Artificial Neurons

AI with Deep Learning
EE4016

Prof. Lai-Man Po
Department of Electrical Engineering
City University of Hong Kong

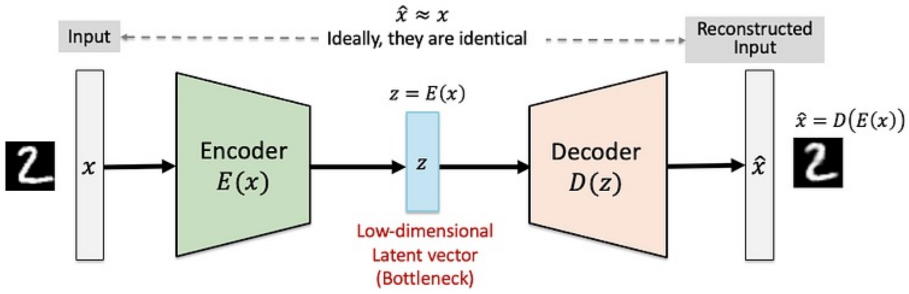
Week 2 Messages

- Recommended Technical Presentation for Group Project Development on "**Upscaling Images with Neural Networks**" by Geoffrey Litt
  - https://www.youtube.com/watch?v=RhUmSeko1ZE
  - This is a great technical presentation for students to learn about industry presentation styles and to identify the topic of your group project.
- Students, please form a **5-person** project team on or before **Jan 31, 2026**, and send your list of members to Lai-Man Po at eelmpo@cityu.edu.hk .
- On the other hand, students are strongly recommended to try Google Colab to practice programming skills using Python and PyTorch.
  - Colab Python Tutorial:
    - https://colab.research.google.com/drive/1MVBWrWYDNEitrAjBmp7F85_sSyXdhZH4
  - Deep Dive in PyTorch:
    - https://www.youtube.com/watch?v=A-rzknbjp5M&list=PLv8Cp2NvcY8D0SrHYWZWyOhV8r9eNierl&index=1

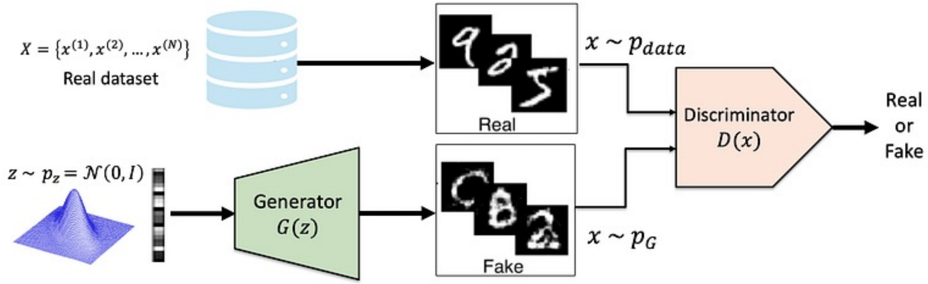
The Evolution and Rise of Diffusion Models in AI

- https://medium.com/@lmpo/from-words-to-pixels-the-evolution-and-rise-of-diffusion-models-in-ai-1053a95deabd

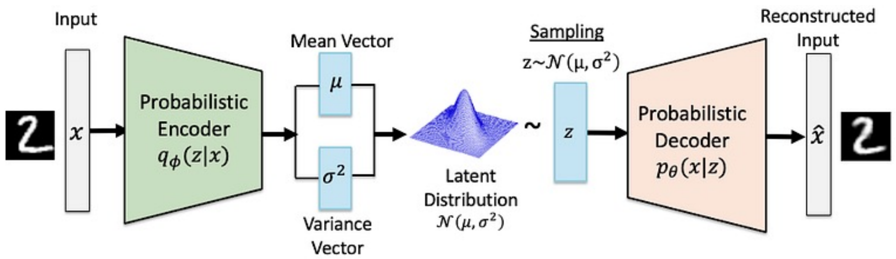
Autoencoders (1987)



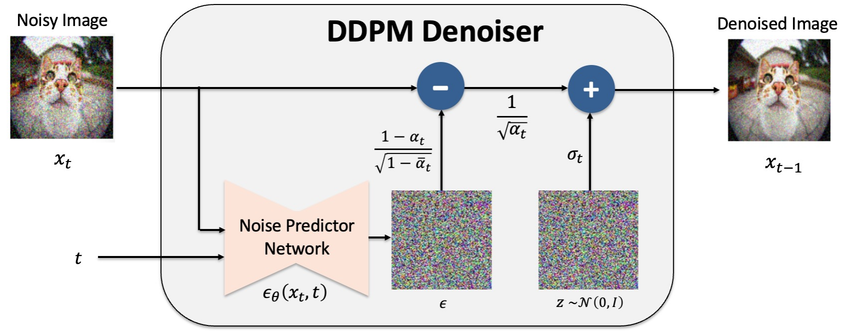
Generative Adversarial Networks (GANs, 2014)



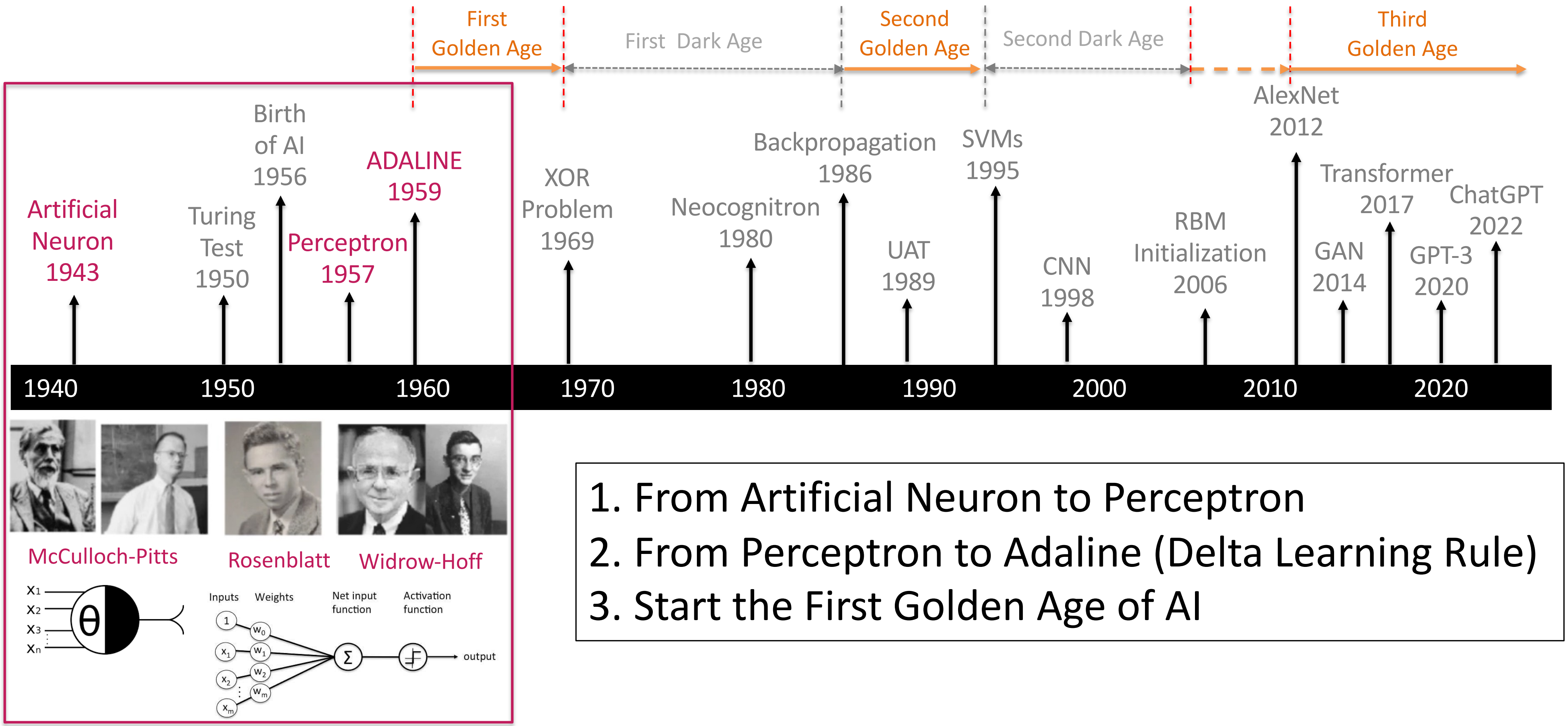
Variational Autoencoders (VAEs, 2013)



Diffusion Models (2015 – Present)



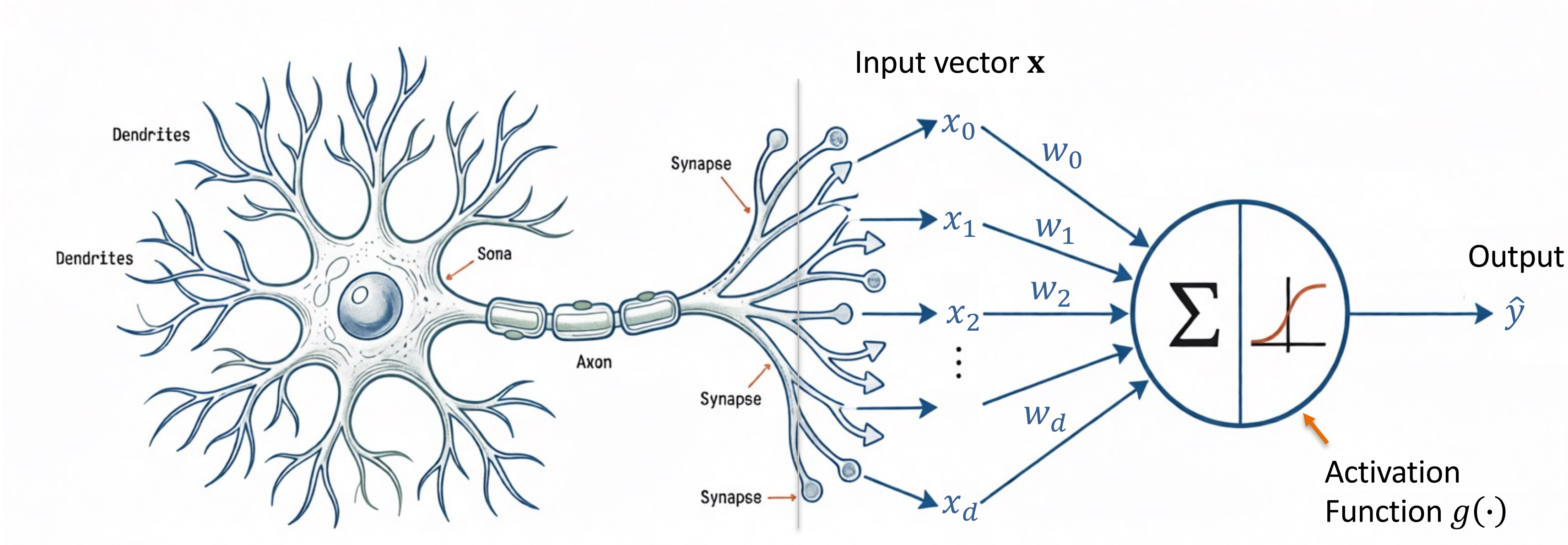
A Brief History AI with Deep Learning



1. From Artificial Neuron to Perceptron
2. From Perceptron to Adaline (Delta Learning Rule)
3. Start the First Golden Age of AI

From Logic Gates to Learning Machines

The Evolution of Artificial Neurons: A Technical Retrospective

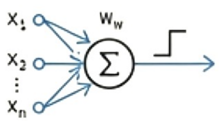


The Journey

1943: McCulloch-Pitts Neuron (Logic)

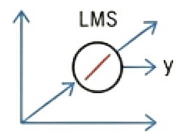
- First mathematical model of a neuron.
- Based on all-or-none logic.
- Introduced the concept of threshold logic units.
- Laid the foundation for digital computers.

