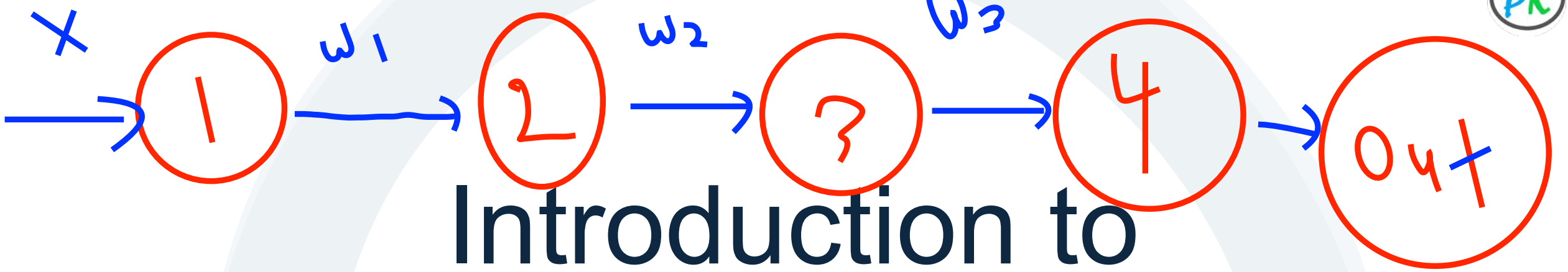
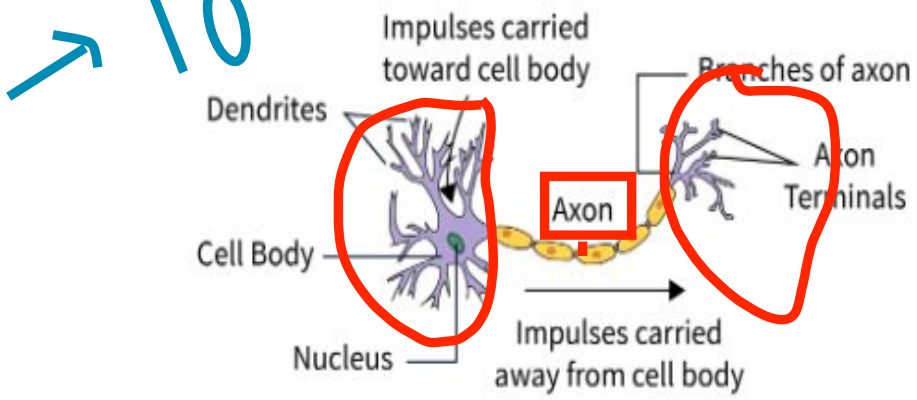


Introduction to Deep Learning

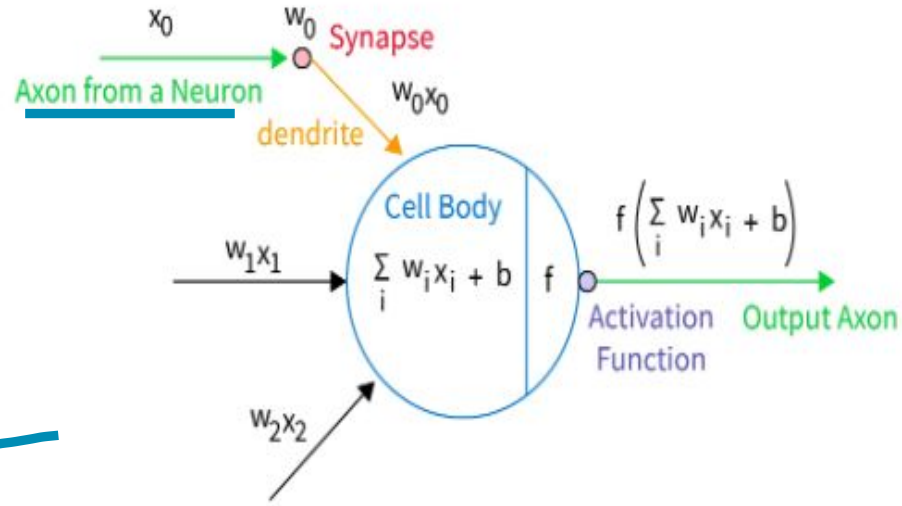
By PrudviKrishna

Deep Learning

Deep learning is a subset of machine learning that focuses on algorithms inspired by the structure and function of the neurons in brain, called **artificial neural networks**.



A cartoon drawing of a biological neuron (top) and its mathematical model (down)



Perceptron

A **Perceptron** is the simplest form of a neural network, introduced by Frank Rosenblatt in the 1950s. It consists of a single neuron with multiple inputs, each associated with a weight. The perceptron takes these weighted inputs, sums them up, applies an activation function (commonly a step function), and produces a binary output (e.g., 0 or 1). It was the foundation for more complex neural networks.

The perceptron model can be represented as:

$$Y = \text{activation}(w1x1+w2x2+\cdots+ wnxn + b)$$

Where:

- w1,w2,…,wn are the weights,
- x1,x2,…,xn are the inputs,
- b is the bias term,

- The **activation** function can be a step function or a sigmoid function.

Deep Learning

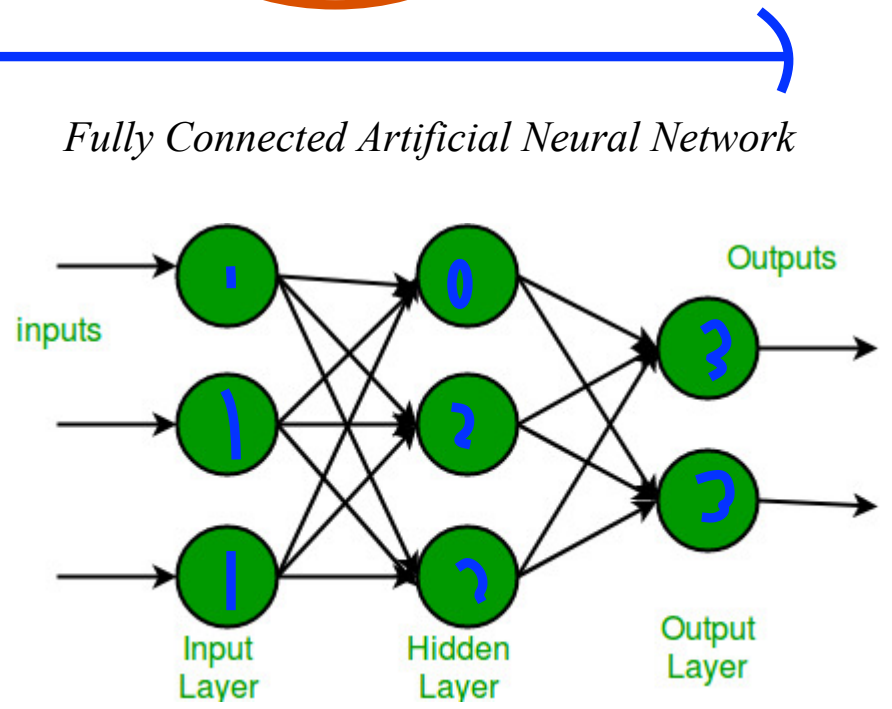
It involves using layers of neurons to learn from large amounts of data in a hierarchical manner. The "deep" in deep learning refers to the use of multiple layers in the network.

Neural Networks:
A **Neural Network** consists of layers of neurons. These layers can be categorized into:

- **Input Layer**: Takes in the input features.
- **Hidden Layers**: Layers between the input and output, where computations are performed.
- **Output Layer**: Produces the final result or prediction.

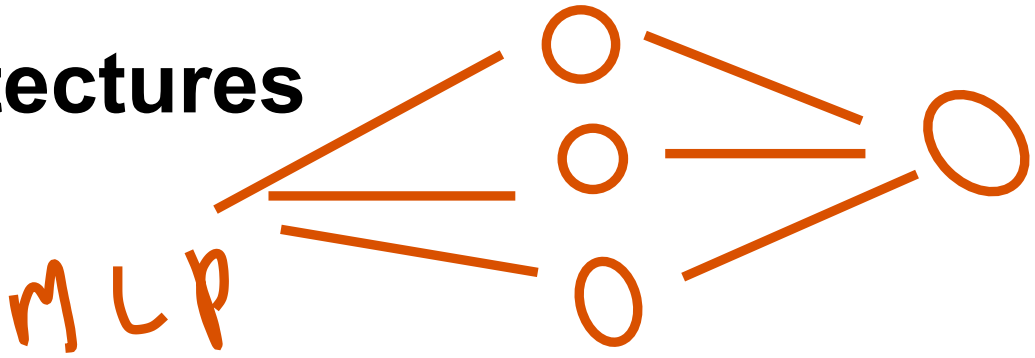
In **deep learning**, these networks are made up of many hidden layers, often referred to as **deep neural networks (DNNs)**.

Multi-Layer Perceptron (MLP)
An **MLP** is a fully connected feedforward neural network, which consists of one or more layers of perceptrons (neurons). Each perceptron is connected to every neuron in the next layer, forming fully connected neural network with multi layesrs called dense network. MLPs are used for classification and regression tasks.



Fully Connected Artificial Neural Network

DL Architectures



Deep Learning Architectures

- **Convolutional Neural Networks (CNNs)**: Used for image recognition tasks. CNNs utilize convolutional layers to detect patterns in images.
- **Recurrent Neural Networks (RNNs)**: Used for sequential data, such as time series or natural language processing. They have loops that allow information to persist.
- **Generative Adversarial Networks (GANs)**: Used for generating new data (e.g., images or text) by pitting two neural networks (generator and discriminator) against each other.
- **Transformers(LLMs) :** Use self-attention mechanisms to process sequential data efficiently, enabling them to model long-range dependencies in text and beyond. LLMs, such as GPT, BERT, and T5, are the foundation of modern NLP applications, from language modeling to machine translation and text generation.

