Assignment 01: Training a Simple Two-Layer Neural Network with NumPy

A. Practicing NumPy

A.1 Matrix Multiplication (Resource Allocation)

```
import numpy as np

Resources_Matrix = np.array([
    [100, 120],
    [80, 90],
    [60, 75]
])

Allocation_Factors = np.array([
    [1.1, 0.9],
    [0.95, 1.05]
])

Result = Resources_Matrix @ Allocation_Factors

print("A.1 Result:")
print(Result)
```

```
A.1 Result:
[[224.   216.  ]
 [173.5  166.5 ]
 [137.25 132.75]]
```

A.2 Element-wise Operations (Production Tracking)

```
Shift_A_Production = np.array([
    [10, 12, 11, 13, 15, 14, 16],
    [8, 9, 10, 11, 9, 10, 12],
    [7, 8, 9, 9, 10, 11, 12]
])

Shift_B_Production = np.array([
    [9, 10, 10, 12, 13, 12, 14],
    [7, 8, 9, 10, 8, 9, 10],
    [6, 7, 8, 8, 9, 9, 10]
])

Total_Production = Shift_A_Production + Shift_B_Production

print("A.2 Result: Total production")
print(Total_Production)
```

```
A.2 Result: Total production
[[19 22 21 25 28 26 30]
 [15 17 19 21 17 19 22]
 [13 15 17 17 19 20 22]]
```

A.3 Activation Function (Sigmoid in Sales Forecasting)

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

test_values = np.array([-2, -1, 0, 1, 2])
print("A.3 Result: Sigmoid test:")
print(sigmoid(test_values))
```

```
A.3 Result: Sigmoid test:
[0.11920292 0.26894142 0.5        0.73105858 0.88079708]
```

A.4 Gradient Calculation (Learning Adjustment)

```
def sigmoid_gradient(x):
    s = sigmoid(x)
    return s * (1 - s)
```

```
print("A.3 Result: Sigmoid Gradient test:")
print(sigmoid_gradient(test_values))
```

```
A.3 Result: Sigmoid Gradient test:
[0.10499359 0.19661193 0.25       0.19661193 0.10499359]
```

C. NumPy

Data preparation

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

X = np.array([
    [20, 15, 30, 25, 35],  # Units sold
    [3, 5, 2, 4, 2],       # Marketing
    [4, 3, 2, 1, 3]        # Satisfaction
])

Y = np.array([[18, 20, 22, 25, 30]])

print("X shape:", X.shape)
print("Y shape:", Y.shape)
```

```
X shape: (3, 5)
Y shape: (1, 5)
```

Sigmoid and ReLU

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

def sigmoid_derivative(z):
    s = sigmoid(z)
    return s * (1 - s)

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

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

Initialization of parameters

```
np.random.seed(1)

W1 = np.random.randn(3,3) * 0.01
b1 = np.zeros((3,1))

W2 = np.random.randn(1,3) * 0.01
b2 = np.zeros((1,1))

print("W1:\n", W1)
print("b1:\n", b1)
print("W2:\n", W2)
print("b2:\n", b2)
```

```
W1:
 [[ 0.01624345 -0.00611756 -0.00528172]
 [-0.01072969  0.00865408 -0.02301539]
 [ 0.01744812 -0.00761207  0.00319039]]
b1:
 [[0.]
 [0.]
 [0.]]
W2:
 [[-0.0024937   0.01462108 -0.02060141]]
b2:
 [[0.]]
```

Forward propagation

```
def forward_propagation(X, W1, b1, W2, b2):

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)

    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    cache = (Z1, A1, Z2, A2)
    return A2, cache
```

```
A2, cache = forward_propagation(X, W1, b1, W2, b2)
print("Initial predictions:")
print(A2)
```

```
Initial predictions:
[[0.4980767  0.49867582 0.49706008 0.49765919 0.49654701]]
```

Loss function

```
def compute_loss(A2, Y):
    return np.mean((A2 - Y)**2)

loss = compute_loss(A2, Y)
print("Initial loss:", loss)
```

```
Initial loss: 523.9627665874224
```

