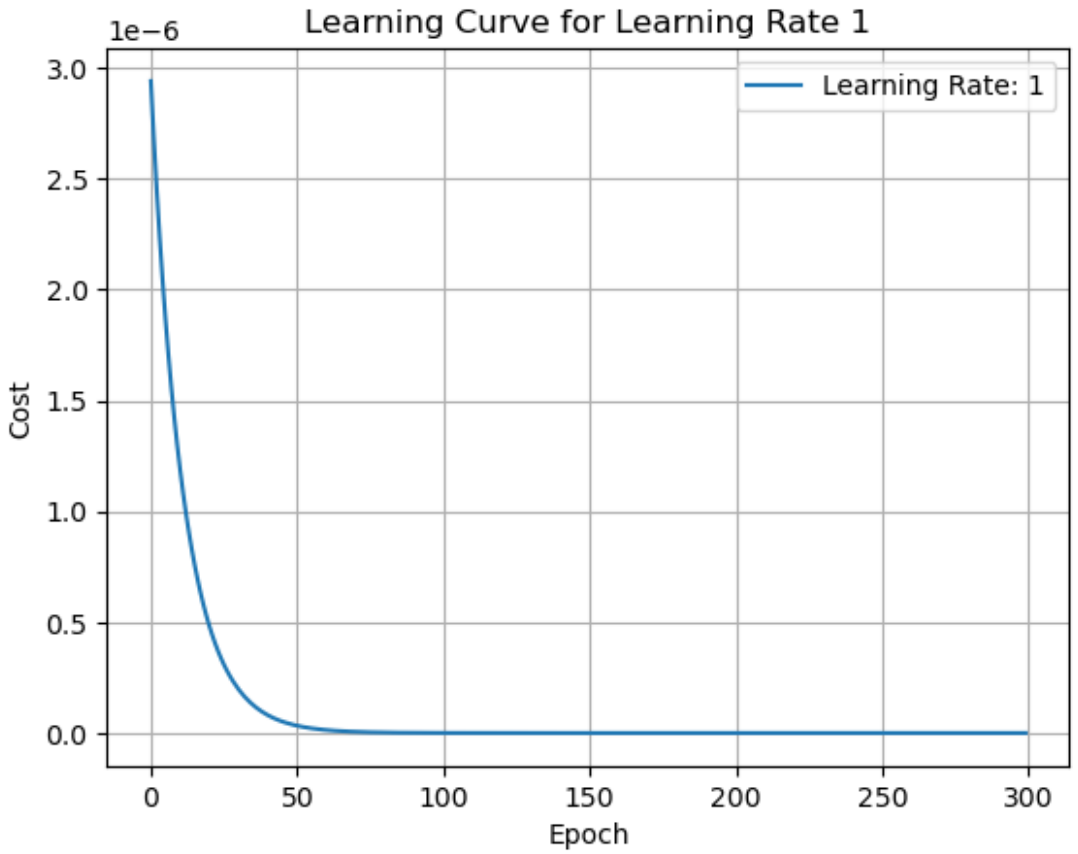
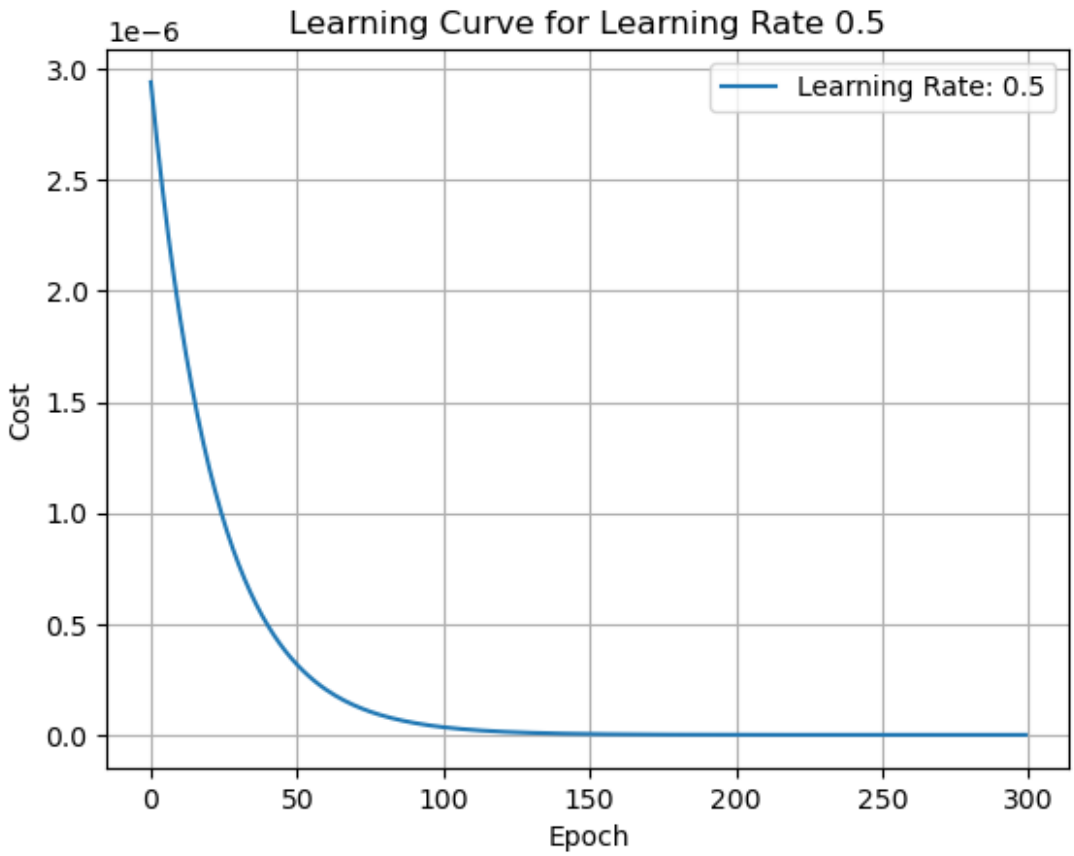
```python
# Import dependencies
import numpy as np
import matplotlib.pyplot as plt
```

