Introduction to Neural Networks

What is a Perceptron?

- A simple model of an artificial neuron.
- Developed in the 1950s/60s by Frank Rosenblatt, inspired by McCulloch and Pitts.
- It's a device that makes decisions by weighing up evidence.

What is a Perceptron?

- A simple model of an artificial neuron.
- Developed in the 1950s/60s by Frank Rosenblatt, inspired by McCulloch and Pitts.
- It's a device that makes decisions by weighing up evidence.
- **Core components:**
  - Takes several **binary inputs** $(x_1, x_2, \dots)$.
  - Produces a **single binary output** (0 or 1).

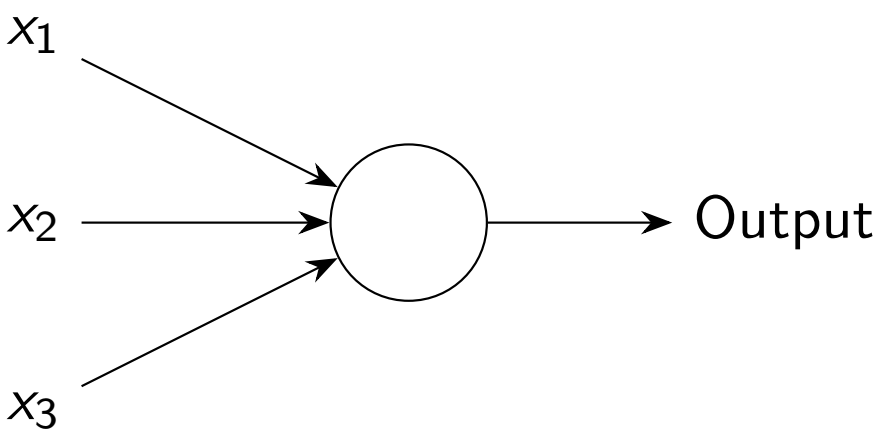


Figure: A perceptron with three inputs (x_1, x_2, x_3) and one output.

How a Perceptron Works

The Perceptron Rule

- Each input x_j has a corresponding **weight**, w_j.
- Weights are real numbers expressing the importance of each input.
- The neuron computes a **weighted sum**: $\sum_j w_j x_j$.
- The output (0 or 1) is determined by comparing this sum to a **threshold** value.

How a Perceptron Works

The Perceptron Rule

- Each input x_j has a corresponding **weight**, w_j.
- Weights are real numbers expressing the importance of each input.
- The neuron computes a **weighted sum**: $\sum_j w_j x_j$.
- The output (0 or 1) is determined by comparing this sum to a **threshold** value.

The Rule

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \textbf{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \textbf{ threshold} \end{cases}$$

Simplification: The Role of Bias

The "threshold" rule is a bit cumbersome. We can simplify it with two notational changes:

1. Write the weighted sum as a dot product: $\sum_j w_j x_j = \mathbf{w} \cdot \mathbf{x}$

Simplification: The Role of Bias

The "threshold" rule is a bit cumbersome. We can simplify it with two notational changes:

1. Write the weighted sum as a dot product: $\sum_j w_j x_j = \mathbf{w} \cdot \mathbf{x}$
2. Move the threshold to the other side and define **bias** as $b = -\text{threshold}$.

Simplification: The Role of Bias

The "threshold" rule is a bit cumbersome. We can simplify it with two notational changes:

1. Write the weighted sum as a dot product: $\sum_j w_j x_j = \mathbf{w} \cdot \mathbf{x}$
2. Move the threshold to the other side and define **bias** as $b = -\text{threshold}$.

The Rule (with Bias)

The perceptron rule becomes:

$$\text{output} = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

Simplification: The Role of Bias

The "threshold" rule is a bit cumbersome. We can simplify it with two notational changes:

1. Write the weighted sum as a dot product: $\sum_j w_j x_j = \mathbf{w} \cdot \mathbf{x}$
2. Move the threshold to the other side and define **bias** as $b = -\text{threshold}$.

The Rule (with Bias)

The perceptron rule becomes:

$$\text{output} = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

- The **bias** can be thought of as a measure of how easy it is for the perceptron to "fire" (output a 1).
- A large positive bias makes it easy to output 1.
- A large negative bias makes it difficult to output 1.

Definition: The NAND (Not AND) Gate

Inputs		**AND**	**NAND (Q)**
A	**B**	$(A \cdot B)$	$(\overline{A \cdot B})$
0	0	0	**1**
0	1	0	**1**
1	0	0	**1**
1	1	1	**0**

A **universal gate** is a logic gate that can be used, by itself, to construct all other fundamental logic gates.
The fundamental building blocks of digital logic are:**NOT**, **AND** and **OR**.
It is possible to prove the NAND gate is universal.

