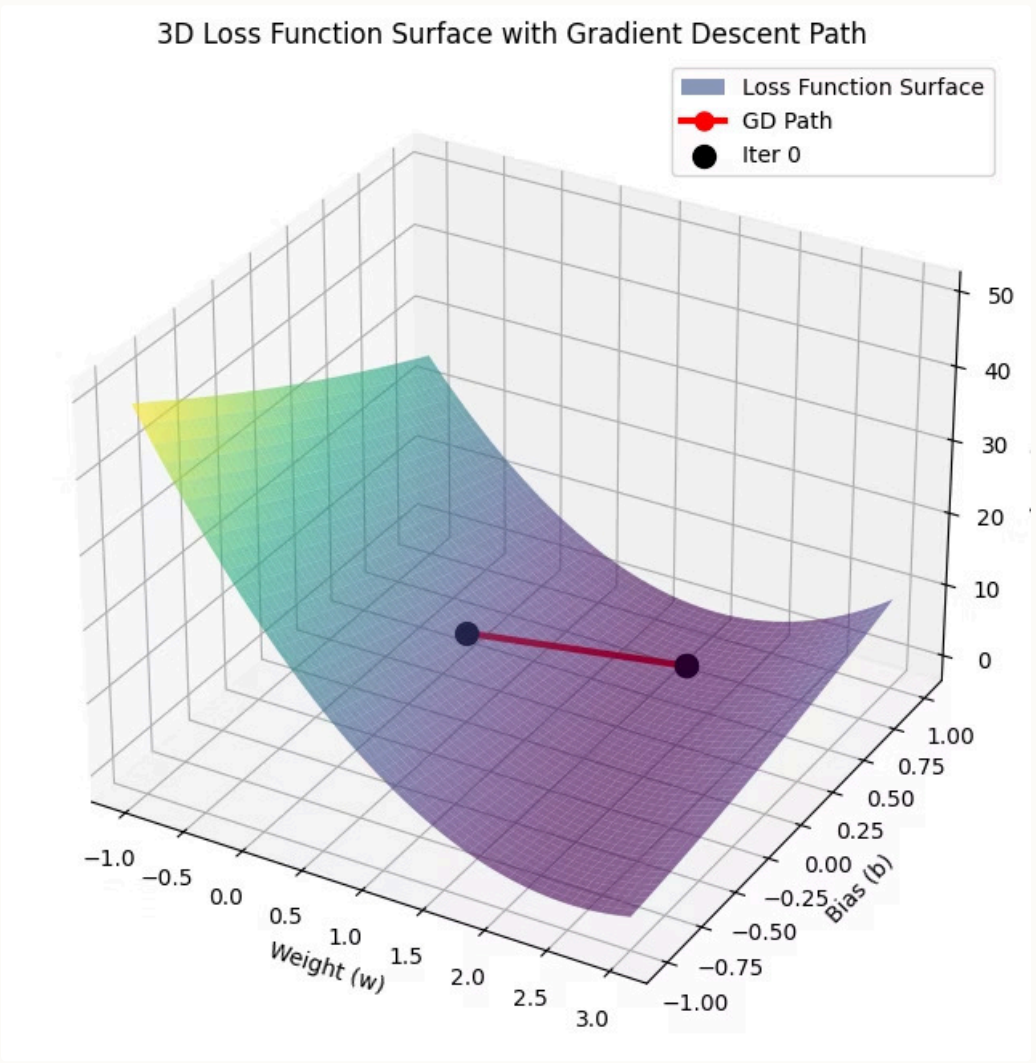
Visualizing gradient descent

The parabola below shows how loss changes with different weight values. Gradient descent moves us down the curve toward the minimum loss. Each iteration takes a step proportional to the gradient.

The 'Loss Function vs. Weight (Gradient Descent Path)' plot visualizes the relationship between the weight (w) and the loss (L), demonstrating how the gradient descent algorithm navigates this landscape.



3D representation of the 'loss' (how wrong the model is) across all possible values for two key adjustable parameters, 'weight' and 'bias'. The lowest points on this surface indicate the best parameter combinations where the model makes the fewest errors.



Weight-Based Training: The Simple Summary

The 5-Step Process:

1. **Pick a Weight:** Start with a random weight value. This is your first guess.
2. **Make a Prediction:** Use the weight to predict: $\hat{y} = \text{weight} \times \text{input}$
   (This is what your network thinks the answer is)
3. **Calculate the Loss (How Wrong Are We?):** Loss = $(\text{actual answer} - \text{predicted answer})^2$
   The bigger the loss, the more wrong you are.
4. **Compute the Gradient (Which Direction to Move?):** The gradient tells you: "Should I make the weight bigger or smaller to reduce loss?"
   It's like asking: "Which way should I walk to get to the bottom of the hill?"
5. **Update the Weight (Take a Step):** New weight = Old weight - (learning rate $\times$ gradient)
   You move the weight in the direction that reduces loss.
6. **Repeat Until Convergence:** Keep doing steps 2-5 until the loss stops getting smaller. When loss is at its minimum, you've found the best weight!

