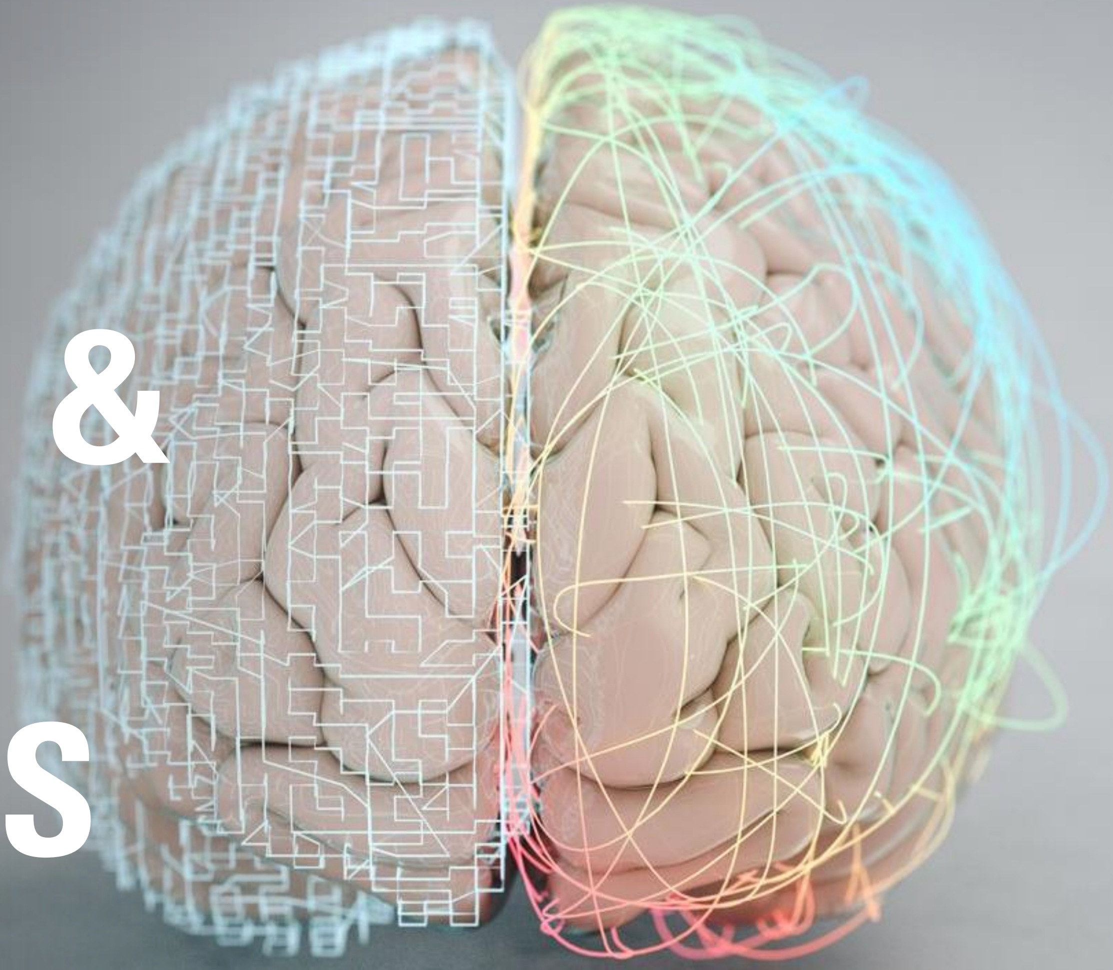
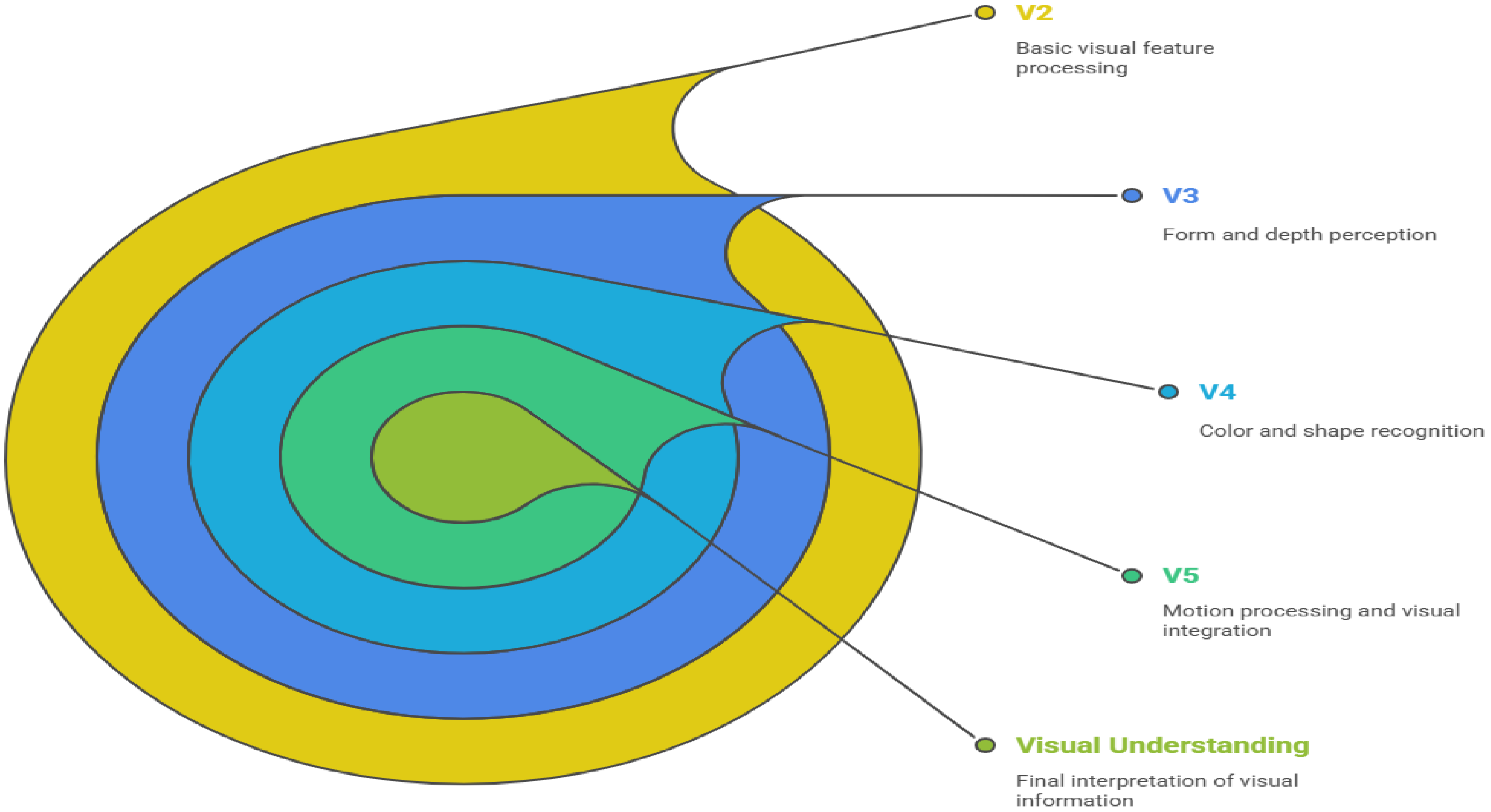
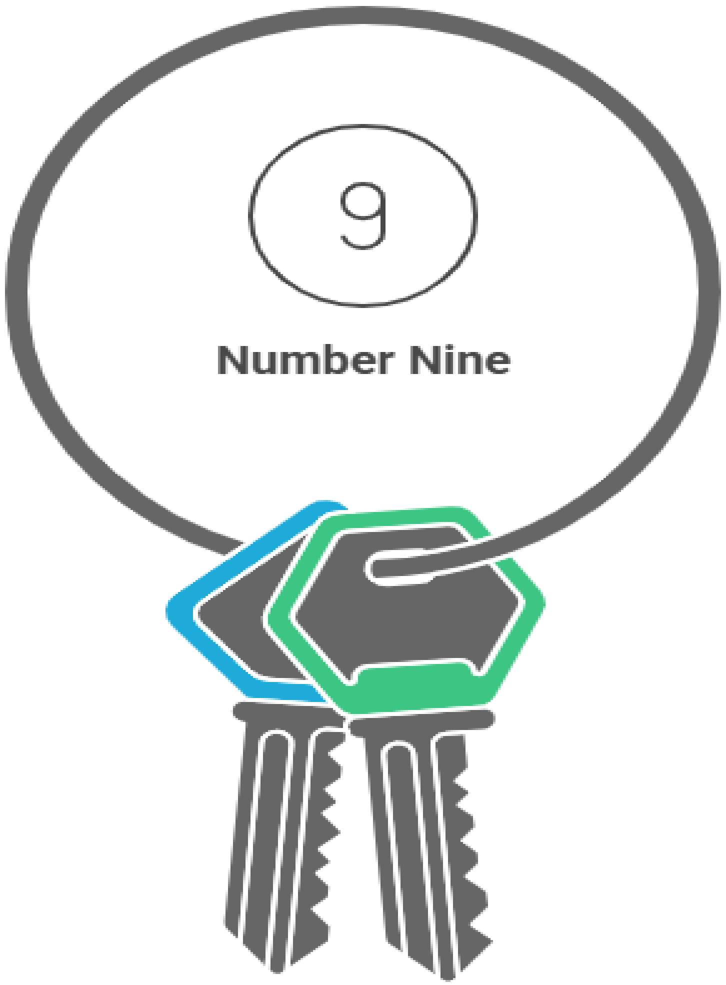
DEEP LEARNING & NEURAL NETWORKS



Visual Processing in the Brain



Structure of the Number Nine

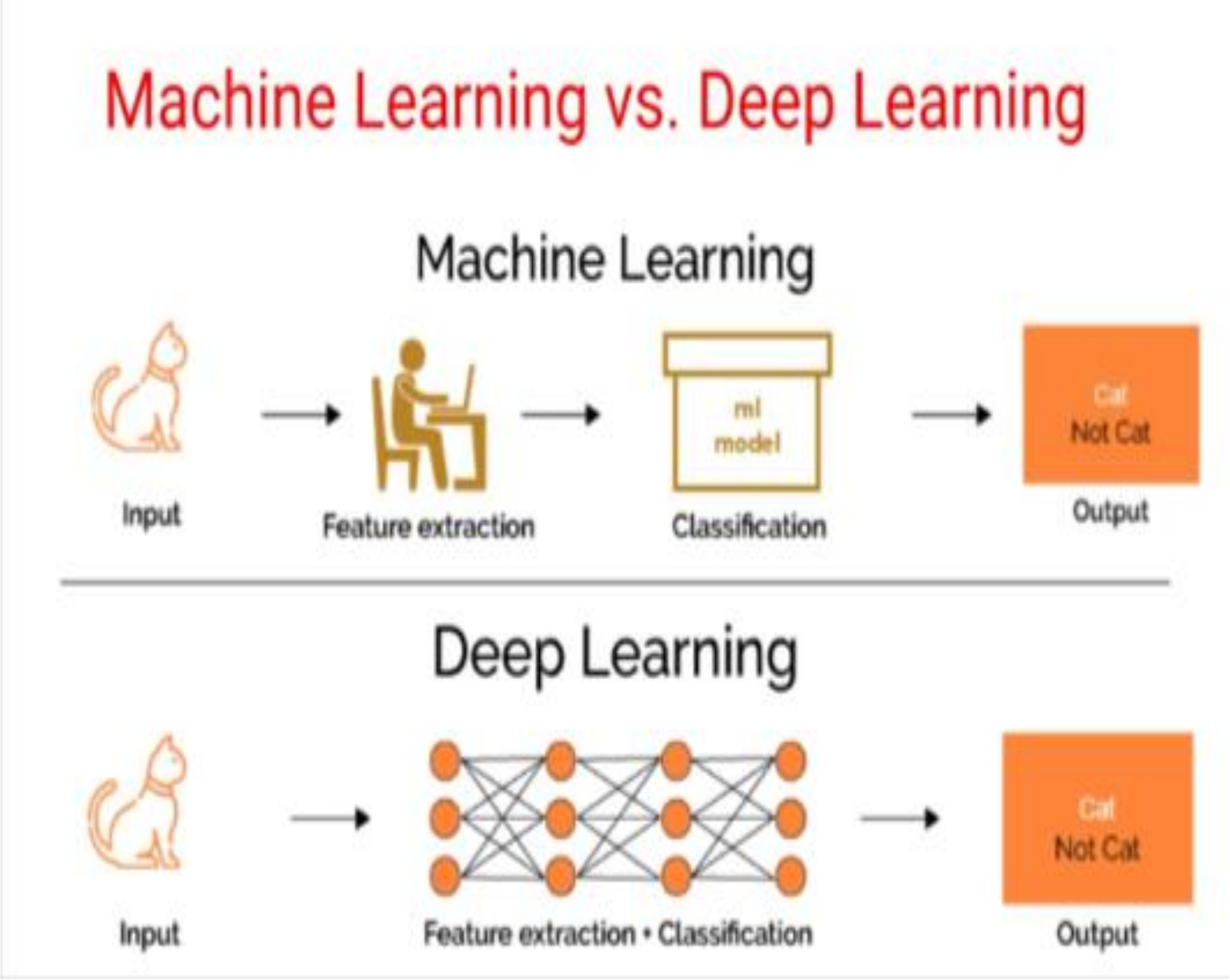


Loop at the Top

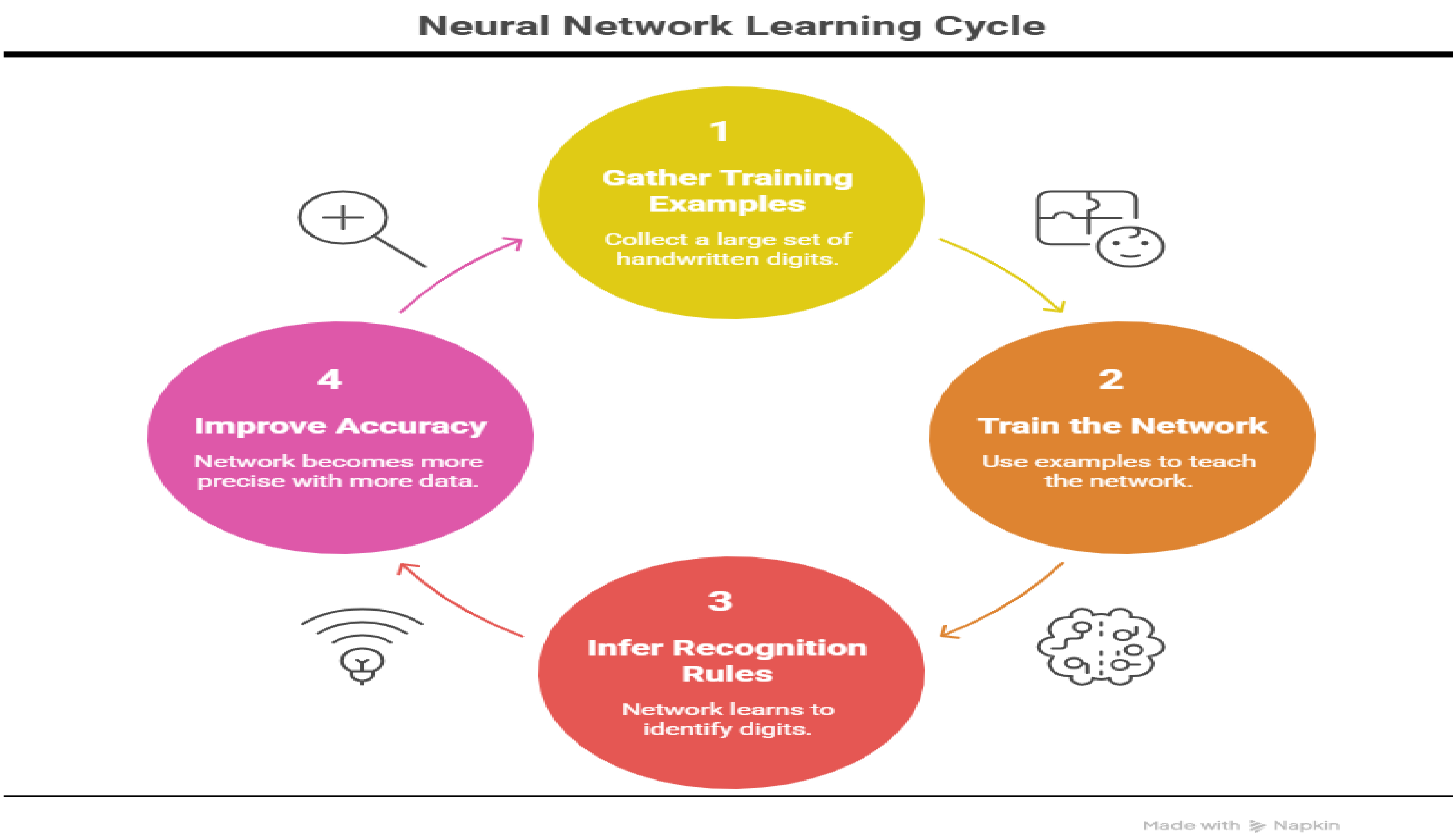
The upper part of the number nine, forming a closed circle.

Vertical Stroke

The straight line extending down from the loop on the right.