Example: Perceptrons as Logic Gates

Perceptrons can model elementary logical functions like AND, OR, and NAND.

A NAND Gate

Let's build a perceptron with two inputs, each with a weight of -2, and an overall bias of 3.

- **Inputs (0, 0):** $(-2 \times 0) + (-2 \times 0) + 3 = 3 \implies$ **Output 1**
- **Inputs (0, 1):** $(-2 \times 0) + (-2 \times 1) + 3 = 1 \implies$ **Output 1**
- **Inputs (1, 0):** $(-2 \times 1) + (-2 \times 0) + 3 = 1 \implies$ **Output 1**
- **Inputs (1, 1):** $(-2 \times 1) + (-2 \times 1) + 3 = -1 \implies$ **Output 0**

This is exactly the behavior of a NAND gate!

A network of perceptrons (e.g., NAND gates) can, in principle, compute any logical function.

The Limitation of Perceptrons

The Problem with Learning

We want to build a network that can **learn**. Learning involves making small, gradual changes to the weights and biases to slowly improve performance.

The Limitation of Perceptrons

The Problem with Learning

We want to build a network that can **learn**. Learning involves making small, gradual changes to the weights and biases to slowly improve performance.

- In a network of perceptrons, a small change in a weight or bias can cause the output of a neuron to **completely flip** (e.g., from 0 to 1).

The Limitation of Perceptrons

The Problem with Learning

We want to build a network that can **learn**. Learning involves making small, gradual changes to the weights and biases to slowly improve performance.

- In a network of perceptrons, a small change in a weight or bias can cause the output of a neuron to **completely flip** (e.g., from 0 to 1).
- This is a "step function" behavior.
- This "jump" can then cause a cascade of changes throughout the network in a very complex, hard-to-control way.
- This makes it extremely difficult to *gradually* modify weights to get closer to a desired behavior.

We need a "smoother" neuron.

What is a Sigmoid Neuron?

To overcome this problem, we introduce **sigmoid neurons**.

- **Similarities to Perceptrons:**
  - They have inputs $(x_1, x_2, \dots)$.
  - They have weights $(w_1, w_2, \dots)$ and a bias (b).
  - They still compute the weighted sum: $z = \mathbf{w} \cdot \mathbf{x} + b$.

What is a Sigmoid Neuron?

To overcome this problem, we introduce **sigmoid neurons**.

- **Similarities to Perceptrons:**
  - They have inputs $(x_1, x_2, \dots)$.
  - They have weights $(w_1, w_2, \dots)$ and a bias (b).
  - They still compute the weighted sum: $z = \mathbf{w} \cdot \mathbf{x} + b$.
- **Key Differences:**
  - Inputs are not restricted to 0 or 1; they can be **any value between 0 and 1**.
  - The output is **not** 0 or 1. It is a continuous value between 0 and 1, computed by the **sigmoid function**, $\sigma(z)$.

The Sigmoid Function

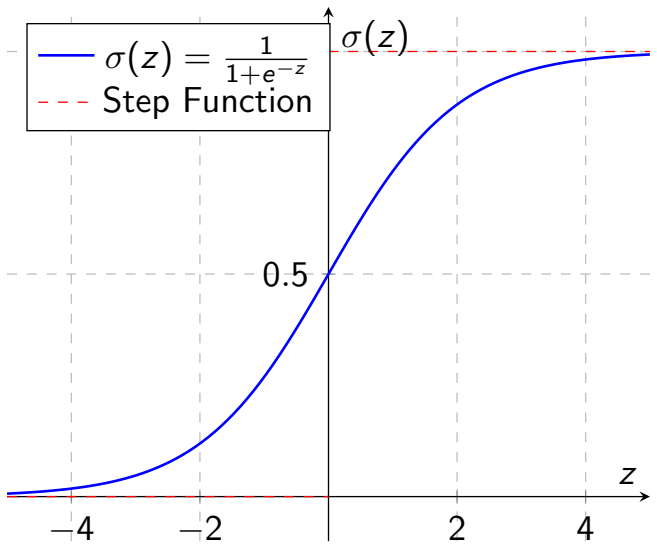
The sigmoid function (also called the logistic function) is defined as:

Sigmoid Function $\sigma(z)$

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

where $z = \mathbf{w} \cdot \mathbf{x} + b$.

- If z is large and positive, $e^{-z} \approx 0$, so $\sigma(z) \approx 1$.
- If z is large and negative, $e^{-z} \to \infty$, so $\sigma(z) \approx 0$.



The Benefit of Smoothness

The crucial property of the sigmoid function is its **smoothness**.

