HW 9

Dylan Zhao dxz365

Exercise 1:

A.

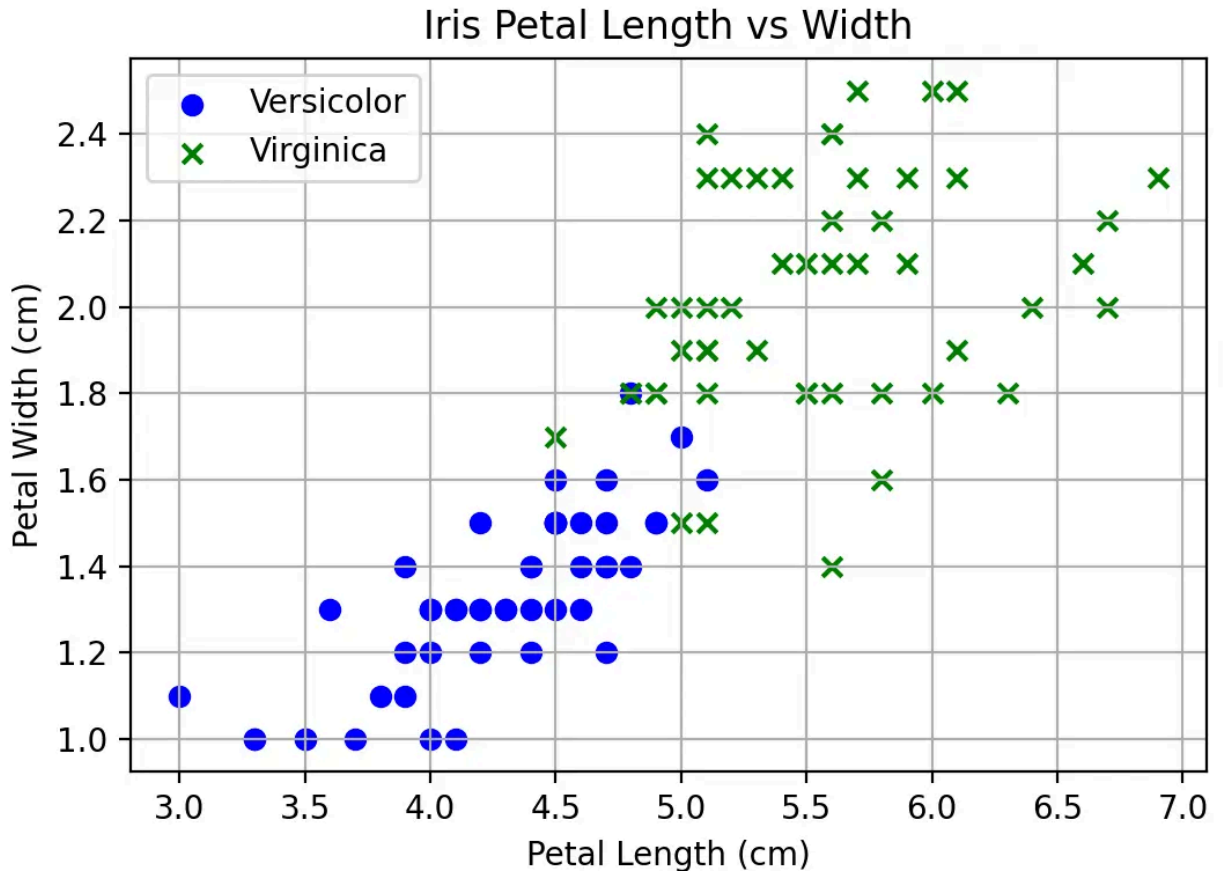
Below is the code used to load the irisdataset.csv and plot the petal width and length dimensions for the 2nd and 3rd classes.

```
# Function for plotting data
def plot_data(vr_petal_length, vr_petal_width, va_petal_length,
va_petal_width):

    plt.figure(figsize=(6, 4))

    plt.scatter(vr_petal_length, vr_petal_width, color='blue', marker='o',
label='Versicolor')
    plt.scatter(va_petal_length, va_petal_width, color='green', marker='x',
label='Virginica')


    plt.title('Iris Petal Length vs Width')
    plt.xlabel('Petal Length (cm)')
    plt.ylabel('Petal Width (cm)')
    plt.grid(True)
    plt.legend()
    plt.show()

# Load data
data = pd.read_csv(os.path.join(os.getcwd(), 'irisdata.csv'))

data = data[data["species"].isin(["versicolor", "virginica"])]

# Separate data by species
vr_petal_length = data[data["species"] == "versicolor"
['petal_length'].to_numpy()
vr_petal_width = data[data["species"] == "versicolor"]
['petal_width'].to_numpy()

# Plot initial data
```

```
va_petal_length = data[data["species"] == "virginica"]
['petal_length'].to_numpy()
va_petal_width = data[data["species"] == "virginica"]
['petal_width'].to_numpy()
```

The data was loaded into a Dataframe and split into four numpy arrays for the two difference species classes and their corresponding length and width measurements.

Below is the resulting plot:



B.

Below is the code used to implement the simple one-layer neural network using sigmoid linearity:

```
# Neural network functions

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

# Sigmoid activation function
```

```
def neural_network(x, W):
    z = np.dot(x, W)
    return sigmoid(z)
```

The sigmoid function calculates:

$$\sigma(z) = \frac{1}{(1 + e^{-z})}$$

while the neural network calculates the linearity function z:

$$z = \sum_{i=0}^{N} w_0 + w_1 x_{1,i} + w_2 x_{2,i}$$

via dot product. This z gets passed into the sigmoid function to return the final probability estimate.

The x variable stores all the data points in matrix form with all values in column 1 equaling to 1, all values in column 2 containing petal lengths, and all values in column 3 containing petal widths.