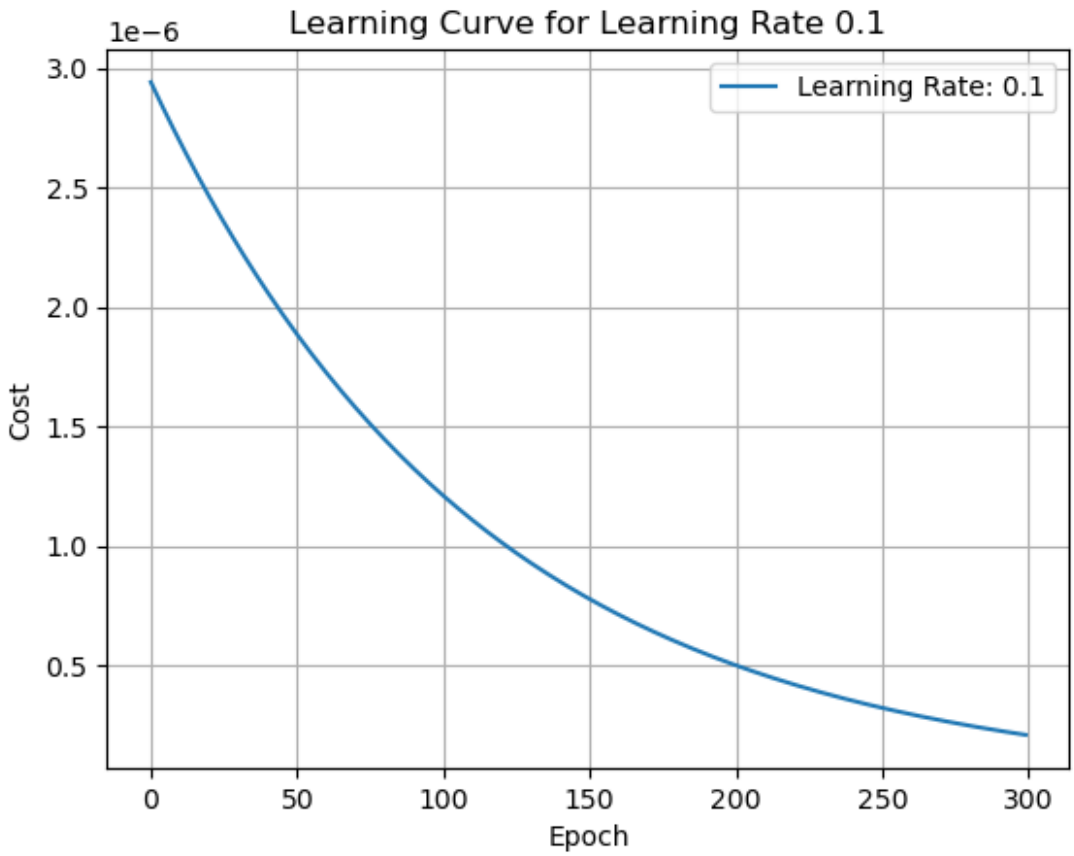
Problem 3

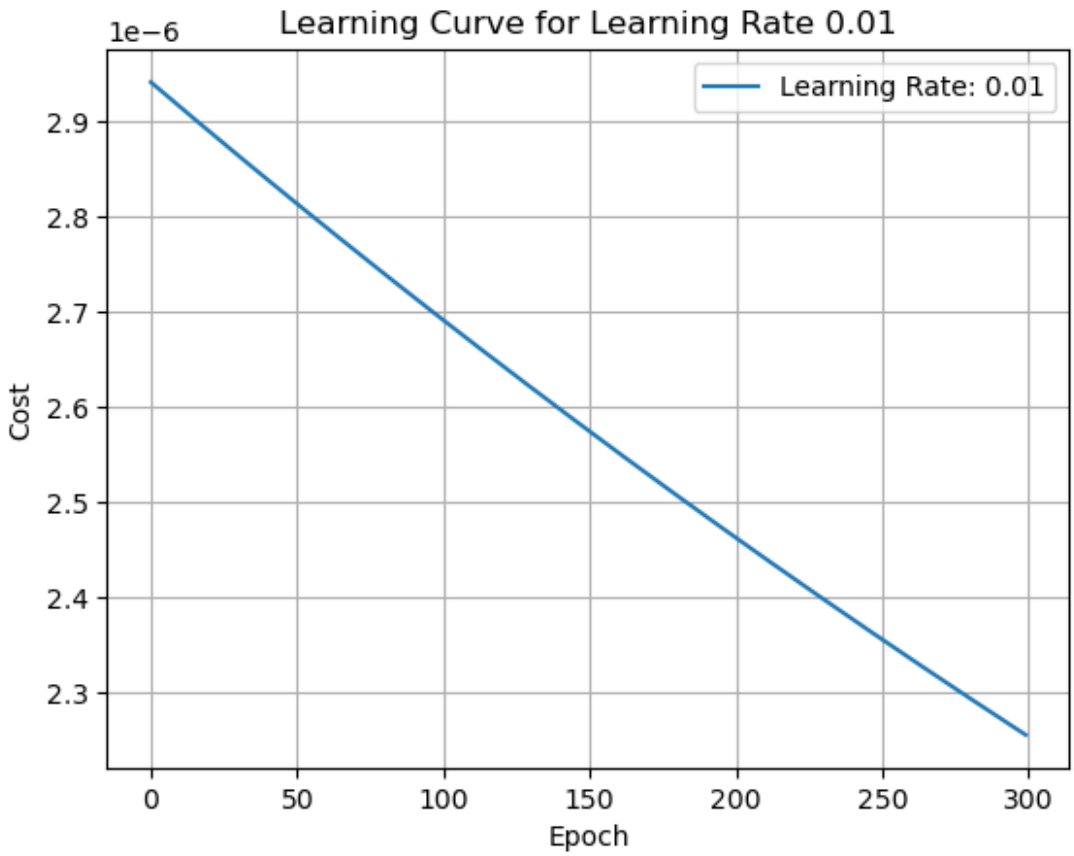
```python
# Define Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Derivative of sigmoid function
def sigmoid_derivative(z):
    return sigmoid(z) * (1 - sigmoid(z))

# Perform gradient descent algorithm
def gradient_descent(w, b, target, learning_rate, epochs):
    costs = []
    for epoch in range(epochs):
        # Forward pass
        z = w * 1 + b  # Input is 1
        output = sigmoid(z)

        # Calculate the cost (loss)
        cost = 0.5 * (output - target) ** 2
        costs.append(cost)

        # Backpropagation - compute gradients
        d_cost_d_output = output - target
        d_output_d_z = sigmoid_derivative(z)

        # Gradients for weight and bias
        d_cost_d_w = d_cost_d_output * d_output_d_z * 1  # Input x = 1
        d_cost_d_b = d_cost_d_output * d_output_d_z

        # Update weight and bias
        w -= learning_rate * d_cost_d_w
        b -= learning_rate * d_cost_d_b

    return w, b, costs

# Parameters for Case 1
w1, b1 = 0.60, 0.90  # Initial weight and bias for case 1
target1 = 0.82       # Target output for case 1
epochs = 300         # Number of epochs

# Define a list of learning rates to try
learning_rates = [1, 0.5, 0.1, 0.01, 0.001]

# Loop through each learning rate
for lr in learning_rates:
```