- Unlike the perceptron's "step", the sigmoid is a continuous function.
- This means a small change in a weight (Δw_j) or a small change in the bias (Δb) will produce only a **small change in the output** (Δoutput).

The Benefit of Smoothness

The crucial property of the sigmoid function is its **smoothness**.

- Unlike the perceptron's "step", the sigmoid is a continuous function.
- This means a small change in a weight (Δw_j) or a small change in the bias (Δb) will produce only a **small change in the output** (Δoutput).
- This change can be calculated using calculus (partial derivatives):

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b$$

The Benefit of Smoothness

The crucial property of the sigmoid function is its **smoothness**.

- Unlike the perceptron's "step", the sigmoid is a continuous function.
- This means a small change in a weight (Δw_j) or a small change in the bias (Δb) will produce only a **small change in the output** (Δoutput).
- This change can be calculated using calculus (partial derivatives):

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b$$

Why this is critical

This "small change in $\rightarrow$ small change out" property is what allows a network of sigmoid neurons to **learn**. It makes gradient-based learning algorithms (like **gradient descent**) possible.

Network Layers I

Neural networks are typically organized into **layers**.

- **Input Layer:**
  - The neurons that receive the initial data (e.g., pixel values of an image).

Network Layers I

Neural networks are typically organized into **layers**.

- **Input Layer:**
  - The neurons that receive the initial data (e.g., pixel values of an image).
- **Output Layer:**
  - The final layer of neurons that produces the network's result (e.g., a classification).

Network Layers I

Neural networks are typically organized into **layers**.

- **Input Layer:**
  - The neurons that receive the initial data (e.g., pixel values of an image).
- **Output Layer:**
  - The final layer of neurons that produces the network's result (e.g., a classification).
- **Hidden Layer(s):**
  - Any layers between the input and output layers.
  - They are "hidden" because their values are not directly observed as inputs or outputs.
  - They allow the network to learn more complex, abstract features.