The Big Picture:

"Think of it like finding the lowest point in a valley. You start somewhere random, measure how high you are (loss), then take a step downhill (gradient descent). Keep stepping downhill until you reach the bottom. That's weight-based training!"

Computational Graphs

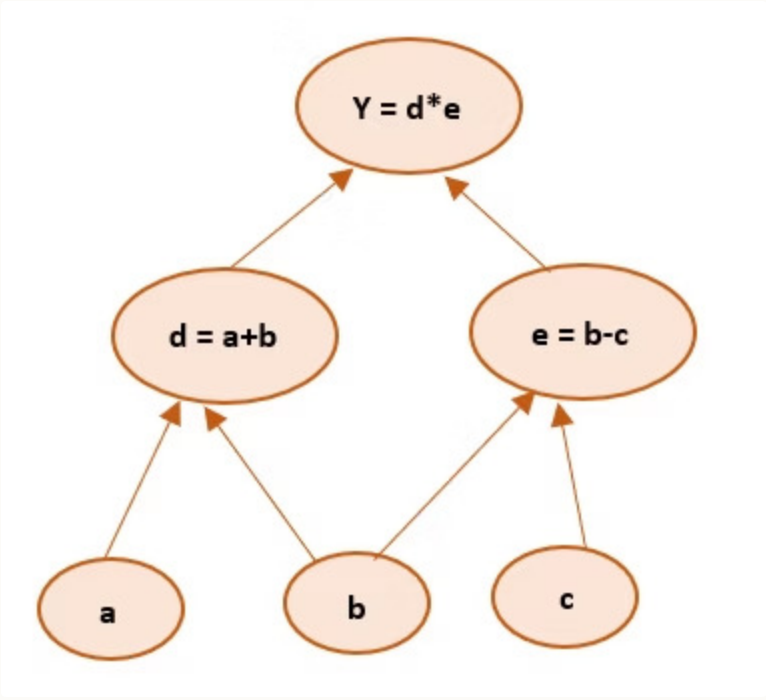
A computational graph is a visual representation of how data flows through mathematical operations. Each node represents an operation (like addition or multiplication), and edges show how data moves between operations.

Example: y = (a+b)*(b-c)

To break this down:

- Step 1: $d = a + b$ (addition operation)
- Step 2: $e = b - c$ (subtraction operation)
- Step 3: $y = d * e$ (multiplication operation)

The graph shows these three operations as nodes, with arrows indicating which variables feed into each operation. This structure is essential for **backpropagation**, as it allows the network to trace how changes in inputs affect the final output.



This diagram visually represents the flow of data from inputs (a, b, c) through intermediate operations to the final output (f).

Computation graph is a key idea in deep learning and it is also how programming frameworks like tensor-flow automatically computes derivatives of neural networks.

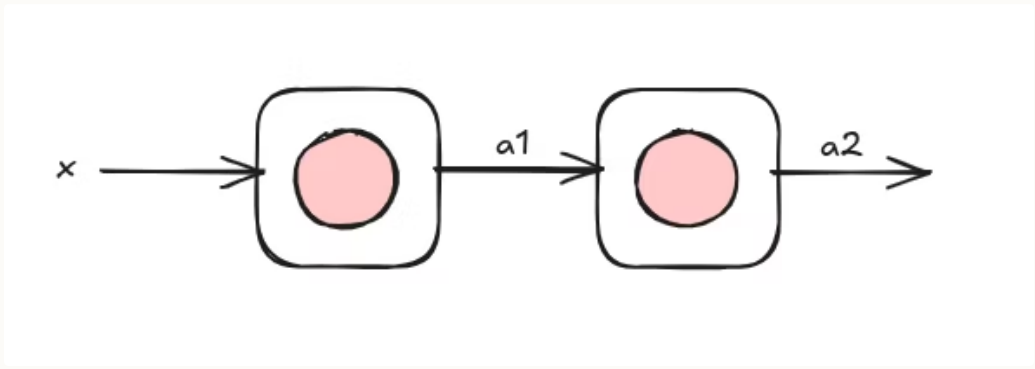
Back-propagation is an efficient way to compute derivatives

if N nodes and P parameters, it computes derivatives in roughly N+P steps rather than N x P steps

N	P	N + P	N * P
10,000	100,000	110,000	1B

Working Example: Forward Pass

Let's walk through a simple neural network with one hidden layer to see how data flows forward and how loss is calculated.



