Fuzzy Logic & Neural Networks (CS-514)

Dr. Sudeep Sharma

IIIT Surat

sudeep.sharma@iiitsurat.ac.in

Backpropagation

Forward Pass

- Let's start with a simplified forward pass with just one neuron.
- Let's backpropagate the ReLU function for a single neuron.
- We're first doing this only as a demonstration to simplify the explanation.
- We'll start by showing how we can understand the chain rule with derivatives and partial derivatives to calculate the impact of each variable on the ReLU activated output.

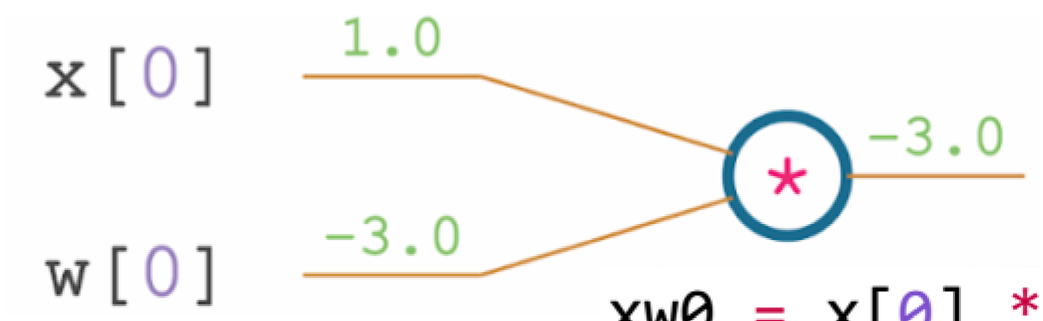
Backpropagation

Forward Pass

- We’ll use an example neuron with 3 inputs, which means that it also has 3 weights and a bias.

```
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias
```

- We have to multiply the input by the weight:

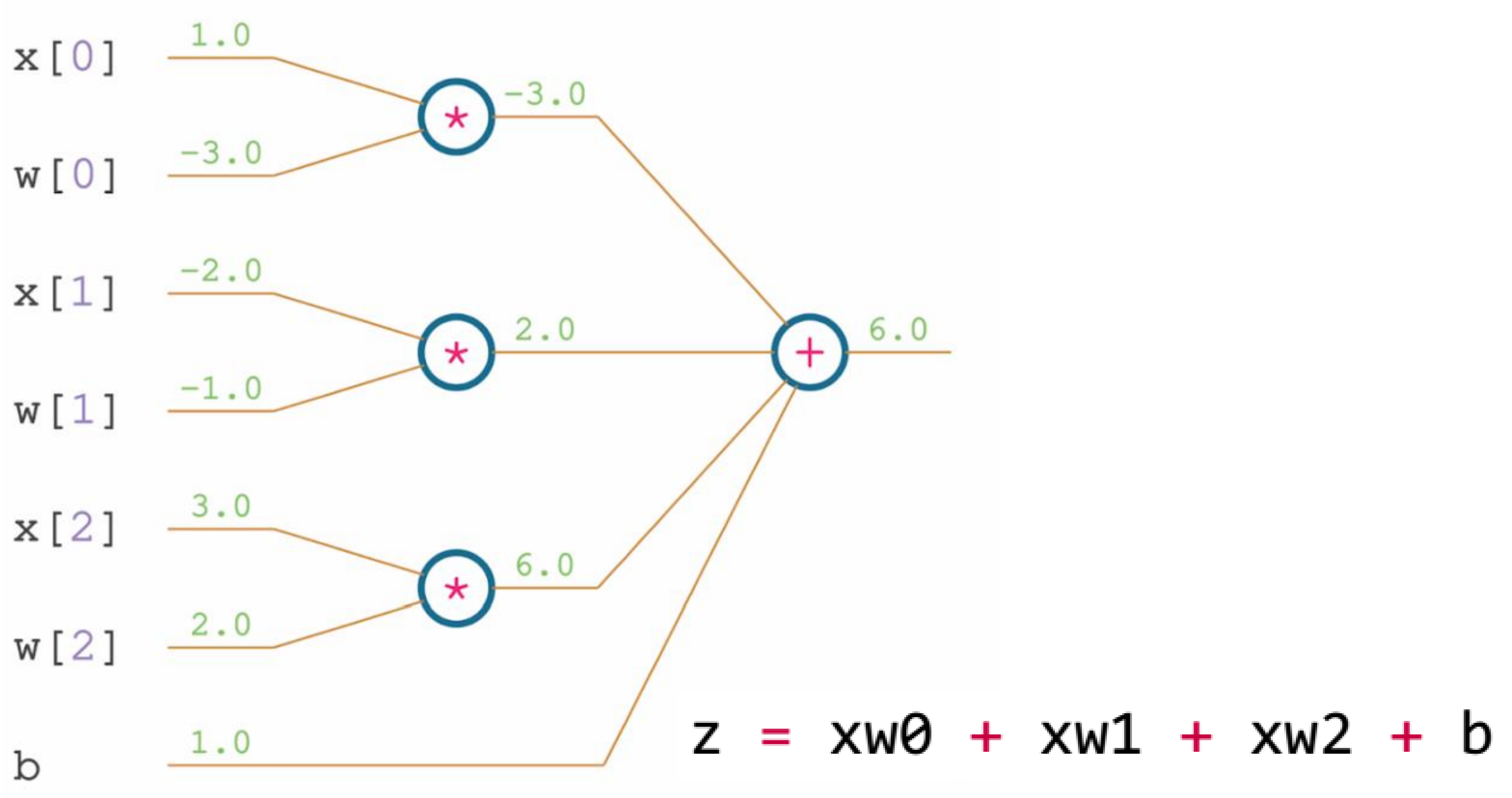



```
xw0 = x[0] * w[0]
```

Backpropagation

Forward Pass

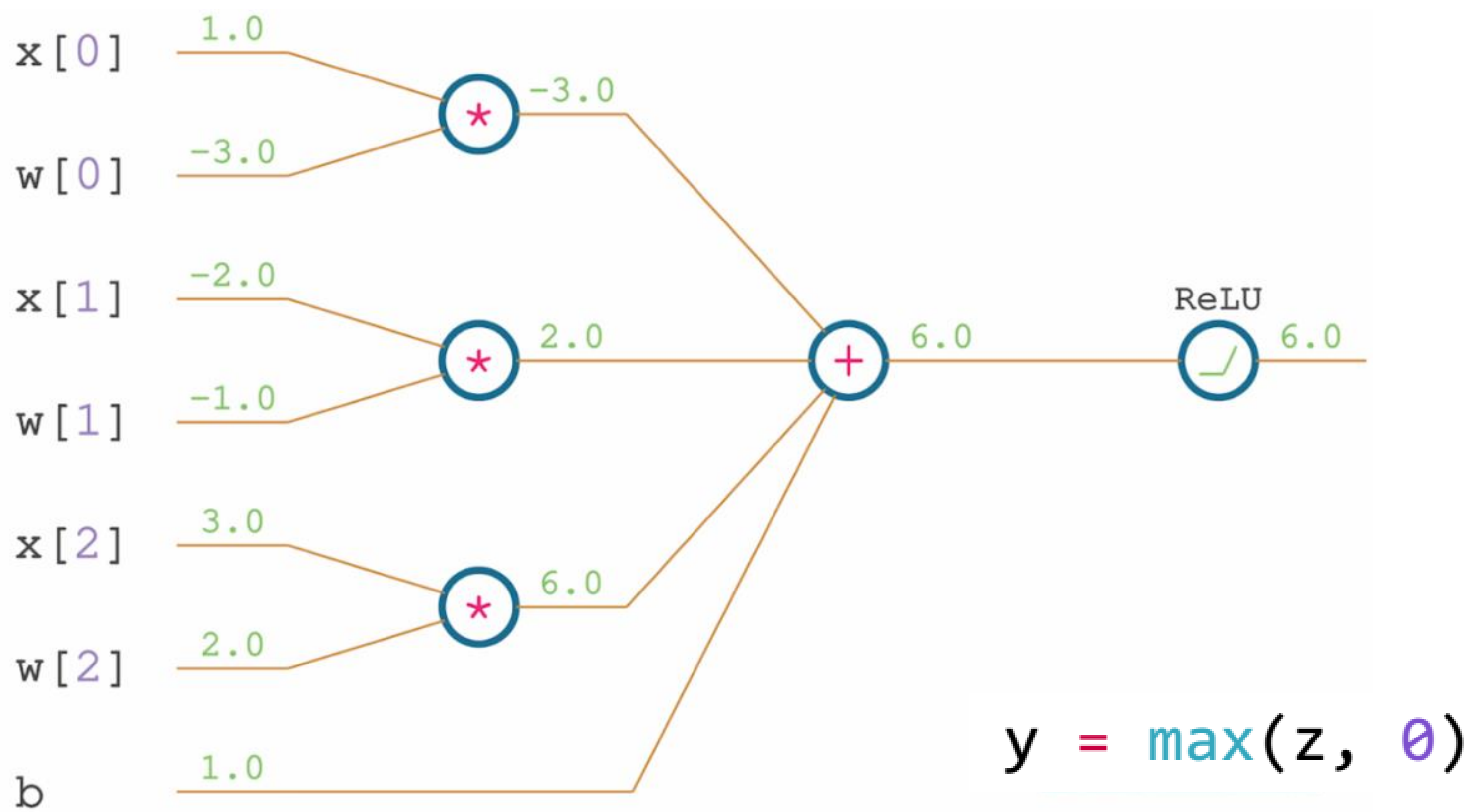
- The next operation to perform is a sum of all weighted inputs with a bias:



Backpropagation

Forward Pass

- The last step is to apply the ReLU activation function on this output:



Backpropagation

The Chain Rule

- The first step is to backpropagate our gradients by calculating derivatives and partial derivatives with respect to each of our parameters.
- To do this, we're going to use the chain rule as the derivative for nested functions:

$$\frac{d}{dx}f(g(x)) = \frac{d}{dg(x)}f(g(x)) \cdot \frac{d}{dx}g(x) = f'(g(x)) \cdot g'(x)$$

- For our neural network:

$$ReLU(\sum[inputs \cdot weights] + bias)$$

Backpropagation

The Chain Rule

- For our neural network:

$$ReLU\left(\sum[inputs \cdot weights] + bias\right)$$

- Or

$$ReLU(x_0 w_0 + x_1 w_1 + x_2 w_2 + b)$$

- Let's rewrite our equation to the form that will allow us to determine how to calculate the derivatives more easily:

$$y = ReLU(sum(mul(x_0, w_0), mul(x_1, w_1), mul(x_2, w_2), b))$$

Backpropagation

The Chain Rule

- The equation contains 3 nested functions: ReLU , a sum of weighted inputs and a bias, and multiplications of the inputs and weights.

$$y = ReLU(sum(mul(x_0, w_0), mul(x_1, w_1), mul(x_2, w_2), b))$$

- To calculate the impact of the parameter, w_0 , on the output, the chain rule tells us:

$$\frac{\partial}{\partial w_0}\left[\mathrm{Re}LU\left(sum\left(mul\left(x_0, w_0\right), mul\left(x_1, w_1\right), mul\left(x_2, w_2\right), b\right)\right)\right] =$$

$$\frac{\partial \mathrm{Re}LU(\ )}{\partial sum(\ )} \cdot \frac{\partial sum(\ )}{\partial mul\left(x_0, w_0\right)} \cdot \frac{\partial\left(x_0 w_0\right)}{\partial w_0}$$

Backpropagation

The Chain Rule

- We want to know the impact of a given weight or bias on the output's loss function.
- During the backward pass, we'll calculate the derivative of the loss function.
- Then derivative of the activation function of the output layer.
- Then derivative of the output layer, and so on, through all of the hidden layers and activation functions.
- The derivative with respect to the weights and biases will form the gradients that we'll use to update the weights and biases.