1957: The Perceptron (Learning)



- Invented by Frank Rosenblatt.
- First trainable neural network.
- Utiized the perceptron learning rule for weight adjustment.
- Capable of learning linear classifications.

1959: ADALINE (Optimization)



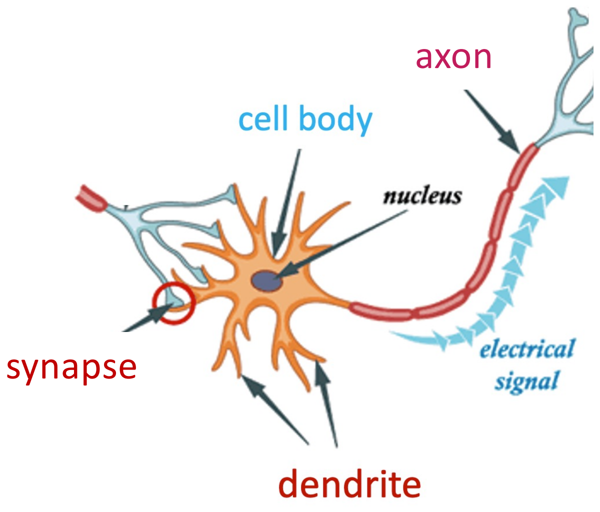
- Developed by Bernard Widrow and Marcian Hoff.
- Used the Delta Rule (LMS algorithm) for learning.
- Minimized mean squared error.
- Precursor to modern backpropagation.

McCulloch & Pitts Neuron Model (1943)

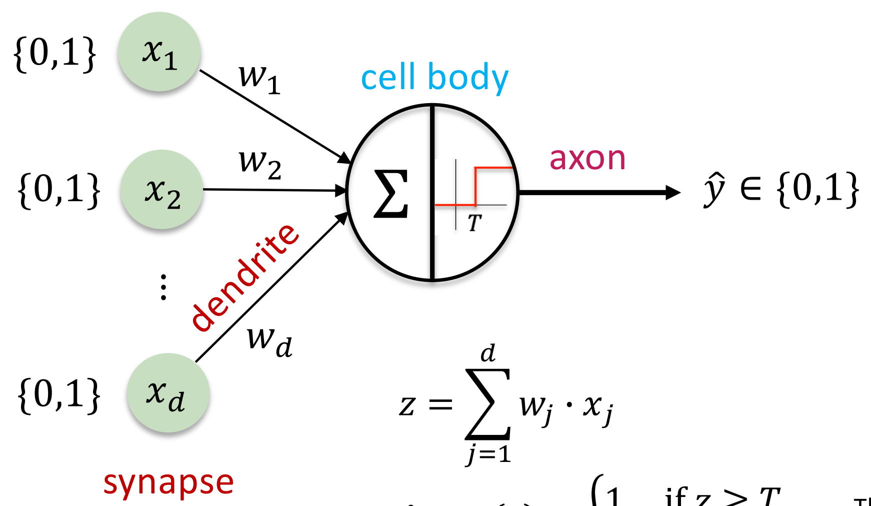
McCulloch & Pitts (MP) Neuron Model (1943)

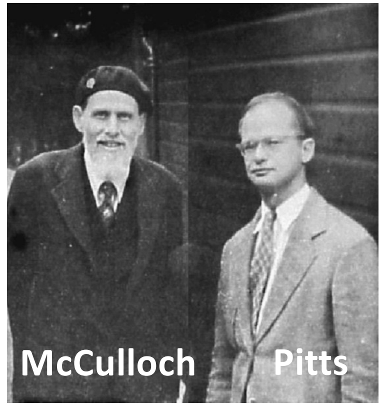
- **MP Neuron** is a highly simplified mathematical model **to mimic biologic neuron**.
- It takes binary inputs (0 or 1), computes their **weighted sum**, and generates a binary output (0 or 1) by applying a **threshold-based activation function**.

A Biologic Neuron



A McCulloch-Pitts Neuron





$$z = \sum_{j=1}^{d} w_j \cdot x_j$$

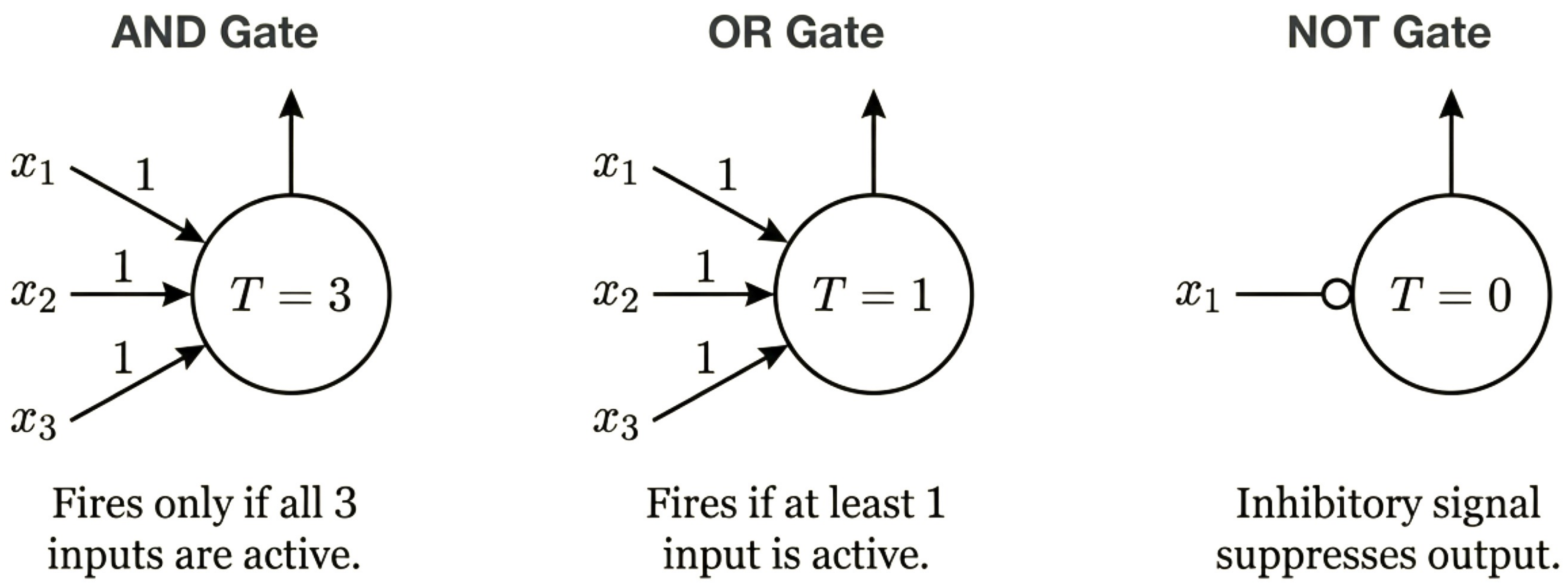
$$\hat{y} = g(z) = \begin{cases} 1 & \text{if } z \geq T \\ 0 & \text{otherwise} \end{cases}$$

Threshold-based Activation Function

McCulloch and Pitts: A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics, 1943

Proving Computation: Neural Logic Gates

- McCulloch and Pitts demonstrated that arranging these simple units could **replicate fundamental Boolean logic**, effectively proving neural networks could compute.

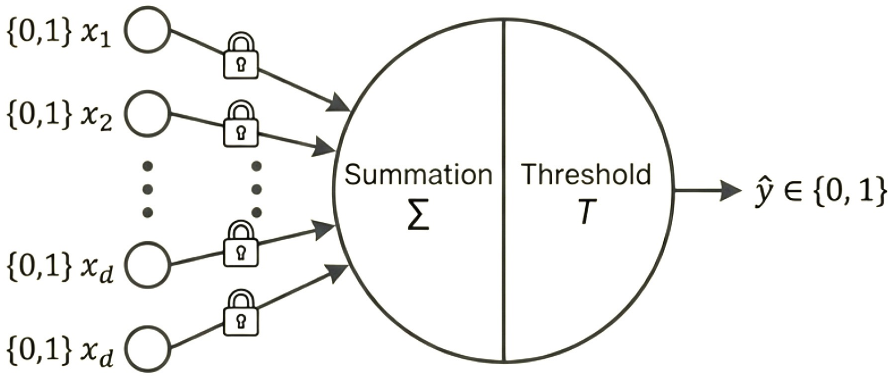


The 'Static' Bottleneck

The fatal flaw of the MP Neuron was the lack of adaptability.

- For every new logical task, a human operator had to manually calculate and set the weights and thresholds.
- The system was a hard-coded circuit, unable to learn from data or correct its own errors.

McCulloch-Pitts Neuron



NO LEARNING ALGORITHM.

No automated learning method was developed to identify these parameters for desired functions, which greatly restricted its practical applications.

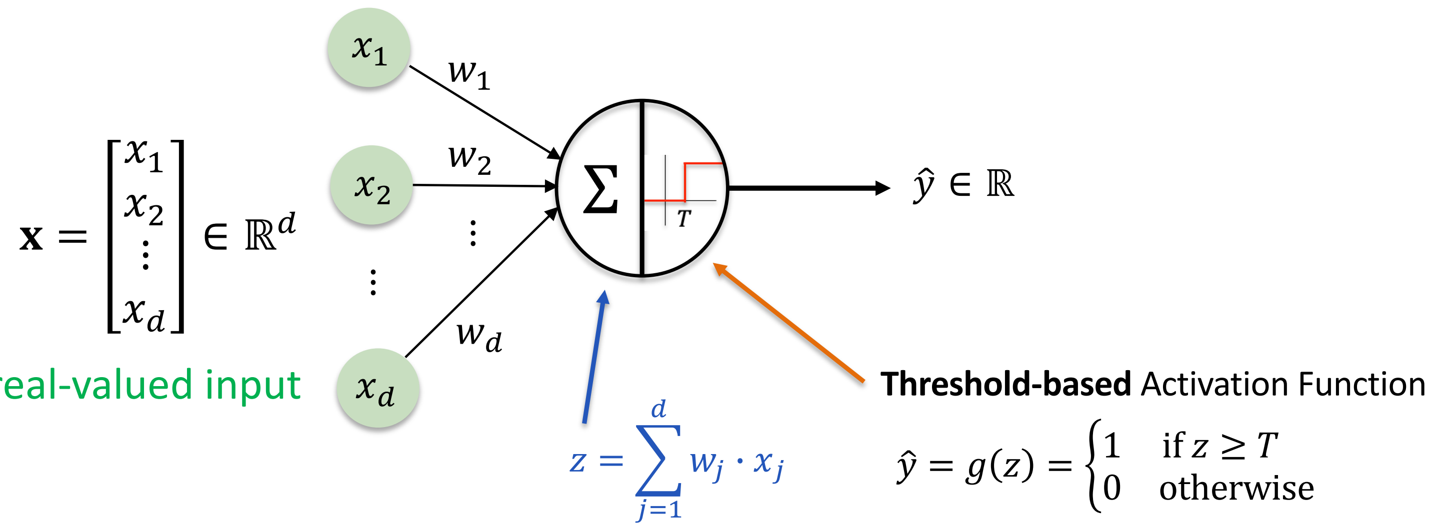
Rosenblatt's Perceptron

Frank Rosenblatt • Cornell Aeronautical Laboratory

(1957)