Network Layers II

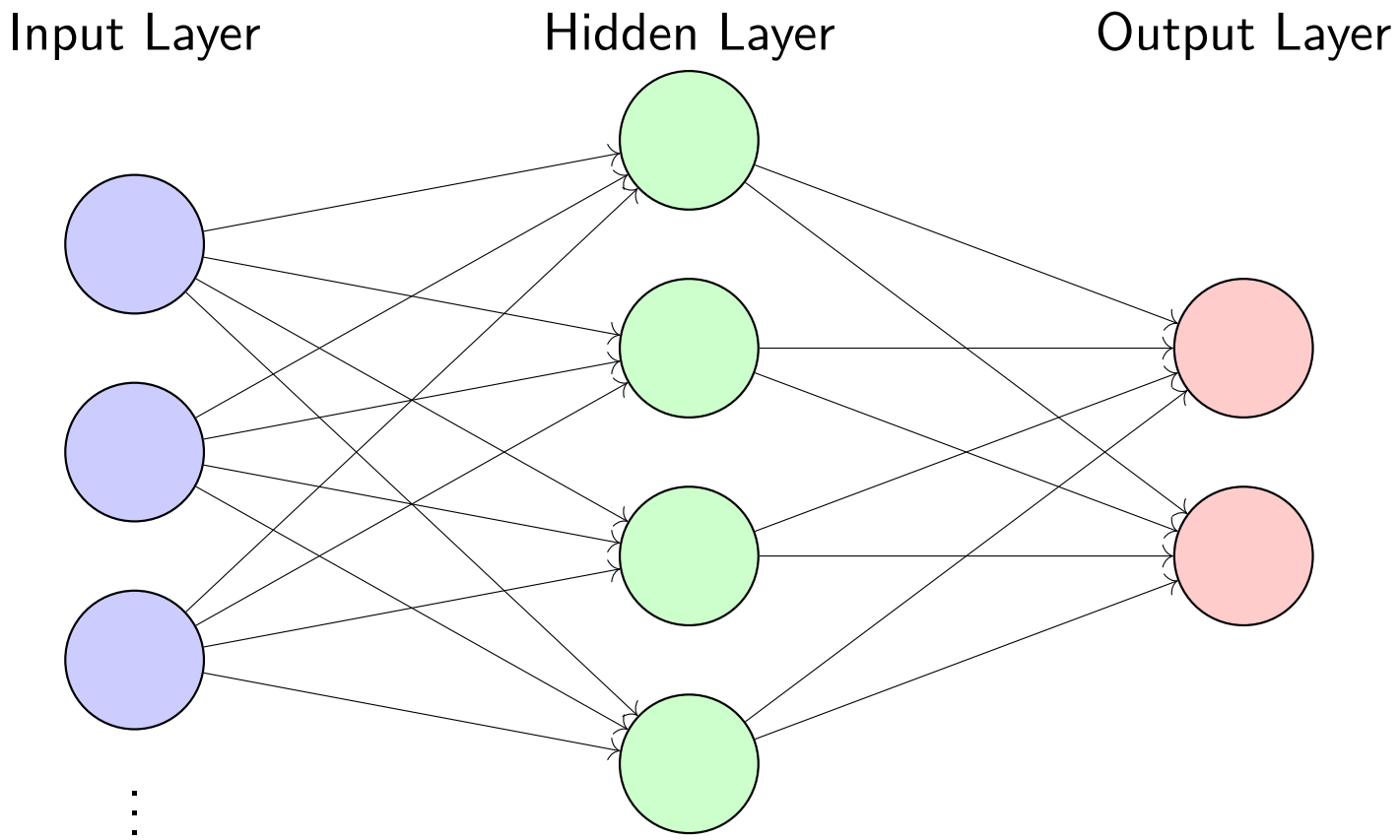


Figure: A simple feedforward network with one input layer, one hidden layer, and one output layer.

Feedforward Networks

- The simplest and most common network architecture.
- The name "feedforward" describes how information flows through the network.

Feedforward Networks

- The simplest and most common network architecture.
- The name "feedforward" describes how information flows through the network.
- Information moves in only **one direction**:
  - From the input layer...
  - ...through any hidden layers...
  - ...to the output layer.

Feedforward Networks

- The simplest and most common network architecture.
- The name "feedforward" describes how information flows through the network.
- Information moves in only **one direction**:
  - From the input layer...
  - ...through any hidden layers...
  - ...to the output layer.
- There are **no loops** or connections that feed information backward. The output of one layer is the input to the next.

Feedforward Networks

- The simplest and most common network architecture.
- The name "feedforward" describes how information flows through the network.
- Information moves in only **one direction**:
  - From the input layer...
  - ...through any hidden layers...
  - ...to the output layer.
- There are **no loops** or connections that feed information backward. The output of one layer is the input to the next.

Contrast: Recurrent Networks

This is different from **recurrent neural networks (RNNs)**, which *do* have feedback loops. RNNs are useful for sequential data (like text or time series), but we are focusing on feedforward networks.

Take home messages

- **Perceptrons** are the simple (but "jumpy") historical model.
- **Sigmoid Neurons** are a "smooth" version that enables learning via gradient descent.
- **Architecture** (like feedforward networks) describes how these neurons are stacked in layers (input, hidden, output) to process information.

These three concepts form the fundamental building blocks for understanding how neural networks compute and learn.

Matrix-Based Approach

Our goal is to compute the network's output in a fast, vectorized way.

Notation

- w^l_{jk}: Weight for the connection from the k^{th} neuron in layer $l-1$ to the j^{th} neuron in layer l.
- b^l_j: Bias of the j^{th} neuron in layer l.
- a^l_j: Activation of the j^{th} neuron in layer l.

Matrix-Based Approach

Our goal is to compute the network's output in a fast, vectorized way.

Notation

- w_{jk}^l: Weight for the connection from the k^{th} neuron in layer $l-1$ to the j^{th} neuron in layer l.
- b_j^l: Bias of the j^{th} neuron in layer l.
- a_j^l: Activation of the j^{th} neuron in layer l.

The activation of a single neuron is:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

where σ is the activation function (e.g., sigmoid).

Vectorizing the Output

We can rewrite the equation in a compact matrix form.

Definitions

- W^l: The **weight matrix** for layer l. Its entries are w^l_{jk}.
- $\mathbf{b}^l$: The **bias vector** for layer l.
- $\mathbf{a}^l$: The **activation vector** for layer l.

Vectorizing the Output

We can rewrite the equation in a compact matrix form.

Definitions

- W^l: The **weight matrix** for layer l. Its entries are w^l_{jk}.
- $\mathbf{b}^l$: The **bias vector** for layer l.
- $\mathbf{a}^l$: The **activation vector** for layer l.

First, we define the **weighted input** to layer l:

$$\mathbf{z}^l \equiv W^l \mathbf{a}^{l-1} + \mathbf{b}^l$$

The components are $z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$.

Vectorizing the Output

We can rewrite the equation in a compact matrix form.

Definitions

- W^l: The **weight matrix** for layer l. Its entries are w^l_{jk}.
- $\mathbf{b}^l$: The **bias vector** for layer l.
- $\mathbf{a}^l$: The **activation vector** for layer l.

