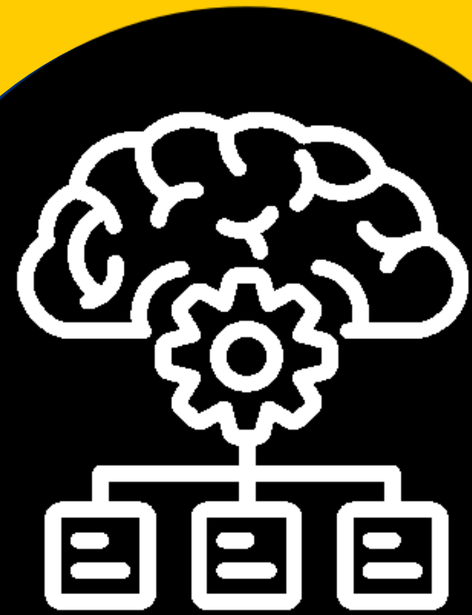
Deep Learning

Artificial Neural Network (ANN)

Equipping You with Research Depth and Industry Skills



By:

Dr. Zohair Ahmed

www.youtube.com/@ZohairAI Subscribe

www.begindiscovery.com

Brain and Machine

Brain:

- Recognizes faces even in poor lighting (pattern recognition).
- Remembers associated facts (association).
- Can handle massive complexity without explicit formulas.
- Tolerant to noisy/incomplete data.
- Example: reading messy handwriting.

Machine:

- Performs billions of multiplications per second with **perfect accuracy**.
- Follows strict logic no ambiguity.
- Excellent at structured computation (e.g., solving equations).

Brain and Machine

- Computers = **serial processing**.
- The processor and memory are separated parts.
- They are connected by “wires” (data bus).
  - The CPU keeps asking memory
  - “Give me the next data!”, “Now give me instructions!”
  - This creates a bottleneck
- **A bottleneck means:** The CPU is faster than the memory can deliver data.
- **Like:** A teacher who teaches very fast, But students walk slowly to bring books from the shelf. Teacher must wait
- Same in computers: CPU is super fast
  - Memory is slower to transfer data
  - CPU wastes time waiting
  - This waiting time = memory bottleneck
- Brain = **parallel processing**.
- Billions of slow neurons, but each fires simultaneously → results in extremely fast recognition.
- **Analogy:**
  - Von Neumann = “one genius mathematician solving problems one by one.”
  - Brain = “a billion average workers, each solving tiny pieces simultaneously.”
- This parallelism is the foundation of ANN design (many neurons connected, simple operations, but together they solve complex problems).

Can CPU work Parallel?

- No: today’s CPUs cannot work in parallel like the human brain.
- The brain has billions of tiny processors (neurons) working at the same time.
- A CPU has few cores (8, 16, 32…), not billions.
- So, a CPU processes most things step-by-step, not massively parallel like the brain.
- Brain = massive parallel processing
- CPU = limited parallel + mostly sequential

Brain’s Power

1. The brain has a HUGE team

- Think of the brain like:
- 10 billion tiny workers (neurons)
- Each worker has 1,000 friends it talks to (connections)
- So, in total:
- $10^{10}\ \text{neurons} \times 1000 = 10^{13}\ \text{connections}\ (\text{synapses})$
- That’s 10 trillion communication lines!

2. Each brain cell is slow

- A single neuron fires only about:
- 200 times per second
- This is extremely slow compared to a CPU that runs in billions of cycles per second.

3. But the brain works in MASSIVE parallel

- Even though each neuron is slow…
- All 10 billion neurons work at the same time.
- One worker = slow
- But 10 billion workers working together = extremely powerful
- This is why the brain is still faster at many tasks.

Brain’s Power

4. Recognizing an image in 200 ms

- Even though neurons are slow…
- Because so many work together, the brain can:
  - See a face
  - Understand it
  - Make a decision
- All in 0.2 seconds (200 ms).
- This speed comes from parallel teamwork, not fast individual neurons.

5. Machines use the same idea

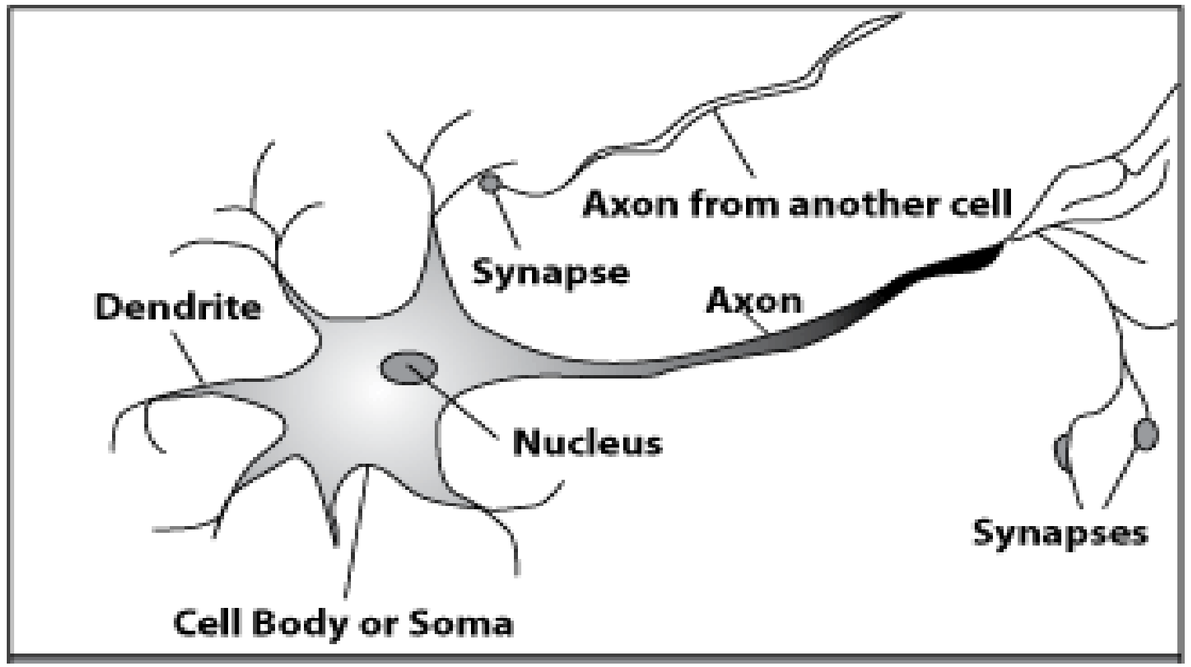
- Computers originally used:
- A few very fast processors
- But the brain taught us something:
- Instead of one super-powerful unit,
- it’s better to have many simple units working together.
- So, modern AI (neural networks) is designed like this:
  - Many simple “nodes”
  - Connected like neurons
  - Working in parallel
- This is how machines try to copy the brain’s style.