A simple neural network with one hidden layer. Data flows from input → hidden layer → output layer, producing a final prediction.

The goal here is to calculate the final output and the resulting loss given specific inputs and weights.

Given Values:

- Input: $x = 1$
- Target output: $y = 5$
- Layer 1 weights: $w_1 = 2$, bias $b_1 = 0$
- Layer 2 weights: $w_2 = 3$, bias $b_2 = 1$

Step-by-Step Calculations:

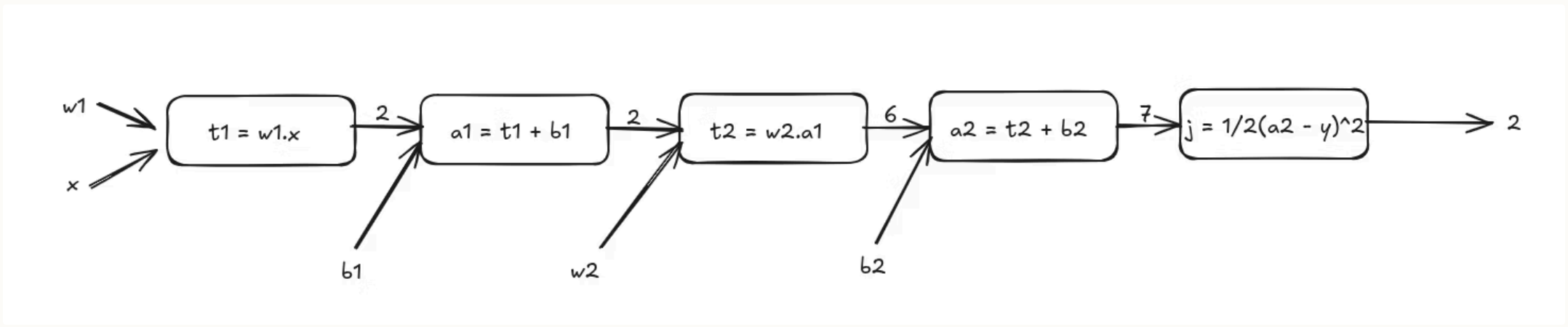
1. **Step 1: Layer 1 Output** $a_1 = w_1 \times x + b_1 = 2 \times 1 + 0 = 2$
2. **Step 2: Layer 2 Output** $a_2 = w_2 \times a_1 + b_2 = 3 \times 2 + 1 = 7$
3. **Step 3: Calculate Loss** $Loss = ½(a_2 - y)^2 = ½(7 - 5)^2 = ½(4) = 2$

What This Means:

Our network predicted 7, but the correct answer was 5. The loss of 2 tells us how far off we were. In the next step (backpropagation), we'll use this loss to adjust the weights to make better predictions.

The Computational Graph:

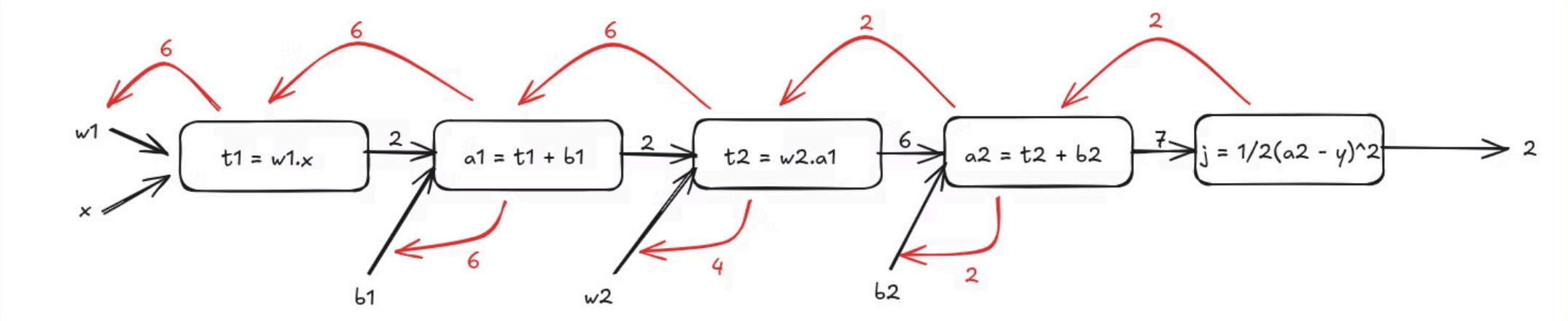
The forward pass can be visualized as a series of operations:



This graph structure is what allows backpropagation to efficiently compute how to adjust each weight.

