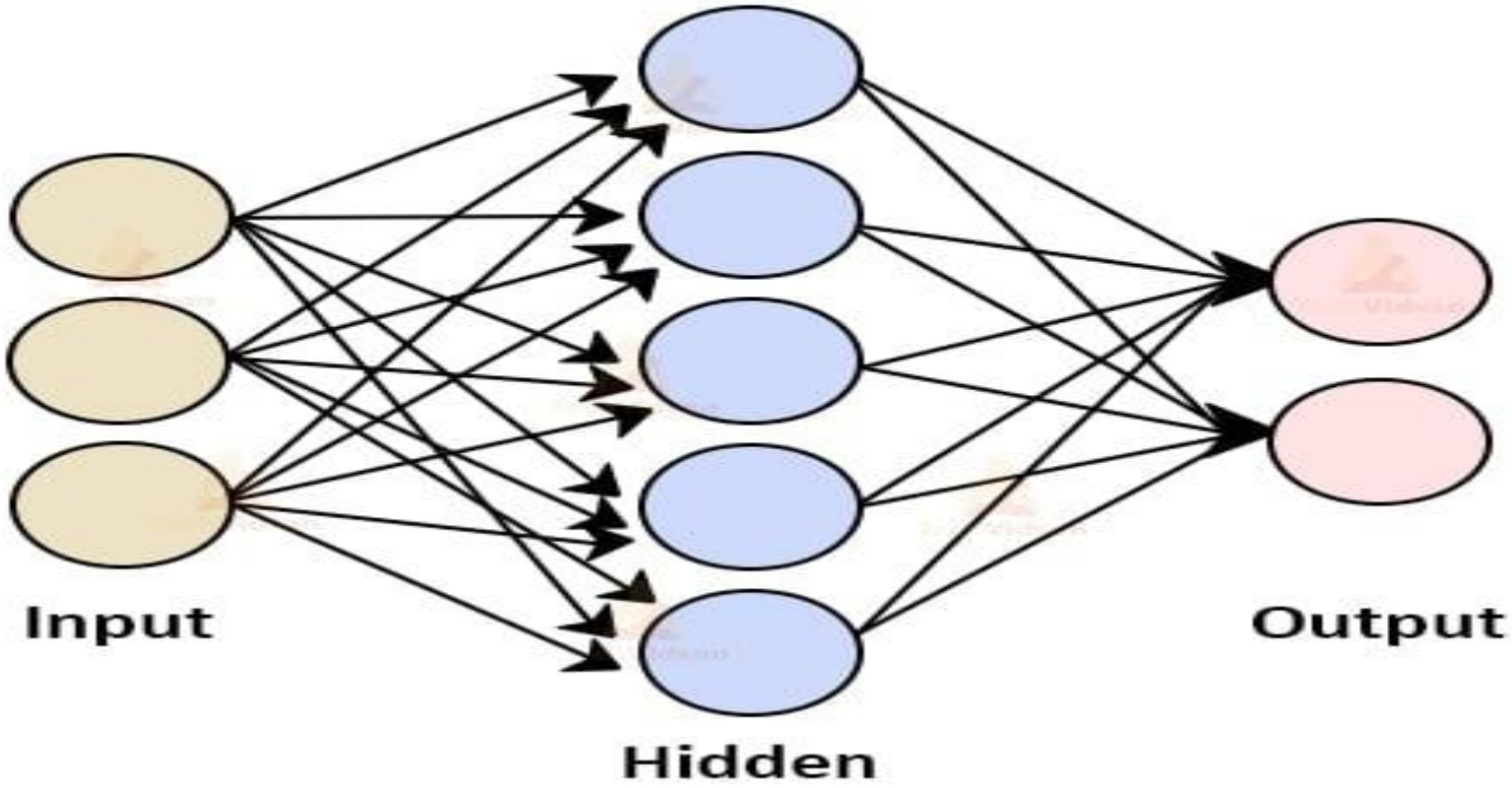
KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanaguda, Hyderabad.