DL is a **particular kind of machine learning** that achieves great power and flexibility by learning to represent the world as a **nested hierarchy of concepts**, with each concept defined in relation to simpler concepts and **more abstract representations computed in terms of less abstract ones**


Neural Network Learning Cycle
1
Gather Training Examples
Collect a large set of handwritten digits.
2
Train the Network
Use examples to teach the network.
3
Infer Recognition Rules
Network learns to identify digits.
4
Improve Accuracy
Network becomes more precise with more data.
Made with Napkin

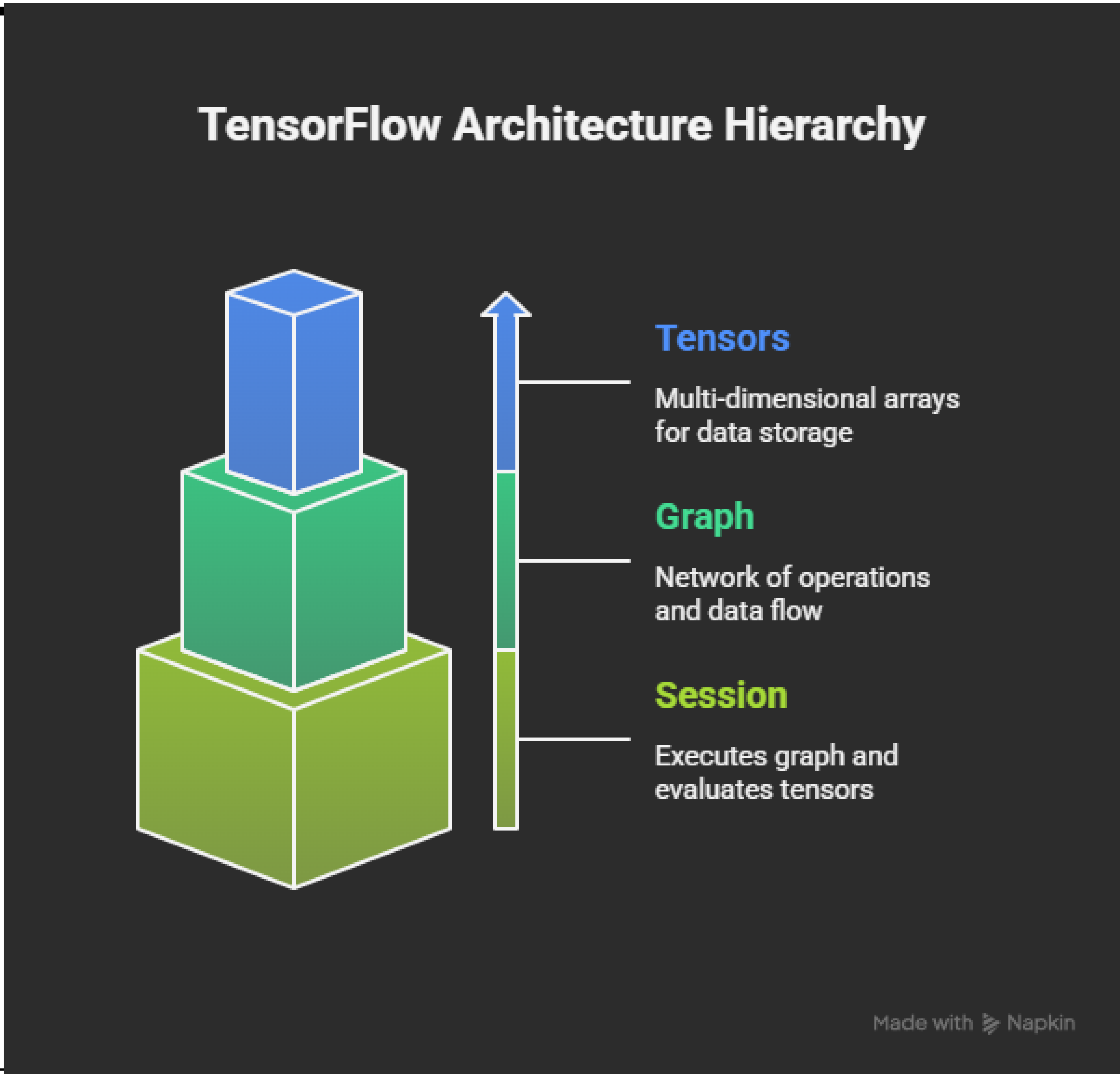
Deep Learning Framework

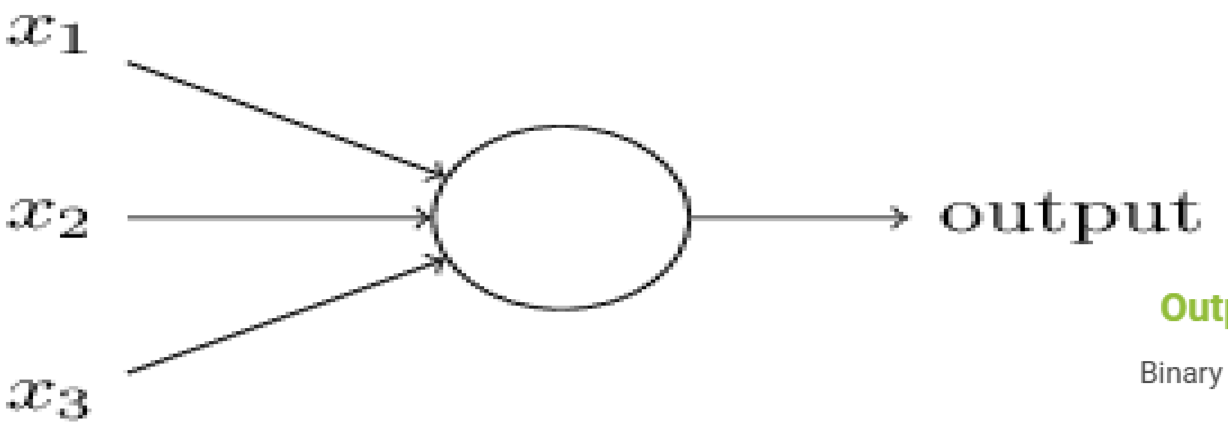
Keras

PYTORCH

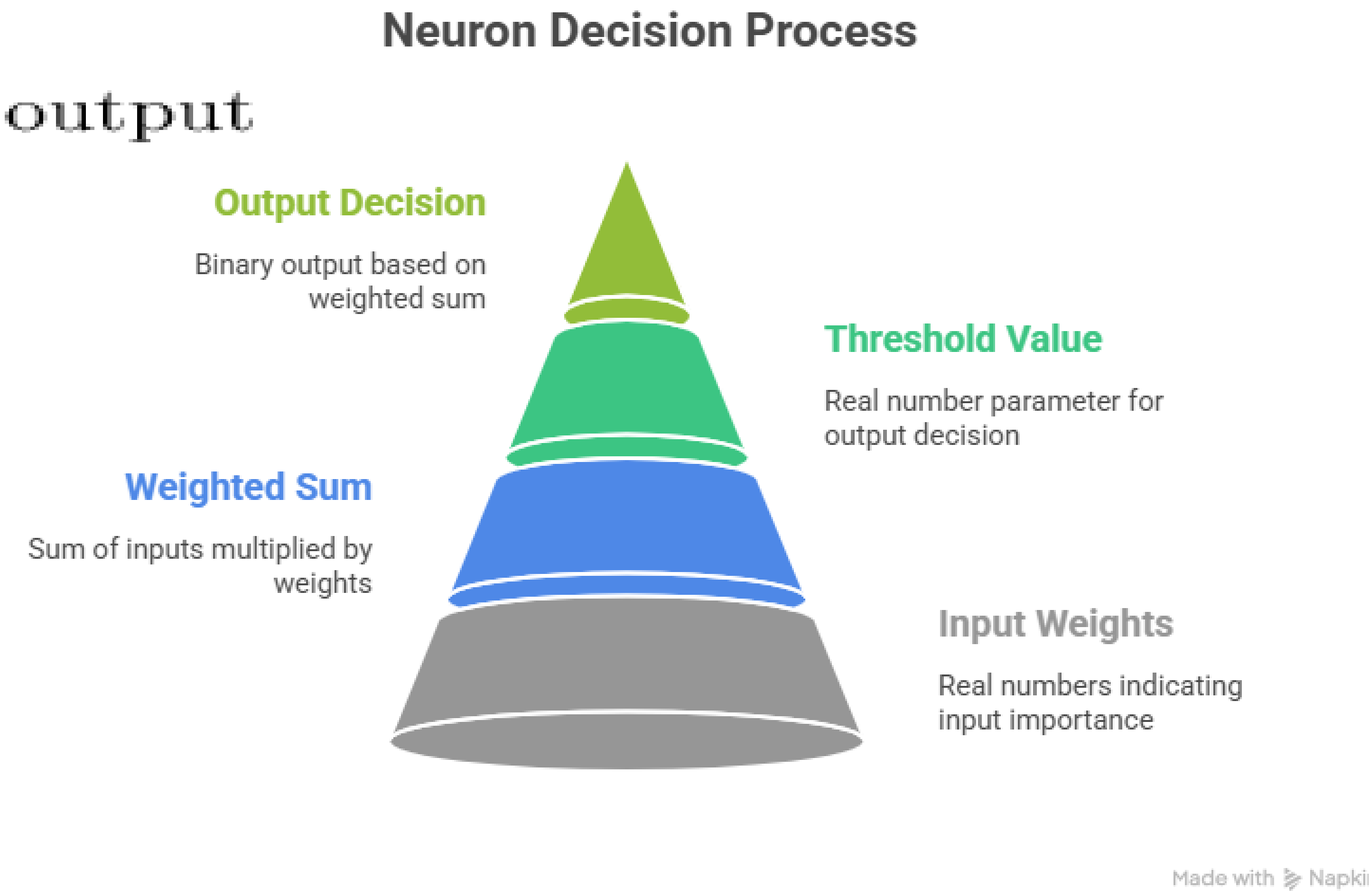
TensorFlow

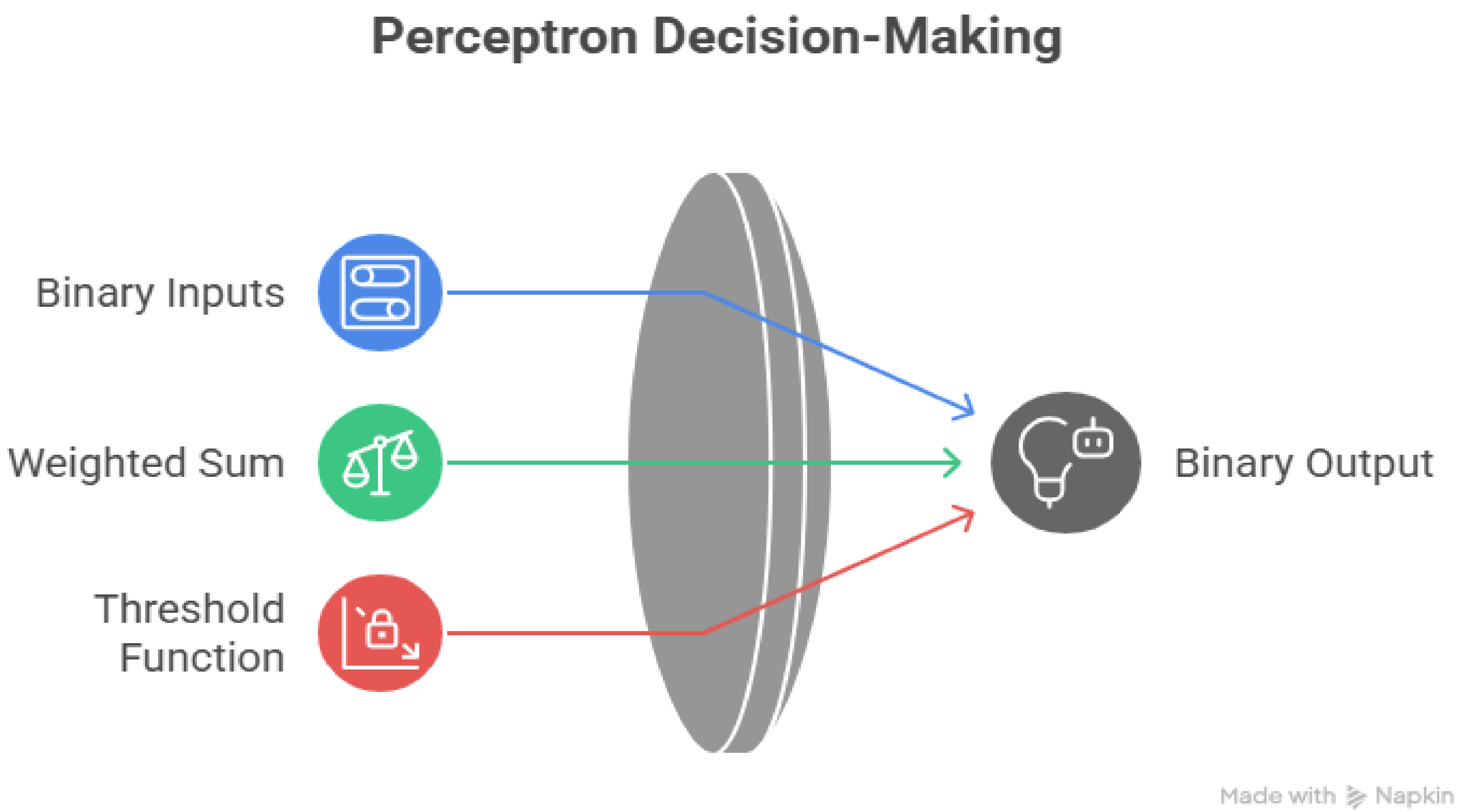
Caffe



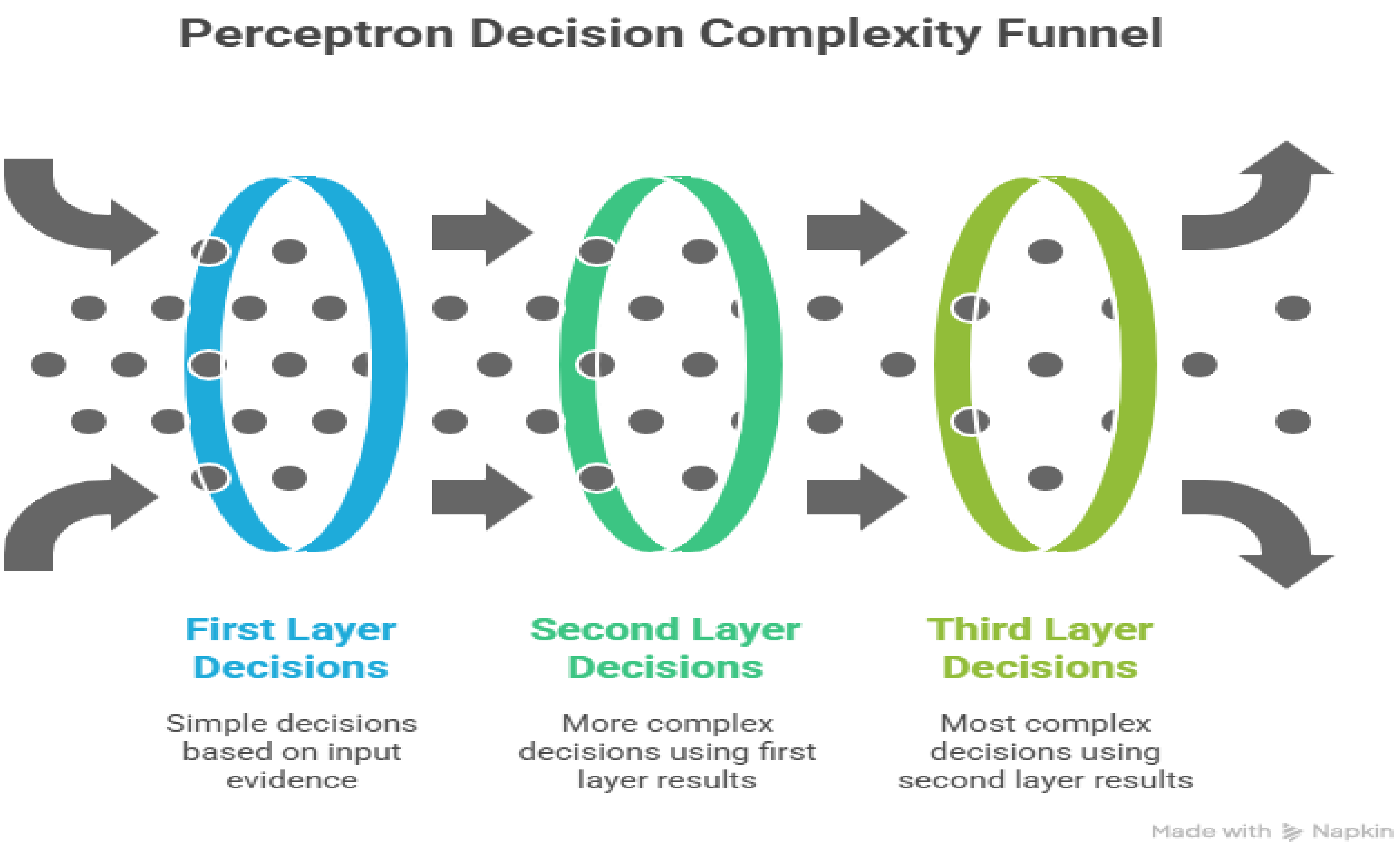


$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

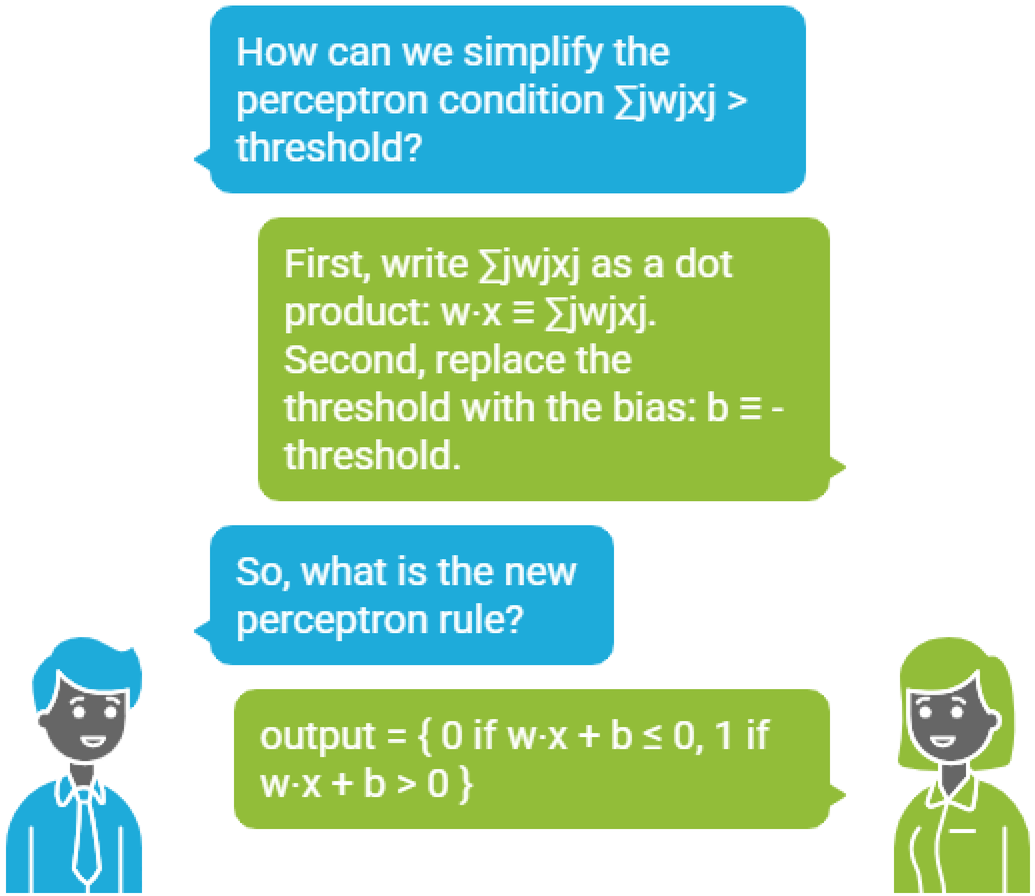




Perceptron Decision-Making
Binary Inputs
Weighted Sum
Threshold Function
Binary Output
Made with Napkin


