The Perceptron Learning Algorithm: A Step-by-Step Guide

The perceptron, a foundational element of neural networks, operates on a straightforward principle: it takes multiple binary inputs and produces a single binary output. It's designed to classify data into one of two categories, making it a powerful tool for linear classification tasks. The learning process involves iteratively adjusting its internal parameters to minimize errors between its predictions and the actual target outputs.

1

Initialization

Weights and bias are set. This could be random small values or even zeros. This initial state is critical as it serves as the starting point for the learning process.

2

Weighted Sum Calculation

For each training sample, the perceptron calculates the weighted sum of its inputs, $z = \sum w_i x_i + b$. This sum represents the combined influence of all inputs on the neuron's decision.

3

Activation and Output

A step activation function is applied: the output is 1 if $z > 0$, otherwise 0. This binary decision is the perceptron's classification for the given input.

4

Weight Update Rule

Weights are updated using the rule $w_j \leftarrow w_j + \eta (target - output) x_j$. Here, η (eta) is the learning rate, controlling the step size of each update. This adjustment moves the decision boundary closer to correctly classifying the misclassified samples.

A remarkable property of the perceptron learning algorithm is its **guaranteed convergence** if the data is linearly separable. This means if a line (or hyperplane in higher dimensions) can perfectly separate the classes, the perceptron will eventually find it. This makes it robust for simple classification problems.

Algorithm: Perceptron Learning Algorithm

$P \leftarrow inputs \quad with \quad label \quad 1;$

$N \leftarrow inputs \quad with \quad label \quad 0;$

Initialize $\mathbf{w} = [w_1, w_2, \ldots, w_N]$ randomly;

while $!convergence$ **do**

  Pick random $\mathbf{x} \in P \cup N$;

  if $\mathbf{x} \in P \quad and \quad \sum_{i=0}^{n} w_i * x_i < 0$ **then**

    $\mathbf{w} = \mathbf{w} + \mathbf{x}$;

  end

  if $\mathbf{x} \in N \quad and \quad \sum_{i=0}^{n} w_i * x_i \geq 0$ **then**

  end

end

//the algorithm converges when all the inputs are classified correctly

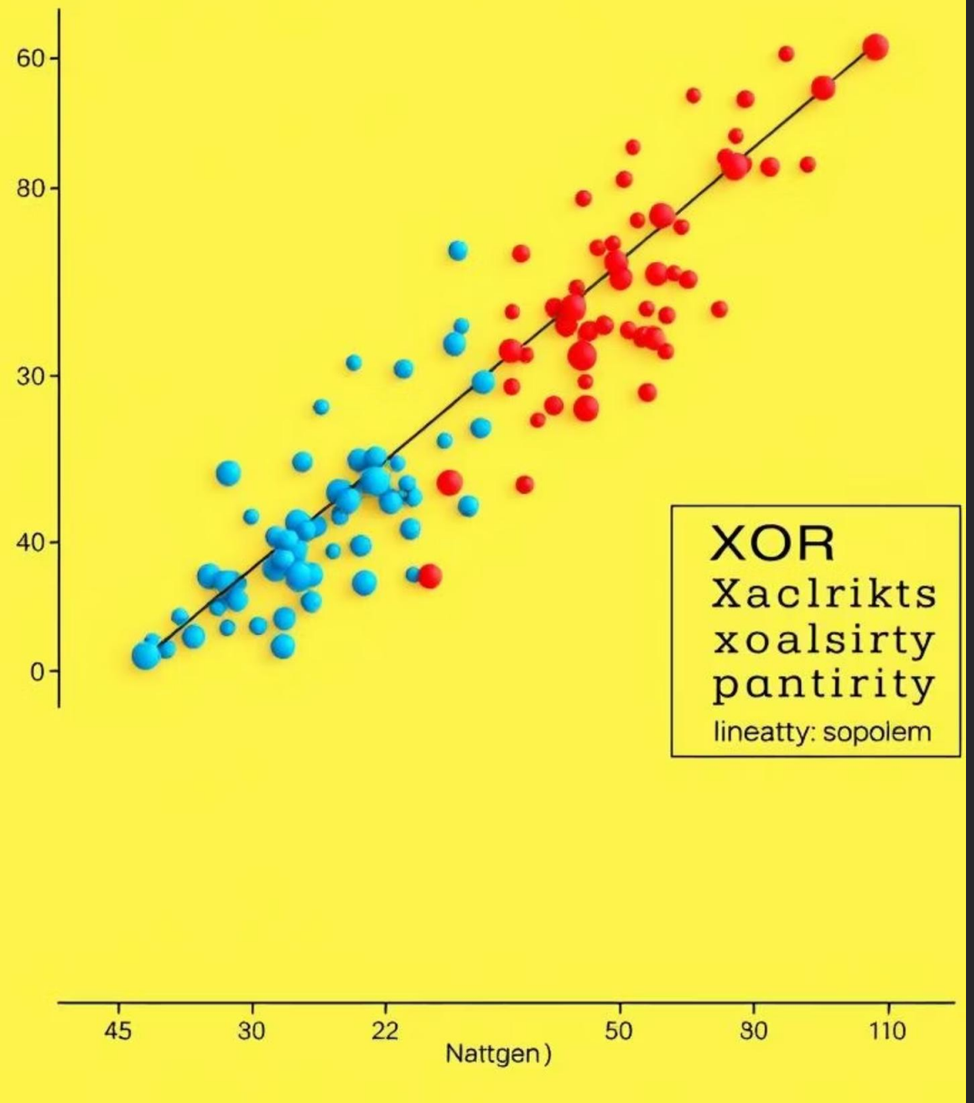
- Consider two vectors $\mathbf{w}$ and $\mathbf{x}$

$$\mathbf{w} = [w_0, w_1, w_2, ..., w_n]$$
$$\mathbf{x} = [1, x_1, x_2, ..., x_n]$$
$$\mathbf{w} \cdot \mathbf{x} = \mathbf{w^T x} = \sum_{i=0}^{n} w_i * x_i$$

- We can thus rewrite the perceptron rule as

$$y = 1 \quad if \quad \mathbf{w^T x} \geq 0$$
$$= 0 \quad if \quad \mathbf{w^T x} < 0$$

- We are interested in finding the line $\mathbf{w^T x} = 0$ which divides the input space into two halves
- Every point ($\mathbf{x}$) on this line satisfies the equation $\mathbf{w^T x} = 0$
- What can you tell about the angle (α) between $\mathbf{w}$ and any point ($\mathbf{x}$) which lies on this line ?
- The angle is 90° ($\because cos\alpha = \frac{w^T x}{||w||||x||} = 0$)
- Since the vector $\mathbf{w}$ is perpendicular to every point on the line it is actually perpendicular to the line itself



Representation Power of Single-Layer Perceptrons: The Linear Boundary

While the perceptron learning algorithm is powerful for its simplicity and guaranteed convergence under certain conditions, a single-layer perceptron inherently possesses a significant limitation: its representation power is restricted to linearly separable problems.

- **Linearly Separable Tasks:** A single-layer perceptron can perfectly classify data where a single straight line (or a hyperplane in higher dimensions) can divide the different categories. Classic examples include the logical **AND** and **OR** gates, where inputs can be easily separated into classes by a linear boundary.
- **The XOR Problem:** The inability to solve non-linear problems is best illustrated by the XOR (exclusive OR) gate. In an XOR problem, points are arranged in such a way that no single straight line can separate the "true" outputs from the "false" outputs. This was a critical challenge for early AI research, highlighting the limitations of single-layer architectures.
- **Decision Boundary:** Fundamentally, the output of a single-layer perceptron is determined by whether the weighted sum of inputs crosses a certain threshold, effectively creating a linear decision boundary. This hyperplane acts as the separator.

This limitation was a major roadblock in the early days of AI, leading to a period known as the "AI winter." However, it also served as a crucial motivation for researchers to explore more complex architectures, ultimately leading to the development of **multilayer networks** capable of modeling intricate, non-linear relationships.

Activation Functions: The Key to Non-Linearity

Activation functions are pivotal components within artificial neurons, determining whether a neuron should "fire" and pass its signal to the next layer in the network. They introduce non-linearity, allowing neural networks to learn complex patterns and model relationships that go beyond simple linear boundaries.

Step Function (Threshold)

Description: The simplest form, outputting a binary value (0 or 1) based on whether the input exceeds a certain threshold.

Use Case: Primarily used in early perceptrons for binary classification.

Limitation: Non-differentiable, meaning it cannot be used with gradient-based learning algorithms like backpropagation.

Sigmoid (Logistic)

Description: An S-shaped curve that squashes input values into a range between 0 and 1.

Use Case: Historically popular for output layers in binary classification, as its output can be interpreted as a probability.

Advantage: Differentiable, enabling gradient descent.

Tanh (Hyperbolic Tangent)

Description: Similar to sigmoid, but squashes inputs into a range between -1 and 1. It is zero-centered.

Use Case: Often preferred over sigmoid for hidden layers because its zero-centered output helps with faster convergence during training.

ReLU (Rectified Linear Unit)

Description: Outputs the input directly if positive, otherwise outputs zero. $f(x) = max(0, x)$.

Use Case: Widely used in hidden layers of deep neural networks due to its computational efficiency and ability to mitigate the vanishing gradient problem.

Leaky ReLU

Description: A variation of ReLU that allows a small, non-zero gradient (e.g., 0.01x) for negative inputs.

Use Case: Addresses the "dying ReLU" problem, where neurons can become inactive for negative inputs, preventing further learning.

The choice of activation function significantly impacts a neural network's ability to learn and its training efficiency. Modern deep learning predominantly uses ReLU and its variants due to their simplicity and effectiveness.

Why We Need Activation Functions

•Without them, a neural network would just be a big linear equation—no matter how many layers you add.

•They help the network capture **non-linear patterns** in data (e.g., recognizing faces, predicting stock prices, classifying images).