Rosenblatt's Perceptron Model (1957)

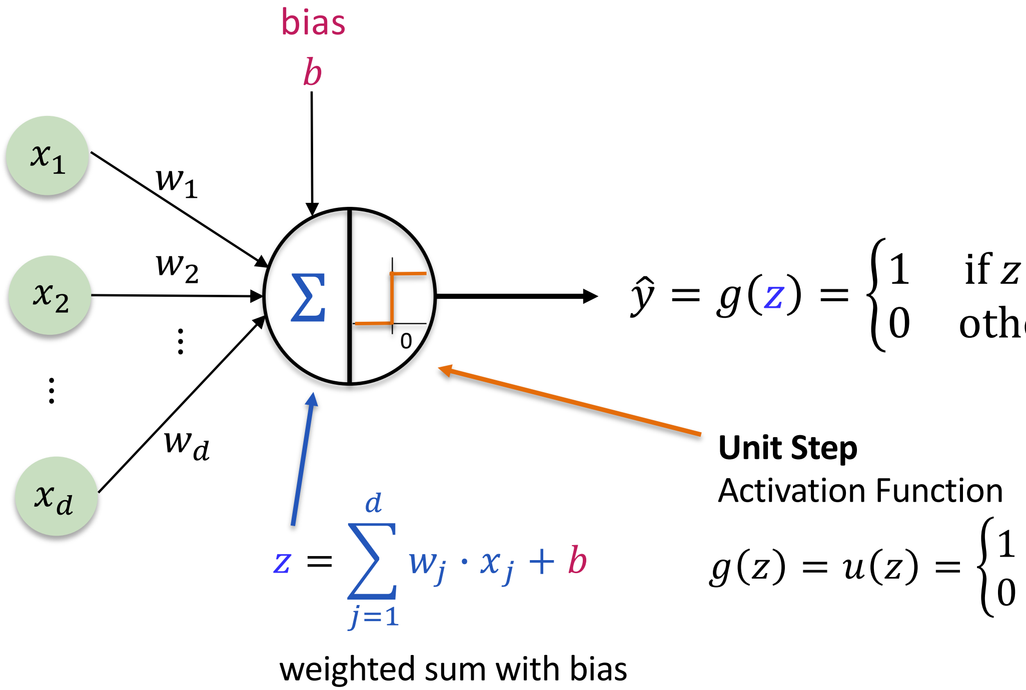
1. **The perceptron** is an advanced form of the MP Neuron, **capable of processing real-valued inputs $x_i \in \mathbb{R}$** and approximating a broad spectrum of complex functions.
2. Rosenblatt introduced the **perceptron learning rule**, a method for adjusting weights to reduce classification errors.



Frank Rosenblatt

Mathematical Reformulation: The Bias Term

- Use the **bias term ($b = -T$) to replace the threshold**, then the activation become a unit step function $u(z)$



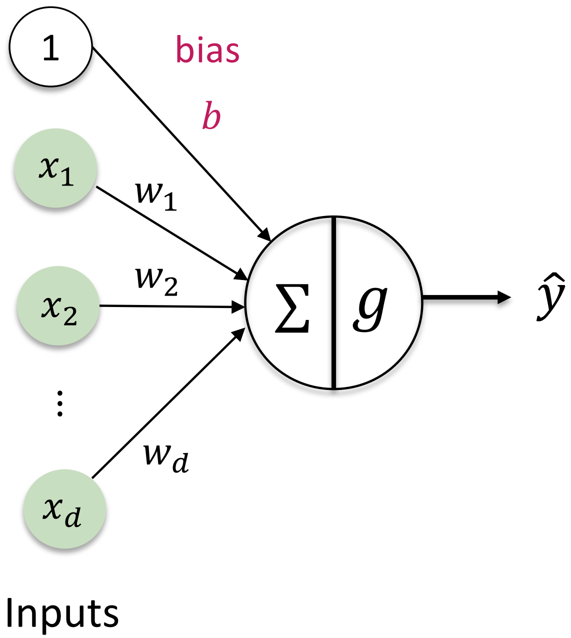
$$\hat{y} = g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$z = \sum_{j=1}^{d} w_j \cdot x_j + b$$

$$g(z) = u(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Perceptron Model Representation (1)

- By folding the threshold into the weights as a 'bias, we simplify the math. Instead of checking if the sum reaches a target, the neuron learns an internal offset.



$$\mathbf{x} = [x_1, x_2, \dots, x_d]^T \qquad \mathbf{w} = [w_1, w_2, \dots, w_d]^T$$

<u>Net Input</u>

$$z = \sum_{j=1}^{d} w_j\, x_j + b = \mathbf{w}^T\mathbf{x} + b \qquad \text{where } b = -T$$

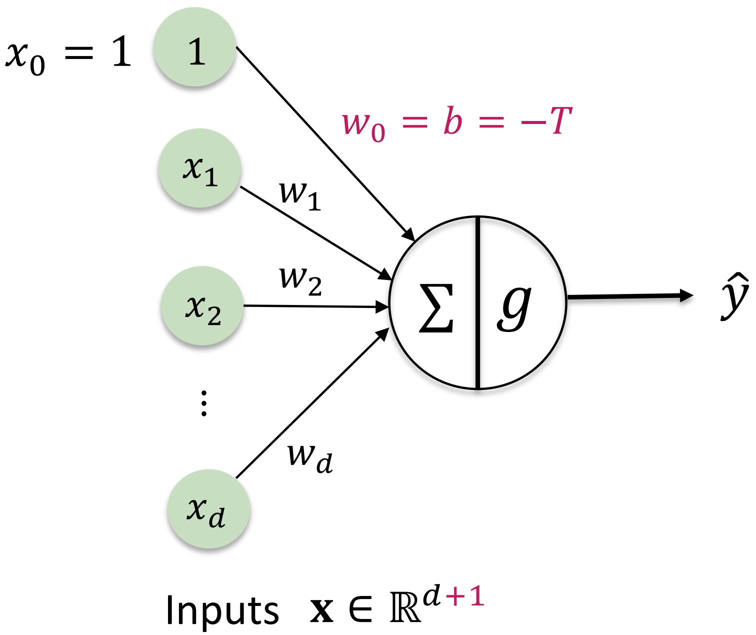
$$\hat{y} = g(z) = g(\mathbf{w}^T\mathbf{x} + b)$$

In **original Perception**, the activation function is a **Unit Step function** $u(z)$:

$$\hat{y} = u(z) = \begin{cases} 0, & \text{for } z < 0 \\ 1, & \text{for } z \geq 0 \end{cases}$$

Perceptron Model Representation (2)

- A more convenient notation is often used, where the bias term b is represented as w_0 , and an additional feature $x_0 = 1$ is prepended to each input vector.



$$\mathbf{x} = [1, x_1, x_2, \dots, x_d]^T$$

$$\mathbf{w} = [w_0, w_1, w_2, \dots, w_d]^T$$

<u>Net Input</u>

$$z = \sum_{j=0}^{d} w_j \, x_j = \mathbf{w}^T \mathbf{x}$$

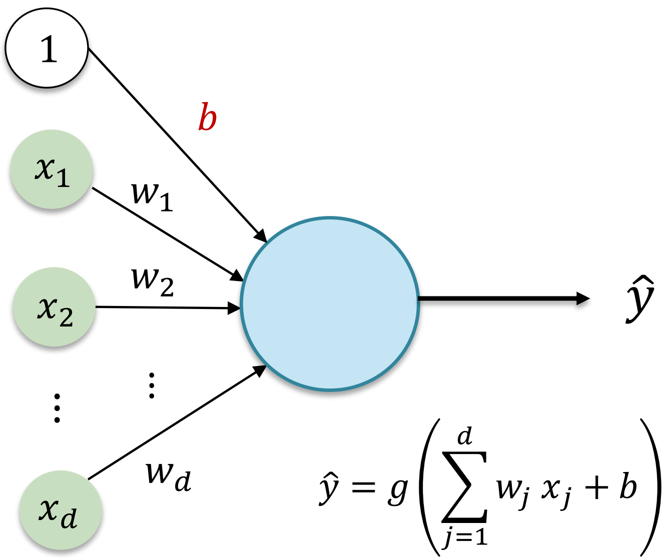
bias unit "included" as $w_0 = b$

$$\hat{y} = g(z) = g(\mathbf{w}^T \mathbf{x})$$

Perceptron Notations

$\mathbf{x} = [x_1, x_2, \dots, x_d]^T$

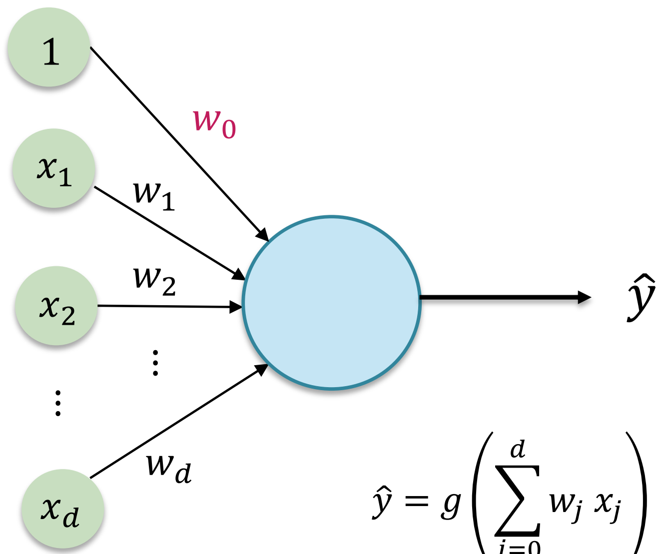
$\mathbf{w} = [w_1, w_2, \dots, w_d]^T \quad b$



$$\hat{y} = g\left(\sum_{j=1}^{d} w_j\, x_j + b\right) = g(\mathbf{w}^T\mathbf{x} + b)$$

$\mathbf{x} = [1, x_1, x_2, \dots, x_d]^T$

$\mathbf{w} = [w_0, w_1, w_2, \dots, w_d]^T$



$$\hat{y} = g\left(\sum_{j=0}^{d} w_j\, x_j\right) = g(\mathbf{w}^T\mathbf{x})$$

In modern neural networks, the activation functions can be Identify (linear) function: $g(z) = z$ for regression applications and Sigmoid function: $\sigma(z) = 1/(1 + e^{-z})$ for binary classification applications

