Simple NN

🧠 Two-Layer Neural Network — Markdown Math (LaTeX inside `$$`)

1. Notation

$$
\begin{aligned}
X &\in \mathbb{R}^{m \times n} \quad \text{(Input data)} \\
y &\in \mathbb{R}^{m \times 1} \quad \text{(Binary labels)} \\
W_1 &\in \mathbb{R}^{n \times h} \quad \text{(Weights for hidden layer)} \\
b_1 &\in \mathbb{R}^{1 \times h} \quad \text{(Biases for hidden layer)} \\
W_2 &\in \mathbb{R}^{h \times 1} \quad \text{(Weights for output layer)} \\
b_2 &\in \mathbb{R}^{1 \times 1} \quad \text{(Bias for output layer)}
\end{aligned}
$$

2. Forward Pass

Hidden Layer:

$$Z^{[1]} = XW_1 + b_1$$

$$A^{[1]} = \mathrm{ReLU}(Z^{[1]})$$

Output Layer:

$$Z^{[2]} = A^{[1]}W_2 + b_2$$

$$\hat{y} = A^{[2]} = \sigma(Z^{[2]}) = \frac{1}{1 + e^{-Z^{[2]}}}$$

3. Loss Function (Binary Cross-Entropy)

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right]$$

4. Backward Pass

Output Layer:

$$dZ^{[2]} = \hat{y} - y$$

$$dW_2 = \frac{1}{m} (A^{[1]})^T dZ^{[2]}$$

$$db_2 = \frac{1}{m} \sum dZ^{[2]}$$

Hidden Layer:

$$dA^{[1]} = dZ^{[2]} W_2^T$$

$$dZ^{[1]} = dA^{[1]} \circ \mathrm{ReLU}'(Z^{[1]})$$

$$dW_1 = \frac{1}{m} X^T dZ^{[1]}$$

$$db_1 = \frac{1}{m} \sum dZ^{[1]}$$

5. Parameter Updates

$$W_1 \leftarrow W_1 - \alpha dW_1$$
$$b_1 \leftarrow b_1 - \alpha db_1$$
$$W_2 \leftarrow W_2 - \alpha dW_2$$
$$b_2 \leftarrow b_2 - \alpha db_2$$

```python
import numpy as np

class TwoLayerNN:
    def __init__(self, input_size, hidden_size, learning_rate=0.1):
        self.lr = learning_rate
        # Initialize weights and biases
        self.W1 = np.random.randn(input_size, hidden_size) * 0.01
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, 1) * 0.01
        self.b2 = np.zeros((1, 1))

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

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

    def relu_derivative(self, z):
        return (z > 0).astype(float)

    def forward(self, X):
        self.Z1 = np.dot(X, self.W1) + self.b1
        self.A1 = self.relu(self.Z1)
        self.Z2 = np.dot(self.A1, self.W2) + self.b2
        self.A2 = self.sigmoid(self.Z2)
        return self.A2

    def compute_loss(self, y, y_hat):
        m = y.shape[0]
        return -np.mean(y * np.log(y_hat + 1e-15) + (1 - y) * np.log(1 - y_hat + 1e-15))

    def backward(self, X, y, y_hat):
        m = X.shape[0]

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

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

```python
        # Update weights
        self.W1 -= self.lr * dW1
        self.b1 -= self.lr * db1
        self.W2 -= self.lr * dW2
        self.b2 -= self.lr * db2

    def fit(self, X, y, epochs=1000):
        for i in range(epochs):
            y_hat = self.forward(X)
            loss = self.compute_loss(y, y_hat)
            self.backward(X, y, y_hat)
            if i % 100 == 0:
                print(f"Epoch {i}, Loss: {loss:.4f}")

    def predict(self, X):
        y_hat = self.forward(X)
        return (y_hat >= 0.5).astype(int)
```