BUILD ANN FROM SCRATCH

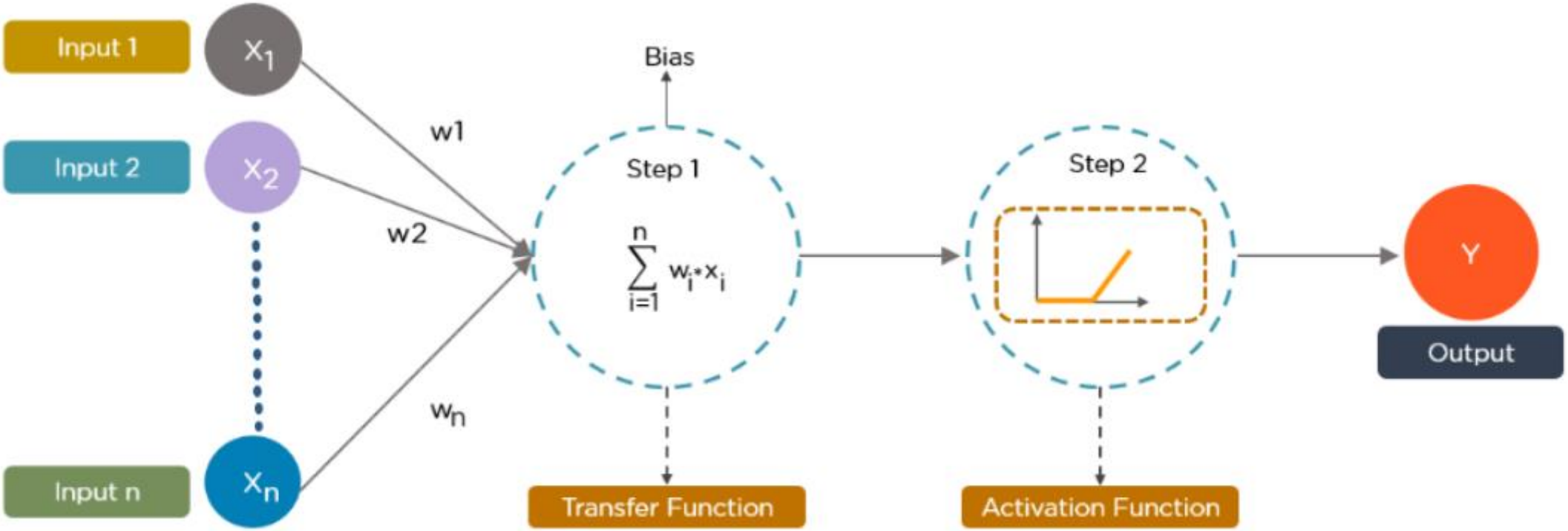
By

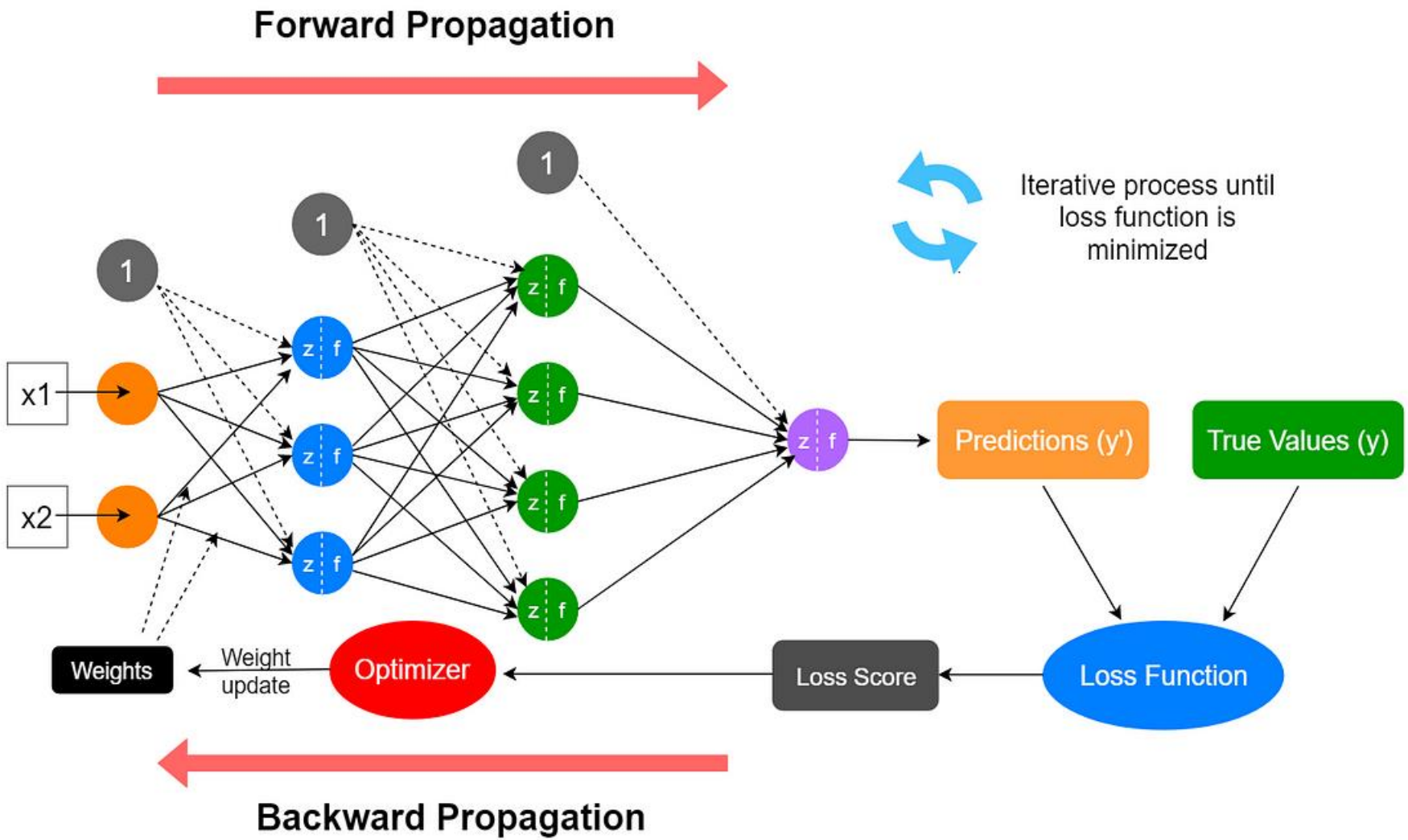
Asha

Architecture of Artificial Neural Network



WORK FLOW


Input 1
Input 2
Input n
X_1
X_2
X_n
w1
w2
w_n
Bias
Step 1
$\sum_{i=1}^{n} w_i * x_i$
Step 2
Y
Output
Transfer Function
Activation Function


Forward Propagation
1
1
1
x1
x2
z f
z f
z f
z f
z f
z f
z f
Iterative process until loss function is minimized
Predictions (y')
True Values (y)
Loss Function
Loss Score
Optimizer
Weight update
Weights
Backward Propagation

BUILDING ANN FROM SCRATCH

Step 1: Load the Dataset

Import necessary libraries and Load the dataset insurance.csv.

Step 2: Separate the features (X) and the target variable (y) and Split the Dataset

Separate the features (X) and the target variable (y) and Split the data into training and testing sets. This helps in evaluating the performance of the ANN on unseen data.

Step 3: Normalize the Features

Normalize the input features to ensure all features contribute equally to the training process. This can be done using methods like standard scaling.

Step 4: Initialize the ANN Parameters

Decide on the architecture of the ANN, including the number of layers and the number of neurons in each layer. Initialize the weights and biases for each layer randomly.