Types of Activation Functions

1. **Sigmoid**

•Formula: $f(x) = \frac{1}{1+e^{-x}}$

•Output range: **(0, 1)**

•Good for: probabilities

•Problem: suffers from **vanishing gradient** (very small changes in large inputs)

2. Tanh (Hyperbolic Tangent)

•Formula: $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

•Output range: **(-1, 1)**
•Zero-centered (better than sigmoid for many cases)
•Still suffers from vanishing gradient

3. ReLU (Rectified Linear Unit)

•Formula: f(x)=max(0,x)
•Output range: **[0, ∞)**
•Very popular, fast, avoids most vanishing gradient issues
•Problem: **Dead ReLU** (neurons stuck at zero output)

4. Leaky ReLU

•Formula: $f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{if } x < 0 \end{cases}$

•Fixes dead ReLU problem

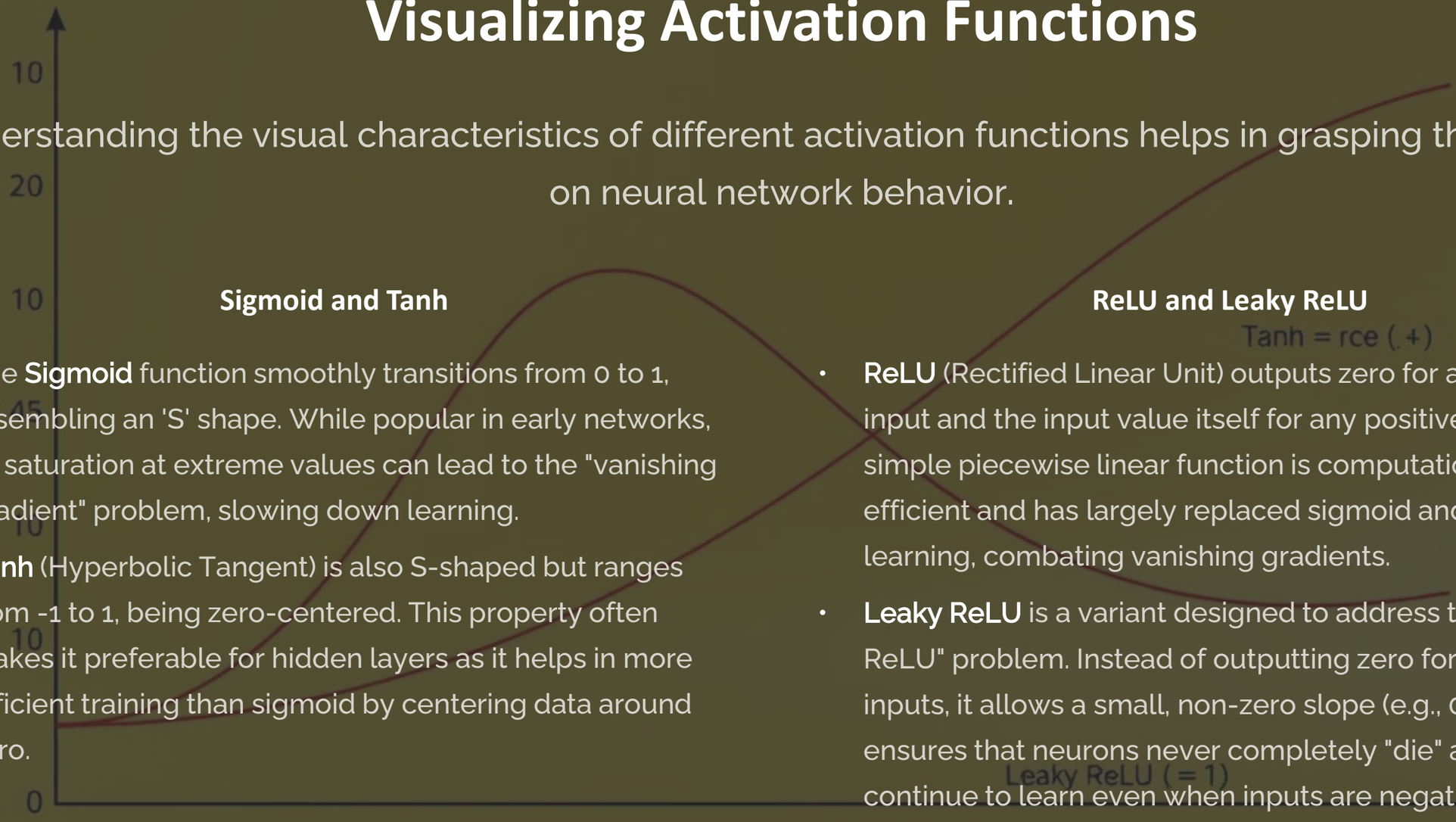
5. Softmax

•Turns outputs into **probability distribution** (sums to 1)
•Used in multi-class classification in the output layer

Function	Output Range	Use Case
Sigmoid	(0, 1)	Binary classification
Tanh	(-1, 1)	Zero-centered classification
ReLU	[0, ∞)	Hidden layers (fast)
Leaky ReLU	(-∞, ∞)	Fix dead ReLU
Softmax	(0, 1), sum=1	Multi-class classification

Visualizing Activation Functions

Understanding the visual characteristics of different activation functions helps in grasping their impact on neural network behavior.



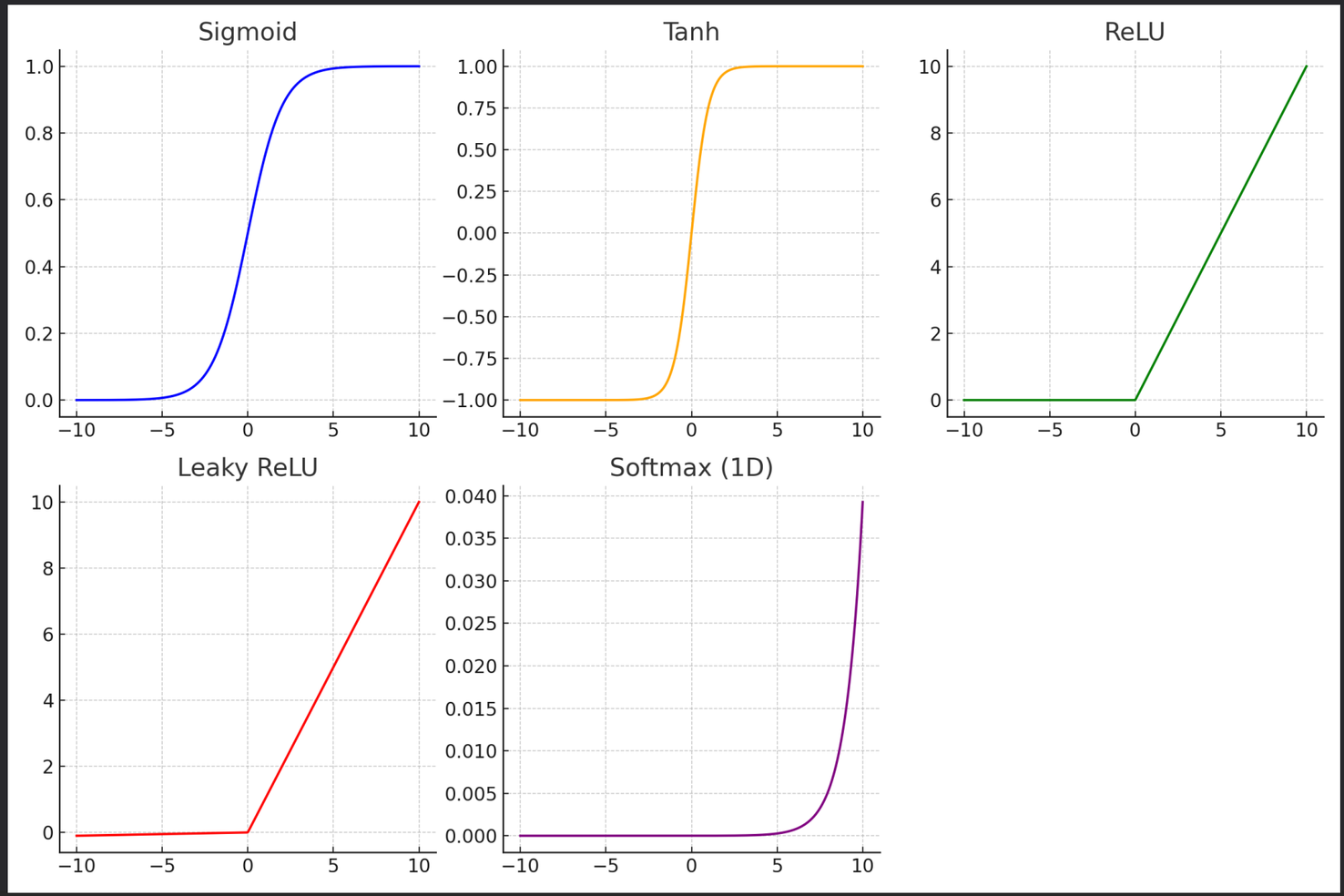
Sigmoid and Tanh

- The **Sigmoid** function smoothly transitions from 0 to 1, resembling an 'S' shape. While popular in early networks, its saturation at extreme values can lead to the "vanishing gradient" problem, slowing down learning.
- **Tanh** (Hyperbolic Tangent) is also S-shaped but ranges from -1 to 1, being zero-centered. This property often makes it preferable for hidden layers as it helps in more efficient training than sigmoid by centering data around zero.

ReLU and Leaky ReLU

- **ReLU** (Rectified Linear Unit) outputs zero for any negative input and the input value itself for any positive input. This simple piecewise linear function is computationally efficient and has largely replaced sigmoid and tanh in deep learning, combating vanishing gradients.
- **Leaky ReLU** is a variant designed to address the "dying ReLU" problem. Instead of outputting zero for negative inputs, it allows a small, non-zero slope (e.g., 0.01x). This ensures that neurons never completely "die" and can continue to learn even when inputs are negative, providing a more robust training process.

- **Sigmoid**: S-curve, squashes values between 0 and 1.
- **Tanh**: Similar to sigmoid but ranges from -1 to 1.
- **ReLU**: Zero for negatives, linear for positives.
- **Leaky ReLU**: Like ReLU but with a small slope for negatives.
- **Softmax (1D)**: Grows rapidly around the largest input value.



Sigmoid neuron

A **sigmoid neuron** is a type of artificial neuron used in neural networks where the activation function is the **sigmoid function**:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

How It Works

1. **Inputs** $(x_1, x_2, \ldots, x_n)$
2. **Weights** $(w_1, w_2, \ldots, w_n)$ and **bias** b
3. Compute the weighted sum:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

4. Pass z through the **sigmoid activation**:

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

5. Output y is between **0 and 1** (good for probability-based decisions).

Why Use Sigmoid in a Neuron?

- Smooth curve, differentiable (needed for backpropagation).
- Output interpretable as a probability.
- Historically popular in early neural networks.