Perceptron Decision Complexity Funnel
First Layer Decisions
Simple decisions based on input evidence
Second Layer Decisions
More complex decisions using first layer results
Third Layer Decisions
Most complex decisions using second layer results
Made with Napkin

Simplifying Perceptron Notation



Key Differences

Aspect	Threshold	Bias
Position	On the right side of inequality	Added to the weighted sum (left side)
Sign	Positive value to exceed	Negative of threshold
Interpretation	"Bar to clear"	"Head start" or "baseline activation"
Training	Harder to treat uniformly	Treated like any other weight

Why Bias is Preferred

1. Simpler math during learning:

- Bias can be adjusted using the same learning rules as weights
- It's just another parameter to optimize

2. Cleaner notation:

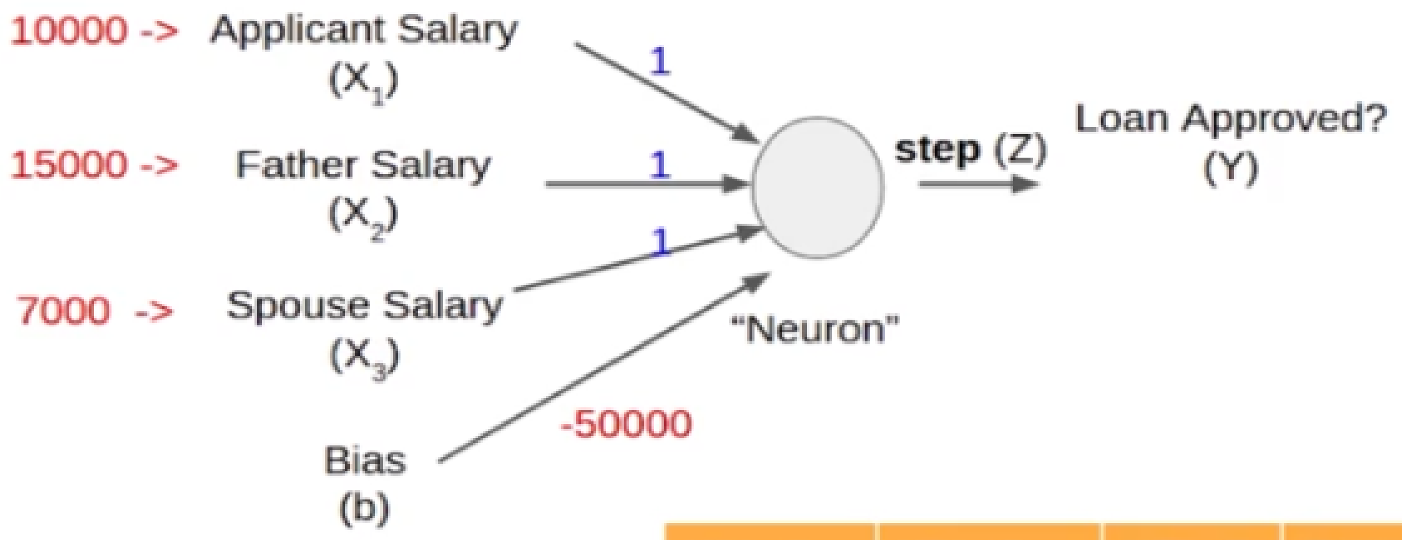
- Everything is on one side: $wx + b > 0$
- No need to move things across the inequality

3. Intuitive interpretation:

- Positive bias = perceptron is "eager" to fire (outputs 1 more easily)
- Negative bias = perceptron is "reluctant" to fire (needs more evidence)

Weights in Perceptron

Example 1:

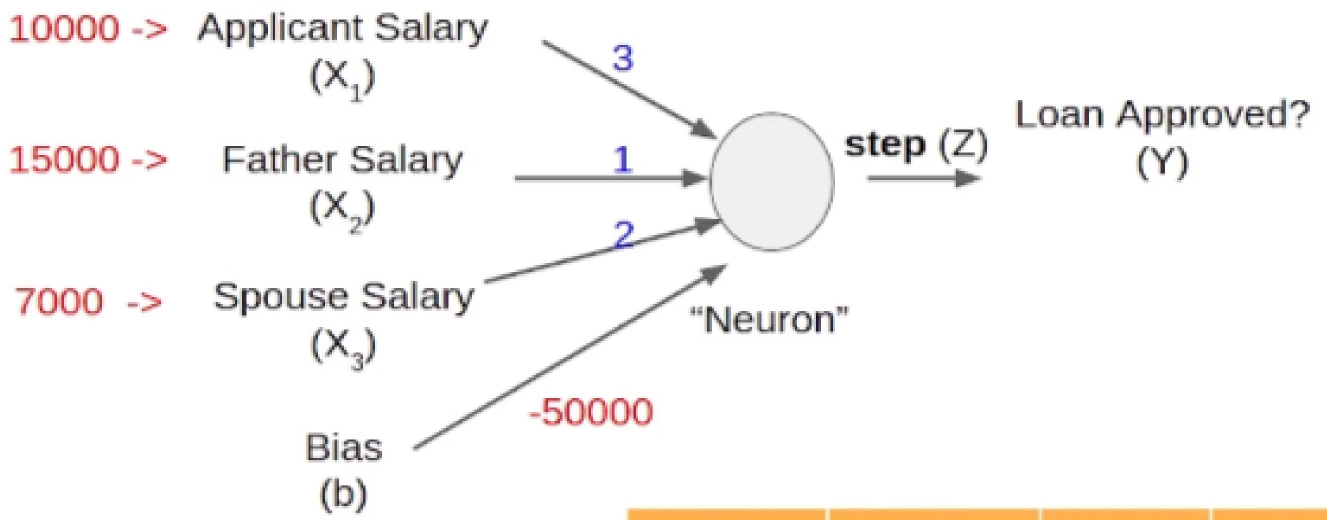


$$\text{Output} = \begin{cases} 1, & Z > 0 \\ 0, & Z \le 0 \end{cases}$$

X1*w1	X2*w2	X3*w3	Sum of inputs	Z (Sum of inputs + bias)	step (Z)
10000*1	15000*1	7000*1	32000	-18000	0

Weights in Perceptron

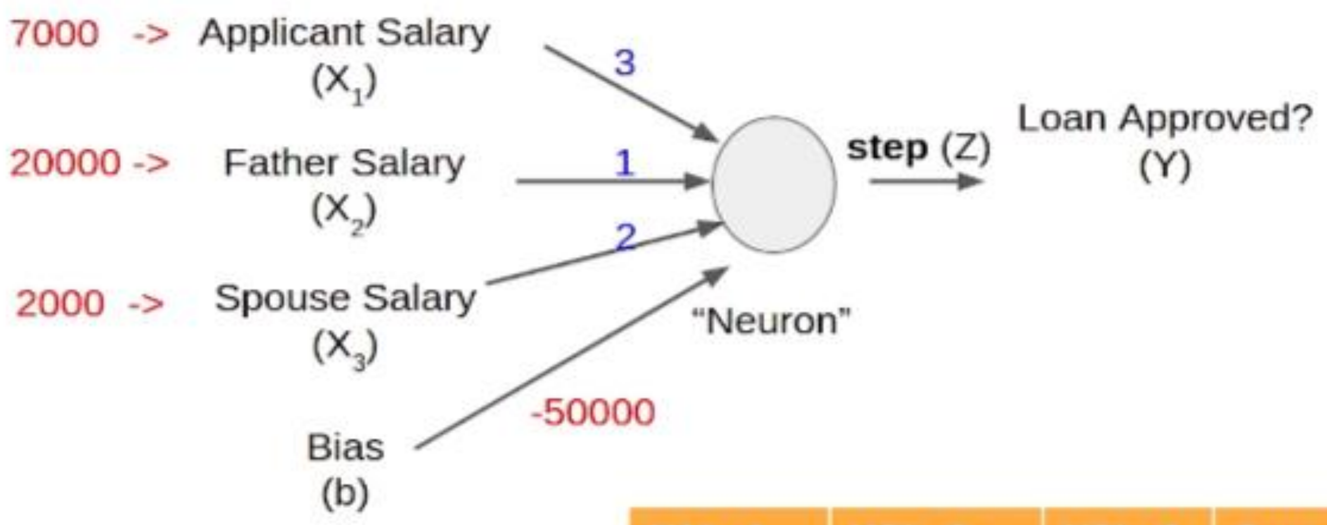
Example 2:



X1*w1	X2*w2	X3*w3	Sum of inputs	Z (Sum of inputs + bias)	step (Z)
10000*3	15000*1	7000*2	59000	9000	

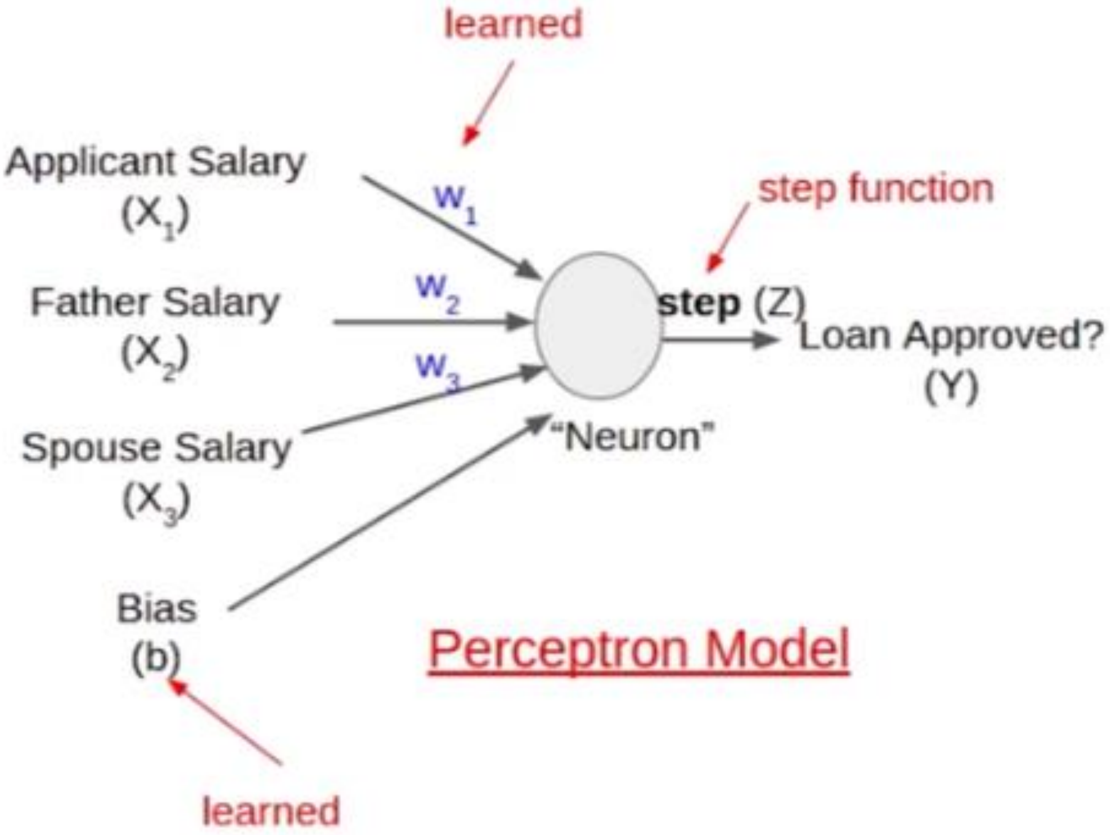
Weights in Perceptron

Example 3:



X1*w1	X2*w2	X3*w3	Sum of inputs	Z (Sum of inputs + bias)	step (Z)

Perceptron Model

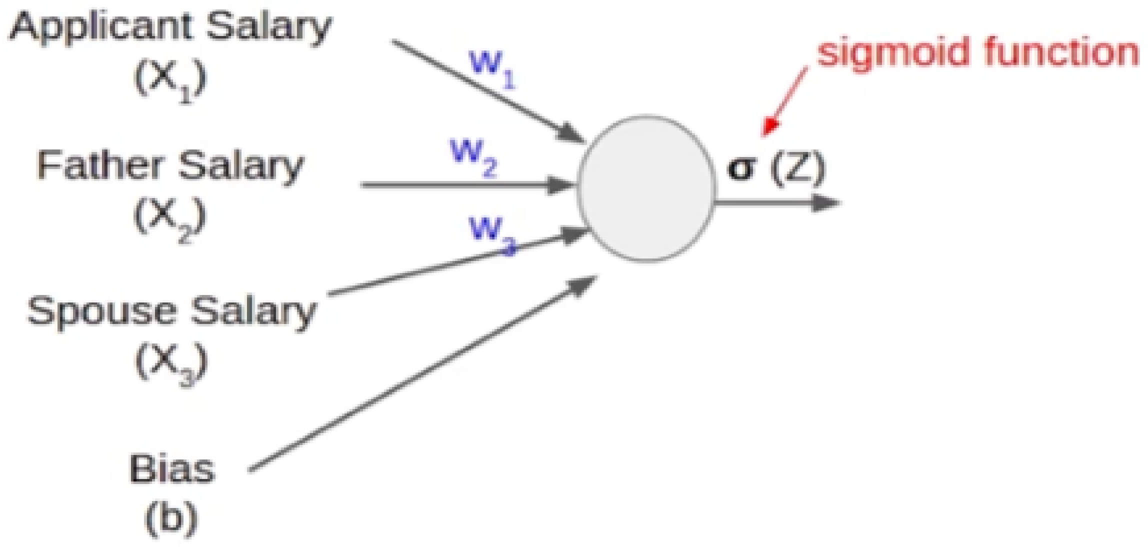


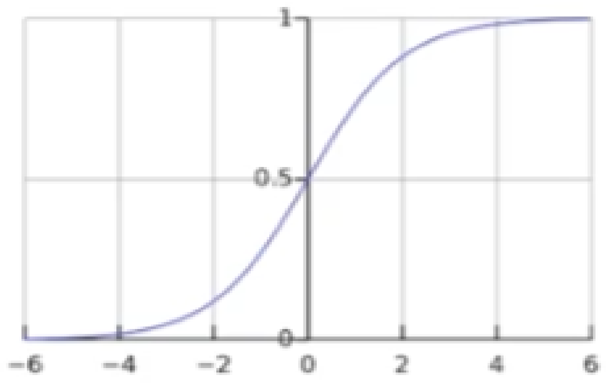
Sum of inputs = $X_1 * w_1 + X_2 * w_2 + X_3 * w_3$

$Z = X_1 * w_1 + X_2 * w_2 + X_3 * w_3 + b$ (bias)

$\hat{Y}$ (output) = **step** (Z)