Perceptron's Vector Representations

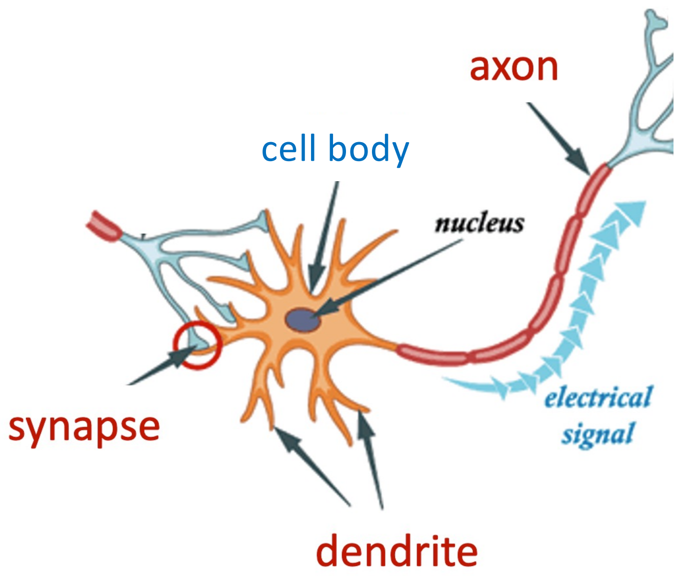
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \qquad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix} \qquad b$$

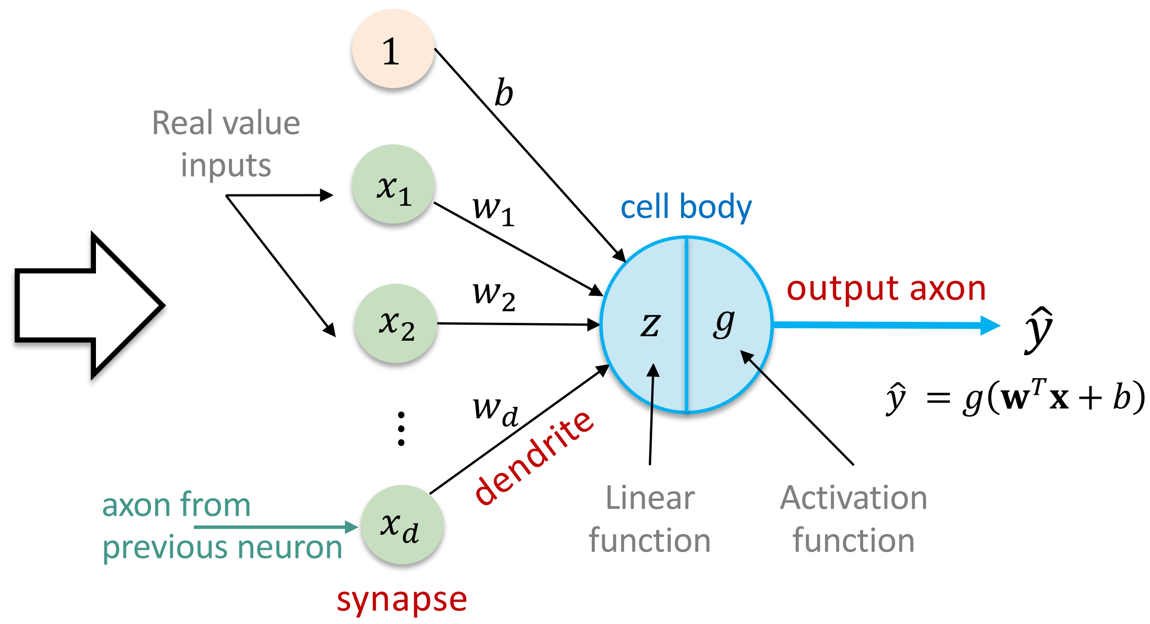
$$\hat{y} = g(\mathbf{w}^T\mathbf{x} + b) = g\left(\begin{bmatrix} w_1 & w_2 & \cdots & w_d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} + b \right) = g(w_1 x_1 + \cdots + w_d x_d + b)$$

$$\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} \qquad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix} \qquad w_0 = b \text{ and } x_0 = 1$$

$$\hat{y} = g(\mathbf{w}^T\mathbf{x}) = g\left(\begin{bmatrix} w_0 & w_1 & \cdots & w_d \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} \right) = g(w_0 + w_1 x_1 + \cdots + w_d x_d)$$

Biological Neuron vs Perceptron



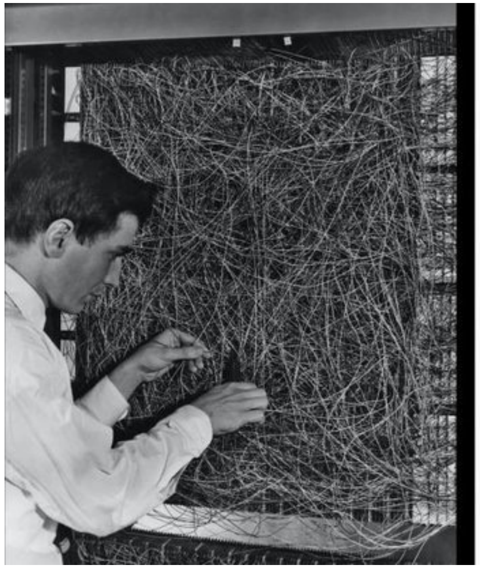


Biological Neuron

Artificial Neuron

Perceptron Pioneers: How Rosenblatt Launched Neural Networks

- Frank Rosenblatt’s perceptron was the first hardware implementation of a trainable neural network, igniting early enthusiasm for the potential of machine learning.
- Its adaptability enabled perceptrons to classify patterns in high-dimensional spaces, laying the groundwork for early image recognition systems.



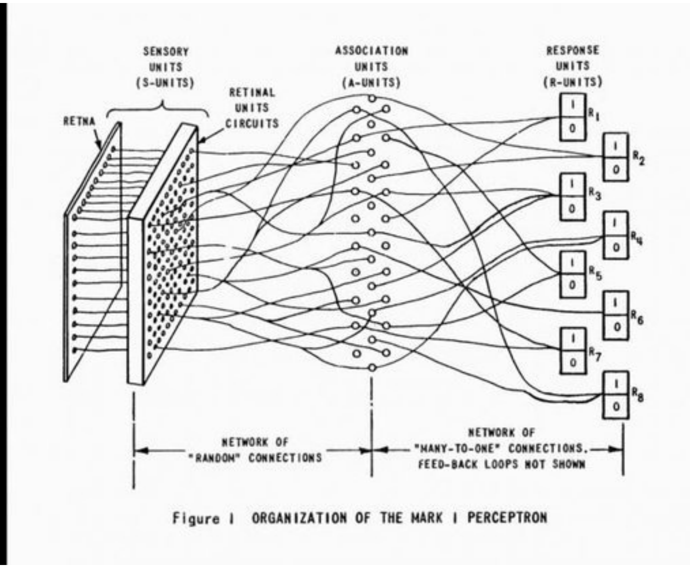
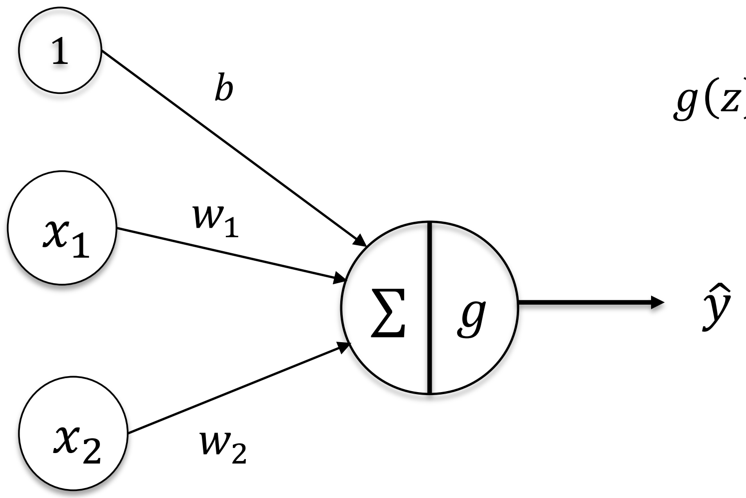


Figure I ORGANIZATION OF THE MARK I PERCEPTRON

Perceptron Exercise 1

- A perceptron is provided with weights $w_1 = 0.7$, $w_2 = 0.6$, and a bias $b = -1$. You are asked to compute the predicted output $\hat{y}$ for different input vectors $\mathbf{x} = [x_1, x_2]^T$: $[0, 0\,]^T$, $[0, 1\,]^T$, $[1, 0\,]^T$, $[1, 1\,]^T$. The perceptron's activation function is a binary step function $g(z) = u(z)$.
- Additionally, you need to determine the Boolean function represented by this perceptron



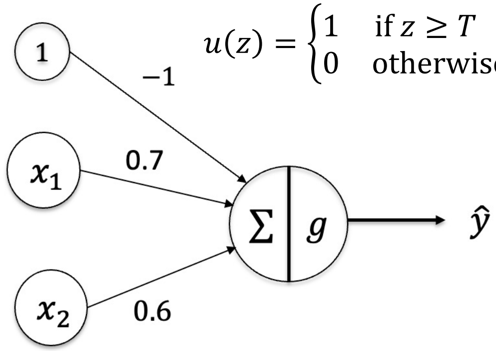
$$g(z) = u(z) = \begin{cases} 1 & \text{if } z \geq T \\ 0 & \text{otherwise} \end{cases}$$

Solution

The perceptron’s output can be computed using the following formula:

$$\hat{y} = g(\mathbf{w}^T\mathbf{x} + b) = u\left([w_1 \quad w_2]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b\right)$$

$$u(z) = \begin{cases} 1 & \text{if } z \geq T \\ 0 & \text{otherwise} \end{cases}$$



- For the input vector $x = [0, 0\,]^T$, the output is

$$\hat{y} = u\left([0.7 \quad 0.6]\begin{bmatrix} 0 \\ 0 \end{bmatrix} - 1\right) = u(-1) = 0$$

- For the input vector $x \ = [0, 1\,]^T$, the output is

$$\hat{y} = u\left([0.7 \quad 0.6]\begin{bmatrix} 0 \\ 1 \end{bmatrix} - 1\right) = u(0.6 - 1) = u(-0.4) = 0$$

- For the input vector $x \ = [1, 0\,]^T$, the output is

$$\hat{y} = u\left([0.7 \quad 0.6]\begin{bmatrix} 1 \\ 0 \end{bmatrix} - 1\right) = u(0.7 - 1) = u(-0.3) = 0$$

- For the input vector $x \ = [1, 1\,]^T$, the output is

$$\hat{y} = u\left([0.7 \quad 0.6]\begin{bmatrix} 1 \\ 1 \end{bmatrix} - 1\right) = u(0.7 + 0.6 - 1) = u(0.3) = 1$$

AND

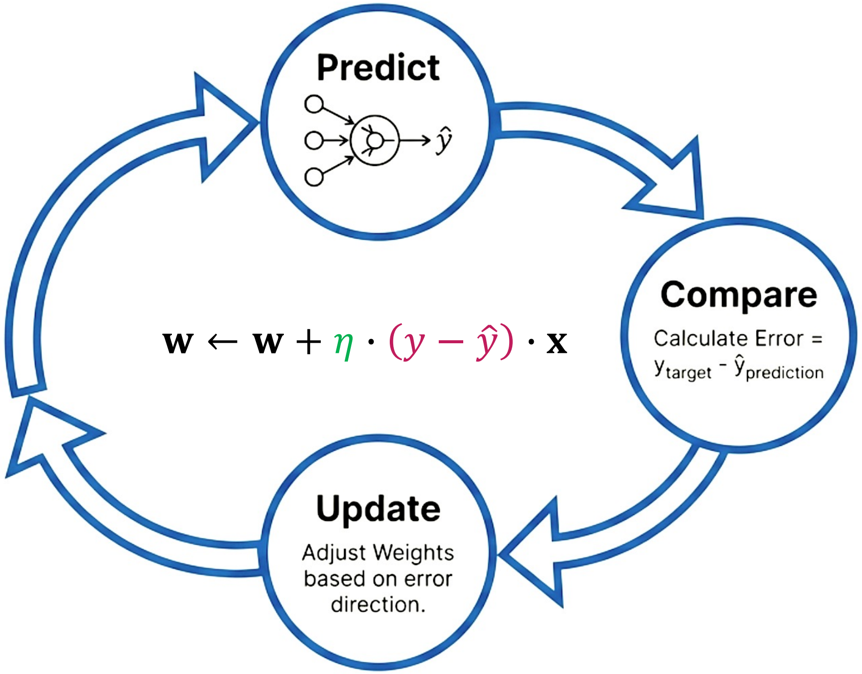
Inputs		Output
A	B	F
0	0	0
1	0	0
0	1	0
1	1	1

Based on the above results, this perceptron represents the Boolean **AND** gate.