Training Neural Networks

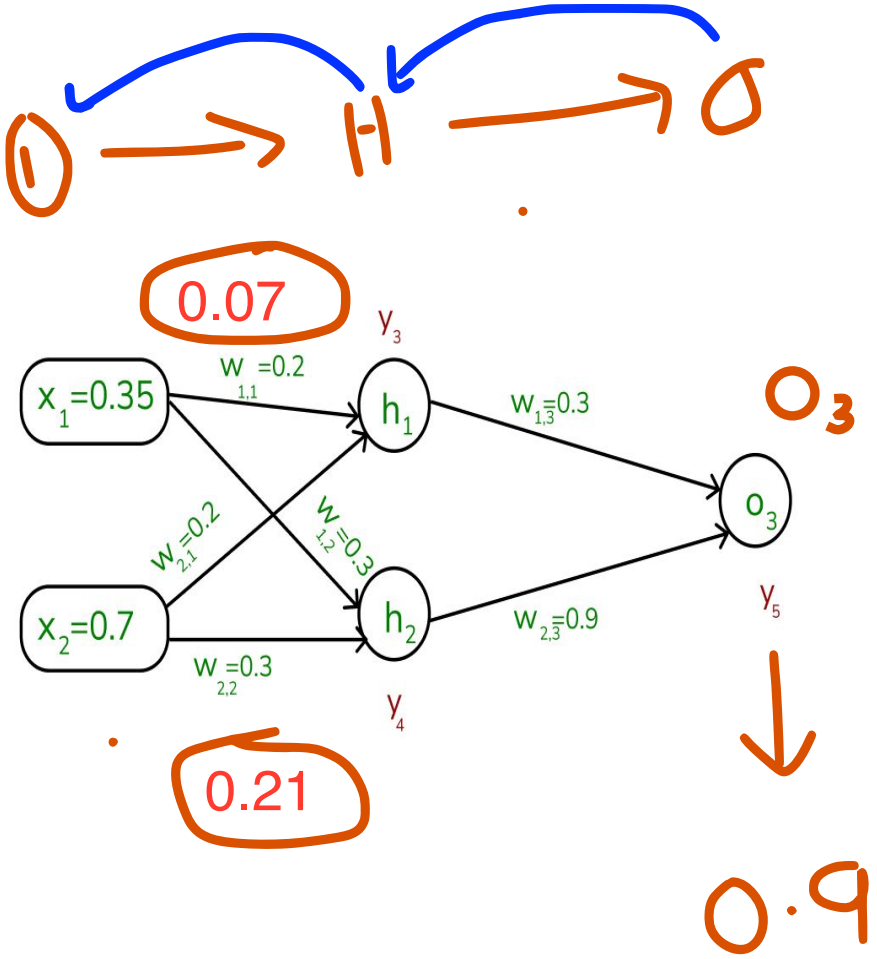
Training Neural Networks
Neural networks learn by adjusting the weights of the connections using a process called **training**, typically achieved through a method called backpropagation.

Forward Propagation: The input is passed through the network layer by layer, producing an output.

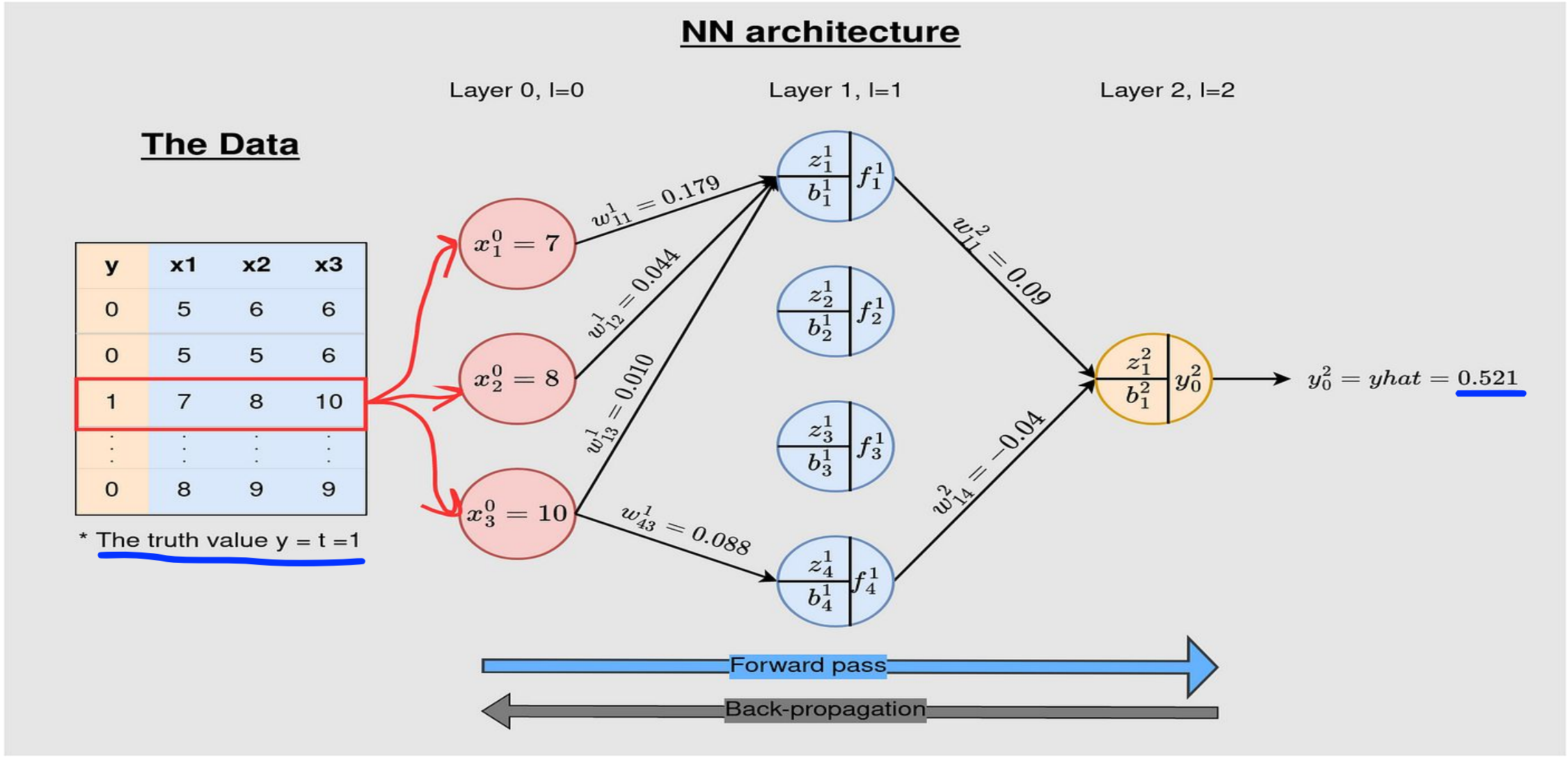
Loss Function: After forward propagation, a loss function computes the error between the predicted output and the actual label.

Backpropagation: The error is propagated backward through the network, adjusting the weights to minimize the error (via gradient descent).

Gradient Descent: A common optimization algorithm used to minimize the loss by updating weights in the direction of the negative gradient.



Training Neural Networks

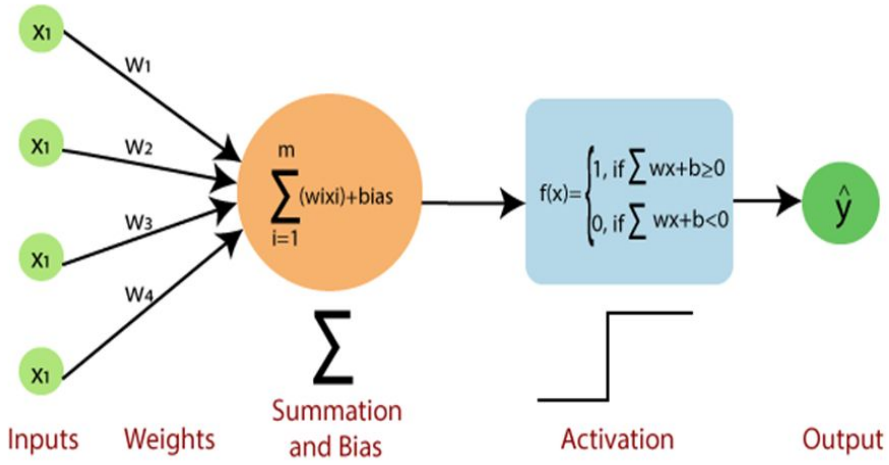


Forward pass

The **forward pass** is a fundamental step in the training and inference process of a neural network. It refers to the computation that occurs as data flows forward through the network, layer by layer, to generate predictions or outputs.

Key steps in a Forward Pass

- **Input Data:**
  The forward pass starts with input data (e.g., images, text, or numerical data) fed into the first layer of the network.
- **Computation at each Neuron:**
  Each Neuron processes the data it receives from the previous input by applying a linear transformation (e.g., matrix multiplication with weights and adding biases) and an **activation function** (e.g., ReLU, sigmoid, or tanh).
- **Propagation to Subsequent Layers:**
  The output from one layer becomes the input to the next.
- **Output Layer:**
  The final layer produces the network's prediction. In classification tasks, this might be a probability distribution (e.g., softmax); in regression tasks, it could be a numerical value.



Activation functions

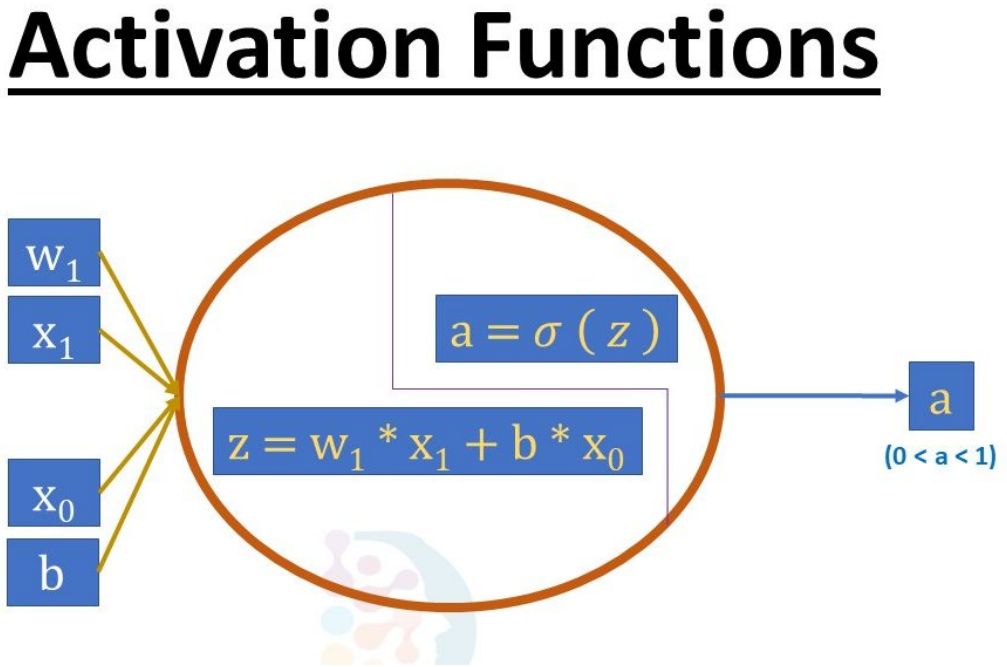
Activation functions play a vital role in Neural Networks. An activation function is a mathematical function that makes the Neural Network special. Without activation functions, a Neural Network wouldn't perform better than a Regression model. The activation function introduces non-linearity into the model.