Sigmoid Activation Function



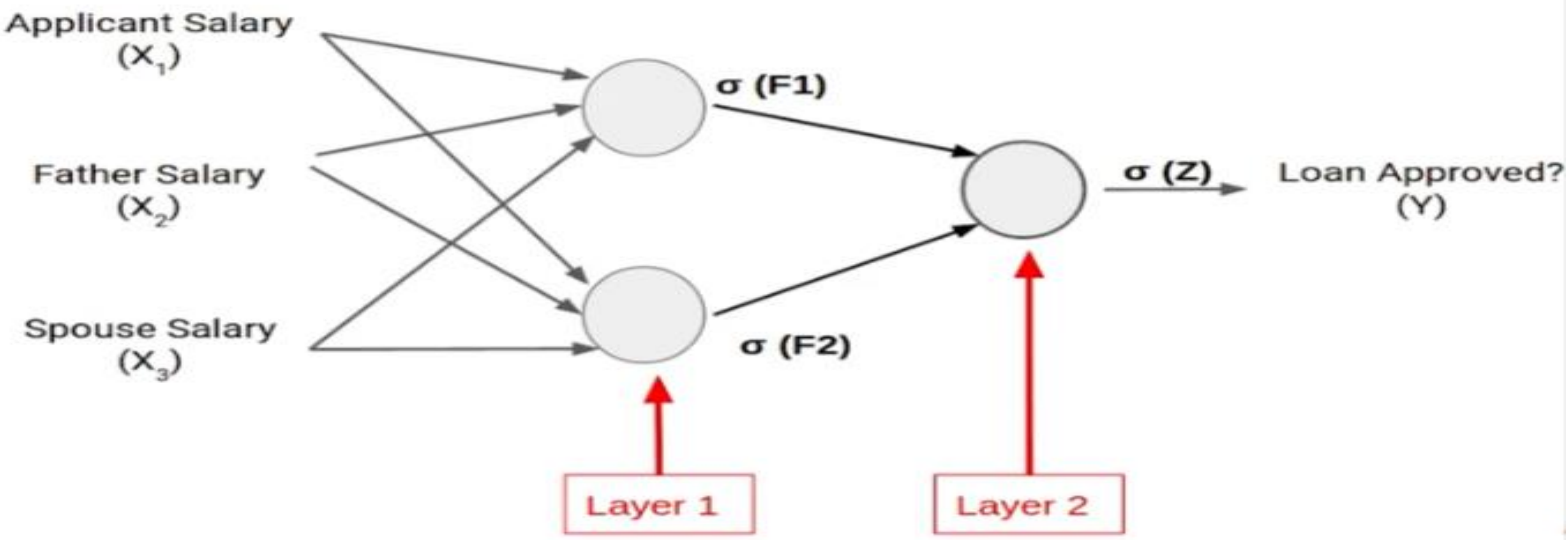


Sum of inputs = $X_1 * w_1 + X_2 * w_2 + X_3 * w_3$

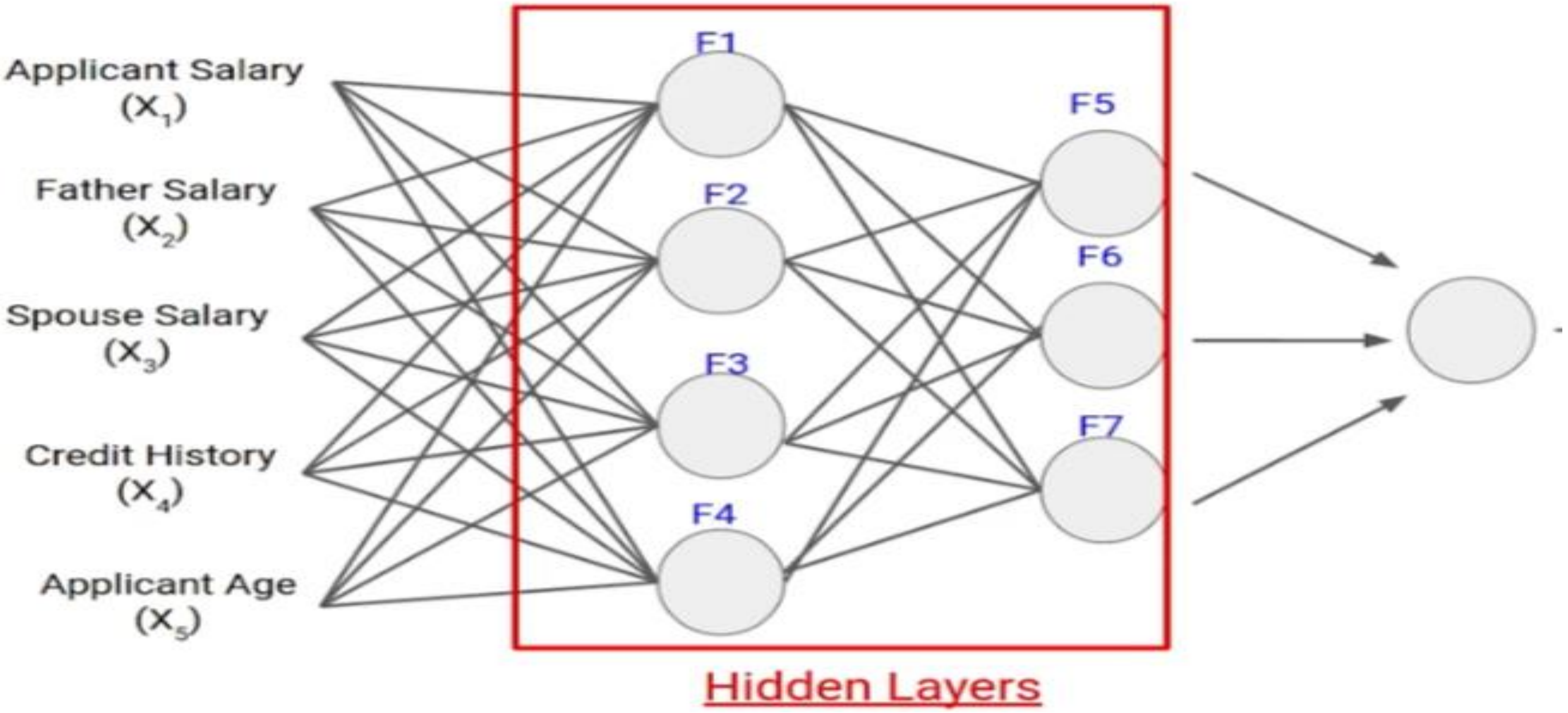
$Z = X_1 * w_1 + X_2 * w_2 + X_3 * w_3 + b$ (bias)

$\hat{Y}$ (output) = σ (Z)

