PR Assignment 4

COE18B045 - G S Santhosh Raghul

COE18B048 - Shresta M

common code which has been reused for the different questions:

perceptron.py

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

class Perceptron:

    def __init__(self,train_data,train_labels):

        assert (len(train_data)==len(train_labels)),"length of train_data and train_labels must match"
        self.__raw_data=train_data

        # append 1 to train data to get y
        self.__train_data=np.append(train_data,np.array([[1]]*len(train_data)),axis=1)

        # check if train_labels is valid
        assert all(np.isin(train_labels,[0,1])),"'train_lables' should contain only 0s or 1s"
        self.train_labels=train_labels

        # negate y values from class 1
        for i,c in enumerate(train_labels):
            if c==1:
                self.__train_data[i]=-self.__train_data[i]

        # initialise weight vector, set default learning rate
        self.__weight=np.zeros_like(self.__train_data[0])
        self.__learning_rate=0.01
        self.__split_for_plotting()

    def __split_for_plotting(self):

        self.x1=[]
        self.y1=[]
        self.x2=[]
        self.y2=[]
        for i,p in enumerate(self.__raw_data):
            if self.train_labels[i]==1:
                self.x1.append(p[0])
                self.y1.append(p[1])
            else:
                self.x2.append(p[0])
                self.y2.append(p[1])

    # to manually set wait vector
    def set_weight(self,weight):
        if weight.shape!=self.__weight.shape:
            raise ValueError(f"given weight vector must be of shape 1x(d+1), 1x{self.__weight.shape[0]} he
        self.__weight=weight
    # to set learning rate
    def set_learning_rate(self,learning_rate):
        self.__learning_rate=learning_rate

    # to do 1 iteration of learning
    def train(self):
        gradient_of_Jp=np.zeros_like(self.__weight)
        for y in self.__train_data:
            if not self.__weight @ y > 0:
                gradient_of_Jp+=y
        print(f'gradient of Jp = {gradient_of_Jp}')
        print(f'new weight = old weight + learning rate * gradient of Jp\n           = {self.__weight} + {
        self.__weight= self.__weight + self.__learning_rate * gradient_of_Jp
        print(f"           = {self.__weight}\n")

    # to plot the data and the decision boundary
    def show_plot(self,title=''):

        # styles
        plt.figure(figsize=(8,8))
        plt.figtext(0.5, 0.9, title, ha="center", fontsize=20)
        plt.axvline(0,color='black',linewidth=.8)
        plt.axhline(0,color='black',linewidth=.8)
        plt.grid(color='grey', linestyle=':', linewidth=.5)

        # plotting data
        plt.scatter(self.x1,self.y1)
        plt.scatter(self.x2,self.y2)

        # plotting decision boundary
        if np.any(self.__weight[:-1]):
            a,b,c=self.__weight
            if b==0:
                plt.axvline(-c/a, c='black', label='decision boundary')
            else:
                y_intercept=-c/b
                slope=-a/b
                plt.axline((0,y_intercept), slope=slope, c='black', label='decision boundary')
            plt.legend(loc='best',fontsize=16)


        plt.figtext(0.5, 0.04, "weight vector : "+str(self.__weight), ha="center", fontsize=20)
        title=title.replace(' ','_')
        plt.savefig(f'output_images/{title}.png')
        plt.show()

    def get_weight(self):
        return self.__weight

def demo(data,label,plot_title='',learning_rate=None,weight=None):

    a=Perceptron(data,label)
    if learning_rate is not None:
        a.set_learning_rate(learning_rate)
    if weight is not None:
        a.set_weight(weight)
    if plot_title!='':
        plot_title=plot_title+' '

    a.show_plot(plot_title+"perceptron before training")
    i=1
    prev_weight=a.get_weight()
    while True:
        print(f"perceptron iteration {i}:\n")
        a.train()
        a.show_plot(plot_title+f"perceptron iteration {i}")
        i+=1
        if np.allclose(prev_weight,a.get_weight()):
            break
        prev_weight=a.get_weight()

    print("no significant change in weight vector after this iteration. stopping.")
```

svm.py

```
import numpy as np
import matplotlib.pyplot as plt
from cvxopt import matrix
from cvxopt import solvers

class svm:

    def __init__(self,train_data,train_labels):

        self.__X=train_data
        self.__Y=np.array([train_labels,])
        self.__weight=[]
        self.__bias=None
        self.__split_for_plotting()

    def train(self):

        n=self.__X.shape[0]

        H=matrix(np.multiply((self.__Y.T @ self.__Y),(self.__X @ self.__X.T)).astype(np.float))
        f=matrix(np.array([-1]*n).astype(np.float),tc='d')
        A=matrix(-np.eye(n).astype(np.float))
        a=matrix(np.array([0.0]*n).astype(np.float))
        B=matrix(self.__Y.astype(np.float),tc='d')
        b=matrix(0.0)

        solvers.options['show_progress'] = False
        solution = solvers.qp(H,f,A,a,B,b)
        alphas = np.array(solution['x'])

        self.__weight=np.zeros_like(self.__X[0],dtype=float)
        for i,alpha in enumerate(alphas):
                self.__weight+=alpha*self.__Y[0][i]*self.__X[i]

        max_index=np.argmax(alphas)
        self.__bias = self.__Y[0][max_index] - self.__weight.T @ self.__X[max_index]
    def __split_for_plotting(self):

        self.x1=[]
        self.y1=[]
        self.x2=[]
        self.y2=[]
        for i,p in enumerate(self.__X):
            if self.__Y[0][i]==1:
                self.x1.append(p[0])
                self.y1.append(p[1])
            else:
                self.x2.append(p[0])
                self.y2.append(p[1])

    def show_plot(self,title=''):

        # styles
        plt.figure(figsize=(8,8))
        plt.figtext(0.5, 0.9, title, ha="center", fontsize=20)
        plt.axvline(0,color='black',linewidth=.8)
        plt.axhline(0,color='black',linewidth=.8)
        plt.grid(color='grey', linestyle=':', linewidth=.5)

        # plotting data
        plt.scatter(self.x2,self.y2)
        plt.scatter(self.x1,self.y1)

        # plotting decision boundary
        if np.any(self.__weight):
            a,b=self.__weight
            c=self.__bias
            if b==0:
                plt.axvline(-c/a, c='black', label='decision boundary')
            else:
                y_intercept=-c/b
                slope=-a/b
                plt.axline((0,y_intercept), slope=slope, c='black', label='decision boundary')
            plt.legend(loc='best',fontsize=16)

        plt.figtext(0.5, 0.01, f'weight : {self.__weight}\nbias : {self.__bias}', ha="center", fontsize=20
        title=title.replace(' ','_')
        plt.savefig(f'output_images/{title}.png')
        plt.show()

def demo(data,label,plot_title=''):

    a=svm(data,label)
    if plot_title!='':
        plot_title=plot_title+' '

    a.show_plot(plot_title+"svm before training")
    a.train()
    a.show_plot(plot_title+"svm after training")
```

multi_class_perceptron.py

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

class multi_Perceptron:

    def __init__(self,train_data,train_labels):

        self.__train_data=np.append(train_data,np.array([[1]]*len(train_data)),axis=1)
        self.__train_labels=train_labels
        self.__classes=np.unique(self.__train_labels)
        self.__c=len(self.__classes)

        # initialise weight vector, set default learning rate
        self.__weight=np.zeros_like(self.__train_data[0])
        self.__learning_rate=0.01
        self.__split_data_points()
        self.__weight=np.zeros((self.__c,self.__train_data.shape[1]),dtype=float)

    def __split_data_points(self):

        self.__data_points=dict([(c,[[],[]]) for c in self.__classes])
        for i,p in enumerate(self.__train_data):
            self.__data_points[self.__train_labels[i]][0].append(p[0])
            self.__data_points[self.__train_labels[i]][1].append(p[1])

        self.__class_wise_train_data=[]
        for i in range(self.__c):
            self.__class_wise_train_data.append(-self.__train_data.copy())

        for i in range(self.__c):
            for j in range(self.__train_data.shape[0]):
                if self.__train_labels[j]==i:
                    self.__class_wise_train_data[i][j]=-self.__class_wise_train_data[i][j]

    # to set learning rate
    def set_learning_rate(self,learning_rate):
        self.__learning_rate=learning_rate

    # to do 1 iteration of learning
    def train(self):
        for i,train_data in enumerate(self.__class_wise_train_data):
            # print(train_data)
            gradient_of_Jp=np.zeros(self.__train_data[0].shape,dtype=float)
            for y in train_data:
                if not self.__weight[i] @ y > 0:
                    gradient_of_Jp+=y
            # print(f'gradient of Jp = {gradient_of_Jp}')
            # print(f'new weight = old weight + learning rate * gradient of Jp\n           = {self.__weigh
            self.__weight[i]= self.__weight[i] + self.__learning_rate * gradient_of_Jp
            # print(f"           = {self.__weight}\n")

    # to plot the data and the decision boundary
    def show_plot(self,title=''):

        # styles
        plt.figure(figsize=(8,8))
        plt.figtext(0.5, 0.9, title, ha="center", fontsize=20)
        plt.axvline(0,color='black',linewidth=.8)
        plt.axhline(0,color='black',linewidth=.8)
        plt.grid(color='grey', linestyle=':', linewidth=.5)

        # plotting data
        for c in self.__data_points:
            plt.scatter(self.__data_points[c][0],self.__data_points[c][1],label=f'class {c}')

        # plotting decision boundary
        def plot_decision_boundary(weight,label,color):
            if np.any(weight[:-1]):
                a,b,c=weight
                if b==0:
                    plt.axvline(-c/a, c=color, label=label)
                else:
                    y_intercept=-c/b
                    slope=-a/b
                    plt.axline((0,y_intercept), slope=slope, c=color, label=label)

        colors=['b','g','y']
        for i in range(self.__weight.shape[0]):
            plot_decision_boundary(self.__weight[i],f'db b/w "class {i}" and "not class {i}"',colors[i])

        plt.legend(loc='best',fontsize=16)
        plt.figtext(0.35, 0.04, "weight vectors :", ha="center", fontsize=16)
        plt.figtext(0.6, 0.005, str(self.__weight), ha="center", fontsize=14)
        title=title.replace(' ','_')
        plt.savefig(f'output_images/{title}.png')
        plt.show()

    def get_weight(self):
        return self.__weight.copy()

def demo(data,label,plot_title='',learning_rate=None,weight=None):

    a=multi_Perceptron(data,label)
    if learning_rate is not None:
        a.set_learning_rate(learning_rate)
    if weight is not None:
        a.set_weight(weight)
    if plot_title!='':
        plot_title=plot_title+' '

    a.show_plot(plot_title+"perceptron before training")
    i=1
    prev_weight=a.get_weight()
    while True:
        a.train()
        if i in [100,500,1000]:
            a.show_plot(plot_title+f"perceptron iteration {i}")

        if i==1000:
            break
        prev_weight=a.get_weight()
        i+=1

    # a.show_plot(plot_title+"perceptron before training")
    # for i in range(1,11):
    #   print(f"perceptron iteration {i}:\n")
    #   a.train()
    #   a.show_plot(plot_title+f"perceptron iteration {i}")
```

multi_class_svm.py

```
import numpy as np
import matplotlib.pyplot as plt
from cvxopt import matrix
from cvxopt import solvers

class svm:

    def __init__(self,train_data,train_labels):

        self.__X=train_data
        self.__train_labels=train_labels
        self.__classes=np.unique(self.__train_labels)
        self.__c=len(self.__classes)
        self.__Y_for_each_class=[]
        for i in range(self.__c):
            self.__Y_for_each_class.append(train_labels.copy())
            for j in range(train_labels.shape[0]):
                self.__Y_for_each_class[i][j] = 1.0 if self.__Y_for_each_class[i][j] == i else -1
        self.__Y_for_each_class=np.array(self.__Y_for_each_class)
        self.__split_for_plotting()
        self.__weight=np.array([])
        self.__bias=np.array([])

    def train(self):

        n=self.__X.shape[0]

        def train_for_one_class(Y):

            H=matrix(np.multiply((Y.T @ Y),(self.__X @ self.__X.T)).astype(np.float))
            f=matrix(np.array([-1]*n).astype(np.float),tc='d')
            A=matrix(-np.eye(n).astype(np.float))
            a=matrix(np.array([0.0]*n).astype(np.float))
            B=matrix(Y.astype(np.float),tc='d')
            b=matrix(0.0)

            solvers.options['show_progress'] = False
            solution = solvers.qp(H,f,A,a,B,b)
            alphas = np.array(solution['x'])

            weight=np.zeros_like(self.__X[0],dtype=float)
            for i,alpha in enumerate(alphas):
                    weight+=alpha*Y[0][i]*self.__X[i]

            # max_index=np.argmax(alphas)
            # bias = Y[0][max_index] - weight.T @ self.__X[max_index]
            l1=[]
            l2=[]
            for i in range(len(self.__X)):
                if Y[0][i]==1:
                    l1.append(weight.T @ self.__X[i])
                else:
                    l2.append(weight.T @ self.__X[i])

            bias=0.5 * (np.min(l1)-np.max(l2))

            return weight,bias

        self.__weight=[]
        self.__bias=[]
        for i in range(self.__c):
            weight,bias=train_for_one_class(self.__Y_for_each_class[i].reshape(1,-1))
            self.__weight.append(weight)
            self.__bias.append(bias)

        self.__weight=np.array(self.__weight)
        self.__bias=np.array(self.__bias)

    def __split_for_plotting(self):

        self.__data_points=dict([(c,[[],[]]) for c in self.__classes])
        for i,p in enumerate(self.__X):
            self.__data_points[self.__train_labels[i]][0].append(p[0])
            self.__data_points[self.__train_labels[i]][1].append(p[1])

    # to plot the data and the decision boundary
    def show_plot(self,title=''):

        # styles
        plt.figure(figsize=(8,8))
        plt.figtext(0.5, 0.9, title, ha="center", fontsize=20)
        plt.axvline(0,color='black',linewidth=.8)
        plt.axhline(0,color='black',linewidth=.8)
        plt.grid(color='grey', linestyle=':', linewidth=.5)

        # plotting data
        for c in self.__data_points:
            plt.scatter(self.__data_points[c][0],self.__data_points[c][1],label=f'class {c}')

        # plotting decision boundary
        def plot_decision_boundary(weight,bias,label,color):
            if np.any(weight[:-1]):
                a,b=weight
                c=bias
                if b==0:
                    plt.axvline(-c/a, c=color, label=label)
                else:
                    y_intercept=-c/b
                    slope=-a/b
                    plt.axline((0,y_intercept), slope=slope, c=color, label=label)

        colors=['b','g','y']
        for i in range(self.__bias.shape[0]):
            plot_decision_boundary(self.__weight[i],self.__bias[i],f'db b/w "class {i}" and "not class {i}

        plt.figtext(0.5, 0.01, f'weight : {self.__weight}\nbias : {self.__bias}', ha="center", fontsize=20
        plt.legend(loc='best',fontsize=16)
        title=title.replace(' ','_')
        plt.savefig(f'output_images/{title}.png')
        plt.show()

def demo(data,label,plot_title=''):

    a=svm(data,label)
    if plot_title!='':
        plot_title=plot_title+' '

    a.show_plot(plot_title+"svm before training")
    a.train()
    a.show_plot(plot_title+"svm after training")
```

