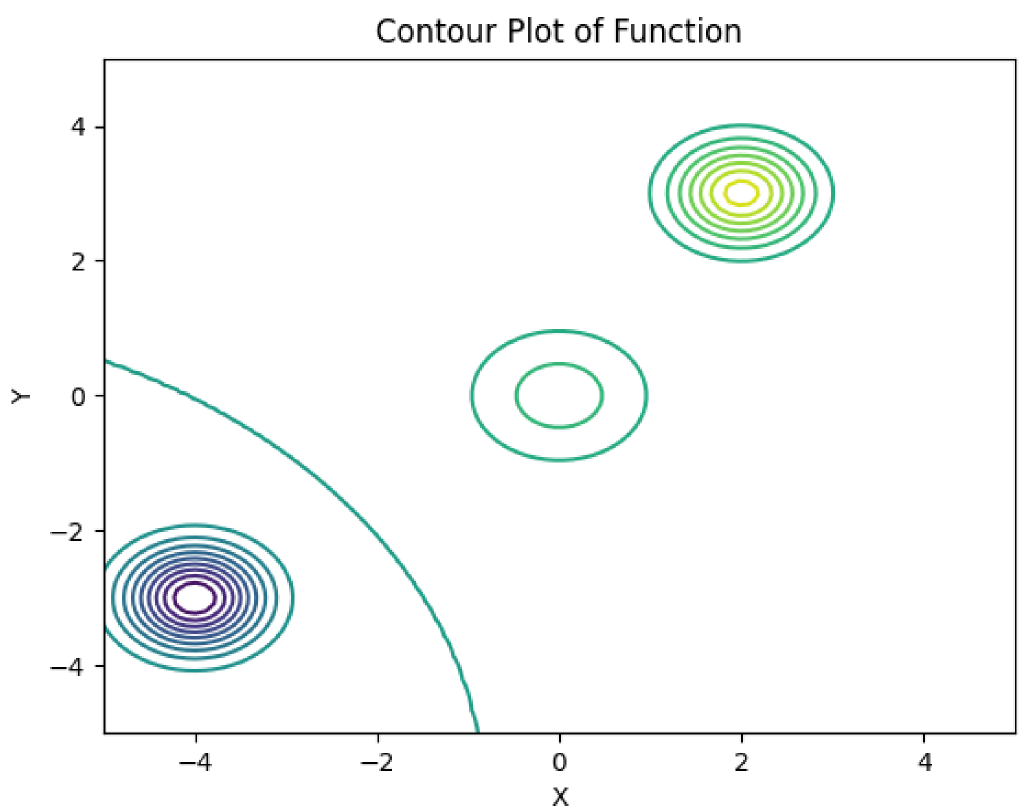
Deep Learning Homework 1

Muhammed Talha Karagül - 150120055

Question 1

First, I manually calculated the derivatives of and. Then, I implemented a function that computes gradient ascent. The gradient descent follows the same logic, with only a change in the sign of the equations.