The Biological Inspiration

- The brain is like a giant city made of billions of tiny people called neurons.

1. Dendrites = “Ears” of the neuron

- A neuron has many dendrites.
- They receive messages from other neurons.
- Think of dendrites as phone lines coming into a person’s phone.
- Dendrites = Inputs



The Biological Inspiration

- **2. Cell Body (Soma) = “Decision Center”**
- Inside the neuron is the soma.
- Collects all incoming messages
- Adds them up
- Decides whether to send a message forward
- If the total input is strong enough → fires.
- If not → stays silent.
- Soma = Weighs the inputs & decides

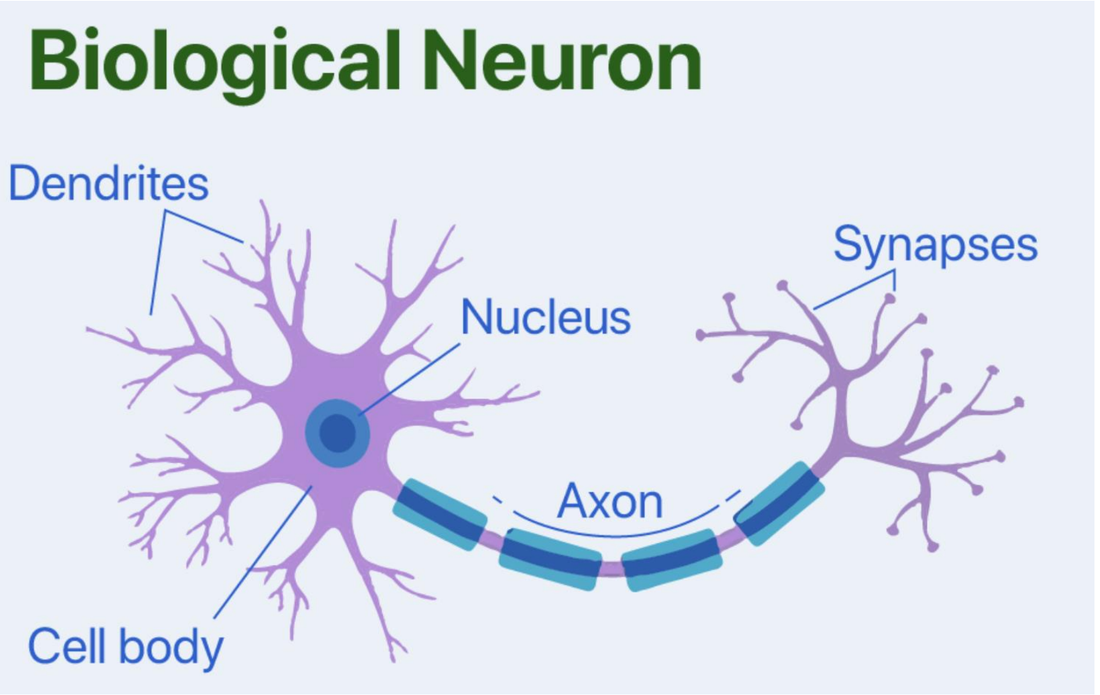
- **3. Axon = “Output wire”**
- Once the neuron decides to send a signal:
- It travels down the axon
- Axon is like a long wire that carries the output message to other neurons
- Axon = Output channel

The Biological Inspiration

- **4. Synapses = “Volume knobs”**
- Synapses are the tiny gaps between neurons where the signal passes.
- **Synapses can:**
- Strengthen signals (positive weight)
- Weaken signals (negative weight)
- They are like volume knobs that control how strong the message is.
- Synapse = Weight control
- **Example: Vision in the brain**
- The occipital lobe at the back of your brain does the first step of seeing:
  - Detects edges
  - Colors
  - Shapes
- But recognizing a face (your friend, your mother, a celebrity) is NOT done by one neuron.
- It is done by many neurons working together across a network.
- No single neuron stores “Mom’s face.”
- Instead, patterns of connections store that knowledge.
- Knowledge in the brain = pattern spread across many neurons