Backpropagation

The Chain Rule

- Recall that the derivative of ReLU() with respect to its input is 1, if the input is greater than 0, and 0 otherwise as:

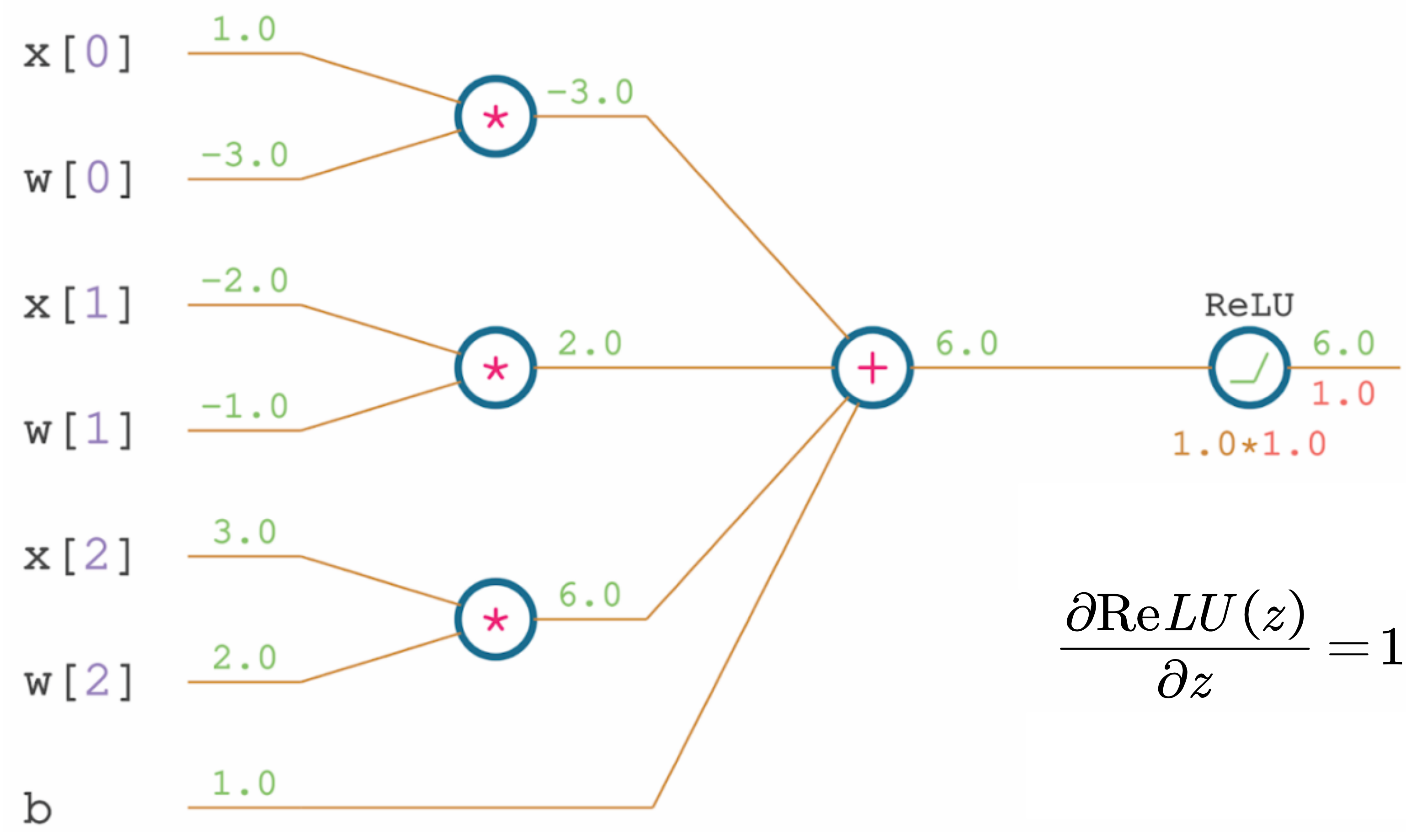
$$\frac{\partial \mathrm{Re}LU(sum(\ ))}{\partial sum(\ )} = \frac{\partial \mathrm{Re}LU(z)}{\partial z} = \frac{\partial}{\partial z}\max(z,0) = 1(z>0)$$

- The input value to the ReLU function is 6 , so the derivative equals 1.

Backpropagation

The Chain Rule

- Use the chain rule and multiply this derivative with the derivative received from the next layer