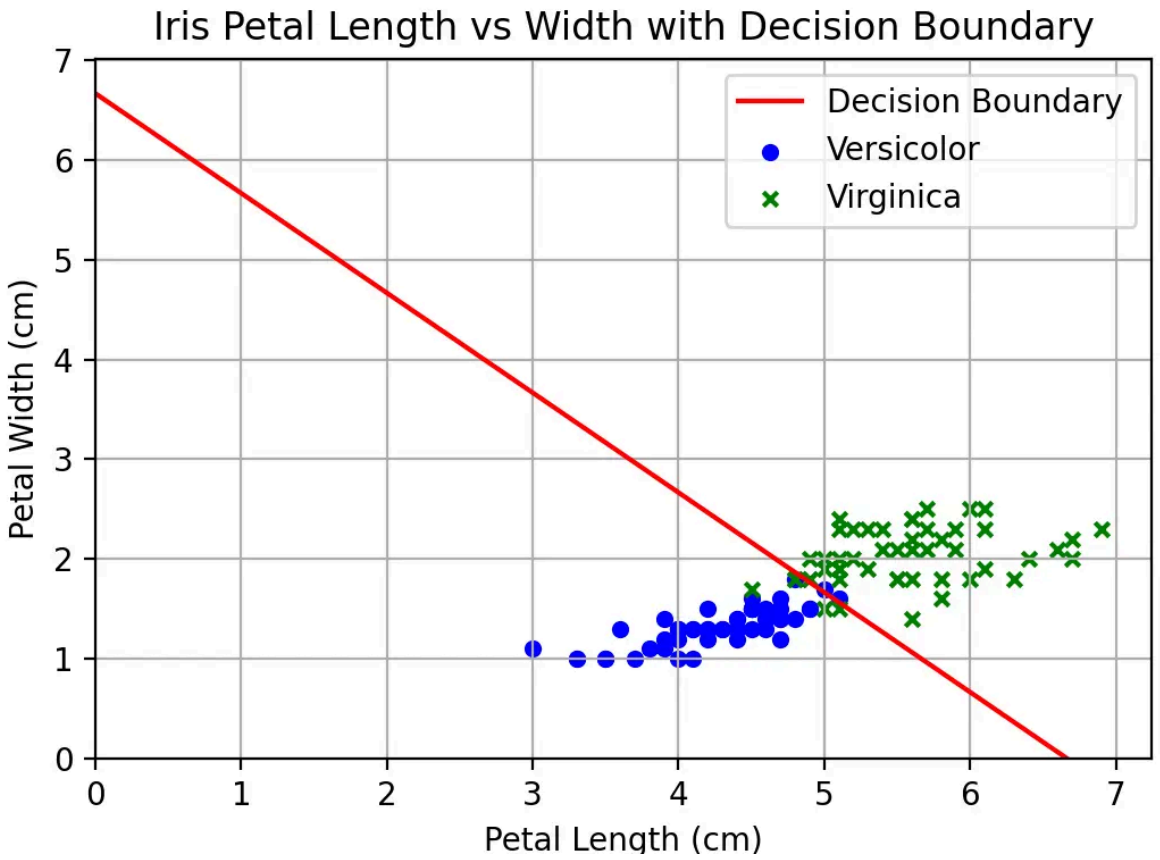
C.

Below is the code used to plot the decision boundary:

```
# Calculate decision boundary points
def calculate_decision_boundary(vr_petal_length, va_petal_length, W):
    x_pl = np.linspace(0, max(vr_petal_length.max(), va_petal_length.max()),
100)


    x_pw = -(W[0] + W[1] * x_pl) / W[2]
    return x_pl, x_pw

# Plot decision boundary
def plot_decision_boundary(vr_petal_length, vr_petal_width, va_petal_length,
va_petal_width, W, filepath = None):

    # Calculate decision boundary points
    x_pl, x_pw = calculate_decision_boundary(vr_petal_length,
va_petal_length, W)

    plt.figure(figsize=(6, 4))
```

```
    plt.plot(x_pl, x_pw, color='red', label='Decision Boundary')


    plt.scatter(vr_petal_length, vr_petal_width, color='blue', marker='o',
s=20, label='Versicolor')
    plt.scatter(va_petal_length, va_petal_width, color='green', marker='x',
s=20, label='Virginica')

    plt.xlim(left=0)
    plt.ylim(bottom=0)

    plt.title('Iris Petal Length vs Width with Decision Boundary')
    plt.xlabel('Petal Length (cm)')
    plt.ylabel('Petal Width (cm)')
    plt.grid(True)
    plt.legend()

    if filepath:
        plt.savefig(filepath)
        plt.close()
    else:
        plt.show()
        plt.close()

    return

# Initial weights and plot decision boundary
print("Initial weights W = [-10, 1.5, 1.5]")
W = np.array([-10, 1.5, 1.5])
plot_decision_boundary(vr_petal_length, vr_petal_width, va_petal_length,
va_petal_width, W)
```

The weights used were -10 for basis and 1.5 for both petal length and petal width. Below is the plot generated:



Left of the decision boundary indicates a label of 0 for the 2nd class while right of the line indicates a label of 1 for the 3rd class.

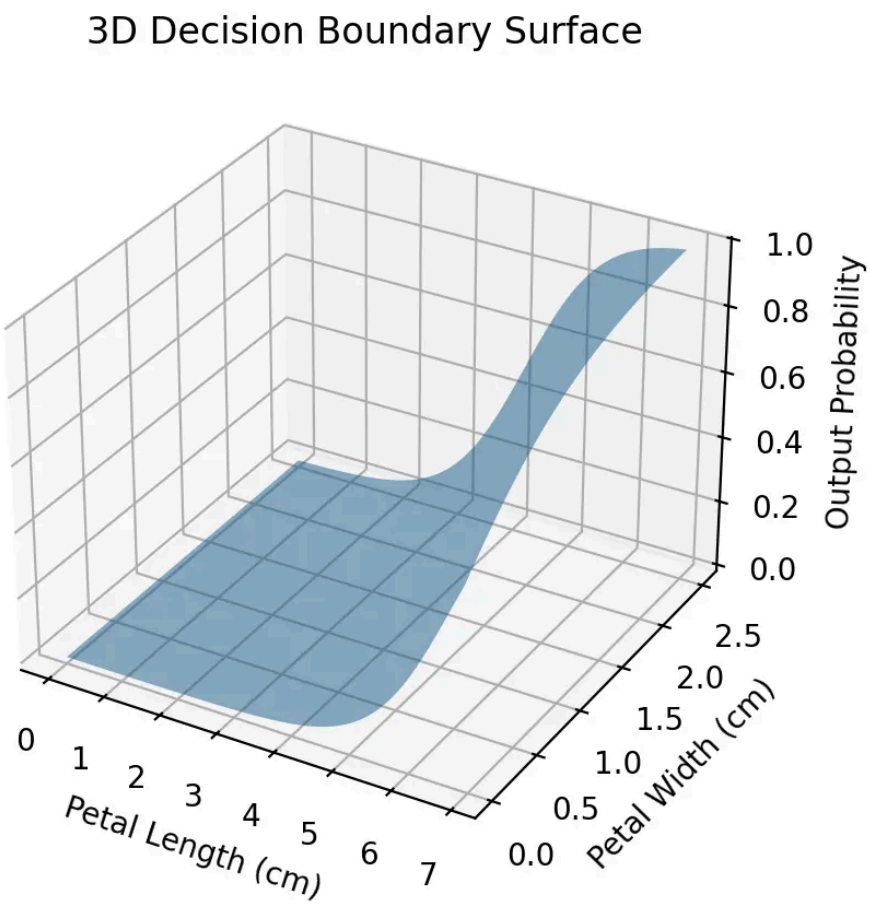
D.

