

# General Steps for MLP Learning (Backpropagation Algorithm)

## Step 1. Initialize the Network

- Decide the **structure** (number of input, hidden, and output neurons).
- Initialize all **weights** and **biases** randomly (usually small values).
- Choose an **activation function** (e.g., sigmoid, tanh, ReLU).
- Set a **learning rate** ( $\eta$ ) and number of **epochs**.

## Step 2. Forward Pass (Feedforward)

For each input sample (  $X$  ):

1. Compute the input to the first hidden layer:

$$Z^{(1)} = XW^{(1)} + b^{(1)}$$

2. Apply activation function:

$$A^{(1)} = f(Z^{(1)})$$

3. Repeat for next layers until the output layer:

$$Z^{(L)} = A^{(L-1)}W^{(L)} + b^{(L)}, \quad A^{(L)} = f(Z^{(L)})$$

4. The final output  $A^{(L)}$  is the **network prediction**.

## Step 3. Compute the Error (Loss)

Compare predicted output with the real output (  $y$  ):  $E = y - A^{(L)}$  Then compute the loss (e.g. Mean Squared Error):  $L = \frac{1}{n} \sum (y - A^{(L)})^2$

## Step 4. Backward Pass (Backpropagation)

Compute how much each neuron contributed to the error.

### 1. Output layer delta:

$$\delta^{(L)} = (y - A^{(L)}) \cdot f'(A^{(L)})$$

### 2. Hidden layer deltas (for all hidden layers):

$$\delta^{(l)} = (\delta^{(l+1)} W^{(l+1)T}) \cdot f'(A^{(l)})$$

(Propagate the error backward layer by layer.)

## Step 5. Update Weights and Biases

Use the deltas to adjust weights and biases:

$$W^{(l)} = W^{(l)} + \eta (A^{(l-1)T} \delta^{(l)})$$

$$b^{(l)} = b^{(l)} + \eta \sum \delta^{(l)}$$

## Step 6. Repeat

Repeat **Steps 2** → **5** for all samples and all epochs until:

- The loss becomes small, or
- The network's output stabilizes (converges).



## Summary in Plain Words

Step	Meaning
<b>1</b> Initialize	Pick network size, random weights, learning rate
<b>2</b> Forward pass	Compute outputs layer by layer

Step	Meaning
3 Compute loss	See how wrong the network was
4 Backpropagate	Send error backward to find what caused it
5 Update	Adjust weights in the direction that reduces error
6 Repeat	Keep learning until the network gets good results

## General MLP Learning Algorithm (Pseudocode)

Given:

$X$  → training inputs  
 $y$  → target outputs  
 $\eta$  → learning rate  
 $f$  → activation function (e.g., sigmoid, tanh, ReLU)  
 $f'$  → derivative of activation function  
 epochs → number of training iterations

-----  
 1. Initialize all weights  $W(1)$  and biases  $b(1)$  randomly for each layer  $l$   
 -----

FOR epoch = 1 to epochs:

-----  
 2. Forward Pass  
 -----  
 Set  $A(0) = X$  # input layer activations  
  
 FOR each layer  $l = 1$  to  $L$ : #  $L$  = total number of layers  
 $Z(l) = A(l-1) * W(l) + b(l)$  # weighted input  
 $A(l) = f(Z(l))$  # activation output  
  
 Output =  $A(L)$  # final prediction  
 -----

3. Compute Error  
 -----

```
Error = y - Output
Loss = MeanSquaredError(Error)    # or any chosen loss function
```

---

#### 4. Backward Pass (Backpropagation)

---

Compute delta for output layer:

$$\delta(L) = (y - A(L)) * f'(A(L))$$

FOR each hidden layer  $l = L-1$  down to 1:

$$\delta(l) = (\delta(l+1) * W(l+1)^T) * f'(A(l))$$

---

#### 5. Update Weights and Biases

---

FOR each layer  $l = 1$  to  $L$ :