Step 5: Define the Activation Functions

Choose activation functions for the neurons in each layer.

Step 6: Forward Propagation

Implement the forward propagation process, where the input data passes through the network layer by layer, applying the weights, biases, and activation functions to produce the final output.

Step 7: Compute the Loss

Calculate the loss using an appropriate loss function. For binary classification, the binary cross-entropy loss is typically used.

Step 8: Backward Propagation

Implement backward propagation to compute the gradients of the loss function with respect to the weights and biases. This involves calculating the gradient of the loss function from the output layer back to the input layer.

Step 9: Update the Weights and Biases

Use an optimization algorithm, such as gradient descent, to update the weights and biases using the gradients computed during backward propagation. This step aims to minimize the loss function.

Step 10: Train the ANN

Iterate through the training dataset for a specified number of epochs. In each epoch, perform forward propagation, compute the loss, perform backward propagation, and update the weights and biases.

Step 11: Evaluate the ANN

After training, evaluate the performance of the ANN on the testing set. This involves using the trained model to make predictions on the test data and comparing these predictions to the true labels to calculate metrics like accuracy.

Step 12: Make Predictions

Use the trained ANN to make predictions on new or unseen data by performing forward propagation with the learned weights and biases.

Step 1: Load the Dataset
Import necessary libraries and Load the dataset insurance.csv.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load the dataset
data = pd.read_csv('insurance.csv')
```

Step 2: Separate the features (X) and the target variable (y) and Split the Dataset

Separate the features (X) and the target variable (y) and Split the data into training and testing sets. This helps in evaluating the performance of the ANN on unseen data.

```
# Separate features and target
X = data.drop('expenses', axis=1).values
y = data['expenses'].values.reshape(-1, 1)


# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,
random_state=42)
```

Step 3: Normalize the Features

Normalize the input features to ensure all features contribute equally to the training process. This can be done using methods like standard scaling.

```
# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Step 4: Initialize the ANN Parameters

Decide on the architecture of the ANN, including the number of layers and the number of neurons in each layer. Initialize the weights and biases for each layer randomly.

```
def initialize_parameters(input_size, hidden_size, output_size):
    np.random.seed(42)
    W1 = np.random.randn(input_size, hidden_size) * 0.01
    b1 = np.zeros((1, hidden_size))
    W2 = np.random.randn(hidden_size, output_size) * 0.01
    b2 = np.zeros((1, output_size))
    return W1, b1, W2, b2
```

Step 5: Define the Activation Functions

Choose activation functions for the neurons in each layer.

```
def relu(z):
    return np.maximum(0, z)

def relu_derivative(z):
    return np.where(z > 0, 1, 0)

def identity(z):
    return z
```

Step 6: Forward Propagation

Implement the forward propagation process, where the input data passes through the network layer by layer, applying the weights, biases, and activation functions to produce the final output.

```
def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = identity(Z2)
    return Z1, A1, Z2, A2
```

Step 8: Backward Propagation

Implement backward propagation to compute the gradients of the loss function with respect to the weights and biases. This involves calculating the gradient of the loss function from the output layer back to the input layer.

```
def backward_propagation(X, y, Z1, A1, A2, W2):
    m = X.shape[0]

    dZ2 = A2 - y
    dW2 = (1/m) * np.dot(A1.T, dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (1/m) * np.dot(X.T, dZ1)
    db1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)

    return dW1, db1, dW2, db2
```

23rd September 2024

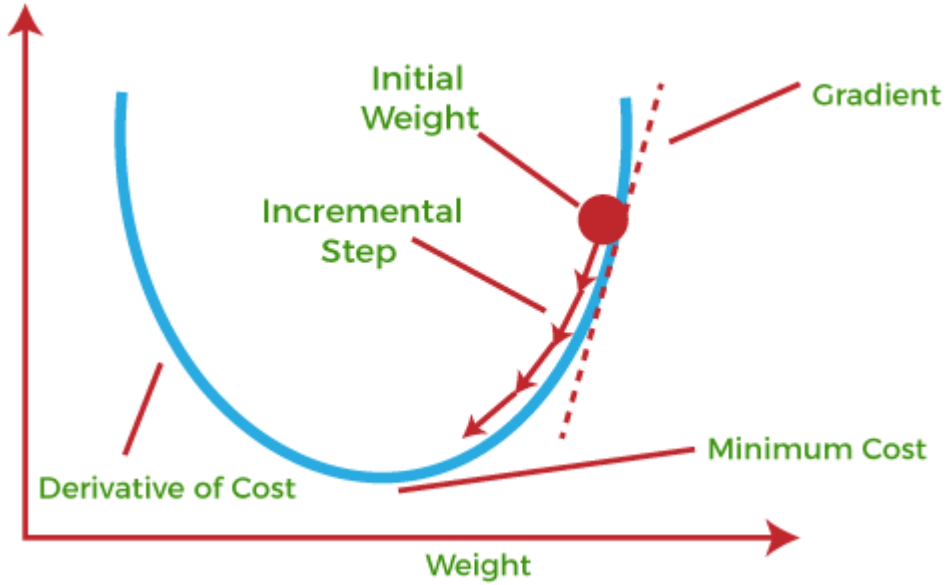
Step 9: Update the Weights and Biases

Use an optimization algorithm, such as gradient descent, to update the weights and biases using the gradients computed during backward propagation. This step aims to minimize the loss function.

Gradient Descent is known as one of the most commonly used optimization algorithms to train Neural Networks.

In mathematical terminology, Optimization algorithm refers to the task of minimizing/maximizing an objective function f(x) parameterized by x.

Similarly, in machine learning, optimization is the task of minimizing the cost function parameterized by the model's parameters.



- Gradient Descent (GD) works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function until the minimum of the cost function is reached.
- The learning happens during the **backpropagation** while training the neural network-based model.
- **Gradient Descent** is used to optimize the weight and biases based on the cost function. The cost function evaluates the difference between the actual and predicted outputs.