In real-life data, we see non-linear data samples. Adding an activation function ensures that we consider the non-linearity in the data. Furthermore, adding the activation function turns the neurons on and off for different inputs, which further helps the Neural network learn about the input and its corresponding output.

Perceptron provides flexibility to update the **activation** function **for desired o/p** to any such as

There are two types of Activation functions:

1. **Linear Activation Functions**
2. **Non-linear Activation Functions**



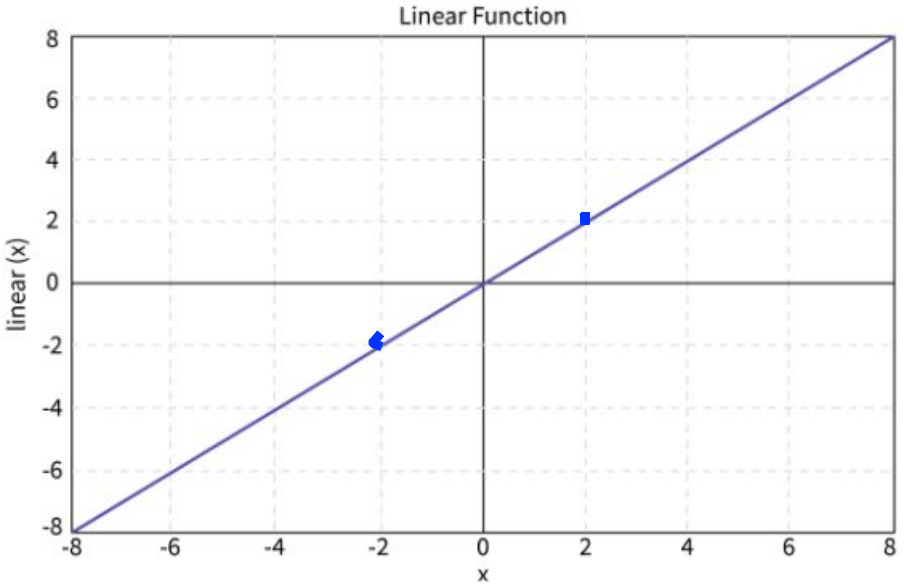
Linear Activation functions

Linear or Identity Activation

Usage: Regression tasks.
Formula:

$$f(x) = x$$

Non-linearity: The linear activation function does not introduce non-linearity. It simply passes the input directly to the output without any transformation. This makes it suitable for regression tasks where the model needs to predict continuous values.



Non Linear Activation functions

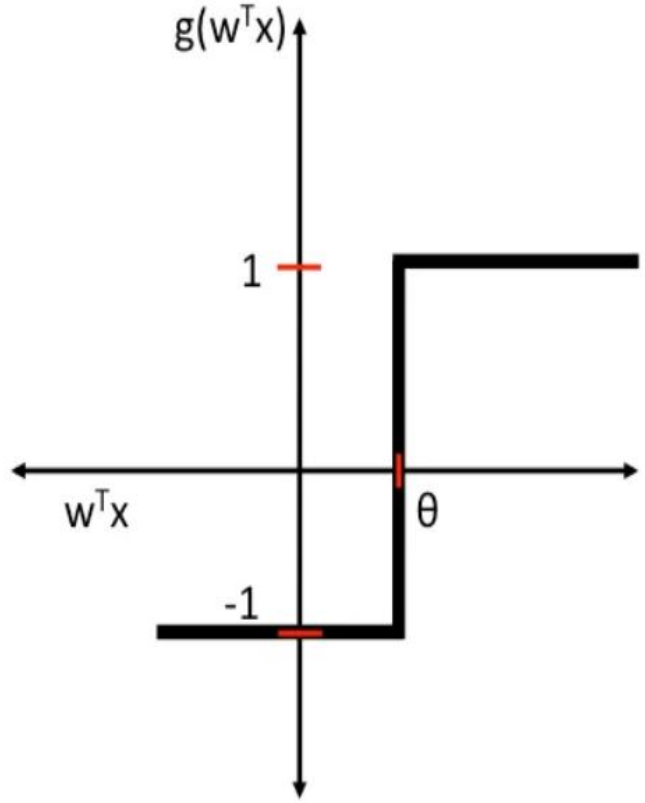
Step Function

Usage: Binary classification (output as 1 or -1).

Formula:

$$\text{Output} = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0 \end{cases}$$

The step function is a simple thresholding function. It outputs a constant value (1 or -1) based on whether the input is above or below a threshold (0). It introduces non-linearity by switching between two states, but it is rarely used in modern deep learning due to its inability to back propagate gradients.



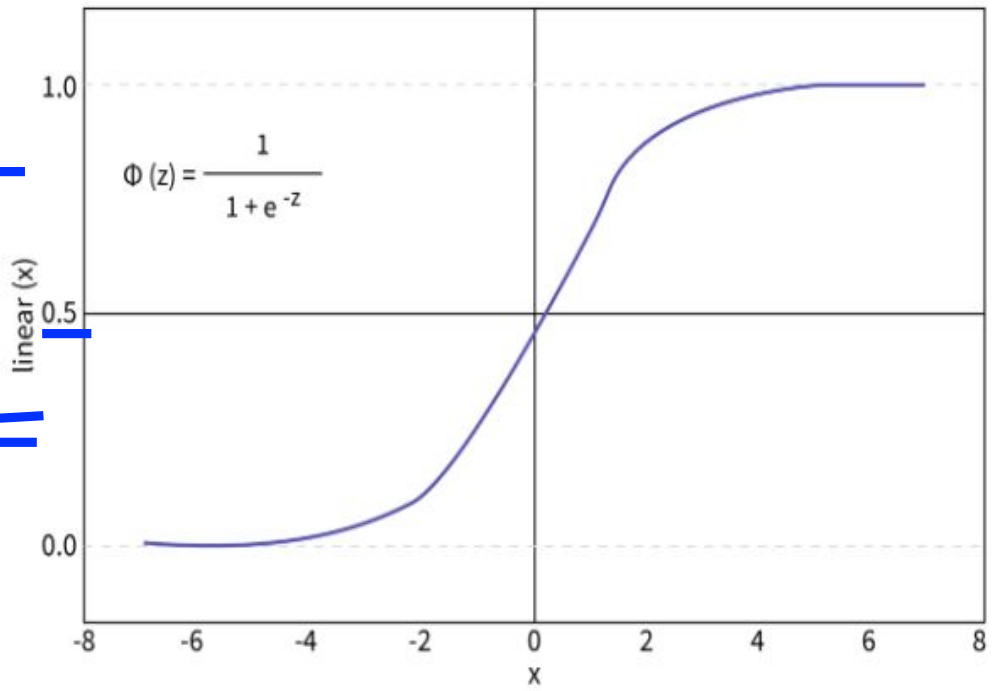
Non Linear Activation functions

Sigmoid (Logistic)

Usage: Binary classification, outputs probabilities.
Formula:

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

The sigmoid function squashes input values between **0 and 1**, making it useful for probabilistic outputs. The non-linearity comes from the exponential function, which causes the output to change gradually for small inputs and rapidly for large inputs. This allows the model to map input values into a probabilistic range.



Non Linear Activation functions

Softmax

Usage: Multi-class classification.

Formula:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

The softmax function takes a vector of values and squashes them into a probability distribution where all outputs sum to 1. It is non-linear due to the **exponential function**, which emphasizes the largest values in the output while diminishing the effect of smaller ones.

Non Linear Activation functions

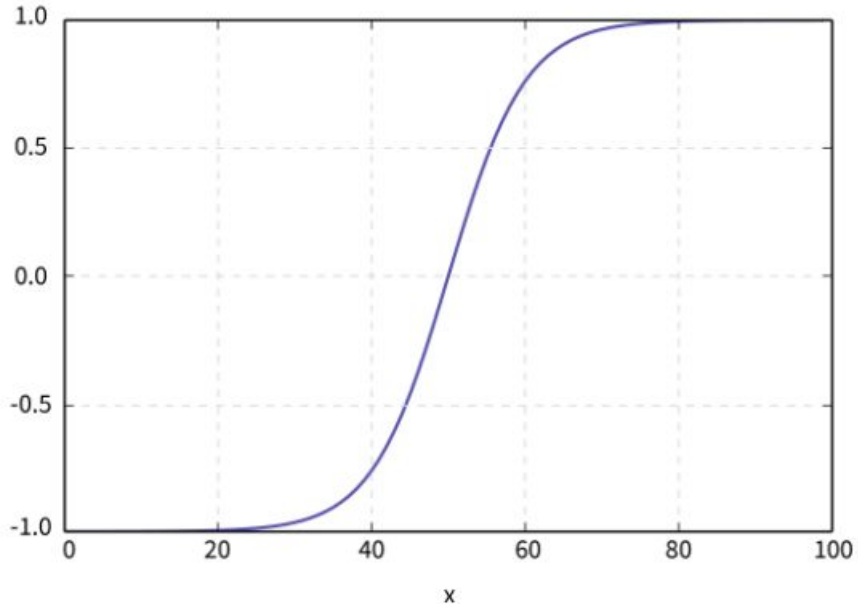
Tanh (Hyperbolic Tangent)

Usage: Hidden layers, especially in RNNs.

Formula:

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

The **tanh** function squashes input values between **-1 and 1**. It introduces non-linearity because of the exponential terms in its formula, making it similar to the sigmoid but with a wider output range. This non-linearity allows neural networks to model complex relationships in the data.
Example: If the input to the tanh function is a large positive number, the output will be close to **1**, and for a large negative number, the output will be close to **-1**. This enables the model to capture positive and negative aspects of the data, which is crucial for complex pattern recognition.