Backward Pass: Computing Gradients

To adjust weights and improve predictions, we need to know how much each weight contributed to the loss. We use the Chain Rule to work backwards from the loss to each weight.



Step 1: Gradient of Loss with respect to Layer 2 Output

$\partial J/\partial a_2 = (a_2 - y) = (7 - 5) = \mathbf{2}$

Step 2: Gradients for Layer 2 Parameters

For bias b_2: $\partial J/\partial b_2 = 2 \times 1 = \mathbf{2}$

For weight w_2: $\partial J/\partial w_2 = 2 \times 1 \times a_1 = 2 \times 2 = \mathbf{4}$

Step 3: Gradient flowing back to Layer 1

$\partial J/\partial a_1 = 2 \times 1 \times w_2 = 2 \times 3 = \mathbf{6}$

Step 4: Gradients for Layer 1 Parameters

For bias b_1: $\partial J/\partial b_1 = 6 \times 1 = \mathbf{6}$

For weight w_1: $\partial J/\partial w_1 = 6 \times 1 \times x = 6 \times 1 = \mathbf{6}$

Summary of Gradients:

- $\partial J/\partial w_1 = \mathbf{6}$
- $\partial J/\partial b_1 = \mathbf{6}$
- $\partial J/\partial w_2 = \mathbf{4}$
- $\partial J/\partial b_2 = \mathbf{2}$

These gradients tell us how much to adjust each weight. In the next step, we'll use these to update the weights and reduce the loss.

Updating Weights and Bias

To update the weights, we use the **Gradient Descent** formula. This is where the network actually "learns" by shifting its parameters in the opposite direction of the gradient to minimize the loss.

The Update Rule:

New Parameter = Old Parameter - (α × Gradient)

Where α (learning rate) = 0.1

Step 1: Updating Layer 2 Parameters

For w_2:

w_2_new = 3 - (0.1 × 4) = 3 - 0.4 = **2.6**

For b_2:

b_2_new = 1 - (0.1 × 2) = 1 - 0.2 = **0.8**

Step 2: Updating Layer 1 Parameters

For w_1:

w_1_new = 2 - (0.1 × 6) = 2 - 0.6 = **1.4**

For b_1:

b_1_new = 0 - (0.1 × 6) = 0 - 0.6 = **-0.6**

Summary of Updated Parameters:

- w_1: 2 → 1.4
- b_1: 0 → -0.6
- w_2: 3 → 2.6
- b_2: 1 → 0.8

What Happened:

We took one step of gradient descent. The weights and biases have been adjusted to reduce the loss. If we repeat this process many times, the network will continue to improve its predictions.

Summary: How Neural Networks Learn

Recap of the key concepts and how they all fit together.

The Big Picture:

Neural networks learn by adjusting their weights and biases to minimize prediction errors. This process happens in three main phases: forward pass, backward pass, and weight updates. Repeat this cycle many times, and the network gets better and better at making predictions.

What We Covered:

1. **Forward Pass (Making Predictions)**
   - Data flows from input layer through hidden layers to output layer
   - Each neuron multiplies inputs by weights, adds bias, and applies activation function
   - Result: a prediction ($\hat{y}$)
2. **Calculate Loss (Measuring Error)**
   - Compare prediction to actual answer: Loss = $(\hat{y} - y)^2$
   - Loss tells us how wrong we were
   - Goal: minimize this loss
3. **Backward Pass (Finding Gradients)**
   - Use Chain Rule to compute how much each weight contributed to the loss
   - Work backwards from output to input
   - Result: gradients for every weight and bias
4. **Update Weights (Learning)**
   - Use Gradient Descent formula: New Weight = Old Weight - (learning rate × gradient)
   - Move weights in the direction that reduces loss
   - Small steps, repeated many times
5. **Repeat**
   - Do forward pass with new weights
   - Calculate new loss
   - Compute new gradients
   - Update weights again
   - Continue until loss stops decreasing (convergence)

Why This Matters:

This cycle is the foundation of all neural network training. Whether you're training a small network or a massive deep learning model, the same principles apply. The only difference is scale and complexity.

Key Takeaways:

- Weights are the "knobs" the network adjusts to learn
- Loss measures how wrong predictions are
- Gradients tell us which direction to turn the knobs
- Gradient descent is the algorithm that turns the knobs
- Backpropagation efficiently computes gradients using the Chain Rule
- Computational graphs organize operations so backpropagation can work

Core Concepts Covered:

This course explains how neural networks learn from data. It covers forward propagation, backpropagation, gradient descent, and weight updates. The calculus behind these concepts (power rule, chain rule, partial derivatives) and why computational graphs matter are also detailed. This is the core knowledge that powers modern AI.