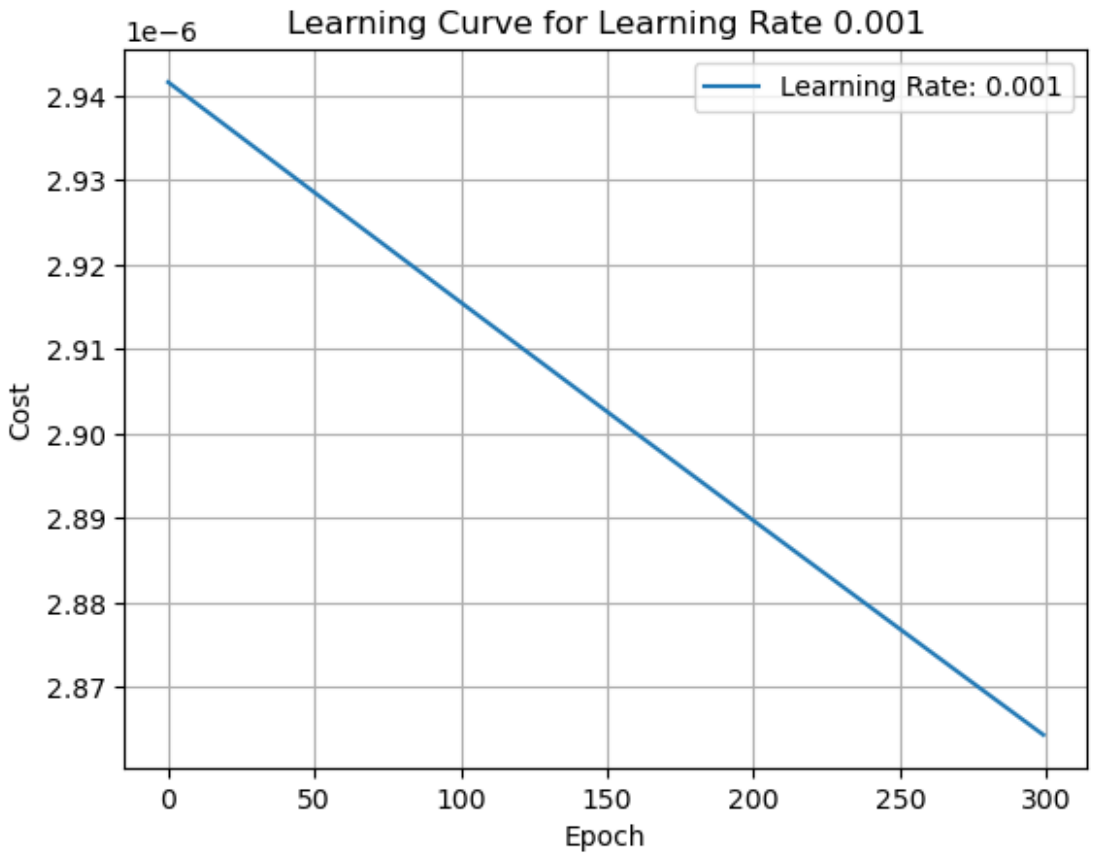
```
    # Re-initialize weight and bias for each learning rate
    w1, b1 = 0.60, 0.90  # Reset initial weight and bias for each run

    # Run gradient descent for each learning rate
    w1, b1, costs1 = gradient_descent(w1, b1, target1, lr, epochs)

    # Plot the learning curve for each learning rate
    plt.plot(costs1, label=f'Learning Rate: {lr}')
    plt.xlabel('Epoch')
    plt.ylabel('Cost')
    plt.title(f'Learning Curve for Learning Rate {lr}')
    plt.legend()
    plt.grid(True)
    plt.show()
```




Learning Curve for Learning Rate 0.5
1e−6
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Cost
0
50
100
150
200
250
300
Epoch
Learning Rate: 0.5

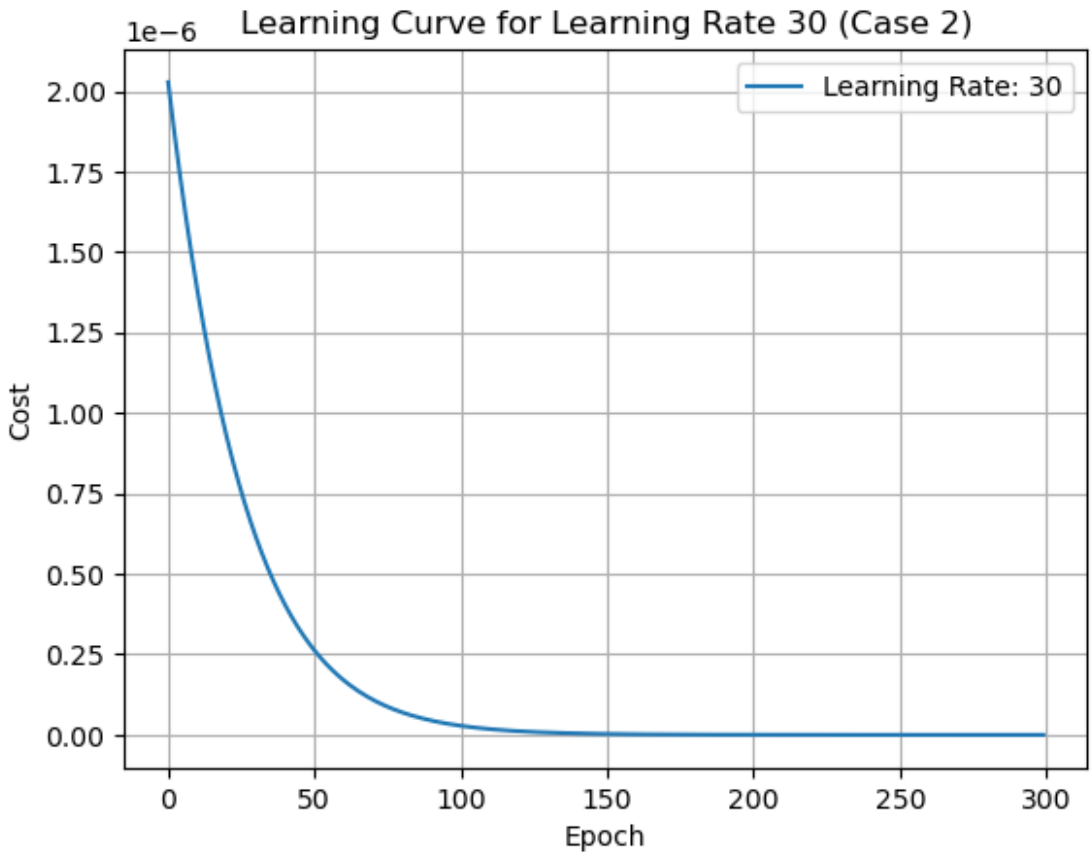

Learning Curve for Learning Rate 0.1
1e−6
3.0
2.5
2.0
1.5
1.0
0.5
Cost
0
50
100
150
200
250
300
Epoch
Learning Rate: 0.1