Question 1

Train a **single perceptron and SVM** to learn an AND gate with two inputs x1 and x2. Assume that all the weights of the perceptron are initialized as 0. Show the calulation for each step and also draw the decisionboundary for each updation.

code

```
import perceptron,svm
import numpy as np

data = np.array([[0,0],[0,1],[1,0],[1,1]])
perc_label = np.array([0, 0, 0,  1])
svm_label  = np.array([1, 1, 1, -1])

perceptron.demo(data,perc_label,learning_rate=0.5,plot_title="q1")

svm.demo(data,svm_label,plot_title="q1")
```

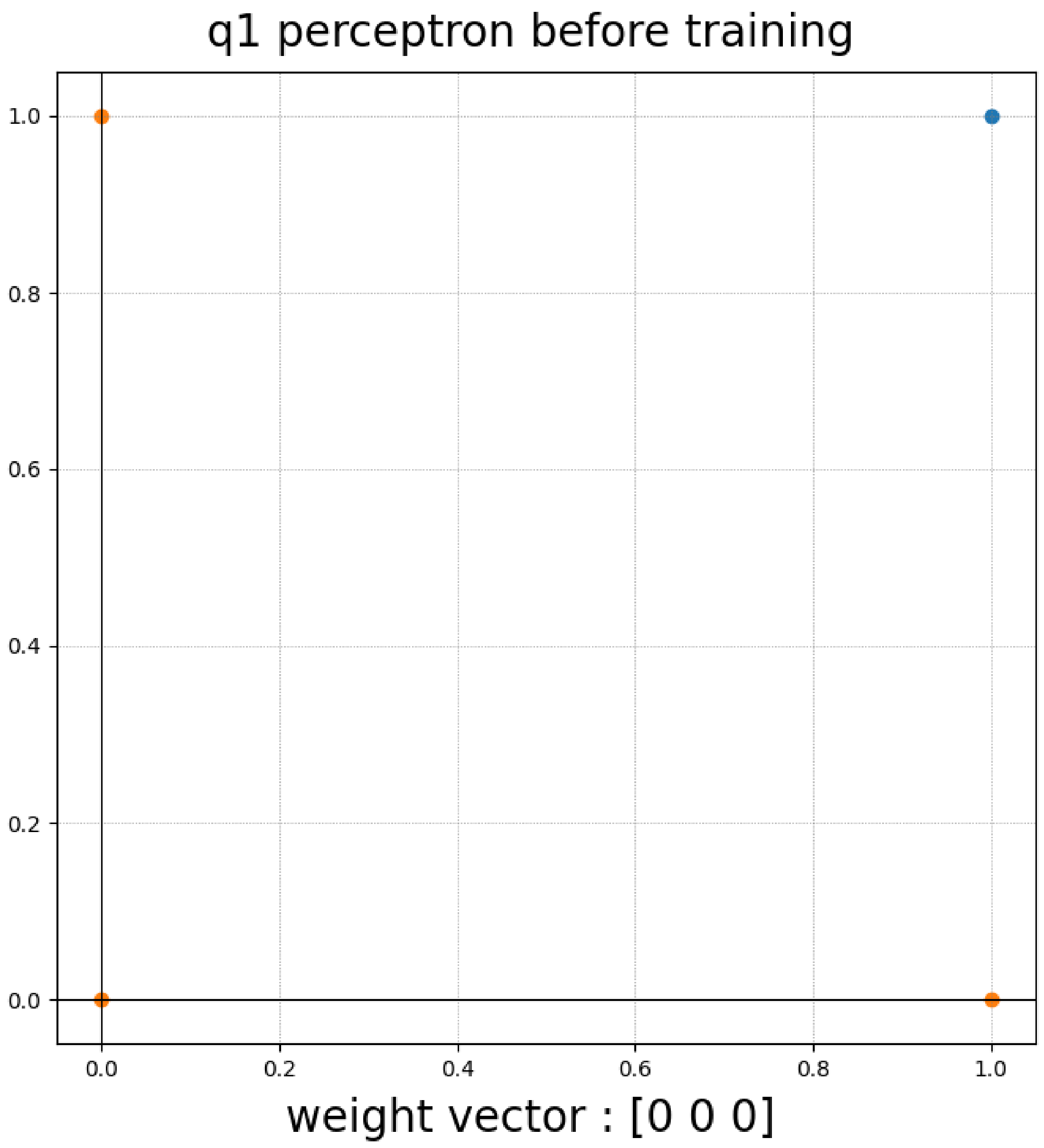
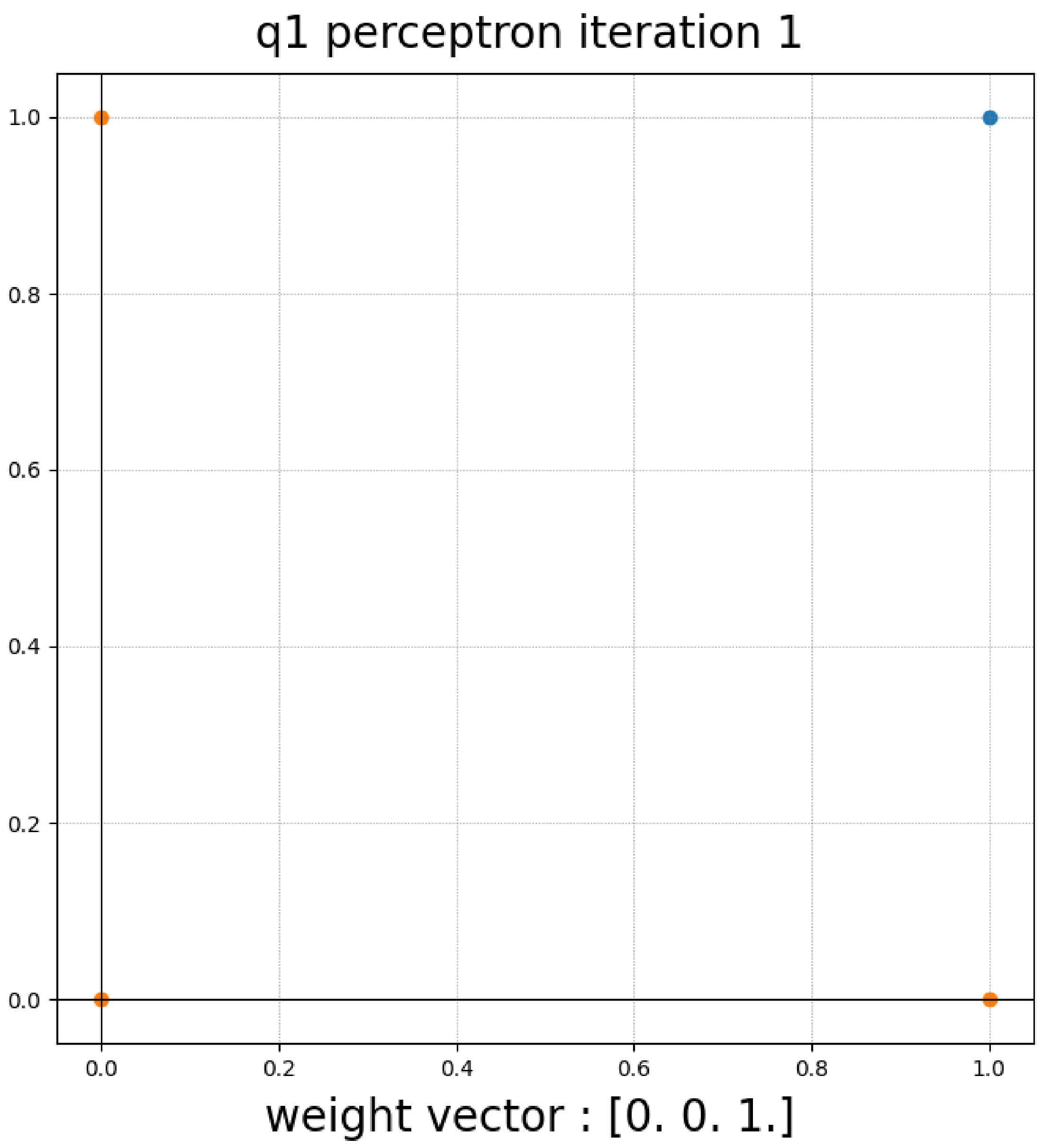