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Example neuron
def sigmoid_neuron(x, weights, bias):
    z = np.dot(weights, x) + bias
    return sigmoid(z)

# Inputs
x = np.array([0.5, 0.8])   # two features
weights = np.array([0.4, 0.6])
bias = -0.2

output = sigmoid_neuron(x, weights, bias)
print("Neuron output:", output)
```

Representation Power of Multilayer Networks of Sigmoid Neurons: Unlocking Unlocking Complexity

The transition from single-layer perceptrons to **Multilayer Perceptrons (MLPs)**, especially those utilizing non-linear activation functions like Sigmoid (and later ReLU), marked a monumental leap in the capabilities of neural networks. This architectural shift enabled AI to address problems previously deemed intractable.

Stacked Layers

MLPs are characterized by at least one "hidden layer" between the input and output layers. Each layer consists of multiple neurons, with the output of neurons in one layer serving as the input for neurons in the subsequent layer. This hierarchical structure allows the network to learn progressively more abstract and complex features of the input data.

Universal Approximation Theorem

This is a cornerstone concept in neural network theory. It states that an MLP with a single hidden layer containing a finite number of neurons (and non-linear activation functions like Sigmoid) can approximate any continuous function to an arbitrary degree of accuracy. This mathematical proof underscores the immense power of MLPs to model highly intricate relationships.

Modeling Non-Linear Patterns

Unlike single-layer perceptrons that are confined to linear decision boundaries, hidden layers with non-linear activations allow MLPs to learn and create complex, curved, and discontinuous decision boundaries. This means they can solve the infamous XOR problem and handle virtually any non-linear pattern present in real-world data.

Real-World Applications

The enhanced representation power of MLPs opened the door to solving a vast array of complex classification and regression tasks. From image recognition and natural language processing to financial forecasting and medical diagnosis, MLPs laid the groundwork for many advancements in artificial intelligence that we see today.

In essence, the combination of multiple layers and non-linear activation functions transforms a simple perceptron into a universal function approximator, capable of understanding and extracting highly intricate patterns from data.

Representation Power in Real-World Scenarios

The ability of multilayer neural networks to learn and represent complex functions is what makes them indispensable for solving real-world challenges that are inherently non-linear and high-dimensional.

Image Recognition

Recognizing objects, faces, and scenes in images is a highly non-linear problem. A cat is not just a linear combination of pixels. Neural networks learn intricate features like edges, textures, and shapes across multiple layers to build a robust representation of what a cat looks like, regardless of pose or lighting.

Natural Language Processing

Understanding human language involves complex relationships between words, syntax, semantics, and context. Neural networks can learn hierarchical representations of sentences, paragraphs, and documents, enabling tasks like sentiment analysis, machine translation, and text summarization, which are far from linearly separable.

Medical Diagnosis

Diagnosing diseases from patient data (e.g., symptoms, lab results, medical images) often involves identifying subtle, non-obvious patterns. Neural networks can learn to detect these complex correlations, assisting clinicians in making more accurate predictions for conditions like cancer or specific syndromes.

Financial Market Prediction

Stock prices and market trends are influenced by a myriad of interconnected, non-linear factors. Neural networks can analyze vast amounts of historical data, news, and economic indicators to identify complex patterns that might signal future movements, a task impossible for linear models.

These examples demonstrate how neural networks, with their capacity to learn complex functions, have become the backbone of modern AI, transforming industries and solving problems once thought to be beyond algorithmic reach.

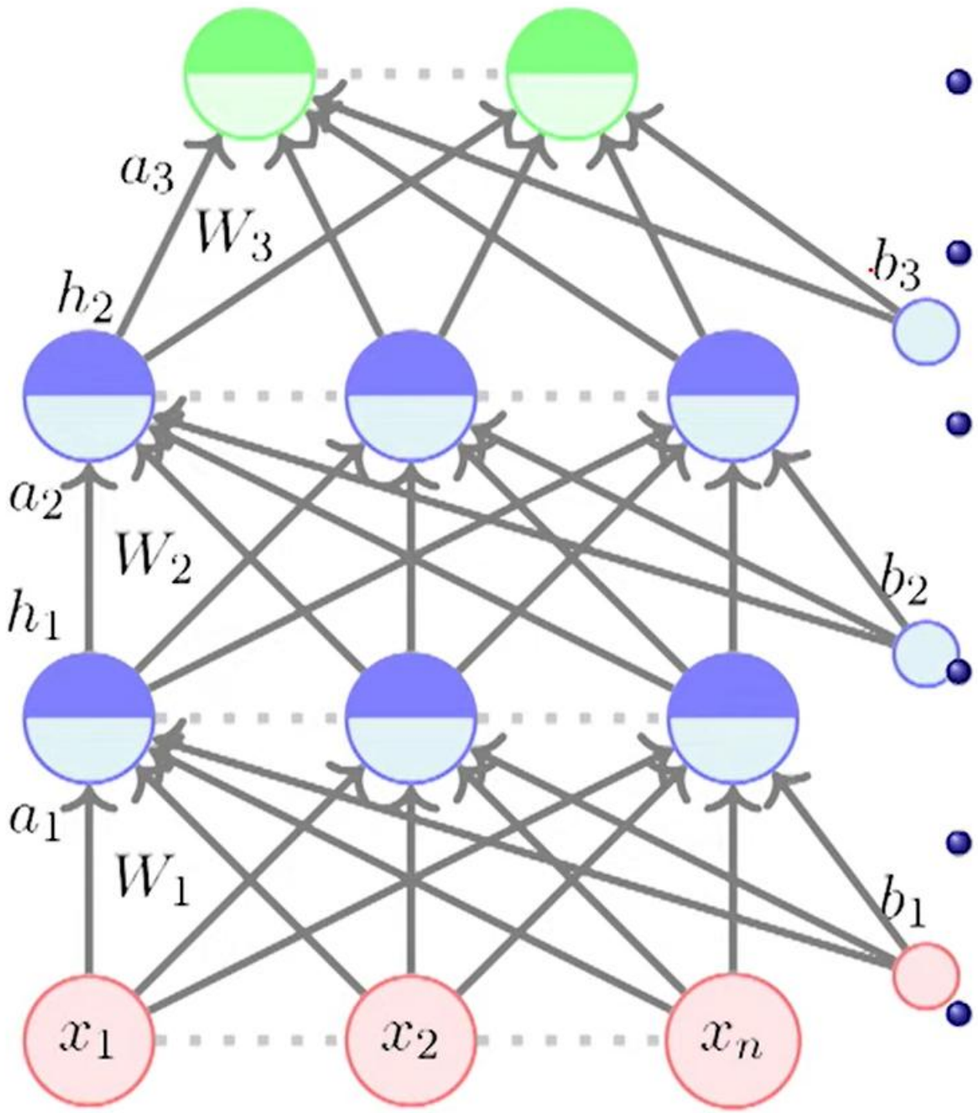
The Feedforward Neural Network Architecture

A feedforward neural network, often referred to as a Multilayer Perceptron (MLP), is the quintessential architecture for many machine learning tasks. Its defining characteristic is the unidirectional flow of information from the input layer, through one or more hidden layers, to the output layer, without any loops or cycles.

- **Input Layer:** This is where the raw data enters the network. Each neuron in the input layer corresponds to a feature in the input dataset. These neurons simply pass the input values to the next layer without performing any computations.