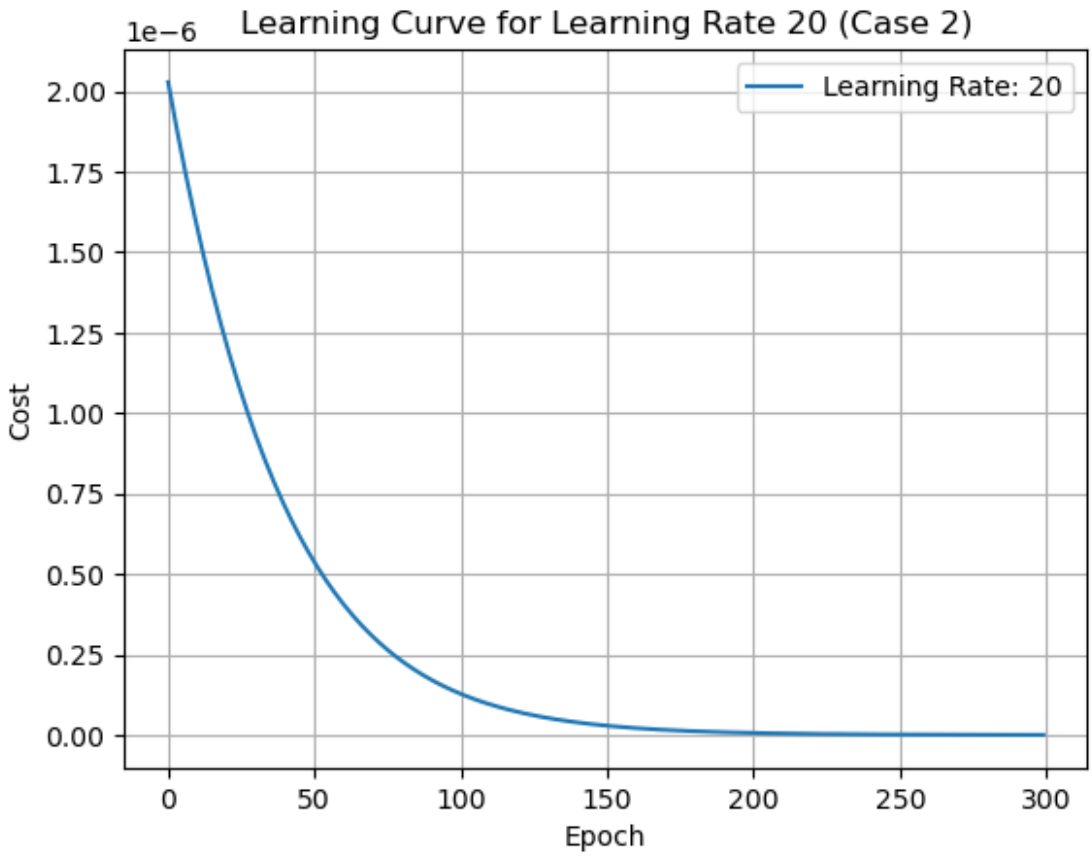

Learning Curve for Learning Rate 0.01
1e−6
Learning Rate: 0.01
2.9
2.8
2.7
2.6
2.5
2.4
2.3
Cost
0
50
100
150
200
250
300
Epoch