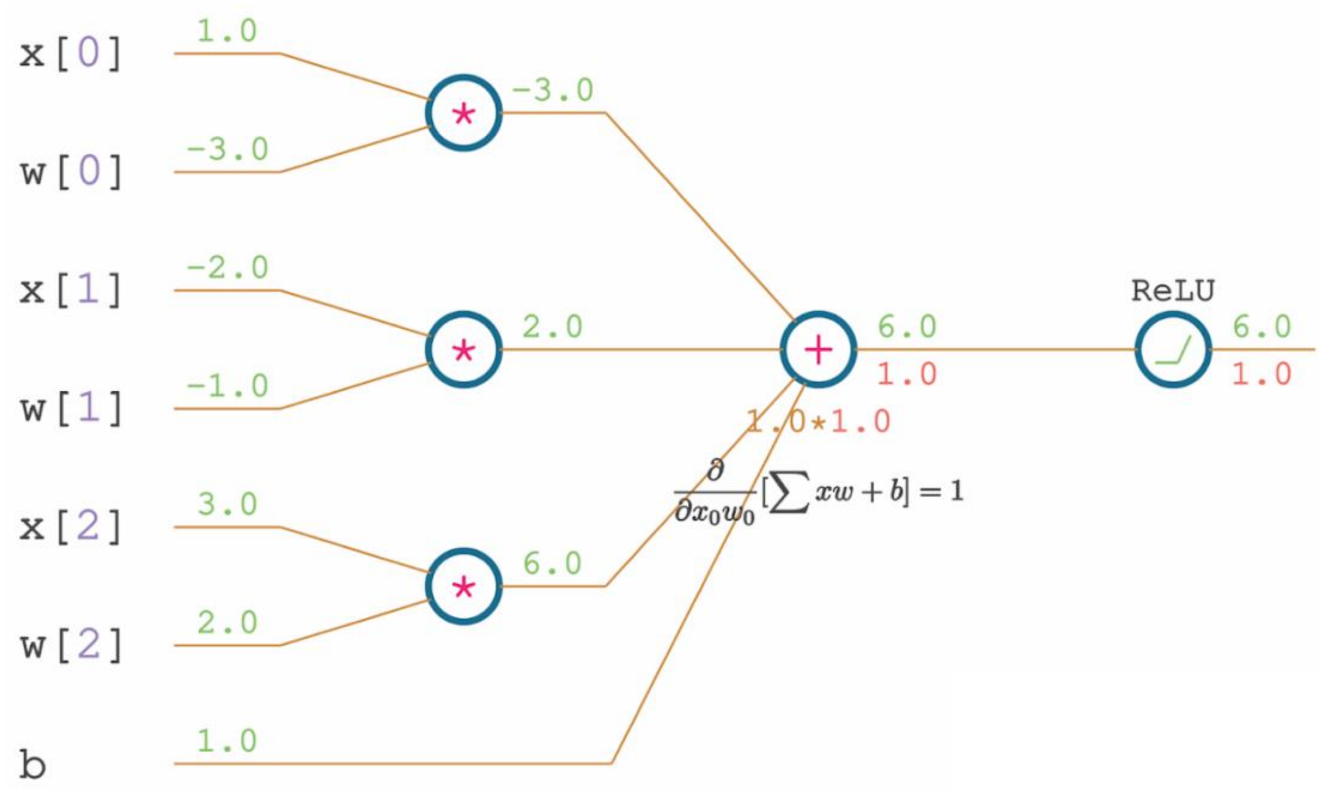
$$\frac{\partial \mathrm{Re}LU(z)}{\partial z} = 1$$

Backpropagation

The Chain Rule

➢ The partial derivative of the sum operation

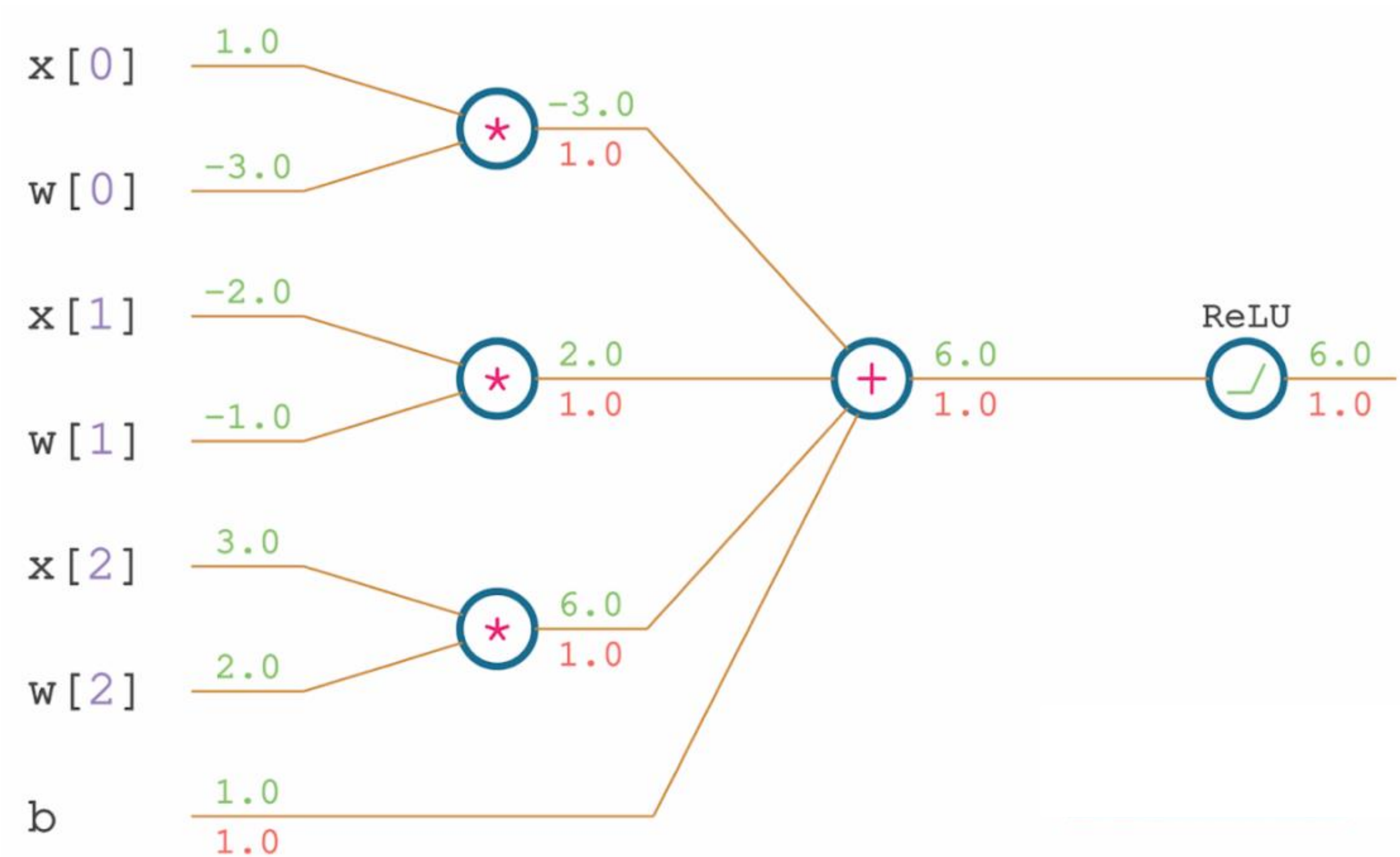
$$\frac{\partial sum(\ )}{\partial mul(x_0, w_0)} = \frac{\partial (x_0 w_0 + x_1 w_1 + x_2 w_2 + b)}{\partial x_0 w_0} = 1$$