The Biological Inspiration

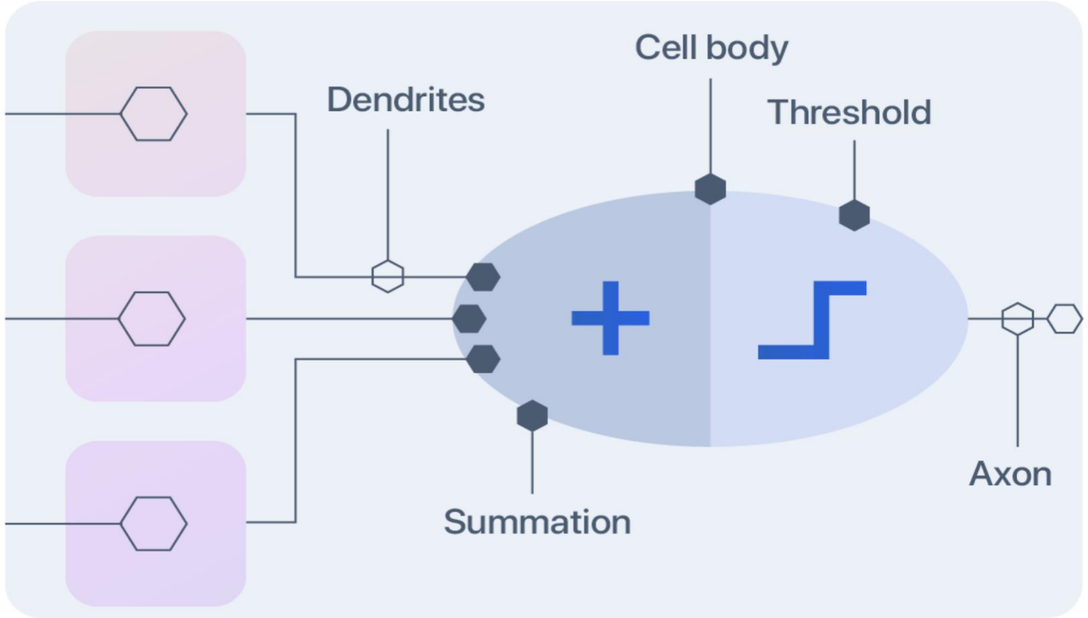
- Every neuron has:
  - One cell body (soma)
  - Many dendrites (input wires)
  - One axon (output wire)
- So:
- One neuron → many dendrites → one axon
- Neuron = whole cell
- Soma = the center of that cell
- Dendrites are just branches coming OUT of the neuron



Artificial Neural Networks (ANN)

In ANNs:

- Nodes = neurons
- Weights = synapses
- Activation = firing
- Layer input = dendrites
- Layer output = axon
- **The MOST important part:**
- ML systems also store knowledge in patterns of weights across many nodes, never in one single neuron.
- So just like your brain:
- One node does not “know” anything alone
- The network together holds the knowledge



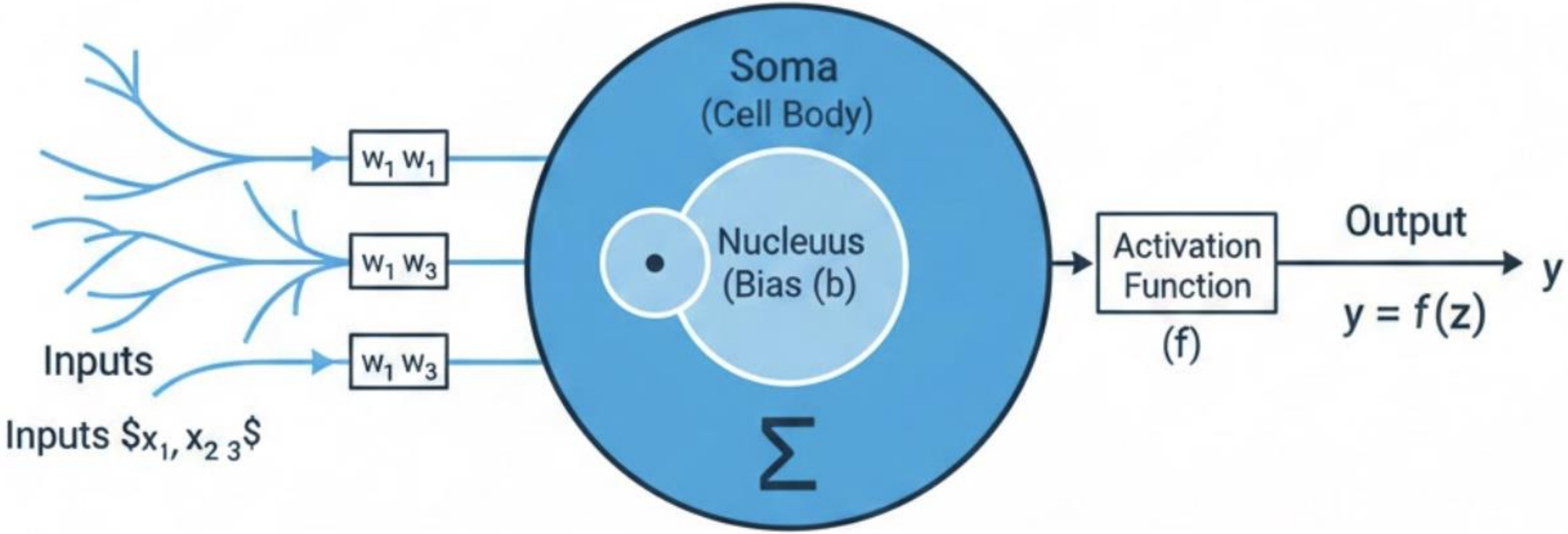
Artificial Neural Networks (ANN)

Biological Neuron	Artificial Neural Network (ANN)	Functional Role
Dendrites	Inputs x_n	Receiving signals from other neurons or sensors.
Soma (Cell Body)	Summation $\sum$	Aggregating all incoming electrical/chemical signals.
Nucleus	Bias (b)	Setting the baseline or threshold for the neuron to activate.
Synapse	Weights w_n	Determining the strength of the connection (The "Learning" part).
Axon	Output Path / Activation	Transmitting the processed signal to the next layer.

