LAB REPORT: 2

Gradient Descent and Backpropagation with Different Activation Functions

Name: Aayush Suthar
Registration No.: 23FE10CAI00275
Course: Deep Learning Laboratory
Semester: VI | **Section:** F
Date: 15th January 2026

---

1. OBJECTIVE

To implement gradient descent and backpropagation algorithms with different activation functions (Sigmoid, Tanh, ReLU) for training neural networks.

---

2. THEORY

Gradient Descent

An optimization algorithm that minimizes loss by iteratively adjusting weights in the direction of steepest descent.

Update Rule: $W = W - \alpha \cdot \nabla W\ L$

Backpropagation

Efficient algorithm for computing gradients by applying the chain rule backward through the network.

Activation Functions

Function	Formula	Derivative	Range
Sigmoid	$\sigma(z) = 1/(1+e^{(-z)})$	$\sigma(z)\cdot(1-\sigma(z))$	(0, 1)
Tanh	$\tanh(z) = (e^z - e^{(-z)})/(e^z + e^{(-z)})$	$1 - \tanh^2(z)$	(-1, 1)
ReLU	$\max(0, z)$	1 if z>0 else 0	$[0, \infty)$

---

3. CODE IMPLEMENTATION

python

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

# XOR Dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Activation Functions
def sigmoid(z): return 1 / (1 + np.exp(-np.clip(z, -500, 500)))
def sigmoid_prime(z): s = sigmoid(z); return s * (1 - s)
def tanh(z): return np.tanh(z)
def tanh_prime(z): return 1 - np.tanh(z) ** 2
def relu(z): return np.maximum(0, z)
def relu_prime(z): return (z > 0).astype(float)

# Neural Network Class
class NeuralNetwork:
    def __init__(self, activation='sigmoid', lr=0.5):
        self.W1 = np.random.randn(2, 4) * 0.5
        self.b1 = np.zeros((1, 4))
        self.W2 = np.random.randn(4, 1) * 0.5
        self.b2 = np.zeros((1, 1))
        self.lr = lr
        self.losses = []

        # Select activation
        if activation == 'sigmoid':
            self.act, self.act_prime = sigmoid, sigmoid_prime
        elif activation == 'tanh':
            self.act, self.act_prime = tanh, tanh_prime
        else:  # relu
            self.act, self.act_prime = relu, relu_prime

    def forward(self, X):
        self.z1 = X @ self.W1 + self.b1
        self.a1 = self.act(self.z1)
        self.z2 = self.a1 @ self.W2 + self.b2
        self.a2 = sigmoid(self.z2)
        return self.a2

    def backward(self, X, y):
        m = X.shape[0]
        dz2 = self.a2 - y
        dW2 = (1/m) * self.a1.T @ dz2
```

```
        db2 = (1/m) * np.sum(dz2, axis=0, keepdims=True)

        da1 = dz2 @ self.W2.T
        dz1 = da1 * self.act_prime(self.z1)
        dW1 = (1/m) * X.T @ dz1
        db1 = (1/m) * np.sum(dz1, axis=0, keepdims=True)

        return dW1, db1, dW2, db2

    def train(self, X, y, epochs=10000):
        for epoch in range(epochs):
            y_pred = self.forward(X)
            loss = -np.mean(y * np.log(y_pred + 1e-15) +
                           (1 - y) * np.log(1 - y_pred + 1e-15))
            self.losses.append(loss)

            dW1, db1, dW2, db2 = self.backward(X, y)
            self.W1 -= self.lr * dW1
            self.b1 -= self.lr * db1
            self.W2 -= self.lr * dW2
            self.b2 -= self.lr * db2

            if (epoch + 1) % 2000 == 0:
                print(f"Epoch {epoch+1}, Loss: {loss:.4f}")

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

# Train with different activations
activations = ['sigmoid', 'tanh', 'relu']
models = {}

for act in activations:
    print(f"\n{'='*50}\nTraining with {act.upper()}\n{'='*50}")
    models[act] = NeuralNetwork(activation=act)
    models[act].train(X, y)

# Compare Results
print("\n" + "="*60)
print("PERFORMANCE COMPARISON")
print("="*60)

for act in activations:
    pred = models[act].predict(X)
    prob = models[act].forward(X)
    acc = np.mean(pred == y) * 100
    loss = models[act].losses[-1]
```

```python
    print(f"\n{act.upper()}: Loss={loss:.4f}, Accuracy={acc:.0f}%")
    print("Input\tTarget\tPred\tProb")
    for i in range(4):
        print(f"{X[i]}\t{y[i][0]}\t{pred[i][0]}\t{prob[i][0]:.3f}")

# Visualize
plt.figure(figsize=(10, 5))
for act in activations:
    plt.plot(models[act].losses, label=act.upper())
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Comparison')
plt.legend()
plt.grid(True)
plt.show()
```

Output:

```
==================================================
Training with SIGMOID
==================================================
Epoch 2000, Loss: 0.1089
Epoch 4000, Loss: 0.0345
Epoch 6000, Loss: 0.0178
Epoch 8000, Loss: 0.0115
Epoch 10000, Loss: 0.0082

==================================================
Training with TANH
==================================================
Epoch 2000, Loss: 0.0891
Epoch 4000, Loss: 0.0241
Epoch 6000, Loss: 0.0120
Epoch 8000, Loss: 0.0075
Epoch 10000, Loss: 0.0053

==================================================
Training with RELU
==================================================
Epoch 2000, Loss: 0.0654
Epoch 4000, Loss: 0.0198
Epoch 6000, Loss: 0.0101
Epoch 8000, Loss: 0.0065
Epoch 10000, Loss: 0.0046

============================================================
PERFORMANCE COMPARISON
============================================================

SIGMOID: Loss=0.0082, Accuracy=100%
Input  Target  Pred Prob
[0 0]  0  0  0.042
[0 1]  1  1  0.963
[1 0]  1  1  0.962
[1 1]  0  0  0.039

TANH: Loss=0.0053, Accuracy=100%
Input  Target  Pred Prob
[0 0]  0  0  0.031
[0 1]  1  1  0.971
[1 0]  1  1  0.971
[1 1]  0  0  0.030

RELU: Loss=0.0046, Accuracy=100%
```

```
Input  Target  Pred Prob
[0 0]  0  0  0.027
[0 1]  1  1  0.975
[1 0]  1  1  0.975
[1 1]  0  0  0.026
```

---

4. OBSERVATIONS

1. **Convergence Speed:**
    - ReLU: Fastest (Loss: 0.0046)
    - Tanh: Moderate (Loss: 0.0053)
    - Sigmoid: Slowest (Loss: 0.0082)
2. **Accuracy:** All achieved 100% on XOR problem
3. **Gradient Flow:**
    - ReLU: No vanishing gradient problem
    - Tanh: Better than Sigmoid, centered at 0
    - Sigmoid: Slower due to gradient saturation
4. **Decision Boundaries:** All create non-linear boundaries separating XOR classes

---

5. CONCLUSION

Successfully implemented gradient descent and backpropagation with three activation functions. ReLU showed fastest convergence and best final loss, while all functions solved the non-linear XOR problem with 100% accuracy. The choice of activation function significantly impacts training speed and gradient flow.