Non Linear Activation functions

ReLU (Rectified Linear Unit)

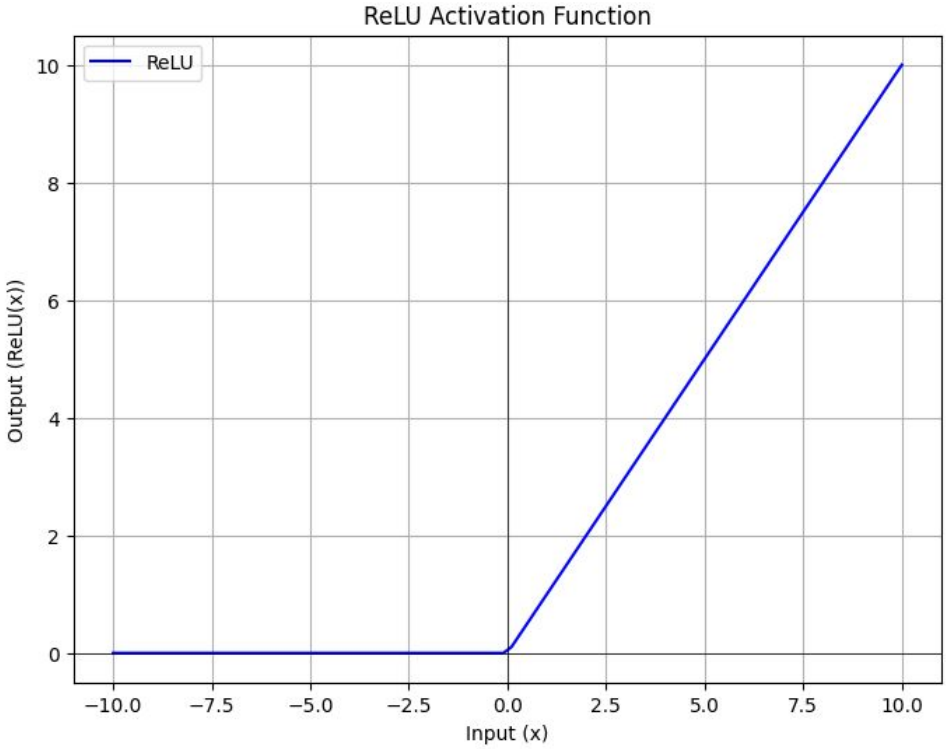
Usage: Hidden layers in most neural networks, especially deep networks.

Formula:

$$\mathrm{ReLU}(x) = \max(0, x)$$

Non-linearity: ReLU is a simple, piecewise linear function that introduces non-linearity by outputting **0** for negative inputs and the input itself for positive inputs. This introduces a "sparse" representation, meaning only the activated units (positive inputs) contribute to the output. This helps the model learn complex relationships by selectively passing forward certain inputs.

Example: If an input to a neuron is **-3**, the ReLU output will be **0**, and if the input is **5**, the output will be **5**. This non-linearity helps the network model relationships that aren't simply linear.

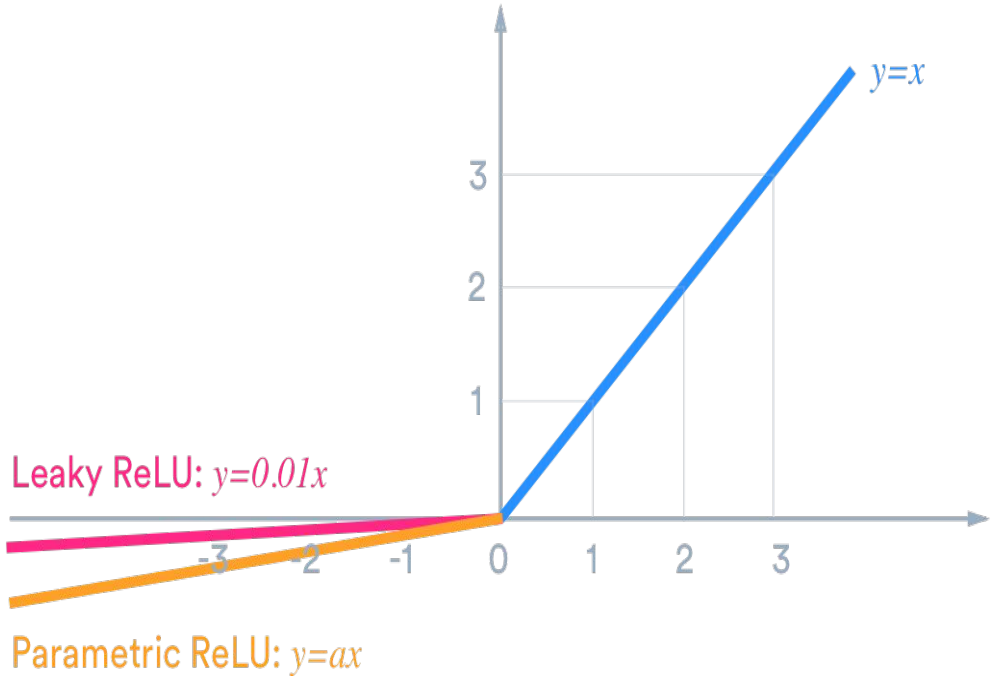


Non Linear Activation functions

Leaky ReLU
Usage: Hidden layers, to address the "dying ReLU" problem.
Formula:

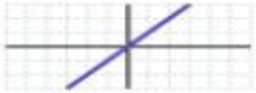
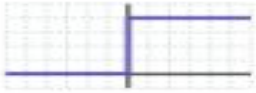
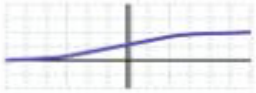
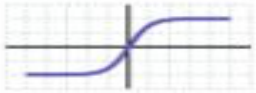
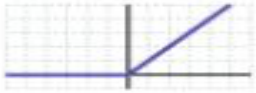
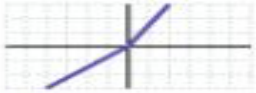
$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

Leaky ReLU is similar to ReLU but with a small slope for negative inputs. This prevents neurons from "dying" (i.e., always outputting zero) when the input is negative, which can happen in standard ReLU. The non-linearity comes from the modification of negative values, which ensures that gradients are still propagated through the network even for negative inputs.
Example: If the input to a neuron is **-3**, the Leaky ReLU output might be **-0.03** (for α = 0.01), and if the input is **5**, the output will be **5**. This ensures that negative inputs do not entirely stop the learning process, making the model more robust.



For leaky Relu α = 0.01 is static but for parametric Relu the α = 0.01 is dynamic parametric

Activation functions

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 \text{ for } x \neq 0 \\ ? \text{ for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)\,(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ X \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$
Parametric Rectified Linear Unit (PReLU)		$f(x) = \begin{cases} ax \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} a \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$

Activation functions – differentiable?

Why Activation fucntion must be Derivative/Differentiable?

The Activation function must be differentiable and monotonic and should converge quickly.
During Backpropagation, the derivative is calculated to train the model. So the activation function that is chosen should be differentiable.

Differentiate:
Change in the y-axis w.r.t. change in the x-axis. It is also known as slope.

Monotonic:
A function that is either entirely non-increasing or non-decreasing.

Note: Generally ReLU, Tanh, Leaky ReLU are used in hidden layers, other activation functions used based on task specific like Sigmoid, Step (classification), Softmax (multi classification) and Linear (Regression)
Most deep learning networks, ReLU is the default choice for hidden layers.

Loss functions

- The choice of an activation function directly affects the scale and range of the output.
- The loss function needs to complement the activation function to properly evaluate the predictions

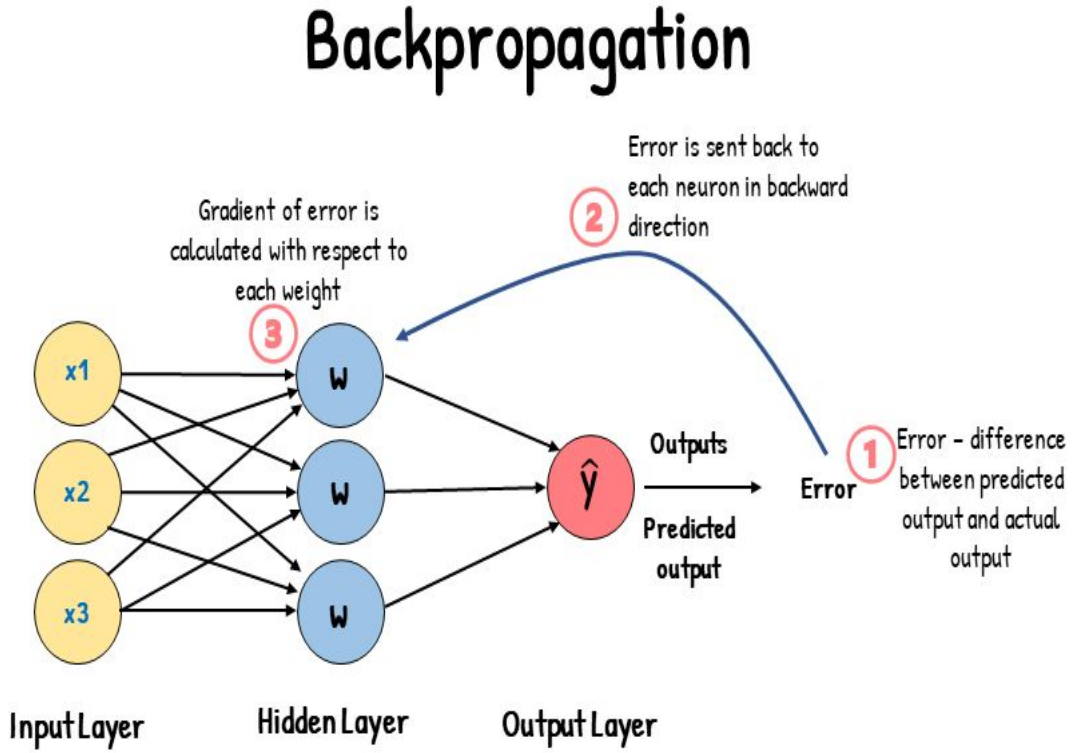
- During the forward pass, the network processes the input data and computes the output for each layer.
- The output at the final layer is **only** compared to the **true target** using the **loss function** (e.g., MSE, Cross-Entropy).
- The **loss** (error) is calculated at the **output layer** and the weights at each layers get's updated accordingly.