Figure 1: q1_perceptron_before_training.p

output

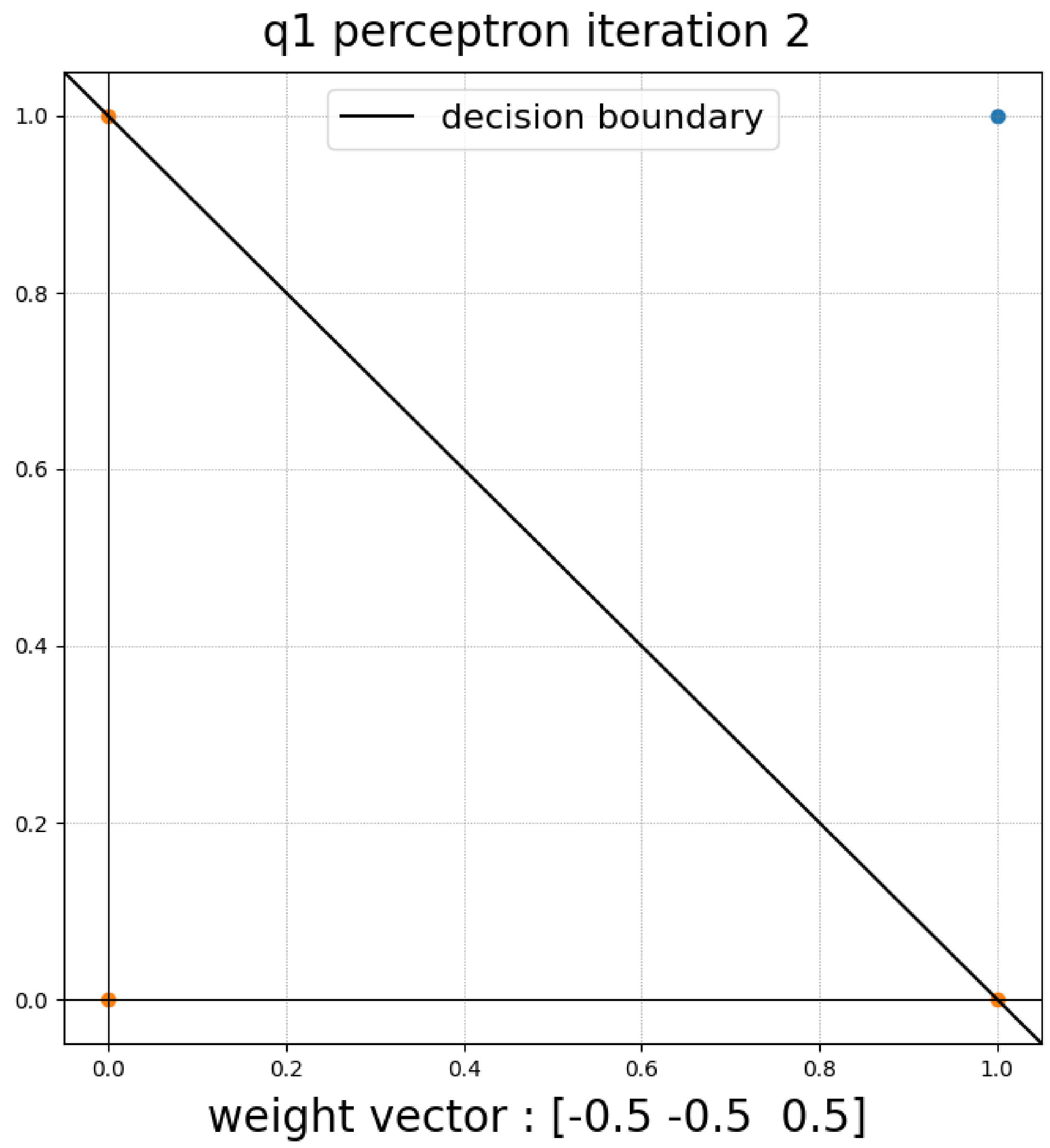
Perceptron

Iteration 1



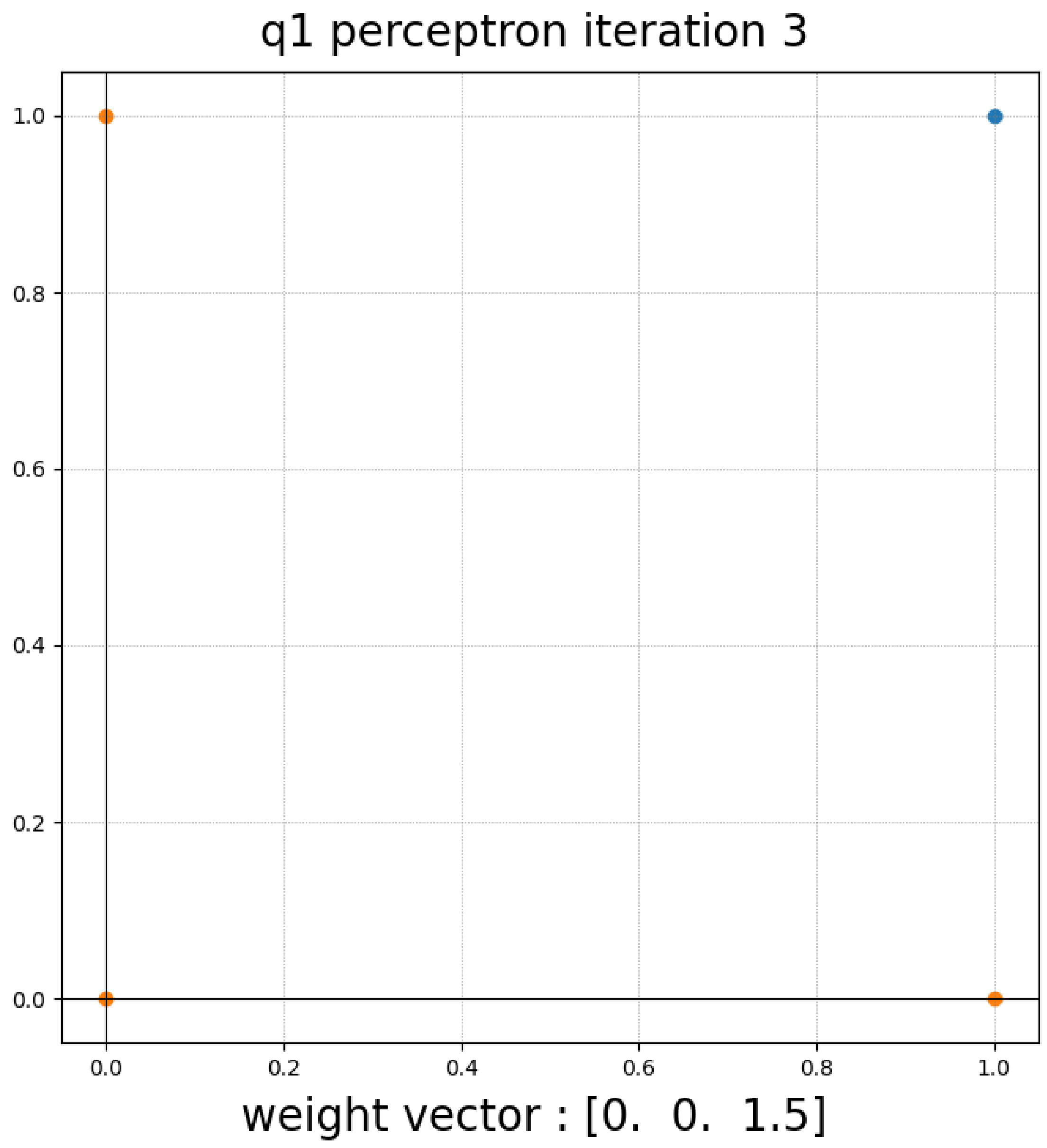
gradient of Jp = [0 0 2]
new weight = old weight + learning rate * gradient of Jp
= [0 0 0] + 0.5 * [0 0 2] = [0. 0. 1.]

Iteration 2:



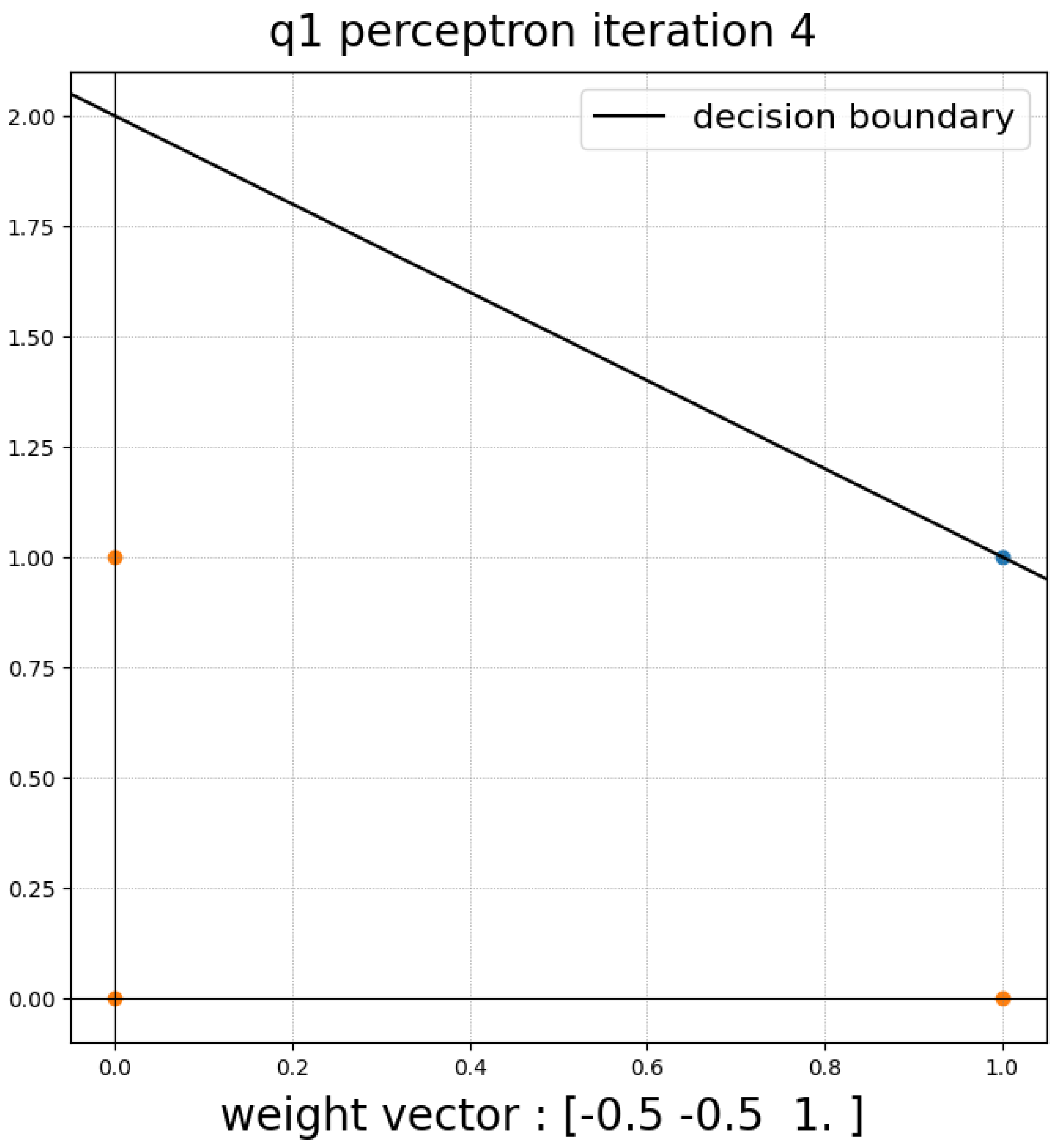
gradient of Jp = [-1. -1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [0. 0. 1.] + 0.5 * [-1. -1. -1.] = [-0.5 -0.5 0.5]

Iteration 3:



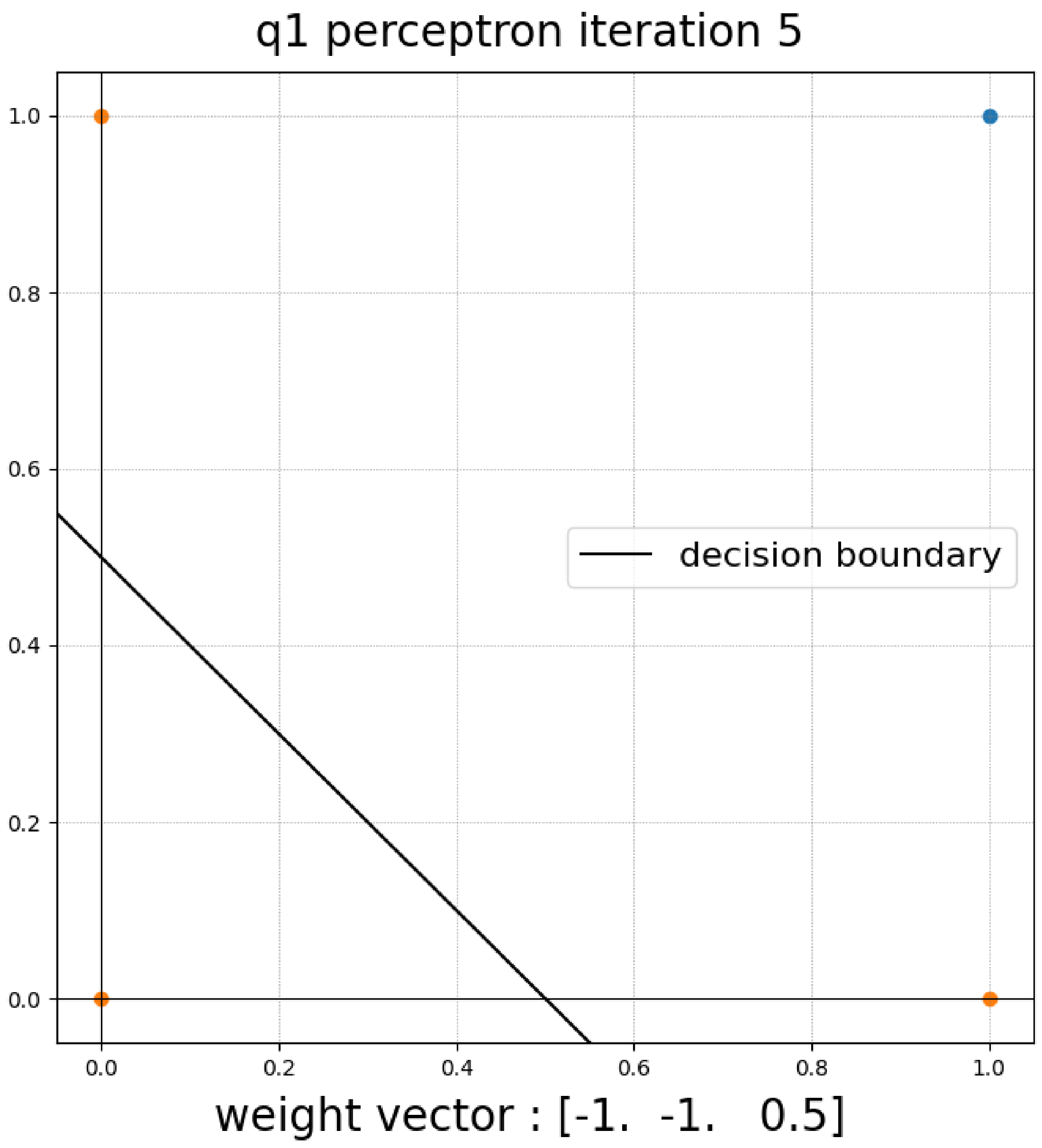
gradient of Jp = [1. 1. 2.]
new weight = old weight + learning rate * gradient of Jp
= [-0.5 -0.5 0.5] + 0.5 * [1. 1. 2.] = [0. 0. 1.5]

Iteration 4:



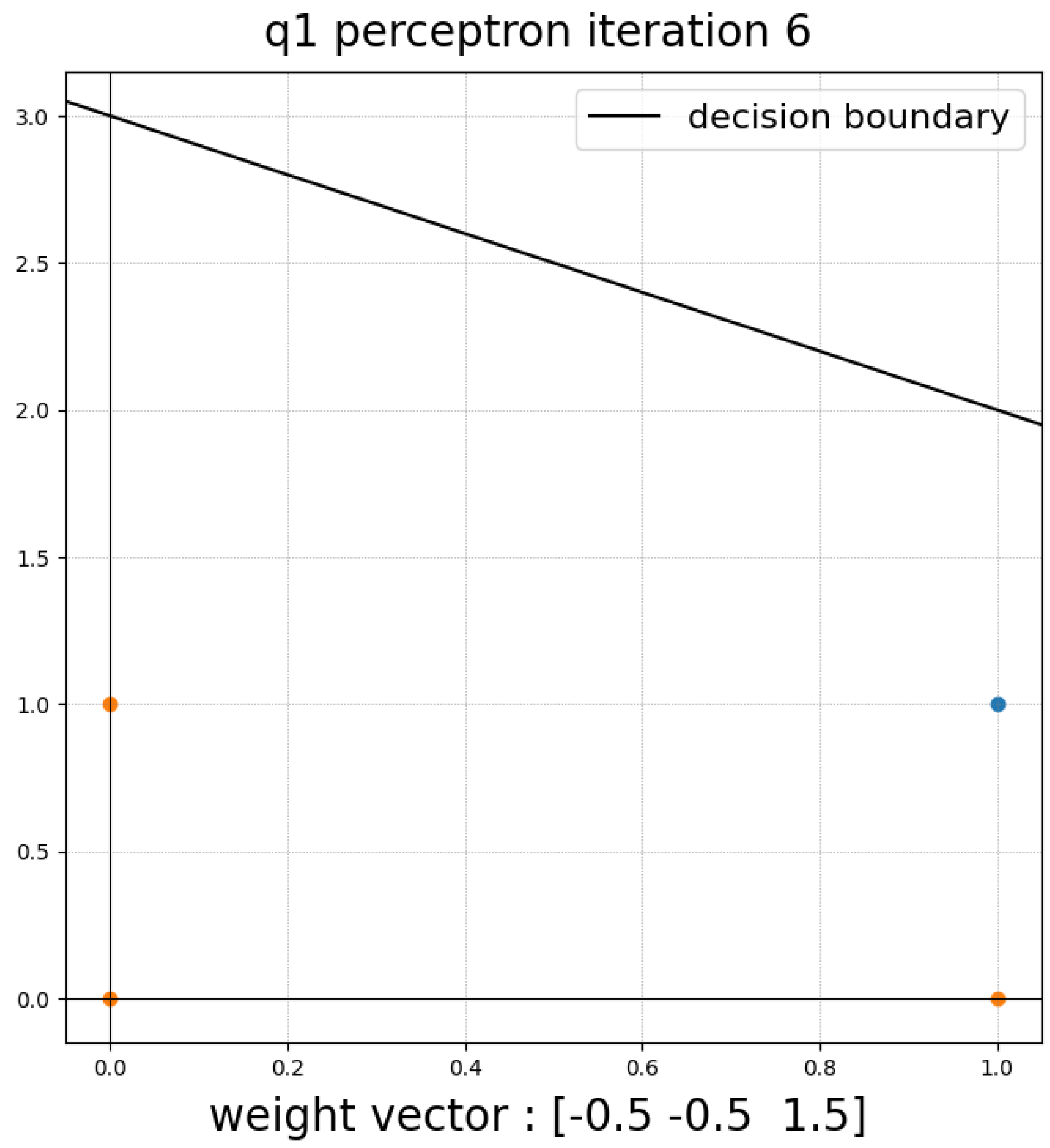
gradient of Jp = [-1. -1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [0. 0. 1.5] + 0.5 * [-1. -1. -1.] = [-0.5 -0.5 1.]