- **Hidden Layers:** These are the computational engines of the network. Each neuron in a hidden layer receives inputs from the previous layer, applies a weighted sum, and then passes the result through an activation function. The "hidden" aspect refers to the fact that these layers are not directly exposed to the external input or output. Deep learning networks are simply feedforward networks with many hidden layers.
- **Output Layer:** This layer produces the final result of the network's processing. The number of neurons in the output layer depends on the problem type: one neuron for binary classification/regression, multiple neurons for multi-class classification (e.g., one for each class), often coupled with a softmax activation for probabilities.
- **Connections (Weights):** Every connection between neurons in adjacent layers has an associated weight. These weights represent the strength or importance of that connection. During training, these weights are iteratively adjusted to minimize the network's prediction error.
- **Bias:** Each neuron (except input neurons) typically has an associated bias term. A bias acts like an intercept in a linear equation, allowing the activation function to be shifted left or right, which is crucial for the network to model a wider range of functions.

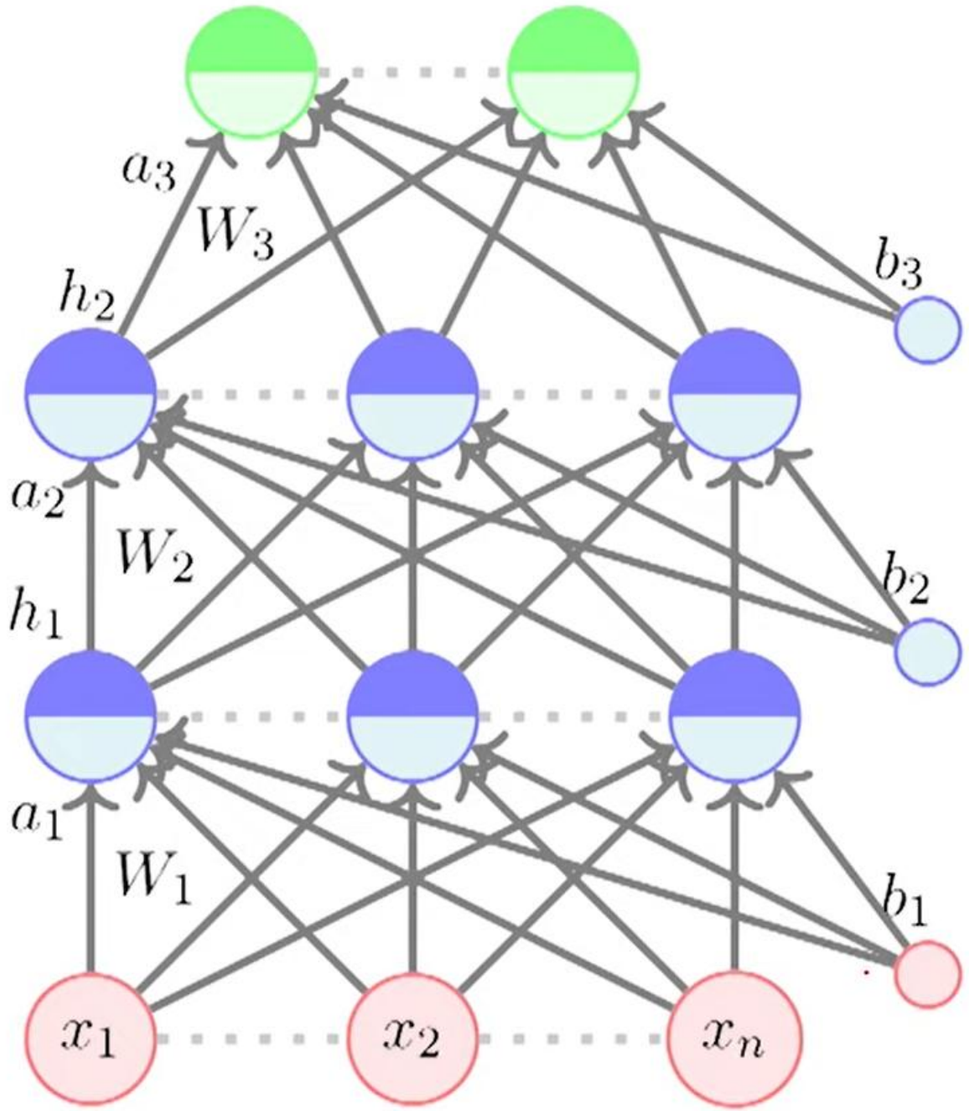
The feedforward nature ensures that information flows in one direction, making the network's behavior predictable and its training process (via backpropagation) computationally feasible.

$h_L = \hat{y} = f(x)$



- The input to the network is an **n**-dimensional vector
- The network contains $\mathbf{L-1}$ hidden layers (2, in this case) having **n** neurons each
- Finally, there is one output layer containing **k** neurons (say, corresponding to **k** classes)
- Each neuron in the hidden layer and output layer can be split into two parts : pre-activation and activation (a_i and h_i are vectors)
- The input layer can be called the 0-th layer and the output layer can be called the (L)-th layer
- $W_i \in \mathbb{R}^{n \times n}$ and $b_i \in \mathbb{R}^n$ are the weight and bias between layers $i-1$ and i $(0 < i < L)$
- $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$ are the weight and bias between the last hidden layer and the output layer ($L = 3$ in this case)

$h_L = \hat{y} = f(x)$



- The pre-activation at layer i is given by

$$a_i(x) = b_i + W_i h_{i-1}(x)$$

For example, $a_1 = b_1 + W_1 h_0$

$$\begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \end{bmatrix} = \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix} + \begin{bmatrix} W_{111} & W_{112} & W_{113} \\ W_{121} & W_{122} & W_{123} \\ W_{131} & W_{132} & W_{133} \end{bmatrix} \begin{bmatrix} h_{01} = x_1 \\ h_{02} = x_2 \\ h_{03} = x_3 \end{bmatrix}$$

$$= \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix} + \begin{bmatrix} W_{111}x_1 + W_{112}x_2 + W_{113}x_3 \\ W_{121}x_1 + W_{122}x_2 + W_{123}x_3 \\ W_{131}x_1 + W_{132}x_2 + W_{133}x_3 \end{bmatrix}$$

$$= \begin{bmatrix} \sum W_{11i}x_i + b_{11} \\ \sum W_{12i}x_i + b_{12} \\ \sum W_{13i}x_i + b_{13} \end{bmatrix}$$

Learning Parameters, Output, and Loss in a Feedforward Network

The process of training a feedforward neural network involves three core components that work in harmony: the learning parameters, the network's output generation, and the quantification of its errors through a loss function.

Learning Parameters

The true "knowledge" of a neural network is encapsulated in its **weights** and **biases**. Weights determine the strength of connections between neurons, while biases allow neurons to activate even with zero input. These are the adjustable parameters that the network "learns" during training, iteratively modifying them to improve performance.

Think of them as the dials and levers that get finely tuned to capture the underlying patterns in the data.

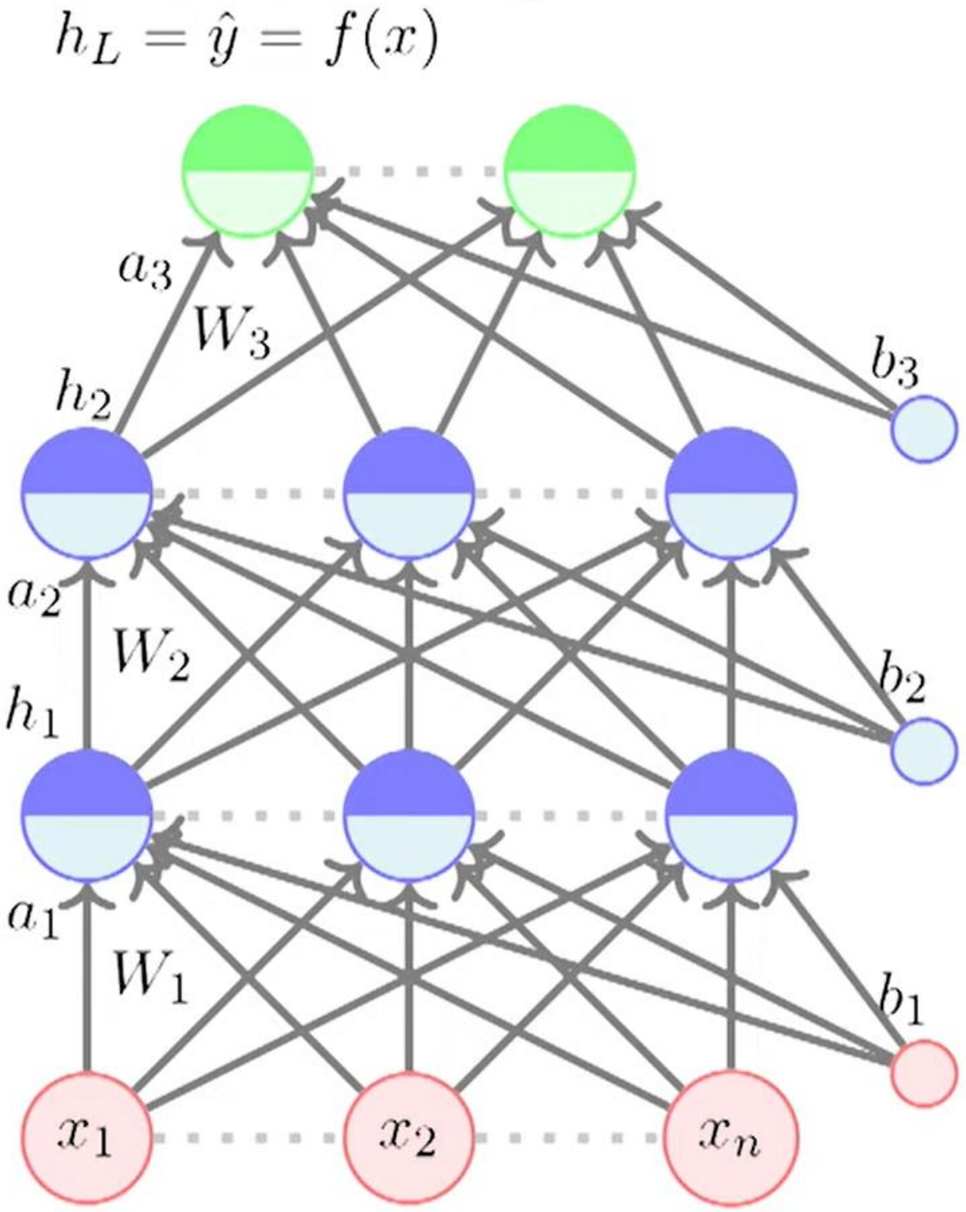
Network Output

During the **forward pass**, input data propagates through the network, from the input layer, through hidden layers, and finally to the output layer. At each neuron, the weighted sum of inputs is computed and passed through an activation function, producing the neuron's output. The final output of the network is the prediction generated by the output layer, which could be a classification label, a numerical value, or a probability distribution.

Loss Function

The **loss function** (or cost function) quantifies the discrepancy between the network's predicted output and the actual target output. It's a single numerical value that represents how "wrong" the network's prediction is. Common loss functions include Mean Squared Error (for regression) and Cross-Entropy Loss (for classification). The goal of the training process is to minimize this loss value, indicating that the network's predictions are becoming more accurate.