Part a

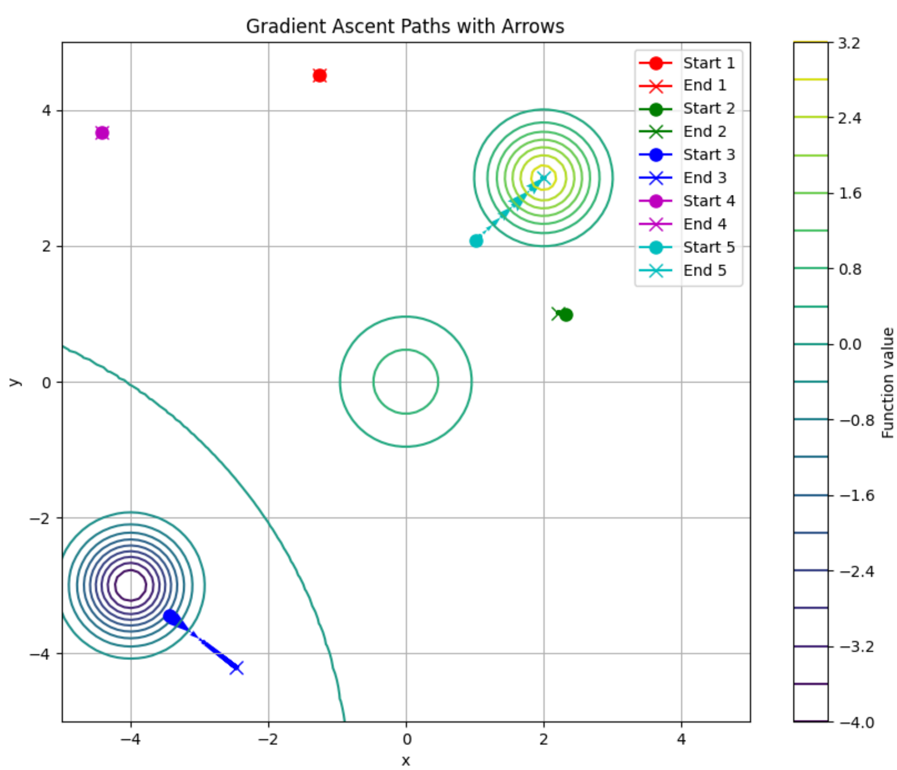


Part b

```
def gradient (x,y):
    df_dx = -2*x*np.exp(-(x**2+y**2)) - 3*2*2*(x-2)*np.exp(-2*((x-2)**2+(y-3)**2)) + 4*2*2*(x+4)*np.exp(-2*((x+4)**2+(y+3)**2))
    df_dy = -2*y*np.exp(-(x**2+y**2)) - 3*2*2*(y-3)*np.exp(-2*((x-2)**2+(y-3)**2)) + 4*2*2*(y+3)*np.exp(-2*((x+4)**2+(y+3)**2))

    return df_dx, df_dy
```

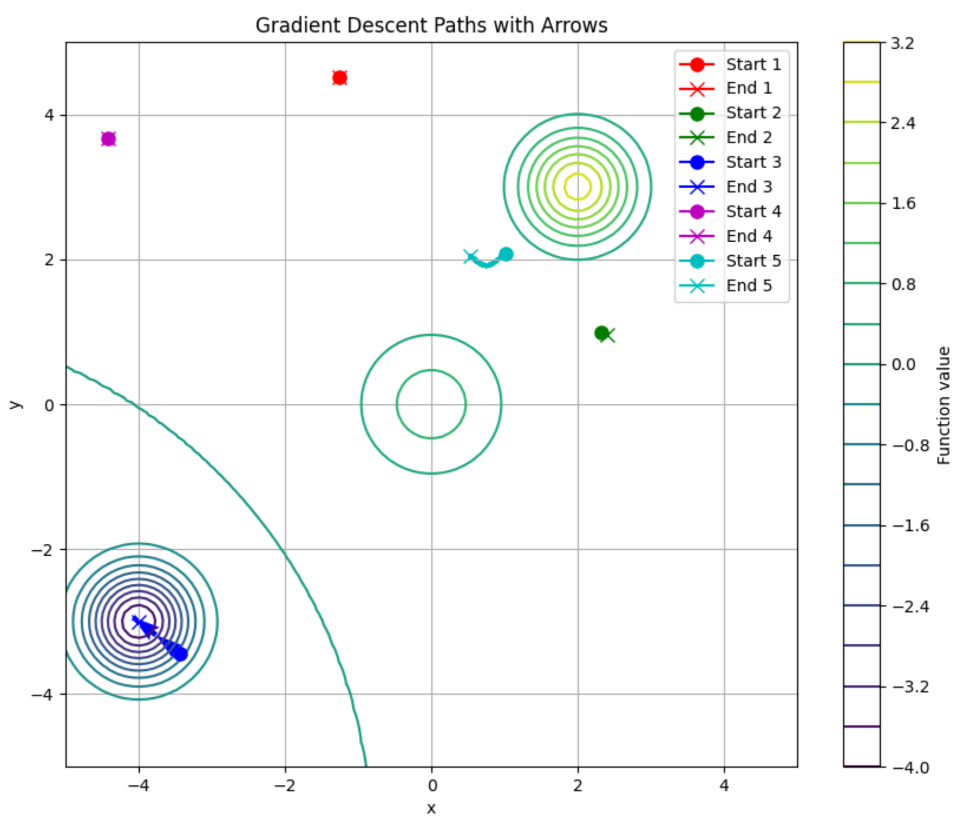
```python
def gradient_ascent(points, learning_rate=0.01, num_iterations=100):
    paths = []
    for point in points:
        x, y = point
        path = [(x, y)]
        for _ in range(num_iterations):
            dx, dy = gradient(x, y)
            x += learning_rate * dx
            y += learning_rate * dy
            path.append((x, y))
        paths.append(path)
    return paths
```