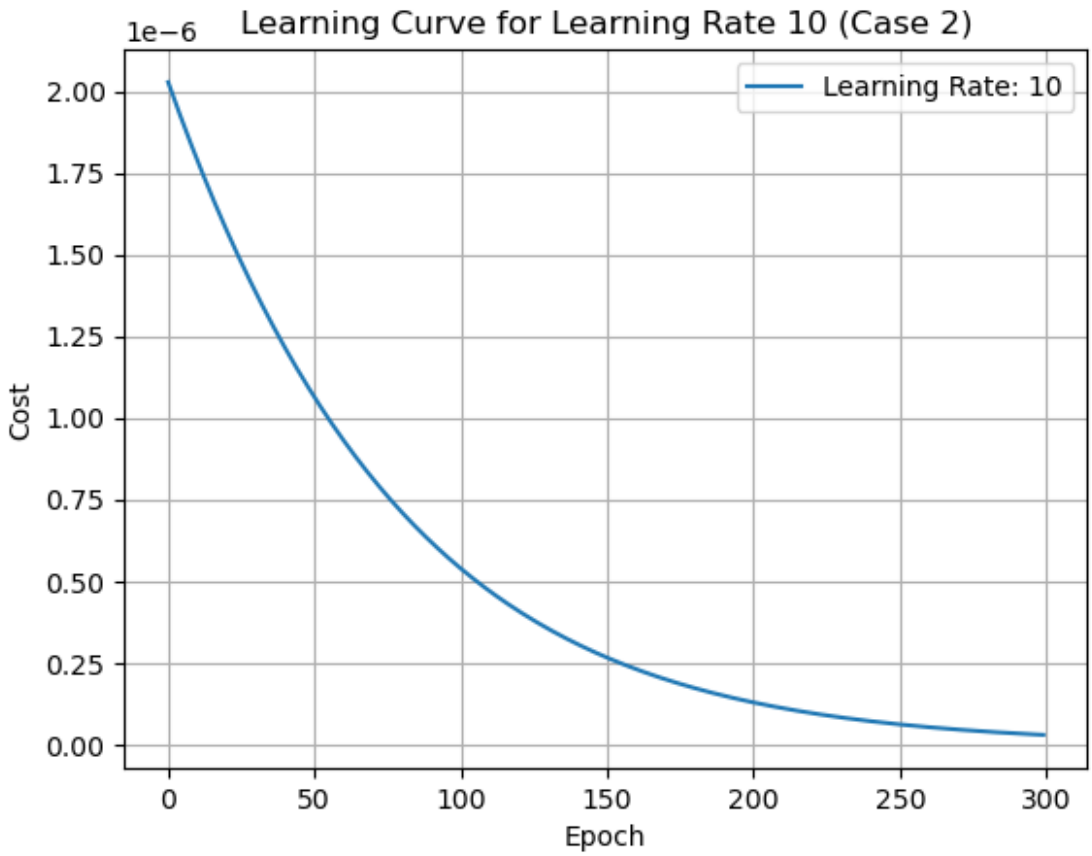


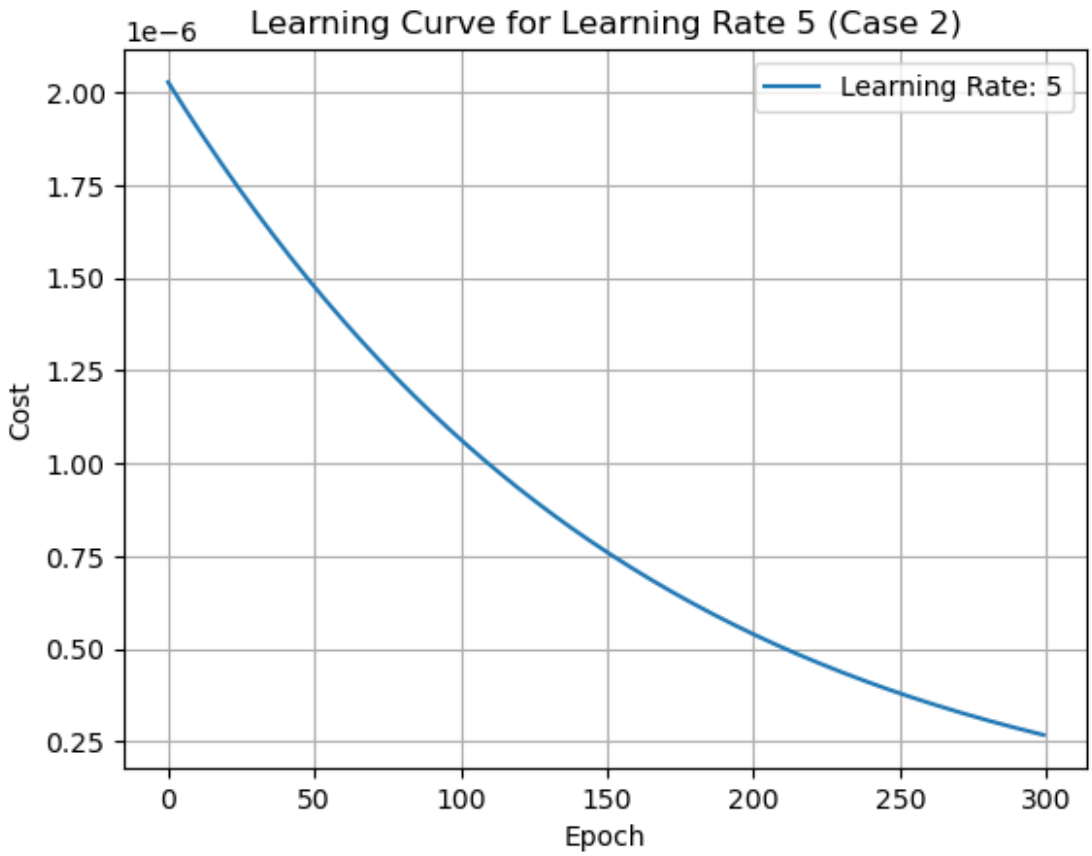

```
# Define Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Derivative of sigmoid function
def sigmoid_derivative(z):
    return sigmoid(z) * (1 - sigmoid(z))

# Perform gradient descent algorithm
def gradient_descent(w, b, target, learning_rate, epochs):
    costs = []
    for epoch in range(epochs):
        # Forward pass
        z = w * 1 + b  # Input is 1
        output = sigmoid(z)

        # Calculate the cost (loss)
        cost = 0.5 * (output - target) ** 2
        costs.append(cost)

        # Backpropagation - compute gradients
        d_cost_d_output = output - target
        d_output_d_z = sigmoid_derivative(z)
```

