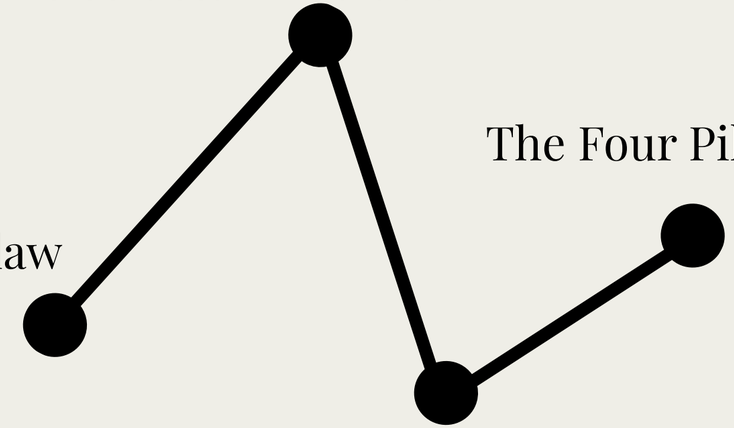
THE CALCULUS OF COGNITION:

From Biological Neuron to Backpropagation

MLP: The Structural Solution

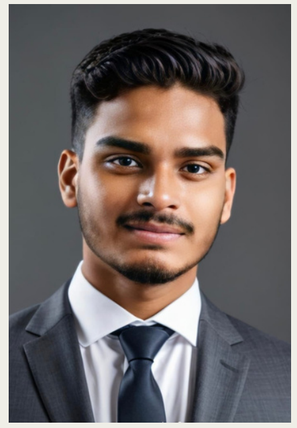
The Four Pillars of Deep Learning

The Perceptron's Fatal Flaw



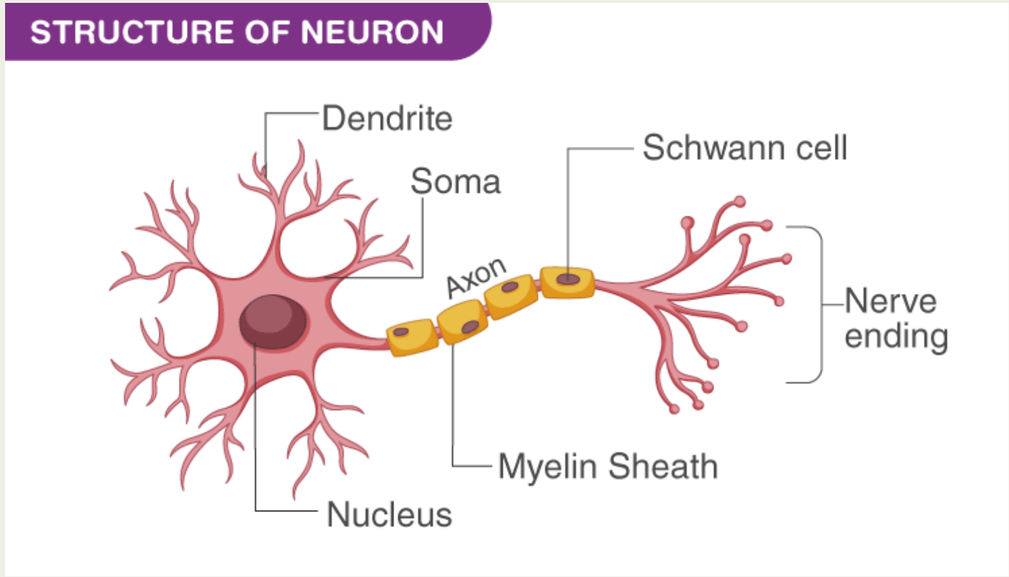
Backpropagation's Calculus Trick

GiGi Koneti

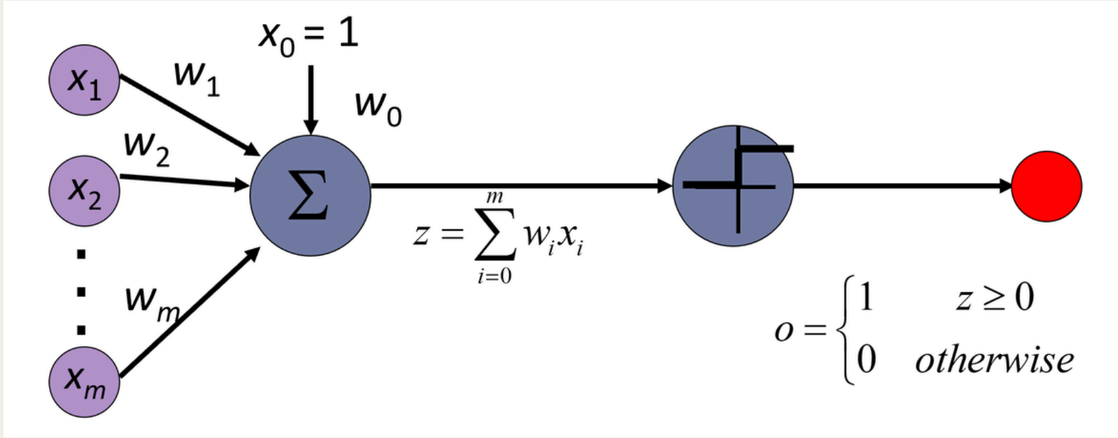


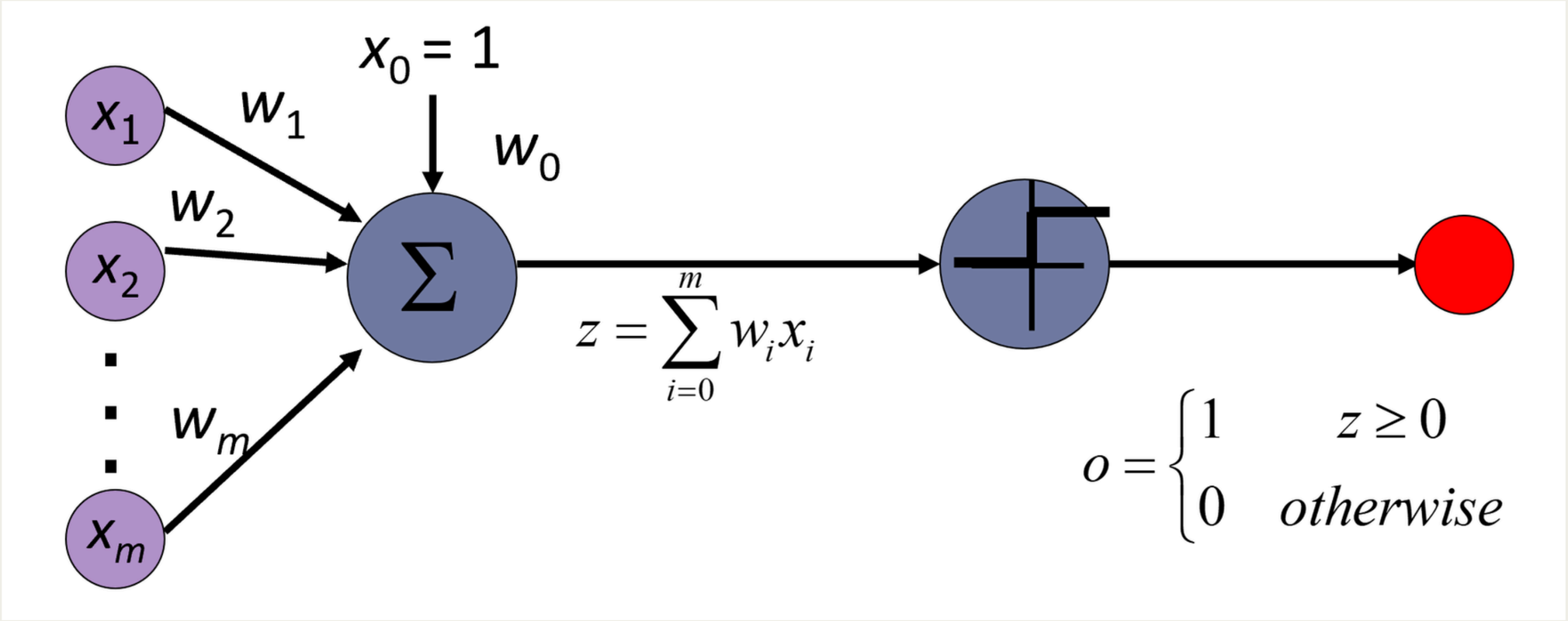
1.1 FRANK ROSENBLATT AND THE PERCEPTRON (1957)

'The perceptron is a machine that is expected to learn to recognize its input signals and correctly classify them, much as a human being learns to recognize different objects in the environment.'



"The Perceptron is a direct functional analogue of the biological neuron, where weighted inputs are summed to determine if a threshold is crossed, mimicking the neuron's all-or-nothing firing mechanism."