Backpropagation

The Chain Rule

- The partial derivative of the sum operation for all input & weight pairs and bias



Backpropagation

The Chain Rule

- The partial derivative of the product terms:

$$f(x,y) = x \cdot y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x,y) = y$$

$$\frac{\partial}{\partial y} f(x,y) = x$$

$$\frac{\partial mul(x_0, w_0)}{\partial w_0} = \frac{\partial (x_0 w_0)}{\partial w_0} = x_0$$

Backpropagation

The Chain Rule

- The overall gradient terms are:

$$\frac{\partial \mathrm{Re}LU(sum(\ ))}{\partial w_0} = \frac{\partial \mathrm{Re}LU(sum(\ ))}{\partial sum(\ )} \times \frac{\partial sum(\ )}{\partial mul(x_0, w_0)} \times \frac{\partial mul(x_0, w_0)}{\partial w_0} = 1 \times 1 \times x_0 = x_0$$

$$\frac{\partial \mathrm{Re}LU(sum(\ ))}{\partial w_1} = \frac{\partial \mathrm{Re}LU(sum(\ ))}{\partial sum(\ )} \times \frac{\partial sum(\ )}{\partial mul(x_1, w_1)} \times \frac{\partial mul(x_1, w_1)}{\partial w_1} = 1 \times 1 \times x_1 = x_1$$

$$\frac{\partial \mathrm{Re}LU(sum(\ ))}{\partial w_2} = \frac{\partial \mathrm{Re}LU(sum(\ ))}{\partial sum(\ )} \times \frac{\partial sum(\ )}{\partial mul(x_2, w_2)} \times \frac{\partial mul(x_2, w_2)}{\partial w_2} = 1 \times 1 \times x_2 = x_2$$

$$\frac{\partial \mathrm{Re}LU(sum(\ ))}{\partial b} = \frac{\partial \mathrm{Re}LU(sum(\ ))}{\partial sum(\ )} \times \frac{\partial sum(\ )}{\partial b} = 1 \times 1 = 1$$

Backpropagation

The Parameter Update Rule

- The i^{th} weight (w_i) is updated by the rule:

$$\boxed{w_i(new) = w_i(old) - \alpha \frac{\partial L}{\partial w_i}}$$

- α is known as "learning rate"

- L is known as the "loss function"

$$b_j(new) = b_j(old) - \alpha \frac{\partial L}{\partial b_j}$$

Backpropagation

The Parameter Update Rule

- For our example:

```
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias
```

- If α = 0.01, then

$$w_0(new) = w_0(old) - \alpha \frac{\partial \mathrm{Re}LU(\ )}{\partial w_0}$$

$$w_0(new) = -3.0 - 0.01 \times x_0$$

$$w_0(new) = -3.0 - 0.01 \times 1 = -3.01$$

Backpropagation

The Parameter Update Rule

- For our example:

```
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias
```

- If α = 0.01, then

$$w_1(new) = w_1(old) - \alpha \frac{\partial \mathrm{Re}LU(\ )}{\partial w_1}$$

$$w_1(new) = -1.0 - 0.01 \times x_1$$

$$w_1(new) = -1.0 - 0.01 \times (-2) = -0.98$$

Backpropagation

The Parameter Update Rule

- For our example:

```
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias
```

- If α = 0.01, then

$$w_2(new) = w_2(old) - \alpha \frac{\partial \mathrm{Re}LU(\ )}{\partial w_2}$$

$$w_2(new) = 2.0 - 0.01 \times x_2$$

$$w_2(new) = 2.0 - 0.01 \times 3 = 1.97$$

Backpropagation

The Parameter Update Rule

- For our example:

```
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias
```

- If $\alpha = 0.01$, then

$$b(new) = b(old) - \alpha \frac{\partial \mathrm{Re}LU(\ )}{\partial b}$$

$$b(new) = 1.0 - 0.01 \times 1.0$$

$$b(new) = 1.0 - 0.01 \times 1.0 = 0.99$$

Backpropagation

The Parameter Update Rule

- For our example:

```
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias
```

- If $\alpha = 0.01$, then

$$w_0(new) = -3.01$$

$$w_1(new) = -0.98$$

$$w_2(new) = 1.97$$

$$b(new) = 0.99$$

Backpropagation

The Parameter Update Rule

- For our example:

```
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias
```

- The new output

$$w_0(new) = -3.01,\ w_1(new) = -0.98,\ w_2(new) = 1.97$$

$$b(new) = 0.99$$

$$out(new) = \mathrm{Re}LU(-3.01 \times 1.0 - 0.98 \times -2.0 + 1.97 \times 3.0 + 0.99)$$

$$= 5.85$$

Logic Gates Implementation

Training Neural Network for "OR Gate"