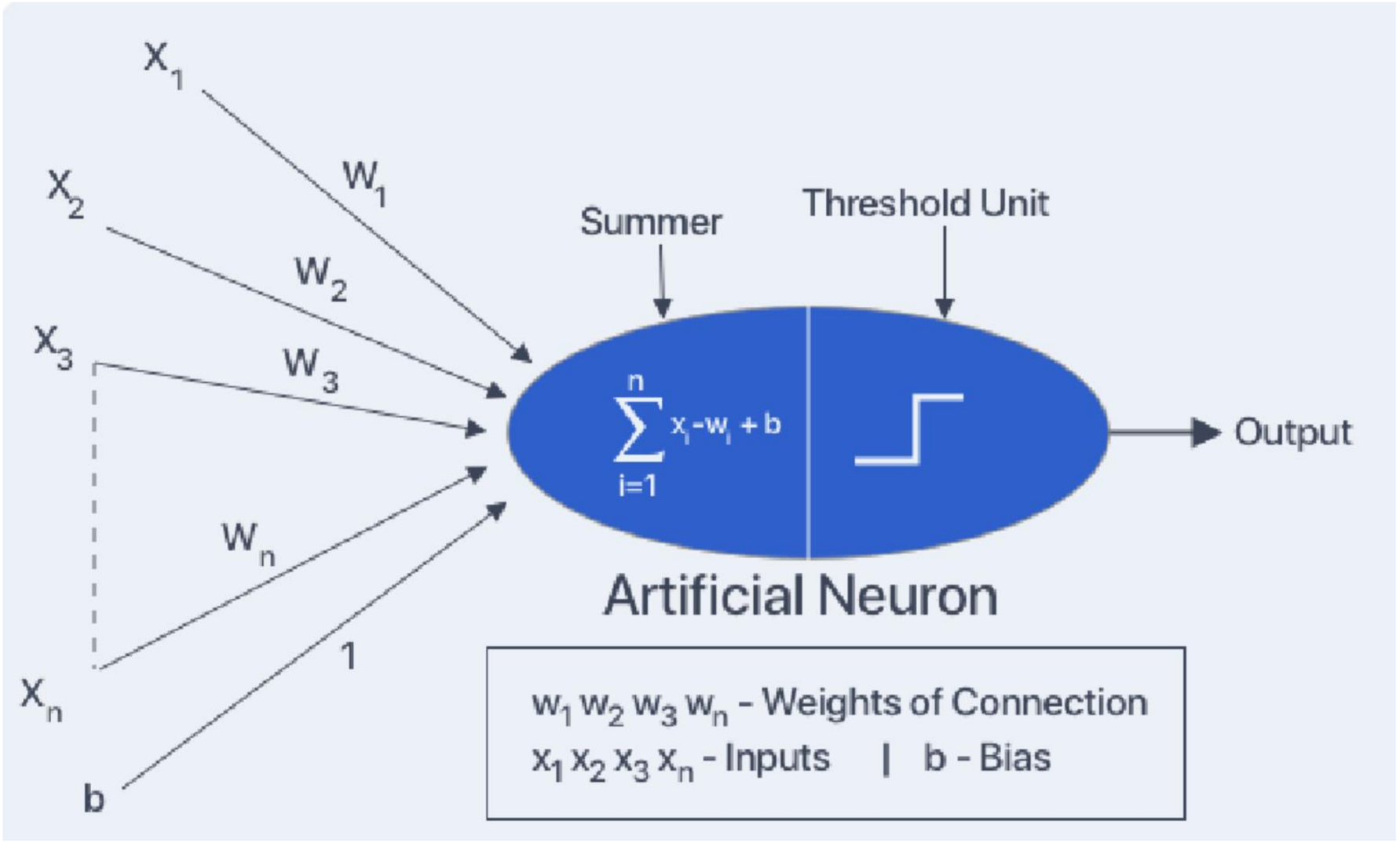
Artificial Neural Networks (ANN)

Biological Neuron	Artificial Neural Network (ANN)	Functional Role
Dendrites	Inputs x_n	Receiving signals from other neurons or sensors.
Soma (Cell Body)	Summation $\sum$	Aggregating all incoming electrical/chemical signals.
Nucleus	Bias (b)	Setting the baseline or threshold for the neuron to activate.
Synapse	Weights w_n	Determining the strength of the connection (The "Learning" part).
Axon	Output Path / Activation	Transmitting the processed signal to the next layer.

Artificial Neural Networks (ANN)



Artificial Neural Networks (ANN)



A Simple Model of a Neuron(Perceptron)

- A Perceptron is the simplest form of a neural network that makes decisions by combining inputs with weights and applying an activation function. It is mainly used for binary classification problems. It forms the basic building block of many deep learning models.
- Each neuron has a threshold value
- Each neuron has weighted inputs from other neurons
- The input signals form a weighted sum
- If the activation level exceeds the threshold, the neuron “fires”

A Simple Model of a Neuron(Perceptron)

1. Each neuron has a threshold value

- Think of the threshold like a minimum score needed to say YES.
- **Example:** "If the total score reaches 10, I will fire."

2. Each neuron gets weighted inputs

- Inputs are just numbers coming from other neurons or data.
- Each input has a weight (importance).
  - A big weight → input matters a lot
  - A small weight → input matters a little
  - A negative weight → input pushes toward “NO”

3. Inputs make a weighted sum

- The neuron calculates:
- $\{\boldsymbol{Total}\} = (\boldsymbol{w_1} \times \boldsymbol{x_1}) + (\boldsymbol{w_2} \times \boldsymbol{x_2}) + \cdots$
- Multiply each input by its importance → then add them up.
- **4. If activation > threshold → neuron fires**
- After adding the weighted inputs, the neuron checks:
- If total ≥ threshold → FIRE (output = 1)
- If total < threshold → Don’t fire (output = 0)
- This is called activation.

An Artificial Neuron

1. Each neuron has weighted input connections from the previous layer

- Think of each neuron like a person receiving messages from several friends.
  - Friends = neurons from the previous layer
  - Messages = input values
  - Loudness of each friend = weight
- So if you have inputs: $x1, x2, x3$
- The neuron receives: w_1x_1, w_2x_2, w_3x_3
- Each input x_i is multiplied by a weight w_i.
- Weights tell the neuron how important each input is.