Below is the code for plotting the neural network over the input space:

```
# 3D plot of decision boundary surface
def plot_3d(x_pl, x_pw, W):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    X1, X2 = np.meshgrid(x_pl, x_pw)
    Z = neural_network(np.c_[np.ones(X1.ravel().shape), X1.ravel(),
X2.ravel()], W)
    Z = Z.reshape(X1.shape)

    ax.plot_surface(X1, X2, Z, alpha=0.5, rstride=100, cstride=100)

    ax.set_xlabel('Petal Length (cm)')
    ax.set_ylabel('Petal Width (cm)')
    ax.set_zlabel('Output Probability')
    ax.set_title('3D Decision Boundary Surface')
    plt.show()
```

```
    return

# Create axes for 3D plot
pl_axis = np.linspace(0, data['petal_length'].max(), 100)
pw_axis = np.linspace(0, data['petal_width'].max(), 100)

# plot 3D decision boundary
plot_3d(pl_axis, pw_axis, W)
```

A meshgrid was created using the maximum data values for both petal length and width, and all these points were put into the neural network to generate the following plot:



E.

Below is the code used to create a simple classifier for the data:

```
# Classify iris based on petal length and width
def classify_iris(petal_length, petal_width, W, threshold=0.5):
    print(f"Classifying iris with petal length: {petal_length}, petal width:
{petal_width}")
    x = np.array([1, petal_length, petal_width])  # Add bias term
```

```
    prob = neural_network(x, W)
    print(f"Computed probability: {prob}")
    print("Classified as:", end=" ")

    # Classify based on threshold with 0 being versicolor and 1 being
virginica
    print('versicolor' if prob <= threshold else 'virginica')
    return 'versicolor' if prob <= threshold else 'virginica'

# Classify some sample irises
print("1e Simple Classifier:")
print("-----------------------------------")
# Versicolor sample
classify_iris(data[data["species"] == "versicolor"]['petal_length'].iloc[0],
data[data["species"] == "versicolor"]['petal_width'].iloc[0], W)

# Virginica sample
classify_iris(data[data["species"] == "virginica"]['petal_length'].iloc[0],
data[data["species"] == "virginica"]['petal_width'].iloc[0], W)

# Versicolor sample that is close to boundary
classify_iris(data['petal_length'].iloc[20], data['petal_width'].iloc[20],
W)

# Virginica sample that is close to boundary
classify_iris(data['petal_length'].iloc[73], data['petal_width'].iloc[73],
W)
```

The threshold was set at 0.5 as default and values less than 0.5 were classified as class 2 and those above were classified as class 3.

Four samples were used. The first was a class 2 iris that was far from the boundary. The second was a class 3 iris that was also far from the boundary. The third was a class 2 that was close to the boundary. The fourth was a class 3 that was also close to the boundary.

Below are the results:

```
1e Simple Classifier:
--------------------------------
Classifying iris with petal length: 4.7, petal width: 1.4
Computed probability: 0.2994328575260271
Classified as: versicolor
Classifying iris with petal length: 6.0, petal width: 2.5
Computed probability: 0.9399133498259924
Classified as: virginica
Classifying iris with petal length: 4.8, petal width: 1.8
Computed probability: 0.4750208125210599
Classified as: versicolor
Classifying iris with petal length: 4.9, petal width: 1.8
Computed probability: 0.5124973964842106
Classified as: virginica
```

Exercise 2

A.

Below is the code for the MSE function:

```python
# Mean Squared Error calculation
def mean_squared_error(x, W, y_true):
    mse = np.mean((y_true - neural_network(x, W)) ** 2)
    return mse
```

The MSE function calculates:

$$\text{MSE} = \frac{1}{N}\sum_{i=0}^{N}\left(\frac{1}{1+e^{-z}} - y_i\right)^2$$

with z equaling:

$$z = w_1 x_{1,i} + w_2 x_{2,i} + w_0$$

B.

Below is the code to calculate the MSE for two different weight settings and plotting their decision boundaries:

```python
print()
print()
print("2b MSE Calculations:")
```