Activation function	Loss function	Output
Step function	Hinge loss	Binary o/p (1,-1)
Sigmoid/logistic	Binary cross entropy or log loss	Binary probabilities (0.8,0.2)
Softmax	Categorical cross entropy	Multi class classification
Tanh	Mean Squared Error (MSE)	Regression ([−1,1])
ReLU	Mean Squared Error (MSE), MAE	Regression ([0,∞))
Linear (None)	Mean Squared Error (MSE), Huber Loss	Regression (unbounded outputs)
Leaky ReLU	MSE, MAE	Regression (robust to outliers)

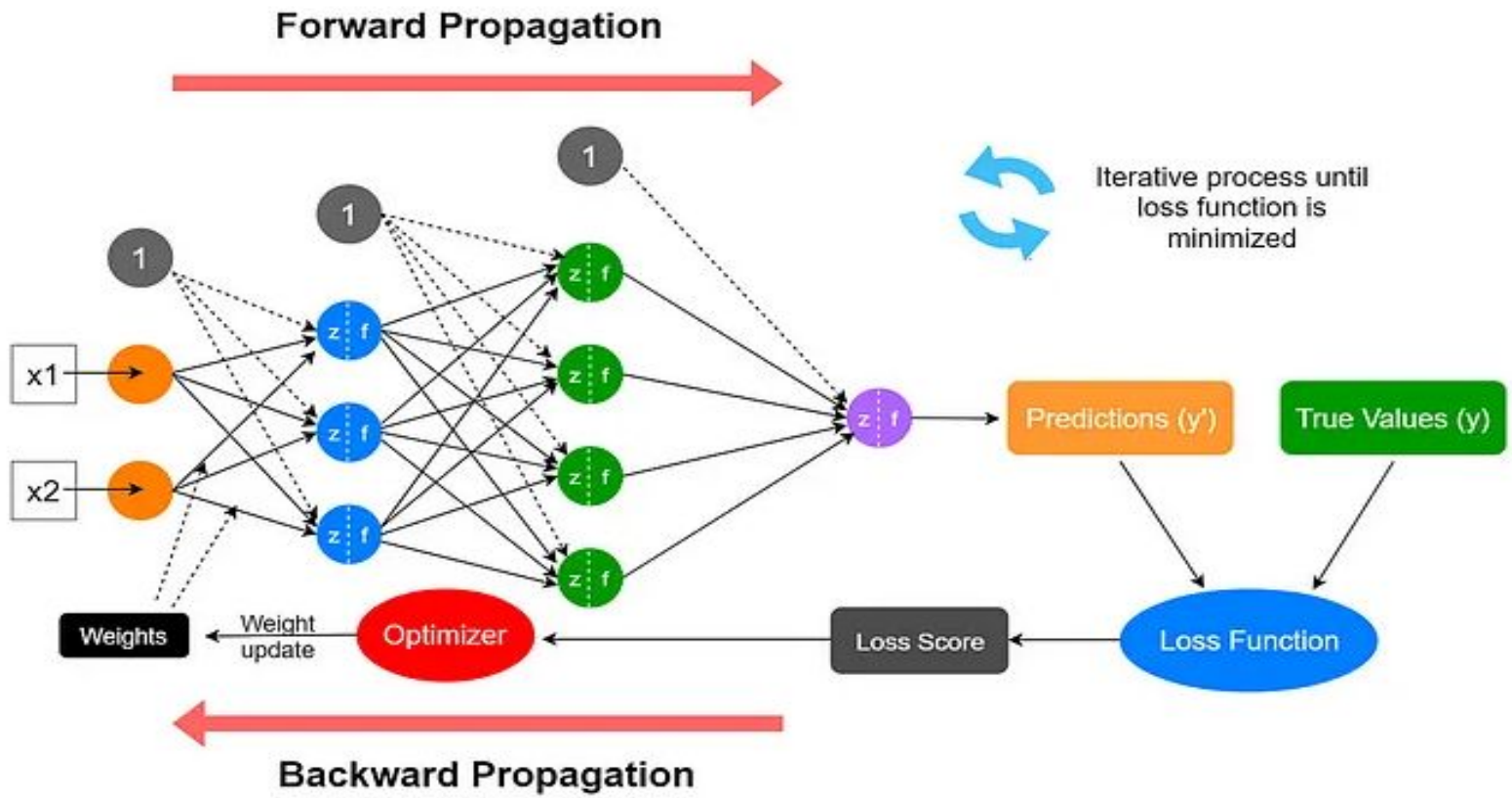
Back propagation

The loss (error) is calculated at the output layer and the weights at each layers get's updated accordingly. At output layer gradient of loss fucntion and at each neuron gradient of activation function are calculated to update the the weights the model.

- **At the output layer**, the loss (error) is calculated by comparing the predicted output to the actual target.
- **The gradient of the loss function** with respect to the output is computed to determine how much the loss changes with respect to the output.
- **The gradient of the activation function** (used in the output layer) is also computed to understand how the activation function's output influences the loss.
- **Backpropagation** combines these gradients to update the weights in the network, starting from the output layer and propagating back through the network.



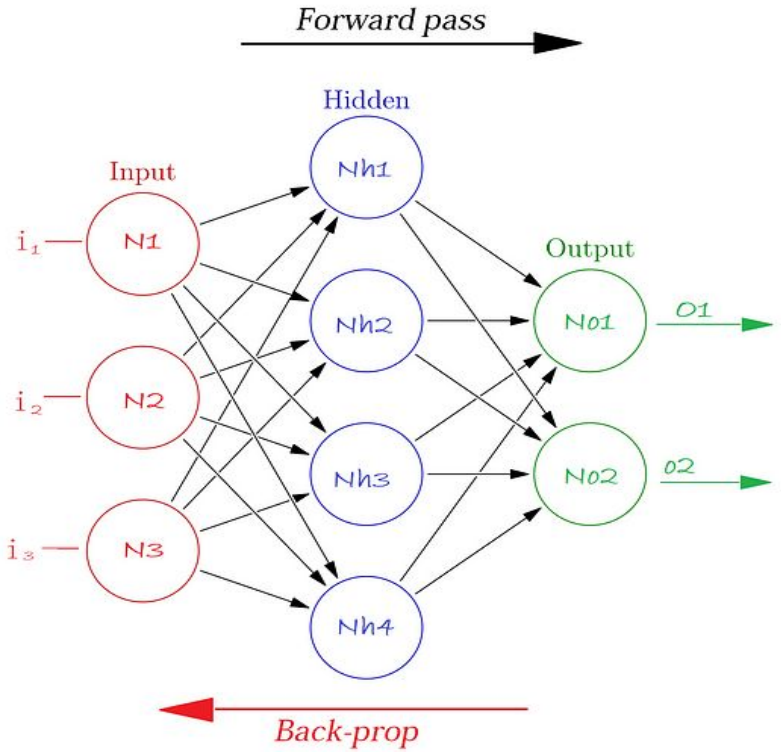
Back propagation



Back propagation - Handson

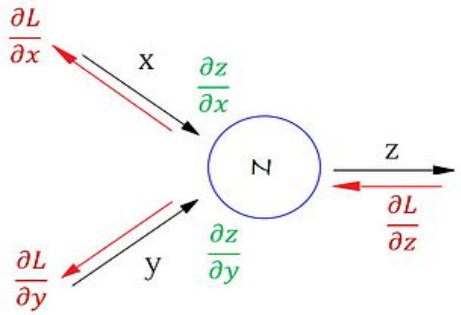
$$W_{new} = W_{old} - \alpha \frac{dJ}{dW}$$

$$b_{new} = b_{old} - \alpha \frac{dJ}{db}$$

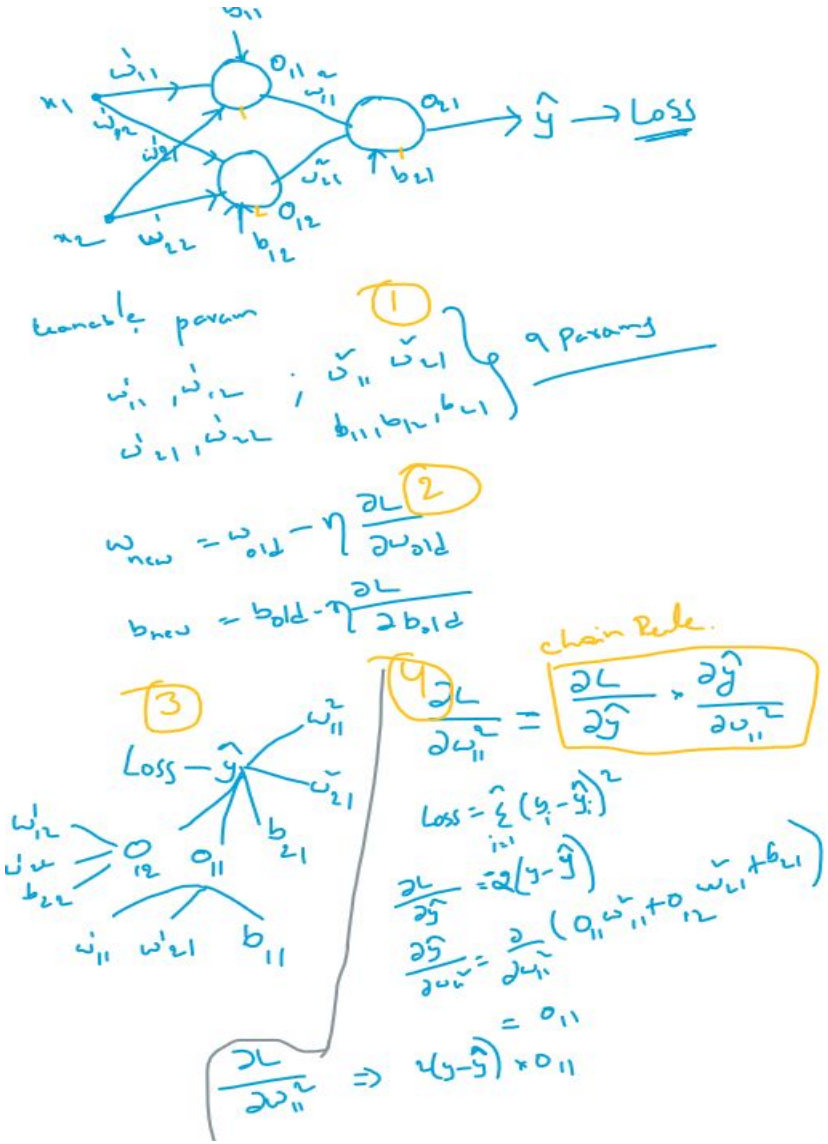


Chain rule

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial x}$$



$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial y}$$



Epoch and Gradient Descent

- **Epochs**
  - An **epoch** is one complete cycle through the entire training dataset. Training a neural network involves multiple epochs, as learning improves with more passes through the data.

- **Gradient Descent**

  Core Concept: An optimization algorithm that iteratively adjusts the model's parameters (weights and biases) to minimize the loss function.

  How it Works:
  - Calculates the gradient (slope) of the loss function with respect to each parameter.
  - Updates the parameters in the direction opposite to the gradient, taking small steps.

- **Variants of gradient descent : Batch, Stochastic, and Mini-Batch Gradient Descent**

Variants of Gradient Descent

- **Batch Gradient Descent / Gradient descent**
  - **Definition:** Uses the entire training dataset to compute the gradient.
  - **Advantages:** More stable convergence.
  - **Disadvantages:** Can be very slow and computationally expensive for large datasets.

- **Stochastic Gradient Descent (SGD)**
  - **Definition:** Uses a single training example to compute the gradient at each iteration.
  - **Advantages:** Faster and more efficient for large datasets.
  - **Disadvantages:** Can produce noisy gradients, leading to fluctuating convergence.