2. Weighted sum → Activation level

- After receiving all inputs, the neuron adds them:
- $\text{activation} = (w_1x_1 + w_2x_2 + w_3x_3 + b)$
- Where b = bias (like the neuron's personal preference).
- So activation is simply: a total score of all weighted inputs

3. Sigmoid activation function → Final output

- The activation score is then passed through the sigmoid function:
- $\text{output} = \sigma(\text{activation})$
- The sigmoid: $\boldsymbol{\sigma(z) = \frac{1}{1+e^{-z}}}$
- Calculate for x=2
- Euler's number e = 2.71828
- Answer: 0.88
- squashes values to a range between 0 and 1
- makes the neuron's output smooth, models probability
- If activation is high → output close to 1
- If activation is low → output close to 0
- Putting It All Together
- **Inputs → weighted → added → sigmoid → output**
- **Mathematically:** $\text{output} = \sigma(w_1x_1 + w_2x_2 + w_3x_3 + b)$

An Artificial Neuron

Imagine a "decision machine" evaluating whether you should drink tea:

- **Inputs =** weather, mood, time of day
- **Weights =** how important each factor is
- **Weighted sum =** overall desire score
- **Sigmoid =** “convert score to YES probability”

The Perceptron Algorithm

Frank Rosenblatt suggested this algorithm:

1.Set a threshold value

2.Multiply all inputs with its weights

3.Sum all the results

4.Activate the output

1. Set a threshold value:

- Threshold = 1.5

2. Multiply all inputs with its weights:

- x1 * w1 = 1 * 0.7 = 0.7
- x2 * w2 = 0 * 0.6 = 0
- x3 * w3 = 1 * 0.5 = 0.5
- x4 * w4 = 0 * 0.3 = 0
- x5 * w5 = 1 * 0.4 = 0.4

3. Sum all the results:

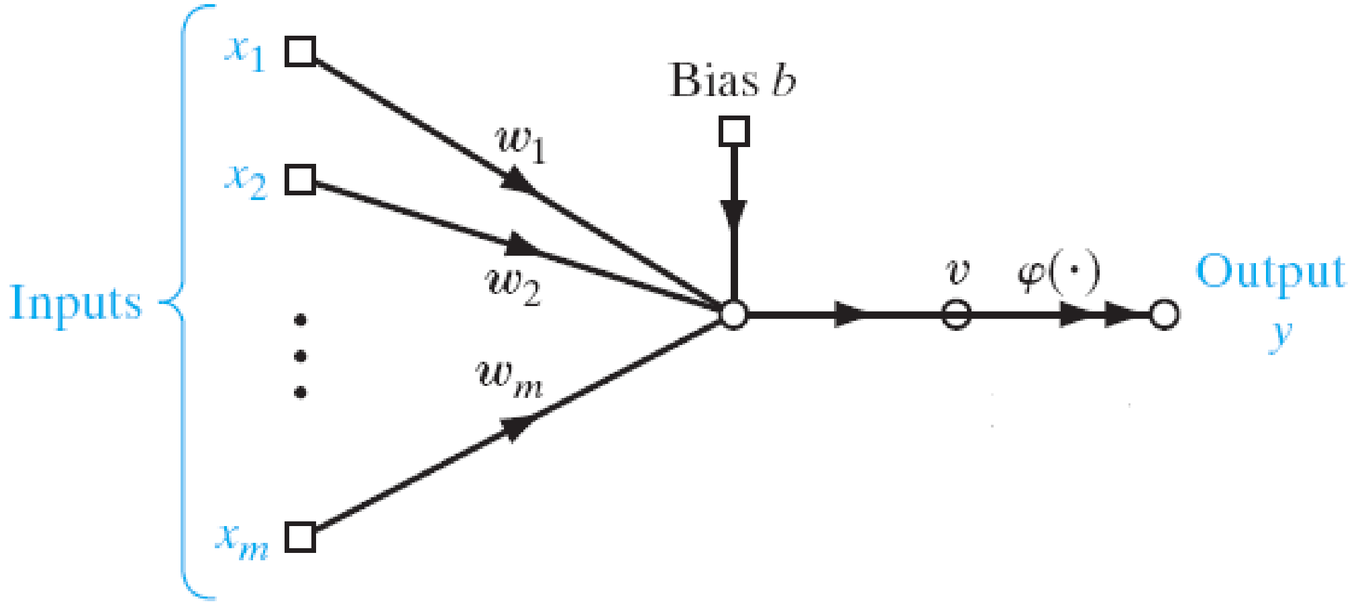
- 0.7 + 0 + 0.5 + 0 + 0.4 = 1.6 (The Weighted Sum)

4. Activate the Output:

- Return true if the sum > 1.5 ("Yes I will go to the Concert")

Rosenblatt's perceptron model

- Rosenblatt's original perceptron model contained only one layer.
- From this, a multi-layered model was derived in 1960.
- The input is summed after multiplying with weights and the output is produced with as positive or negative.



Rosenblatt's perceptron model

- **Weights** are numerical values assigned to the connections between neurons. They find how much influence each input has on the network’s final output.
- **Example:** In a neural network predicting house prices, the weight for the "size of the house" find how much the house size influences the price prediction. The larger the weight, the bigger the impact size will have on the final result.
- **Biases** are additional parameters that adjust the output of a neuron. Unlike weights, they are not tied to any specific input but instead shift the activation function to better fit the data.
- **Example:** In a house price prediction network, the bias might ensure that even for a house with a size of zero, the model predicts a non-zero price. This could reflect a fixed value such as land value or other baseline costs.

Rosenblatt’s Perceptron model

- At first, the use of the multi-layer perceptron (MLP) was complicated by the lack of an appropriate learning algorithm.
- In 1974, Werbos came to introduce a so-called backpropagation algorithm for the three-layered perceptron network.