Part c



Part d

```
def gradient_descent(points, learning_rate=0.01, num_iterations=100):
    paths = []
    for point in points:
        x, y = point
        path = [(x, y)]
        for _ in range(num_iterations):
            dx, dy = gradient(x, y)
            x -= learning_rate * dx
            y -= learning_rate * dy
            path.append((x, y))
        paths.append(path)
    return paths
```



Question 2

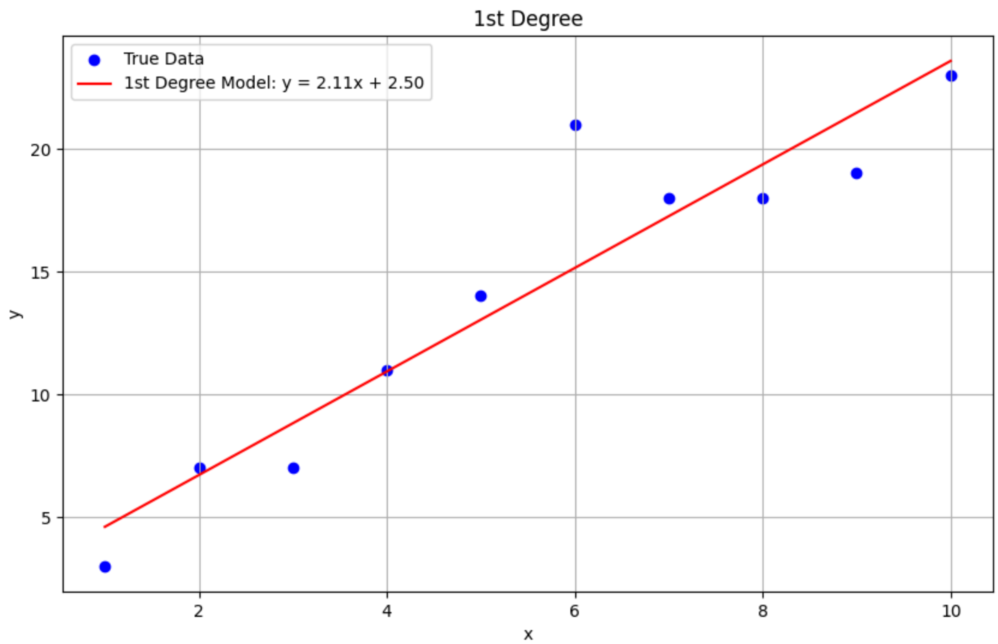
For Question 2, I determined different threshold values and learning rates for parts (a) and (b). This was necessary because there was an overflow issue in part (b) due to the tenth-order nature of the data. To prevent overflow, I decreased the learning rate.

The `poly_predict` function calculates the polynomial function based on the given weights, while the `poly_grad` function computes the gradients of those weights. In part (c), I implemented ridge regularization, as mentioned by the professor in the lecture. The only modification was made to the gradient function. Additionally, I introduced an early stopping condition to improve efficiency.