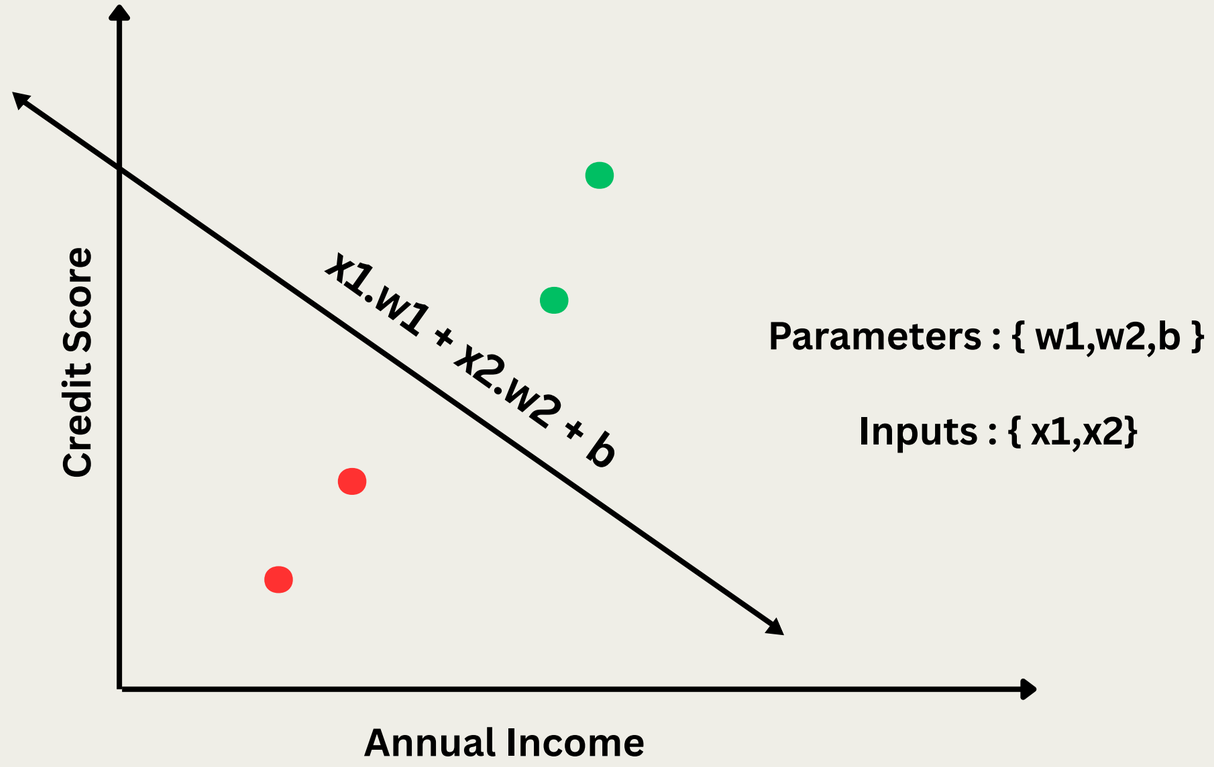



$x_0 = 1$
x_1
w_1
w_0
w_2
x_2
Σ
w_m
x_m
$z = \sum_{i=0}^{m} w_i x_i$
$o = \begin{cases} 1 & z \geq 0 \\ 0 & otherwise \end{cases}$

z = x1.w1 + x2.w2 + b

GRAPHICAL REPRESENTATION OF LINEAR SEPARABLE DATA

Label	Annual Income	Credit Score	Loan Aprroved
🟢	60	720	1
🔴	35	580	0
🟢	95	780	1
🔴	42	640	0



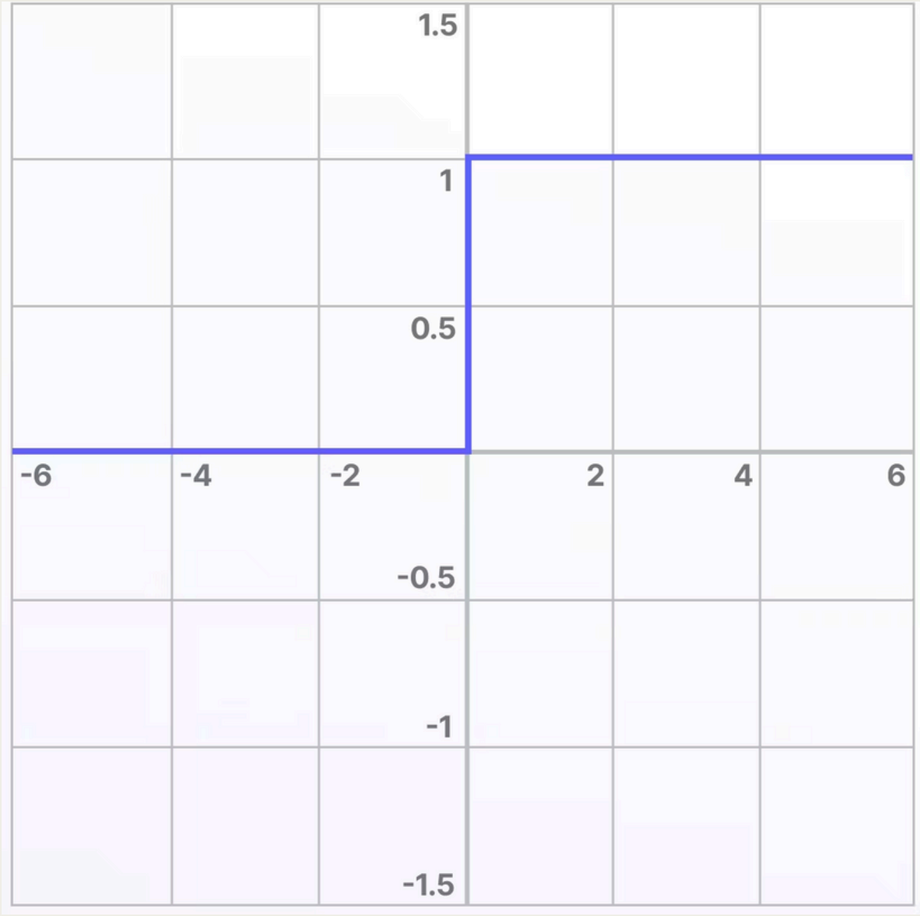
Parameters : { w1,w2,b }

Inputs : { x1,x2}

$$z = [\, b \; w1 \; w2 \,] \cdot \begin{bmatrix} 1 \\ x1 \\ x2 \end{bmatrix} = x1.w1 + x2.w2 + b$$

ACTIVATION FUNCTION (BINARY STEP)

$$f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \end{cases}$$



How the Parameters are Learned : The Update Rule

we check if a point is on the correct side. When the model makes a mistake (like with the red dot in your image), we need to "nudge" the line. This nudge is calculated using an update rule.

The most basic update rule looks like this for each weight:

$$w_{new} = w_{old} + \alpha \cdot (y - \hat{y}) \cdot x$$

Let's break that down:

w_new: The new, adjusted weight.

w_old: The weight we had before the update.

α (alpha): This is the Learning Rate. It's a small number (e.g., 0.01) that controls how big of a "nudge" we give the line. It prevents us from overcorrecting.

x: The input value for that weight (e.g., the point's 'Annual Income' value).

(y - y^): This is the error.

y is the actual correct label (e.g., 1 for green, 0 for red).

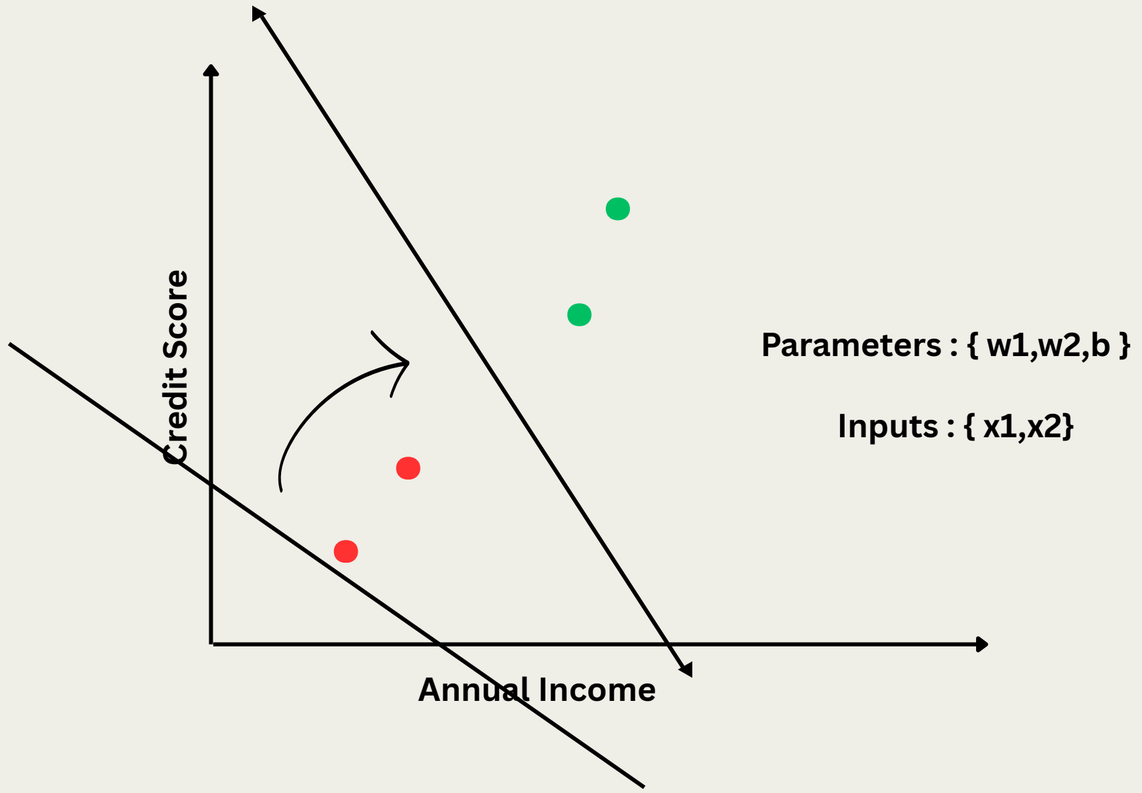
y^ (y-hat) is the predicted label that the model currently gives.

This simple formula is powerful. If the prediction is correct (y=y^), the error is zero, and the weights do not change. If the prediction is wrong, the error term is non-zero, and the weights get updated, moving the line slightly to better classify that point.