Rosenblatt's Perceptron Learning (1957)

Rosenblatt also devised a **supervised learning algorithm** for the Perceptron, enabling it to learn from a training dataset $\mathcal{D} \coloneqq \left\{\left(\mathbf{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}$.

- Crucially, the Perceptron represented a major breakthrough by introducing the idea of learning through adaptive weight updates.
- Its learning rule adjusts the model's weights iteratively based on prediction errors, allowing it to solve problems that are linearly separable.
- As a result, the Perceptron can effectively discover a linear decision boundary to classify data points.



Perceptron Learning Rule

1. Initialization: Start with random weights w_j and a bias term as w_0.

2. Forward Pass: For each training example $\mathbf{x} = [1, x_1, x_2, \ldots, x_d]^T$ with label $y \in \{0,1\}$, compute the predicted output $\hat{y}$ as follows:

$$z = \sum_{j=0}^{d} w_j \, x_j \quad \text{and} \quad \hat{y} = u(z) = \begin{cases} 1 & \text{if } z \geq T \\ 0 & \text{otherwise} \end{cases}$$

3. Error Calculation: Calculate the error as the difference between the true label y and the predicted label $\hat{y}$:

$$error = y - \hat{y}$$

4. Weight Update: Update the weights and bias based on the error:

$$w_j = w_j + \eta \cdot error \cdot x_j$$

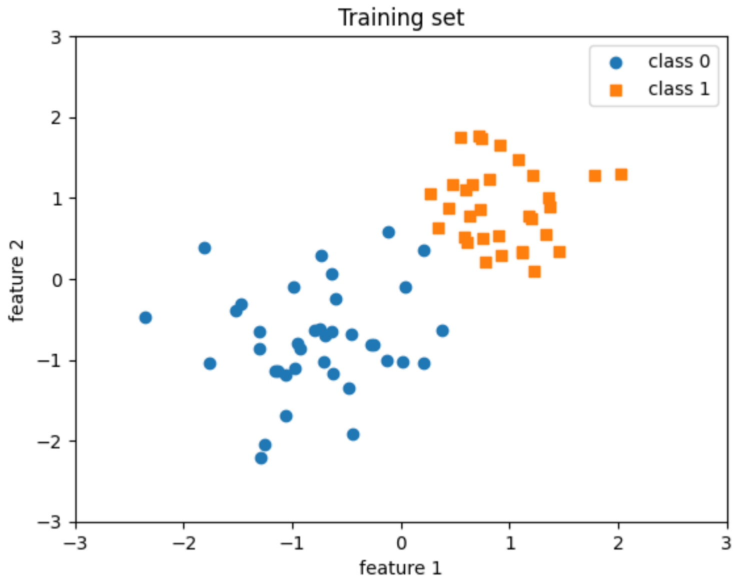
5. Iteration: Repeat steps 2–4 for a fixed number of iterations or until the weights converge.

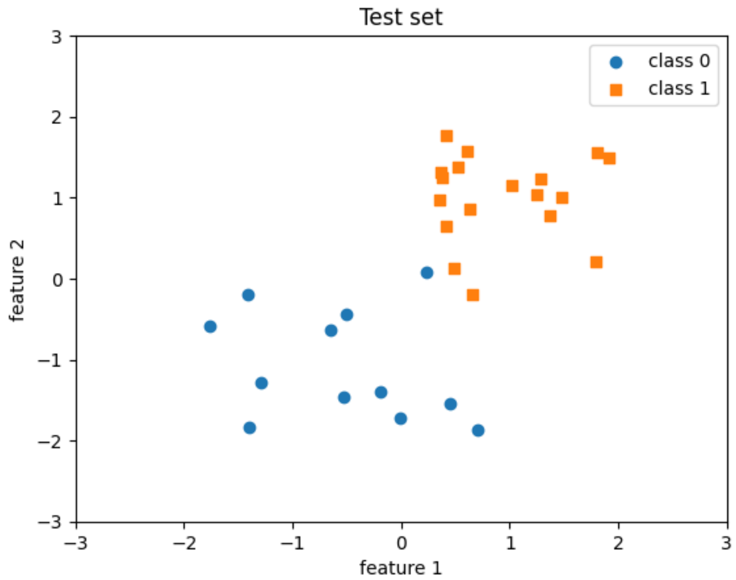
where η is the learning rate between 0 and 1.

This algorithm converge when all the training samples are classified correctly.

Perceptron Learning Example (PyTorch)

- **Colab:** https://colab.research.google.com/drive/1HGt_XwybylY1UMuQF3dHHYdqhHPZIo-5#scrollTo=me_F1WpPDX5e
  - In this example, a **linearly separable toy dataset** is used to training a Perceptron using **Rosenblatt's Perceptron Learning Algorithm**





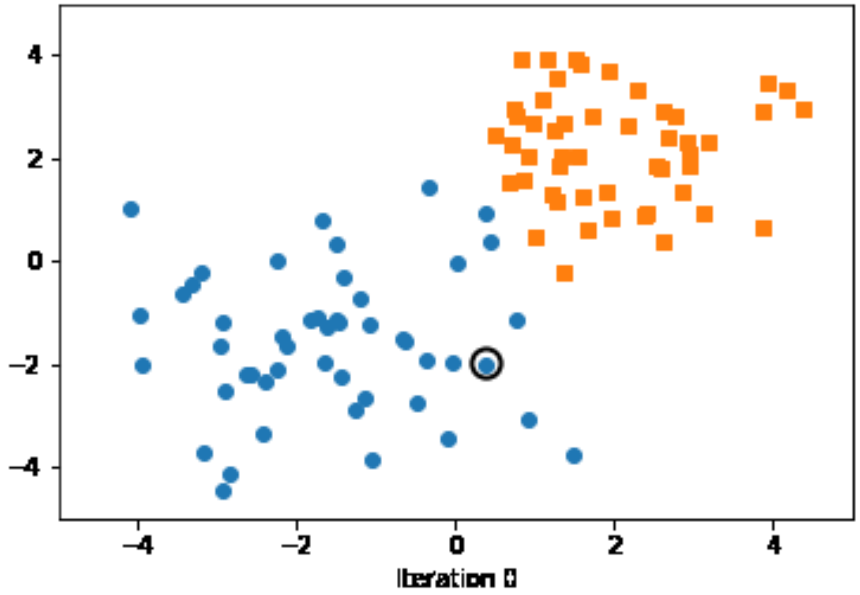
Define the Perceptron Model using PyTorch

```python
class Perceptron():
    def __init__(self, num_features):
        self.num_features = num_features
        self.weights = torch.zeros(num_features, 1, dtype=torch.float32)
        self.bias = torch.zeros(1, dtype=torch.float32)

        # Placeholder vectors so they don't need to be recreated each time
        self.ones = torch.ones(1)
        self.zeros = torch.zeros(1)

    def forward(self, x):
        linear = torch.mm(x, self.weights) + self.bias
        predictions = torch.where(linear > 0., self.ones, self.zeros)
        return predictions

    def backward(self, x, y):
        predictions = self.forward(x)
        errors = y - predictions
        return errors

    def train(self, x, y, epochs):
        for e in range(epochs):
            for i in range(y.shape[0]):
                # use view because backward expects a matrix (i.e., 2D tensor)
                errors = self.backward(x[i].reshape(1, self.num_features), y[i]).reshape(-1)
                self.weights += (errors * x[i]).reshape(self.num_features, 1)
                self.bias += errors

    def evaluate(self, x, y):
        predictions = self.forward(x).reshape(-1)
        accuracy = torch.sum(predictions == y).float() / y.shape[0]
        return accuracy
```

Training the Perceptron

```
ppn = Perceptron(num_features=2)

X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)

ppn.train(X_train_tensor, y_train_tensor, epochs=5)

print('Model parameters:')
print('  Weights: %s' % ppn.weights)
print('  Bias: %s' % ppn.bias)
```

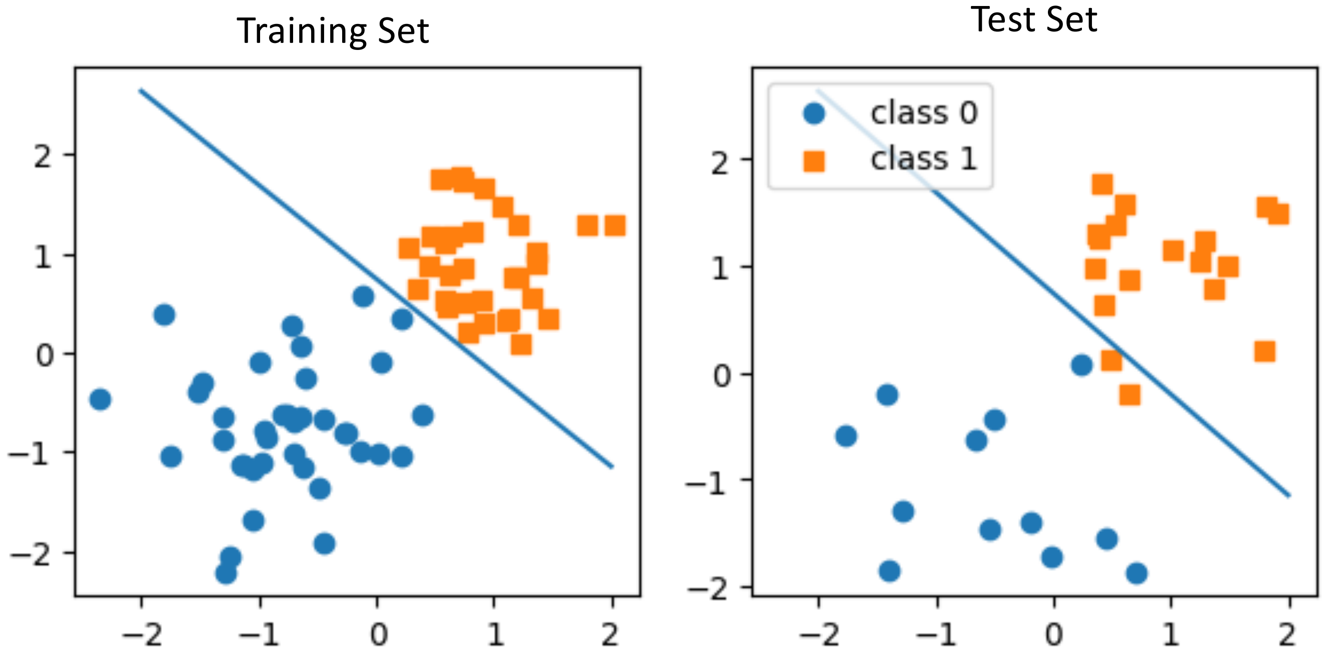
```
Model parameters:
  Weights: tensor([[1.2734],
        [1.3464]])
  Bias: tensor([-1.])
```