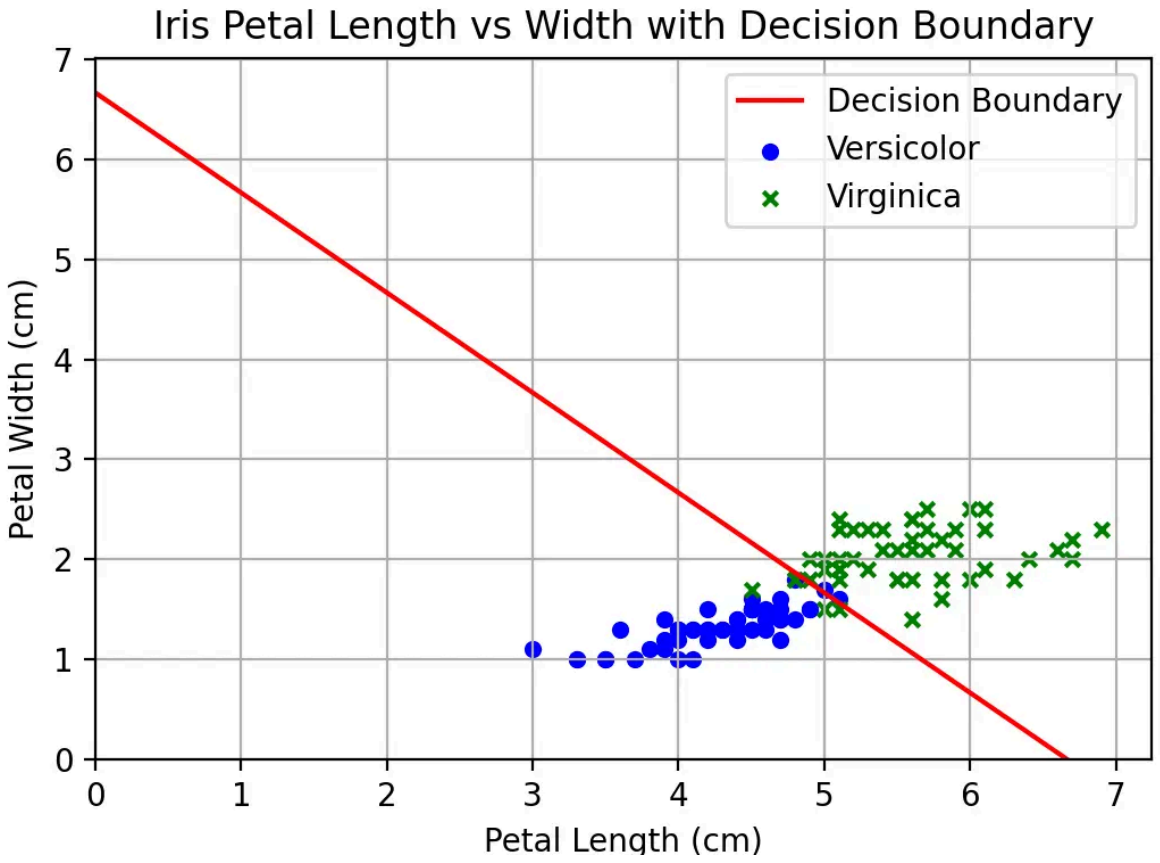
```
print("-----------------------------------")

# Prepare input data with bias term
x = np.c_[np.ones(data.shape[0]), data['petal_length'].to_numpy().ravel(),
data['petal_width'].to_numpy().ravel()]

# True labels: 0 for versicolor, 1 for virginica
y_true = np.array([1 if species == 'virginica' else 0 for species in
data['species']])

# Calculate and print MSE with initial weights
mse = mean_squared_error(x, W, y_true)
print("Current weights:")
print(f"basis = {W[0]}, w1 = {W[1]}, w2 = {W[2]}")
print(f"Initial MSE: {mse}")
plot_decision_boundary(vr_petal_length, vr_petal_width, va_petal_length,
va_petal_width, W)


# Update weights to new values and calculate MSE
W = np.array([-11, 1, 2])

print("Updated weights to W =", W)

mse = mean_squared_error(x, W, y_true)

print(f"Updated MSE: {mse}")
plot_decision_boundary(vr_petal_length, vr_petal_width, va_petal_length,
va_petal_width, W)
```

The weights from initialized from before were used for the first calculations and a weight of w_0 = -11, $w_1 = 1$, and $w_2 = 2$ were used for the second set of weights.

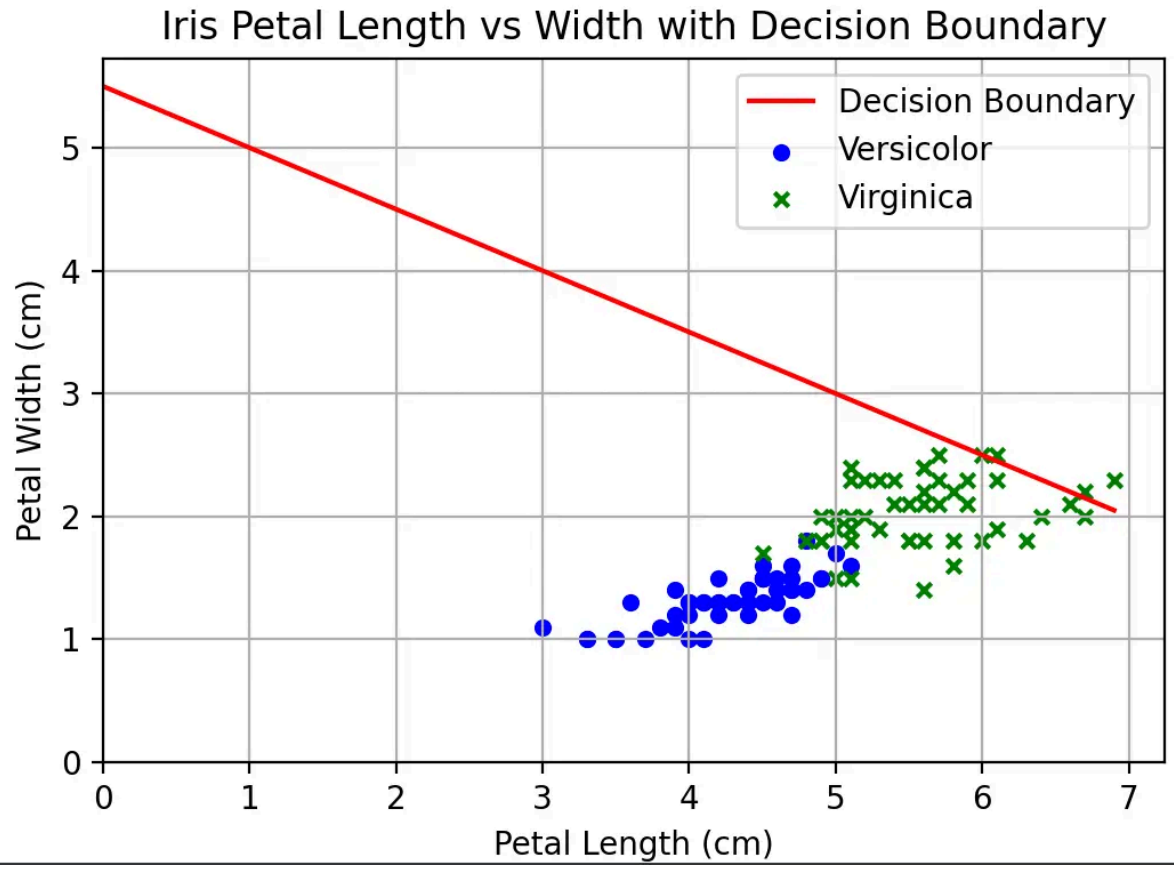
The following is the output MSEs for both weights:

```
2b MSE Calculations:
-----------------------------------
Current weights:
basis = -10.0, w1 = 1.5, w2 = 1.5
Initial MSE: 0.07162495816666498
Updated weights to W = [-11   1   2]
Updated MSE: 0.30615363878361784
Computed gradient: [-0.11091972 -0.62669362 -0.23160165]
Updated weights to W = [-10.98890803   1.06266936   2.02316017]
```