First, we define the **weighted input** to layer l:

$$\mathbf{z}^l \equiv W^l \mathbf{a}^{l-1} + \mathbf{b}^l$$

The components are $z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$.
The activation vector for the layer is then simply:

$$\mathbf{a}^l = \sigma(\mathbf{z}^l)$$

where σ is applied **element-wise** to the vector $\mathbf{z}^l$.

Architecture of a Network: again...

This matrix equation $\mathbf{a}^l = \sigma(W^l \mathbf{a}^{l-1} + \mathbf{b}^l)$ represents the feedforward pass for one layer.

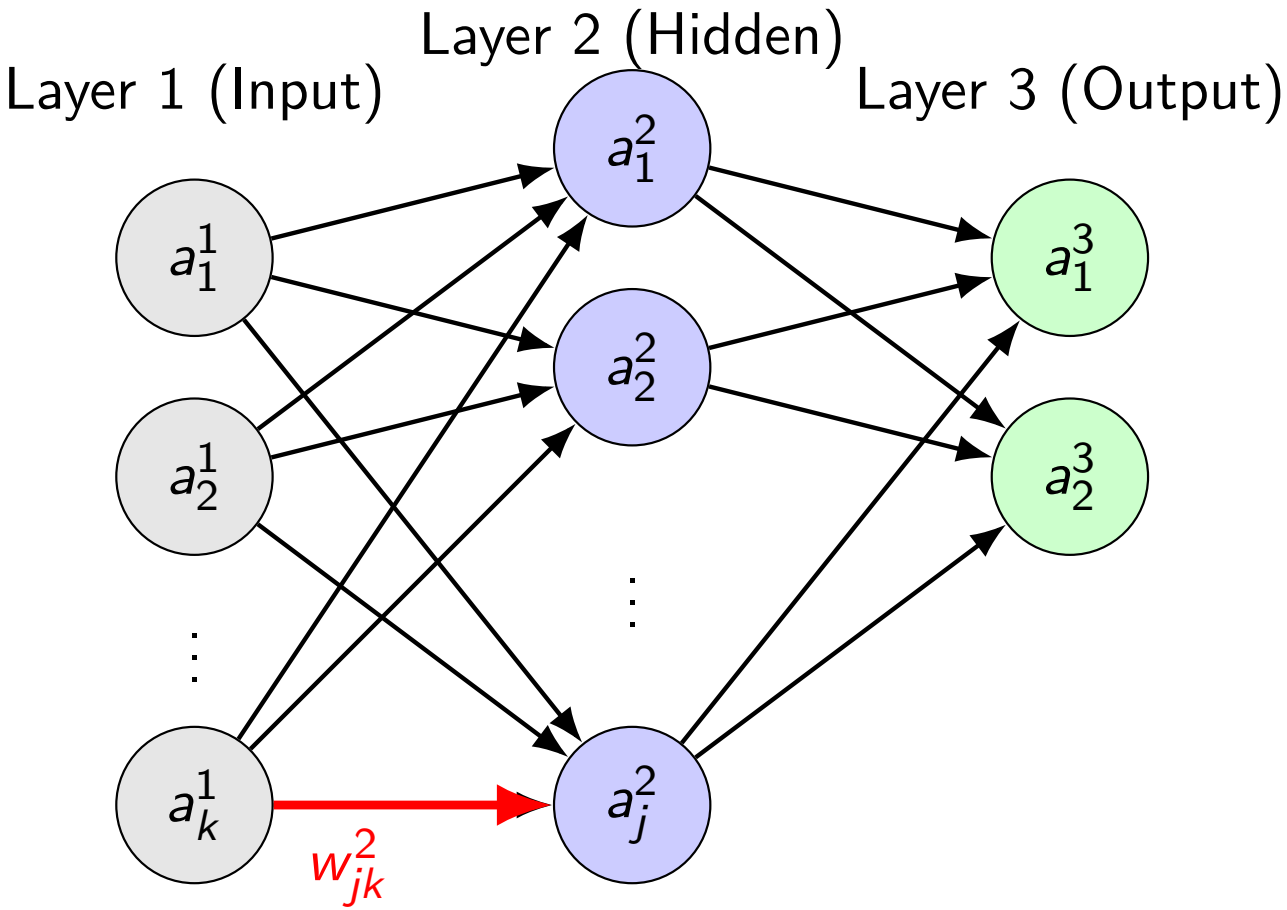


Figure: A simple feedforward network.

The Two Assumptions About the Cost Function

The goal of backpropagation is to compute the partial derivatives $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$ for the cost function J.
To use backpropagation, we need two main assumptions.

The Two Assumptions About the Cost Function

The goal of backpropagation is to compute the partial derivatives $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$ for the cost function J.
To use backpropagation, we need two main assumptions.