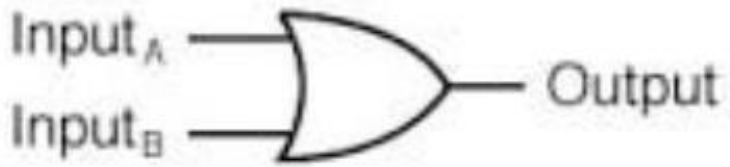
- Step 1: The Truth Table
- Step 2: Initialize Weights and Bias
- Step 3: Define the Activation Function
- Step 4: Forward Pass
- Step 5: Compute Loss
- Step 6: Compute the Gradient/Backward Pass
- Step 7: Update Weights and Bias
- Step 8: Repeat the Process

Logic Gates Implementation

Training Neural Network for "OR Gate"

- Step 1: The Truth Table/Input & Outputs

2 - input OR gate



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

Source: Google Images

Logic Gates Implementation

Training Neural Network for "OR Gate"

- Step 1: The Truth Table/Input & Outputs

```
import numpy as np

# OR gate inputs and outputs

inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
outputs = np.array([[0], [1], [1], [1]])
```

Logic Gates Implementation

Training Neural Network for "OR Gate"

- Step 2: Initialize Weights and Bias

```
# Initialize weights and bias

np.random.seed(43)
weights = np.random.rand(2, 1)
bias = np.random.rand(1)
learning_rate = 0.2
```

Logic Gates Implementation

Training Neural Network for "OR Gate"

- Step 3: Define the Activation Function

```
# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of the Sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)
```

Logic Gates Implementation

Training Neural Network for "OR Gate"

- Step 4: Forward Pass
- Step 5: Compute Loss
- Step 6: Compute the Gradient/Backward Pass
- Step 7: Update Weights and Bias
- Step 8: Repeat the Process

Model Training

Logic Gates Implementation

Training Neural Network for "OR Gate"

Model Training

```
# Training the model
for epoch in range(10000):
    # Forward pass
    z = np.dot(inputs, weights) + bias
    predictions = sigmoid(z)

    # Compute loss (Mean Squared Error)
    loss = (1/2) * (predictions - outputs) ** 2

    # Backpropagation
    d_loss = predictions - outputs
    d_pred = d_loss * sigmoid_derivative(predictions)

    # Gradient descent
    weights =weights - learning_rate * np.dot(inputs.T, d_pred)
    bias = bias - learning_rate * np.sum(d_pred)

    # Optionally print loss to see the progress
    if epoch % 1000 == 0:
        print('Epoch', epoch)
        print("Loss:", np.mean(loss))
```

Logic Gates Implementation

Training Neural Network for "OR Gate"

```
# Testing the model
print("Weights after training:", weights)
print("Bias after training:", bias)

# Final predictions
final_predictions = sigmoid(np.dot(inputs, weights) + bias)
print("Final predictions:", final_predictions.round())
```

Model Results

```
Weights after training: [[6.95515984]
 [6.95517589]]
Bias after training: [-3.23702087]
Final predictions: [[0.]
 [1.]
 [1.]
 [1.]]
```

Logic Gates Implementation

Training Neural Network for “AND Gate"

2 - input AND gate

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

Logic Gates Implementation

Training Neural Network for “AND Gate"

3 Input AND Gate Truth Table

Inputs			Outputs
A	B	C	X
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Logic Gates Implementation

Training NN for "3-Input AND Gate"

```
import numpy as np

# 3-Input AND gate inputs and outputs

inputs = np.array([[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1],
                   [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]])
outputs = np.array([[0], [0], [0], [0], [0], [0], [0], [1]])
```

Logic Gates Implementation

Training NN for “3-Input AND Gate"

```
# Initialize weights and bias

np.random.seed(43)
weights = np.random.rand(3, 1)
bias = np.random.rand(1)
learning_rate = 0.2
```

```
# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of the Sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)
```

Logic Gates Implementation

Training NN for “3-Input AND Gate"

```python
# Training the model
for epoch in range(10000):
    # Forward pass
    z = np.dot(inputs, weights) + bias
    predictions = sigmoid(z)

    # Compute loss (Mean Squared Error)
    loss = (1/2) * (predictions - outputs) ** 2

    # Backpropagation
    d_loss = predictions - outputs
    d_pred = d_loss * sigmoid_derivative(predictions)

    # Gradient descent
    weights =weights - learning_rate * np.dot(inputs.T, d_pred)
    bias = bias - learning_rate * np.sum(d_pred)

    # Optionally print loss to see the progress
    if epoch % 1000 == 0:
        print('Epoch', epoch)
        print("Loss:", np.mean(loss))
```

Logic Gates Implementation

Training NN for “3-Input AND Gate"

```
Epoch 0
Loss: 0.1875910245650087
Epoch 1000
Loss: 0.008572265054800333
Epoch 2000
Loss: 0.004200706242711879 5
Epoch 3000
Loss: 0.0026951472840618003
Epoch 4000
Loss: 0.0019588809680481903
Epoch 5000
Loss: 0.0015286926391372003
Epoch 6000
Loss: 0.001248755158471711
Epoch 7000
Loss: 0.001052968744502725
Epoch 8000
Loss: 0.0009087847292562013
Epoch 9000
Loss: 0.0007984096086282896
```

Logic Gates Implementation

Training NN for "3-Input AND Gate"