Multi layer Perceptron

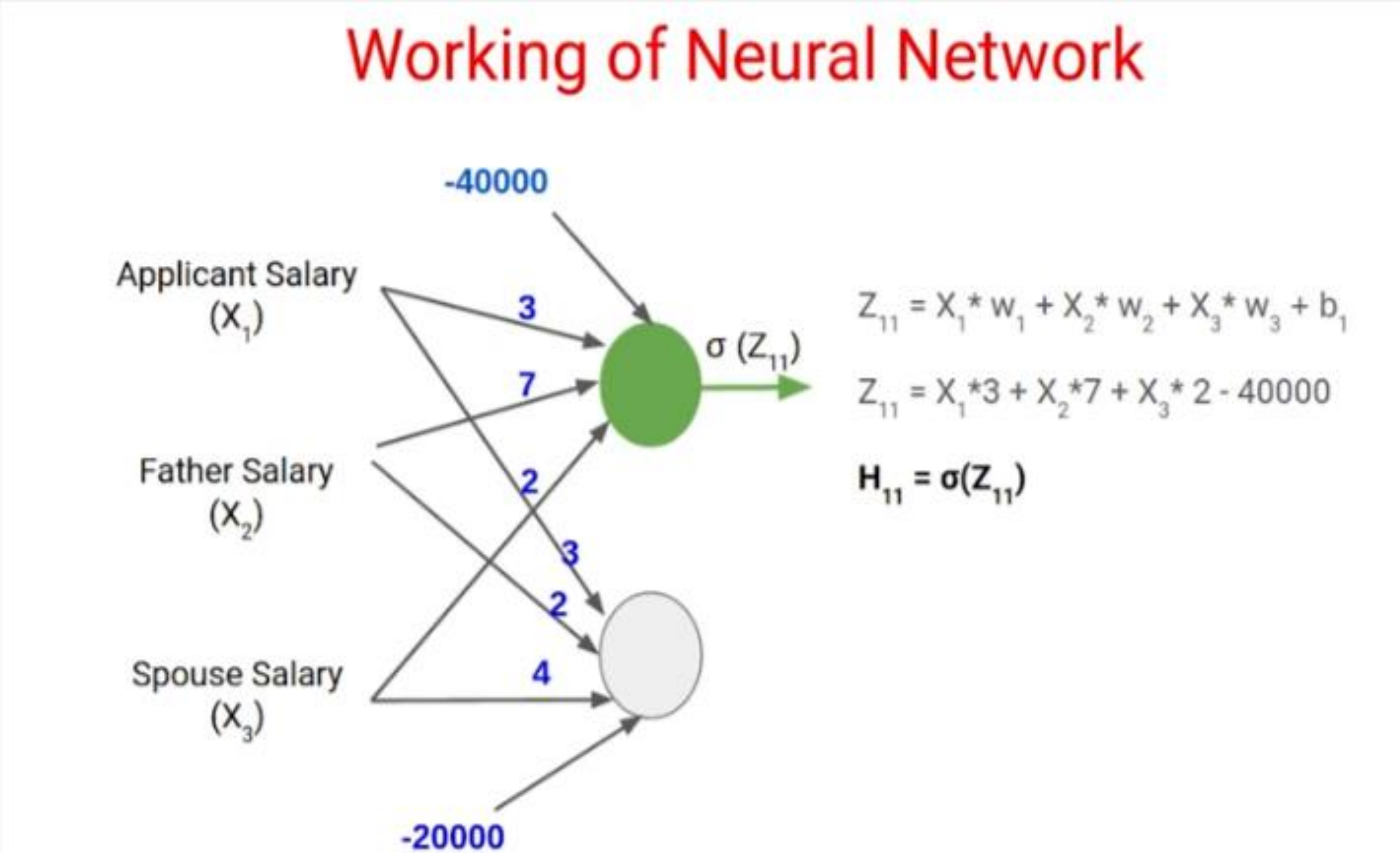


Multi layer Perceptron

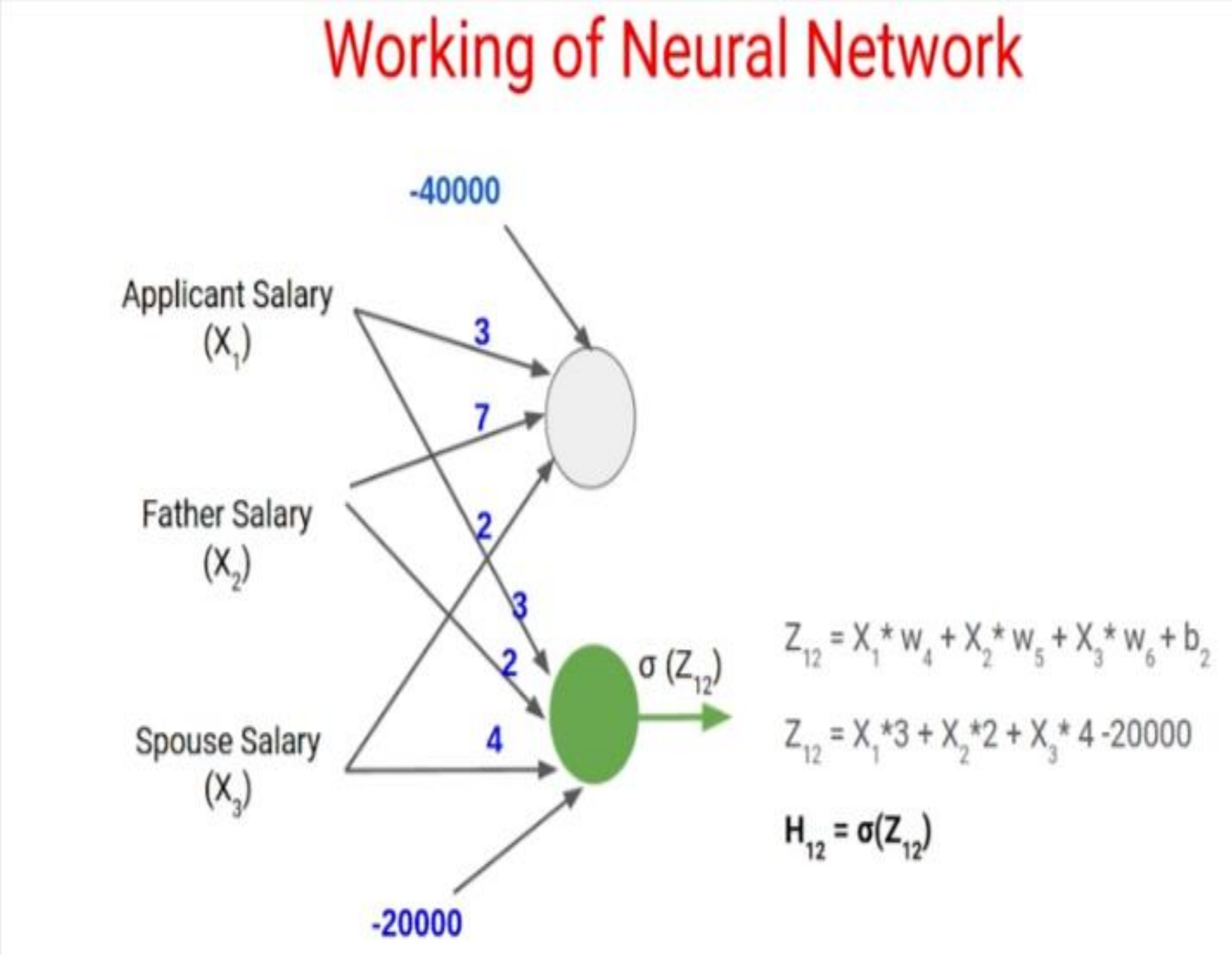


TFP:-https://playground.tensorflow.org/

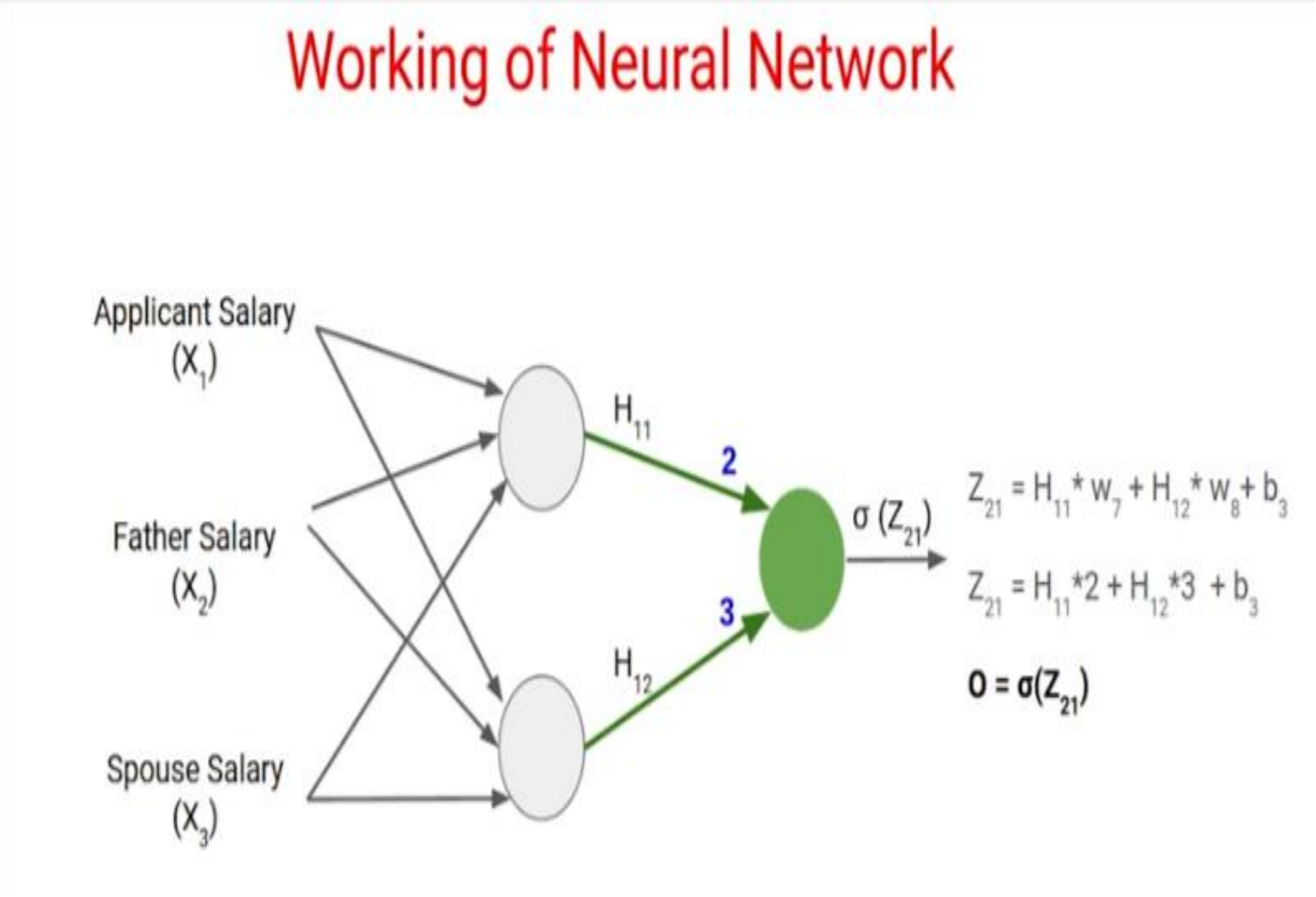
FORWARD PROPAGATION



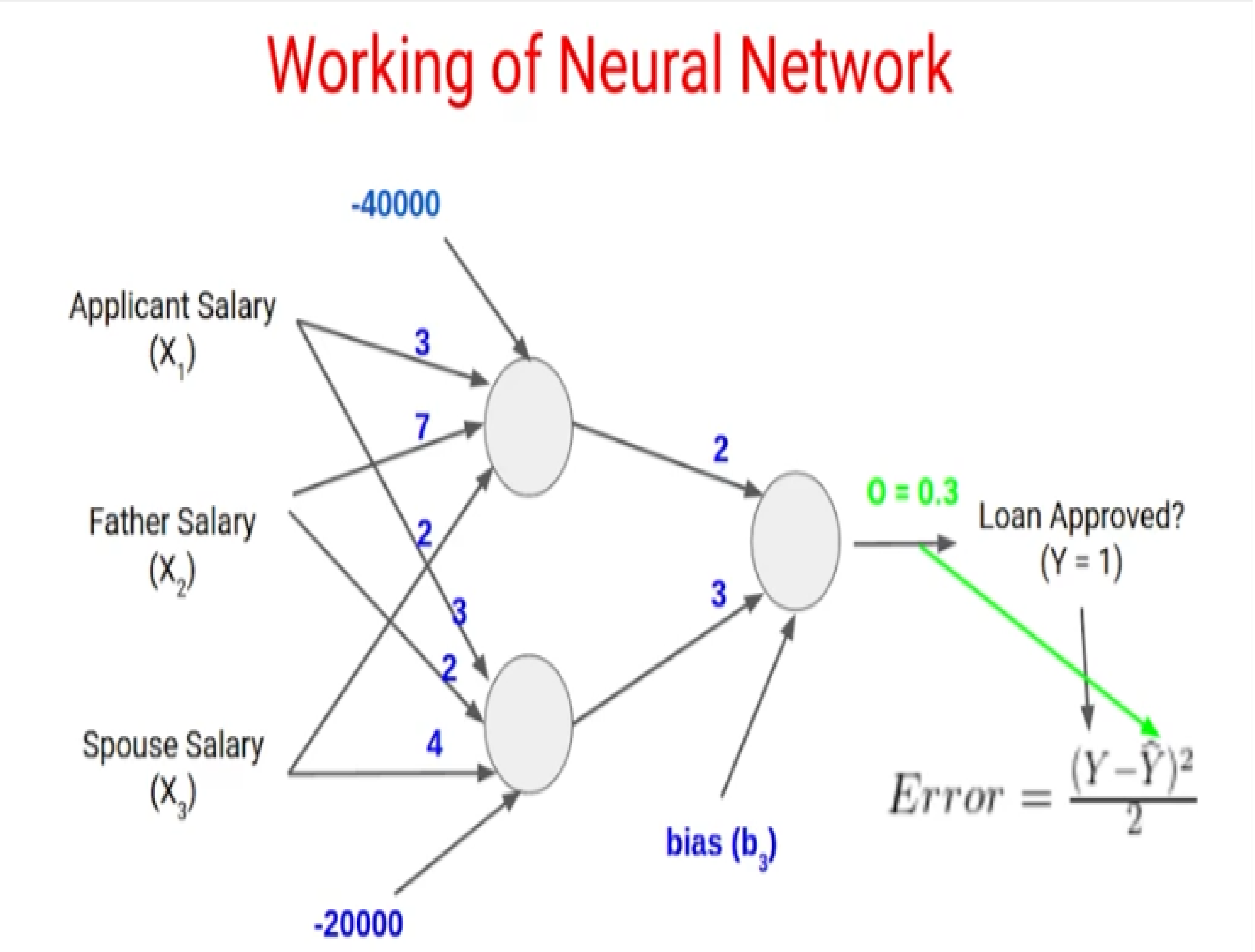
FORWARD PROPAGATION

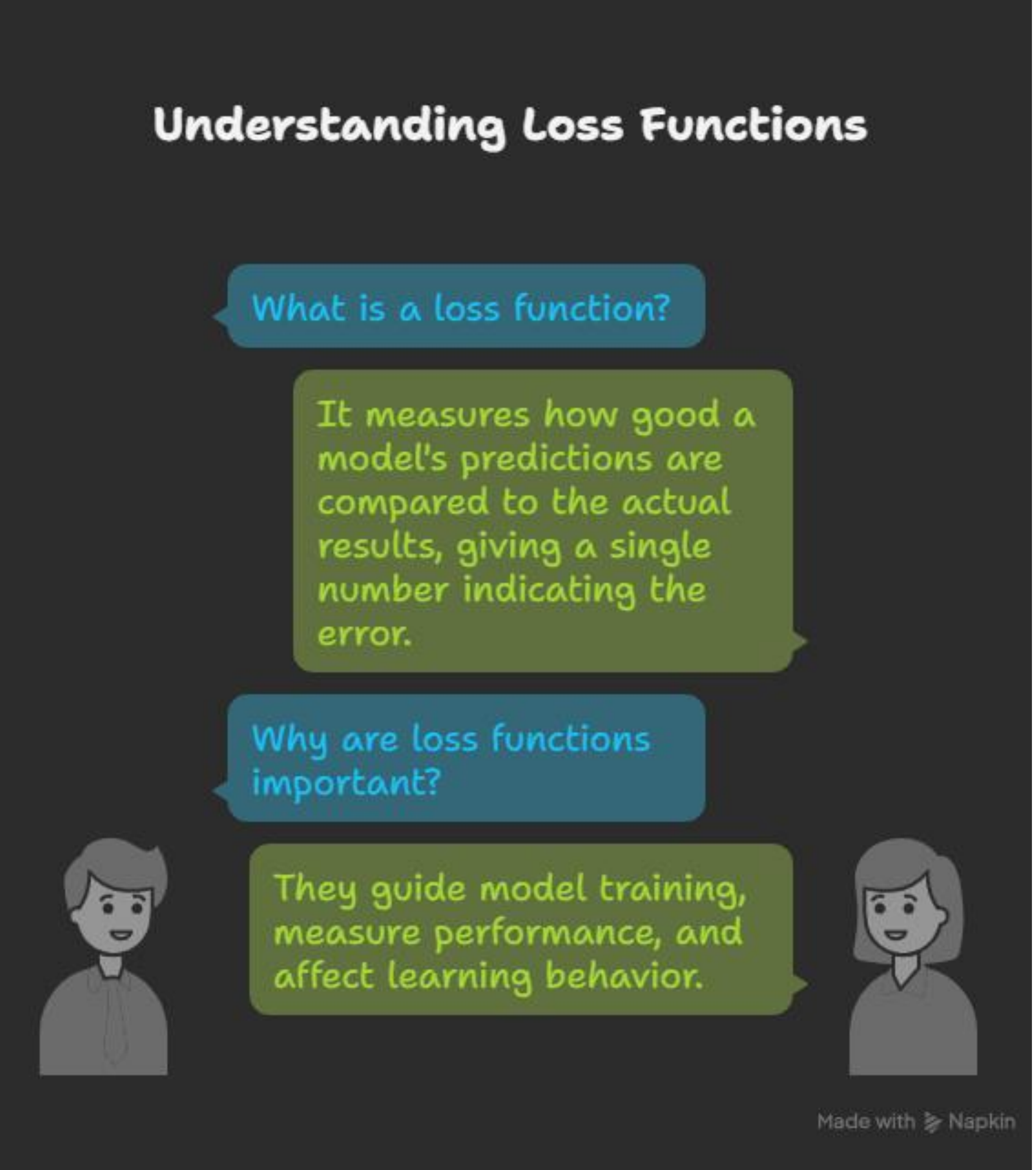


FORWARD PROPAGATION



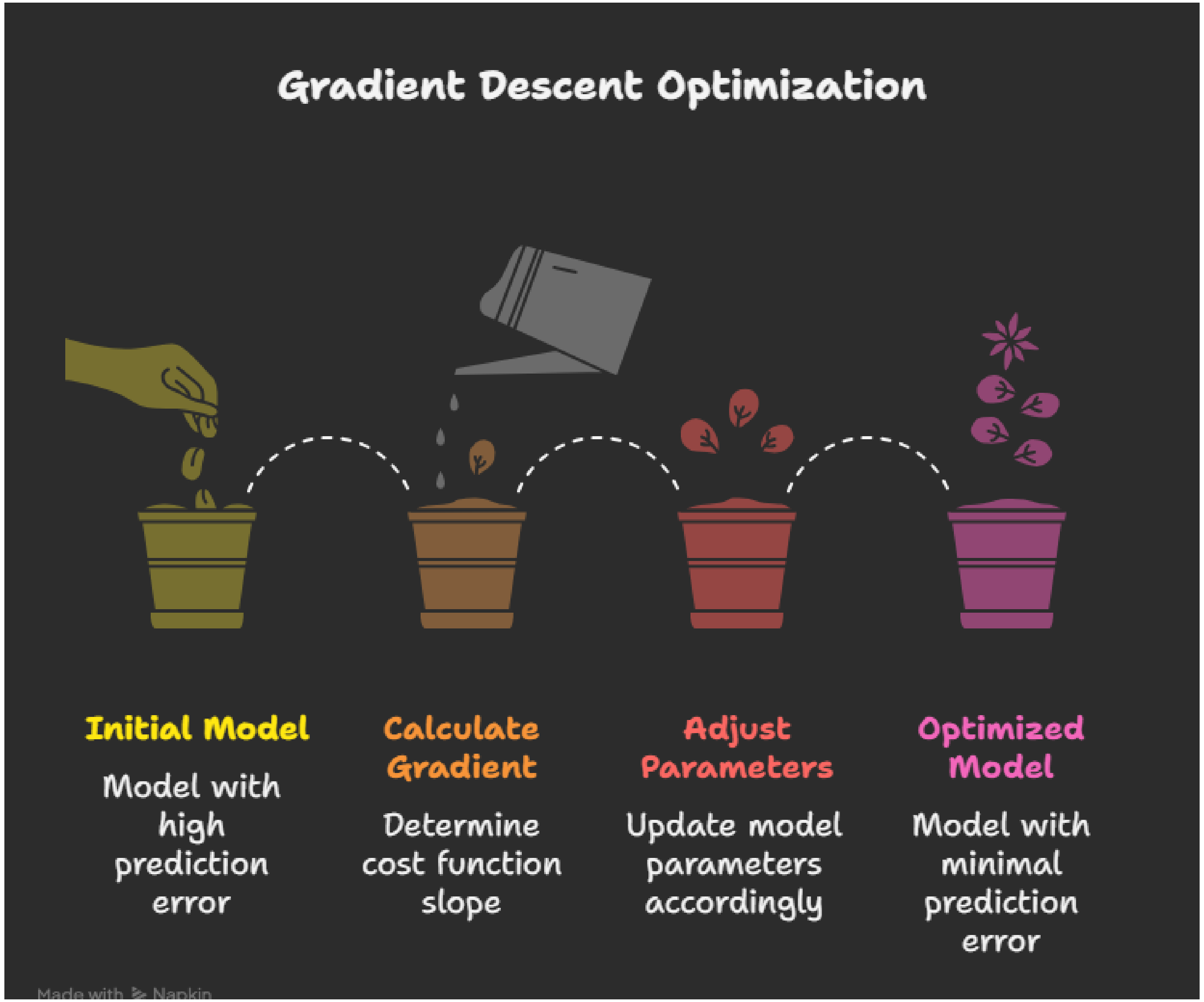
BACKWARD PROPAGATION

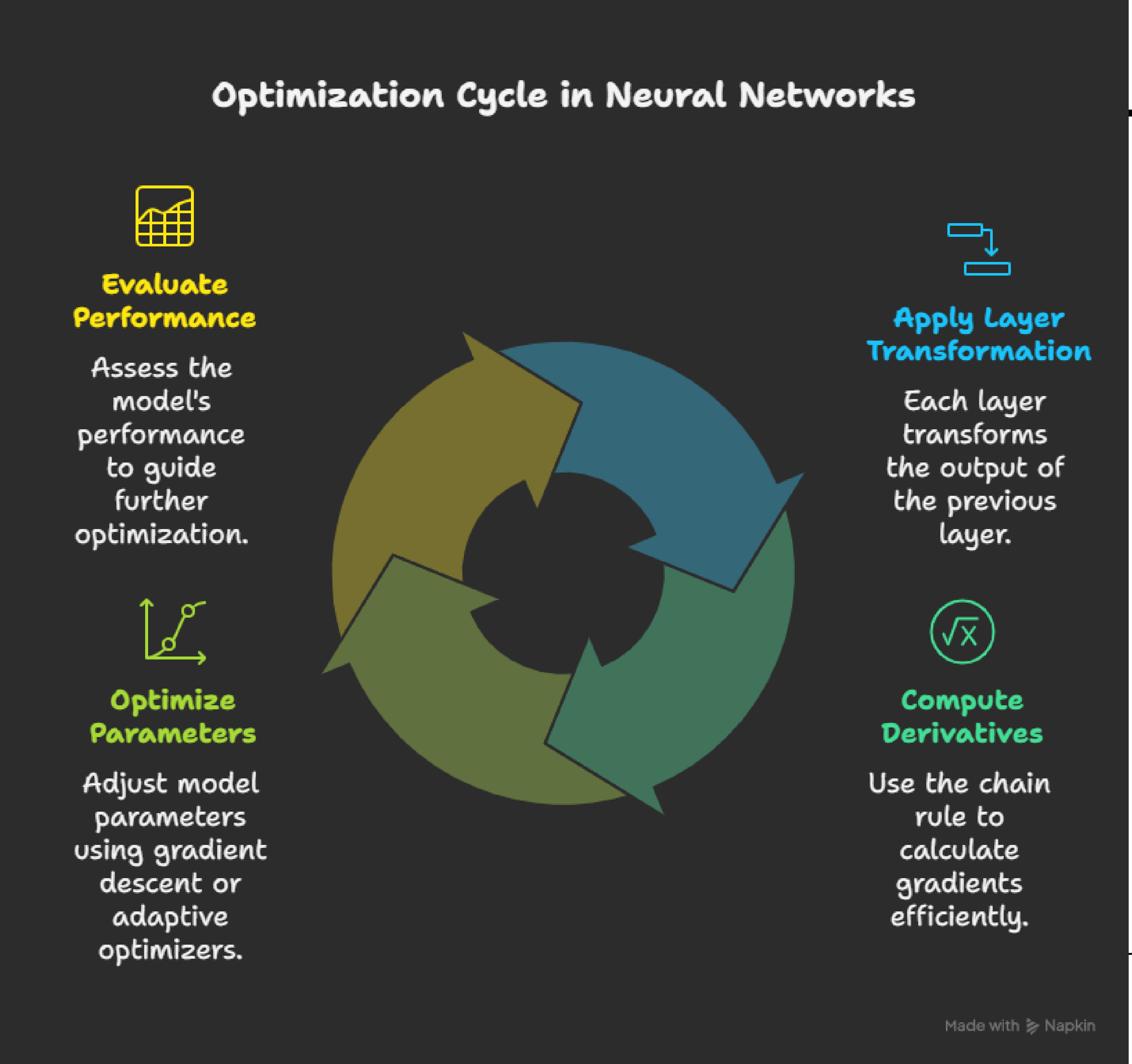




- A **loss function** is a **mathematical way to measure how good or bad a model's predictions are compared to the actual results**. It gives a single number that tells us how far off the predictions are. **The smaller the number, the better the model is doing**. Loss functions are used to train models.

- GD serves as a fundamental optimization technique to minimize the cost function of a model by **iteratively adjusting the model parameters to reduce the difference between predicted and actual values, improving the model's performance**.





Models are built from **multiple layers** where **each layer applies a transformation to the output of the previous layer**. The **chain rule allows us to efficiently compute derivatives (gradients) of complex, composite functions, which is important for optimizing** model parameters using methods such as gradient descent and adaptive optimizers (Adam, RMSProp).

if we have a function, ***y=f(g(x))***, where **g is a function of x and f is a function of g,** then the **derivative of y with respect to x is given by:**

$$\frac{dy}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

This means that the chain rule enables us to compute:

- The derivative of the loss with respect to output.
- The derivative of output with respect to weights (and biases), layer by layer.

REGRESSION LOSS FUNCTIONS

These are used when your model needs to predict a continuous number, such as predicting the price of a product or the age of a person.

Mean Squared Error:- It measures the **average of the squares of the errors**—that is, the **average squared difference between the predicted and actual values**.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where:

- y_i is the actual value (true value).
- $\hat{y}_i$ is the predicted value (from the model).
- n is the total number of data points.

Mean Absolute Error (MAE) Loss:-It calculates the average of the absolute differences between the predicted values and the actual values

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |\ y_i - \widehat{y_i}\ |$$

Huber Loss:- combines the advantages of MSE and MAE. **It is less sensitive to outliers than MSE** and **differentiable everywhere unlike MAE**. It requires tuning of the parameter δ

$$\begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2 & \text{for } |y_i - \hat{y}_i| \leq \delta \\ \delta|y_i - \hat{y}_i| - \frac{1}{2}\delta^2 & \text{for } |y_i - \hat{y}_i| > \delta \end{cases}$$

Binary Cross Entropy/Log Loss for Binary Classification

It **quantifies the difference between the actual class labels (0 or 1)** and the **predicted probabilities output by the model.** The **lower the binary cross-entropy value, the better the model's predictions align with the true labels**.

Mathematically, Binary Cross-Entropy (BCE) is defined as:

$$\text{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)\right]$$

where:

- N is the number of observations
- y_i is the actual binary label (0 or 1) of the i^{th} observation.
- p_i is the predicted probability of the i^{th} observation being in class 1.