The same logic applies to the bias term, b:

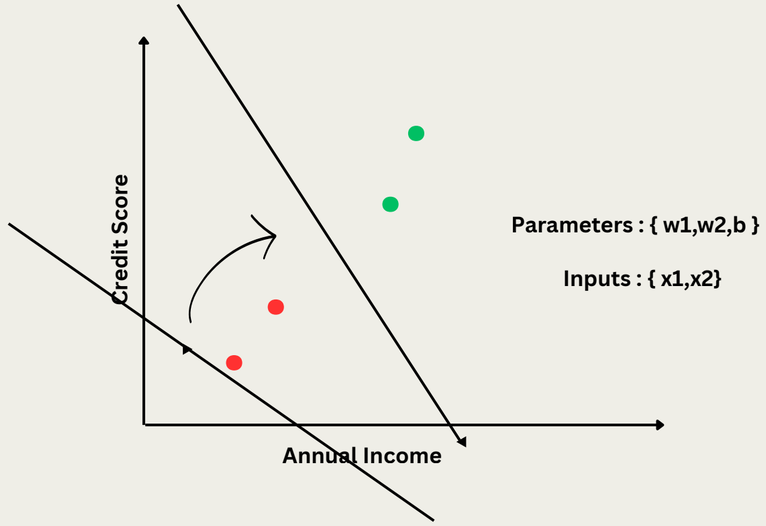
$$b_{new} = b_{old} + \alpha \cdot (y - \hat{y})$$

This process of checking a point and updating the parameters is repeated for every data point over many cycles (called epochs). With each small correction, the line gets progressively better until it correctly separates all the groups.



```
for i in range(epochs):
        Randomly select a point (X_i)

            // Calculate prediction (y_hat) for the point
            // Calculate the weighted sum and apply the step function
            z = dot_product(weights, X_i)
            y_hat = f(z)
            // Case 1: A negative point is on the positive side (z >= 0)
            if X_i belongs to Negative class AND z >= 0 {
                w_new = w_old - Alpha * X_i
            }
            // Case 2: A positive point is on the negative side (z < 0)
            else if X_i belongs to Positive class AND z < 0 {
                w_new = w_old + Alpha * X_i
            }
```

epochs = 1000, Alpha = 0.01:



Actual (y)	Predicted (ŷ)	Error (y - ŷ)	Result
1	1	0	Correctly classified
0	0	0	Correctly classified
1	0	**1**	**Wrongly classified**
0	1	**-1**	**Wrongly classified**

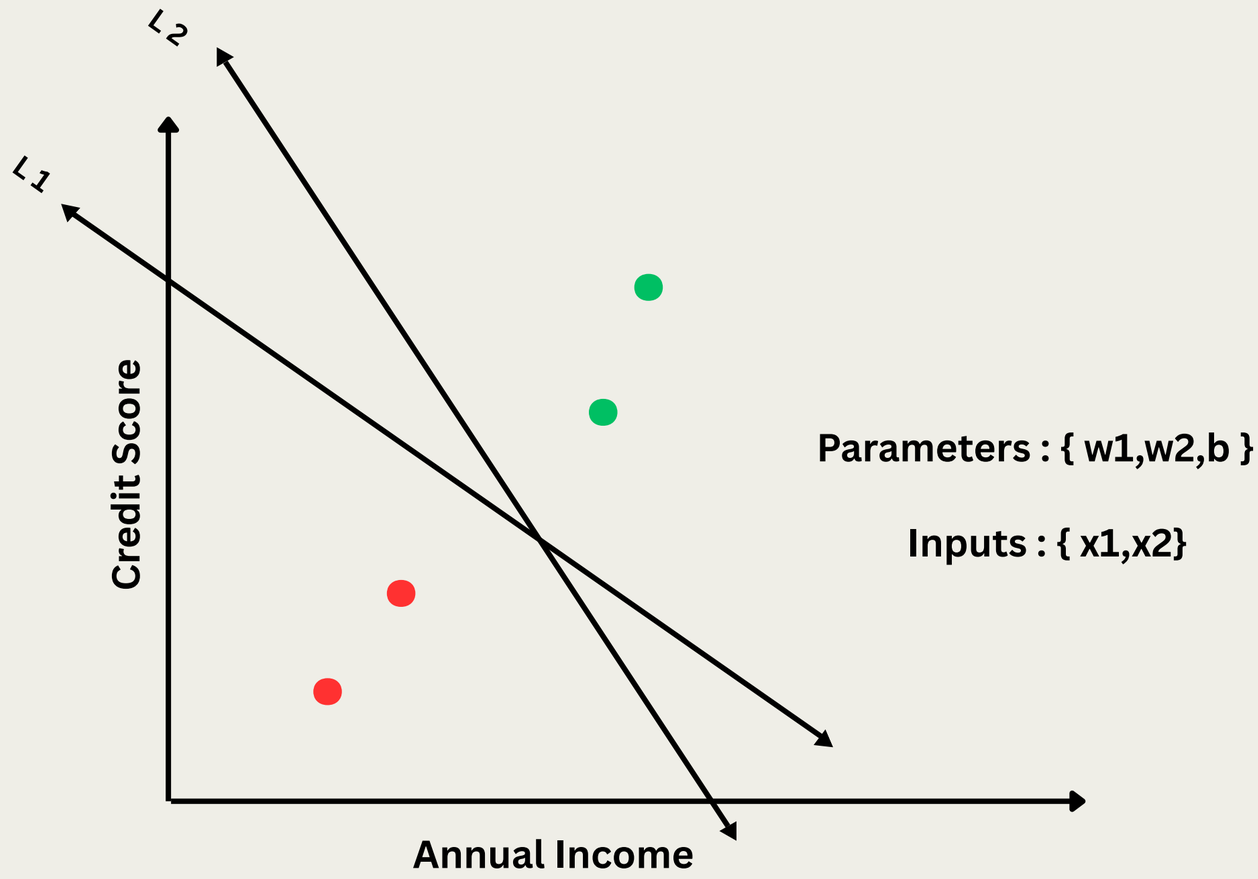
```
for i in range(epochs):
        Randomly select a point (X_i, y_i)

            // Calculate prediction (y_hat) for the point
            // Calculate the weighted sum and apply the step function
            z = dot_product(weights, X_i)
            y_hat = f(z)
            // Update weights using the single, simplified rule
            w_new = w_old + Alpha * (y_i - y_hat) * X_i
```

BEST FIT LINE (line which classify accurately)

Which Line is Better? The Role of Error 🎯

While both lines L1 and L2 correctly separate the green and red dots in our current dataset, L1 is the better and more accurate model.

The reason is that L1 provides a larger margin or "safety buffer" around the data points. It's not just correct; it's also more confident. If a new red dot appears slightly to the right, L1 is more likely to still classify it correctly, whereas L2 might make a mistake.



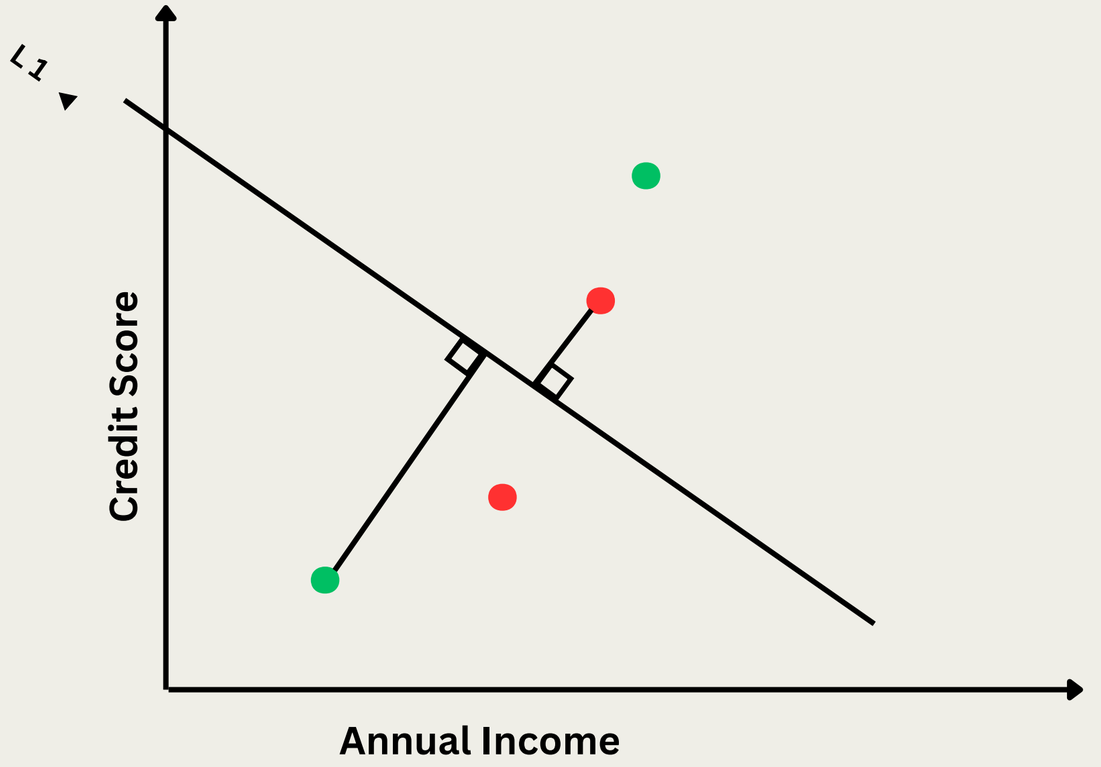
The Quest for a Good Loss Function

To train a model, we first need a way to measure its error. The journey to the right loss function involves a few steps.