Through the iterative process of forward propagation (generating output) and backpropagation (updating parameters based on loss), feedforward neural networks learn to perform complex tasks, adapting their internal representations to model intricate data relationships.

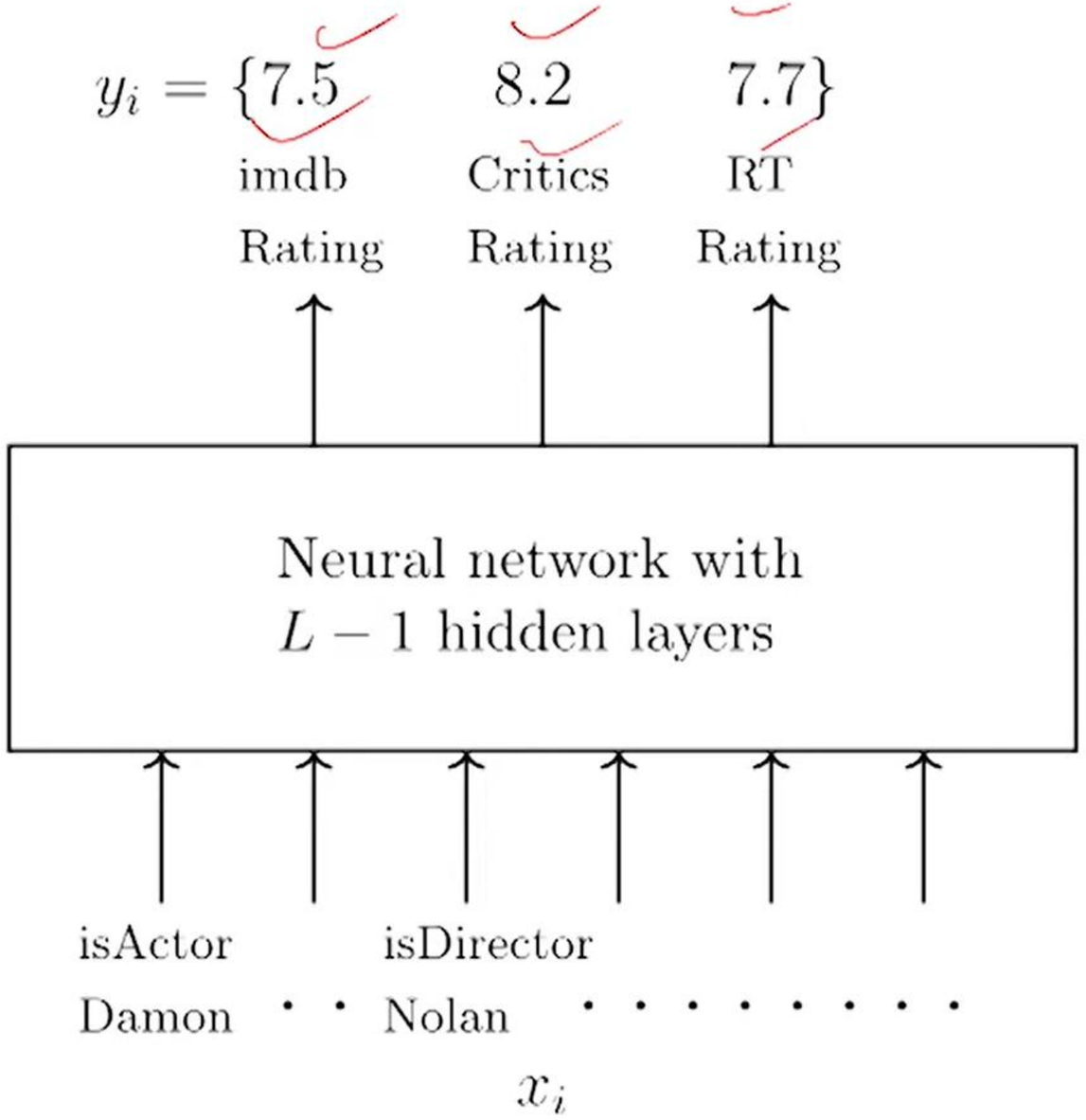


- Recall our gradient descent algorithm
- We can write it more concisely as

Algorithm: gradient_descent()

$t \leftarrow 0$;
$max\_iterations \leftarrow 1000$;
$Initialize \quad \theta_0 = [W_1^0, ..., W_L^0, b_1^0, ..., b_L^0]$;
while $t{+}{+} < max\_iterations$ **do**
  $\theta_{t+1} \leftarrow \theta_t - \eta \nabla \theta_t$;
end

- where $\nabla \theta_l = \left[\frac{\partial \mathscr{L}(\theta)}{\partial w_t}, \frac{\partial \mathscr{L}(\theta)}{\partial b_t}\right]^T$
- Now, in this feedforward neural network, instead of $\theta = [w, b]$ we have $\theta = W_1, W_2, .., W_L, b_1, b_2, .., b_L$
- We can still use the same algorithm for learning the parameters of our model



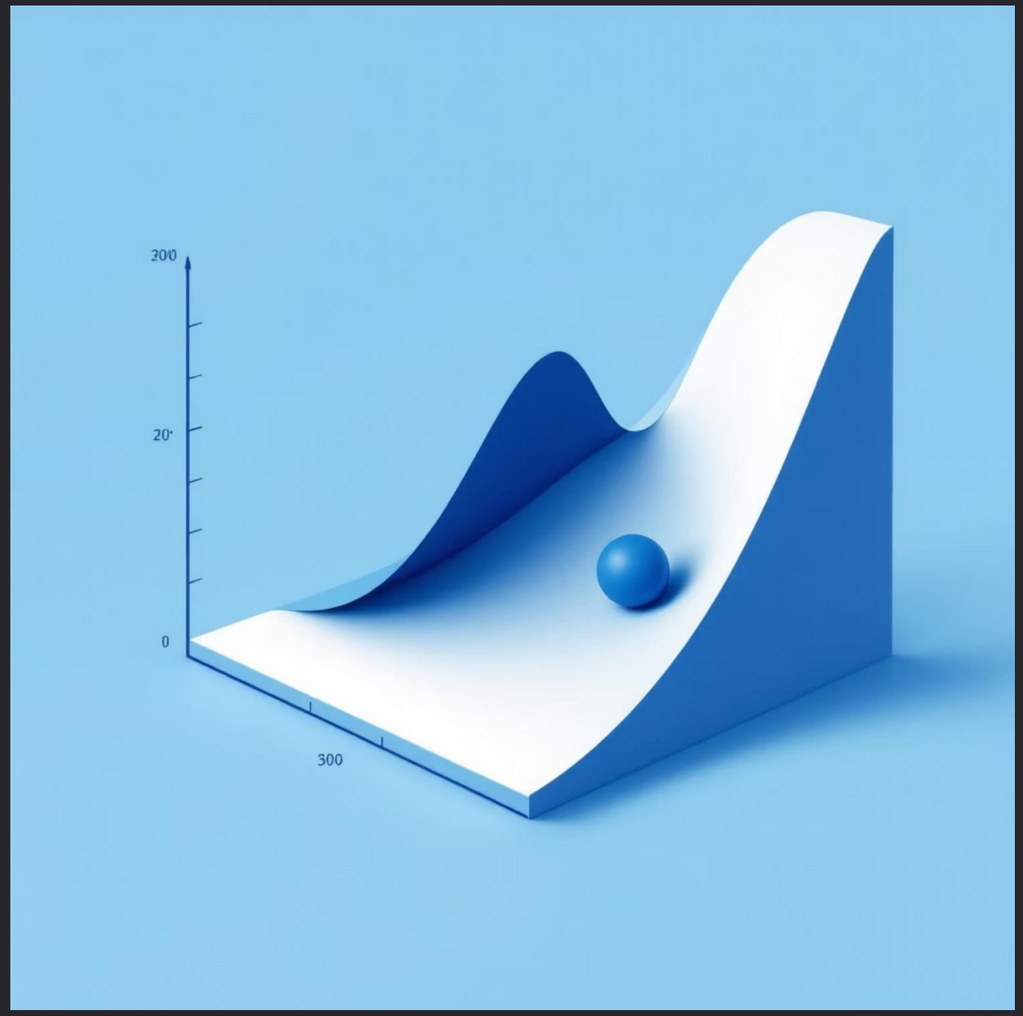
- The choice of loss function depends on the problem at hand
- We will illustrate this with the help of two examples
- Consider our movie example again but this time we are interested in predicting ratings
- Here $y_i \in \mathbb{R}^3$
- The loss function should capture how much $\hat{y}_i$ deviates from y_i
- If $y_i \in \mathbb{R}^n$ then the squared error loss can capture this deviation

$$\mathscr{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{3} (\hat{y}_{ij} - y_{ij})^2$$

What is Gradient Descent?

Gradient Descent is a fundamental optimization algorithm that powers the learning process in neural networks. Its primary goal is to iteratively adjust the model's parameters (weights and biases) to minimize the loss (cost) function.

Think of it like a ball rolling down a hill. The algorithm guides the parameters "downhill" along the steepest slope of the loss function's surface, aiming to find the lowest point, which represents the minimum error in the model's predictions.



Key takeaway: Gradient Descent is the engine that drives parameter updates to reduce prediction errors.

Gradient Descent vs. Backpropagation: Roles Explained

Backpropagation

The "accountant" of the network, calculating the exact error contribution (gradient) for each weight.

- Determines how much each weight is responsible for the overall prediction error.

Gradient Descent

The "optimizer" that uses these calculations to actually adjust the weights, minimizing error.

- Applies the gradients to update weights, iteratively moving towards a better model.

Together, they form the core learning loop, enabling neural networks to continuously improve their predictive accuracy.

Training Machine Learning Models

Neural networks are trained using Gradient Descent (or its variants) in combination with backpropagation. Backpropagation computes the gradients of the loss function with respect to each parameter (weights and biases) in the network by applying the chain rule. The process involves:

Forward Propagation: Computes the output for a given input by passing data through the layers.
Backward Propagation: Uses the chain rule to calculate gradients of the loss with respect to each parameter (weights and biases) across all layers.
Gradients are then used by Gradient Descent to update the parameters layer-by-layer, moving toward minimizing the loss function.

Minimizing the Cost Function

The algorithm minimizes a cost function, which quantifies the error or loss of the model's predictions compared to the true labels for:

1. Linear Regression

Gradient descent minimizes the Mean Squared Error (MSE) which serves as the loss function to find the best-fit line. Gradient Descent is used to iteratively update the weights (coefficients) and bias by computing the gradient of the MSE with respect to these parameters.

The algorithm computes the gradient of the MSE with respect to the weights and biases.

It updates the weights (w) and bias (b) using the formula:

•Calculating the gradient of the log-loss with respect to the weights.

•Updating weights and biases iteratively to maximize the likelihood of the correct classification:

$w=w-\alpha\cdot\partial J(w,b)\partial w, b=b-\alpha\cdot\partial J(w,b)\partial b$ $w=w-\alpha\cdot\partial w\partial J(w,b), b=b-\alpha\cdot\partial b\partial J(w,b)$

The formula is the **parameter update rule for gradient descent**, which adjusts the weights w and biases b to minimize a cost function. This process iteratively adjusts the line's slope and intercept to minimize the error.

Logistic Regression

In logistic regression, gradient descent minimizes the Log Loss (Cross-Entropy Loss) to optimize the decision boundary for binary classification. Since the output is probabilistic (between 0 and 1), the sigmoid function is applied. The process involves:

Calculating the gradient of the log-loss with respect to the weights.

Updating weights and biases iteratively to maximize the likelihood of the correct classification:

$$w = w - \alpha \cdot \frac{\partial J(w)}{\partial w}$$

This adjustment shifts the decision boundary to separate classes more effectively.

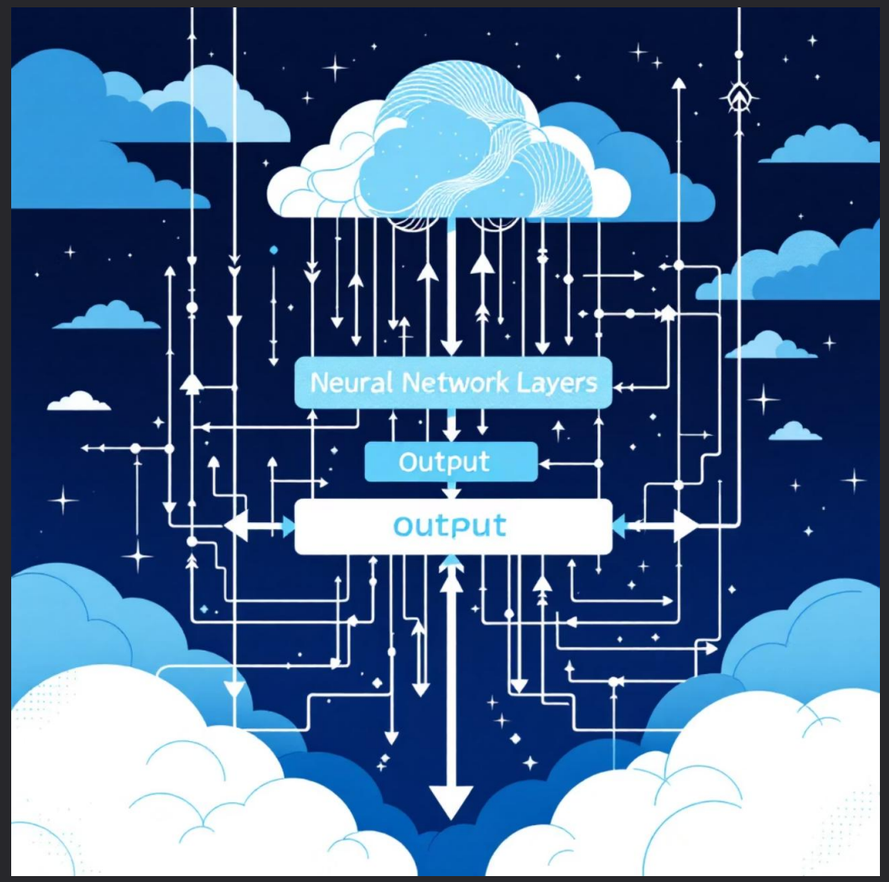
Support Vector Machines (SVMs)

For SVMs, gradient descent optimizes the hinge loss, which ensures a maximum-margin hyperplane. The algorithm:

Calculates gradients for the hinge loss and the regularization term (if used, such as L2 regularization).

Updates the weights to maximize the margin between classes while minimizing misclassification penalties with same formula provided above.

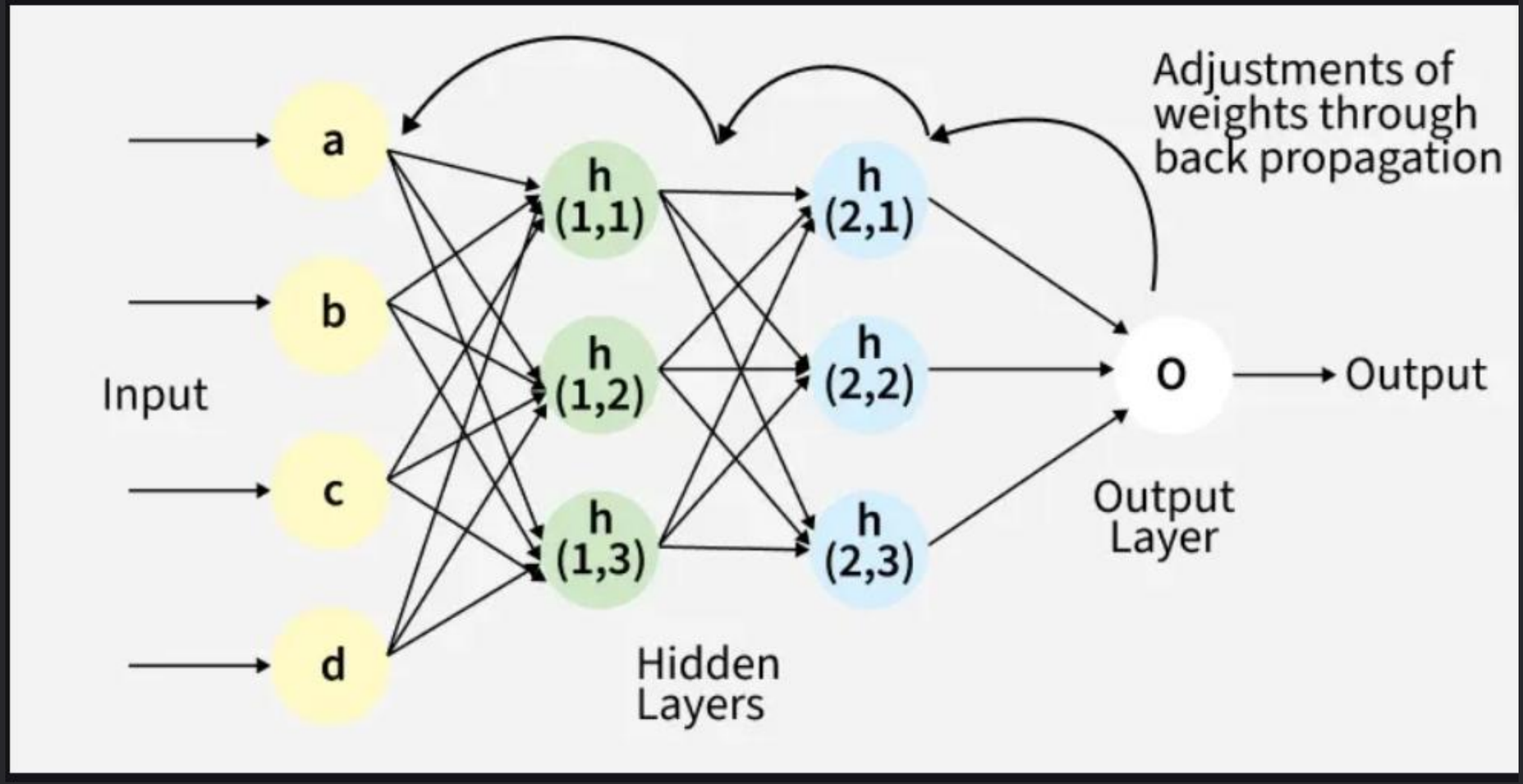
What is Backpropagation?



Error signals flowing backward through the network.

Backpropagation, short for "backward propagation of errors," is a crucial algorithm that efficiently calculates the gradients of the loss function with respect to every weight and bias in a neural network. These gradients are essential for Gradient Descent to know how to update the parameters.

It works by applying the chain rule of calculus to propagate the prediction error backward from the output layer through the hidden layers all the way to the input layer. This allows the network to understand precisely how much each individual weight and bias contributed to the final error, enabling precise adjustments for improved accuracy. It's the engine that powers learning in deep neural networks.



Back Propagation plays a critical role in how neural networks improve over time. Here's why:

1.Efficient Weight Update: It computes the gradient of the loss function with respect to each weight using the chain rule making it possible to update weights efficiently.

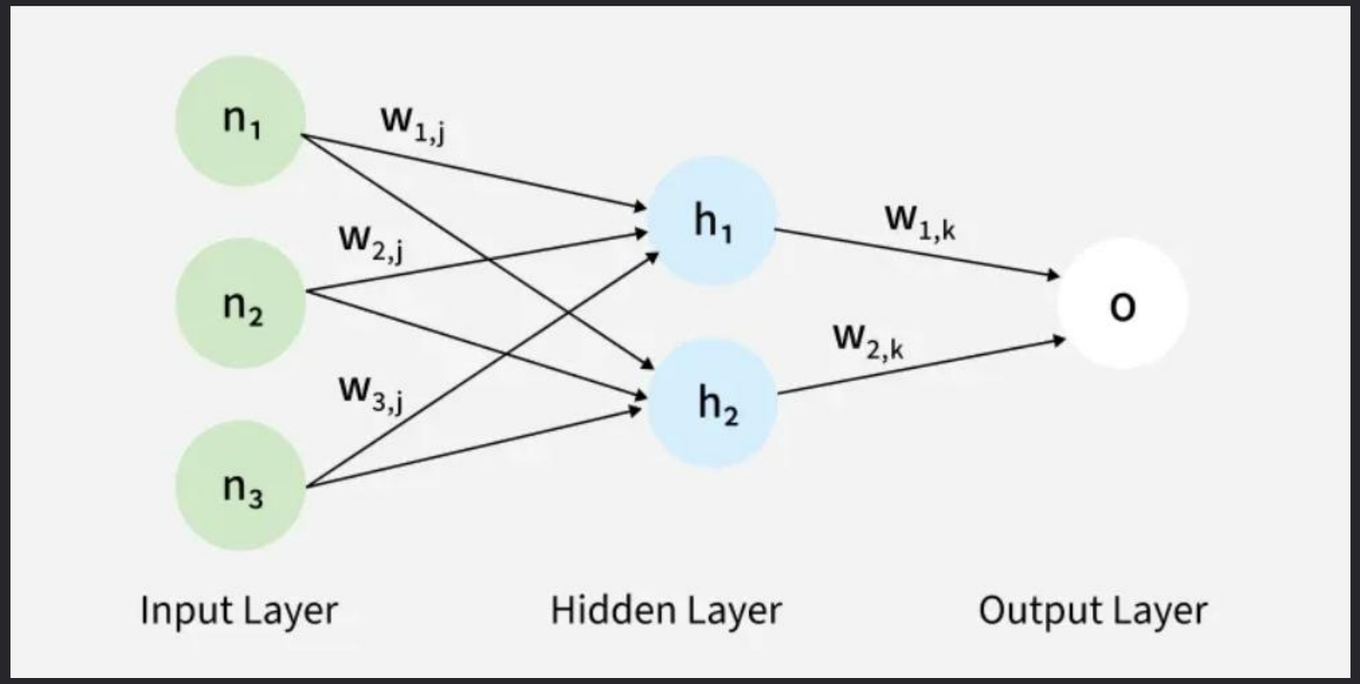
2.Scalability: The Back Propagation algorithm scales well to networks with multiple layers and complex architectures making deep learning feasible.

3.Automated Learning: With Back Propagation the learning process becomes automated and the model can adjust itself to optimize its performance.

Working of Back Propagation Algorithm
The Back Propagation algorithm involves two main steps: the Forward Pass and the Backward Pass.

1. Forward Pass Work
In forward pass the input data is fed into the input layer. These inputs combined with their respective weights are passed to hidden layers. For example in a network with two hidden layers (h1 and h2) the output from h1 serves as the input to h2. Before applying an activation function, a bias is added to the weighted inputs.

Each hidden layer computes the weighted sum (`a`) of the inputs then applies an activation function like ReLU (Rectified Linear Unit) to obtain the output (`o`). The output is passed to the next layer where an activation function such as softmax converts the weighted outputs into probabilities for classification.



Backward Pass

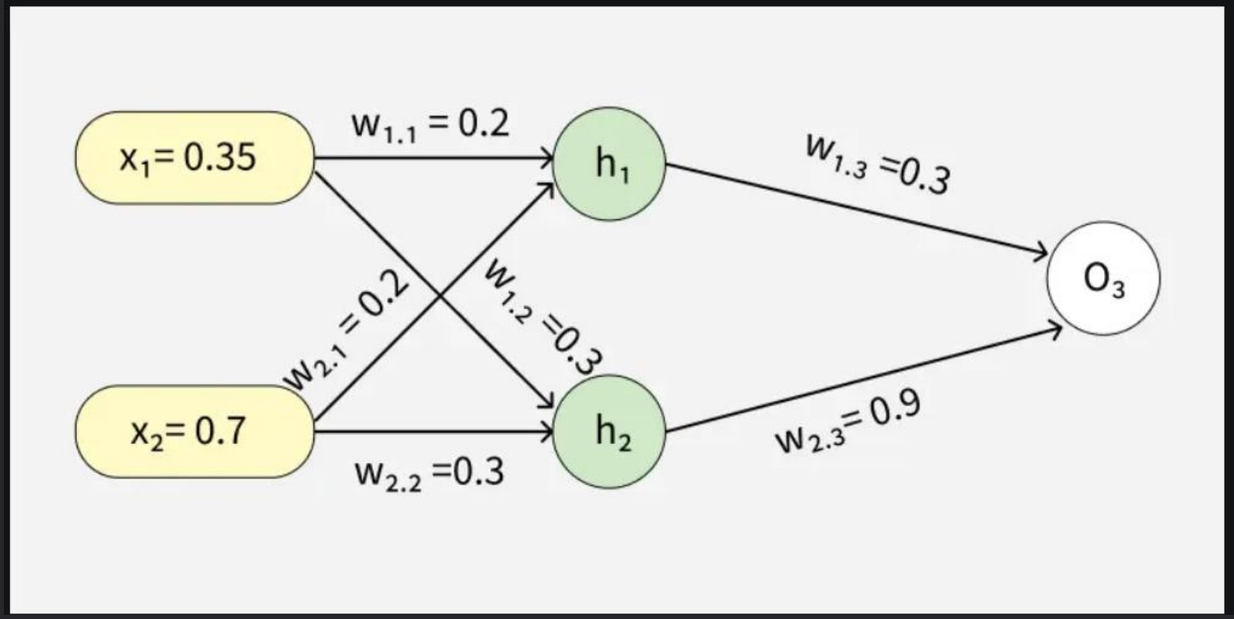
In the backward pass the error (the difference between the predicted and actual output) is propagated back through the network to adjust the weights and biases. One common method for error calculation is the Mean Squared Error (MSE) given by:

$$MSE = (\text{Predicted Output} - \text{Actual Output})^2$$

Once the error is calculated the network adjusts weights using gradients which are computed with the chain rule. These gradients indicate how much each weight and bias should be adjusted to minimize the error in the next iteration. The backward pass continues layer by layer ensuring that the network learns and improves its performance. The activation function through its derivative plays a crucial role in computing these gradients during Back Propagation.

Example of Back Propagation in Machine Learning

Let’s walk through an example of Back Propagation in machine learning. Assume the neurons use the sigmoid activation function for the forward and backward pass. The target output is 0.5 and the learning rate is 1.



Forward Propagation

1. Initial Calculation

The weighted sum at each node is calculated using:

$$aj = \sum(wi, j * xi)$$

Where,

- a_j is the weighted sum of all the inputs and weights at each node
- $w_{i,j}$ represents the weights between the i^{th}input and the j^{th} neuron
- x_i represents the value of the i^{th} input

o (output): After applying the activation function to a, we get the output of the neuron:

$$o_j = activation\ function(a_j)$$

3. Computing Outputs

At h1 node

$$
\begin{aligned}
a_1 &= (w_{1,1}x_1) + (w_{2,1}x_2) \\
&= (0.2 * 0.35) + (0.2 * 0.7) \\
&= 0.21
\end{aligned}
$$

Once we calculated the a_1 value, we can now proceed to find the y_3 value:

$$y_j = F(a_j) = \frac{1}{1+e^{-a_1}}$$

$$y_3 = F(0.21) = \frac{1}{1+e^{-0.21}}$$

$$y_3 = 0.56$$

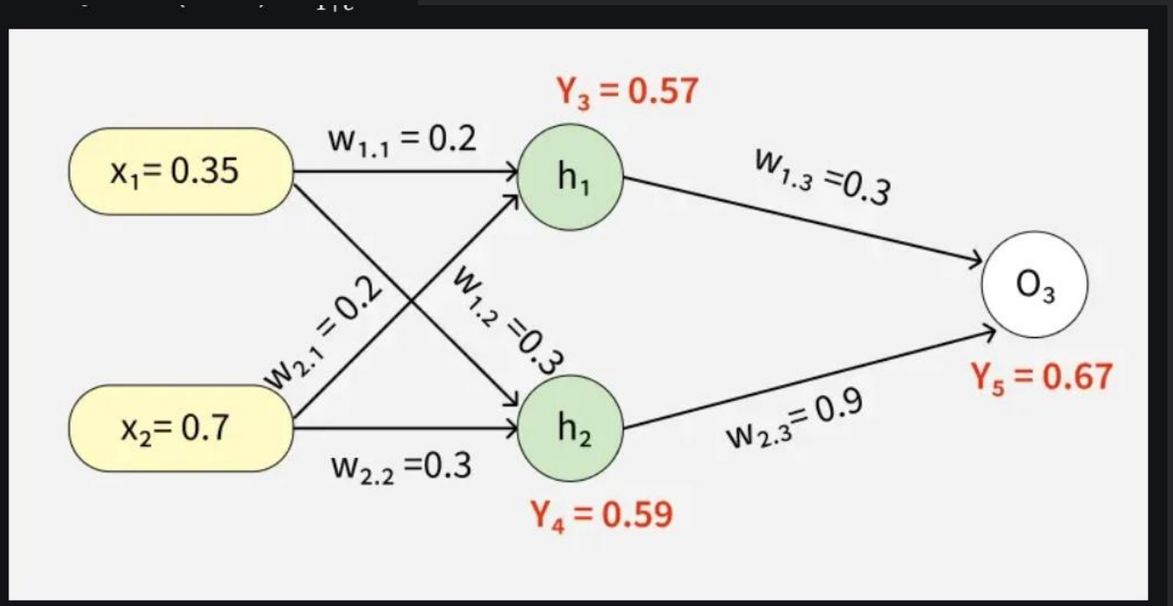
Similarly find the values of y_4 at $\mathbf{h_2}$ and y_5 at O_3

$$a_2 = (w_{1,2} * x_1) + (w_{2,2} * x_2) = (0.3 * 0.35) + (0.3 * 0.7) = 0.315$$

$$y_4 = F(0.315) = \frac{1}{1+e^{-0.315}}$$