How do Gradients Work in Backpropagation?

In a neural network, the training process involves multiple layers of neurons. The process of computing the gradients for all the layers is called **backpropagation**. Here's how it works:

- **Forward Pass**: Inputs are passed through the network, layer by layer, and predictions are made.
- **Loss Calculation**: The loss (or error) is calculated by comparing the predicted output to the actual target values.
- **Backpropagation**: Using the chain rule from calculus, we calculate the gradients of the loss with respect to the network's parameters (weights and biases) by moving backward through the network, layer by layer.
    - The key here is that gradients for one layer depend on the gradients from the next layer, which is where the **chain rule** comes in.

Gradient Descent Algorithm

Once gradients are computed, they are used in **gradient descent** to update the parameters and reduce the loss. This process works as follows:

1.Initialize: Start with random weights and biases.

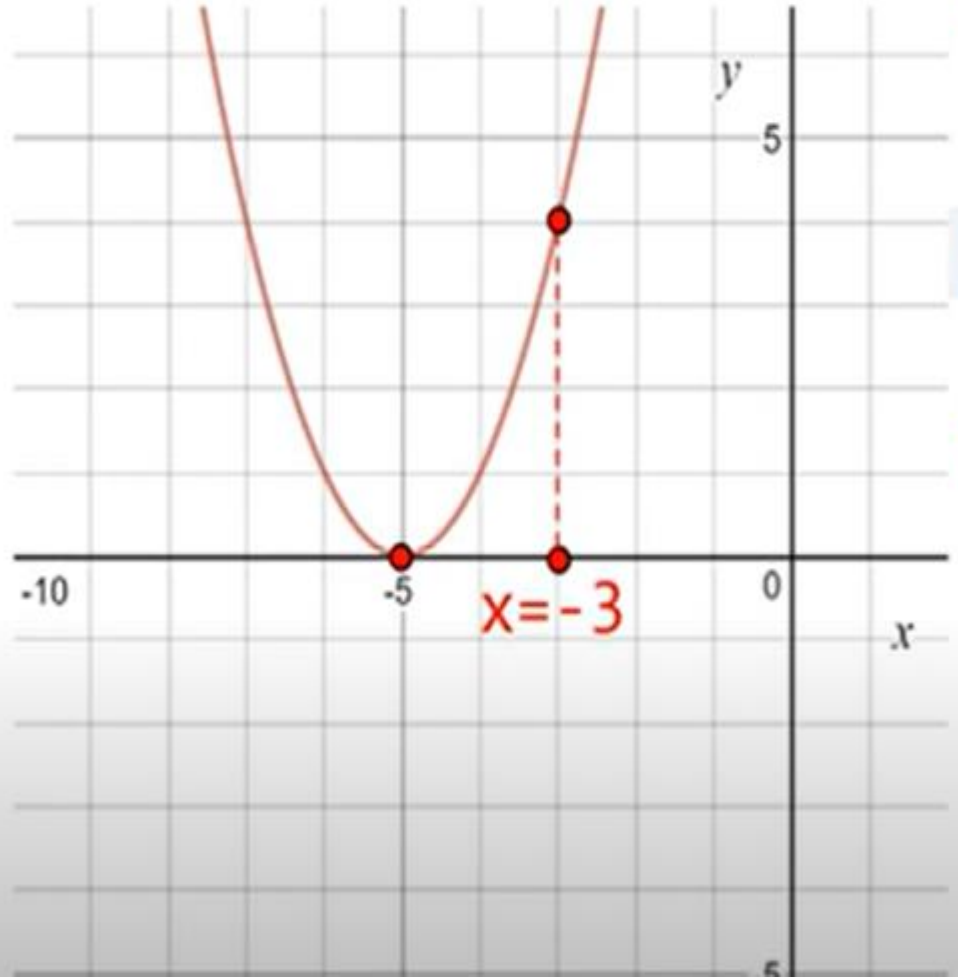
2.Compute Gradients: For each parameter (weight, bias), compute how much the loss changes when that parameter changes (this is the gradient).

3.Update Parameters: Update each parameter by moving in the direction of the negative gradient.

$$\theta = \theta - \eta \frac{\partial L}{\partial \theta}$$

where θ is the parameter (weights or biases), ∂L/∂θ is the gradient of the loss with respect to θ and η is the learning rate (step size).