Rosenblatt's Perceptron model

- The synaptic weights of the perceptron are denoted by w1,w2, ...,wm.
- The inputs applied to the perceptron are denoted by x1, x2, ..., xm.
- The externally applied bias is denoted by b.
- The output can be calculated by:

$$v = \sum_{i=1}^{m} w_i x_i + b$$

Rosenblatt's Perceptron Model

- In the simplest form of the perceptron, there are two decision regions separated by a *hyperplane*, which is defined by

$$\sum_{i=1}^{m} w_i x_i + b = 0$$

Rosenblatt's perceptron Model

- A perceptron computes a weighted sum:
- $v = \mathbf{w}^T\mathbf{x}$
- where:
- **w** = weight vector
- **x** = input vector
- $\mathbf{w}^T\mathbf{x}$= dot product (weighted sum)
- Then it applies the sign function:
- $y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$
- **This gives a quantized response:**

- +1 = belongs to class 1
- –1 = belongs to class 2
- “Quantized” means **converted into discrete outputs**.

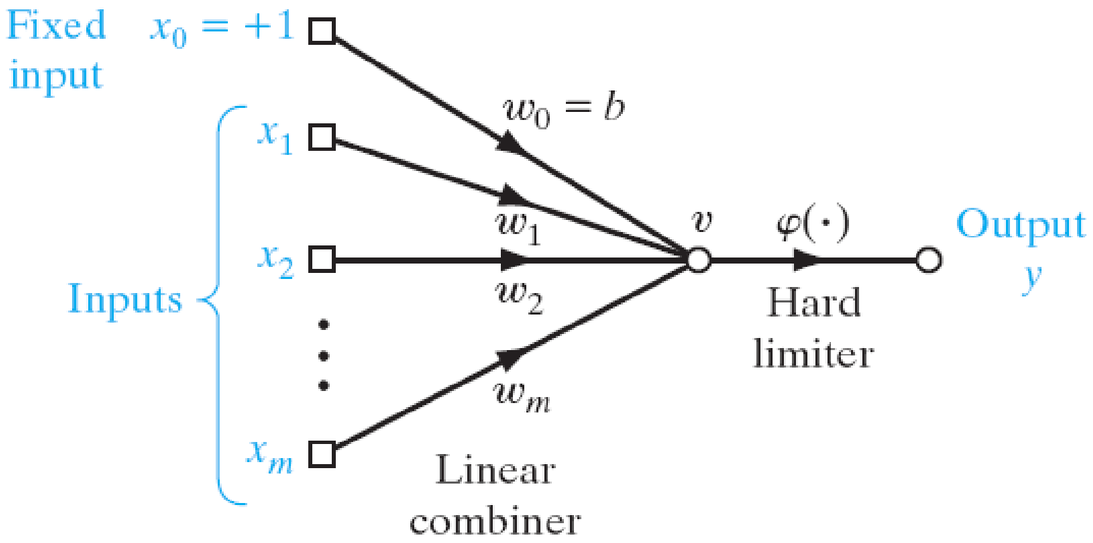
$$\text{sgn}(v) = \begin{cases} +1 & \text{if } v > 0 \\ -1 & \text{if } v < 0 \end{cases}$$

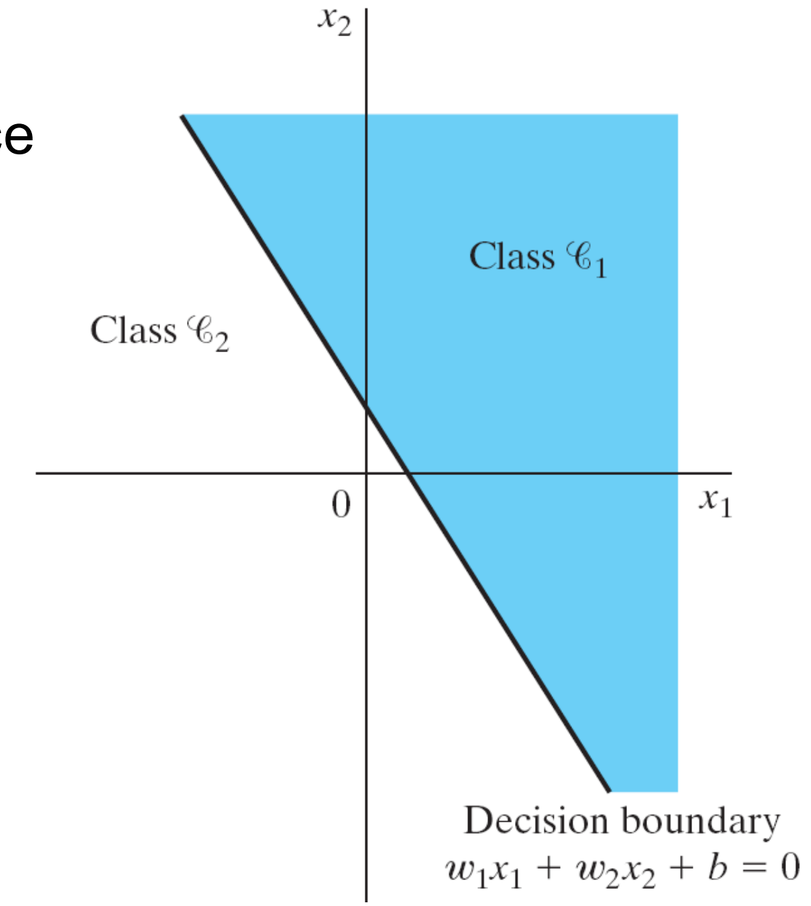
the *quantized response* $y(n)$ of the perceptron in the compact form

$$y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$$

Rosenblatt's perceptron Model

- Decision boundary
- A hard limiter is an activation function that forces the perce ed on a strict threshold.