The boundary plot for the first set of weights is below:



The boundary plot for the second set of weights is below:



C.

The following is the MSE function for the sigmoidal objection function:

$$\text{MSE} = \frac{1}{N}\sum_{i=0}^{N}\left(\frac{1}{1+e^{-z}} - y_i\right)^2$$

z in the equation is equal to:

$$z = w_1 x_{1,i} + w_2 x_{2,i} + w_0$$

Taking the derivative with respect to the z results in:

$$\frac{\partial \text{MSE}}{\partial z} = \frac{-2}{N}\sum_{i=0}^{N}\left(\frac{1}{1+e^{-z}} - y_i\right)\left(\frac{1}{1+e^{-z}}\right)^2(e^{-z})$$

$$\frac{\partial \text{MSE}}{\partial w_1} = \frac{\partial \text{MSE}}{\partial w_1}\frac{\partial z}{\partial w_1}(e^{-z}) = \frac{2}{N}\sum_{i=0}^{N}\left(\frac{1}{1+e^{-z}} - y_i\right)\left(\frac{1}{1+e^{-z}}\right)^2\frac{\partial z}{\partial w_1}(e^{-z})$$

The derivative of e^{-z} with respect to w_1 is equal to:

$$\frac{\partial z}{\partial w_1}(e^{-z}) = (e^{-z})x_{1,i}$$

For w_2:

$$\frac{\partial z}{\partial w_2}(e^{-z}) = (e^{-z})x_{2,i}$$

For w_0:

$$\frac{\partial z}{\partial w_0}(e^{-z}) = (e^{-z})$$

Thus the final partial derivative is:

$$\frac{\partial MSE}{\partial w_1} = \frac{2}{N}\sum_{i=0}^{N}\left(\frac{1}{1+e^{-z}} - y_i\right)\left(\frac{1}{1+e^{-z}}\right)^2(e^{-z})x_{1,i}$$

For w_2 the partial derivative is:

$$\frac{\partial MSE}{\partial w_2} = \frac{2}{N}\sum_{i=0}^{N}\left(\frac{1}{1+e^{-z}} - y_i\right)\left(\frac{1}{1+e^{-z}}\right)^2(e^{-z})x_{2,i}$$

For b, the derivative is:

$$\frac{\partial MSE}{\partial w_1} = \frac{2}{N}\sum_{i=0}^{N}\left(\frac{1}{1+e^{-z}} - y_i\right)\left(\frac{1}{1+e^{-z}}\right)^2(e^{-z})$$

D.

Below is the code used to compute the gradient as well as plot the original and updated boundary line:

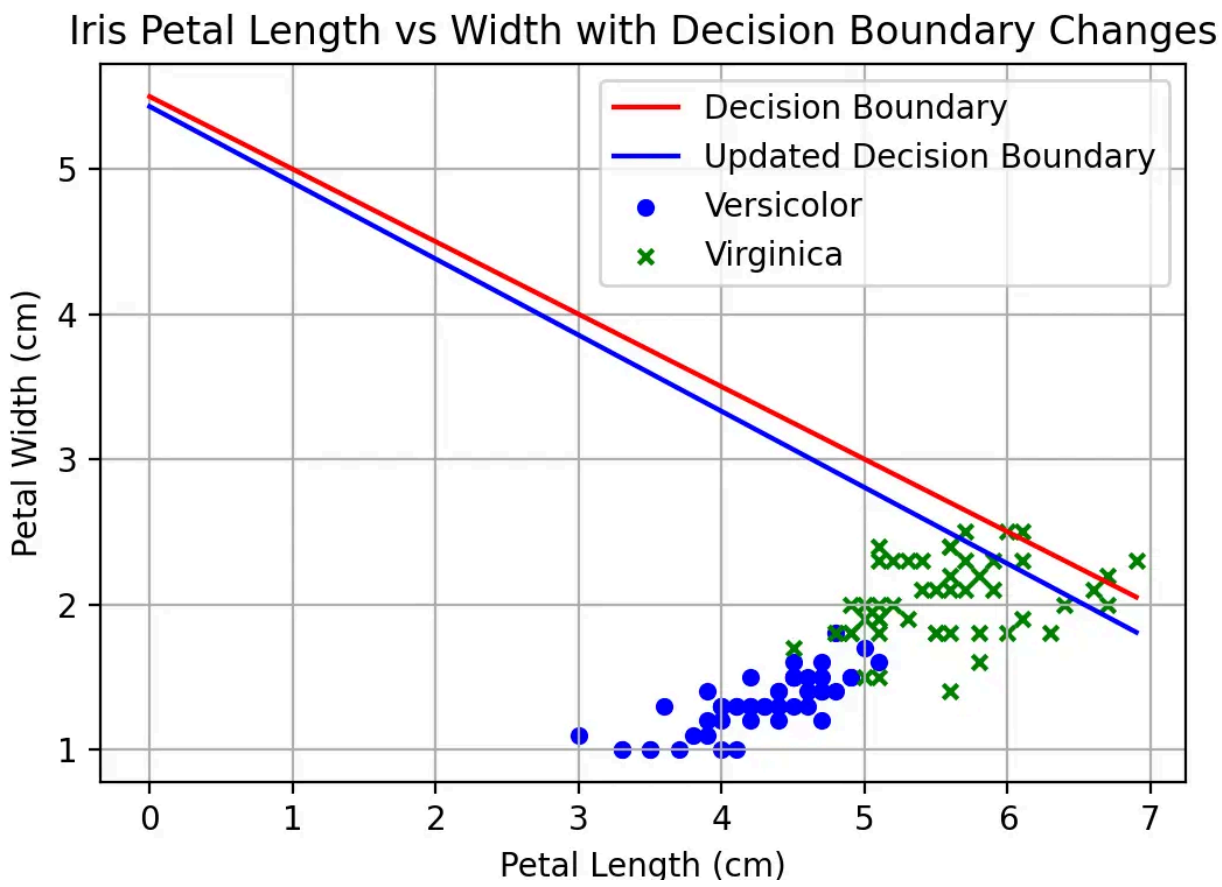
```
# Exercise 2c:
# Compute gradient of MSE with respect to weights
def compute_gradient(x, W, y_true):

    gradient_w1 = 2 * np.mean((neural_network(x, W) - y_true) *
neural_network(x, W) * (1 - neural_network(x, W)) * x[:, 1])
    gradient_w2 = 2 * np.mean((neural_network(x, W) - y_true) *
neural_network(x, W) * (1 - neural_network(x, W)) * x[:, 2])
    gradient_w0 = 2 * np.mean((neural_network(x, W) - y_true) *
neural_network(x, W) * (1 - neural_network(x, W)) * x[:, 0])

    gradient = np.array([gradient_w0, gradient_w1, gradient_w2])

    return gradient


# Test gradient computation
gradient = compute_gradient(x, W, y_true)
print(f"Computed gradient: {gradient}")
```

```
x_pl, x_pw = calculate_decision_boundary(vr_petal_length, va_petal_length,
W)

# Update weights using gradient
W = W - 0.1* gradient
print(f"Updated weights to W = {W}")


x_pl2, x_pw2 = calculate_decision_boundary(vr_petal_length, va_petal_length,
W)

x_pl = np.c_[x_pl, x_pl2]
x_pw = np.c_[x_pw, x_pw2]

# Plot changes in decision boundary
def plot_changes_in_decision_boundary(vr_petal_length, vr_petal_width,
va_petal_length, va_petal_width, x_pl, x_pw):

    plt.figure(figsize=(6, 4))
    colors = ['red', 'blue', 'green', 'orange', 'purple', 'brown', 'pink',
'gray', 'olive', 'cyan']

    for i in range(len(x_pl.T)):

        plt.plot(x_pl[:, i], x_pw[:, i], color=colors[i], label='Decision
Boundary' if i == 0 else 'Updated Decision Boundary')


    plt.scatter(vr_petal_length, vr_petal_width, color='blue', marker='o',
s=20, label='Versicolor')
    plt.scatter(va_petal_length, va_petal_width, color='green', marker='x',
s=20, label='Virginica')

    plt.title('Iris Petal Length vs Width with Decision Boundary Changes')
    plt.xlabel('Petal Length (cm)')
    plt.ylabel('Petal Width (cm)')
    plt.grid(True)
    plt.legend()
    plt.show()

    return

# Plot the changes in decision boundary after weight update
```

```
plot_changes_in_decision_boundary(vr_petal_length, vr_petal_width,
va_petal_length, va_petal_width, x_pl, x_pw)
```

A step of 0.1 was used for this comparison between the initial and the updated. calculate_decision_boundary function was used to determine the points of both lines and a new function plot_decision_boundary_change was used to plot both lines on top of the same graph.

Below is the resulting plot:



Exercise 3:

A.

Below is the code for implementing gradient descent:

```
# Gradient Descent Implementation
def gradient_descent(x, W, y_true, learning_rate=0.01, iterations=100,
convergence=False):

    # Store history for plotting
    W_history = [W.copy()]

    mse_history = [mean_squared_error(x, W, y_true)]
```

```
    # Gradient descent loop
    for i in range(iterations):
        # Compute gradient
        grad = compute_gradient(x, W, y_true)

        # Update weights
        W -= learning_rate * grad
        W_history.append(W.copy())
        mse_history.append(mean_squared_error(x, W, y_true))

        #Used in 3d
        if convergence and i > 0 and abs(mse_history[-2] - mse_history[-1])
< 1e-6:
            break

    # Return final weights, history of weights, and MSE history
    return W, W_history, mse_history
```

The function takes in the data points, weights, true labels, learning rate, and iterations. A history of the weights and MSE values are kept and tracked throughout the iterations. Weights are updated by multiplying the learning rate with the computed gradient. The final weights, weight history, and MSE history are then returned.

The convergence argument and if statement is used in a exercise 3d.

B.

Below is the code for the optimizing the decision boundary through gradient descent using preset initial weights, learning rate, and iterations:

```
# Single run of gradient descent
print()
print()
print("3b Single Run:")
print("-----------------------------------")

print("Weights initialized to:")
print(f"basis: {W[0]}, w1: {W[1]}, w2: {W[2]}")

# Run gradient descent with specified learning rate and iterations
print("Learning Rate: 0.1, Iterations: 10")
print("initial MSE:", mean_squared_error(x, W, y_true))
iterations = 10
W, W_history, mse_history = gradient_descent(x, W, y_true,
```

```
learning_rate=0.1, iterations=iterations)

# Print final weights and MSE
print("final weights after gradient descent:")
print(f"basis: {W[0]}, w1: {W[1]}, w2: {W[2]}")
print("final MSE:", mean_squared_error(x, W, y_true))

# Create directory for saving plots
threeb = os.path.join(os.getcwd(), '3b')
os.makedirs(threeb, exist_ok=True)

# Plot decision boundary at each iteration
for i in range(iterations + 1):
    filepath = os.path.join(threeb, f"decision_boundary_{i}.png")
    plot_decision_boundary(vr_petal_length, vr_petal_width, va_petal_length,
va_petal_width, W_history[i], filepath=filepath)

# Plot MSE over iterations
def plot_mse_over_iterations(mse_history, filepath=None):
    plt.figure(figsize=(6, 4))
    plt.xlim(left=0)
    plt.ylim(bottom=0)
    plt.figure(figsize=(8, 6))
    plt.plot(range(len(mse_history)), mse_history)
    plt.title('Mean Squared Error over Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Mean Squared Error')
    plt.grid(True)

    if filepath:
        plt.savefig(filepath)
        plt.close()

    else:
        plt.show()
        plt.close()

    return

# Save MSE plot
mse_filepath = os.path.join(threeb, "mse_over_iterations.png")
plot_mse_over_iterations(mse_history, filepath=mse_filepath)

print("Plots for 3b saved in directory: ", threeb)
```

The weights used are w_0 = -11, w_1 = 1, and w_2 = 2. The learning rate was set to 0.1 and the number of iterations was set to 10.