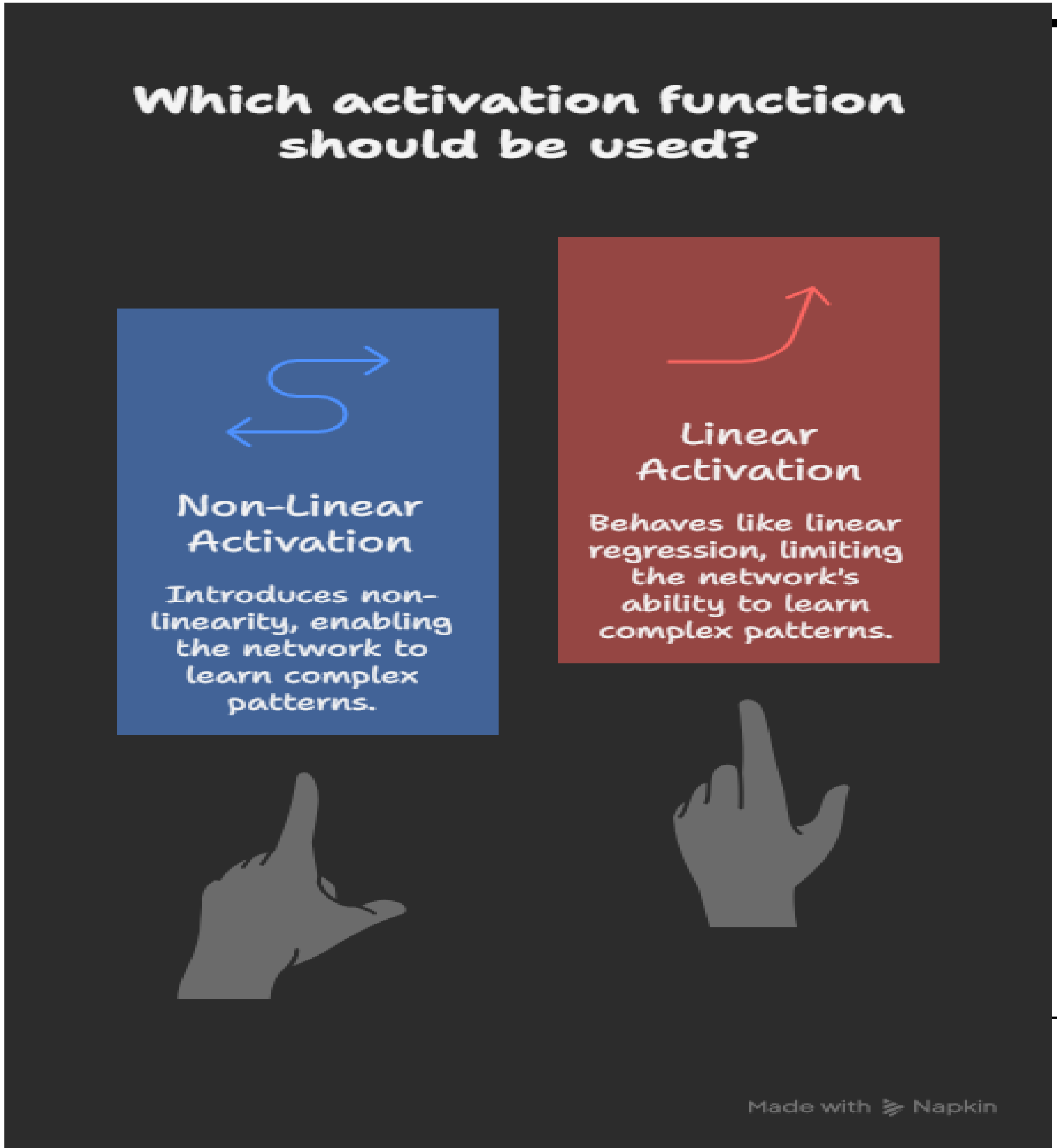
Categorical Cross-Entropy Loss

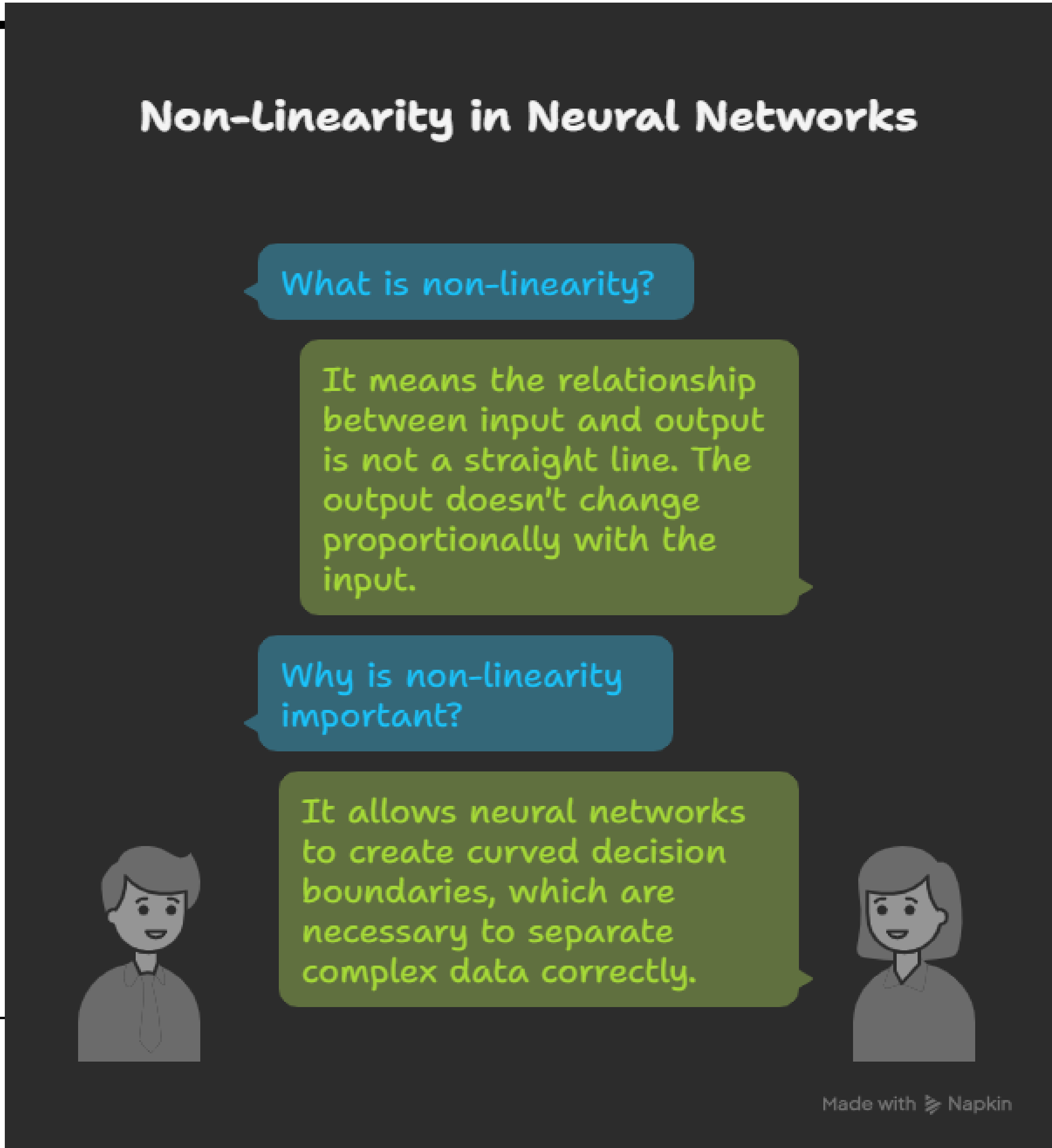
Used for **multiclass classification problems**. It **measures the performance of a classification model** whose **output is a probability distribution over multiple classes**.

$$\text{Categorical Cross-Entropy} = -\sum_{i=1}^{n}\sum_{j=1}^{k} y_{ij}\log(\hat{y}_{ij})$$

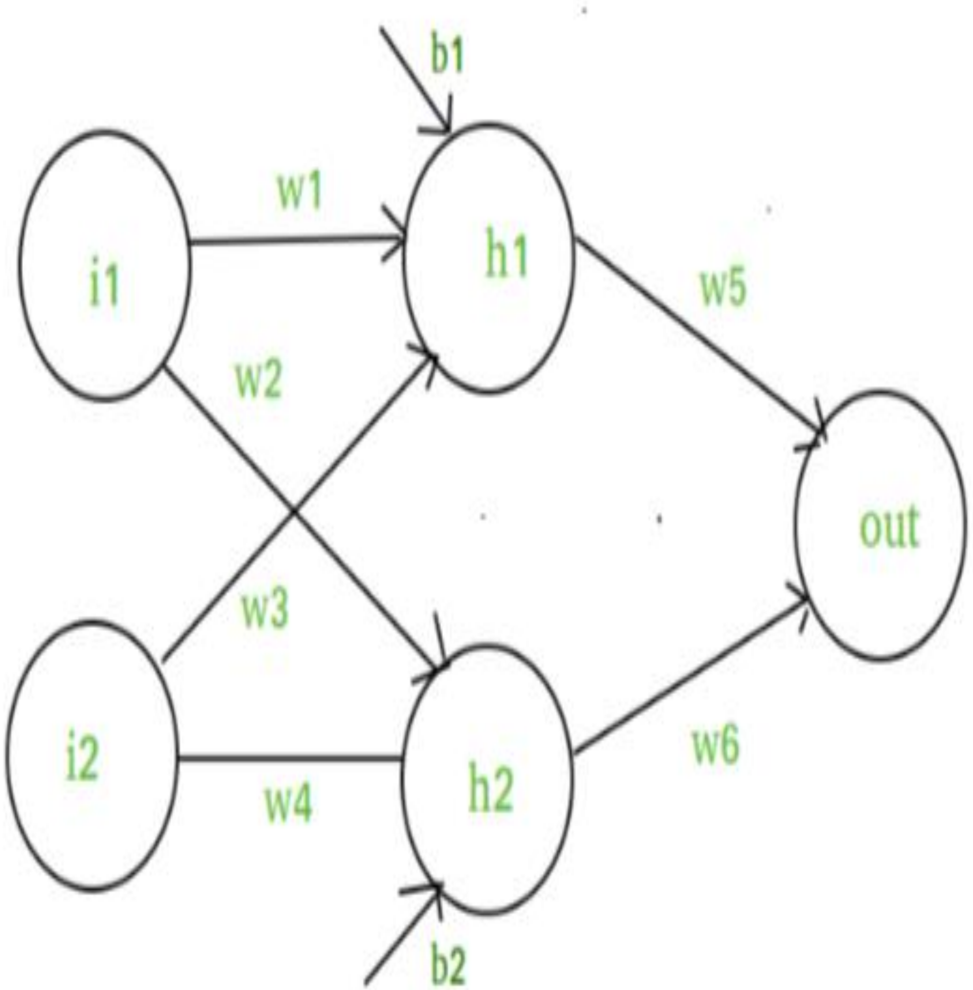
where:

- n is the number of data points
- k is the number of classes,
- y_{ij} is the binary indicator (0 or 1) if class label j is the correct classification for data point i
- $\hat{y}_{ij}$ is the predicted probability for class j.


Which activation function should be used?
Non-Linear Activation
Introduces non-linearity, enabling the network to learn complex patterns.
Linear Activation
Behaves like linear regression, limiting the network's ability to learn complex patterns.
Made with Napkin


Non-Linearity in Neural Networks
What is non-linearity?
It means the relationship between input and output is not a straight line. The output doesn't change proportionally with the input.
Why is non-linearity important?
It allows neural networks to create curved decision boundaries, which are necessary to separate complex data correctly.
Made with Napkin

1. **Input Layer**: Two inputs i_1 and i_2.
2. **Hidden Layer**: Two neuron h_1 and h_2
3. **Output Layer**: One output neuron.



The hidden layer outputs are:

$$h_1 = i_1.w_1 + i_2.w_3 + b_1$$

$$h_2 = i_1.w_2 + i_2.w_4 + b_2$$

The output before activation is:

$$\text{output} = h_1.w_5 + h_2.w_6 + \text{bias}$$

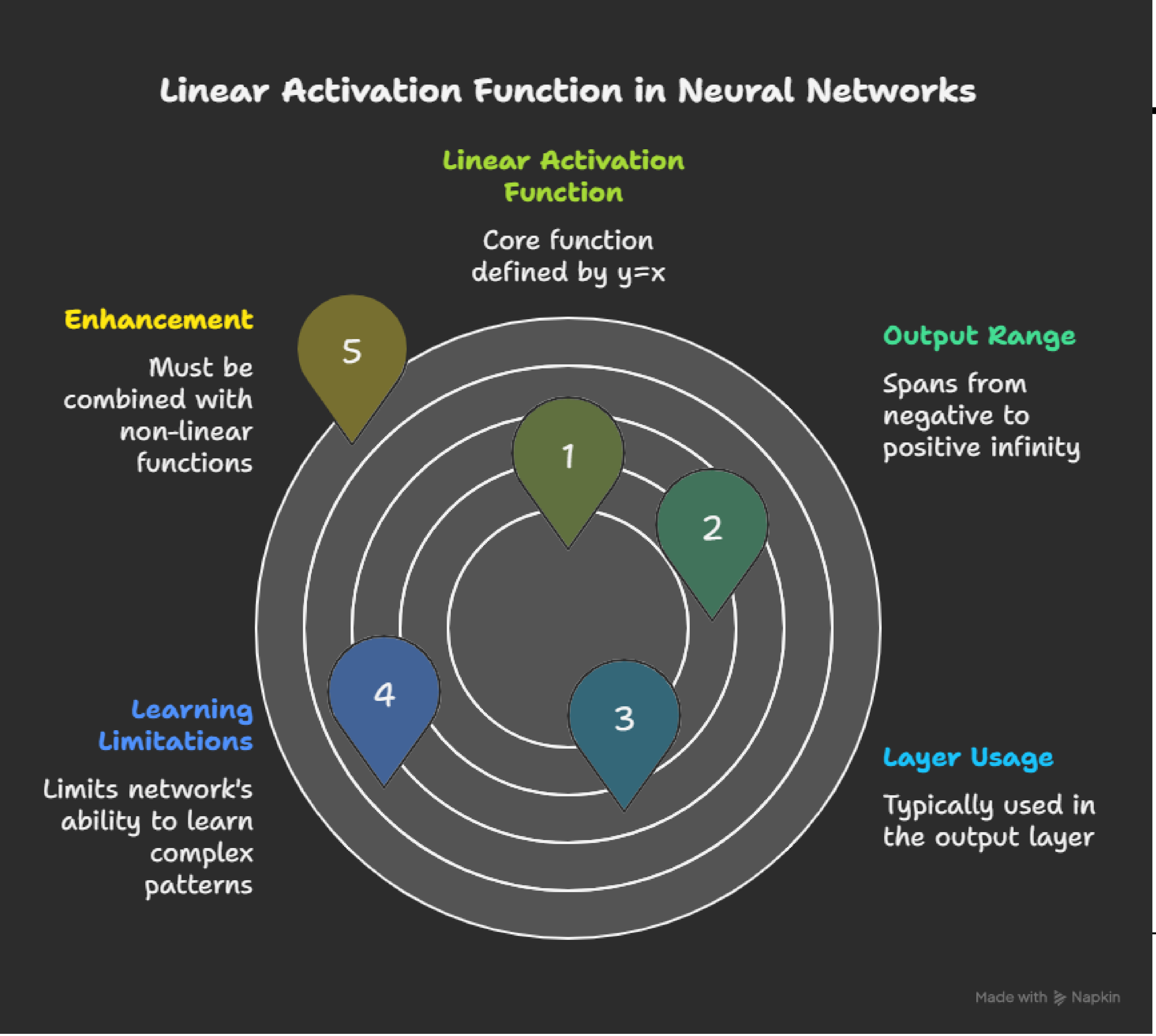
Without activation, these are linear equations.

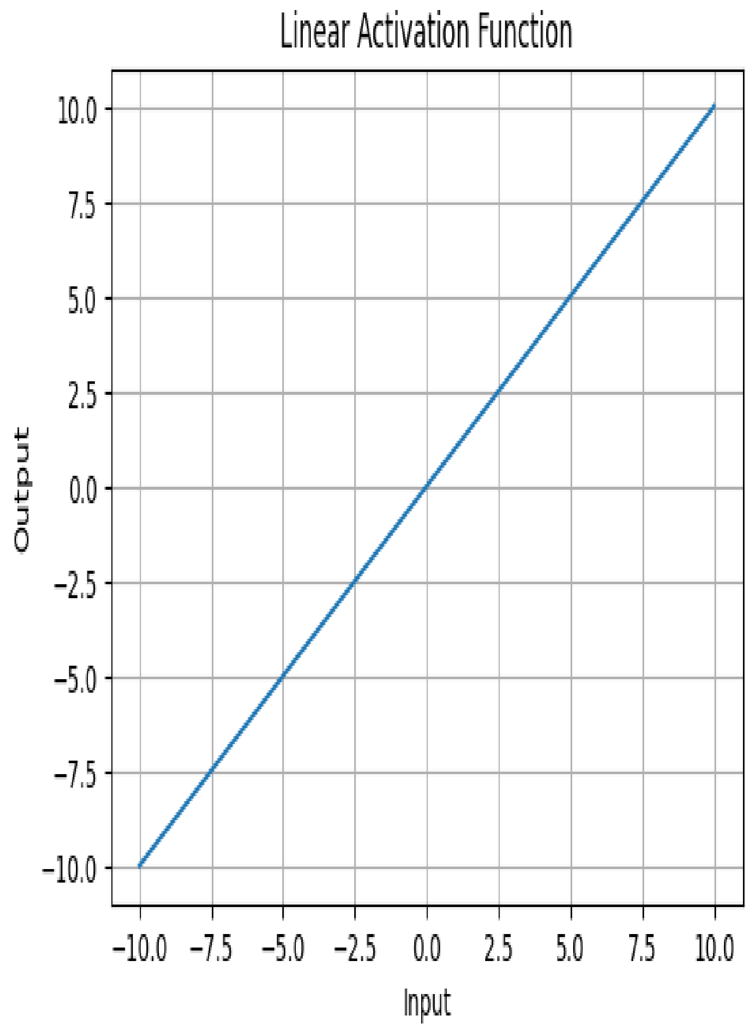
To introduce non-linearity, we apply a sigmoid activation:

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

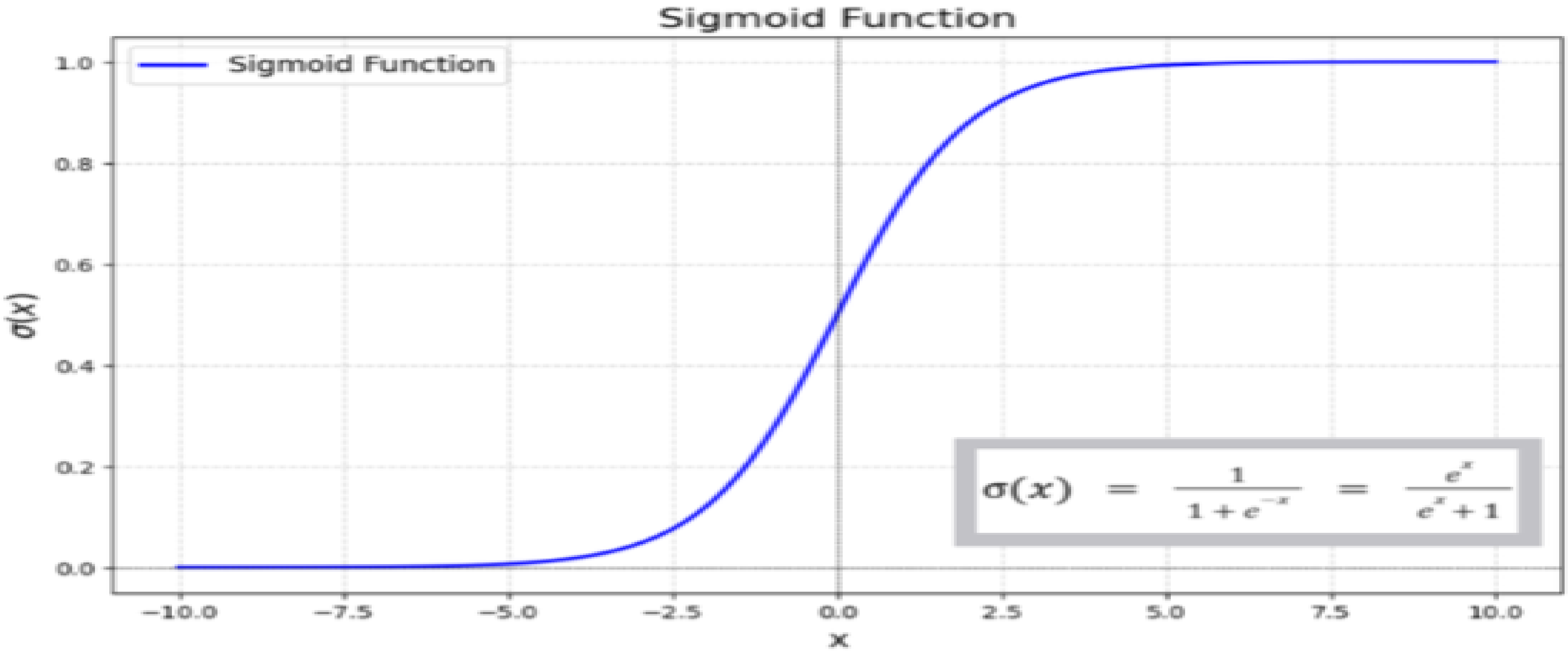
$$\text{final output} = \sigma(h_1.w_5 + h_2.w_6 + \text{bias})$$

This gives the final output of the network after applying the sigmoid activation function in output layers, introducing the desired non-linearity.