Idea 1: Count Wrongly Classified Points This is the simplest approach. However, it fails because it only considers which side of the line a datapoint is on. It neglects the magnitude of the error. A point that is barely wrong is treated the same as a point that is extremely wrong.

Idea 2: Sum of Perpendicular Distances A better idea is to find the sum of the perpendicular distances from the wrongly classified points to the line and minimize it. This accounts for the magnitude of the error. However, calculating these geometric distances for every point during every step is computationally difficult and inefficient.

Idea 3: Use the "Power of the Point" To simplify, we can use the value of **$z_i = w \cdot x_i + b$** (the "power of the point"). While this is computationally easy, simply summing these values creates new problems. For example, a point that is very far on the correct side would contribute a large value, confusing the minimization process.



The Solution: Hinge Loss

The standard and most effective loss function for this task is a variation of the Hinge Loss, used in Scikit-learn's SGDClassifier. The formula calculates the average loss over all 'n' data points:

$$L = \frac{1}{n}\sum_{i=1}^{n}\max(0, -y_i \cdot f(x_i))$$

Where **f(xi) = w1*xi1 + w2*xi2 + b** and yi is the class label (+1 or -1). The term max(0, ...) is the key:

If a point is correctly classified, yi · f(xi) is positive, so −yi · f(xi) is negative. The loss for that point becomes 0.

If a point is wrongly classified, yi · f(xi) is positive, so −yi · f(xi) is positive. The loss becomes a positive value proportional to how wrong the prediction is.

Actual (y_i)	Prediction Sign ($f(x_i)$)	Loss `max(0, -y_i * f(x_i))`	Result
1	1	`max(0, -1) = 0`	Correct
-1	-1	`max(0, -1) = 0`	Correct
1	-1	`max(0, 1) > 0`	**Wrongly Classified**
-1	1	`max(0, 1) > 0`	**Wrongly Classified**

Minimizing the Loss with Gradient Descent

Our goal is to find the values of **w1, w2, and b** that **minimize the total loss L**. We do this by taking small steps in the direction opposite to the gradient.

```
for epoch in epochs:
    w1 = w1 - Alpha * (∂L / ∂w1)
    w2 = w2 - Alpha * (∂L / ∂w2)
    b  = b  - Alpha * (∂L / ∂b)
```

Calculating the Gradients (The Chain Rule)

To find the partial derivatives, we apply the chain rule.

- $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f(x_i)} \cdot \frac{\partial f(x_i)}{\partial w_1}$
- $\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial f(x_i)} \cdot \frac{\partial f(x_i)}{\partial w_2}$
- $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial f(x_i)} \cdot \frac{\partial f(x_i)}{\partial b}$

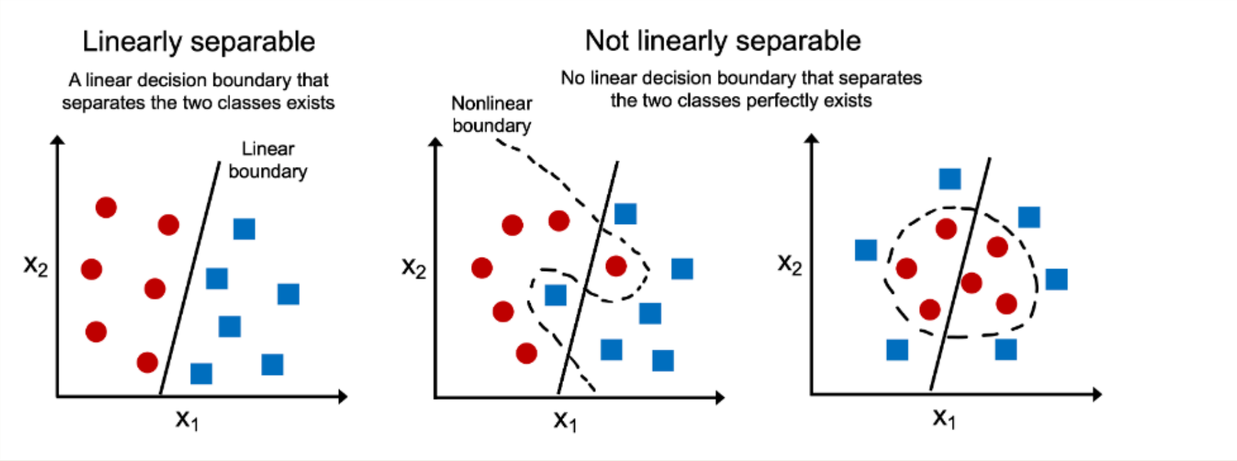
The individual components are calculated as follows:

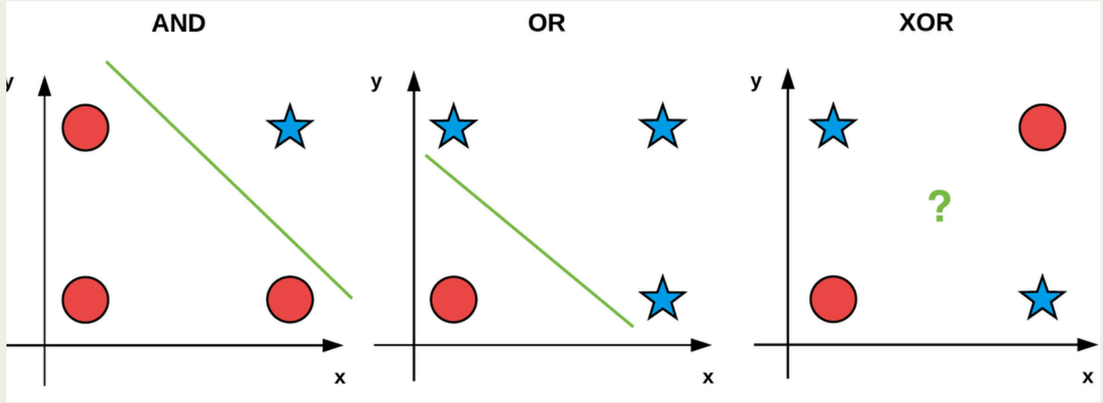
1. **Derivative of Loss w.r.t. the function output:**
   - $\frac{\partial L}{\partial f(x_i)} = 0$, if $y_i \cdot f(x_i) \geq 0$ (Correctly classified)
   - $\frac{\partial L}{\partial f(x_i)} = -y_i$, if $y_i \cdot f(x_i) < 0$ (Wrongly classified)
2. **Derivative of the function output w.r.t. each parameter:**
   - $\frac{\partial f(x_i)}{\partial w_1} = x_{i1}$
   - $\frac{\partial f(x_i)}{\partial w_2} = x_{i2}$
   - $\frac{\partial f(x_i)}{\partial b} = 1$

1.2 THE SETBACK: MINSKY & THE XOR PROBLEM (1969)

Minsky delivered a definitive, mathematical proof demonstrating the fundamental limitation of Rosenblatt's model. He showed that a single-layer Perceptron could only solve problems that are linearly separable.

Since the single-layer Perceptron could do nothing but draw one straight line, Minsky concluded that it was utterly useless for solving any real-world, complex problem that requires non-linear decision boundaries.

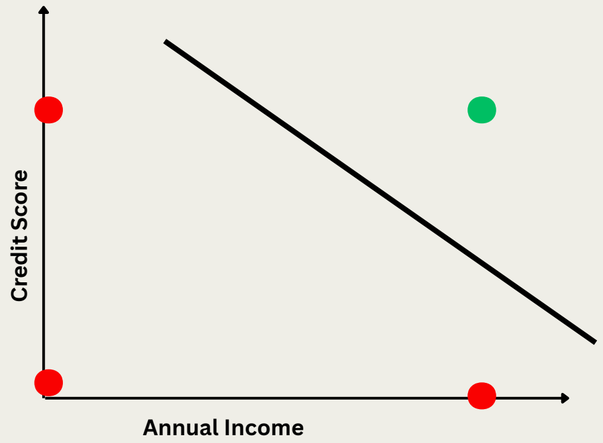




AND Gate

The AND gate outputs 1 only when both inputs are 1.

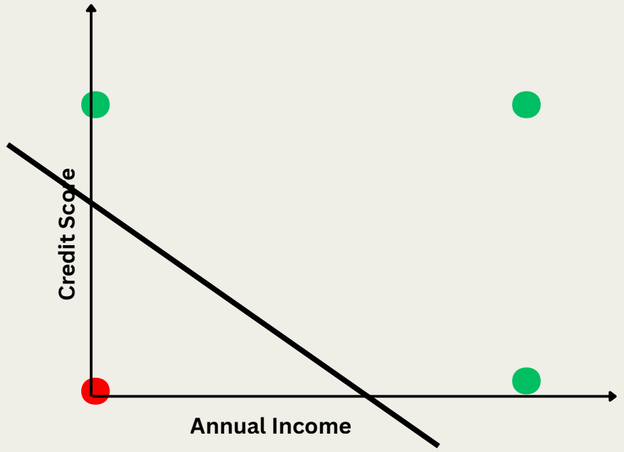
Input A	Input B	Output
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**



OR Gate

The OR gate outputs 1 1 if at least one of the inputs is 1.