Iteration 5:



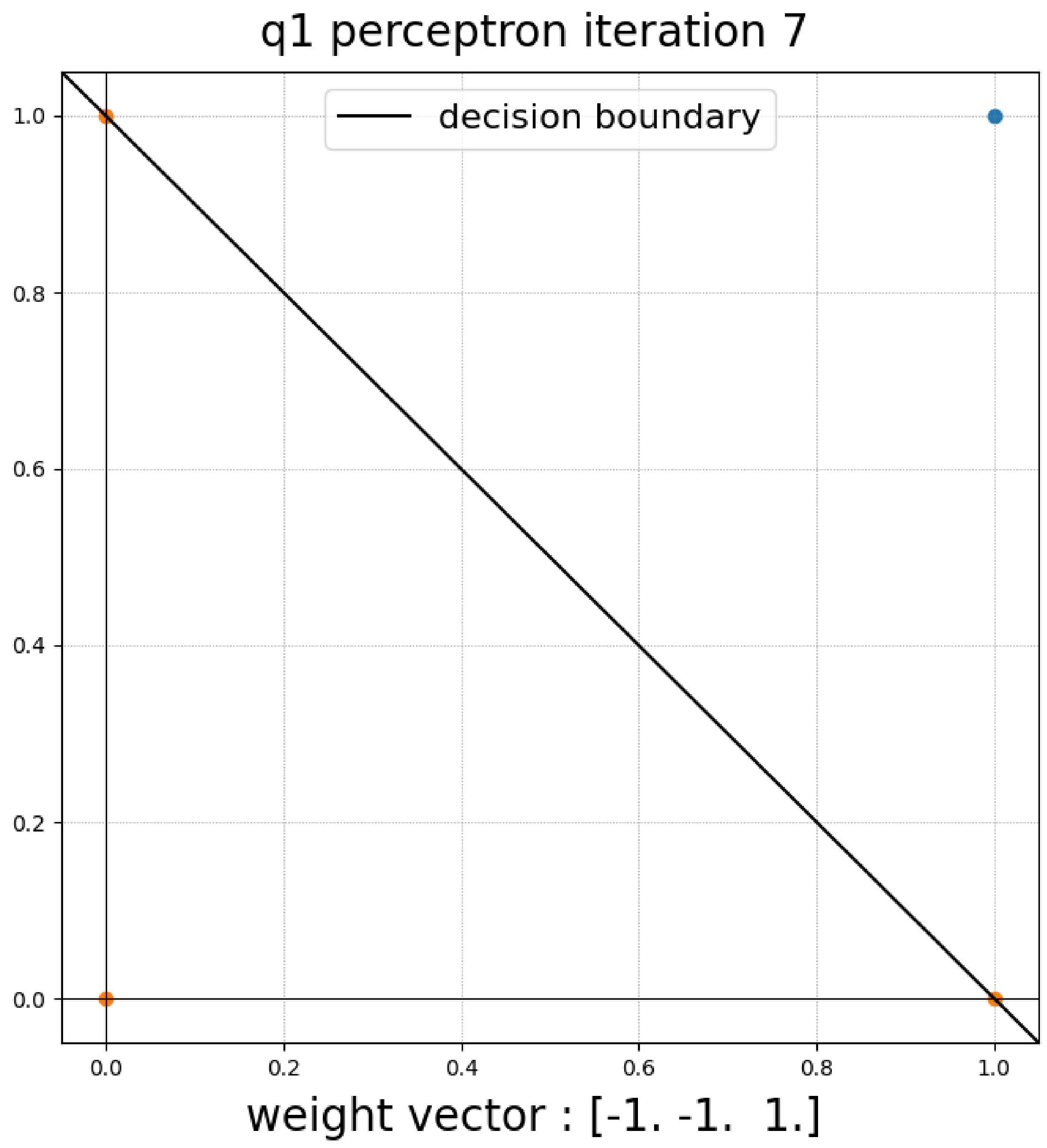
gradient of Jp = [-1. -1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [-0.5 -0.5 1.] + 0.5 * [-1. -1. -1.] = [-1. -1. 0.5]

Iteration 6:



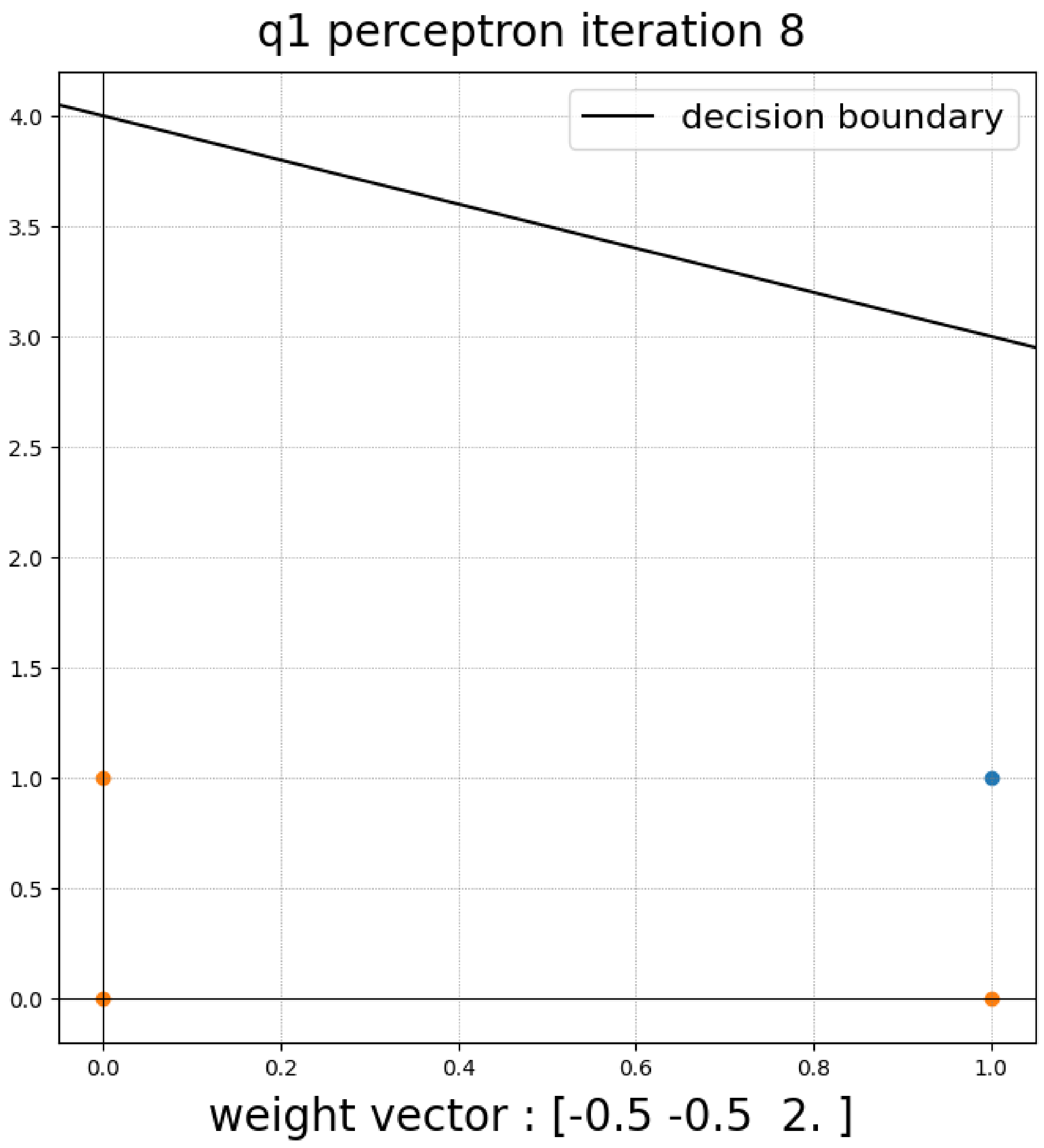
gradient of Jp = [1. 1. 2.]
new weight = old weight + learning rate * gradient of Jp
= [-1. -1. 0.5] + 0.5 * [1. 1. 2.] = [-0.5 -0.5 1.5]

Iteration 7:



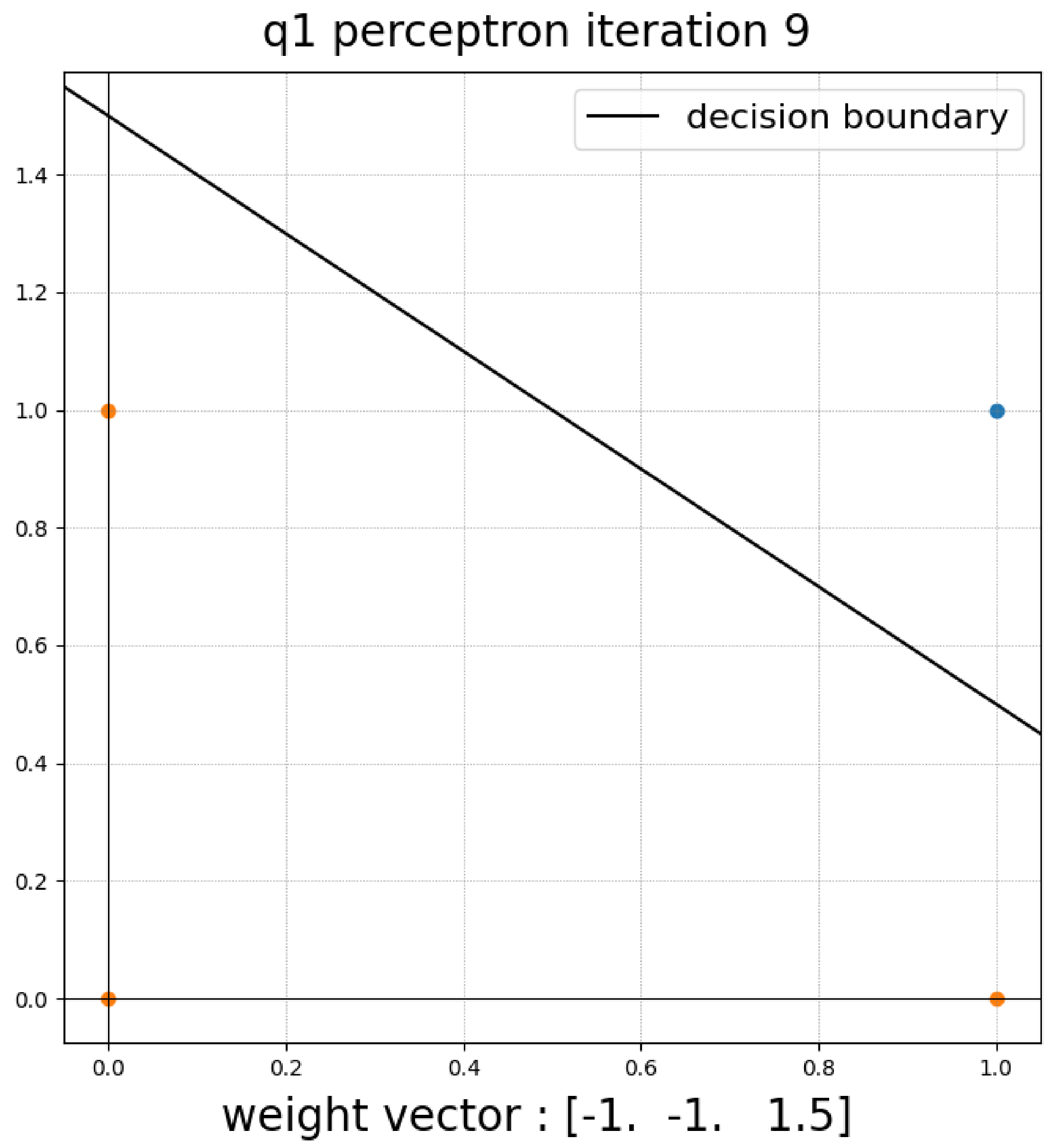
gradient of Jp = [-1. -1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [-0.5 -0.5 1.5] + 0.5 * [-1. -1. -1.] = [-1. -1. 1.]

Iteration 8:



gradient of Jp = [1. 1. 2.]
new weight = old weight + learning rate * gradient of Jp
= [-1. -1. 1.] + 0.5 * [1. 1. 2.] = [-0.5 -0.5 2.]

Iteration 9:



gradient of Jp = [-1. -1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [-0.5 -0.5 2.] + 0.5 * [-1. -1. -1.] = [-1. -1. 1.5]

Iteration 10:

gradient of Jp = [0. 0. 0.]
new weight = old weight + learning rate * gradient of Jp
= [-1. -1. 1.5] + 0.5 * [0. 0. 0.] = [-1. -1. 1.5]
As there is no significant change in the new weight vector after this iteration, let us stop the training process.