Assumption 1: Average Cost

The cost function can be written as an average over per-example cost functions J_x.

$$J = \frac{1}{n} \sum_x J_x$$

Backpropagation allows us to compute $\frac{\partial J_x}{\partial w}$ and $\frac{\partial J_x}{\partial b}$ for a **single example** x. We then get the full gradient by **averaging** these over all training examples. (From now on, we'll consider the cost J for a single example x.)

The Two Assumptions About the Cost Function

Assumption 2: Function of Outputs

The cost J can be written as a function of the **output activations** $\mathbf{a}^L$ from the network.

For example, the **quadratic cost function**:

$$J = \frac{1}{2}\|\mathbf{y} - \mathbf{a}^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2$$

This clearly depends on the output activations a_j^L. (The desired output $\mathbf{y}$ is a fixed parameter).

The Hadamard Product, $\odot$

Backpropagation uses common linear algebra operations, plus one less common one.

Definition: The Hadamard Product

Suppose **s** and **t** are two vectors of the **same dimension**.
The Hadamard product, denoted $\mathbf{s} \odot \mathbf{t}$, is the **element-wise product** of the two vectors.

$$(s \odot t)_j = s_j t_j$$

The Hadamard Product, $\odot$

Backpropagation uses common linear algebra operations, plus one less common one.

Definition: The Hadamard Product

Suppose **s** and **t** are two vectors of the **same dimension**.
The Hadamard product, denoted $\mathbf{s} \odot \mathbf{t}$, is the **element-wise product** of the two vectors.

$$(s \odot t)_j = s_j t_j$$

Example

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \times 3 \\ 2 \times 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

This operation is crucial for the matrix-based equations in backpropagation.

The Four Fundamental Equations

Backpropagation is built on four key equations. To get there, we must first define **error**.

Definition: Error (δ_j^l)

We define the **error** δ_j^l of neuron j in layer l as the partial derivative of the cost J with respect to the neuron's **weighted input** z_j^l.

$$\delta_j^l \equiv \frac{\partial J}{\partial z_j^l}$$

We use $\boldsymbol{\delta}^l$ for the vector of errors in layer l.

Backpropagation gives us a way to compute $\boldsymbol{\delta}^l$ for every layer, and then relates this error to the gradients we want ($\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$).

Equation (Eq1): Error in the Output Layer

The error δ^L in the output layer L is given by:

$$\boxed{\delta^L = \nabla_{\mathbf{a}} J \odot \sigma'(\mathbf{z}^L) \quad \text{(Eq1)}}$$

- $\nabla_{\mathbf{a}} J$ is the vector of partial derivatives $\frac{\partial J}{\partial a_j^L}$. This measures how fast the cost changes with respect to the output activations.
- $\sigma'(\mathbf{z}^L)$ is the vector of derivatives $\sigma'(z_j^L)$ of the activation function, evaluated at the weighted inputs z_j^L.
- $\odot$ is the **Hadamard product** (element-wise multiplication).

Component form: $\delta_j^L = \frac{\partial J}{\partial a_j^L} \sigma'(z_j^L)$

Equation (Eq1): Error in the Output Layer

The error $\boldsymbol{\delta}^L$ in the output layer L is given by:

$$\boxed{\boldsymbol{\delta}^L = \nabla_{\mathbf{a}} J \odot \sigma'(\mathbf{z}^L) \quad \text{(Eq1)}}$$

- $\nabla_{\mathbf{a}} J$ is the vector of partial derivatives $\frac{\partial J}{\partial a_j^L}$. This measures how fast the cost changes with respect to the output activations.
- $\sigma'(\mathbf{z}^L)$ is the vector of derivatives $\sigma'(z_j^L)$ of the activation function, evaluated at the weighted inputs z_j^L.
- $\odot$ is the **Hadamard product** (element-wise multiplication).

Component form: $\delta_j^L = \frac{\partial J}{\partial a_j^L} \sigma'(z_j^L)$

Remark

Recall from the graph of the sigmoid function that σ becomes very flat when $\sigma(z_j^L)$ is approximately 0 or 1. When this occurs we will have $\sigma'(z_j^L) \approx 0 \Rightarrow$ **a weight in the final layer will learn slowly if the output neuron is either low or high in activation (saturated neuron)**. Similar remarks hold also for the biases of output neuron.