Evaluating the Model

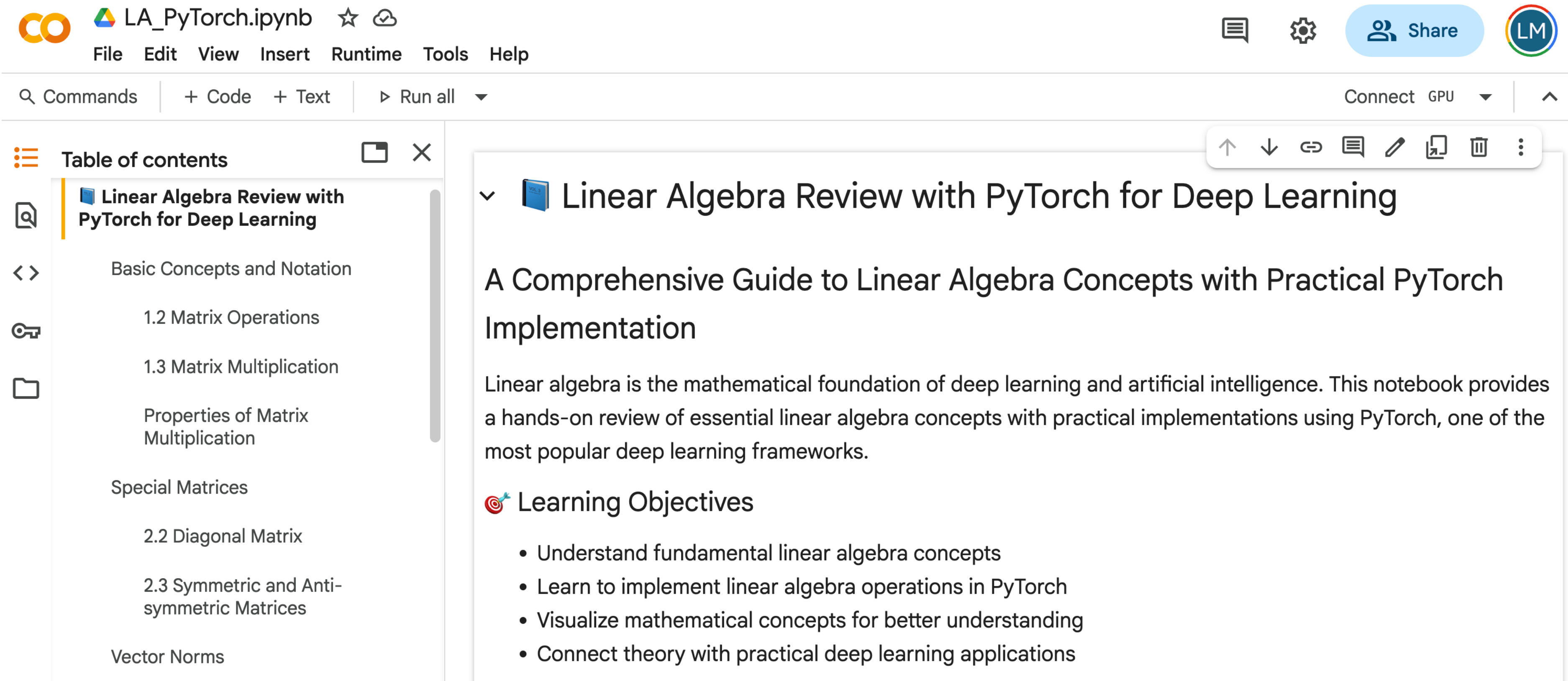
```
[ ] X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
    y_test_tensor = torch.tensor(y_test, dtype=torch.float32)


    test_acc = ppn.evaluate(X_test_tensor, y_test_tensor)
    print('Test set accuracy: %.2f%%' % (test_acc*100))
```

```
Test set accuracy: 93.33%
```



Python Tutorial with Google Colab

https://colab.research.google.com/drive/1MVBWrWYDNEitrAjBmp7F85_sSyXdhZH4



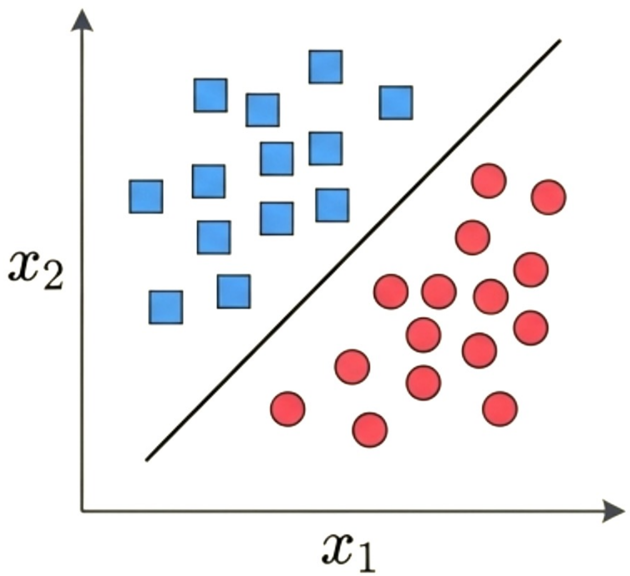
1957 News about the Rosenblatt's Perceptron

- In the 1950s, Rosenblatt predicted to the New York Times that Perceptrons would be capable of:
  - Recognizing individuals and addressing them by name
  - Translating speech from one language to another, either verbally or in written form
- These ambitious claims, reminiscent of 2022's AI breakthrough of ChatGPT, generated significant excitement and anticipation for the potential of artificial intelligence.

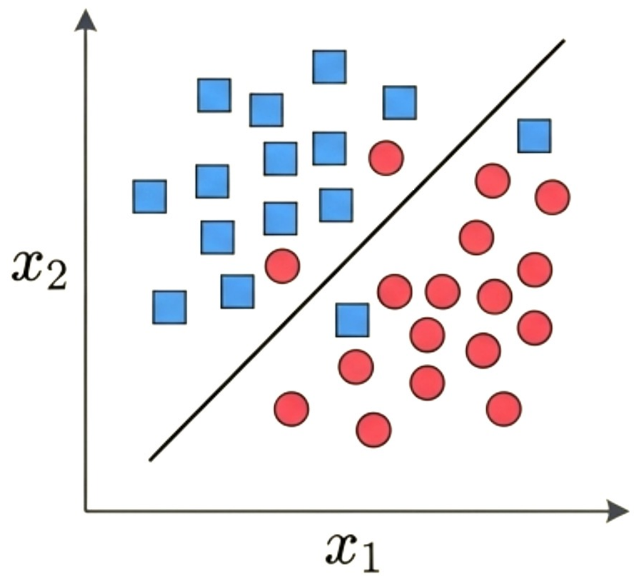
https://www.youtube.com/watch?v=cNxadbrN_aI

The Linear Trap: Limitations of the Step Function

Linearly Separable



Non-Linearly Separable

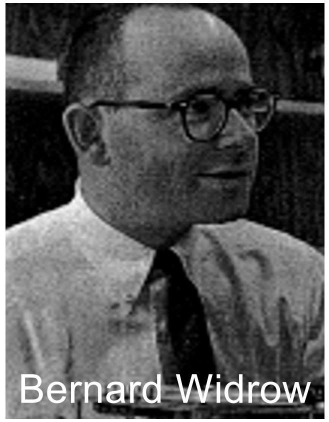


1. Rosenblatt's Convergence Theorem guarantees a solution only for linearly separable data. For non-linearly separable, the Perceptron learning will oscillate infinitely.
2. Furthermore, because the Step Function is discrete (jumping from 0 to 1), the error signal provides no information about "how close" the prediction was.

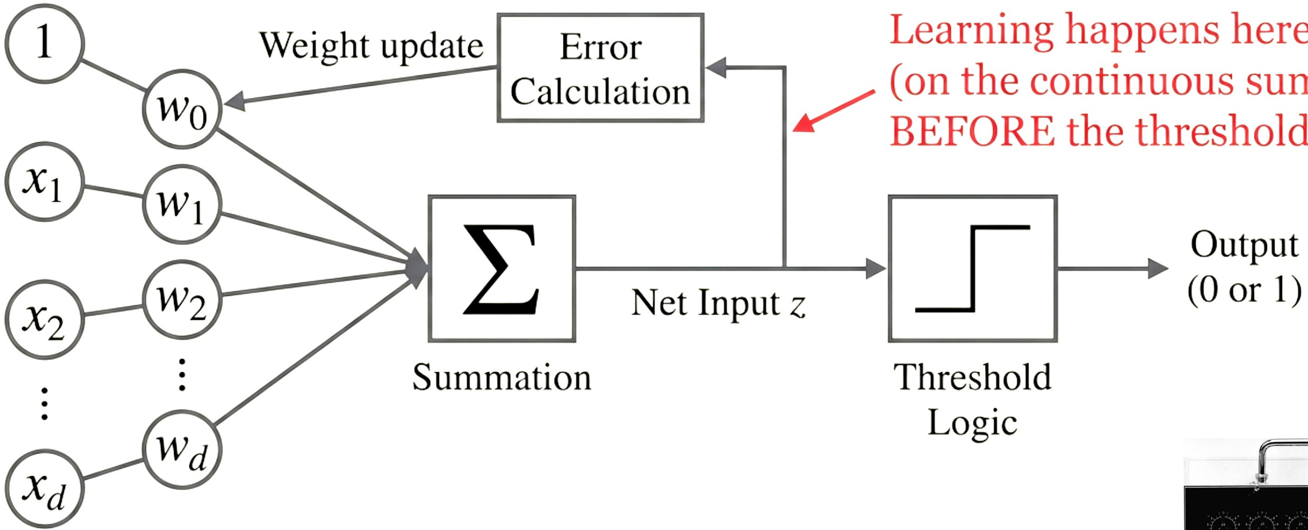
ADALINE (aka Delta Rule Learning) (1959)

1959: ADALINE (Adaptive Linear Neron)

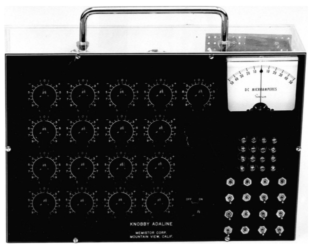
Widrow & Hoff • Stanford University



$$error = y - g(\mathbf{w}^T\mathbf{x}) = y - \mathbf{w}^T\mathbf{x} = y - z$$



Learning happens here (on the continuous sum z), BEFORE the threshold.



https://www.youtube.com/watch?v=skfNlwEbqck

ADALINE (or Delta Learning Rule)

1. Initialization: Start with random weights w_j and a bias term as w_0.

2. Forward Pass: For each training example $\mathbf{x} = [1, x_1, x_2, \dots, x_d]^{\mathrm{T}}$ with label $y \in \{0,1\}$, compute the predicted output $\hat{y}$ as follows:

$$z = \sum_{j=1}^{d} w_j\, x_j \quad \text{and} \quad \hat{y} = u(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