Rosenblatt's perceptron Model

- **Understanding the update rule**

1. Compute actual output

- $y(n) = \text{sgn}\big(w^T(n)x(n)\big)$

2. Check if it matches desired output

- If **wrong** → adjust weights
- **3. Weight update** $\Delta w = \eta\big(d(n) - y(n)\big)x(n)$
- $d(n)$= correct label
- $y(n)$= predicted label
- $x(n)$= input vector
- **Suppose:**
- Desired output d=+1
- Actual output y=−1
- **Weight update:** $\Delta w = \eta(+1 - (-1))x = 2\eta x$
- This pushes the boundary **toward the misclassified point**.
- If the opposite:
- Desired output d=−1
- Actual output y=+1
- Then: $\Delta w = \eta(-1 - 1)x = -2\eta$
- Pushes the boundary **away** from the point.

Think of η as the **speed of learning**:

- η BIG → fast learning (but risky)
- η SMALL → slow, steady learning (but safe)
- Just like adjusting how big each step is when walking toward a target.

Perceptron Learning

TABLE 1.1 Summary of the Perceptron Convergence Algorithm

Variables and Parameters:

$\mathbf{x}(n)$ = $(m+1)$-by-1 input vector
$= [+1, x_1(n), x_2(n), \ldots, x_m(n)]^T$
$\mathbf{w}(n)$ = $(m+1)$-by-1 weight vector
$= [b, w_1(n), w_2(n), \ldots, w_m(n)]^T$
b = bias
$y(n)$ = actual response (quantized)
$d(n)$ = desired response
η = learning-rate parameter, a positive constant less than unity

1. *Initialization.* Set $\mathbf{w}(0) = \mathbf{0}$. Then perform the following computations for time-step $n = 1, 2, \ldots$.
2. *Activation.* At time-step n, activate the perceptron by applying continuous-valued input vector $\mathbf{x}(n)$ and desired response $d(n)$.
3. *Computation of Actual Response.* Compute the actual response of the perceptron as

$$y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$$

where $\text{sgn}(\cdot)$ is the signum function.

4. *Adaptation of Weight Vector.* Update the weight vector of the perceptron to obtain

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta[d(n) - y(n)]\mathbf{x}(n)$$

where

$$d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1 \\ -1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2 \end{cases}$$

5. *Continuation.* Increment time step n by one and go back to step 2.

Rosenblatt's Perceptron Model

- No need to update weights if correctly predicted:

$$\mathbf{w}(n+1) = \mathbf{w}(n) \quad \text{if } \mathbf{w}^T\mathbf{x}(n) > 0 \text{ class } \mathscr{C}_1$$
$$\mathbf{w}(n+1) = \mathbf{w}(n) \quad \text{if } \mathbf{w}^T\mathbf{x}(n) \leq 0 \text{ class } \mathscr{C}_2$$

- The weight is updated for incorrect predictions as:

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta(n)\mathbf{x}(n) \quad \text{if } \mathbf{w}^T(n)\mathbf{x}(n) > 0$$
$$\mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2$$

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta(n)\mathbf{x}(n) \quad \text{if } \mathbf{w}^T(n)\mathbf{x}(n) \leq 0$$
$$\mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1$$

Rosenblatt's perceptron model

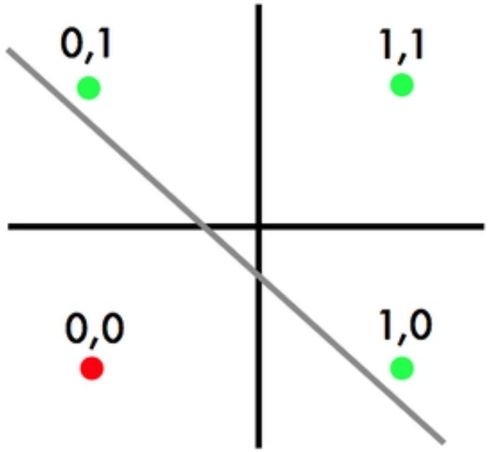
- learning-rate parameter $\eta(n)$ controls the adjustment applied to the weight vector at iteration n.
- Usually the value of is set to $\eta>0$
- It can be set to a fixed value
- It can also be adaptively updated for each iteration.

Rosenblatt's perceptron model

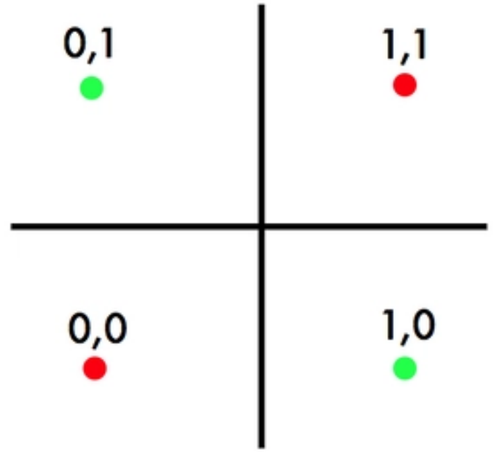
- Rosenblatt’s perceptron, which is basically a single-layer neural network
- Network is limited to the classification of linearly separable patterns.
- What if a more complex and deeper network architecture is introduced?

Decision boundaries

- In simple cases, divide feature space by drawing a hyperplane across it.
- Known as a decision boundary.
- Discriminant function: returns different values on opposite sides. (straight line)
- Problems which can be thus classified are linearly separable.
- **AND gate** → linearly separable.
- **OR gate** → linearly separable.
- **XOR gate** → not linearly separable → needs multiple layers.