Example

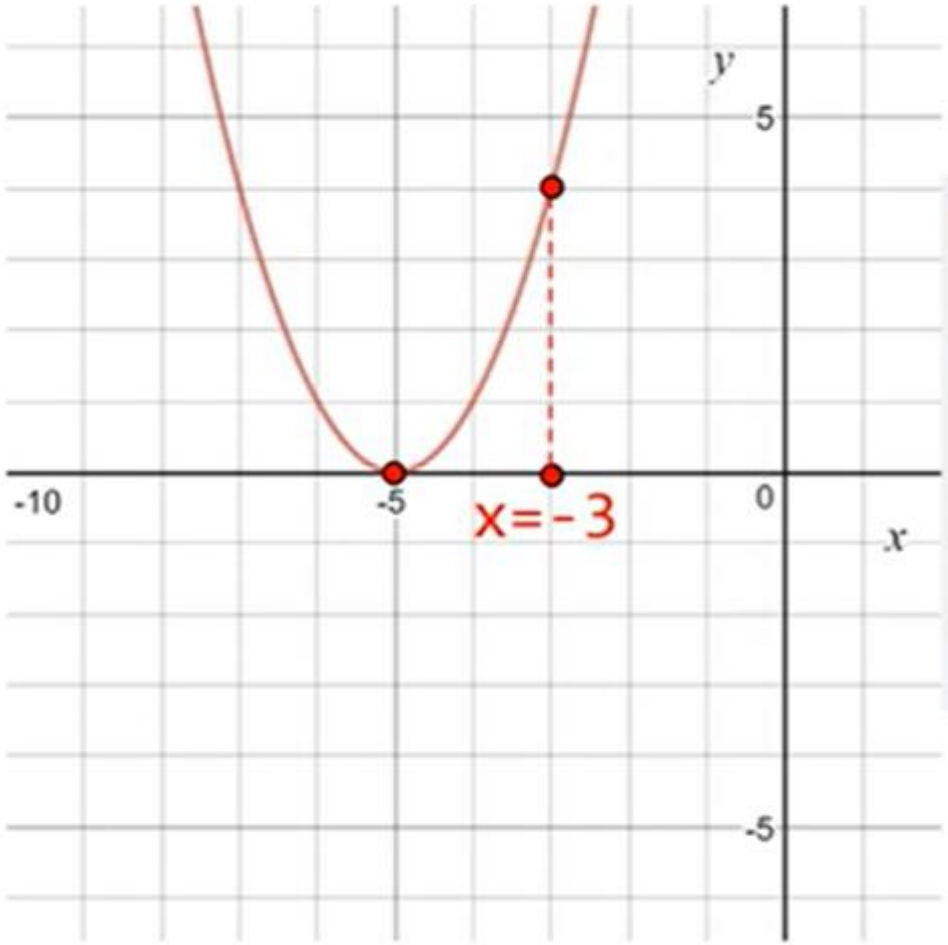


$y=(x+5)^2$

#Step1

Let's start from random point $x=-3$.

Then, find the gradient of the function, $dy/dx = 2\times(x+5)$.



$y=(x+5)^2$

#Step1

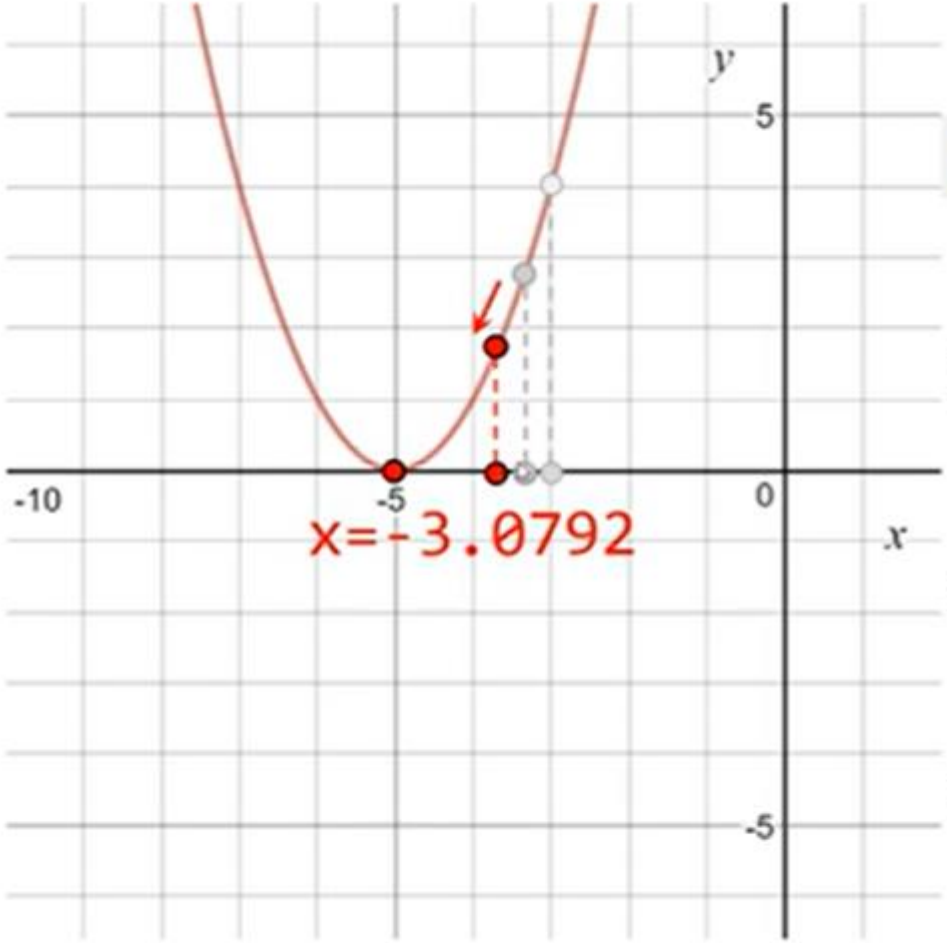
Let's start from random point $x=-3$.

Then, find the gradient of the function, $dy/dx = 2\times(x+5)$.

#Step2

Move in the direction of the **negative of the gradient.**

But: HOW MUCH to move? For that, we define a **learning rate:** `learning_rate = 0.01`



$y=(x+5)^2$

#Step3

Perform **2** iterations of gradient descent.