Due to the large number of images, for each iteration, the decision boundary was plotted and stored in "…/ZhaoDylan_HW9_files/3b" (if running from the stored folder) as well as the MSE plot was also stored in there.

The code with create a "3b" directory if it does not exist already.

Below is the output numerical results:

```
3b Single Run:
--------------------------------
Weights initialized to:
basis: -10.988908028037118, w1: 1.0626693623747547, w2: 2.0231601651699047
Learning Rate: 0.1, Iterations: 10
initial MSE: 0.2593191329833751
final weights after gradient descent:
basis: -10.919445648723979, w1: 1.442948094159604, w2: 2.163130946925883
final MSE: 0.06470663705801899
Plots for 3b saved in directory:  /Users/dylanzhao/Documents/College/Junior/CSDS 391/Zha
oDylan_HW9_files/3b
```

C.

Below is the code for the running gradient descent using random weights, learning rate, and number of iterations:

```
# Multiple runs of gradient descent with random initialization
print()
print()
print("3c Multiple Runs:")
print("--------------------------------")

# Set random seed for reproducibility
np.random.seed(42)

# Perform 5 runs with different random initializations
for i in range(5):
    print()
    print()

    # Randomly initialize weights
    W = np.array([
        np.random.uniform(-12, -8),   # bias
        np.random.uniform(0.1, 2),    # w1
        np.random.uniform(0.1, 2)     # w2
    ])

    # Print run information
```

```
    print(f"--- Run {i+1} ---")
    print("weights initialized to:")
    print(f"basis: {W[0]}, w1: {W[1]}, w2: {W[2]}")
    iterations = np.random.randint(100, 500)
    learningrate = np.random.uniform(0.01, 0.5)
    print(f"Learning Rate: {learningrate}, Iterations: {iterations}")
    print("initial MSE:", mean_squared_error(x, W, y_true))

    # Run gradient descent
    W, W_history, mse_history = gradient_descent(x, W, y_true,
learning_rate=learningrate, iterations=iterations)

    print()
    print("final weights after gradient descent:")
    print(f"basis: {W[0]}, w1: {W[1]}, w2: {W[2]}")
    print("final MSE:", mean_squared_error(x, W, y_true))

    # Create directory for this run's plots
    threec = os.path.join(os.getcwd(), '3c', f'run_{i+1}')
    os.makedirs(threec, exist_ok=True)

    # Plot decision boundary at each iteration
    for j in range(iterations + 1):
        plot_decision_boundary(vr_petal_length, vr_petal_width,
va_petal_length, va_petal_width, W_history[j], filepath=os.path.join(threec,
f"decision_boundary_{j}.png"))

    # Plot MSE over iterations
    plot_mse_over_iterations(mse_history, filepath=os.path.join(threec,
"mse_over_iterations.png"))

    print(f"Plots for Run {i+1} saved in directory: {threec}")
```

The random values for the basis were set to be contained from -8 to -10. This was determined because the x_2 values are calculated via:

$$x_2 = -\frac{w_0}{w_2} - \frac{w_1}{w_2}x_1$$

Thus a negative value was used for w_0. The specific values were used to constrain the graph from going off the plot as well as to prevent the weights from being to inaccurate that the MSE plateaus at 0.5.

The weights for w_1 and w_2 were constrained to 0.1 to 2 to make sure the slope was negative as well as to make sure that the boundary line didn't go off the plot with super high and or super low slopes. 0.1 was chosen as the minimum due to as w_2 cannot equal 0.

The iterations range was chosen to be from 100 to 500 to make sure that gradient descent will not prematurely end before a somewhat optimized boundary line is found. Additionally a cap of 500 was used to prevent too many plots and images from being generated.

The learning rate was chosen to range from 0.01 to 0.5. This is due to if the learning rate is too low, it would take too many iterations before reaching a somewhat optimal boundary line. A cap was also created for the same reason.

All images generated at each iteration as well as each run's MSE over iteration plot are stored within the folder "3c" under its corresponding run. This folder is created if it does not exist already.