$$a3 = (w_{1,3} * y_3) + (w_{2,3} * y_4) = (0.3 * 0.57) + (0.9 * 0.59) = 0.702$$

$$y_5 = F(0.702) = \frac{1}{1+e^{-0.702}} = 0.67$$



4. Error Calculation

Our actual output is 0.5 but we obtained 0.67. To calculate the error we can use the below formula:

$$Error_j = y_{target} - y_5$$

$$\Rightarrow 0.5 - 0.67 = -0.17$$

Using this error value we will be backpropagating.

Back Propagation

1. Calculating Gradients

The change in each weight is calculated as:

$$\Delta w_{ij} = \eta \times \delta_j \times O_j$$

Where:

- δ_j is the error term for each unit,
- η is the learning rate.

2. Output Unit Error

For O3:

$$\delta_5 = y_5(1 - y_5)(y_{target} - y_5)$$
$$= 0.67(1 - 0.67)(-0.17) = -0.0376$$

3. Hidden Unit Error

For h1:

$$\delta_3 = y_3(1 - y_3)(w_{1,3} \times \delta_5)$$

$$= 0.56(1 - 0.56)(0.3 \times -0.0376) = -0.0027$$

For h2:

$$\delta_4 = y_4(1 - y_4)(w_{2,3} \times \delta_5)$$

$$= 0.59(1 - 0.59)(0.9 \times -0.0376) = -0.0819$$

4. Weight Updates

For the weights from hidden to output layer:

$$\Delta w_{2,3} = 1 \times (-0.0376) \times 0.59 = -0.022184$$

New weight:

$$w_{2,3}(\text{new}) = -0.022184 + 0.9 = 0.877816$$

For weights from input to hidden layer:

$$\Delta w_{1,1} = 1 \times (-0.0027) \times 0.35 = 0.000945$$

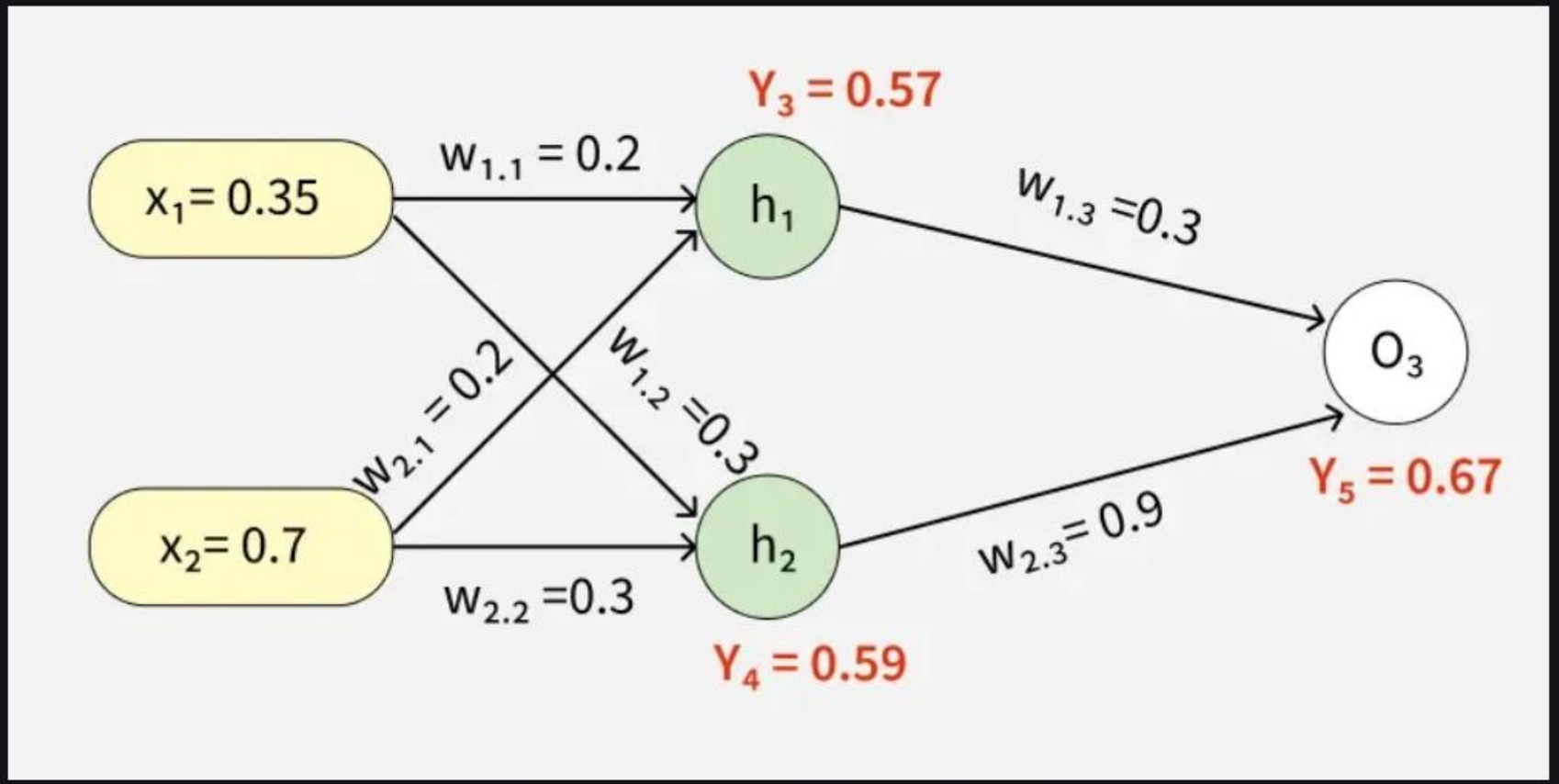
New weight:

$$w_{1,1}(\text{new}) = 0.000945 + 0.2 = 0.200945$$

Similarly other weights are updated:

- $w_{1,2}(\text{new}) = 0.273225$
- $w_{1,3}(\text{new}) = 0.086615$
- $w_{2,1}(\text{new}) = 0.269445$
- $w_{2,2}(\text{new}) = 0.18534$

The updated weights are illustrated below



Through backward pass the weights are updated

After updating the weights the forward pass is repeated yielding:

- $y_3 = 0.57$
- $y_4 = 0.56$
- $y_5 = 0.61$

The Role of Backpropagation

While Gradient Descent tells us which way to go to minimize loss, Backpropagation provides the crucial how. It's an algorithm dedicated to efficiently computing the partial derivatives of the loss function with respect to every single weight and bias in the neural network. Without these derivatives, Gradient Descent wouldn't know how much or in what direction to adjust each parameter.

Error Propagation

Backpropagation starts at the output layer, where the error is directly measurable, and then propagates these error gradients backward through the network's layers.

Chain Rule Application

It masterfully employs the chain rule of calculus to break down the complex derivative calculation into manageable layer-by-layer computations.

Enabling Deep Learning

This process makes it feasible for Gradient Descent to update weights correctly in even very deep, multilayer neural networks, making deep learning possible.

In essence, Backpropagation is the engine that provides the necessary "feedback" to the network, telling each parameter how much it contributed to the overall error.

Advantages of Back Propagation for Neural Network Training
The key benefits of using the Back Propagation algorithm are:

1.Ease of Implementation: Back Propagation is beginner-friendly requiring no prior neural network knowledge and simplifies programming by adjusting weights with error derivatives.

2.Simplicity and Flexibility: Its straightforward design suits a range of tasks from basic feedforward to complex convolutional or recurrent networks.

3.Efficiency: Back Propagation accelerates learning by directly updating weights based on error especially in deep networks.

4.Generalization: It helps models generalize well to new data improving prediction accuracy on unseen examples.

5.Scalability: The algorithm scales efficiently with larger datasets and more complex networks making it ideal for large-scale tasks.

Challenges with Back Propagation
While Back Propagation is useful it does face some challenges:

1.Vanishing Gradient Problem: In deep networks the gradients can become very small during Back Propagation making it difficult for the network to learn. This is common when using activation functions like sigmoid or tanh.

2.Exploding Gradients: The gradients can also become excessively large causing the network to diverge during training.

3.Overfitting: If the network is too complex it might memorize the training data instead of learning general patterns.

Gradient Descent vs. Backpropagation: Roles Explained

Backpropagation

The "accountant" of the network, calculating the exact error contribution (gradient) for each weight.

- Determines how much each weight is responsible for the overall prediction error.

Gradient Descent

The "optimizer" that uses these calculations to actually adjust the weights, minimizing error.

- Applies the gradients to update weights, iteratively moving towards a better model.

Together, they form the core learning loop, enabling neural networks to continuously improve their predictive accuracy.

Applying the Chain Rule in Neural Networks