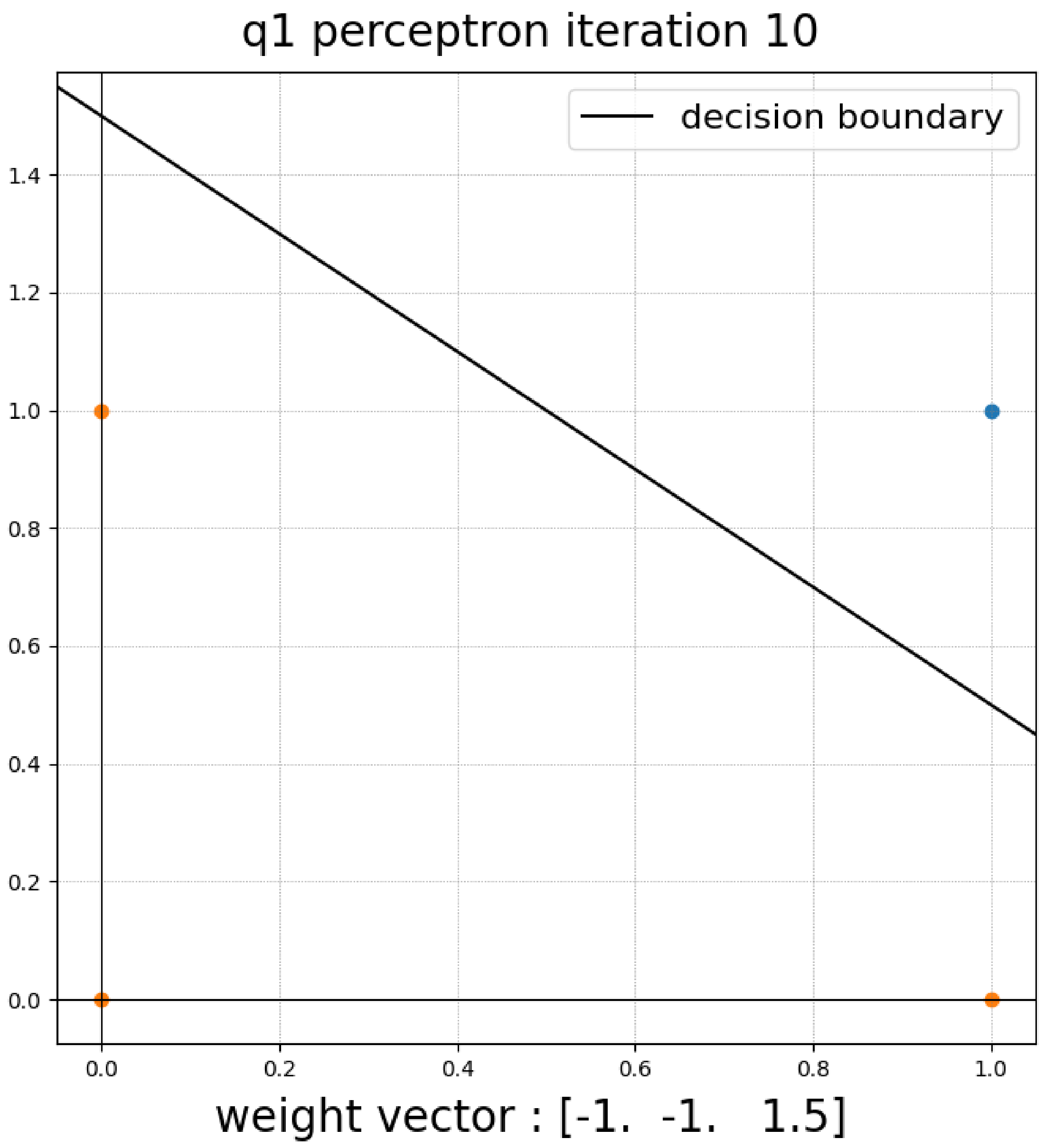


Figure 2: q1__perceptron__iteration__10.p

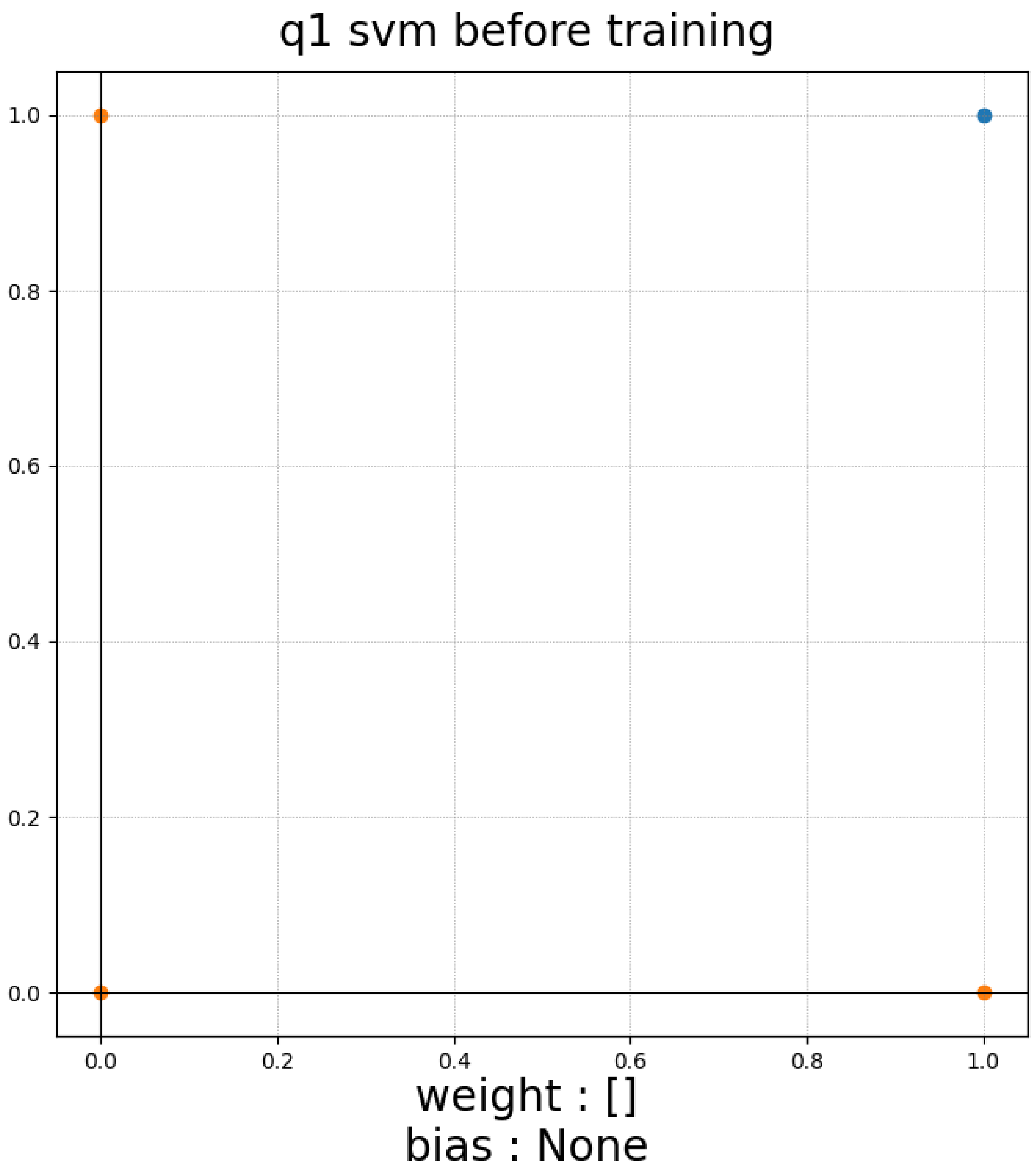
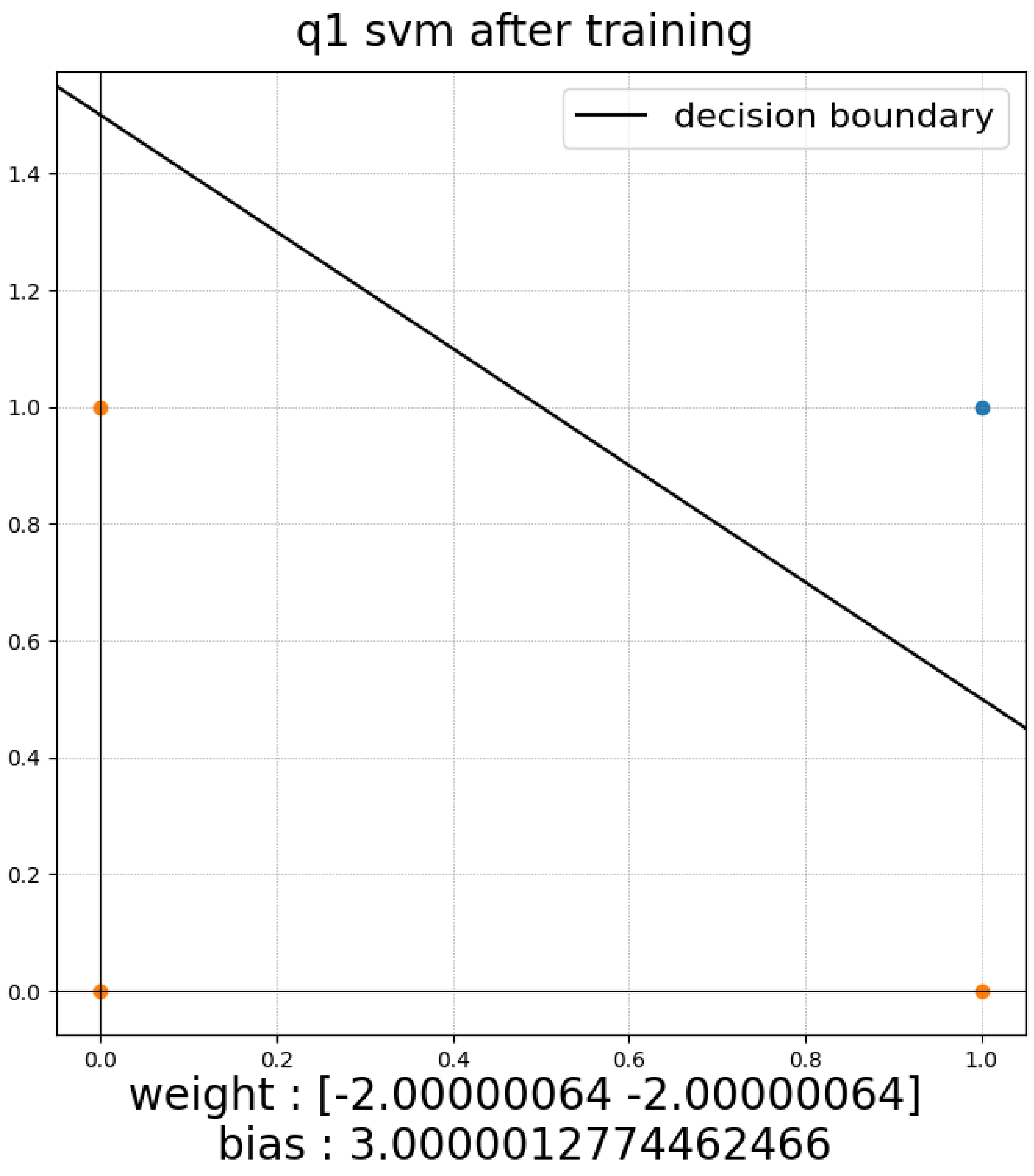


Figure 3: q1__svm__before__training.p

SVM



After training, w = [-2 -2] and bias = 3 approximately

Question 2

Train a **single perceptron and SVM** to learn the two classes in the following table.

x1	x2	w
2	2	1
-1	-3	0
-1	2	1
0	-1	0
1	3	1

x1	x2	w
-1	-2	0
1	-2	0
-1	-1	1

where x1 and x2 are the inputs and w is the target class. Assume that all the weights of the perceptron are initialized as 0 with learning rate 0.01 and 0.5 separately. Also, tabulate the number of iterations required to converge the perception algorithm with these two learning rates.

code

```
import perceptron, svm
import numpy as np

data = np.array([[2,2],[-1,-3],[-1,2],[0,-1],[1,3],[-1,-2],[1,-2],[-1,-1]])
perc_label = np.array([1,  0, 1,  0, 1,  0,  0, 1])
svm_label  = np.array([1, -1, 1, -1, 1, -1, -1, 1])

svm.demo(data,svm_label,plot_title="q2")

print('learning rate = 0.01\n')
perceptron.demo(data,perc_label,learning_rate=0.01,plot_title="q2 lr1")
print('\nlearning rate = 0.5\n')
perceptron.demo(data,perc_label,learning_rate=0.5,plot_title="q2 lr2")
```

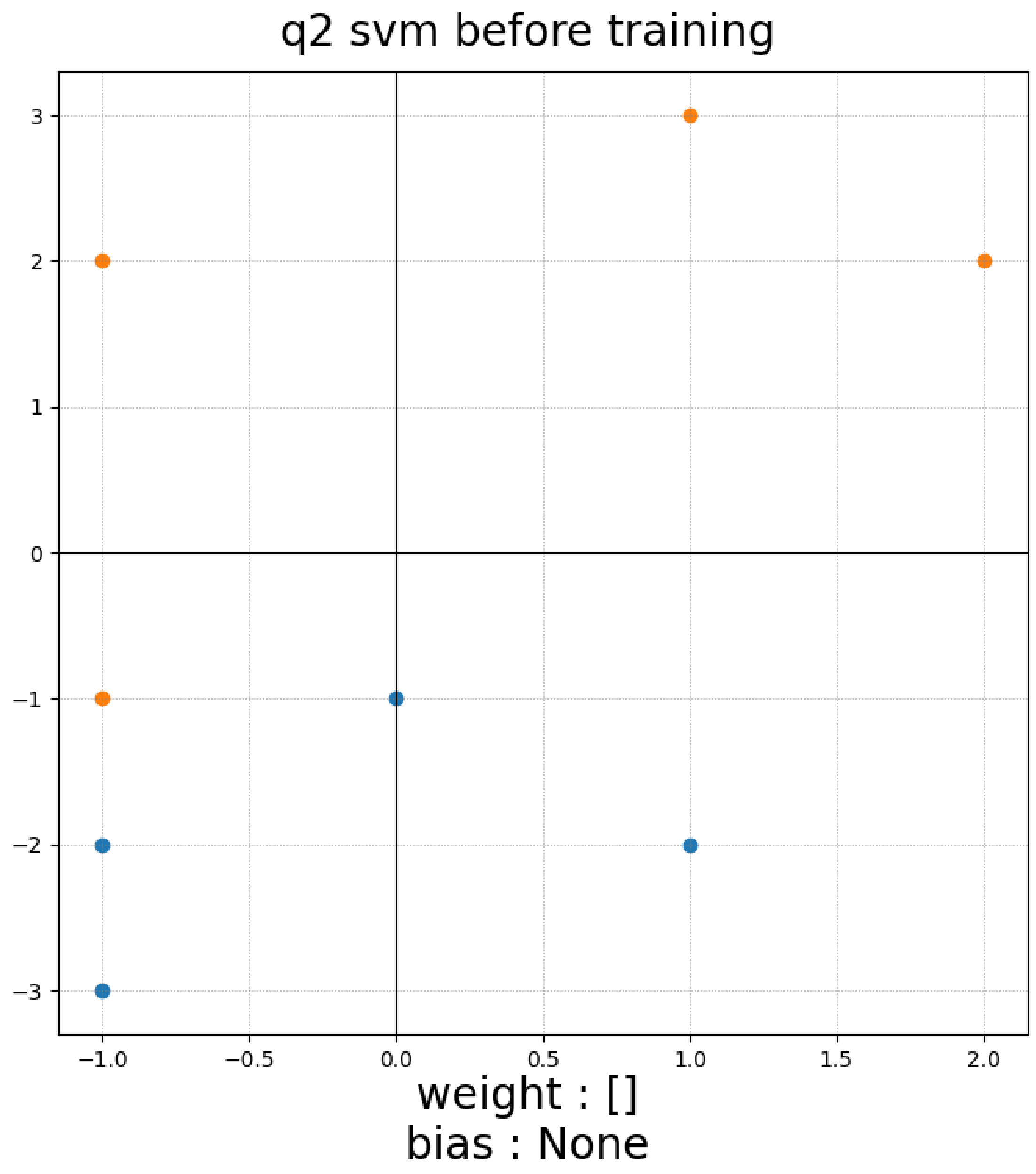
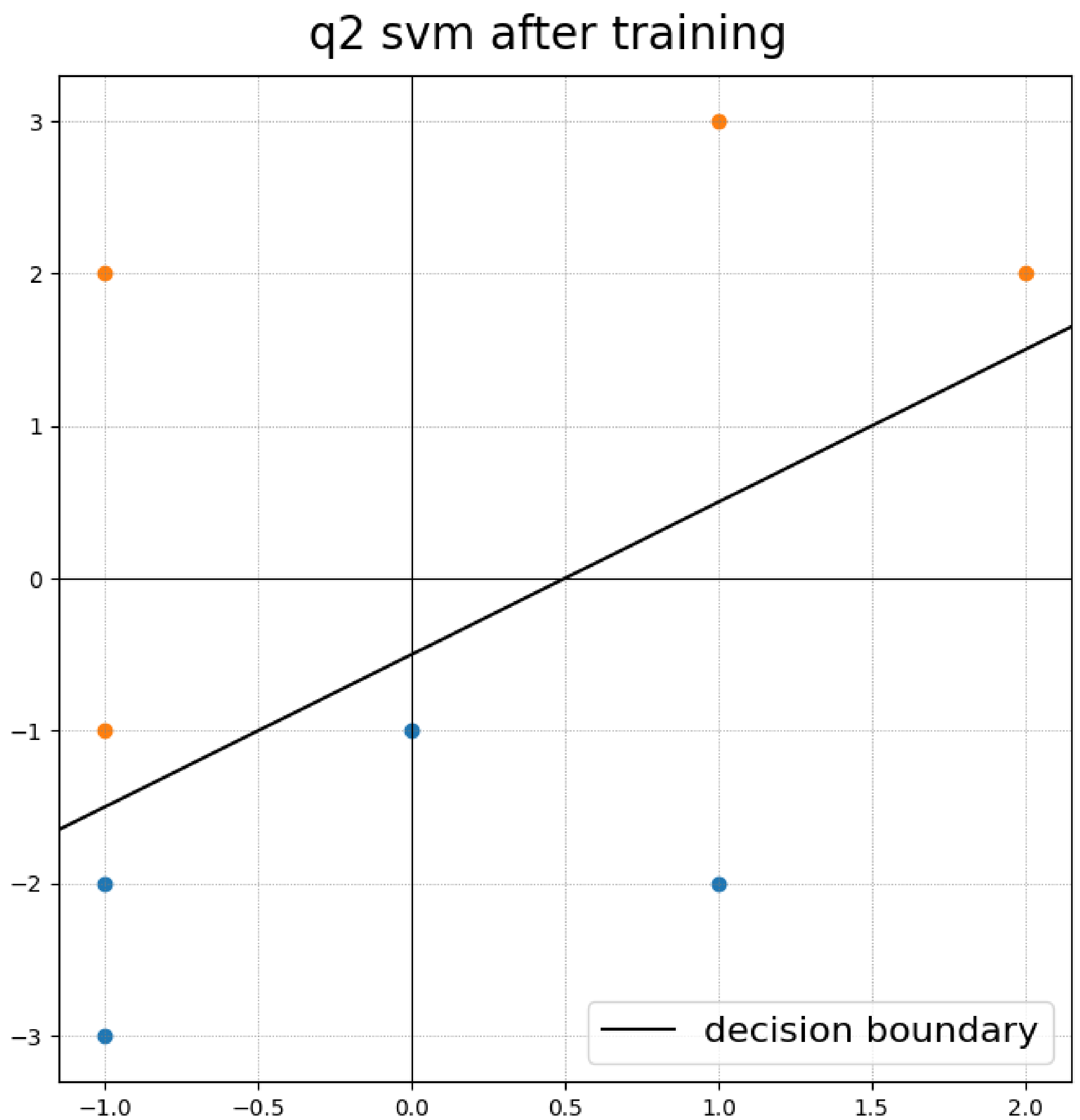