Backward propagation

```
def backward_propagation(X, Y, cache, W2):

    Z1, A1, Z2, A2 = cache
    m = X.shape[1]

    dA2 = 2 * (A2 - Y)
    dZ2 = dA2 * sigmoid_derivative(Z2)

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

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = dA1 * relu_derivative(Z1)

    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis=1, keepdims=True) / m

    return dW1, db1, dW2, db2
```

```
dW1, db1, dW2, db2 = backward_propagation(X, Y, cache, W2)

print("dW1:\n", dW1)
print("db1:\n", db1)
print("dW2:\n", dW2)
print("db2:\n", db2)
```

```
dW1:
 [[0.7313369  0.08728627 0.0709515 ]
 [0.         0.         0.        ]
 [6.04184407 0.72110411 0.5861565 ]]
db1:
 [[0.02805639]
 [0.        ]
 [0.23178416]]
dW2:
 [[-4.39936457  0.         -4.94139938]]
db2:
 [[-11.25088979]]
```

Gradient Descent

```
def update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, lr):

    W1 = W1 - lr * dW1
    b1 = b1 - lr * db1
    W2 = W2 - lr * dW2
    b2 = b2 - lr * db2
```

```
    return W1, b1, W2, b2
```

Network training

```
learning_rate = 0.001
epochs = 5000
losses = []

for i in range(epochs):

    # Forward
    A2, cache = forward_propagation(X, W1, b1, W2, b2)

    # Loss
    loss = compute_loss(A2, Y)
    losses.append(loss)

    # Backward
    dW1, db1, dW2, db2 = backward_propagation(X, Y, cache, W2)

    # Update
    W1, b1, W2, b2 = update_parameters(
        W1, b1, W2, b2,
        dW1, db1, dW2, db2,
        learning_rate
    )
```

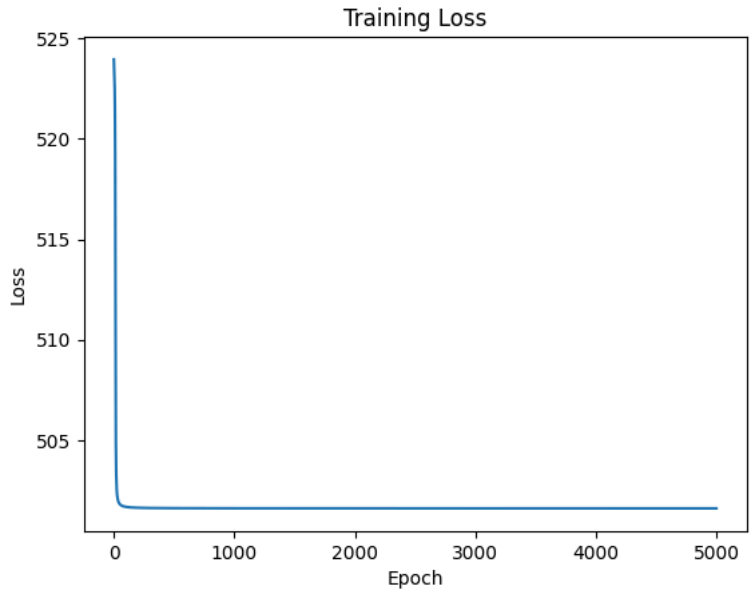
Final Predictions

```
A2, _ = forward_propagation(X, W1, b1, W2, b2)

print("Final W1:\n", W1)
print("\nFinal b1:\n", b1)
print("\nFinal W2:\n", W2)
print("\nFinal b2:\n", b2)

print("\nPredictions:")
print(A2)
```

```
Final W1:
 [[ 7.34670458e-01  1.40969247e-01  9.73968474e-02]
 [-1.07296862e-02  8.65407629e-03 -2.30153870e-02]
 [-2.25946277e-05 -9.69721158e-03  1.49546541e-03]]

Final b1:
 [[ 0.03614688]
 [ 0.        ]
 [-0.00067023]]

Final W2:
 [[ 0.75512589  0.01462108 -0.01015278]]

Final b2:
 [[0.21793038]]

Predictions:
[[0.99999357 0.99991036 0.99999997 0.99999955 1.        ]]
```

Error graph

```
plt.plot(losses)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```


Training Loss
525
520
515
510
505
Loss
0
1000
2000
3000
4000
5000
Epoch