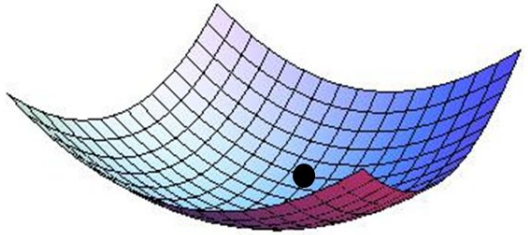
- **Mini-Batch Gradient Descent**
  - **Definition:** Uses a small, random subset of the training data to compute the gradient.
  - **Advantages:** Balances the trade-offs between batch and stochastic gradient descent.
  - **Disadvantages:** Still requires careful selection of batch size.

Variants of Gradient Descent

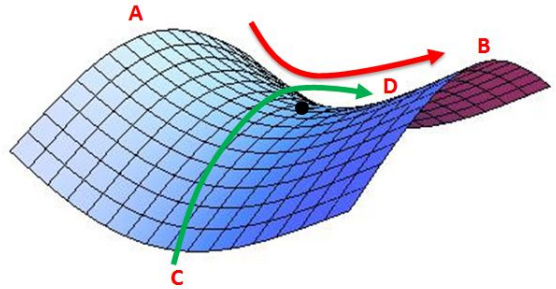
- **Batch Gradient Descent / Gradient descent**
  - **Definition:** Uses the entire training dataset to compute the gradient.
  - **Advantages:** More stable convergence.
  - **Disadvantages:** Can be very slow and computationally expensive for large datasets.

- **Stochastic Gradient Descent (SGD)**
  - **Definition:** Uses a single training example to compute the gradient at each iteration.
  - **Advantages:** Faster and more efficient for large datasets.
  - **Disadvantages:** Can produce noisy gradients, leading to fluctuating convergence.

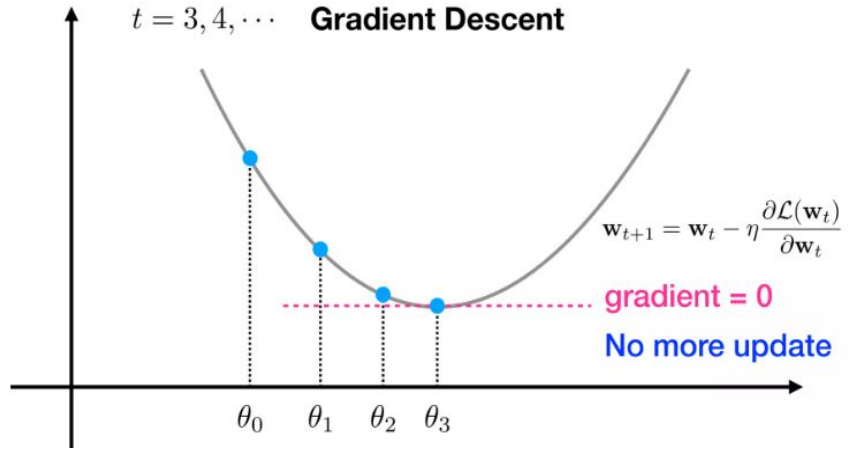
- **Mini-Batch Gradient Descent**
  - **Definition:** Uses a small, random subset of the training data to compute the gradient.
  - **Advantages:** Balances the trade-offs between batch and stochastic gradient descent.
  - **Disadvantages:** Still requires careful selection of batch size.



Minima- Black dot placed on the PES shows a minimum energy point. Note how a PES resembles a well around the minimum point.



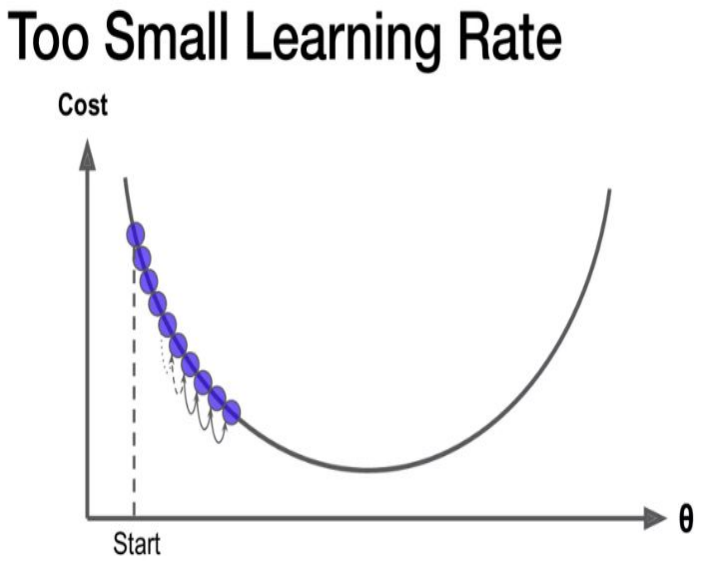
Saddle Point- Black dot placed on the PES shows a minima along path A-B and a maxima along path C-D. It represents a transition state along path C-D which, in this case, is the reaction coordinate.

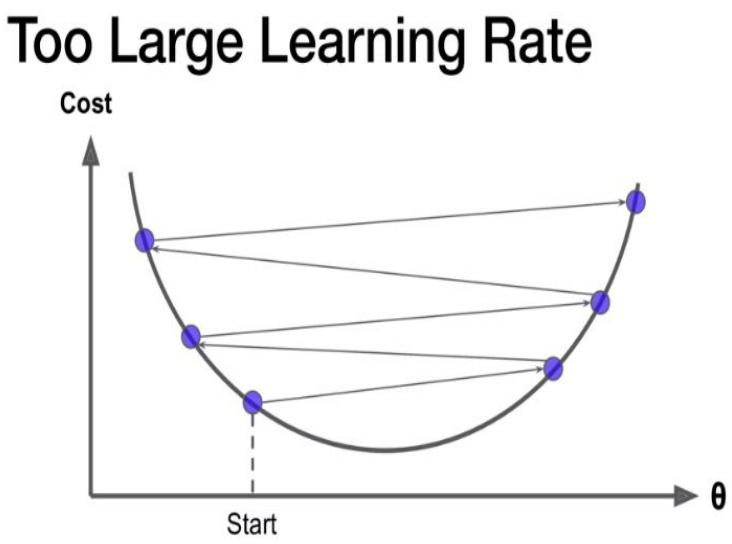


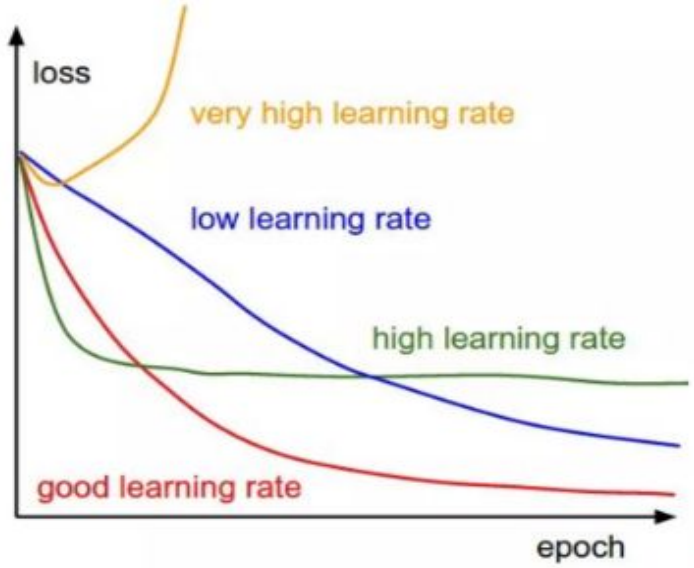
How to pick the Learning rate?

- Too big = diverge, too small = slow convergence
- No "one learning rate to rule them all"
- Start from a high value and keep cutting by half if model diverges

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{\partial \mathcal{L}(\mathbf{w}_t)}{\partial \mathbf{w}_t}$$





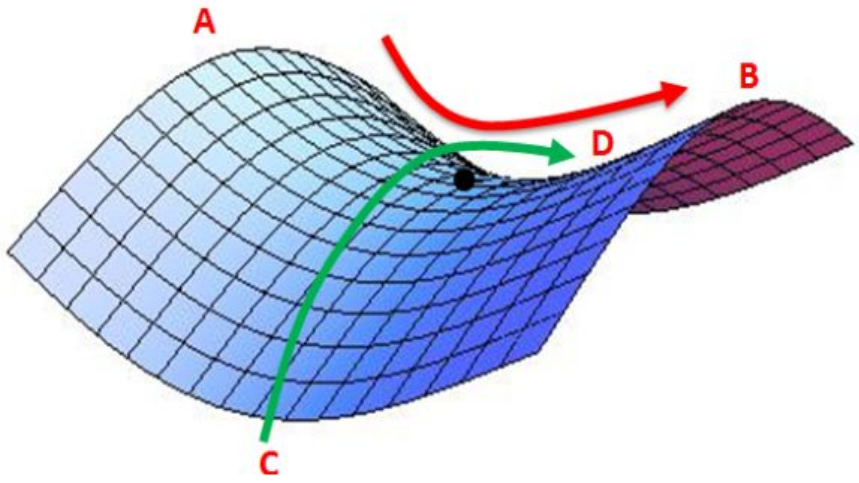


Issue with gradient descent

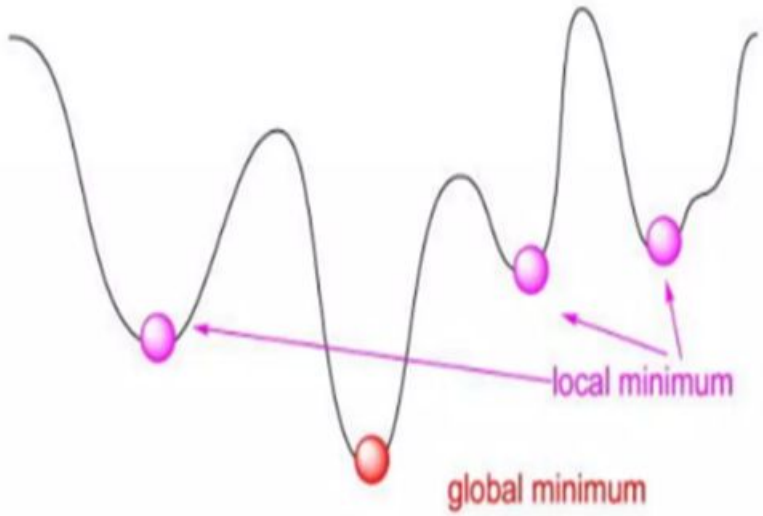
- **Local Minima** : Gradient descent can converge to a local minimum, especially in non-convex optimization problems like neural networks.
- **Slow convergence**: Flat Regions (Plateaus): In areas where the gradient is very small (e.g., near saddle points or flat regions of the loss function), gradient descent progresses very slowly.
- **Learning Rate Issues:** Too Small - Convergence is very slow. Too Large - The algorithm may overshoot the minimum, diverge, or oscillate.