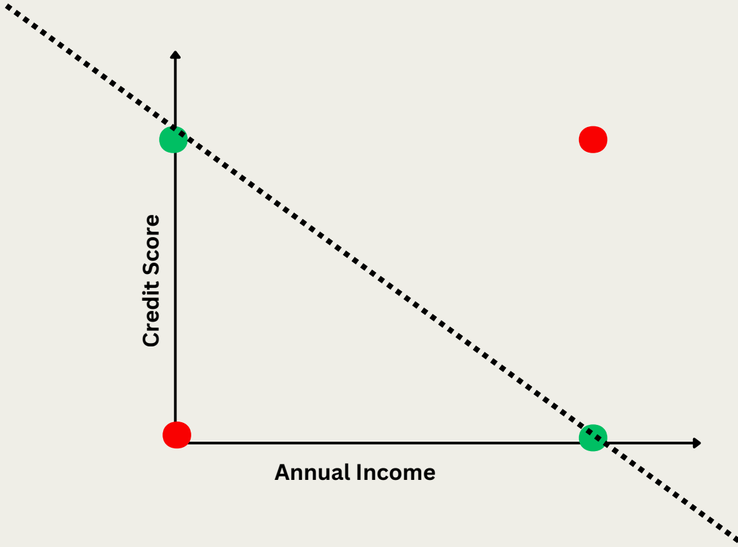
Input A	Input B	Output
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**



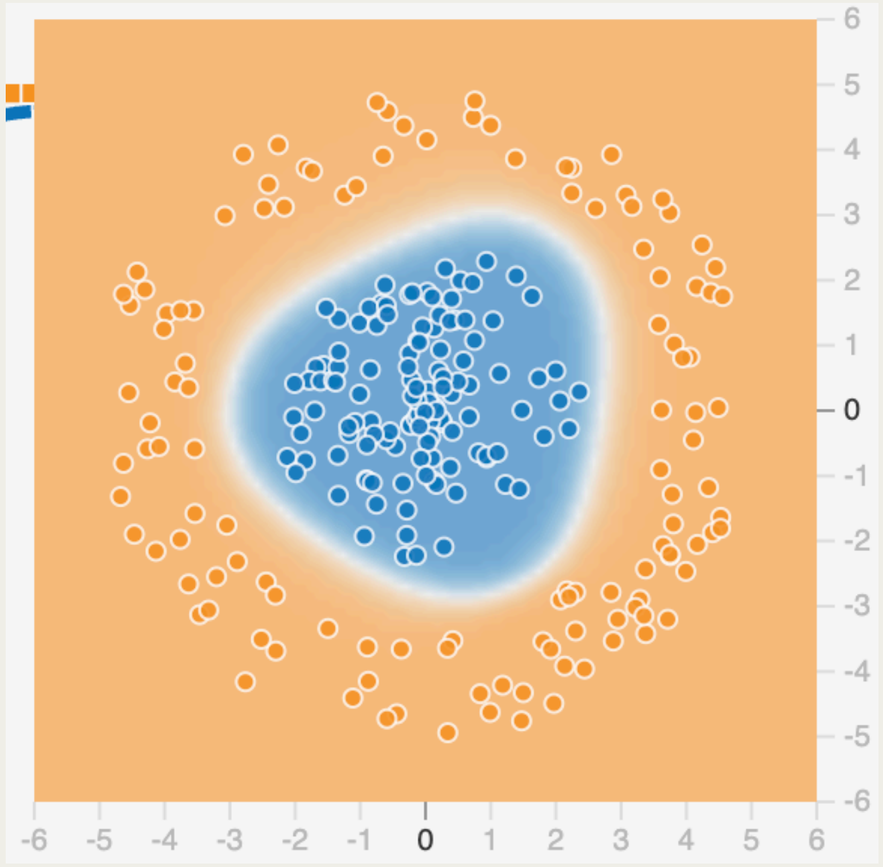
XOR Gate

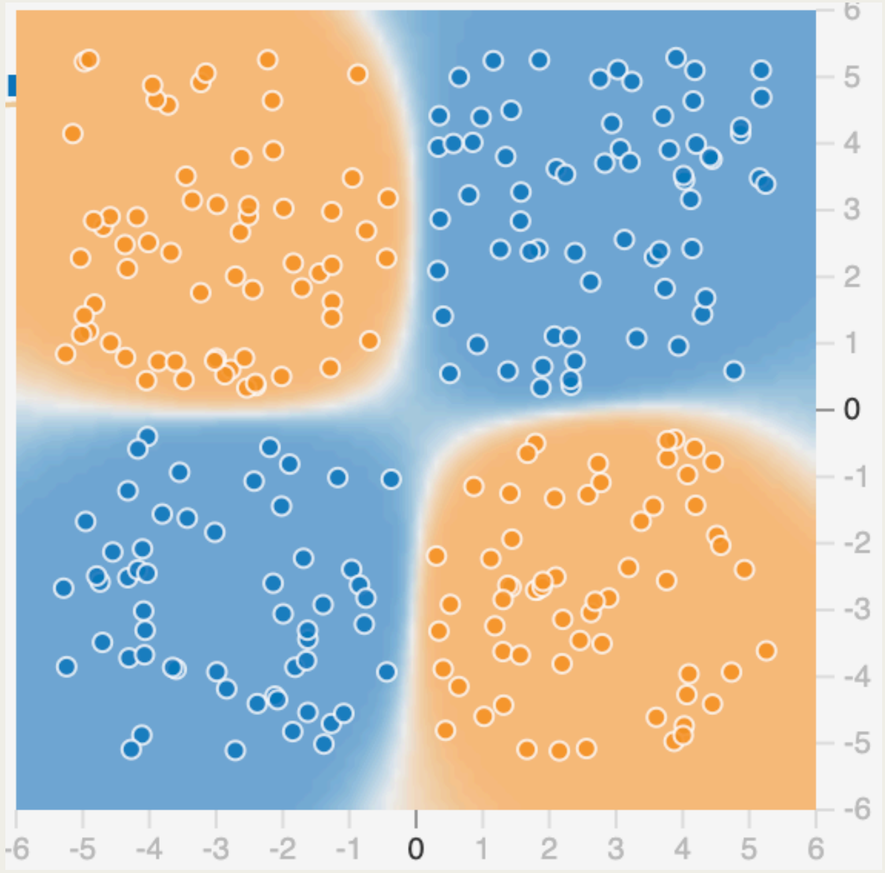
TThe XOR (Exclusive OR) gate outputs 1 only when the inputs are different.

Input A	Input B	Output
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**



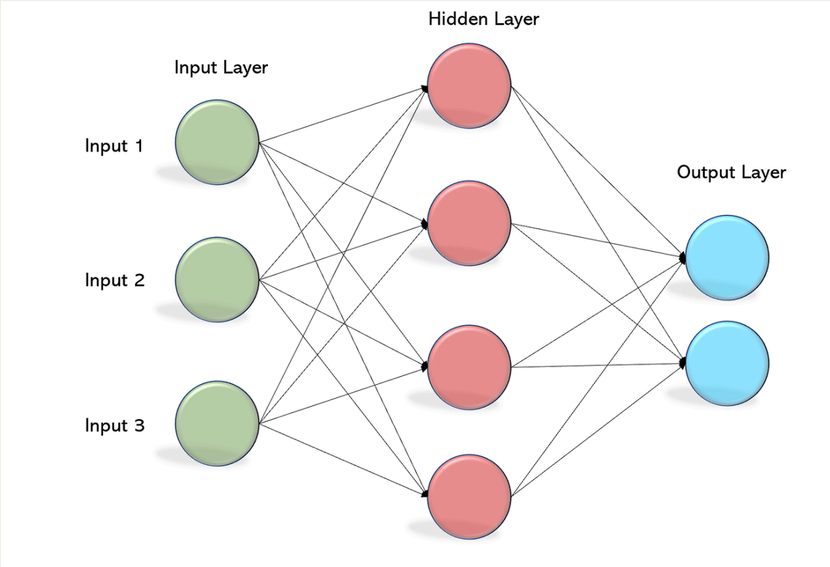
NON LINEAR DATA SET





1.3 THE SOLUTION: BACKPROPAGATION & MLP (1980)

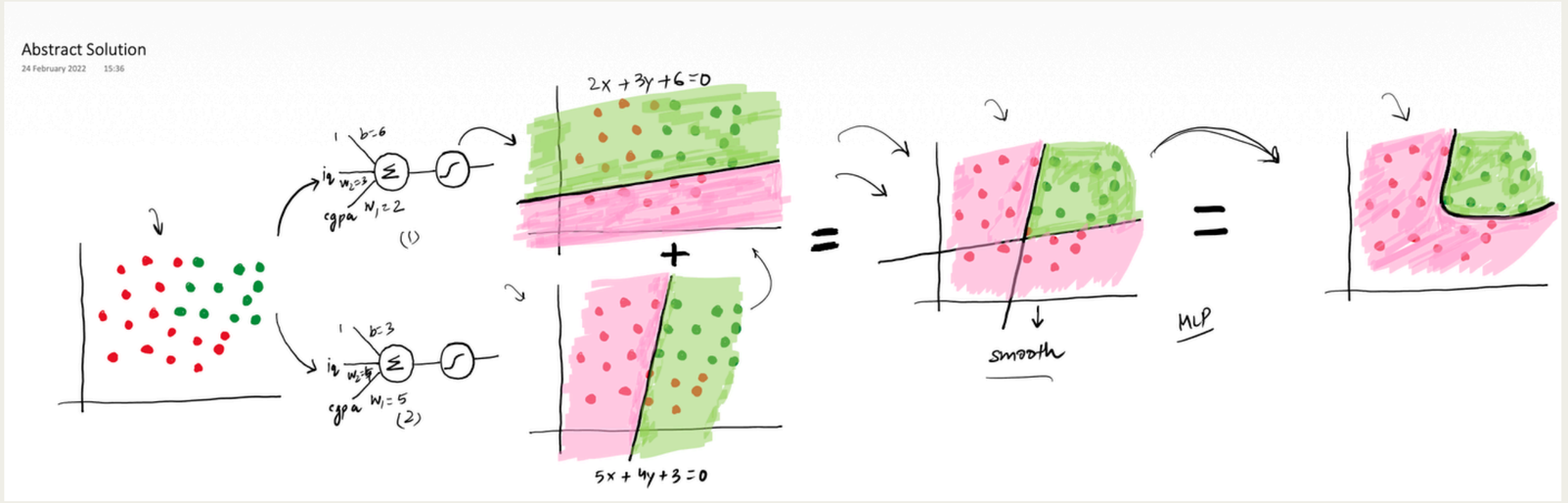
Minsky was right about the single-layer Perceptron, but he missed the potential of stacking them. Researchers knew the solution to non-linear problems like XOR had to involve Multi-Layer Perceptrons (MLPs), which have one or more Hidden Layers.