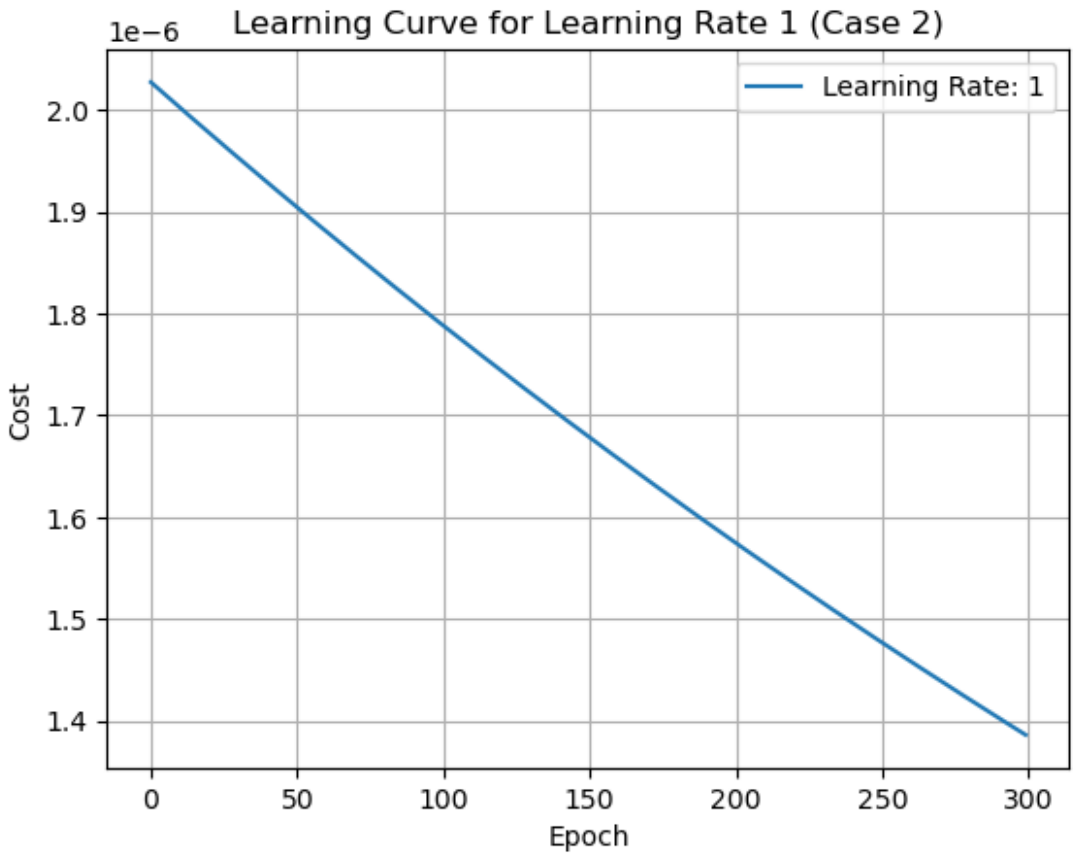
```
        # Gradients for weight and bias
        d_cost_d_w = d_cost_d_output * d_output_d_z * 1  # Input x = 1
        d_cost_d_b = d_cost_d_output * d_output_d_z

        # Update weight and bias
        w -= learning_rate * d_cost_d_w
        b -= learning_rate * d_cost_d_b

    return w, b, costs

# Parameters for Case 2
w2, b2 = 2.00, 2.00  # Initial weight and bias for case 2
target2 = 0.98       # Target output for case 2
epochs = 300         # Number of epochs

# Define a list of learning rates to try
learning_rates = [30, 20, 10, 5, 1]

# Loop through each learning rate
for lr in learning_rates:
    # Re-initialize weight and bias for each learning rate
    w2, b2 = 2.00, 2.00  # Reset initial weight and bias for each run

    # Run gradient descent for each learning rate
    w2, b2, costs2 = gradient_descent(w2, b2, target2, lr, epochs)

    # Plot the learning curve for each learning rate
    plt.plot(costs2, label=f'Learning Rate: {lr}')
    plt.xlabel('Epoch')
    plt.ylabel('Cost')
    plt.title(f'Learning Curve for Learning Rate {lr} (Case 2)')
    plt.legend()
    plt.grid(True)
    plt.show()
```


Learning Curve for Learning Rate 30 (Case 2)
1e−6
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Cost
0
50
100
150
200
250
300
Epoch
Learning Rate: 30


Learning Curve for Learning Rate 20 (Case 2)
1e−6
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Cost
0
50
100
150
200
250
300
Epoch
Learning Rate: 20