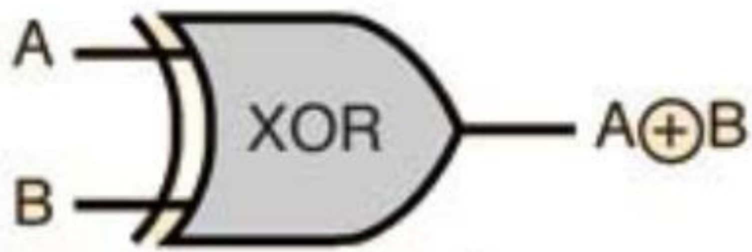
```
# Testing the model
print("Weights after training:", weights)
print("Bias after training:", bias)

# Final predictions
final_predictions = sigmoid(np.dot(inputs, weights) + bias)
print("Final predictions:", final_predictions.round())
```

```
Weights after training: [[5.60522228]
 [5.60522228]
 [5.60522228]]
Bias after training: [-14.23936938]
Final predictions: [[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [1.]]
```

Logic Gates Implementation

Training NN for “XOR Gate"



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Fig: Two Input XOR Gate

Source: Google Images

Logic Gates Implementation

Training NN for “XOR Gate"

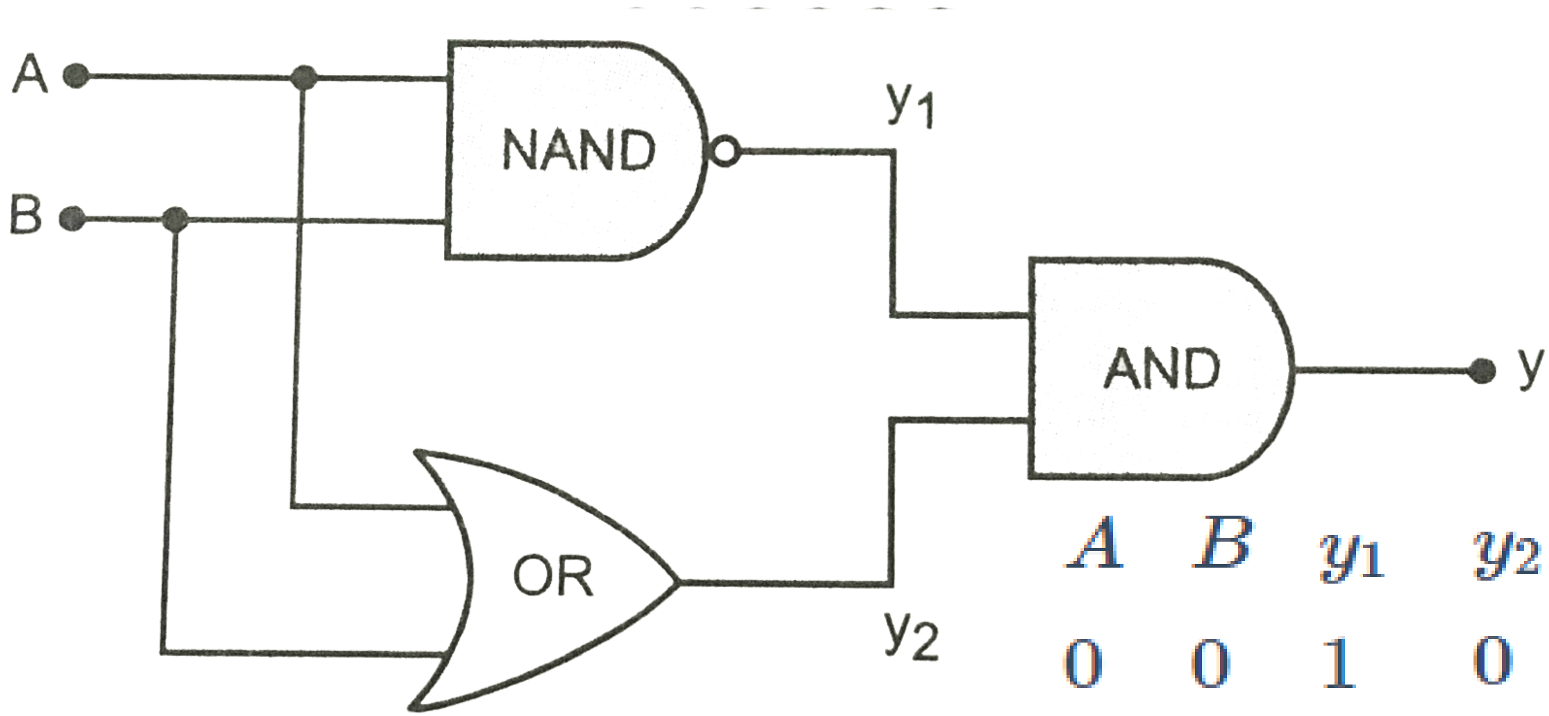


Fig: Two Input XOR Gate using NAND, OR and AND gates

A	B	y_1	y_2	y
0	0	1	0	0
0	1	1	1	1
1	0	1	1	1
1	1	0	1	0

Source: Google Images

Logic Gates Implementation

Training NN for “XOR Gate"

```
import numpy as np

# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of the Sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)

# XOR gate inputs and outputs
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
outputs = np.array([[0], [1], [1], [0]])
```

Logic Gates Implementation

Training NN for “XOR Gate"

```
# Initialize weights and biases
np.random.seed(42)
input_layer_neurons = 2
hidden_layer_neurons = 2
output_neuron = 1

# Weights and biases for layers
weights_input_hidden = np.random.rand(input_layer_neurons, hidden_layer_neurons)
bias_hidden = np.random.rand(1, hidden_layer_neurons)
weights_hidden_output = np.random.rand(hidden_layer_neurons, output_neuron)
bias_output = np.random.rand(1, output_neuron)

learning_rate = 0.1
```