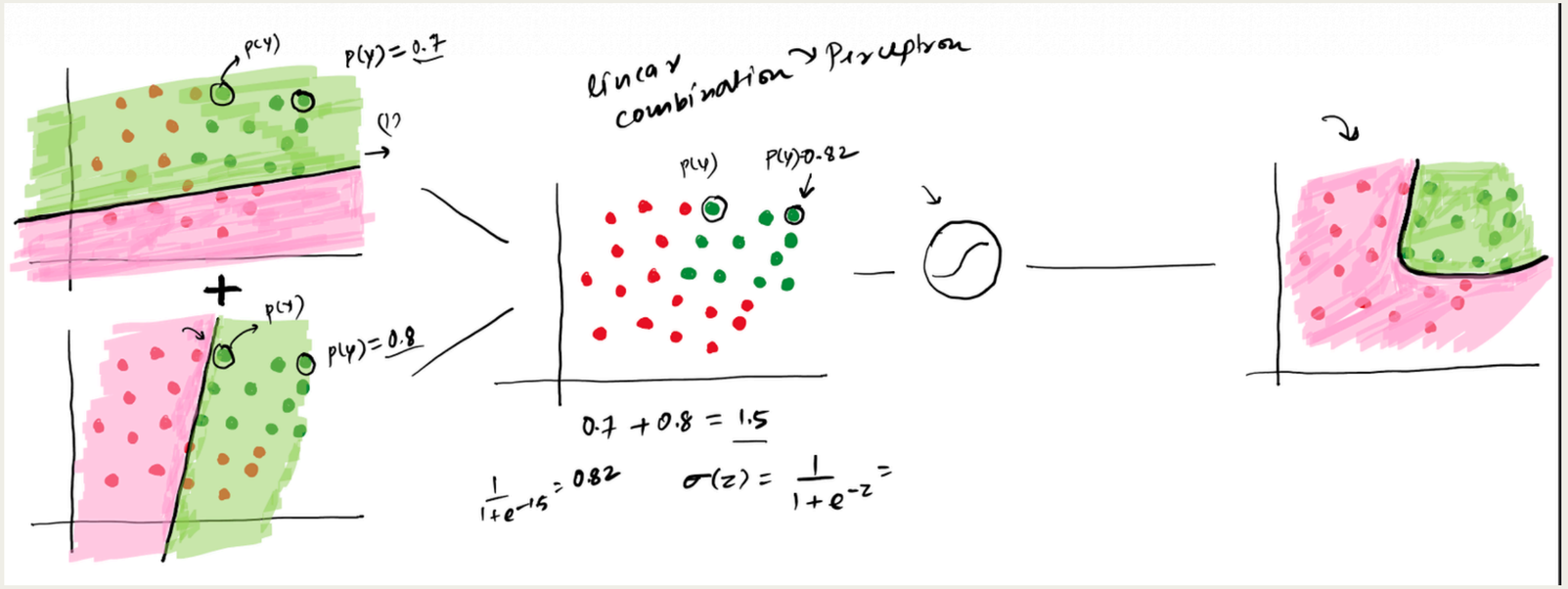


However, a massive problem remained: How do you train those hidden layers? When the network makes an error, that error only appears at the final output layer. How do you adjust the weights deep inside a hidden layer when you don't know that layer's specific contribution to the error?

In the mid-1980s, the key was formalized and popularized by Geoffrey Hinton (alongside David Rumelhart and Ronald Williams) in their seminal paper: “Learning representations by back-propagating errors”.

STACKING OF PERCEPTRON





How Stacking Happens

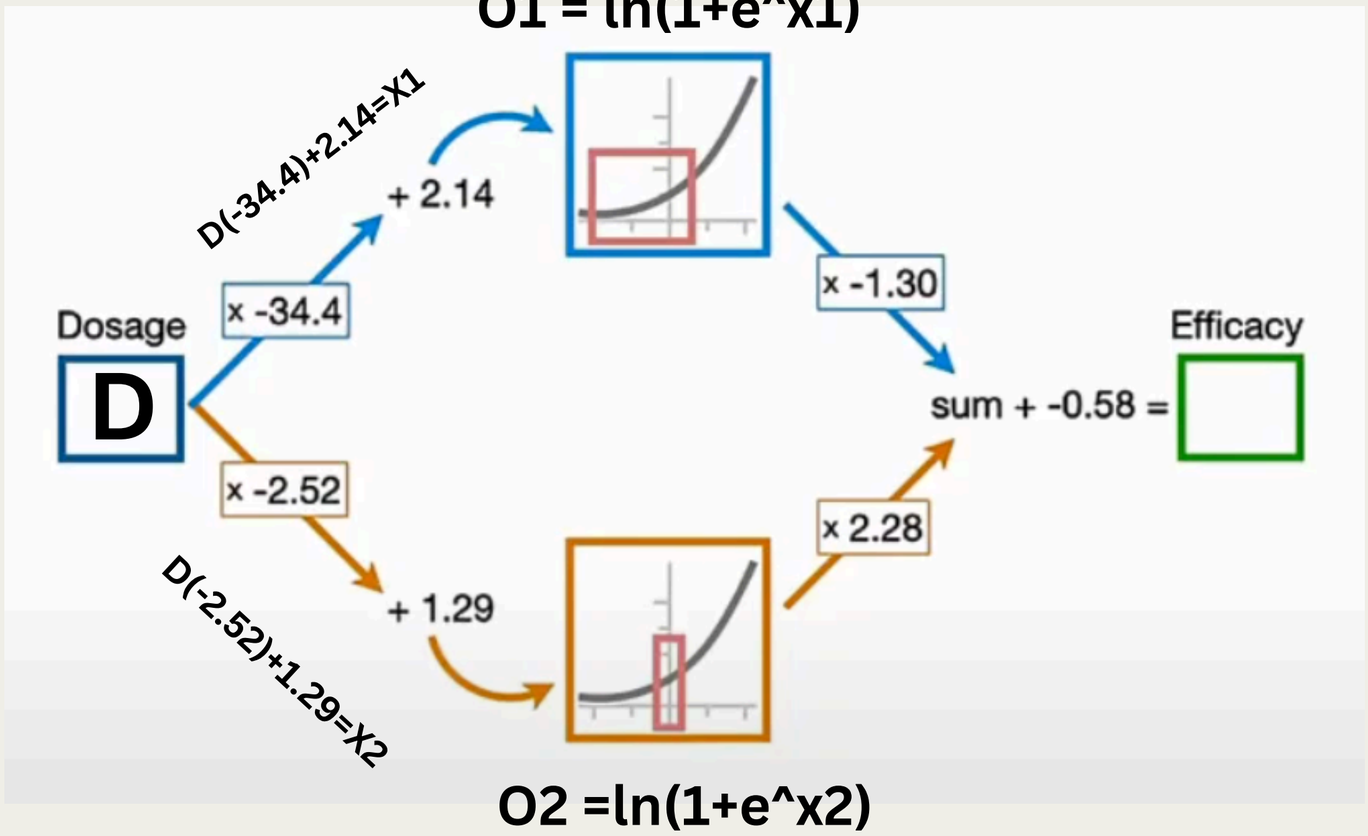
Stacking is a layered process. The outputs from the first "hidden" layer of perceptrons are collected and used as the inputs for a second "output" layer. This final layer then combines this information to make a single, sophisticated decision.

What is "Smoothing"?

"Smoothing" is the job of the non-linear activation function (like Sigmoid). It takes the sharp-angled boundary created by combining the hidden layer's straight lines and transforms it into a continuous, curved line by outputting a smooth range of probabilities instead of a hard 0-or-1 switch.

How It Solves Non-Linearity

It's a two-step solution. First, the hidden layer combines multiple straight lines to surround the data with a complex, angled boundary. Then, the activation function bends and smooths this angled boundary into a curve, allowing it to perfectly separate data that a single straight line never could.