Mitigation:

- Use optimizers with adaptive learning rates like AdaGrad, RMSProp, or Adam etc.



Saddle Point- Black dot placed on the PES shows a minima along path A-B and a maxima along path C-D. It represents a transition state along path C-D which, in this case, is the reaction coordinate.



Optimizers - SGD with momentum

- Momentum introduces a velocity term to smooth and accelerate gradient updates.
- If the gradients are pointing in the same direction (e.g., far from the minimum), the velocity (momentum term) builds up.
- This results in larger updates (effectively acting as a larger learning rate) and faster movement in that direction.
- As the model approaches the minimum, the gradient magnitude decreases, reducing the contribution to the velocity term.

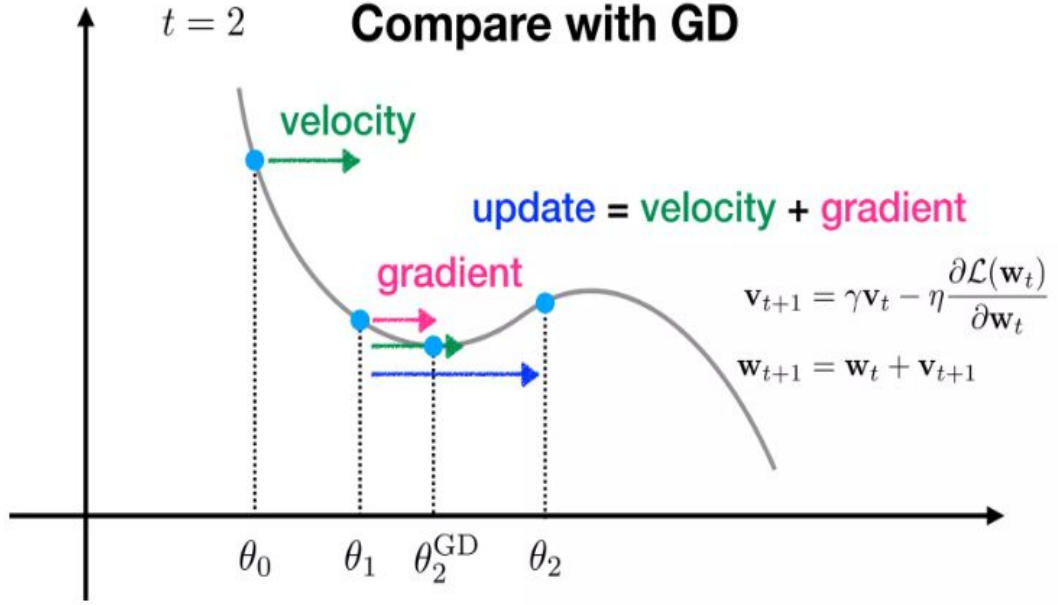
SGD

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{\partial \mathcal{L}(\mathbf{w}_t)}{\partial \mathbf{w}_t}$$

SGD + Momentum

$$\mathbf{v}_{t+1} = \gamma \mathbf{v}_t - \eta \frac{\partial \mathcal{L}(\mathbf{w}_t)}{\partial \mathbf{w}_t}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{v}_{t+1}$$



SGD with momentum

How initially the large weight updates are taken and smaller steps when closer to minimum loss

a. Velocity Accumulates Gradients

- In the first few iterations, V_t starts small because $V_0 = 0$ (initial velocity is zero).
- As gradients consistently point in the same direction, $\eta \nabla L(w_t)$ keeps adding to V_t, and the momentum term γV_{t-1} retains a fraction of the past velocity.

For example:

$$V_1 = \eta \nabla L(w_0)$$

$$V_2 = \gamma V_1 + \eta \nabla L(w_1)$$

$$V_3 = \gamma V_2 + \eta \nabla L(w_2)$$

b. Accumulation Over Iterations

Substituting for V_2 and V_3:

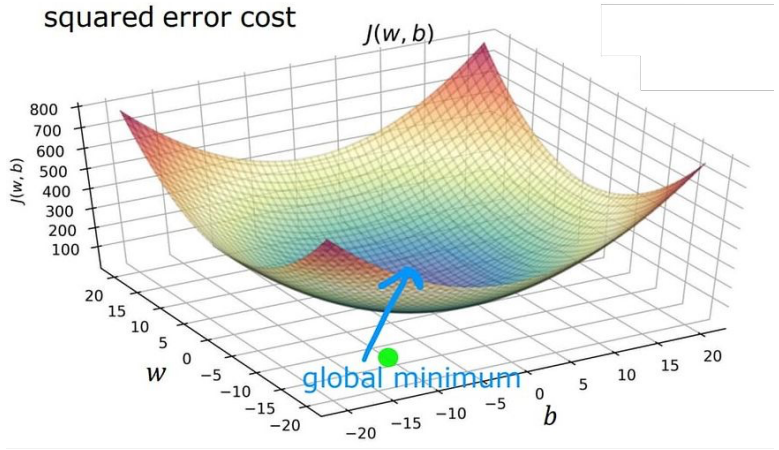
$$V_2 = \gamma(\eta \nabla L(w_0)) + \eta \nabla L(w_1)$$

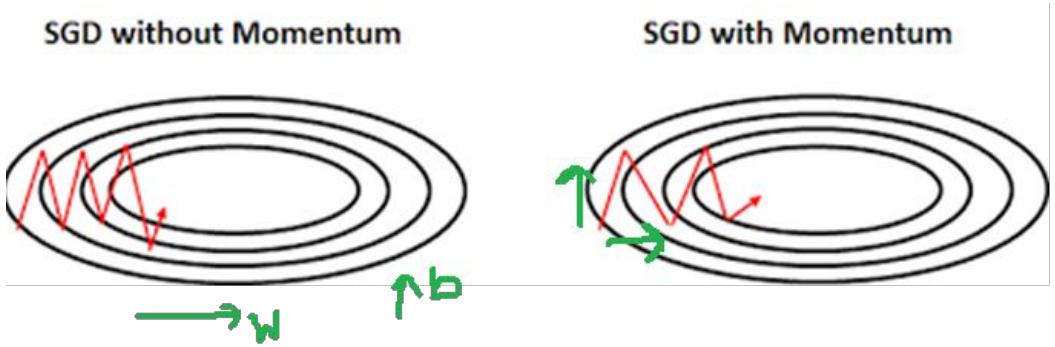
$$V_3 = \gamma^2(\eta \nabla L(w_0)) + \gamma(\eta \nabla L(w_1)) + \eta \nabla L(w_2)$$

Thus, the velocity term becomes a **weighted sum of past gradients**, with higher weights given to more recent gradients due to γ^k decay.

c. Faster Updates

- If the gradients are consistently pointing in the same direction (e.g., all $\nabla L(w_t) > 0$), the **magnitude of V_t** grows larger with each iteration.
- This results in **larger steps**, accelerating movement toward the minimum.





Root mean square propagation(RMS Prop)

- Adjusts the learning rate for each parameter dynamically based on the magnitude of the gradients.

i.e., $$\frac{\eta}{\sqrt{v_t}+\epsilon}\nabla L(w_t)$$

- v_t scales the learning rate inversely with the gradient magnitude.
- Large gradients → Smaller learning rate.
- Small gradients → Larger learning rate.

Tuning Parameters:

RMSprop:

H(eta)=0.001, β(beta) = 0.9

- **Formula:**
  - Compute the moving average of squared gradients:
    $$v_t = \beta v_{t-1} + (1-\beta)(\nabla L(w_t))^2$$
  - Update rule:
    $$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t}+\epsilon}\nabla L(w_t)$$
  - η: Learning rate.
  - β: Decay rate (typically 0.9).
  - ϵ: Small constant to avoid division by zero.

$(\nabla L(w_t))^2$: Squared gradient at time step t.

Root mean square propagation(RMS Prop)

- In the expansion of vt, the contribution from older epochs diminishes because β is raised to higher powers. For example, if β=0.9, then β2=0.81, β3=0.729, and so on.
- This exponential decay ensures that older gradients have progressively less influence on the current value of vt.
- This ensures that vt focuses more on recent updates, making RMSprop adaptive to recent trends in the gradient.

Tuning Parameters:
RMSprop:
η=0.001,
β=0.9

Assume v0=0(commonly used as initialization).

For each time step, expand the terms:

1. $v_1 = (1-\beta)(\nabla L(w_1))^2$
2. $v_2 = \beta v_1 + (1-\beta)(\nabla L(w_2))^2$

   Substituting v_1:

   $$v_2 = \beta[(1-\beta)(\nabla L(w_1))^2] + (1-\beta)(\nabla L(w_2))^2$$
3. $v_3 = \beta v_2 + (1-\beta)(\nabla L(w_3))^2$

   Substituting v_2:

   $$v_3 = \beta^2[(1-\beta)(\nabla L(w_1))^2] + \beta[(1-\beta)(\nabla L(w_2))^2] + (1-\beta)(\nabla L(w_3))^2$$
4. $v_4 = \beta v_3 + (1-\beta)(\nabla L(w_4))^2$

   Substituting v_3:

   $$v_4 = \beta^3[(1-\beta)(\nabla L(w_1))^2] + \beta^2[(1-\beta)(\nabla L(w_2))^2] + \beta[(1-\beta)(\nabla L(w_3))^2] + (1-\beta)(\nabla L(w_4))^2$$

Adaptive momentum(Adam)

- Combines the strengths of momentum and RMSprop.
- Combines momentum (mt) for direction and scaling (vt) for learning rate adjustment.

Why is Bias Correction Needed?
Initialization Bias:

At the beginning (t=1,2,…), the moving averages mt and vt are computed using:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

- Since m0=0 and v0=0, the values of mt and vt are initially smaller than their true expectations. This leads to underestimation of both the first and second moments.
- To counteract this underestimation, bias-corrected versions of mt and vt are used.