Part a

Final coefficients: a = 2.1077, b = 2.5008

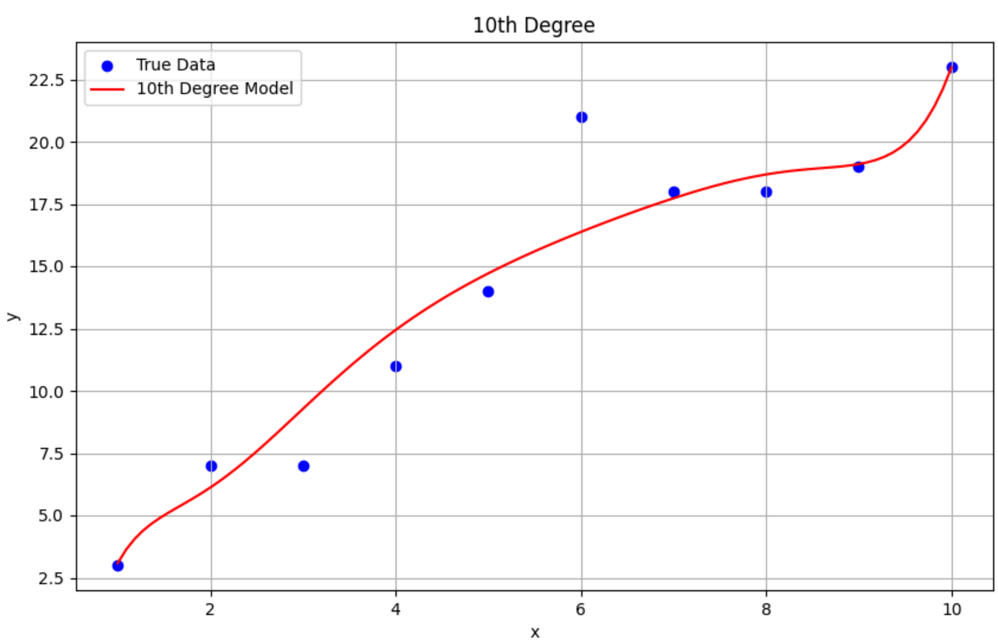
```python
##### Part A - Linear Regression #####

a, b = 0, 0
learning_rate = 0.01
epochs = 1000
threshold = 0.0001

for i in range(epochs):
    y_pred = a * x + b
    error = y_pred - y
    cost = np.mean(error ** 2)

    da = (2 / len(x)) * np.sum(error * x)
    db = (2 / len(x)) * np.sum(error)

    a -= learning_rate * da
    b -= learning_rate * db

    if abs(da) < threshold and abs(db) < threshold:
        print(f"Early stopping at epoch {i}")
        break

print(f"Final coefficients: a = {a:.4f}, b = {b:.4f}")
```



Part b

Epoch 0, Cost: 1.000000, Gradient Norm: 42.301538
Epoch 1000, Cost: 0.184331, Gradient Norm: 0.323765
Epoch 2000, Cost: 0.153692, Gradient Norm: 0.205885
Epoch 3000, Cost: 0.135430, Gradient Norm: 0.178933
Epoch 4000, Cost: 0.121040, Gradient Norm: 0.160921
Epoch 5000, Cost: 0.109270, Gradient Norm: 0.146234
Epoch 6000, Cost: 0.099491, Gradient Norm: 0.133664
Epoch 7000, Cost: 0.091288, Gradient Norm: 0.122649
Epoch 8000, Cost: 0.084360, Gradient Norm: 0.112881
Epoch 9000, Cost: 0.078477, Gradient Norm: 0.104166
10th degree final coefficients:
Weight 0: 0.2322
Weight 1: 0.7360
Weight 2: -0.2584
Weight 3: 0.1953
Weight 4: -0.1229
Weight 5: -0.0189
Weight 6: -0.0037
Weight 7: -0.1097
Weight 8: 0.0623
Weight 9: 0.0415
Weight 10: -0.0138