Dosage (D)	Efficacy
0.1	0
0.2	0
0.3	0
0.4	1
0.5	1
0.6	1
0.7	0
0.8	0
0.9	0

D(-2.52)+1.29=X2

D(-34.4)+2.14=X1

O1 = ln(1+e^(D(-2.52)+1.29))

O2= ln(1+e^(D(-34.4)+2.14))

O1(-1.30)+**O2**(2.28)+(-0.58)=**O3**= Y_cap

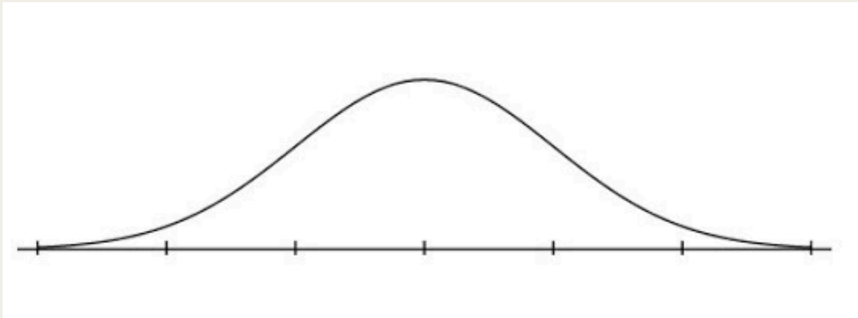
SoftPlus : ln(1+e^x)

Modeling Non-Linear Data

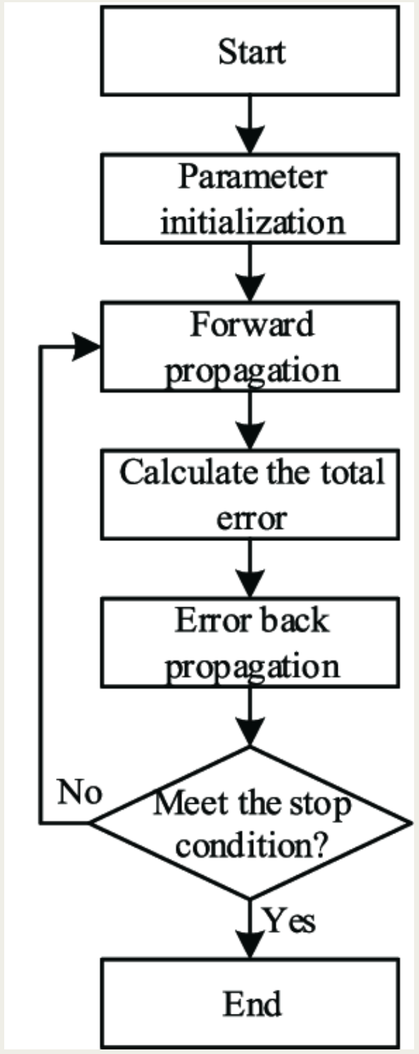
The Goal: To model the non-linear "bump" in the Dosage-Efficacy data, which a single Perceptron cannot.

The Method: We use a Multi-Layer Perceptron (MLP). This slide shows a forward pass calculation through a simple network with a 2-neuron hidden layer.

The Mechanism: The two hidden neurons, activated by the smooth SoftPlus function, work together to form the required peak shape. The output neuron then combines their signals to produce the final prediction.



BACKPROPAGATION



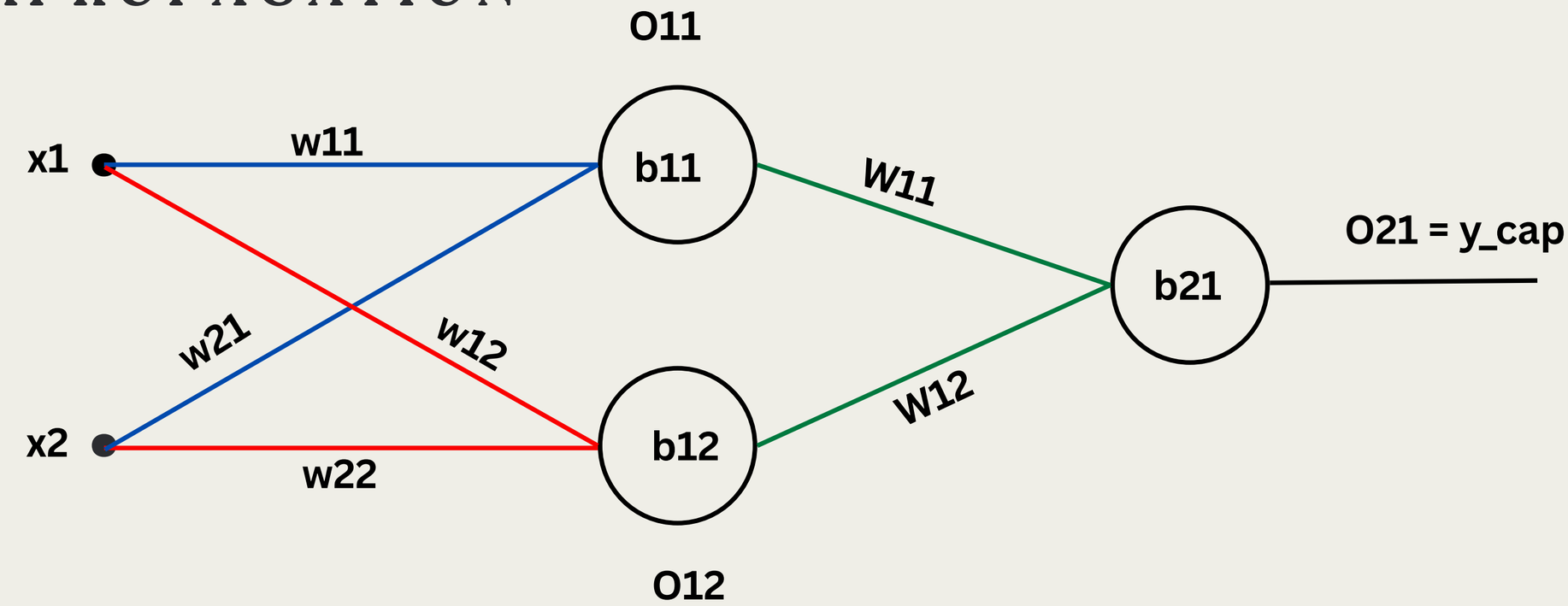
The solution, named Backpropagation, is a brilliant application of differential calculus,specifically the Chain Rule. It works in two steps:

1. Forward Pass: The data moves forward, creating the final error (Loss).

2. Backward Pass: The error is calculated at the output and then propagated backward through the network, layer by layer.

Using the Chain Rule, Backpropagation mathematically calculates the gradient—the precise direction and amount of change needed—for every single weight in the network, allowing even the deepest layers to learn.
This breakthrough finally made MLPs trainable and mathematically validated their true potential, confirming what we call the Universal Approximation Theorem: A neural network with at least one hidden layer can approximate any continuous function. The XOR problem was officially solved!

UNDER THE HOOD: THE CALCULUS OF BACKPROPAGATION



Trainable Parameters :

w11,w12,w21,w22,b11,b12,W11,W12,b21

z11= x1(w11)+x2(w21)+b11 **O11= σ(z11)**

z12= x1(w12)+x2(w22)+b12 **O12 = σ(z12)**

z21=**O11**(W11)+**O12**(W12)+b21 **O21 = σ(z21)**

BACKPROPAGATION

Gradients for the Output Layer (Layer 2)

The gradients for the weights connecting the hidden layer to the output layer (`W11` , `W12` , `b21`) are calculated first.

- **Partial derivative for W_{11}:**

$$\frac{\partial L}{\partial W_{11}} = \frac{\partial L}{\partial y_{cap}} \cdot \frac{\partial y_{cap}}{\partial z_{21}} \cdot \frac{\partial z_{21}}{\partial W_{11}} = \frac{\partial L}{\partial y_{cap}} \cdot \sigma'(z_{21}) \cdot O_{11}$$

- **Partial derivative for W_{12}:**

$$\frac{\partial L}{\partial W_{12}} = \frac{\partial L}{\partial y_{cap}} \cdot \frac{\partial y_{cap}}{\partial z_{21}} \cdot \frac{\partial z_{21}}{\partial W_{12}} = \frac{\partial L}{\partial y_{cap}} \cdot \sigma'(z_{21}) \cdot O_{12}$$

- **Partial derivative for b_{21}:**

$$\frac{\partial L}{\partial b_{21}} = \frac{\partial L}{\partial y_{cap}} \cdot \frac{\partial y_{cap}}{\partial z_{21}} \cdot \frac{\partial z_{21}}{\partial b_{21}} = \frac{\partial L}{\partial y_{cap}} \cdot \sigma'(z_{21}) \cdot 1$$

$$A_{\text{new}} = A_{\text{old}} - \alpha \frac{\partial L}{\partial A_{\text{old}}}$$

x 9 times

x epochs times

Gradients for the Hidden Layer (Layer 1)

The gradients for the weights connecting the input layer to the hidden layer are calculated next by continuing the chain rule backward.

- **Partial derivative for w_{11}:**

$$\frac{\partial L}{\partial w_{11}} = \left(\frac{\partial L}{\partial y_{cap}} \cdot \frac{\partial y_{cap}}{\partial z_{21}} \cdot \frac{\partial z_{21}}{\partial O_{11}} \right) \cdot \frac{\partial O_{11}}{\partial z_{11}} \cdot \frac{\partial z_{11}}{\partial w_{11}} = \left(\frac{\partial L}{\partial y_{cap}} \cdot \sigma'(z_{21}) \cdot W_{11} \right) \cdot \sigma'(z_{11}) \cdot x_1$$

- **Partial derivative for w_{21}:**

$$\frac{\partial L}{\partial w_{21}} = \left(\frac{\partial L}{\partial y_{cap}} \cdot \sigma'(z_{21}) \cdot W_{11} \right) \cdot \sigma'(z_{11}) \cdot x_2$$

- **Partial derivative for b_{11}:**

$$\frac{\partial L}{\partial b_{11}} = \left(\frac{\partial L}{\partial y_{cap}} \cdot \sigma'(z_{21}) \cdot W_{11} \right) \cdot \sigma'(z_{11}) \cdot 1$$

- **Partial derivative for w_{12}:**

$$\frac{\partial L}{\partial w_{12}} = \left(\frac{\partial L}{\partial y_{cap}} \cdot \frac{\partial y_{cap}}{\partial z_{21}} \cdot \frac{\partial z_{21}}{\partial O_{12}} \right) \cdot \frac{\partial O_{12}}{\partial z_{12}} \cdot \frac{\partial z_{12}}{\partial w_{12}} = \left(\frac{\partial L}{\partial y_{cap}} \cdot \sigma'(z_{21}) \cdot W_{12} \right) \cdot \sigma'(z_{12}) \cdot x_1$$

- **Partial derivative for w_{22}:**

$$\frac{\partial L}{\partial w_{22}} = \left(\frac{\partial L}{\partial y_{cap}} \cdot \sigma'(z_{21}) \cdot W_{12} \right) \cdot \sigma'(z_{12}) \cdot x_2$$

- **Partial derivative for b_{12}:**

$$\frac{\partial L}{\partial b_{12}} = \left(\frac{\partial L}{\partial y_{cap}} \cdot \sigma'(z_{21}) \cdot W_{12} \right) \cdot \sigma'(z_{12}) \cdot 1$$

THE SECOND AI WINTER: PRACTICAL ROADBLOCKS (90S)

"So, the math was solved in the 80s, but the technology wasn't ready. The 1990s gave us the Second AI Winter because Deep Learning hit a wall of practical limitations.