Below is the numerical initial and final results for exact values:

```
--- Run 1 ---
weights initialized to:
basis: -10.50183952461055, w1: 1.9063571821788405, w2: 1.4907884894416696
Learning Rate: 0.3024565773937786, Iterations: 288
initial MSE: 0.12108187262629257

final weights after gradient descent:
basis: -10.90377500140357, w1: 1.5476879895156328, w2: 2.0293529625374784
final MSE: 0.06231866287653572
Plots for Run 1 saved in directory: /Users/dylanzhao/Documents/College/Junior/CSDS 391/ZhaoDylan_HW9_files/3
c/run_1


--- Run 2 ---
weights initialized to:
basis: -10.216668988585635, w1: 0.2899523400542055, w2: 0.9725728947351475
Learning Rate: 0.3045463557541723, Iterations: 472
initial MSE: 0.49861047810341774

final weights after gradient descent:
basis: -10.52499658150977, w1: 1.3899319423947416, w2: 2.266928560730054
final MSE: 0.06225449662353278
Plots for Run 2 saved in directory: /Users/dylanzhao/Documents/College/Junior/CSDS 391/ZhaoDylan_HW9_files/3
c/run_2


--- Run 3 ---
weights initialized to:
basis: -9.167709688815819, w1: 0.13911053916202465, w2: 1.942828719107789
Learning Rate: 0.010381595262097022, Iterations: 393
initial MSE: 0.4869148811285877

final weights after gradient descent:
basis: -9.006836619935076, w1: 1.0633199044640738, w2: 2.2902088429490397
final MSE: 0.06877417368501054
Plots for Run 3 saved in directory: /Users/dylanzhao/Documents/College/Junior/CSDS 391/ZhaoDylan_HW9_files/3
c/run_3


--- Run 4 ---
weights initialized to:
basis: -8.03115376283513, w1: 1.2732148682926614, w2: 1.2621410049277337
Learning Rate: 0.22165305913463673, Iterations: 352
initial MSE: 0.08577397923058415

final weights after gradient descent:
basis: -8.537151113239302, w1: 1.1010258489763807, w2: 1.9233551068228845
final MSE: 0.07366783319341329
Plots for Run 4 saved in directory: /Users/dylanzhao/Documents/College/Junior/CSDS 391/ZhaoDylan_HW9_files/3
c/run_4
```

```
--- Run 5 ---
weights initialized to:
basis: -10.835083439207832, w1: 1.262520499972521, w2: 0.36503833523887946
Learning Rate: 0.18951730321390894, Iterations: 287
initial MSE: 0.4473606347113727

final weights after gradient descent:
basis: -10.949079543680083, w1: 1.8514558412184894, w2: 1.1663122874505192
final MSE: 0.06909697205595444
Plots for Run 5 saved in directory: /Users/dylanzhao/Documents/College/Junior/CSDS 391/ZhaoDylan_HW9_files/3
c/run_5
```

D.

Below is the code used for creating a convergence criteria:

```
# Gradient Descent Implementation
def gradient_descent(x, W, y_true, learning_rate=0.01, iterations=100,
convergence=False):

    # Store history for plotting
    W_history = [W.copy()]

    mse_history = [mean_squared_error(x, W, y_true)]

    # Gradient descent loop
    for i in range(iterations):
        # Compute gradient
        grad = compute_gradient(x, W, y_true)

        # Update weights
        W -= learning_rate * grad
        W_history.append(W.copy())
        mse_history.append(mean_squared_error(x, W, y_true))

        #Used in 3d
        if convergence and i > 0 and abs(mse_history[-2] - mse_history[-1])
< 1e-6:
            break

    # Return final weights, history of weights, and MSE history
    return W, W_history, mse_history


# Utilizing a Convergece Criterion
print()
print()
print("3d Convergence:")
```

```
print("-----------------------------------")

# Randomly initialize weights
W = np.array([
        np.random.uniform(-12, -8),   # bias
        np.random.uniform(0.1, 2),    # w1
        np.random.uniform(0.1, 2)     # w2
    ])

# Print initial weights and settings
print("weights initialized to:")
print(f"basis: {W[0]}, w1: {W[1]}, w2: {W[2]}")
# Set learning rate and maximum iterations
iterations = 1000
learningrate = 0.01

# Print learning rate and iterations
print(f"Learning Rate: {learningrate}, Iterations: {iterations}")
print("initial MSE:", mean_squared_error(x, W, y_true))
plot_decision_boundary(vr_petal_length, vr_petal_width, va_petal_length,
va_petal_width, W, filepath=os.path.join(os.getcwd(), '3d',
f"decision_boundary_initial.png"))

# Run gradient descent with convergence criterion
W, W_history, mse_history = gradient_descent(x, W, y_true,
learning_rate=learningrate, iterations=iterations, convergence = True)

# Print final results
print()
print(f"Converged in {len(mse_history)-1} iterations.")
print("final weights after gradient descent:")
print(f"basis: {W[0]}, w1: {W[1]}, w2: {W[2]}")
print("final MSE:", mean_squared_error(x, W, y_true))

# Check for convergence to less than 50% of initial MSE
for i in range(len(mse_history)):
    if mse_history[i]/mse_history[0] < 0.5:
        print()
        print(f"Converged to less than 50% of initial MSE at iteration {i}")
        print("Weights at this iteration:")
        print(f"basis: {W_history[i][0]}, w1: {W_history[i][1]}, w2:
{W_history[i][2]}")
        print("MSE at this iteration:", mse_history[i])
        print()
        plot_decision_boundary(vr_petal_length, vr_petal_width,
va_petal_length, va_petal_width, W_history[i],
```

```
filepath=os.path.join(os.getcwd(), '3d',
f"decision_boundary_convergence_half.png"))
        break

# Create directory for saving plots
threed = os.path.join(os.getcwd(), '3d')

os.makedirs(threed, exist_ok=True)

# Plot decision boundary at final iteration
plot_decision_boundary(vr_petal_length, vr_petal_width, va_petal_length,
va_petal_width, W, filepath=os.path.join(threed,
f"decision_boundary_final.png"))

print("Plots for 3d saved in directory: ", threed)
```

The weights were chosen by random using the same restrictions. The convergence was implemented in the original gradient descent function. The optional convergence parameter is set to false by default, and by passing it as true, it will converge when the previous MSE and current MSE value have a difference less than 1×10^{-6}.

The number of iterations of 1000 and a learning rate of 0.01 was chosen.

The iteration where MSE was first less than half the original was plotted as well as initial and final iterations. These plots are stored in folder "3d" to prevent any confusion from the other prior plots. Each plot file is labeled accordingly to which state that got plotted. The folder "3d" will be created if it does not exist already.

Below is the numerical output for exact values:

```
3d Convergence:
--------------------------------
weights initialized to:
basis: -10.175720063131855, w1: 1.5918343266467259, w2: 0.4793801861008835
Learning Rate: 0.01, Iterations: 1000
initial MSE: 0.1921268930960538

Converged in 1000 iterations.
final weights after gradient descent:
basis: -10.172381139996919, w1: 1.8536852414898872, w2: 0.6991541381531962
final MSE: 0.07759600357031517

Converged to less than 50% of initial MSE at iteration 42
Weights at this iteration:
basis: -10.142536418384696, w1: 1.772728704501166, w2: 0.5484563448066844
MSE at this iteration: 0.09602608467010677

Plots for 3d saved in directory:  /Users/dylanzhao/Documents/College/Junior/CSDS 391/ZhaoDylan_HW9_files/3d
```