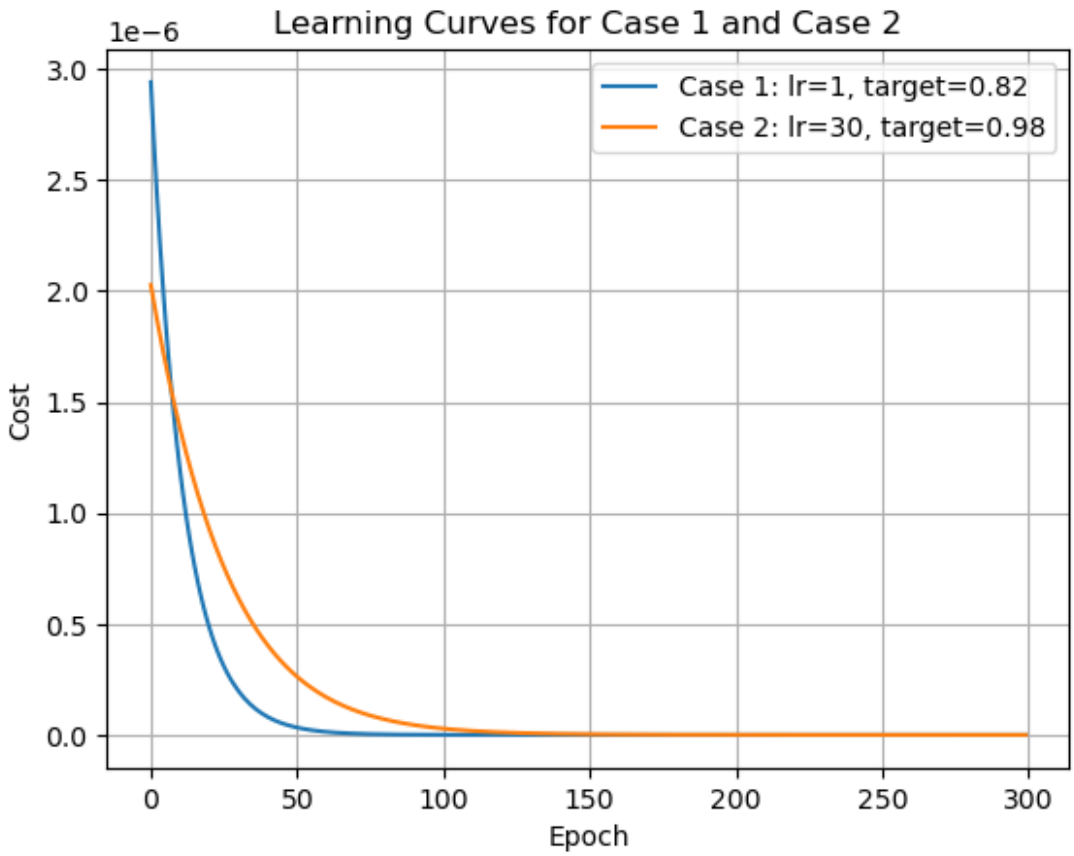

Learning Curve for Learning Rate 10 (Case 2)
1e−6
Learning Rate: 10
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Cost
0
50
100
150
200
250
300
Epoch


Learning Curve for Learning Rate 5 (Case 2)
1e−6
Learning Rate: 5
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
Cost
0
50
100
150
200
250
300
Epoch




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

# Derivative of sigmoid function
def sigmoid_derivative(z):
    return sigmoid(z) * (1 - sigmoid(z))

# Perform gradient descent algorithm
def gradient_descent(w, b, target, learning_rate, epochs):
    costs = []
    for epoch in range(epochs):
        # Forward pass
        z = w * 1 + b  # Input is 1
        output = sigmoid(z)

        # Calculate the cost (loss)
        cost = 0.5 * (output - target) ** 2
        costs.append(cost)

        # Backpropagation - compute gradients
        d_cost_d_output = output - target
        d_output_d_z = sigmoid_derivative(z)
```

```
        # Gradients for weight and bias
        d_cost_d_w = d_cost_d_output * d_output_d_z * 1  # Input x = 1
        d_cost_d_b = d_cost_d_output * d_output_d_z

        # Update weight and bias
        w -= learning_rate * d_cost_d_w
        b -= learning_rate * d_cost_d_b

    return w, b, costs

# Parameters for Case 1
w1, b1 = 0.60, 0.90  # Initial weight and bias for Case 1
target1 = 0.82       # Target output for Case 1
learning_rate1 = 1  # Learning rate for Case 1
epochs = 300         # Number of epochs

# Parameters for Case 2
w2, b2 = 2.00, 2.00  # Initial weight and bias for Case 2
target2 = 0.98       # Target output for Case 2
learning_rate2 = 30  # Learning rate for Case 2

# Run gradient descent for Case 1
w1, b1, costs1 = gradient_descent(w1, b1, target1, learning_rate1,
epochs)

# Run gradient descent for Case 2
w2, b2, costs2 = gradient_descent(w2, b2, target2, learning_rate2,
epochs)

# Plot the learning curves for both cases on the same plot
plt.plot(costs1, label=f'Case 1: lr={learning_rate1}, target=0.82')
plt.plot(costs2, label=f'Case 2: lr={learning_rate2}, target=0.98')
plt.xlabel('Epoch')
plt.ylabel('Cost')
plt.title('Learning Curves for Case 1 and Case 2')
plt.legend()
plt.grid(True)
plt.show()
```


Learning Curves for Case 1 and Case 2
1e−6
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Cost
0
50
100
150
200
250
300
Epoch
Case 1: lr=1, target=0.82
Case 2: lr=30, target=0.98