$$W(l) = W(l) + \eta * (A(l-1)^T * \delta(l))$$

$$b(l) = b(l) + \eta * \text{sum}(\delta(l))$$

---

(optional) Print loss every few epochs

---

END FOR



## Summary of What Happens

Phase	Description
<b>Forward pass</b>	Compute activations layer by layer using current weights.
<b>Loss computation</b>	Measure how wrong the output is.
<b>Backward pass</b>	Compute error signals (deltas) layer by layer backward.
<b>Weight update</b>	Adjust weights and biases slightly to reduce future error.
<b>Repeat</b>	Continue until network learns (loss stops decreasing).



**You Can Apply This To:**

- Any number of **hidden layers**
- Any number of **neurons** per layer
- Any **activation function** that has a derivative
- Any **dataset**

## General MLP learning algorithm (from scratch using NumPy)

— works for *any* number of layers.

```
import numpy as np

# -----
# 1. Activation functions
# -----
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x) # assuming x is already sigmoid(x)

# -----
# 2. Initialize network
# -----
def initialize_network(layer_sizes):
    """layer_sizes = [n_input, n_hidden1, ..., n_output]"""
    weights = []
    biases = []
    for i in range(len(layer_sizes) - 1):
        w = np.random.randn(layer_sizes[i], layer_sizes[i + 1]) * 0.1
        b = np.zeros((1, layer_sizes[i + 1]))
        weights.append(w)
        biases.append(b)
    return weights, biases

# -----
# 3. Forward pass
```

```

# -----
def forward_pass(X, weights, biases):
    activations = [X]
    zs = []
    for w, b in zip(weights, biases):
        z = np.dot(activations[-1], w) + b
        a = sigmoid(z)
        zs.append(z)
        activations.append(a)
    return activations, zs

# -----
# 4. Backpropagation
# -----
def backward_pass(y, activations, weights):
    deltas = [None] * len(weights)
    # Output layer delta
    deltas[-1] = (y - activations[-1]) * sigmoid_derivative(activations[-1])

    # Hidden layers deltas (backward)
    for i in reversed(range(len(deltas) - 1)):
        deltas[i] = np.dot(deltas[i + 1], weights[i + 1].T) * sigmoid_derivative(activations[i + 1])
    return deltas

# -----
# 5. Update weights and biases
# -----
def update_parameters(weights, biases, activations, deltas, lr):
    for i in range(len(weights)):
        weights[i] += lr * np.dot(activations[i].T, deltas[i])
        biases[i] += lr * np.sum(deltas[i], axis=0, keepdims=True)
    return weights, biases

# -----
# 6. Train the network
# -----
def train(X, y, layer_sizes, lr=0.1, epochs=10000):
    weights, biases = initialize_network(layer_sizes)

    for epoch in range(epochs):
        activations, _ = forward_pass(X, weights, biases)
        deltas = backward_pass(y, activations, weights)
        weights, biases = update_parameters(weights, biases, activations, deltas, lr)

```

```

        if epoch % 1000 == 0:
            loss = np.mean((y - activations[-1]) ** 2)
            print(f"Epoch {epoch}, Loss = {loss:.4f}")

    return weights, biases





# -----
# Example: XOR problem
# -----
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

layer_sizes = [2, 3, 1] # input=2, hidden=3, output=1
weights, biases = train(X, y, layer_sizes, lr=0.5, epochs=10000)

# Test
outputs, _ = forward_pass(X, weights, biases)
print("\nPredictions:")
print(np.round(outputs[-1], 3))

```

## Key Features

 Flexible structure → just change `layer_sizes = [2, 4, 3, 1]` for deeper MLP   
 Uses **sigmoid** and its derivative  Implements **forward**, **backward**, and **update** steps  
 cleanly  Works for **XOR** or any small dataset

Q: How to modify this version to **support any activation function** (e.g., ReLU, tanh, softplus) by passing it as parameters?