 # Initialize Parameters

$X_0 = -3$

learning_rate = 0.01

$dy/dx = 2\times(x+5)$

 # Iteration 1

$X_1 = X_0 - (\text{Learning rate}) \times (dy/dx)$

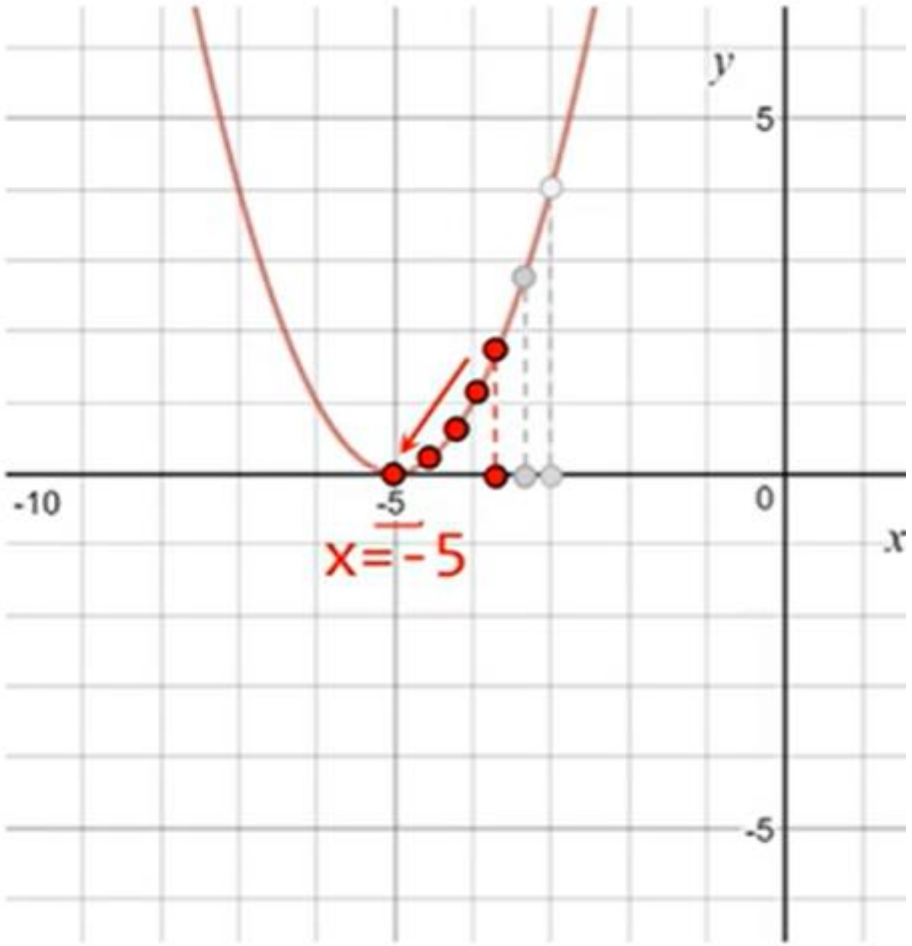
$X_1 = (-3) - (0.01) \times (2 \times ((-3)+5) = -3.04$

 # Iteration 2

$X_2 = X_1 - (\text{Learning rate}) \times (dy/dx)$

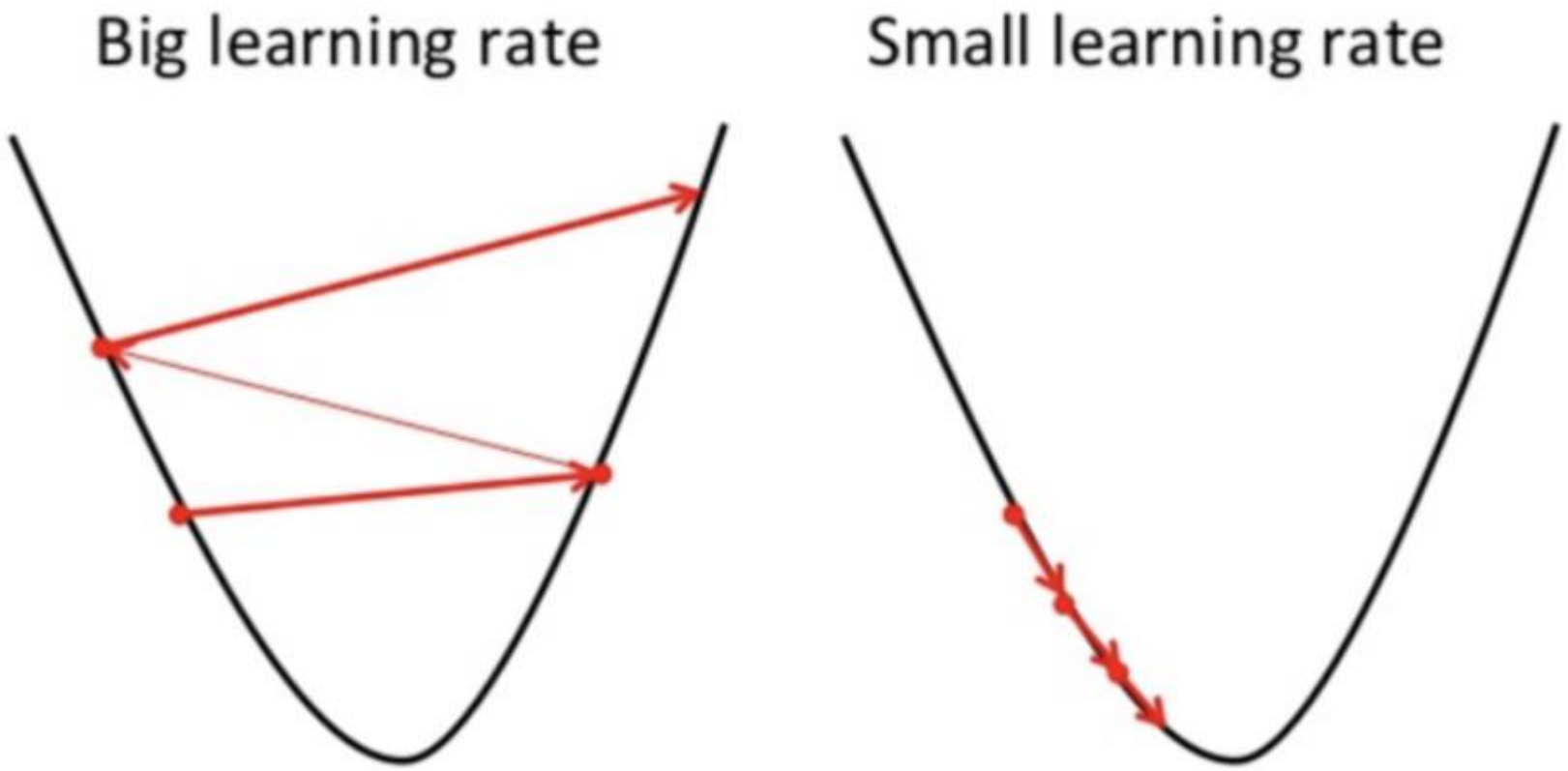
$X_1 = (-3.04) - (0.01) \times (2 \times ((-3.04)+5) =$