Linear Activation Function in Neural Networks
Linear Activation Function
Core function defined by y=x
Output Range
Spans from negative to positive infinity
Layer Usage
Typically used in the output layer
Learning Limitations
Limits network's ability to learn complex patterns
Enhancement
Must be combined with non-linear functions
1
2
3
4
5
Made with Napkin


Linear Activation Function
Output
Input
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Sigmoid function is used as an activation function in machine learning and neural networks for modeling binary classification problems, smoothing outputs, and introducing non-linearity into models.



Rule	Formula	Example
Constant Rule	$\frac{d}{dx}(c) = 0$	$\frac{d}{dx}(5) = 0$
Constant Multiple Rule	$\frac{d}{dx}(c \cdot f(x)) = c \cdot f'(x)$	$\frac{d}{dx}(3x^4) = 3 \cdot \frac{d}{dx}(x^4)$
Sum and Difference Rule	$\frac{d}{dx}(f \pm g) = f'(x) \pm g'(x)$	$\frac{d}{dx}(x^2 + \sin x) = 2x + \cos x$
Power Rule	$\frac{d}{dx}(x^n) = nx^{n-1}$	$\frac{d}{dx}(x^5) = 5x^4$
Product Rule	$\frac{d}{dx}(f \cdot g) = f'g + fg'$	$\frac{d}{dx}(x \cdot e^x) = (1)e^x + x(e^x)$
Quotient Rule	$\frac{d}{dx}\left(\frac{f}{g}\right) = \frac{f'g - fg'}{g^2}$	$\frac{d}{dx}\left(\frac{\sin x}{x}\right) = \frac{(\cos x)x - (\sin x)(1)}{x^2}$
Chain Rule	$\frac{d}{dx}(f(g(x))) = f'(g(x)) \cdot g'(x)$	$\frac{d}{dx}(\sin(x^2)) = \cos(x^2) \cdot 2x$

Polynomial/Power Functions

Function $f(x)$	Derivative $\frac{d}{dx}f(x)$
x^n	nx^{n-1}
x	1
c (constant)	0

Exponential and Logarithmic Functions

Function $f(x)$	Derivative $\frac{d}{dx}f(x)$
e^x	e^x
a^x	$a^x \ln a$
$\ln	x
$\log_a x$	$\frac{1}{x \ln a}$

Trigonometric Functions

Function $f(x)$	Derivative $\frac{d}{dx}f(x)$
$\sin x$	$\cos x$
$\cos x$	$-\sin x$
$\tan x$	$\sec^2 x$
$\cot x$	$-\csc^2 x$
$\sec x$	$\sec x \tan x$
$\csc x$	$-\csc x \cot x$

Derivative of Sigmoid Function

The derivative of the sigmoid function, denoted as $\sigma'(x)$, is given by $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$.

Let's see how the derivative of sigmoid function is computed.

We know that, sigmoid function is defined as:

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

Define:

$$u = 1 + e^{-x}$$

Rewriting the sigmoid function:

$$y = \frac{1}{u}$$

Differentiating u with respect to x:

$$\frac{du}{dx} = -e^{-x}$$

Differentiating y with respect to u:

$$\frac{dy}{du} = -\frac{1}{u^2}$$

Using the chain rule:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

$$\frac{dy}{dx} = \left(-\frac{1}{u^2}\right) \cdot \left(e^{-x}\right)$$

$$\frac{dy}{dx} = \frac{e^{-x}}{u^2}$$

Since $u = 1 + e^{-x}$, substituting:

$$\frac{dy}{dx} = \frac{e^{-x}}{(1+e^{-x})^2}$$

Since:

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

Rewriting:

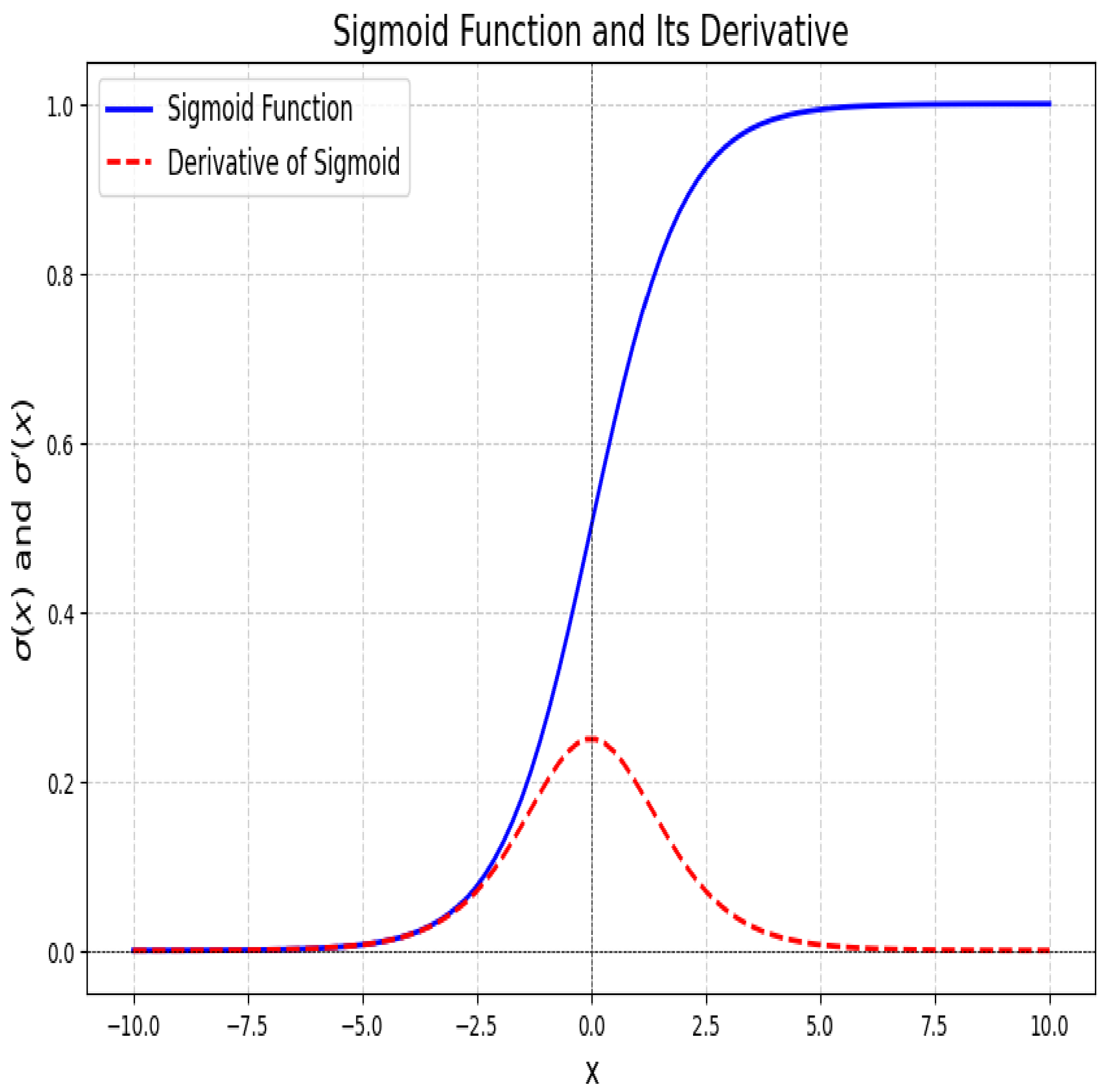
$$1 - \sigma(x) = \frac{e^{-x}}{1+e^{-x}}$$

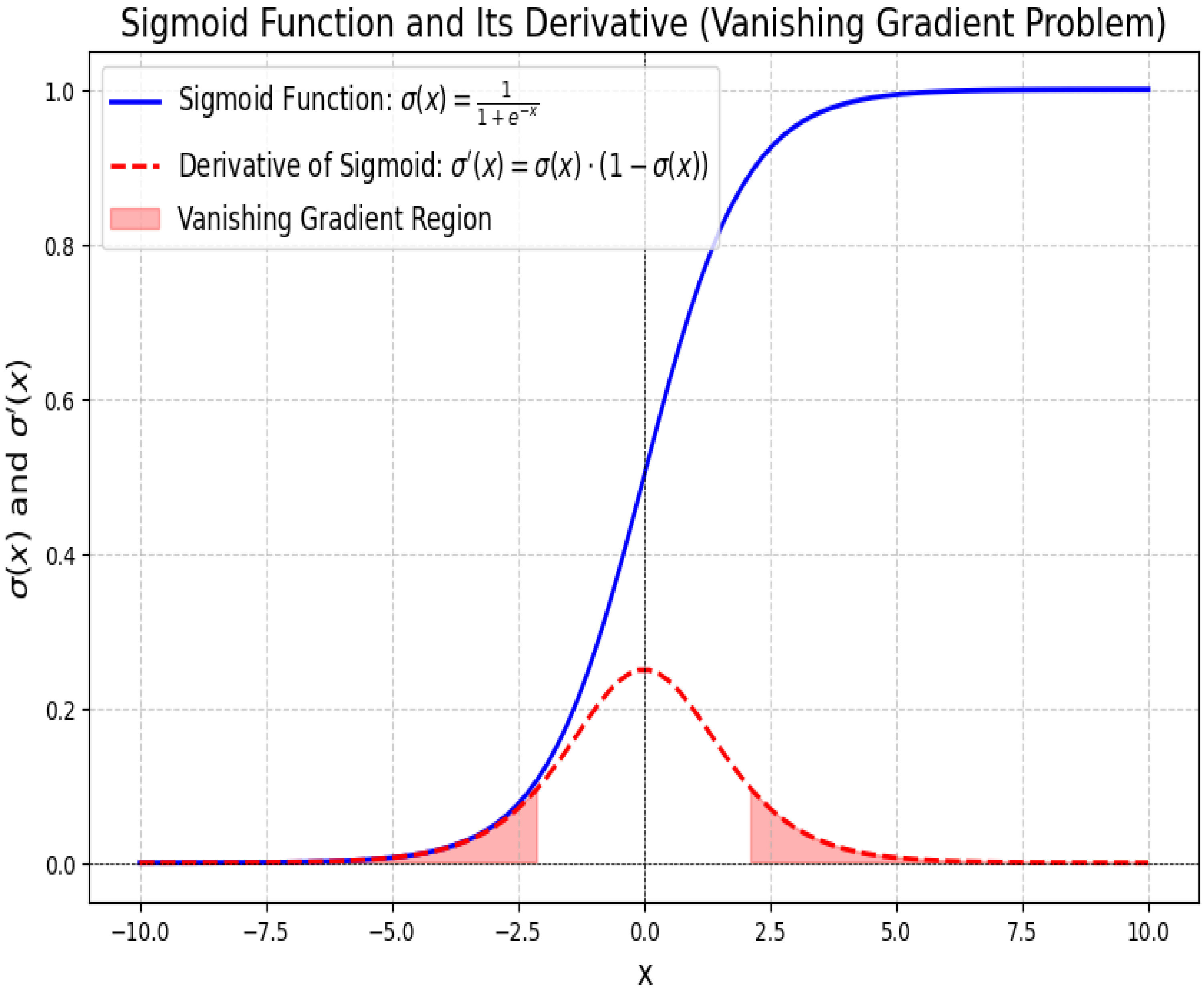
Substituting:

$$\frac{dy}{dx} = \sigma(x) \cdot (1 - \sigma(x))$$

Final Result

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$





The **shades red region** highlights the areas where the **derivative $\sigma'(x)\sigma'(x)$ is very small (close to 0).** In these regions, the **gradients used to update weights and biases during backpropagation become extremely small.** As a result, the **model learns very slowly or stops learning altogether, which is a major issue in deep neural networks.**

Tanh Activation Function

The **hyperbolic tangent function** is a **shifted version of the sigmoid**, allowing it to stretch across the y-axis.

$$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1.$$

Alternatively, it can be expressed using the sigmoid function:

$$\tanh(x) = 2 \times \text{sigmoid}(2x) - 1$$

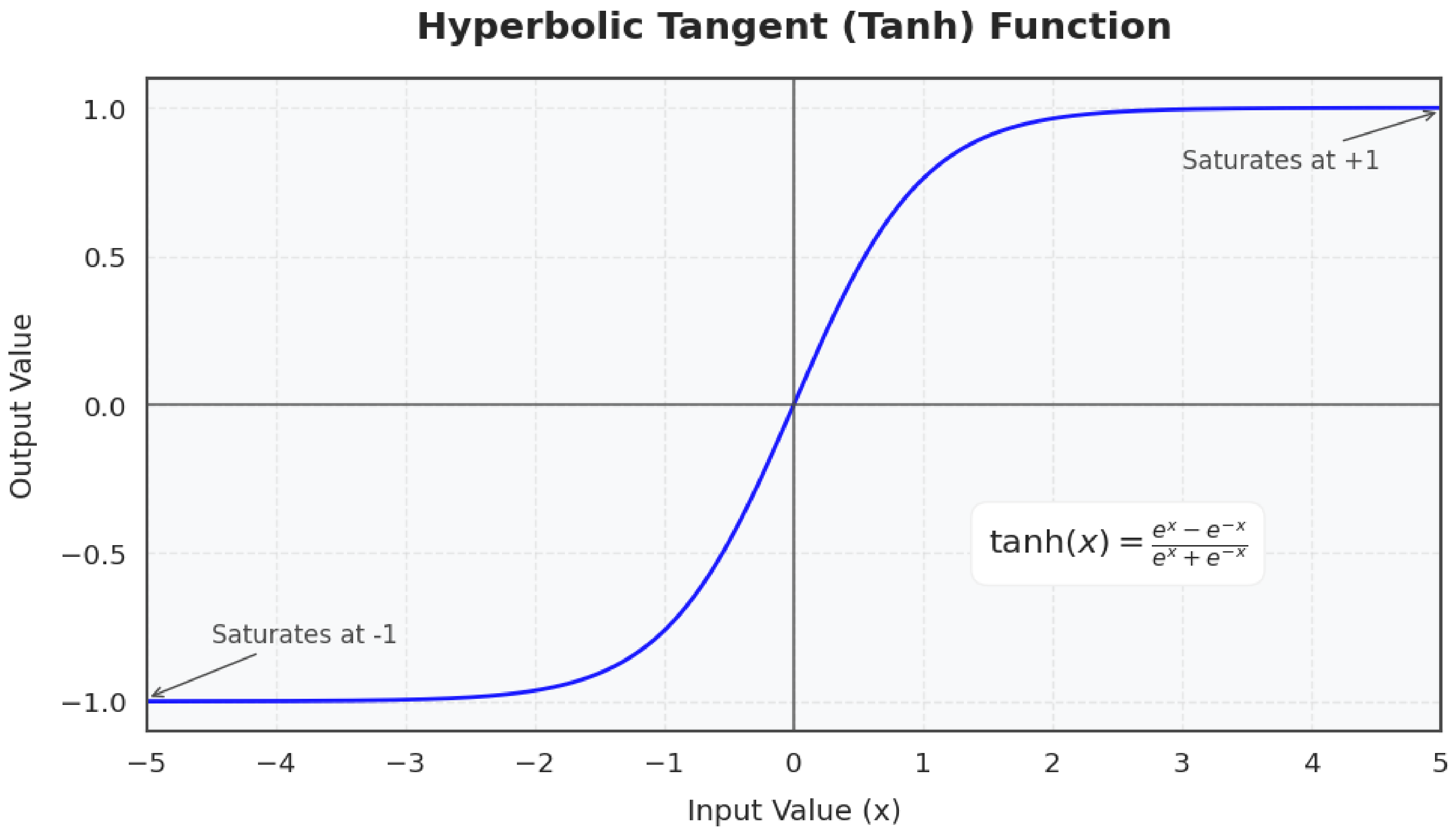
- **Value Range**: Outputs values from -1 to +1.
- **Non-linear**: Enables modeling of complex data patterns.
- **Use in Hidden Layers**: Commonly used in hidden layers due to its zero-centered output, facilitating easier learning for subsequent layers.