Logic Gates Implementation

Training NN for “XOR Gate"

```
# Training the model
for epoch in range(10000):
    # Forward pass
    hidden_layer_activation = np.dot(inputs, weights_input_hidden) + bias_hidden
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
    predicted_output = sigmoid(output_layer_activation)

    # Compute loss (Mean Squared Error)
    loss = (1/2) * (predicted_output - outputs) ** 2

    # Backpropagation
    error = predicted_output - outputs
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer = d_predicted_output.dot(weights_hidden_output.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    # Update weights and biases
    weights_hidden_output -= learning_rate * hidden_layer_output.T.dot(d_predicted_output)
    bias_output -= learning_rate * np.sum(d_predicted_output, axis=0, keepdims=True)

    weights_input_hidden -= learning_rate * inputs.T.dot(d_hidden_layer)
    bias_hidden -= learning_rate * np.sum(d_hidden_layer, axis=0, keepdims=True)
```

Logic Gates Implementation

Training NN for “XOR Gate"

```
    # Print loss to see the progress
    if epoch % 1000 == 0:
        print(f'Epoch {epoch} Loss: {np.mean(loss)}')

# Testing the model
print("Weights after training (Input to Hidden):", weights_input_hidden)
print("Bias after training (Hidden):", bias_hidden)
print("Weights after training (Hidden to Output):", weights_hidden_output)
print("Bias after training (Output):", bias_output)

# Final predictions
final_predictions = sigmoid(np.dot(sigmoid(np.dot(inputs, weights_input_hidden)
                    + bias_hidden), weights_hidden_output) + bias_output)
print("Final predictions:", final_predictions.round())
```

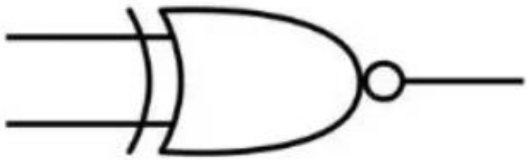
Logic Gates Implementation

Training NN for “XOR Gate"

```
Epoch 0 Loss: 0.162329290732212
Epoch 1000 Loss: 0.12029469015989666
Epoch 2000 Loss: 0.0980148391990633
Epoch 3000 Loss: 0.060331635667642 12
Epoch 4000 Loss: 0.015229506425732262
Epoch 5000 Loss: 0.006270561346238988
Epoch 6000 Loss: 0.0036842395233663574
Epoch 7000 Loss: 0.0025463194370964996
Epoch 8000 Loss: 0.0019234352833803127
Epoch 9000 Loss: 0.00153558770449285
Weights after training (Input to Hidden): [[3.79198478 5.81661184]
 [3.80004873 5.8545897 ]]
Bias after training (Hidden): [[-5.82020057 -2.46277158]]
Weights after training (Hidden to Output): [[-8.32186051]
 [ 7.66063503]]
Bias after training (Output): [[-3.45550373]]
Final predictions: [[0.]
 [1.]
 [1.]
 [0.]]
```

Logic Gates Implementation

Training NN for “XNOR Gate"



INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	1

Fig: Two Input XNOR Gate

Source: Google Images

Logic Gates Implementation

Training NN for “XNOR Gate"

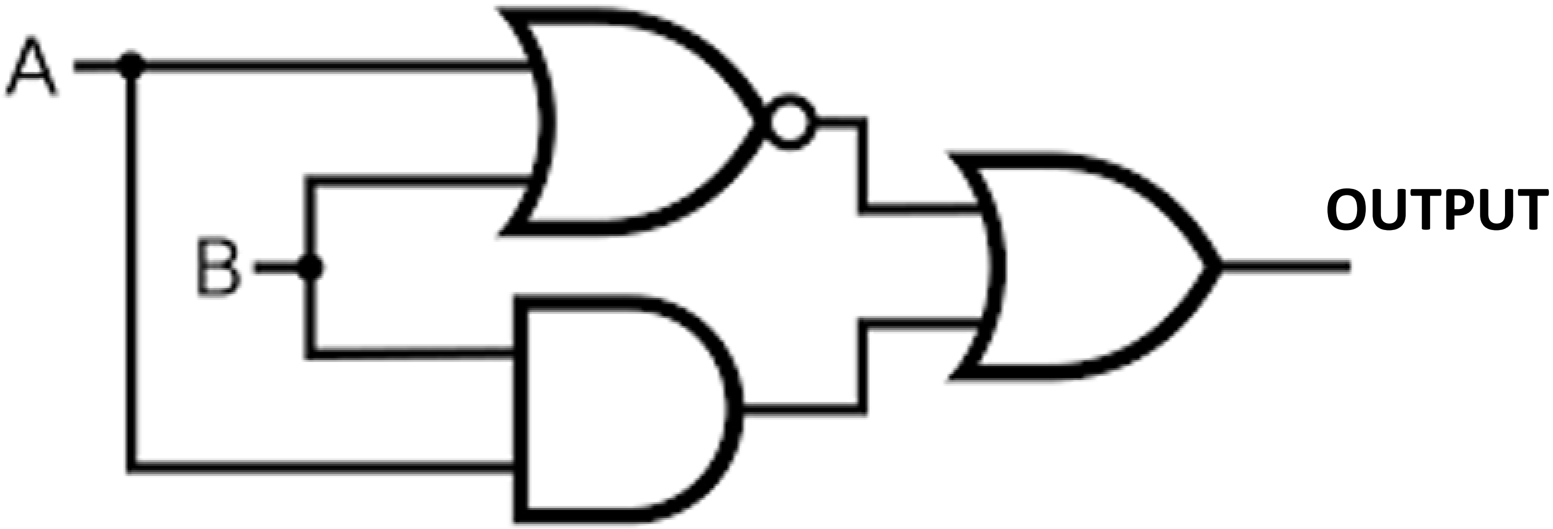


Fig: Two Input XNOR Gate using NOR, AND and OR gates

Source: Google Images