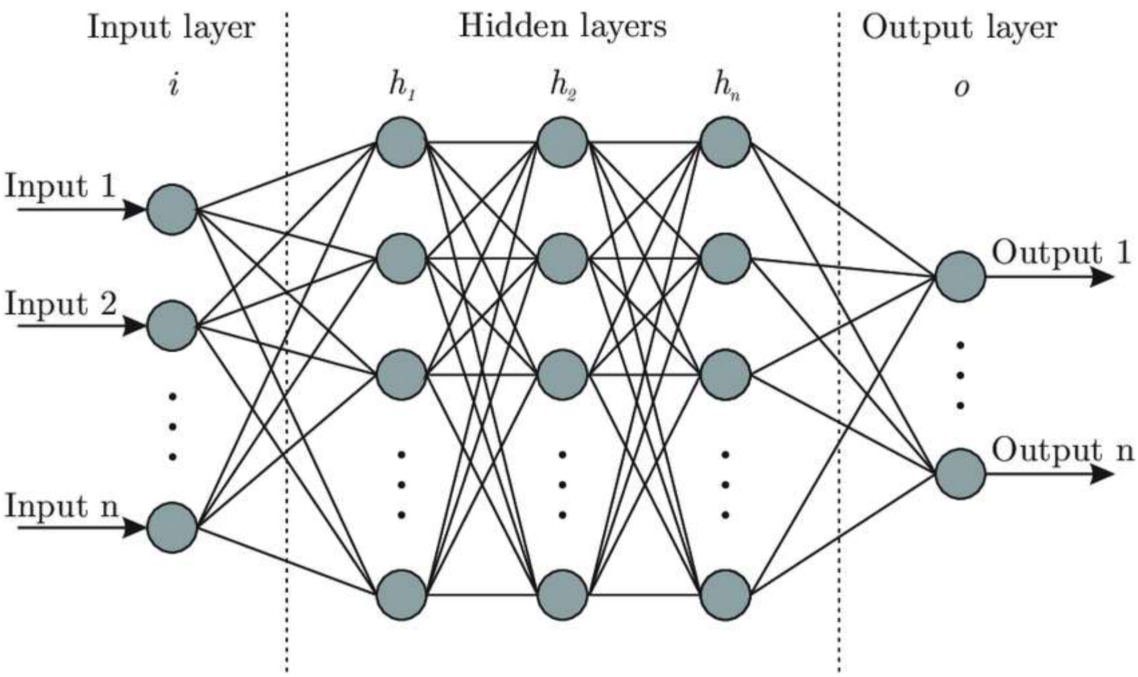
Artificial Neural Networks (ANN)

ANN is a Multilayer Perceptron

Artificial Neural Networks

- Artificial Neural Network (ANN) is a computational model based on the biological neural networks of brains.
- ANN is modeled with three types of layers:
    1. an input layer
    2. hidden layers (one or more), and
    3. an output layer.
- Each layer comprises nodes (like biological neurons) are called Artificial Neurons.
- All nodes are connected with weighted edges (like synapses in biological brains) between two layers.



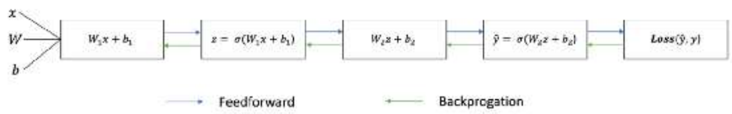
Training the Neural Network

Training the Neural Network

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

- Notice that in the equation above, the weights W and the biases b are the only variables that affects the output ŷ.
- Naturally, **the right values for the weights and biases determines the strength of the predictions**.
- The process of fine-tuning the weights and biases from the input data is known as training the Neural Network.
- Each iteration of the training process consists of the following steps:
  1. Calculating the predicted output ŷ, known as **feedforward**
  2. Updating the weights and biases, known as **backpropagation**

The sequential graph below illustrates the process.

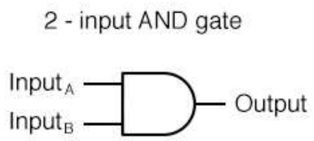


- Initially, **with the forward propagation function, the output is predicted**.
- Then **through backpropagation, the weight and bias to the nodes are updated to minimizing the error in prediction to attain the convergence of cost function in determining the final output**.

Implementation of Artificial Neural Network for AND Logic Gate with 2-bit Binary Input

Approach: 2-input AND gate

AND logical function truth table for 2-bit binary variables, i.e, the input vector $\boldsymbol{x} : (\boldsymbol{x_1}, \boldsymbol{x_2})$ and the corresponding output $\boldsymbol{y}$ –



A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Implementation of the Artificial Neural Network for AND Logic Gate with 2-bit Binary Input:

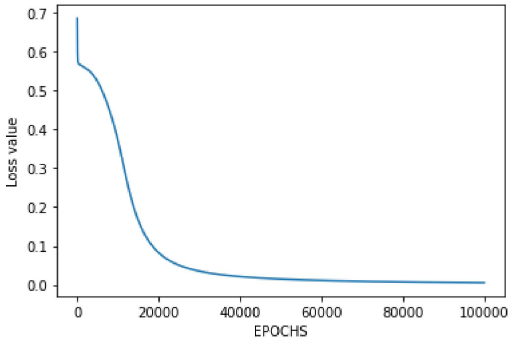
- Step1: Import the required Python libraries
- Step2: Define Activation Function : Sigmoid Function
- Step3: Initialize neural network parameters (weights, bias) and define model hyperparameters (number of iterations, learning rate)
- Step4: Forward Propagation
- Step5: Backward Propagation
- Step6: Update weight and bias parameters
- Step7: Train the learning model
- Step8: Plot Loss value vs Epoch
- Step9: Test the model performance

```
# Step1: Import the required Python libraries
import numpy as np
from matplotlib import pyplot as plt
```

```
# Step2: Define Activation Function : Sigmoid Function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))


# Step3: Initialize neural network parameters (weights, bias) and define model hyperparameters (number of iterations, learning rate)
# Initialization of the neural network parameters
# Initialized all the weights in the range of between 0 and 1
# Bias values are initialized to 0
def initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures):
    W1 = np.random.randn(neuronsInHiddenLayers, inputFeatures)
    W2 = np.random.randn(outputFeatures, neuronsInHiddenLayers)
    b1 = np.zeros((neuronsInHiddenLayers, 1))
    b2 = np.zeros((outputFeatures, 1))

    parameters = {"W1" : W1, "b1": b1,
                  "W2" : W2, "b2": b2}
    return parameters


# Step4: Forward Propagation
def forwardPropagation(X, Y, parameters):
    m = X.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]

    Z1 = np.dot(W1, X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2)
    logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), (1 - Y))
    cost = -np.sum(logprobs) / m
    return cost, cache, A2


# Step5: Backward Propagation
def backwardPropagation(X, Y, cache):
    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2) = cache

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

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, A1 * (1- A1))
    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis = 1, keepdims = True) / m

    gradients = {"dZ2": dZ2, "dW2": dW2, "db2": db2,
                 "dZ1": dZ1, "dW1": dW1, "db1": db1}
    return gradients


# Step6: Update weight and bias parameters
# Updating the weights based on the negative gradients
def updateParameters(parameters, gradients, learningRate):
    parameters["W1"] = parameters["W1"] - learningRate * gradients["dW1"]
    parameters["W2"] = parameters["W2"] - learningRate * gradients["dW2"]
    parameters["b1"] = parameters["b1"] - learningRate * gradients["db1"]
    parameters["b2"] = parameters["b2"] - learningRate * gradients["db2"]
    return parameters


# Step7: Train the learning model
# Model to learn the AND truth table
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]]) # AND input
Y = np.array([[0, 0, 0, 1]]) # AND output

# Define model parameters
neuronsInHiddenLayers = 2 # number of hidden layer neurons (2)
inputFeatures = X.shape[0] # number of input features (2)
outputFeatures = Y.shape[0] # number of output features (1)
parameters = initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures)
epoch = 100000
learningRate = 0.01
losses = np.zeros((epoch, 1))

for i in range(epoch):
    losses[i, 0], cache, A2 = forwardPropagation(X, Y, parameters)
```

```
    gradients = backwardPropagation(X, Y, cache)
    parameters = updateParameters(parameters, gradients, learningRate)
```

```
# Step8: Plot Loss value vs Epoch
# Evaluating the performance
plt.figure()
plt.plot(losses)
plt.xlabel("EPOCHS")
plt.ylabel("Loss value")
plt.show()
```




```
# Step9: Test the model performance
# Testing
X = np.array([[1, 1, 0, 0], [0, 1, 0, 1]]) # AND input
cost, _, A2 = forwardPropagation(X, Y, parameters)
prediction = (A2 > 0.5) * 1.0
# print(A2)
print(prediction)
```

```
[[0. 1. 0. 0.]]
```

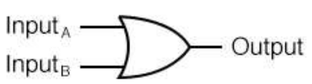
Here, the model predicted output for each of the test inputs are exactly matched with the AND logic gate conventional output (y) according to the truth table and the cost function is also continuously converging.

Latihan 1

Implementasikan artificial neural network untuk gate logika **OR** dengan input 2 bit biner.

Lakukan modifikasi pada contoh program di bagian Step7: Train the learning model: ***Model to learn the AND truth table***.

2 - input OR gate



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Langkah-langkah:

- Step1: Import the required Python libraries
- Step2: Define Activation Function : Sigmoid Function
- Step3: Intialize neural network parameters (weights, bias) and define model hyperparameters (number of iterations, learning rate)
- Step4: Forward Propagation
- Step5: Backward Propagation
- Step6: Update weight and bias parameters
- Step7: Train the learning model
- Step8: Plot Loss value vs Epoch
- Step9: Test the model performance

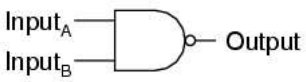
Latihan 2

Implementasikan artificial neural network untuk gate logika **NAND** dengan input 2 bit biner.

Gunakan langkah-langkah penyelesaian yang sama dengan Latihan 1.

Lakukan modifikasi pada contoh program di bagian Step7: Train the learning model: ***Model to learn the AND truth table***.

2-input NAND gate



A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Equivalent gate circuit

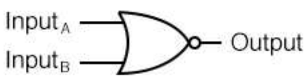
Latihan 3

Implementasikan artificial neural network untuk gate logika **NOR** dengan input 2 bit biner.

Gunakan langkah-langkah penyelesaian yang sama dengan Latihan 1.

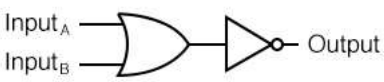
Lakukan modifikasi pada contoh program di bagian Step7: Train the learning model: ***Model to learn the AND truth table***.

2 - input NOR gate



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

Equivalent Gate Circuit



References:

- https://www.geeksforgeeks.org/implementation-of-artificial-neural-network-for-and-logic-gate-with-2-bit-binary-input/
- https://www.allaboutcircuits.com/textbook/digital/chpt-3/multiple-input-gates/