It is mathematically defined as:

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{\sinh(x)}{\cosh(x)}$$

Where:

- e is Euler's number (approximately 2.718).
- x is the input to the function.



$$\sinh x = \frac{e^x - e^{-x}}{2}$$

$$\cosh x = \frac{e^x + e^{-x}}{2}$$

1. Derivative of Hyperbolic Sine ($\sinh x$)

We use the definition of $\sinh x$ and the rule $\frac{d}{dx}e^{ax} = ae^{ax}$:

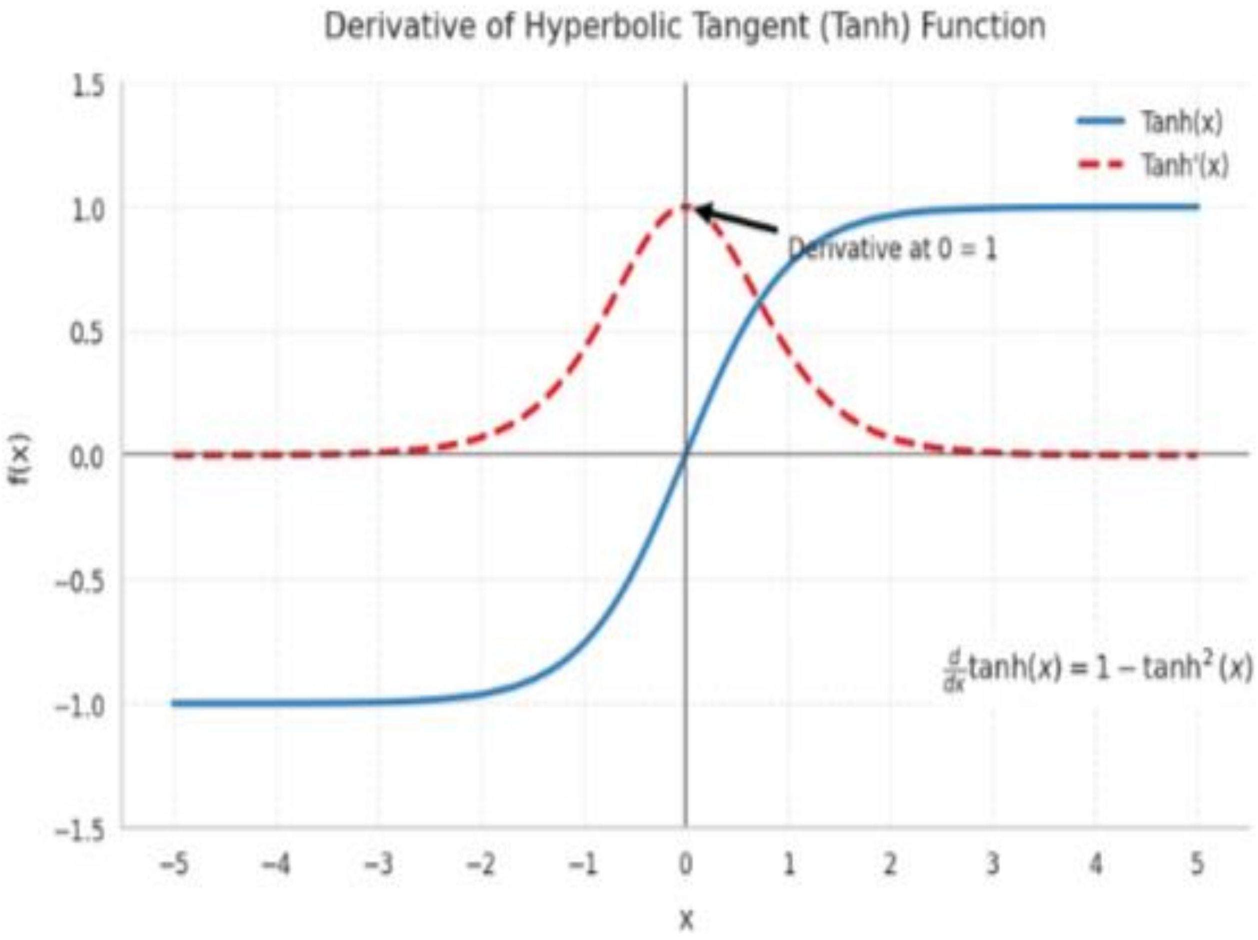
$$\begin{aligned}
\frac{d}{dx}(\sinh x) &= \frac{d}{dx}\left(\frac{e^x - e^{-x}}{2}\right) \\
&= \frac{1}{2}\left(\frac{d}{dx}e^x - \frac{d}{dx}e^{-x}\right) \\
&= \frac{1}{2}\left(e^x - (-e^{-x})\right) \\
&= \frac{e^x + e^{-x}}{2} \\
\frac{d}{dx}(\sinh x) &= \cosh x
\end{aligned}$$

Function, $f(x)$	Derivative, $\frac{d}{dx}f(x)$
$\sinh x$	$\cosh x$
$\cosh x$	$\sinh x$
$\tanh x$	$\operatorname{sech}^2 x$
$\coth x$	$-\operatorname{csch}^2 x$
$\operatorname{sech} x$	$-\operatorname{sech} x \tanh x$
$\operatorname{csch} x$	$-\operatorname{csch} x \coth x$

Derivative of Tanh

The **derivative** of the tanh function is also useful in the backpropagation step of training neural networks:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$



The standard derivative of the hyperbolic tangent is:

$$\frac{d}{dx}\tanh(x) = \operatorname{sech}^2(x)$$

The identity you gave, $\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$, is correct because of the hyperbolic identity:

$$\operatorname{sech}^2(x) = 1 - \tanh^2(x)$$

When to Use Tanh?

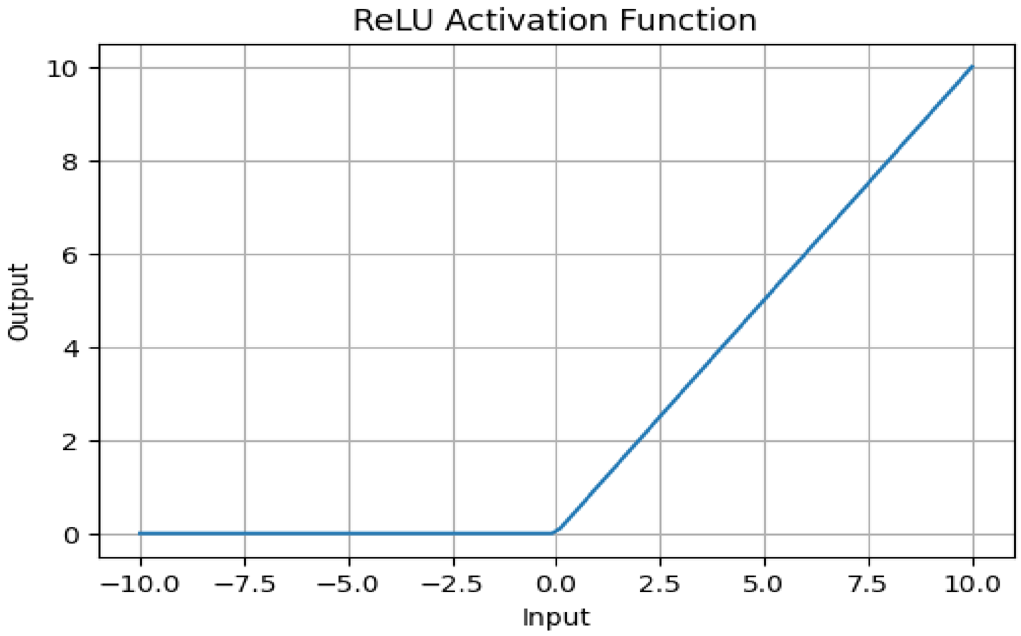
The tanh function is useful when:

- The **data is already normalized** or **centered around zero**.
- You are **building shallow neural networks (i.e., networks with fewer layers)**.
- You are **working with data where negative values are significant and should be retained**.

Rectified Linear Unit (ReLU)

Is defined by A(x)=max(0,x) this means that if the input x is positive, ReLU returns x, if the input is negative, it returns 0.

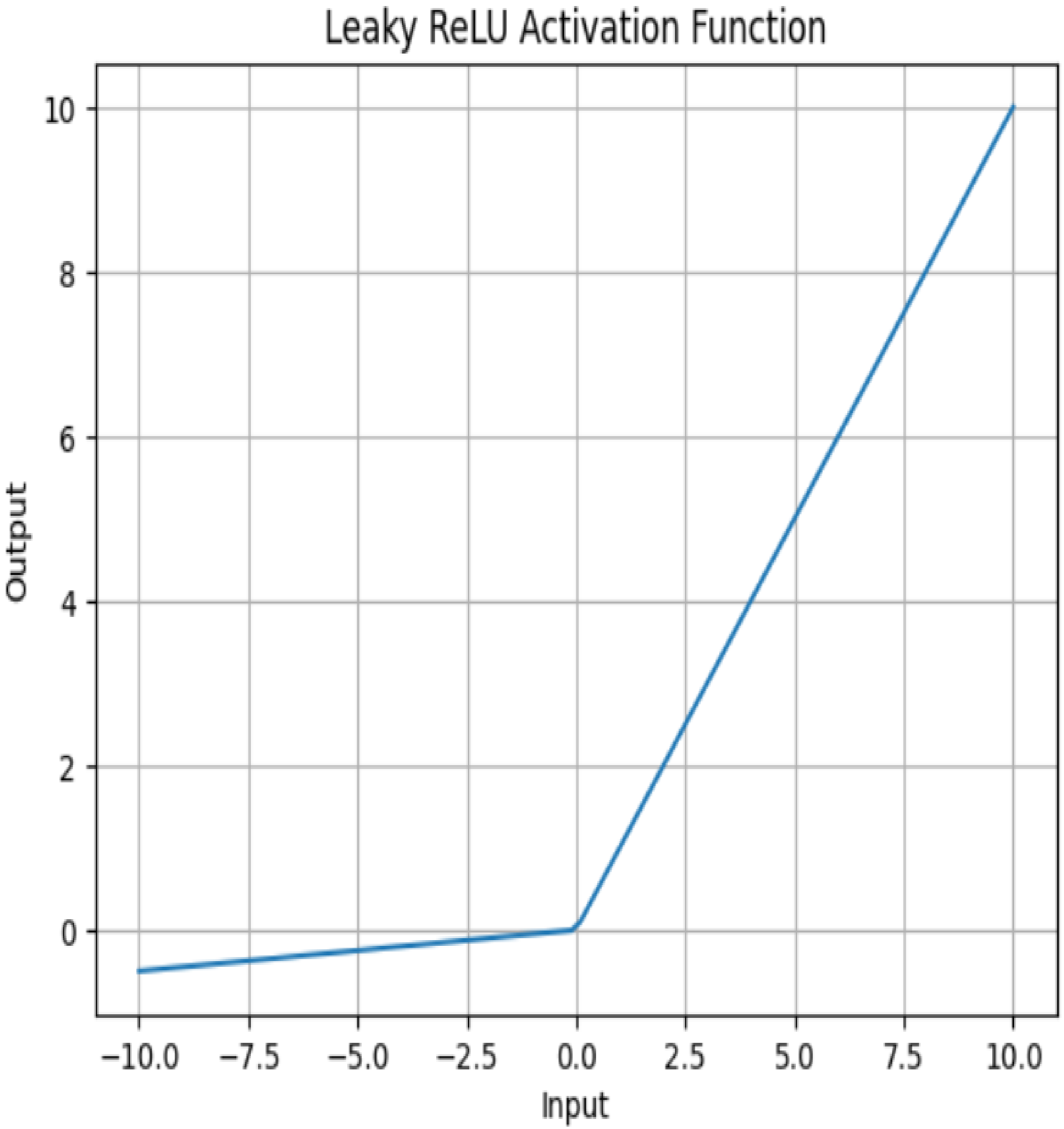
- **Value Range**: $[0, \infty)$, meaning the function only outputs non-negative values.
- **Nature**: It is a non-linear activation function, allowing neural networks to learn complex patterns and making backpropagation more efficient.
- **Advantage over other Activation:** ReLU is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.



d) Leaky ReLU

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$$

- Leaky ReLU is similar to ReLU but allows a small negative slope (α, e.g., 0.01) instead of zero.
- Solves the "dying ReLU" problem, where neurons get stuck with zero outputs.
- Range: $(-\infty, \infty)$.
- Preferred in some cases for better gradient flow.



OPTIMIZER

- A **loss function evaluates a model's effectiveness** by **computing the difference between expected and actual outputs**. Common loss functions include log loss, hinge loss and mean square loss.

- An **optimizer improves the model by adjusting its parameters (weights and biases) to minimize the loss function value**. Examples include RMSProp, ADAM and SGD (Stochastic Gradient Descent).

- The **optimizer's role is to find the best combination of weights and biases that leads to the most accurate predictions**.
- **wnew =wold −learning rate×gradient**

Learning Rate Value	Effect on Training	Issues
Too Large (High η)	Fast convergence initially.	Overshooting the minimum, causing the loss to diverge, oscillate widely, or get stuck in suboptimal valleys.
Too Small (Low η)	Slow but stable convergence.	Training takes too long, and the model might get stuck in a shallow local minimum because the steps aren't large enough to escape it.
Just Right	Optimally fast and stable approach to the minimum.	Finding this value often requires experimentation and tuning.

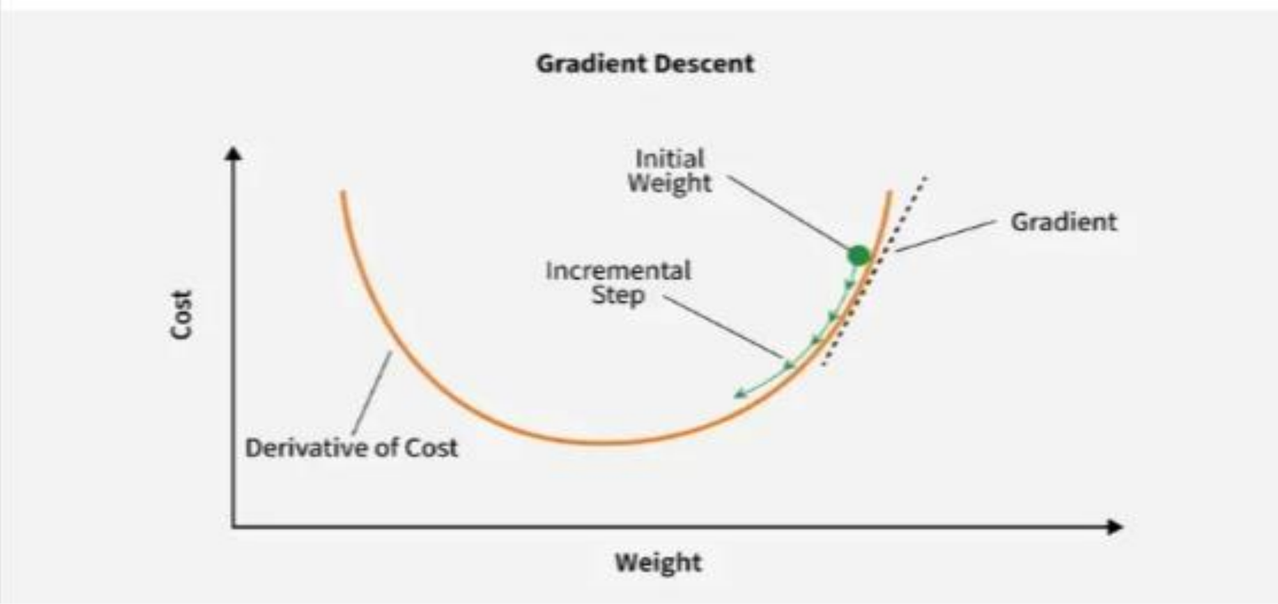
Techniques for Management

Instead of using a single fixed learning rate throughout training (a **static** learning rate), modern practice often involves using **learning rate schedulers** (also called **decay** or **adaptive** learning rates) to change η dynamically:

1. **Step Decay:** Reduces the learning rate by a factor (e.g., 0.1) every few epochs.
2. **Exponential Decay:** Reduces the learning rate exponentially over time.
3. **Cosine Annealing:** Gradually decreases the learning rate following a cosine curve, often resetting periodically.
4. **Adaptive Optimizers (e.g., Adam, Adagrad, RMSprop):** These optimizers **calculate individual, adaptive learning rates for each weight/parameter** based on the historical gradients. While they manage the "rate" internally, they still require a *base* learning rate hyperparameter to be set.

GRADIENT DESCENT

Works by iteratively adjusting the model parameters in the direction that minimizes the loss function.



Key Steps in Gradient Descent

- **Initialize parameters**: Randomly initialize the model parameters.
- **Compute the gradient**: Calculate the gradient (derivative) of the loss function with respect to the parameters.
- **Update parameters**: Adjust the parameters by moving in the opposite direction of the gradient, scaled by the learning rate.

Formula :

$$\theta_{(k+1)} = \theta_k - \alpha \nabla J(\theta_k)$$

where:

- $\theta_{(k+1)}$ is the updated parameter vector at the $(k+1)^{th}$ iteration.
- θ_k is the current parameter vector at the kth iteration.
- α is the learning rate, which is a positive scalar that determines the step size for each iteration.
- $\nabla J(\theta_k)$ is the gradient of the cost or loss function J with respect to the parameters θ_k

STOCHASTIC GRADIENT DESCENT (SGD)

- In traditional gradient descent, **the gradients are computed based on the entire dataset which can be computationally expensive** for large datasets.

- In Stochastic Gradient Descent, **the gradient is calculated for each training example (or a small subset of training examples) rather than the entire dataset.**

Stochastic Gradient Descent update rule is:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x_i, y_i)$$

Where:

- x_i and y_i represent the features and target of the i-th training example.
- The gradient $\nabla_\theta J(\theta; x_i, y_i)$ is now calculated for a single data point or a small batch.

1. AdaGrad

AdaGrad adapts the learning rate for each parameter based on the historical gradient information. The learning rate decreases over time, making AdaGrad effective for sparse features.

$$\theta_{(t+1)} = \theta_t - \frac{\alpha}{\sqrt{G_t + \varepsilon}} * \nabla J(\theta_t)$$

Where:

- G_t is the sum of squared gradients.
- ε is a small constant to avoid division by zero.

Advantages: Adapts the learning rate, improving training efficiency.

Disadvantages: Learning rate decays too quickly, causing slow convergence.

RMSProp improves upon AdaGrad by introducing a decay factor to prevent the learning rate from decreasing too rapidly.

$$E[g^2]_t = \gamma * E[g^2]_{(t-1)} + (1-\gamma) * (\nabla J(\theta_t))^2$$
$$\theta_{(t+1)} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \varepsilon}} * \nabla J(\theta_t)$$

Where:

- γ is the decay rate.
- $E[g^2]_t$ is the exponentially moving average of squared gradients.

Advantages: Prevents excessive decay of learning rates.

Disadvantages: Computationally expensive due to the additional parameter.

SGD	Simple, easy to implement	Slow convergence, requires tuning
Mini-Batch SGD	Faster than SGD	Computationally expensive, stuck in local minima
SGD with Momentum	Faster convergence, reduces noise	Requires careful tuning of β
AdaGrad	Adaptive learning rates	Decays too fast, slow convergence
RMSProp	Prevents fast decay of learning rates	Computationally expensive
Adam	Fast, combines momentum and RMSProp	Memory-intensive, computationally expensive