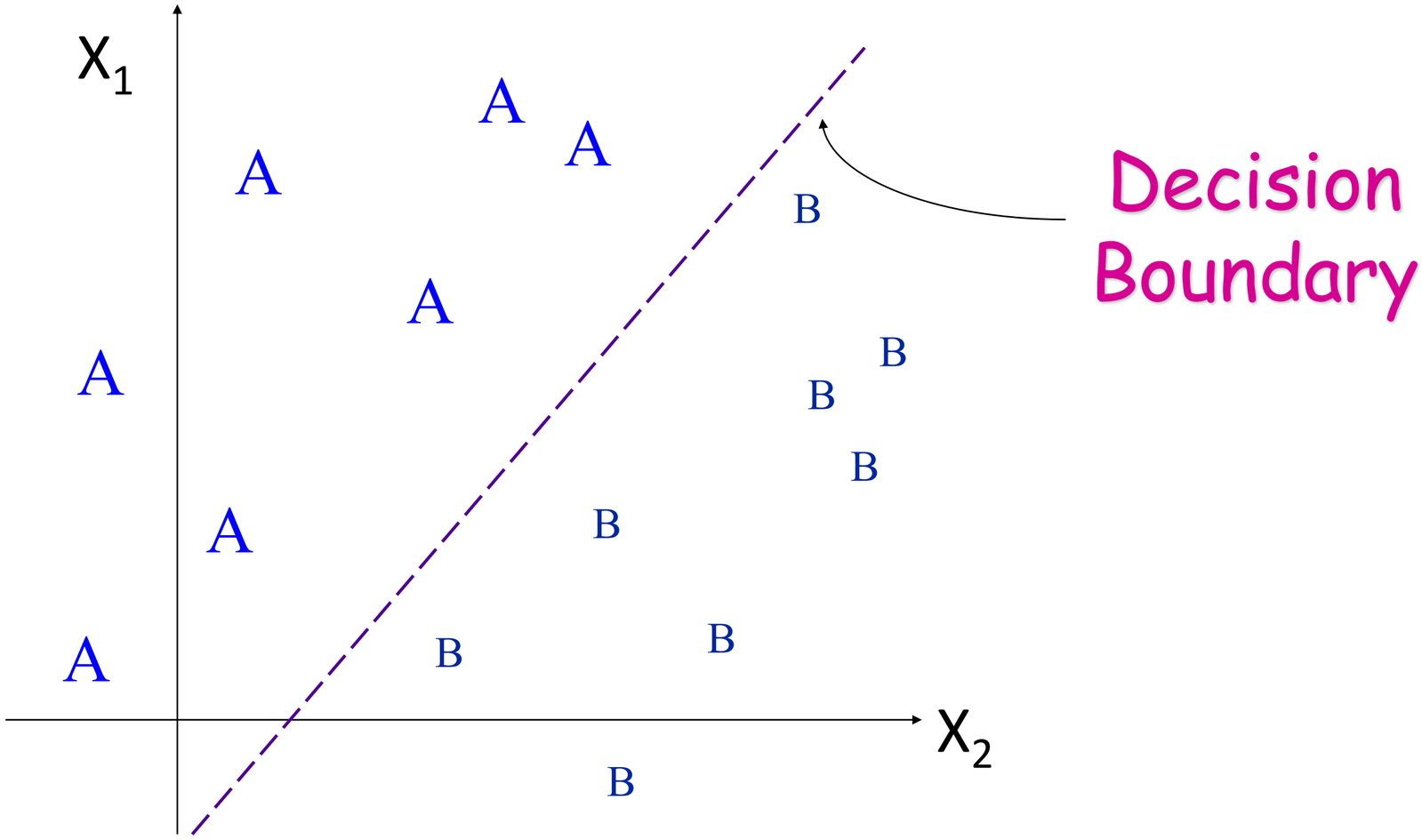


OR



XOR

Linear Separability



Hyperplane Partitions

- A single Perceptron (i.e. output unit) with connections from each input can perform, and learn, a linear separation.
- It splits the feature space into two regions, So it can separate linearly separable classes,
- Perceptron have a step function activation.
- A Single Perceptron = One Hyperplane
- This is why perceptrons work for AND and OR gates, both are linearly separable.
- Because XOR outputs look like this on a graph:
- Class 1: (0,1) and (1,0) → opposite corners
- Class 0: (0,0) and (1,1) → opposite corners
- No single straight line can separate them.
- To solve XOR, you need more than one hyperplane, which means:
  - Multiple perceptron
  - Hidden layers
  - A Multi-Layer Perceptron (MLP)
  - Backpropagation

Learning in Neural Networks

- Learn values of weights from I/O pairs
- Start with random weights
- Load training example’s input
- Observe computed input
- Modify weights to reduce difference
- Iterate over all training examples
- Terminate when weights stop changing OR when error is very small

Task: Implement a Perceptron on Logic Gates (Python)

- Implement the **perceptron training algorithm** (with bias).
- Train and test on **logic gates**: AND, OR, NAND, NOR, NOT.
- **Visualize** decision boundaries for 2D gates.
- Understand **linear separability** and **failure modes** (e.g., XOR).

Task Breakdown

- **Part A: Implement the Perceptron**
- Use a **single neuron** (no hidden layers).
- Activation: **hard limiter** (step function): y = 1 if net ≥ 0 else 0.
- Initialize **weights and bias randomly** (small values).
- Stop when either:
  - all samples correct in an **epoch**, or
  - you hit **max_epochs**.
- **Part B: Train & Test on Logic Gates**
- **AND OR NAND and NOR NOT** For each gate:
  - Train until convergence (or max_epochs)
  - Report **final weights/bias**, **epochs to converge**, **accuracy**
  - Plot **decision boundary** (2D gates)
- **Part C: Analysis**
- Compare convergence speed across gates.
- Show **decision boundaries** and interpret their slopes/offsets.
- Explain **why NOT, AND, OR, NAND, NOR are linearly separable**.
- **(Optional)** Try **XOR** and explain **why it fails** with a single perceptron.
- **Part D: Deliverables**
- A single **.ipynb** file including:
  - Implementation code
  - Training logs (epochs, mistakes per epoch)
  - Final metrics per gate
  - Plots for 2D gates
- A short **write-up** (5–10 bullets): observations, insights, issues.

Python Implementation

- https://medium.com/@robdelacruz/frank-rosenblatts-perceptron-19fcce9d627f