- **Computational Scarcity**: Standard CPUs were too slow. We needed hardware optimized for parallel matrix multiplication, but it simply didn't exist.
- **The Data Drought**: Deep Networks are incredibly data hungry, and in the 90s, we had no massive, labeled public datasets.
- **Fragile Initialization**: We had no reliable way to set the initial weights and biases, leading to unreliable training and vanishing gradients.

Because of this, reliable Machine Learning models like SVMs gave far superior results on the small datasets we had. Deep Learning became mathematically sound but technologically impractical.
The field went into hibernation, waiting for the hardware and data to catch up."

THE ICE MELTS: DEEP LEARNING IS NAMED (2006)

After the Second AI Winter, the theoretical fire was still burning for researchers like Yann LeCun and Geoffrey Hinton. We had the engine (Backpropagation), but we needed the ignition key—a way to reliable start the network.

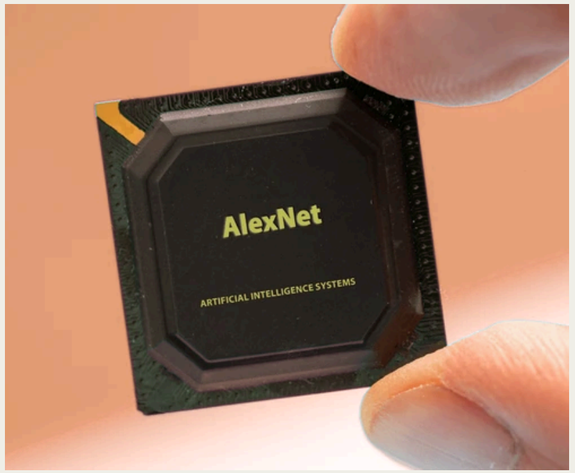
In 2006, Hinton found that key. He introduced Unsupervised Pre-training. Instead of starting with random, garbage weights, you use the available unlabeled data to teach the network how to set its own good initial weights, layer by layer. This solved the fragile initialization problem that plagued the 90s!

This conceptual leap—treating networks as multi-layered feature extractors—finally necessitated a new name. This is when the term 'Deep Learning' was officially framed.
It was no longer just a neural network; it was a system designed to learn deep, hierarchical representations of data. The stage was set for the biggest comeback in computer science history!"

THE BREAKTHROUGH: GPU POWER & IMAGENET (2012)

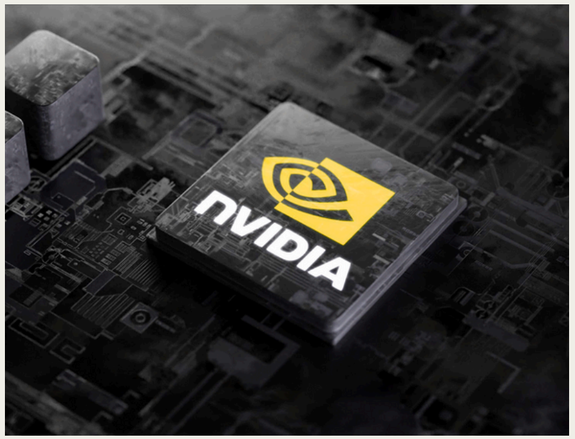
"**The Second AI Winter** was about waiting for hardware. By 2012, two things converged: massive labeled data, and the perfect processor.

That year, **Geoffrey Hinton** and his students entered the annual **ImageNet** image classification contest. Their network was called **AlexNet**.



Their secret weapon? They didn't use slow **CPUs**. They leveraged the **parallel processing power** of **NVIDIA GPUs**—Graphics Processing Units—the same chips designed to run video games!

The math of a neural network is just **massive matrix multiplication**, which is exactly what a GPU excels at.



The result was stunning. AlexNet dropped the error rate from about **26%** to an unprecedented **15.3%**. It wasn't just a victory; it was a total demolition of all previous methods.

This was the moment the world took notice. The theory was finally matched by the computational power. Deep learning was no longer a theoretical pursuit; it was the superior technology, sparking the **AI revolution** we are living in today."

THE GOLD RUSH: INVESTMENT & DEMOCRATIZATION

"The 15.3% error rate wasn't just a paper; it was a warning shot that triggered a Deep Learning Gold Rush.

Google, **Facebook (Meta),** and other giants saw that neural networks were the future, and they invested billions. This corporate focus drove two massive accelerations:

Research Powerhouses: Companies like **DeepMind** (acquired by Google) pushed the boundaries, turning theory into world-changing applications.

Democratization: To speed up development, these companies open-sourced their core tools. **Google** gave us **TensorFlow** and **Keras**; **Facebook** gave us **PyTorch** .

These frameworks made the complex math we're discussing accessible to everyone. You no longer needed a PhD to build a deep network—you just needed a few lines of Python. This shift accelerated the field exponentially, setting the stage for the true public demonstrations of AI power, which we'll look at next."

ALPHAGO: STRATEGY MASTERED (2016)

"If **2012** was the big bang for **computer vision**, **2016** was the moment AI proved it could master **human strategy** and **intuition**.

The world was still adjusting to ImageNet when Google DeepMind dropped a bombshell: **AlphaGo**.

Go is the hardest strategy game ever invented—the number of possible moves is **greater than** the atoms in the universe. It requires intuition, not just calculation.
The result? AlphaGo soundly defeated **Lee Sedol**, the **human world champion**.

This victory was huge because it wasn't just raw computation; it was a demonstration of **Deep Reinforcement Learning**—the application of neural networks to high-level, complex decision-making.

AlphaGo showed the world that **Deep Learning** wasn't just classifying cats and dogs; it could learn to think."

AlphaGo

CREATIVE AI: THE FORGER AND THE DETECTIVE (GANS)

"After conquering strategy with AlphaGo, the next frontier was creativity itself.

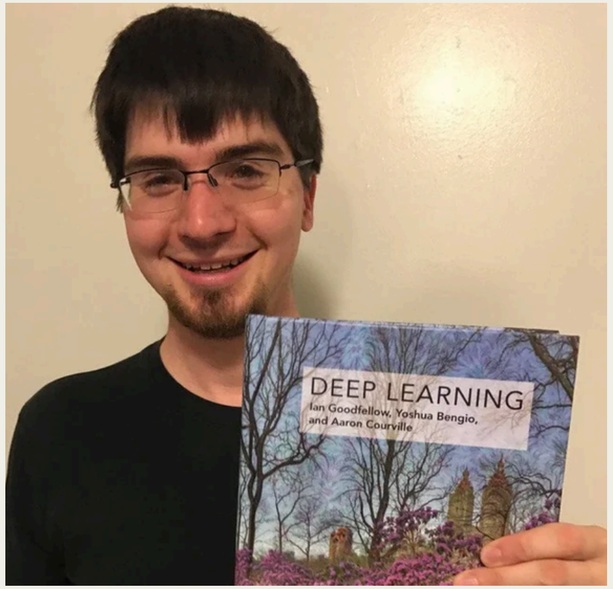
In **2014**, **Ian Goodfellow** introduced a revolutionary architecture called the **Generative Adversarial Network**, or **GAN**.

GANs use a clever concept: they pit two neural networks against each other in a constant, adversarial game:

1. **The Generator** tries to create photorealistic fake content. Think of it as the ultimate artistic forger.
2. **The Discriminator** tries to distinguish that fake content from the real data. It's the art detective.

By constantly competing, they both rapidly improve. The Generator **becomes better at faking**, and the Discriminator **becomes better at detecting**.

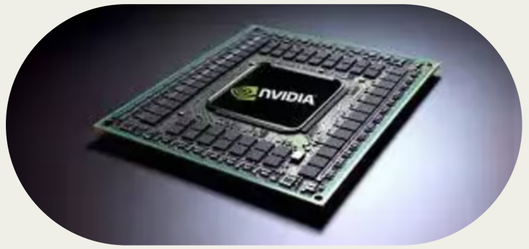
The results are astonishing—from **photorealistic faces** that don't exist, to **high-resolution art**, and even deepfake technology. This innovation marked the true beginning of the **Generative AI era**, transforming AI from a tool of analysis to a tool of creation."



THE FOUR PILLARS OF MODERN DEEP LEARNING

"So, the question is: **Why is the revolution happening now?** It's the perfect convergence of **four**, equally critical pillars. Think of it like building a super-powered car:

* **The Fuel (Data)**: Deep Learning requires oceans of data. We now have massive, publicly available, labeled datasets—from ImageNet for images to **YouTube-8M** for video. No data, no learning.



* **The Engine (Chips)**: The math demands speed. Traditional **CPUs** couldn't handle the load. The rise of **NVIDIA CUDA GPUs** and **Google TPUs** gave us the massive, parallel processing power necessary to turn simple matrix operations into lightning speed.

* **The Toolkit (Frameworks)**: To scale, we needed accessible tools. Open-source frameworks like **TensorFlow** and **PyTorch** democratized the field, allowing millions of developers to experiment.

* **The Design (Architectures)**: We evolved beyond simple **MLPs**. Specialized architectures like **CNNs** for images, **RNNs** for sequences, and **BERT** for text gave us models tailored for every specific task.

Thank You