3. Error Calculation: Calculate the error as the difference between the true label y and the net input z :

$$error = y - z$$

4. Weight Update: Update the weights and bias based on the error:

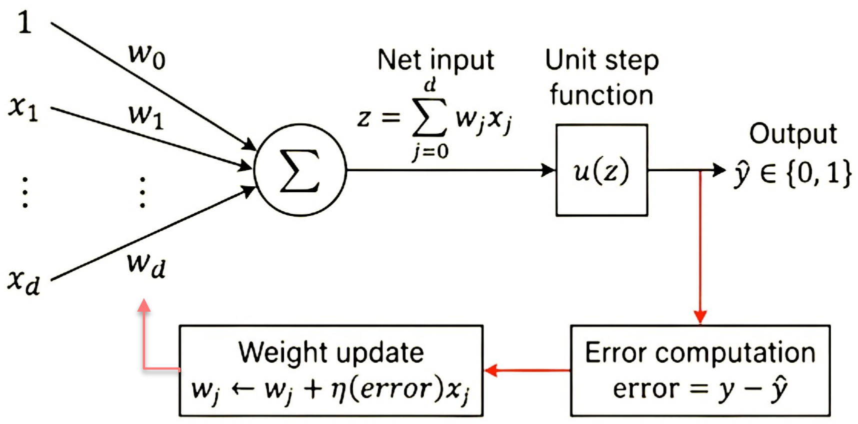
$$w_j = w_j + \eta \cdot error \cdot x_j$$

5. Iteration: Repeat steps 2–4 for a fixed number of iterations or until the weights converge.

ADALINE enables smoother weight adjustment and **convergence on non-linearly separable datasets.**

The Shift to Continuous Error

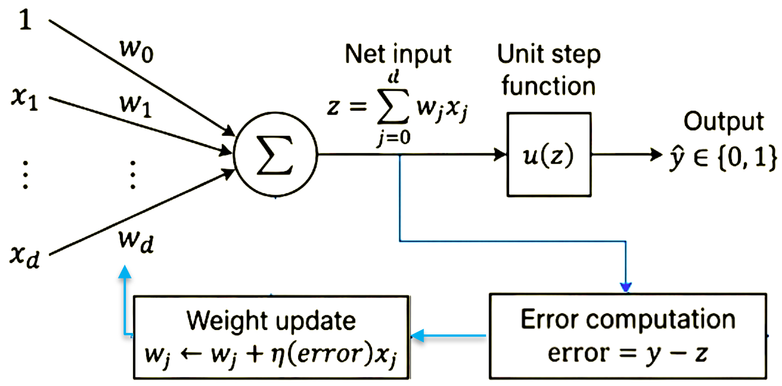
Perceptron Training Loop



error $\in \{-1, 0, 1\}$ (Discrete)

Coarse adjustments. Hard to optime.

ADALINE Training Loop



error $\in \mathbb{R}$ (Continuous Real Value)

Precise adjustments. Minimizes magnitude of error.

ADALINE asks "How much were we wrong?", not just "Were we wrong?"

Stochastic Gradient Descent Algorithm

- **SGD Algorithm**

  1. Initialize $\mathbf{w} = 0 \in \mathbb{R}^{d+1}$
  2. For every training epoch:

     A. For every $\left(\mathbf{x}^{(i)}, y^{(i)}\right) \in \mathcal{D}$:

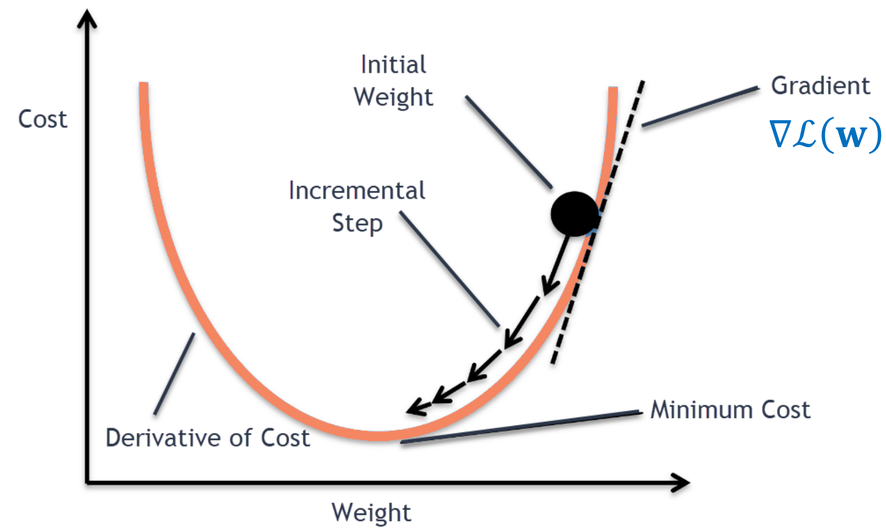
        a) $\hat{y}^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)}$

        b) $\nabla \mathcal{L}(\mathbf{w}) = \left(\hat{y}^{(i)} - y^{(i)}\right)\mathbf{x}^{(i)}$

        c) $\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla \mathcal{L}(\mathbf{w})$

Learning rate $0 < \eta \leq 1$

Gradient
(Slope of the cost function)



Move along the negative direction of the slope of the cost function $\mathcal{L}(\mathbf{w})$ until we find a minimum value

SGD using MSE Cost Function

- We assume the error of the model is measured by **Mean Square Error** (MSE). Then, the **cost function** $\mathcal{L}(\mathbf{w})$ can be defined as

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 = \frac{1}{2}\sum_{i=1}^{N}\left(y^{(i)} - f\left(\mathbf{x}^{(i)}\right)\right)^{\mathbf{2}}$$

  where $\hat{y}^{(i)}$ is the predicted output and $y^{(i)}$ is the target output (label) of a training example $\mathbf{x}^{(i)}$ in a training dataset $\mathcal{D} := \left\{\left(\mathbf{x}^{(1)}, y^{(1)}\right), \left(\mathbf{x}^{(2)}, y^{(2)}\right), \dots, \left(\mathbf{x}^{(N)}, y^{(N)}\right)\right\}$

- Based on this cost function, we need to find the gradient for updating the weights

https://towardsdatascience.com/from-the-perceptron-to-adaline-1730e33d41c5

How to find the Gradient $\nabla\mathcal{L}(\mathbf{w})$?

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Mean Squared Error (MSE) loss often scaled by factor ½ for convenience

(Note that the activation function is the identity function in Delta Learning Rule: $g(z) = z \Rightarrow g'(z) = 1$)

$$\nabla\mathcal{L}(\mathbf{w}) = \frac{\partial\mathcal{L}(\mathbf{w})}{\partial w_j} = \frac{\partial}{\partial w_j}\left(\frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left(y^{(i)} - \hat{y}^{(i)}\right)^2\right) = \frac{1}{2N}\frac{\partial}{\partial w_j}\sum_{i=1}^{N}\left(y^{(i)} - g\left(\mathbf{w}^T x^{(i)}\right)\right)^2$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - g\left(\mathbf{w}^T\mathbf{x}^{(i)}\right)\right)\frac{\partial}{\partial w_j}\left(y^{(i)} - g\left(\mathbf{w}^T\mathbf{x}^{(i)}\right)\right) = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - g\left(\mathbf{w}^T\mathbf{x}^{(i)}\right)\right)\left(-\frac{\partial g}{\partial\left(\mathbf{w}^T\mathbf{x}^{(i)}\right)}\cdot\frac{\partial}{\partial w_j}\left(\mathbf{w}^T\mathbf{x}^{(i)}\right)\right)$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - g\left(\mathbf{w}^T\mathbf{x}^{(i)}\right)\right)\left(-\frac{\partial}{\partial w_j}\left(\mathbf{w}^T\mathbf{x}^{(i)}\right)\right) = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - g\left(\mathbf{w}^T\mathbf{x}^{(i)}\right)\right)\left(-x_j^{(i)}\right) = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{y}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

Vector Gradients:

$$\nabla\mathcal{L}(\mathbf{w}) = \frac{\partial\mathcal{L}(\mathbf{w})}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{y}^{(i)} - y^{(i)}\right)\mathbf{x}^{(i)}$$

https://towardsdatascience.com/from-the-perceptron-to-adaline-1730e33d41c5

SGD Weight Update Rule

- In SGD, the model parameters w are **updated for each sample** $\left(\mathbf{x}^{(i)}, y^{(i)}\right) \in \mathcal{D}$.
- The gradient of the cost function $\mathcal{L}(\mathbf{w})$ is defined with $n = 1$:

$$\nabla\mathcal{L}(\mathbf{w}) = \frac{\partial\mathcal{L}(\mathbf{w})}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{y}^{(i)} - y^{(i)}\right)\mathbf{x}^{(i)} = \left(\hat{y}^{(i)} - y^{(i)}\right)\mathbf{x}^{(i)} = \left(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)}\right)\mathbf{x}^{(i)}$$

- The parameters update at iteration can be expressed as

$$w_j \leftarrow w_j - \eta\frac{\partial\mathcal{L}(\mathbf{w})}{\partial w_j} = w_j - \eta\left(\hat{y}^{(i)} - y^{(i)}\right)\mathbf{x}_j^{(i)}$$

Vector Representation:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\nabla\mathcal{L}(\mathbf{w}) = \mathbf{w} - \eta\left(\hat{y}^{(i)} - y^{(i)}\right)x^{(i)}$$

https://towardsdatascience.com/from-the-perceptron-to-adaline-1730e33d41c5

SGD vs ADALINE Rule

- **SGD Algorithm**
  1. Initialize $\mathbf{w} = 0 \in \mathbb{R}^{d+1}$
  2. For every training epoch:
     A. For every $\left(\mathbf{x}^{(i)}, y^{(i)}\right) \in \mathcal{D}$:
        a) $\nabla\mathcal{L}(\mathbf{w}) = \left(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)}\right)\mathbf{x}^{(i)}$
        b) $\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla\mathcal{L}(\mathbf{w})$