Equation (Eq2): Error in terms of Next Layer

The error δ^l in layer l is computed in terms of the error δ^{l+1} from the **next** layer:

$$\boxed{\delta^l = \left((W^{l+1})^T \delta^{l+1}\right) \odot \sigma'(\mathbf{z}^l)} \quad \text{(Eq2)}$$

- $(W^{l+1})^T$ is the **transpose** of the weight matrix for layer $l+1$.
- This equation is the heart of "backpropagation". We start with δ^L (using Eq1) and use (Eq2) to compute δ^{L-1}, δ^{L-2}, and so on, **moving backward** through the network.

Remark

Equations (Eq3): The Gradients

The error $\boldsymbol{\delta}^l$ tells us the rate of change of the cost with respect to the **biases** and **weights**.

Equation (Eq3): Gradient for the Biases

The gradient for any bias b_j^l is **exactly equal** to the error δ_j^l.

$$\boxed{\frac{\partial J}{\partial b_j^l} = \delta_j^l} \quad \text{(Eq3)}$$

Vector form: $\frac{\partial J}{\partial \mathbf{b}^l} = \boldsymbol{\delta}^l$

Equations (Eq4): The Gradients

Equation (Eq4): Gradient for the Weights

The gradient for any weight w_{jk}^{l} is the product of the input activation from the **previous** layer and the error of the **current** layer.

$$\boxed{\frac{\partial J}{\partial w_{jk}^{l}} = a_{k}^{l-1}\delta_{j}^{l}} \quad \text{(Eq4)}$$

Proof of (Eq1): Output Layer Error I

Goal: Prove $\delta_j^L = \frac{\partial J}{\partial a_j^L}\sigma'(z_j^L)$.

1. **Start with the definition** of error:

$$\delta_j^L = \frac{\partial J}{\partial z_j^L}$$

2. Apply the **chain rule**, summing over all neurons k in the output layer L:

$$\delta_j^L = \sum_k \frac{\partial J}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L}$$

3. The activation $a_k^L = \sigma(z_k^L)$. The derivative $\frac{\partial a_k^L}{\partial z_j^L}$ is non-zero **only when** $k = j$.

$$\frac{\partial a_k^L}{\partial z_j^L} = \begin{cases} \sigma'(z_j^L) & \text{if } k = j \\ 0 & \text{if } k \neq j \end{cases}$$

Proof of (Eq1): Output Layer Error II

4. The sum simplifies to just the $k = j$ term:

$$\delta_j^L = \frac{\partial J}{\partial a_j^L} \sigma'(z_j^L)$$

This proves (Eq1). In matrix form: $\boldsymbol{\delta}^L = \nabla_{\mathbf{a}} J \odot \sigma'(\mathbf{z}^L)$.

Proof of (Eq2): Propagating the Error I

Goal: Prove $\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$.

1. **Start with the definition** of δ_j^l:

$$\delta_j^l = \frac{\partial J}{\partial z_j^l}$$

2. Apply the **chain rule** to relate this to the error in the *next* layer $l+1$:

$$\delta_j^l = \sum_k \frac{\partial J}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

3. We need $\frac{\partial z_k^{l+1}}{\partial z_j^l}$. Recall the definition of z_k^{l+1}:

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}$$

Proof of (Eq2): Propagating the Error II

4. Taking the partial derivative with respect to z_j^l:

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$

5. **Substitute** this back into step 2:

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l) = \sigma'(z_j^l) \sum_k w_{kj}^{l+1} \delta_k^{l+1}$$

This proves (Eq2). In matrix form: $\boldsymbol{\delta}^l = \left((W^{l+1})^T \boldsymbol{\delta}^{l+1}\right) \odot \sigma'(\mathbf{z}^l)$.

Proof of (Eq3): Bias Gradients

Goal: Prove $\frac{\partial J}{\partial b_j^l} = \delta_j^l$.

1. By chain rule: $\frac{\partial J}{\partial b_j^l} = \frac{\partial J}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l}$.
2. By definition, $\frac{\partial J}{\partial z_j^l} = \delta_j^l$.
3. We know $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$.
4. Therefore, $\frac{\partial z_j^l}{\partial b_j^l} = 1$.
5. Substituting back: $\frac{\partial J}{\partial b_j^l} = \delta_j^l \cdot 1 = \delta_j^l$.

Proof of (Eq4): Weight Gradients

Goal: Prove $\frac{\partial J}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$.

1. By chain rule: $\frac{\partial J}{\partial w_{jk}^l} = \frac{\partial J}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l}$.
2. By definition, $\frac{\partial J}{\partial z_j^l} = \delta_j^l$.
3. We know $z_j^l = \sum_i w_{ji}^l a_i^{l-1} + b_j^l$.
4. Therefore, $\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1}$.
5. Substituting back: $\frac{\partial J}{\partial w_{jk}^l} = \delta_j^l \cdot a_k^{l-1} = a_k^{l-1}\delta_j^l$.