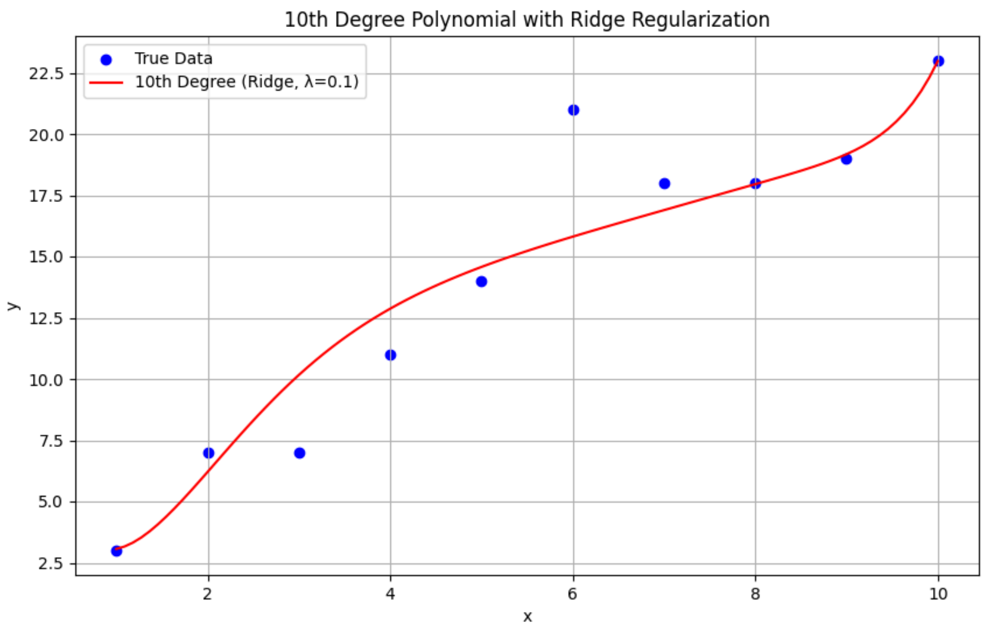
```python
degree = 10
learning_rate_2 = 0.0005
epochs = 10000
weights = np.zeros(degree+1)

x_mean = x.mean()
x_std = x.std()
y_mean = y.mean()
y_std = y.std() if y.std() != 0 else 1

x_norm = (x - x_mean) / x_std
y_norm = (y - y_mean) / y_std

def poly_predict(x, weights):
    return sum(w * (x ** i) for i, w in enumerate(weights))

def poly_grad(x, error, weights):
    grad = np.zeros_like(weights)
    for i in range(len(weights)):
        grad[i] = (2 / len(x)) * np.sum(error * (x ** i))
    return grad

for i in range(epochs):
    y_pred_norm = poly_predict(x_norm, weights)
    error = y_pred_norm - y_norm
    cost = np.mean(error ** 2)

    grad = poly_grad(x_norm, error, weights)
    weights -= learning_rate_2 * grad

    if i % 1000 == 0:
        print(f"Epoch {i}, Cost: {cost:.6f}, Gradient Norm: {np.linalg.norm(grad):.6f}")

    if np.any(np.isnan(weights)) or np.any(np.isnan(grad)):
        print("NaN detected! Stopping training.")
        break

    if np.all(np.abs(grad) < threshold):
        print(f"Early stopping at epoch {i}")
        break

print("10th degree final coefficients:")
for i in range(len(weights)):
    print(f"Weight {i}: {weights[i]:.4f}")

x_smooth = np.linspace(min(x), max(x), 100)
x_smooth_norm = (x_smooth - x_mean) / x_std
y_smooth_pred = poly_predict(x_smooth_norm, weights) * y_std + y_mean
```