Neural networks are fundamentally compositions of functions. Each layer applies a transformation, and the output of one layer becomes the input for the next. This functional composition is precisely where the chain rule becomes indispensable for calculating gradients during backpropagation.

> ### Chain Rule Analogy
>
> If you want to find out how a change in 'A' affects 'C', but 'C' depends on 'B', and 'B' depends on 'A', the chain rule allows you to break this down. You first find how 'C' changes with 'B', and then how 'B' changes with 'A', and multiply those rates together to get the overall effect of 'A' on 'C'.

In a neural network, this translates to calculating how much the final loss changes with respect to a weight in an early layer, by multiplying the sensitivities of all intermediate layers.

$$\frac{dy}{dx} = \frac{dy}{dg} \times \frac{dg}{dx} \text{ (where } y = f(g(x)))$$

This formula demonstrates that the derivative of a composite function is the product of the derivatives of its individual functions. Backpropagation systematically applies this principle across every neuron and connection, from the output layer all the way back to the input.

The beauty of backpropagation lies in its ability to automate this complex differentiation across hundreds or even thousands of layers and millions of parameters, enabling efficient learning in deep architectures.

Computational Graphs: Visualizing Backpropagation

To truly grasp backpropagation, it's helpful to visualize a neural network as a computational graph. This graph illustrates the flow of data and operations within the network.

Nodes & Edges

Each node in the graph represents either a variable (like inputs, weights, biases) or an operation (like addition, multiplication, activation functions). Edges represent the dependencies and flow of data between these nodes.

Forward Pass

During the forward pass, we traverse the graph from input to output, computing the activations and predictions of the network layer by layer.

Backward Pass

The backward pass is where backpropagation happens. We traverse the graph in reverse, starting from the loss and computing gradients at each node by applying the chain rule.

This visual representation simplifies understanding how gradients are systematically calculated and propagated, ensuring every parameter receives its update instruction.

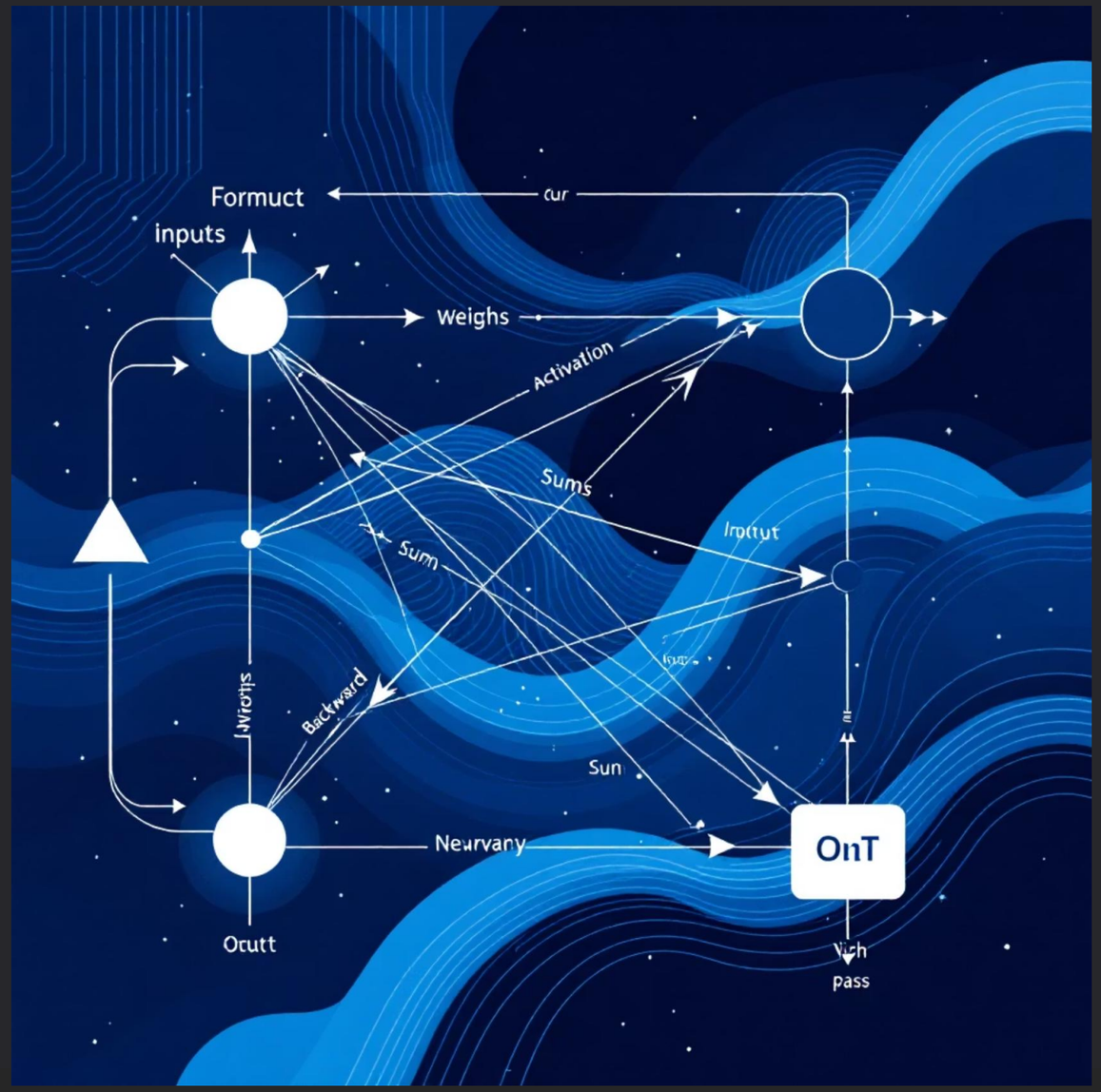
Visualizing the Computational Graph & Derivatives

A neural network can be conceptualized as a computational graph. In this graph:

- Nodes represent operations (like addition, multiplication, activation functions) or variables (inputs, weights, biases, neuron activations).
- Edges represent the flow of data and dependencies between these operations and variables.

During the forward pass, input data flows from left to right, computing activations layer by layer until the final output and loss are determined.

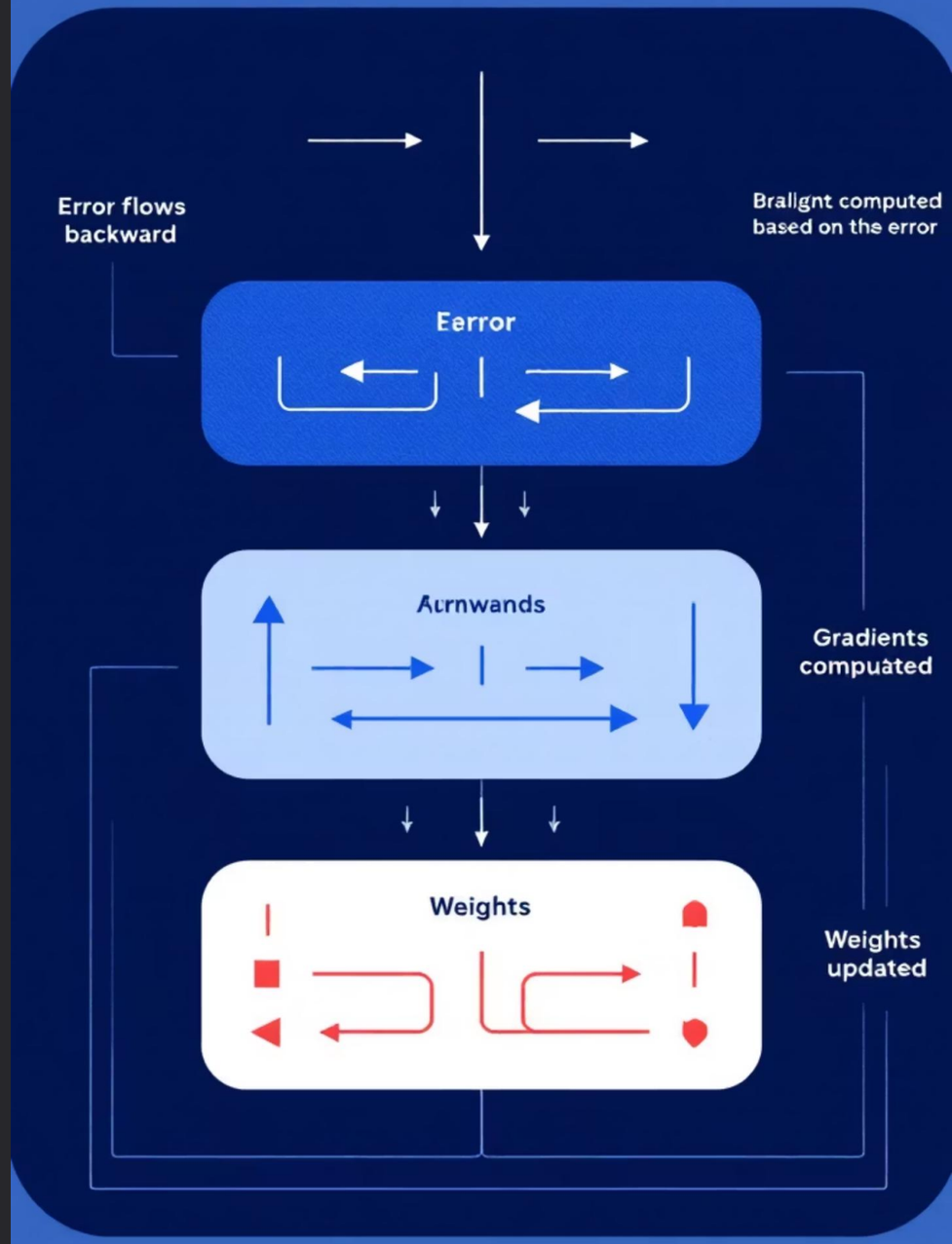
During the backward pass (backpropagation), error gradients flow backward along these same edges, from the output layer towards the input layer. Each node computes its local gradient and then multiplies it by the incoming gradient from the subsequent node (thanks to the chain rule). This efficient flow allows for the calculation of all gradients needed for optimization.



Forward and Backward Pass in a Computational Graph.

Visualizing Backpropagation & Gradient Descent

The cyclical process of learning: forward prediction, backward error correction, and iterative weight refinement.



Example: Partial Derivative w.r.t a Weight in a Single Neuron

Let's walk through a simplified example to understand how the partial derivative of the loss with respect to a single weight is calculated. Consider a basic neuron with one input (x), one weight (w), and a bias (b).

The Neuron's Process:

- Linear combination: $z = wx + b$ (input multiplied by weight, plus bias)
- Activation: $y = \sigma(z)$ (applying a sigmoid activation function to z)
- Loss: $L = \frac{1}{2}(y - t)^2$ (mean squared error, where t is the target)

To update the weight w, we need to find $\frac{\partial L}{\partial w}$.
Using the chain rule, we decompose this into:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \times \frac{\partial y}{\partial z} \times \frac{\partial z}{\partial w}$$

Calculating Each Component:

- $\frac{\partial L}{\partial y} = (y - t)$ (how much the loss changes with the output)
- $\frac{\partial y}{\partial z} = \sigma'(z)$ (the derivative of the activation function, often pre-calculated)
- $\frac{\partial z}{\partial w} = x$ (how much the linear combination changes with the weight)

Combining Them:

$$\frac{\partial L}{\partial w} = (y - t) \times \sigma'(z) \times x$$

This final formula elegantly shows how the error signal ($(y-t)$), the sensitivity of the activation function ($\sigma'(z)$), and the input value (x) all contribute to determining the adjustment needed for weight w. This process scales up for all weights in a network.