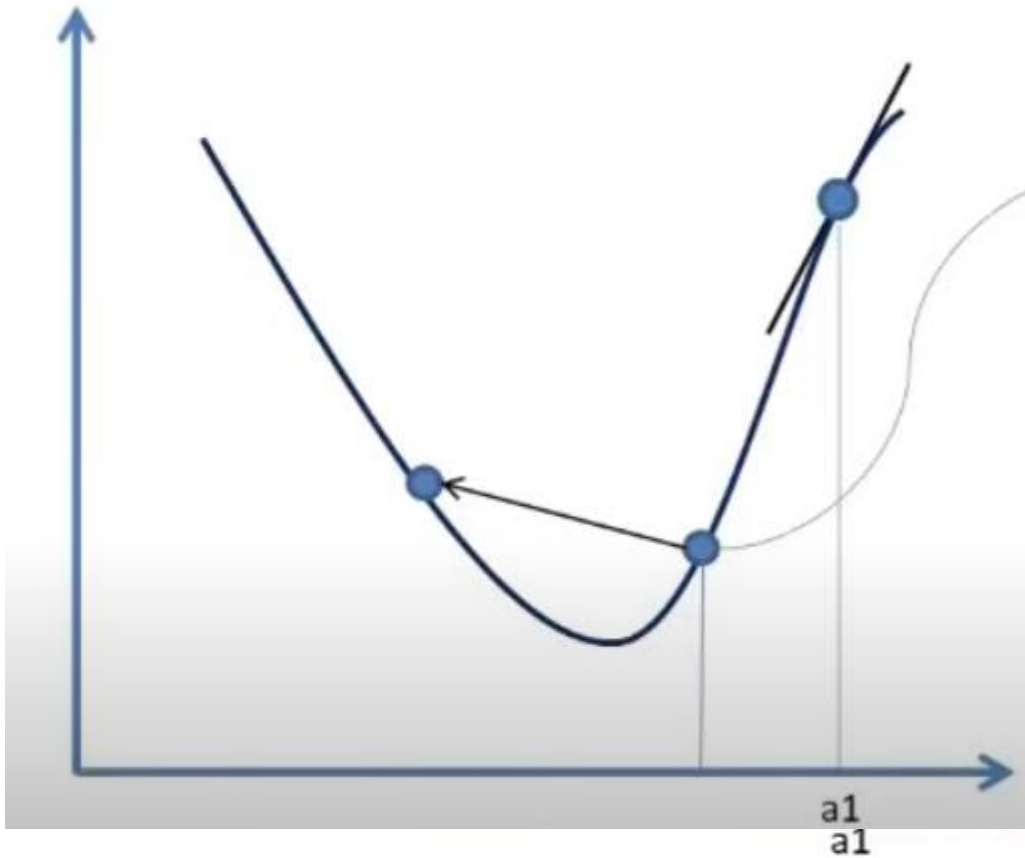
$= -3.0792$


y
5
-10
-5
0
x
-5
x=-5

Learning rate (also referred to as step size or the alpha)

The **learning rate** controls how much to adjust the model's parameters (such as weights and biases) with respect to the gradients in each iteration of the training process.





We reached here in a step from a1 by setting **α to 100.**

Can you guess what can potentially happen in the next iteration?

It will **fail to converge to minimum** and will keep on **oscillating around minimum** but can never converge even in a billion iteration.