Part c

Epoch 0, Cost: 1.000000, Gradient Norm: 42.301538
Epoch 1000, Cost: 0.198285, Gradient Norm: 0.292927
Epoch 2000, Cost: 0.175209, Gradient Norm: 0.168556
Epoch 3000, Cost: 0.164063, Gradient Norm: 0.132551
Epoch 4000, Cost: 0.156881, Gradient Norm: 0.107862
Epoch 5000, Cost: 0.152073, Gradient Norm: 0.088689
Epoch 6000, Cost: 0.148802, Gradient Norm: 0.073350
Epoch 7000, Cost: 0.146557, Gradient Norm: 0.060900
Epoch 8000, Cost: 0.145005, Gradient Norm: 0.050715
Epoch 9000, Cost: 0.143925, Gradient Norm: 0.042346
Regularized 10th degree final coefficients:
Weight 0: 0.1747
Weight 1: 0.5456
Weight 2: -0.1700
Weight 3: 0.1837
Weight 4: -0.0913
Weight 5: 0.0391
Weight 6: -0.0191
Weight 7: -0.0353
Weight 8: 0.0258
Weight 9: 0.0076
Weight 10: -0.0003

```python
##### Part C - 10th Degree Polynomial with Ridge Regularization #####

lambda_reg = 0.1  # Ridge regularization constant
epochs = 10000

weights = np.zeros(degree + 1)

def poly_grad_ridge(x, error, weights, lambda_reg):
    grad = np.zeros_like(weights)
    for i in range(len(weights)):
        grad[i] = (2 / len(x)) * np.sum(error * (x ** i)) + 2 * lambda_reg * weights[i]
    return grad

for i in range(epochs):
    y_pred_norm = poly_predict(x_norm, weights)
    error = y_pred_norm - y_norm
    cost = np.mean(error ** 2) + lambda_reg * np.sum(weights ** 2)

    grad = poly_grad_ridge(x_norm, error, weights, lambda_reg)
    weights -= learning_rate_2 * grad

    if i % 1000 == 0:
        print(f"Epoch {i}, Cost: {cost:.6f}, Gradient Norm: {np.linalg.norm(grad):.6f}")

    if np.all(np.abs(grad) < threshold):
        print(f"Early stopping at epoch {i}")
        break

print("Regularized 10th degree final coefficients:")
for i, w in enumerate(weights):
    print(f"Weight {i}: {w:.4f}")
```



Question 3

I used the sigmoid function as the activation function because the output values fit within its definition. The range between 40 and 80 is where the sigmoid function exhibits significant change, as there are data points in this range that influence the transition.

beta_0: -7.309693473622141
beta_1: 0.11686123974430668

```python
# Veri seti
X = np.array([10, 15, 20, 40, 50, 60, 60, 70, 80, 90, 95, 100, 100])
y = np.array([0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1])

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

# Gradient Descent ile parametreleri bulma
def logistic_regression(X, y, learning_rate=0.01, epochs=10000):
    # Başlangıç parametreleri
    beta_0 = 0.0
    beta_1 = 0.0
    n = len(X)

    # Gradient Descent
    for _ in range(epochs):
        # Tahmin
        z = beta_0 + beta_1 * X
        y_pred = sigmoid(z)

        # Gradient'ları hesapla
        error = y_pred - y
        grad_beta_0 = np.sum(error) / n
        grad_beta_1 = np.sum(error * X) / n

        # Parametreleri güncelle
        beta_0 -= learning_rate * grad_beta_0
        beta_1 -= learning_rate * grad_beta_1

    return beta_0, beta_1

# Modeli eğit
beta_0, beta_1 = logistic_regression(X, y)
print(f"beta_0: {beta_0}")
print(f"beta_1: {beta_1}")

# Modeli görselleştirme
X_plot = np.linspace(min(X)-10, max(X)+10, 100)
P = sigmoid(beta_0 + beta_1 * X_plot)
```