The Backpropagation Algorithm

The four equations (Eq1-Eq4) give us a complete algorithm to compute the gradient for a single training example x.

1. **Input** x: Set the activation $\mathbf{a}^1$ for the input layer.
2. **Feedforward**: For each layer $l = 2, 3, \ldots, L$:
   - Compute $\mathbf{z}^l = W^l \mathbf{a}^{l-1} + \mathbf{b}^l$
   - Compute $\mathbf{a}^l = \sigma(\mathbf{z}^l)$
3. **Output Error** $\boldsymbol{\delta}^L$: Compute the error for the output layer:
   - $\boldsymbol{\delta}^L = \nabla_{\mathbf{a}} J \odot \sigma'(\mathbf{z}^L)$
4. **Backpropagate Error**: For each layer $l = L-1, L-2, \ldots, 2$:
   - $\boldsymbol{\delta}^l = \left((W^{l+1})^T \boldsymbol{\delta}^{l+1}\right) \odot \sigma'(\mathbf{z}^l)$
5. **Output Gradient**: The gradient of the cost J_x is:
   - $\frac{\partial J_x}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$
   - $\frac{\partial J_x}{\partial b^l_j} = \delta^l_j$

In What Sense is Backpropagation Fast?

Let's compare backpropagation to a "naive" way of computing the gradient.

The Naive (Slow) Approach

We can use the definition of a partial derivative as a finite difference:

$$\frac{\partial J}{\partial w_j} \approx \frac{J(w + \epsilon e_j) - J(w)}{\epsilon}$$

where e_j is a unit vector in the j^{th} direction.

In What Sense is Backpropagation Fast?

Let's compare backpropagation to a "naive" way of computing the gradient.

The Naive (Slow) Approach

We can use the definition of a partial derivative as a finite difference:

$$\frac{\partial J}{\partial w_j} \approx \frac{J(w + \epsilon e_j) - J(w)}{\epsilon}$$

where e_j is a unit vector in the j^{th} direction.
To compute the gradient, we must do this for **every single weight** w_j.
If our network has **1,000,000 weights**, we must:

- Perform one forward pass to compute $J(w)$.
- Perform **1,000,000** additional forward passes to compute $J(w + \epsilon e_j)$ for each j.

Total cost: **1,000,001 forward passes**. This is extremely slow.

In What Sense is Backpropagation Fast?

The Backpropagation (Fast) Approach

Backpropagation computes **all partial derivatives** $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$ **simultaneously**.

What is the computational cost?

- **One forward pass**: to compute all activations a^l and weighted inputs z^l.
- **One backward pass**: to compute all errors δ^l.

The cost of the backward pass is roughly the same as the forward pass (both are dominated by matrix-vector products).

Total cost: Roughly the same as **two forward passes**.

In What Sense is Backpropagation Fast?

The Backpropagation (Fast) Approach

Backpropagation computes **all partial derivatives** $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$ **simultaneously**.

What is the computational cost?

- **One forward pass**: to compute all activations a^l and weighted inputs z^l.
- **One backward pass**: to compute all errors δ^l.

The cost of the backward pass is roughly the same as the forward pass (both are dominated by matrix-vector products).

Total cost: Roughly the same as **two forward passes**.
1,000,001 passes (naive) vs. **2 passes** (backprop)
This is why backpropagation is a fast algorithm.

Backpropagation: The Big Picture

What is backpropagation *really* doing?

- A small change Δw^l_{jk} to a weight in layer l causes a small change in the activation Δa^l_j.
- This change Δa^l_j **propagates forward** through all subsequent layers $(l+1, l+2, \ldots, L)$, causing a change in each of them.
- Finally, the change reaches the output layer L and causes a change in the total cost, ΔJ.
- The partial derivative $\frac{\partial J}{\partial w^l_{jk}}$ is simply a measure of this total change ΔJ caused by the initial Δw^l_{jk}.

Backpropagation: The Big Picture

What is backpropagation *really* doing?

- A small change Δw^l_{jk} to a weight in layer l causes a small change in the activation Δa^l_j.
- This change Δa^l_j **propagates forward** through all subsequent layers $(l+1, l+2, \ldots, L)$, causing a change in each of them.
- Finally, the change reaches the output layer L and causes a change in the total cost, ΔJ.
- The partial derivative $\frac{\partial J}{\partial w^l_{jk}}$ is simply a measure of this total change ΔJ caused by the initial Δw^l_{jk}.

Backpropagation is a systematic, efficient algorithm that uses the chain rule to track and compute the precise value of this propagated change for every weight and bias in the network, all at once.
It works **backward** from the cost ΔJ to figure out the contribution of each parameter.