Tuning Parameters:

- Adam:
  - $\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}.$

Formula:

- First moment (momentum):

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla L(w_t)$$

- Second moment (variance):

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla L(w_t))^2$$

- Bias correction:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

- Update rule:

$$w_{t+1} = w_t - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Optimizers - Summary

Optimizer	Algorithm	Key Features	Hyperparameters	Use Case
SGD (Stochastic Gradient Descent)	Updates parameters based on the gradient of the loss function using a fixed learning rate.	- Basic gradient descent - Uses only the current gradient for updates	learning_rate, momentum, nesterov	Simple models, and smaller datasets. Suitable for well-behaved loss functions.
Momentum	Same as SGD but with momentum term that accelerates gradients in relevant directions.	- Smooths the updates by considering previous gradients. - Helps with oscillations	learning_rate, momentum	Optimizing for deeper networks, where oscillations can be a problem.
RMSprop	Divides the learning rate by the moving average of the squared gradients.	- Adaptive learning rate for each parameter. - Works well with RNNs	learning_rate, rho, momentum, epsilon	Effective for recurrent neural networks (RNNs) and when gradients are sparse.
Adam (Adaptive Moment Estimation)	Combines momentum and RMSprop by maintaining moving averages of both first and second moments of the gradients.	- Adaptive learning rate. - Handles sparse gradients better.	learning_rate, beta_1, beta_2, epsilon	Works well for most models and is the default choice for most deep learning tasks.
AdaGrad	Adjusts the learning rate for each parameter based on the past squared gradients.	- Great for sparse data. - Learning rate decreases rapidly, which can hurt in long training.	learning_rate, epsilon	Best for sparse data (e.g., text or collaborative filtering tasks).
AdaDelta	An extension of AdaGrad that adapts the learning rate based on the moving average of squared gradients.	- Prevents the learning rate from decaying too quickly. - Reduces the need to set an initial learning rate.	learning_rate, rho, epsilon	Works well when you want to avoid the rapid decay problem seen in AdaGrad.
Nesterov Accelerated Gradient (NAG)	Momentum with a "look-ahead" feature where the update uses the gradient at the future position.	- Can achieve faster convergence. - Helps with oscillations like regular momentum.	learning_rate, momentum, nesterov=True	Helps accelerate convergence and smooths out gradient steps, especially in deep networks.

Optimizers - Practice

```
batch_size = len(x_train)  # Batch size equal to the size of the dataset

# Using SGD optimizer
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
              loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10, batch_size=batch_size)
```

The difference comes from the **batch_size** parameter:

- For **Batch GD**, set batch_size to the **entire dataset size**.
- For **SGD**, set batch_size = 1.
- For **Mini-Batch GD**, set batch_size to any value greater than 1 and smaller than the total dataset size.

SGD with momentum when momentum parameter is passed

```
import tensorflow as tf

# Using SGD with momentum
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
              loss='sparse_categorical_crossentropy',   metrics=['accuracy'])
```

Optimizers - Practice

```
import tensorflow as tf

# Using SGD with momentum
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
              loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Fit the model
model.fit(x_train, y_train, epochs=10, batch_size=32)
```

```
# Using RMSprop optimizer directly as a string (default settings)
model.compile(optimizer='RMSprop',  # Default RMSprop (momentum=0.0)
              loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001),
              loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9,
beta_2=0.999), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
# Using shorthand for Adam optimizer
model.compile(optimizer='Adam',  # Equivalent to tf.keras.optimizers.Adam()
              loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Vanishing or exploding Gradients

The **vanishing gradient problem** is an issue that arises during the training of deep neural networks, especially those with many layers. It occurs when the gradients of the loss function diminish as they are propagated backward through the network. In contrast if the gradients are large then gradients of loss will explode.

What Happens in Vanishing Gradient?

1. **During Backpropagation**:
    1. Neural networks use the chain rule to compute gradients.
    2. The gradient at each layer is the product of the derivatives of the activation functions and weights.
    3. If these derivatives are small (e.g., for activation functions like sigmoid or tanh), the gradient becomes smaller as it propagates backward through layers.
2. **Result**:
    1. Gradients in earlier layers (close to the input) approach **zero**.
    2. These layers fail to update effectively, while the later layers (closer to the output) continue to learn.

Vanishing or exploding Gradients

Solutions

1.Activation Functions:

Use **ReLU** (Rectified Linear Unit) instead of sigmoid or tanh.

- ReLU's derivative is either 1 or 0, avoiding the issue of small gradients.
- Variants like Leaky ReLU or ELU are also effective.

The **ReLU function** is defined as:

$$f(x) = \max(0, x)$$

Its derivative is:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

2.Weight Initialization:

1. Initialize weights using methods like **He initialization** or **Xavier initialization** to prevent extremely small or large activations.

3. Batch Normalization:

2. Normalize activations at each layer to reduce the risk of gradients vanishing.

4.Residual Networks (ResNets):

3. Introduce skip connections to allow gradients to flow more easily.

Overfitting NN

Regularization and **dropout** are key methods for achieving these goals.

1. Regularization Techniques

a) L1 Regularization (Lasso):

Adds a penalty term proportional to the absolute value of weights: $Loss = Loss\ Function + \lambda \sum |w_i|$

b) L2 Regularization (Ridge or Weight Decay):

Adds a penalty term proportional to the square of the weights: $Loss = Loss\ Function + \lambda \sum w_i^2$

c) Elastic Net:

Combines L1 and L2 regularization: $Loss = Loss\ Function + \lambda_1 \sum |w_i| + \lambda_2 \sum w_i^2$

2. Early Stopping:

Monitor validation loss during training. Stop when validation loss stops decreasing or starts increasing.

Effect: Prevents overfitting by halting training before the model memorizes the data.

3. Dropout

Dropout randomly sets a fraction of neurons to zero during each training step to prevent over-reliance on specific neurons.

Dropout(0.5) means 50% of the neurons are "dropped out."

Overfitting NN - practice

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l1, l2, l1_l2
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Mock Ads Dataset
# Features: user demographics, browsing history, etc.
# Target: click or no-click
np.random.seed(42)
X = np.random.rand(1000, 10)  # 1000 samples, 10 features
y = np.random.randint(0, 2, 1000)  # Binary classification: 0 or 1

# Split data
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)

# Define the model
def build_model(regularizer=None, dropout_rate=0.5):
    model = Sequential()
    model.add(Dense(64, activation='relu', input_dim=X_train.shape[1],
kernel_regularizer=regularizer))
    model.add(Dropout(dropout_rate))  # Dropout layer
    model.add(Dense(32, activation='relu', kernel_regularizer=regularizer))
    model.add(Dropout(dropout_rate))  # Dropout layer
    model.add(Dense(1, activation='sigmoid'))
    return model

# Compile the model with different regularization techniques
models = {
    "L1 Regularization": build_model(regularizer=l1(0.01)),
    "L2 Regularization": build_model(regularizer=l2(0.01)),
    "Elastic Net Regularization": build_model(regularizer=l1_l2(l1=0.01, l2=0.01)),
}

# Early Stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Training each model
histories = {}
for name, model in models.items():
    print(f"\nTraining model with {name}...\n")
    model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy',
metrics=['accuracy'])
    histories[name] = model.fit(
        X_train,
        y_train,
        epochs=50,
        batch_size=32,
        validation_data=(X_val, y_val),
        callbacks=[early_stopping],
        verbose=1
    )

# Evaluate the models
for name, model in models.items():
    val_loss, val_acc = model.evaluate(X_val, y_val, verbose=0)
    print(f"{name} - Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc:.4f}")
```

Tensors

Tensors are **multi-dimensional arrays** that generalize scalars, vectors, and matrices to higher dimensions.

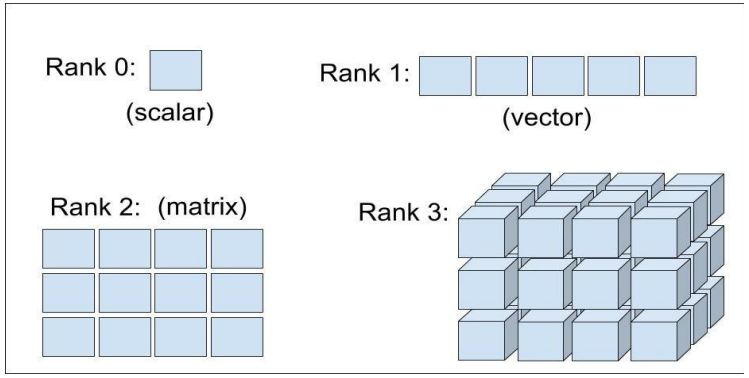
Tensors are fundamental in deep learning as they efficiently handle the multi-dimensional data structures needed for neural networks. Understanding how to manipulate them is crucial for building and training models.

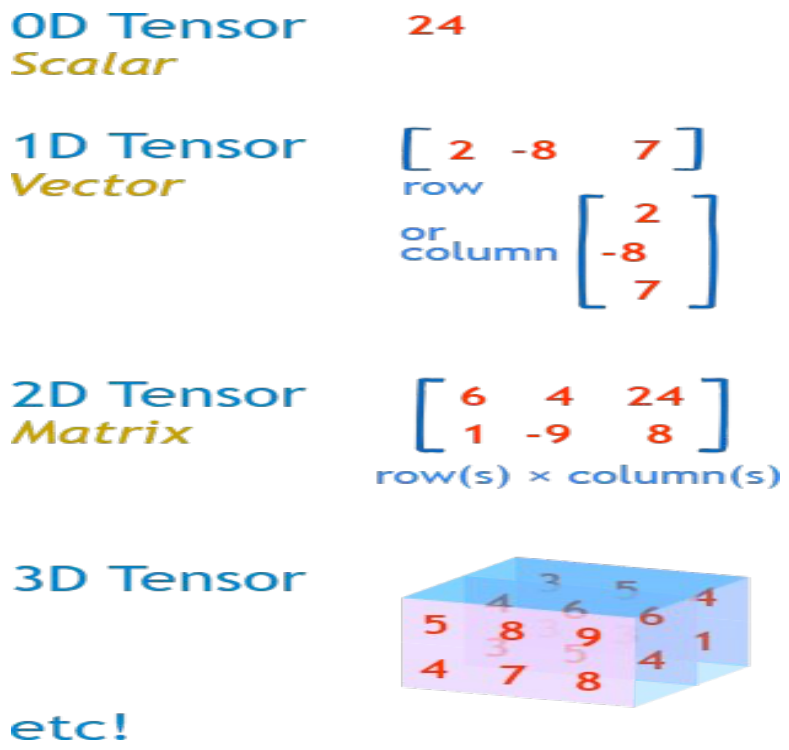
To work with tensors in deep learning, two powerful frameworks are available **TensorFlow** and **PyTorch**. The choice between them often depends on your specific needs and preferences:

TensorFlow: Ideal for large-scale production deployments and complex models. Offers a broader ecosystem of tools and libraries.

Keras: A high-level API built on top of TensorFlow, known for its user-friendliness and ease of use.

PyTorch: Well-suited for research and rapid prototyping due to its dynamic nature and Pythonic interface.





Thank you