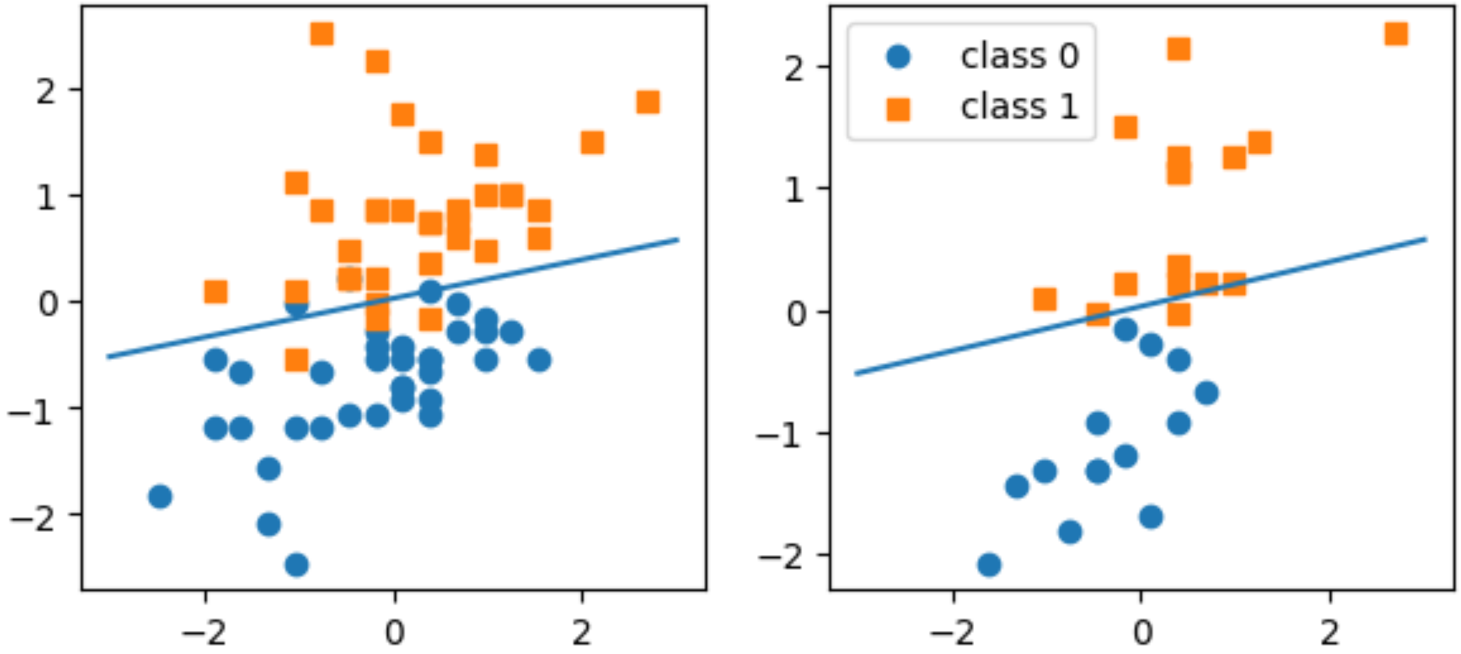
- **ADALINE Learning Rule**
  1. Initialize $\mathbf{w} = 0 \in \mathbb{R}^{d+1}$
  2. For every training epoch:
     A. For every $\left(\mathbf{x}^{(i)}, y^{(i)}\right) \in \mathcal{D}$:
        a) $error = y^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)}$
        b) $\mathbf{w} \leftarrow \mathbf{w} + \eta \cdot error \cdot \mathbf{x}^{(i)}$

$$-\eta \cdot \nabla\mathcal{L}(\mathbf{w}) = -\eta\left(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)}\right)\mathbf{x}^{(i)} = \eta\left(y^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)}\right)\mathbf{x}^{(i)} = \eta \cdot error \cdot \mathbf{x}^{(i)}$$

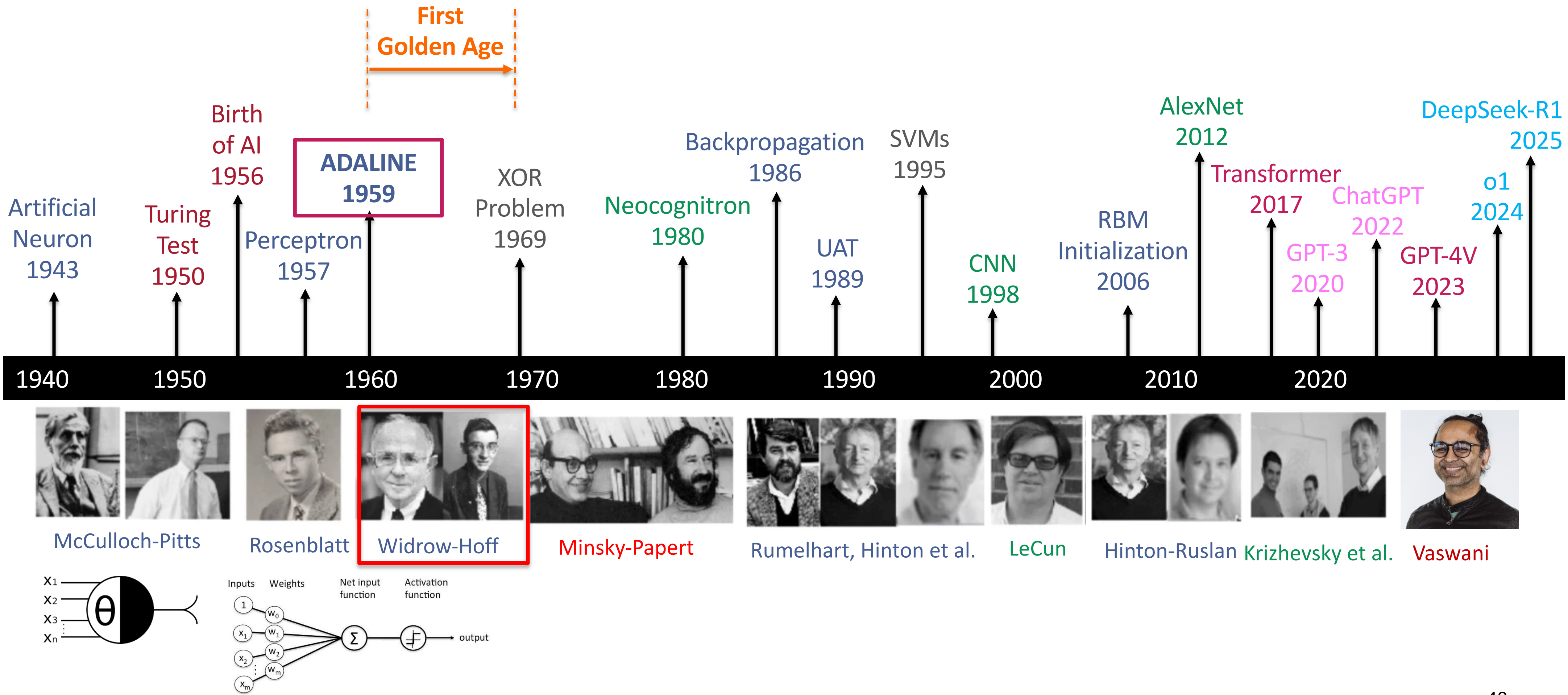
The ADALINE Learning Rule and Gradient Descent (GD) share similarities in their objective of minimizing an error function, the ADALINE Learning Rule can be considered as a special case of GD when applied to a single training example at a time, which is called Stochastic Gradient Descent (SGD).

ADALINE in Python

- **Colab:** https://colab.research.google.com/drive/1riUZ2DmV_3s4ngdHImmjMMX6kyoC3dYL?usp=sharing
- In this notebook, ADALINE is implemented in Python, which is based on the source code of Stat453.



ADALINE Open Up the 1st Golden Age



Artificial Neuron Evolution Summary

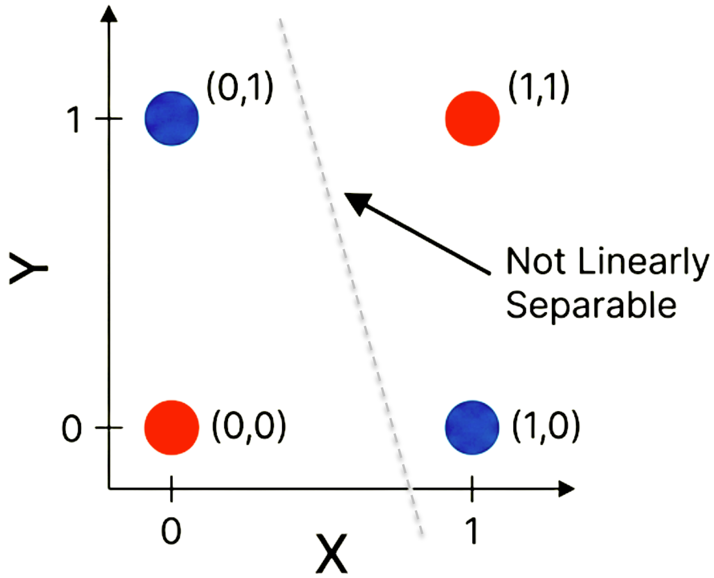
Model	Inputs	Activation	Learning Mechanism
MP Neuron (1943)	Binary {0. 1}	Threshold	None (Fixed Weights)
Perceptron (1957)	Real $\mathbb{R}$	Unit Step	Perceptron Rule (Discrete Error)
ADALINE (1959)	Real $\mathbb{R}$	Linear (for learning)	LMS / Gradient Descent (Continuous Error)

The XOR Problem (1969)

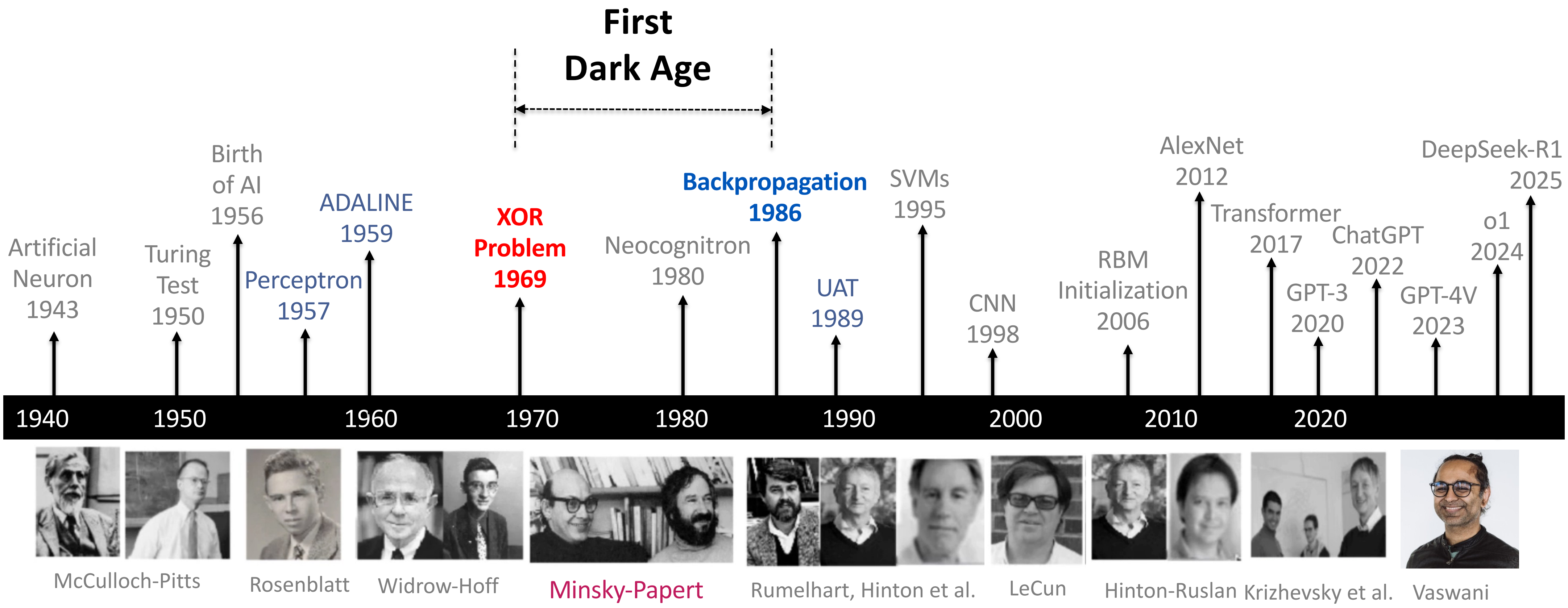
- In 1969, **Minsky** and **Papert** proved that **single-layer neurons cannot solve non-linear problems** like XOR. This led to the first 'AI Winter'.

Truth Table for XOR

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0



XOR Problem Started the First Dark Age Winter (1969-1986)



Stacking Neurons to Bend Boundaries

- The Solution to XOR: **Hidden layers** allow the network to combine multiple linear decisions **to create non-linear shapes**.