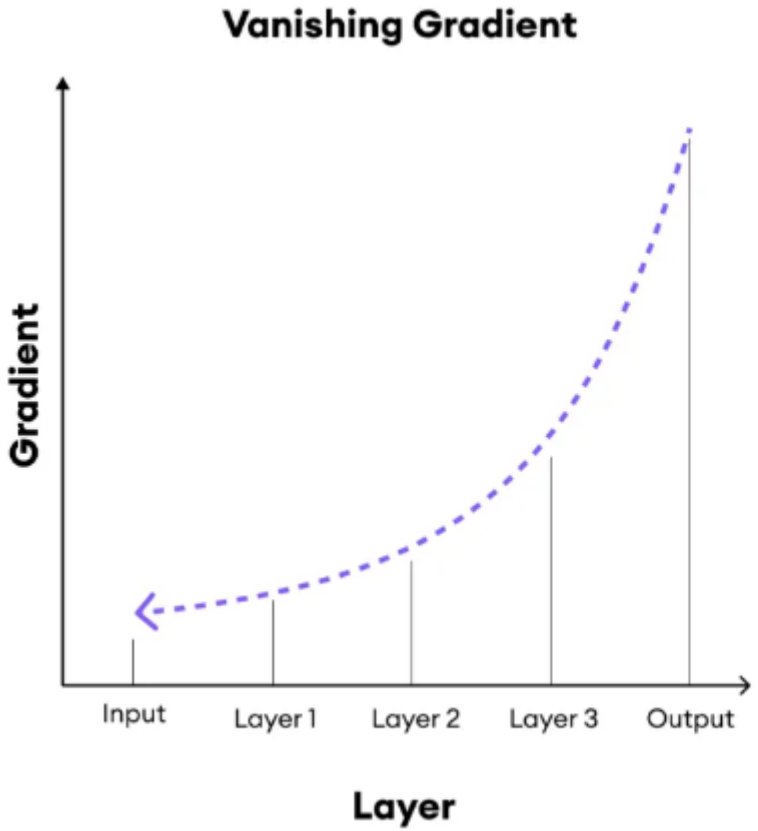
Kmit

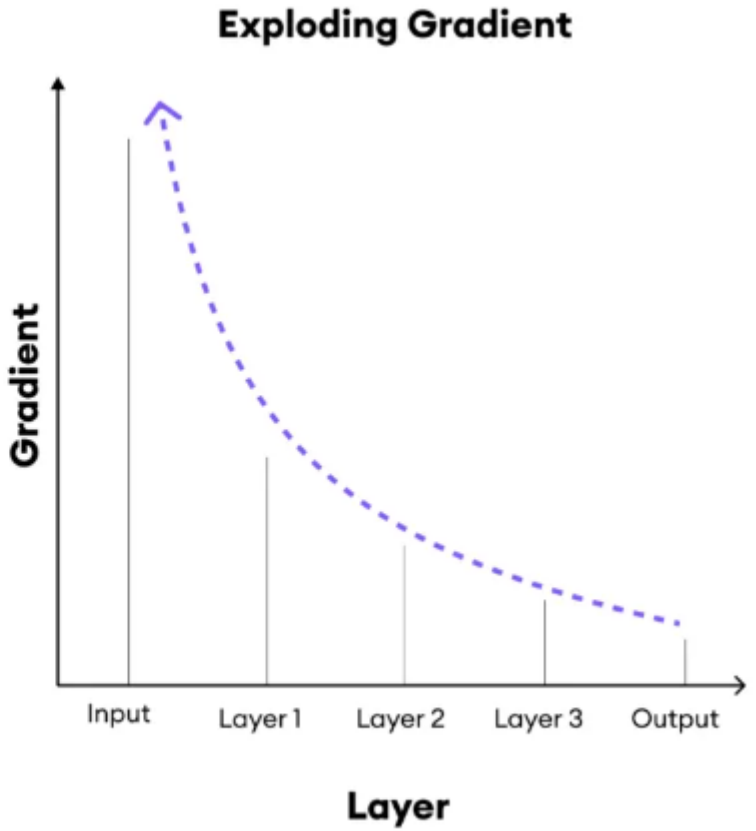
Solution??

Keep **α small** , its value is kept around **0.01** so that it **neither makes gradient descent too slow nor does it fail to converge**.

Challenges with Gradients

- **Vanishing Gradients**: In deep networks, the gradient can become very small in the early layers (close to zero), making learning slow or ineffective. This is called the vanishing gradient problem.
- **Exploding Gradients**: Gradients can also become excessively large, causing the model's parameters to diverge (get too large), which is the exploding gradient problem.


Vanishing Gradient
Gradient
Input
Layer 1
Layer 2
Layer 3
Output
Layer


Exploding Gradient
Gradient
Input
Layer 1
Layer 2
Layer 3
Output
Layer

Step 10: Train the ANN

Iterate through the training dataset for a specified number of epochs. In each epoch, perform forward propagation, compute the loss, perform backward propagation, and update the weights and biases.

An **epoch** in machine learning refers to one complete pass of the entire training dataset through the model. During an epoch, the model processes all the input data once, updating its weights based on the error at each step.

The **training loop** in machine learning refers to the repeated process of passing data through the model, updating the model's parameters, and improving its performance over time. Here's a basic breakdown of a training loop:

Steps in a Training Loop:

1.Initialize Parameters: Before the loop starts, you initialize the model's parameters (weights and biases) randomly or with some small values.

2.Loop for Each Epoch: The model processes the entire training dataset for a certain number of epochs (iterations). Each epoch involves:

1. **Forward Propagation**: The input data is passed through the model, producing predictions.
2. **Loss/Cost Calculation**: The error (cost or loss) between the predicted output and the actual output is computed using a cost function.
3. **Backward Propagation**: The gradients (derivatives) of the loss function with respect to the model's parameters are calculated, showing how much to adjust each parameter.
4. **Parameter Update**: Using the gradients and a learning rate, the model's parameters (weights and biases) are updated using gradient descent.

3.Track Progress: Optionally, after a certain number of epochs, you can monitor the training progress by printing the cost, accuracy, or other performance metrics.

4.Repeat: The loop continues for a predefined number of epochs, or until the model achieves satisfactory performance.

Key Concepts in the Training Loop:

- **Epochs**: The number of times the model processes the entire dataset.
- **Batch**: Often, data is divided into smaller batches to process efficiently (Batch Gradient Descent).
- **Learning Rate**: Controls the size of the steps during the parameter update.

The training loop continues until the model reaches the desired performance or until the maximum number of epochs is reached.

Step 11: Evaluate the ANN

After training, evaluate the performance of the ANN on the testing set. This involves using the trained model to make predictions on the test data and comparing these predictions to the true labels to calculate metrics like accuracy.

```
def predict(X, W1, b1, W2, b2):
    # Your code here
    return predictions
```

The predict function effectively uses the trained neural network to compute and return predictions for new input data by leveraging the forward propagation process.

Step 12: Make Predictions

Use the trained ANN to make predictions on new or unseen data by performing forward propagation with the learned weights and biases.

1.Train the Model: The neural network learns from training data by adjusting its parameters based on the inputs and corresponding outputs over several iterations (epochs).

2.Make Predictions: The trained model is used to predict outputs for a new test dataset that it hasn't seen before.

3.Calculate Mean Squared Error (MSE): The predictions are compared to the actual outcomes from the test dataset to measure the prediction error using MSE.

4.Print MSE: The MSE value is displayed, indicating the model's accuracy—lower values mean better performance.

•**High Cost and MSE**: If Both the cost after the first epoch and the test MSE are very high, suggesting that the model is not learning well. This could be due to various reasons, such as:

•An inappropriate learning rate (too high or too low).

•Insufficient training.

•Poor data quality or features.

•Model architecture issues