Figure 4: q2__svm__before__training.p

output

SVM



```
weight : [-2.00000023  2.00000031]
bias : 1.0000000792363046
```

After training, w = [-2 2] and bias = 1 approximately

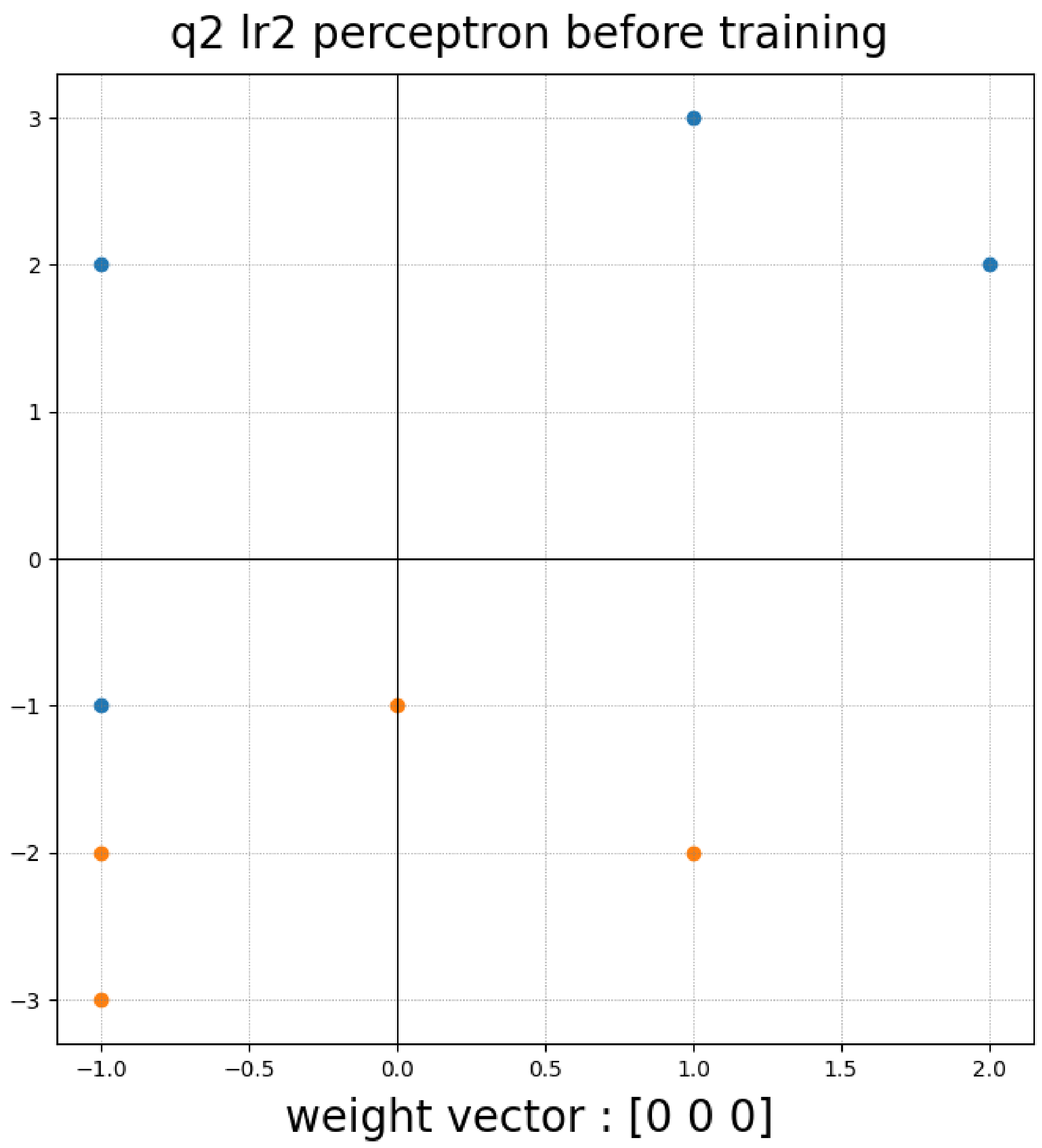
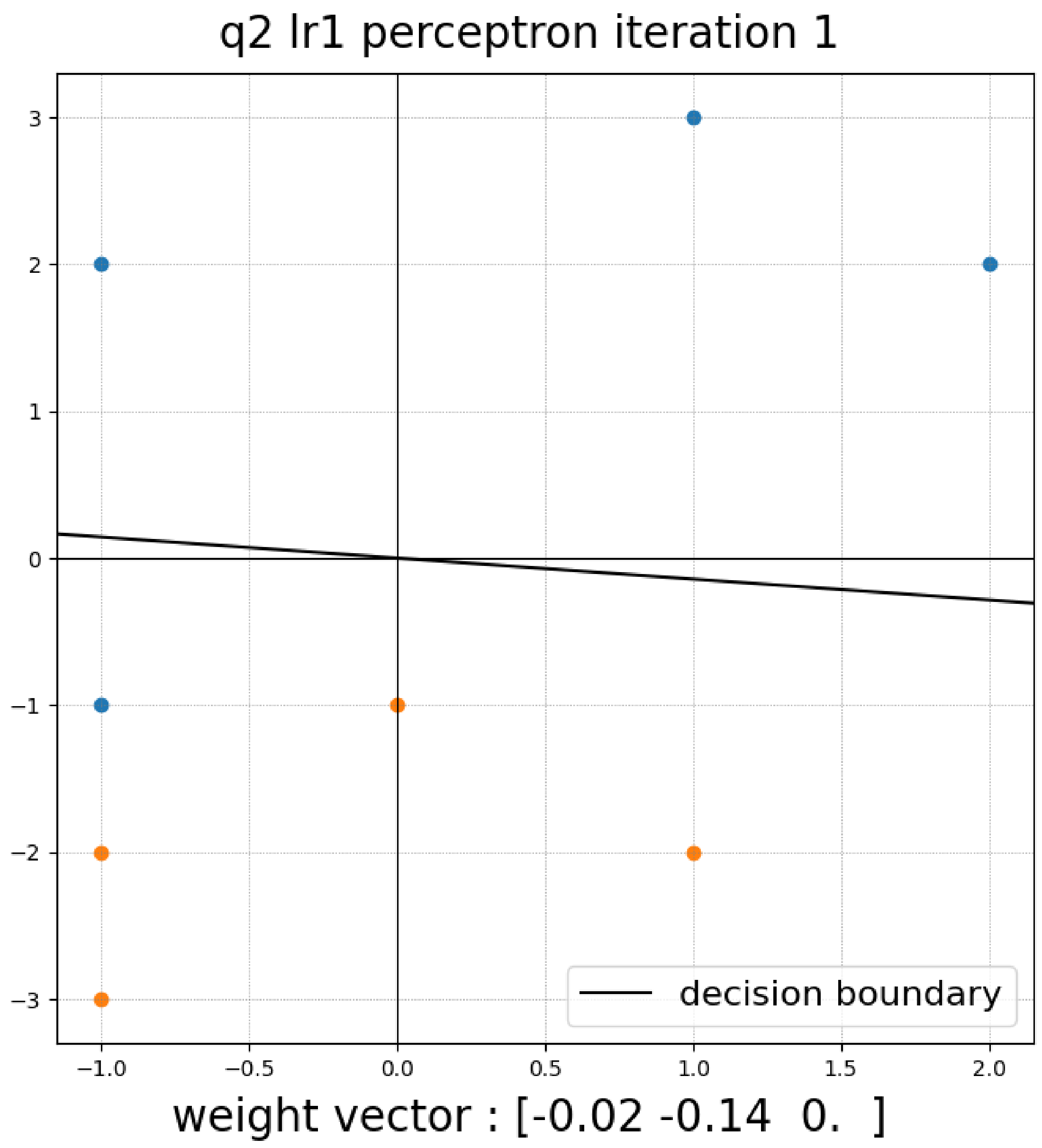


Figure 5: q2__perceptron__before__training.p

Perceptron

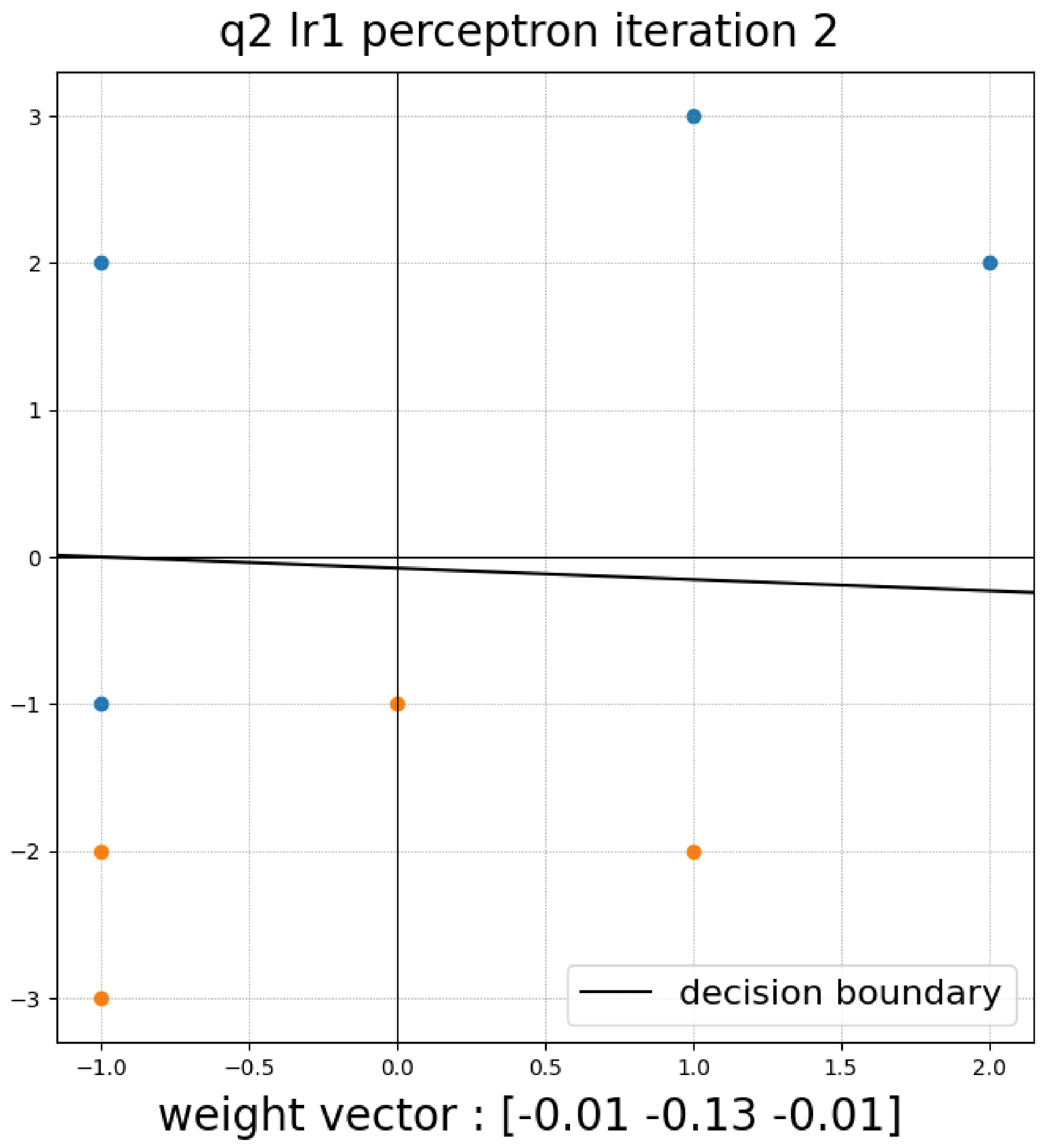
learning rate = 0.01

Iteration 1:



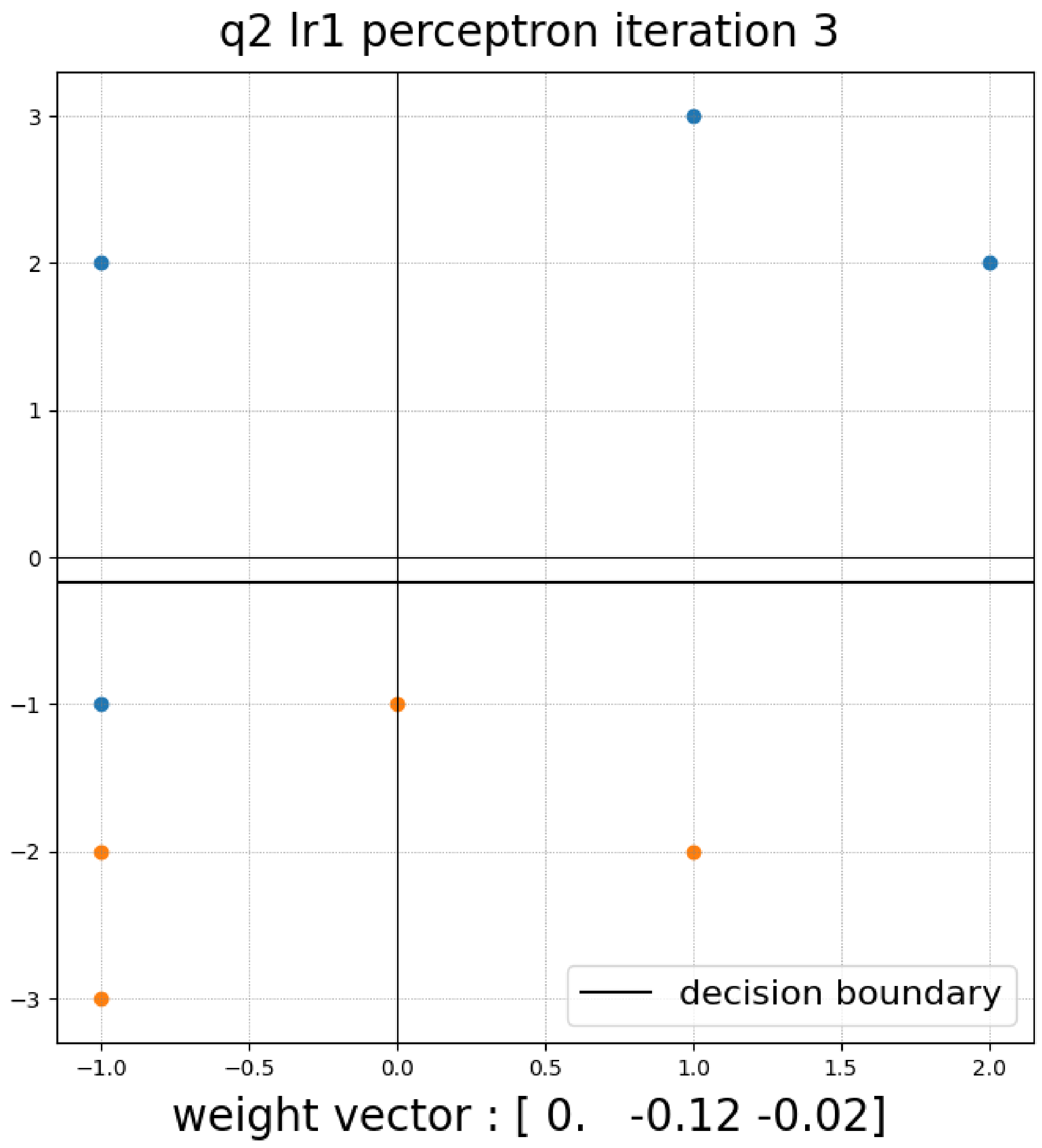
gradient of Jp = [-2 -14 0] new weight = old weight + learning rate * gradient of Jp = [0 0 0] + 0.01 * [-2 -14 0] = [-0.02 -0.14 0.]

Iteration 2:



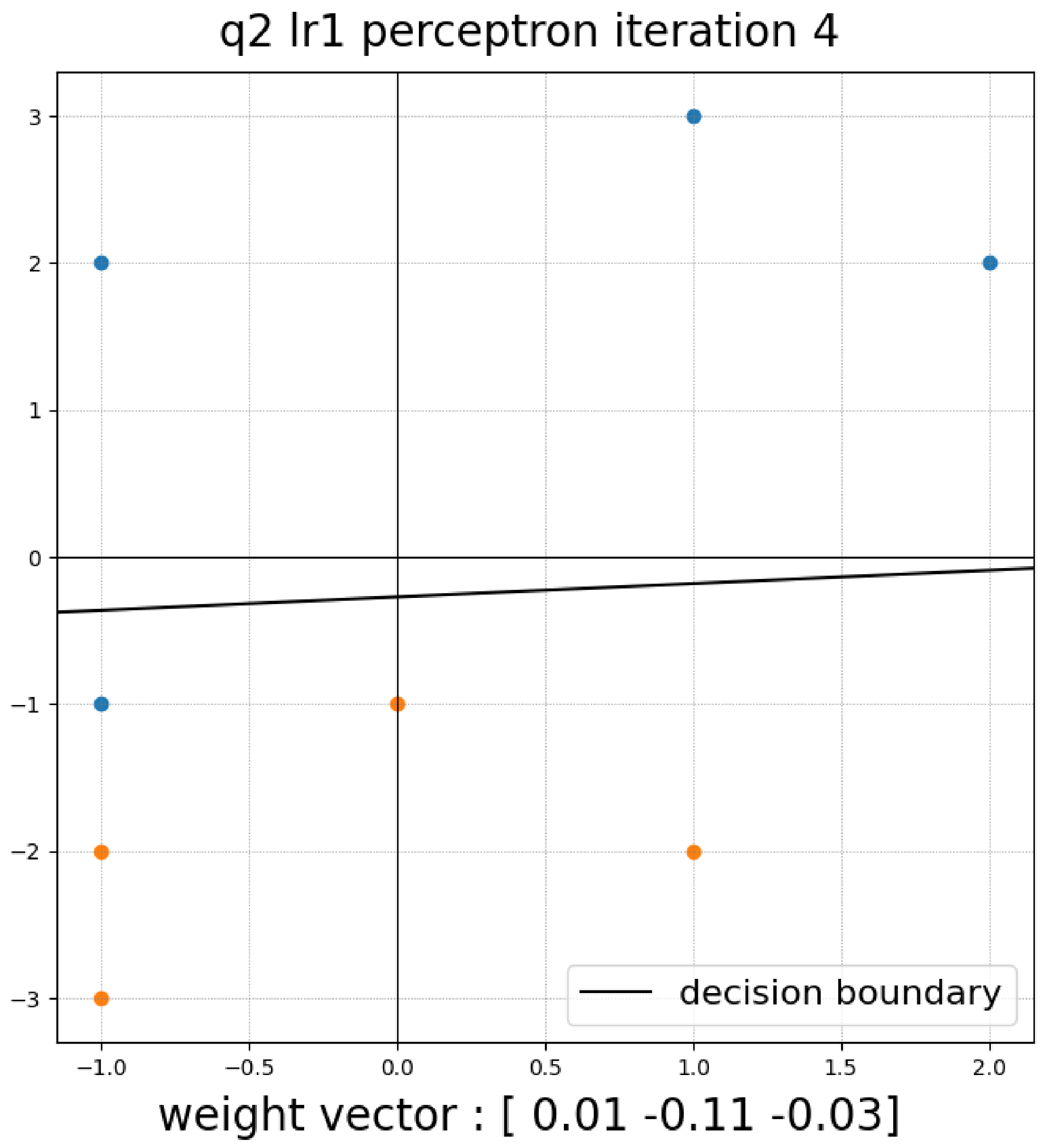
gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [-0.02 -0.14 0.] + 0.01 * [1. 1. -1.]
= [-0.01 -0.13 -0.01]

Iteration 3:



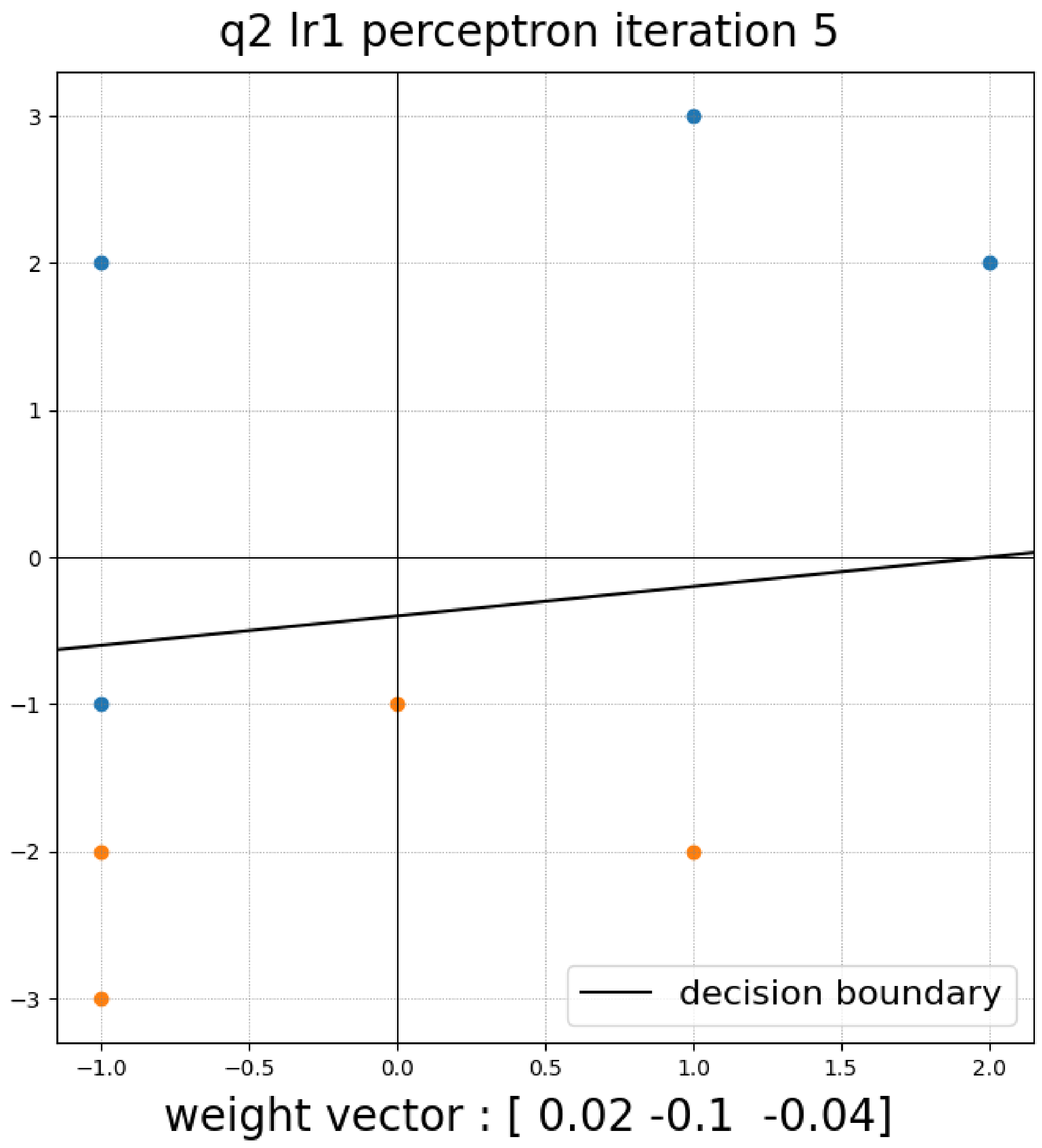
gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [-0.01 -0.13 -0.01] + 0.01 * [1. 1. -1.]
= [0. -0.12 -0.02]

Iteration 4:



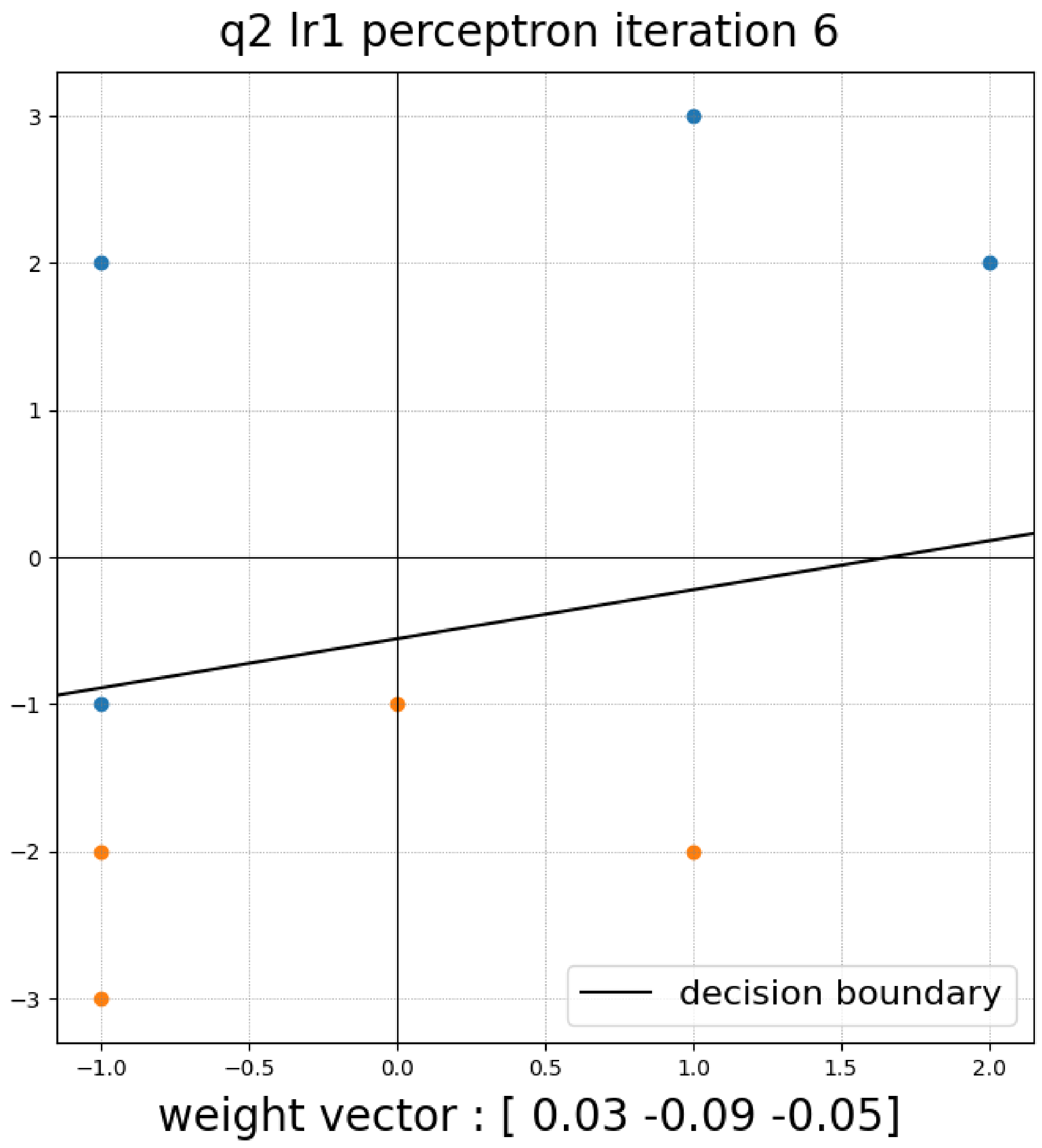
gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [0. -0.12 -0.02] + 0.01 * [1. 1. -1.]
= [0.01 -0.11 -0.03]

Iteration 5:



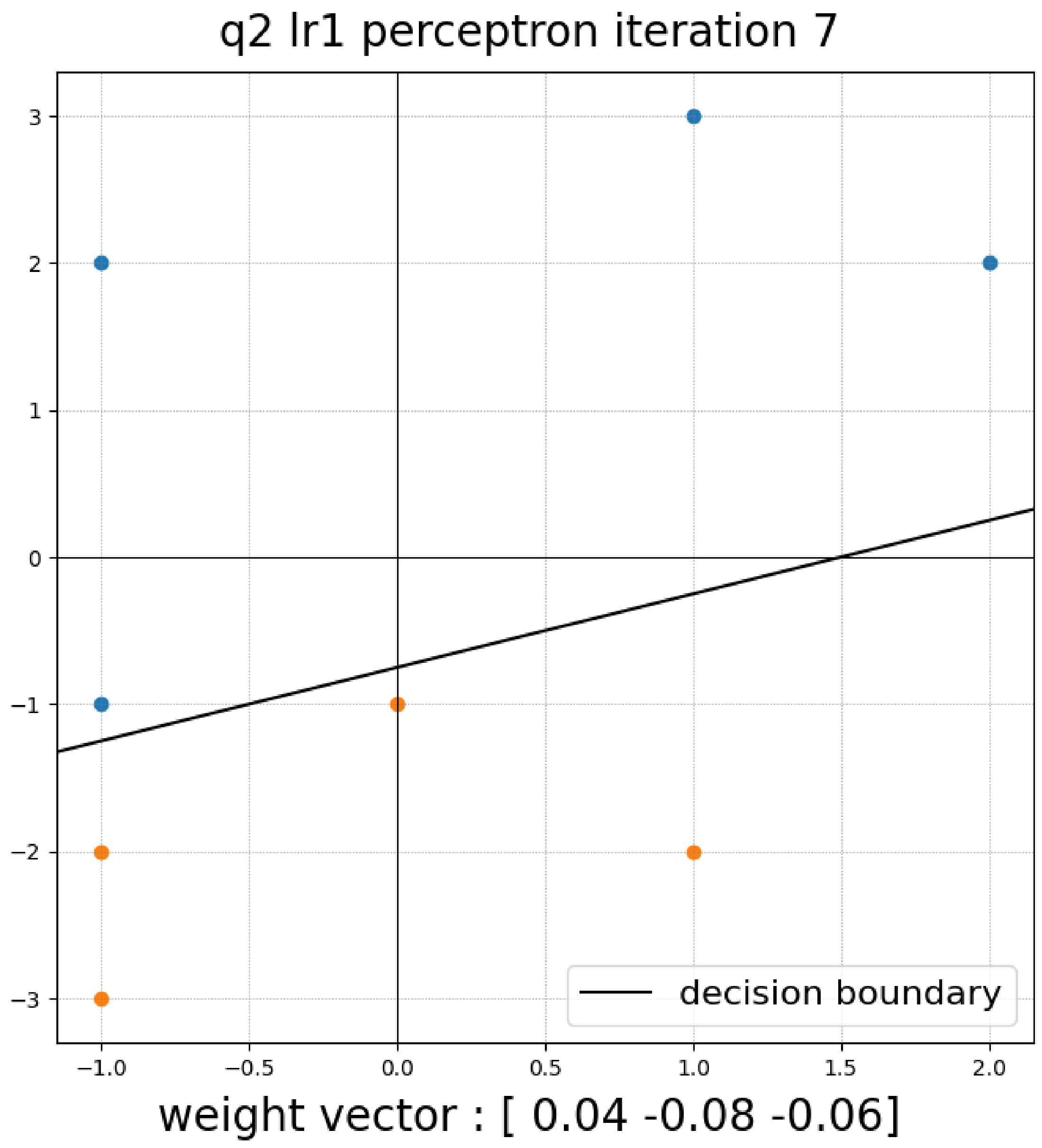
gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [0.01 -0.11 -0.03] + 0.01 * [1. 1. -1.]
= [0.02 -0.1 -0.04]

Iteration 6:



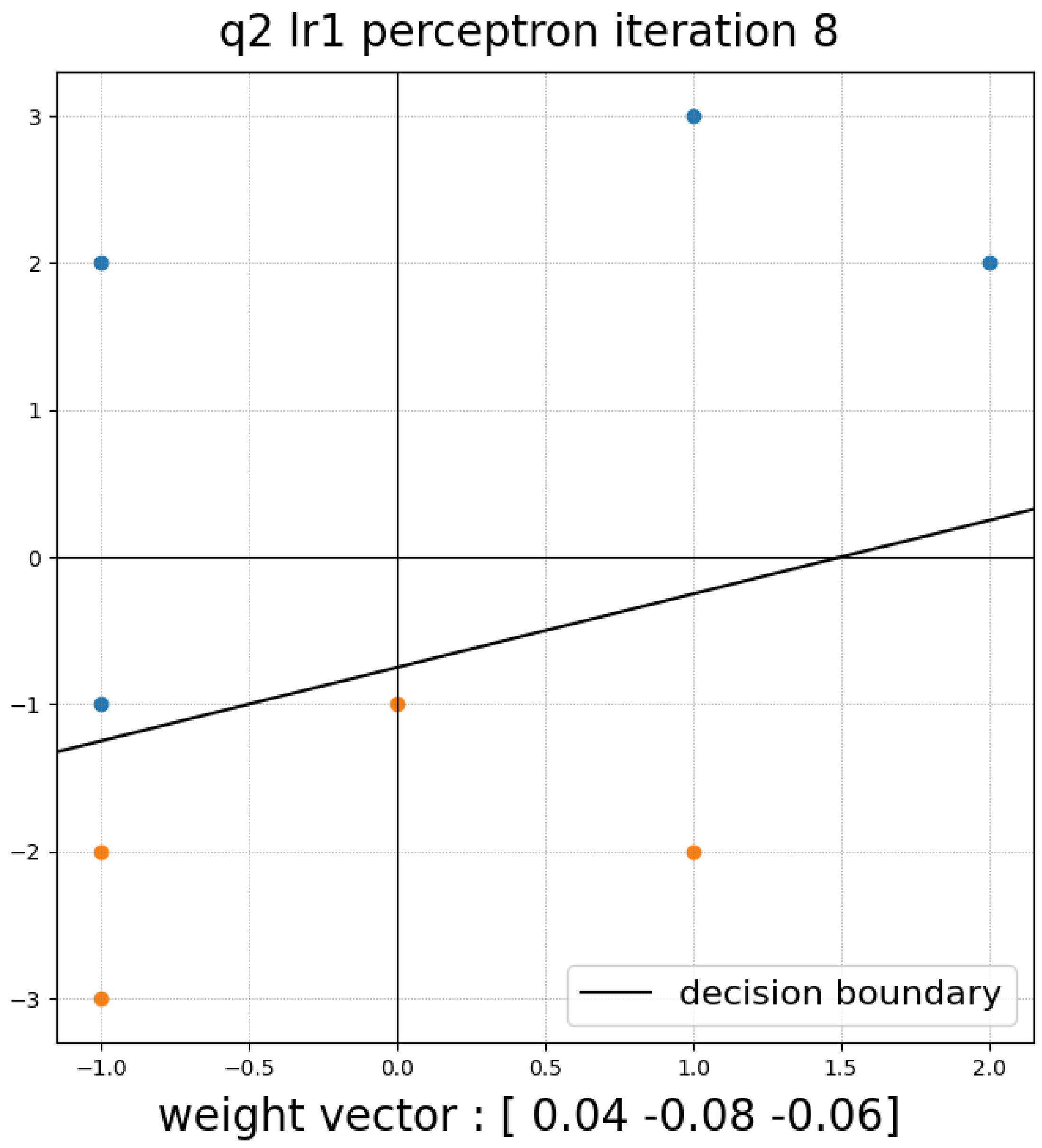
gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [0.02 -0.1 -0.04] + 0.01 * [1. 1. -1.]
= [0.03 -0.09 -0.05]

Iteration 7:



gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [0.03 -0.09 -0.05] + 0.01 * [1. 1. -1.]
= [0.04 -0.08 -0.06]

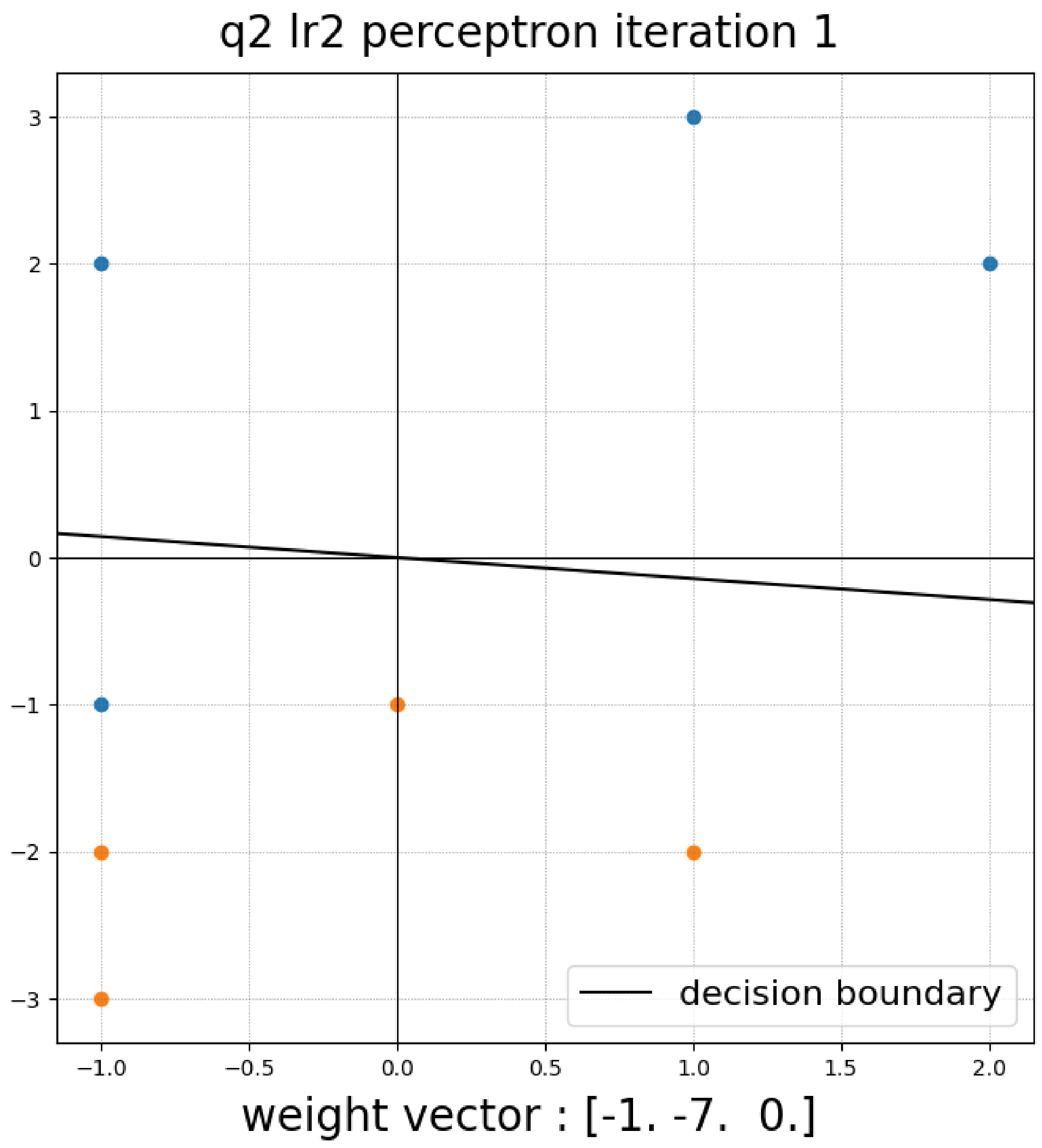
Iteration 8:



gradient of Jp = [0. 0. 0.]
new weight = old weight + learning rate * gradient of Jp
= [0.04 -0.08 -0.06] + 0.01 * [0. 0. 0.]
= [0.04 -0.08 -0.06]
As there is no significant change in the new weight vector after this iteration, let us stop the training process.

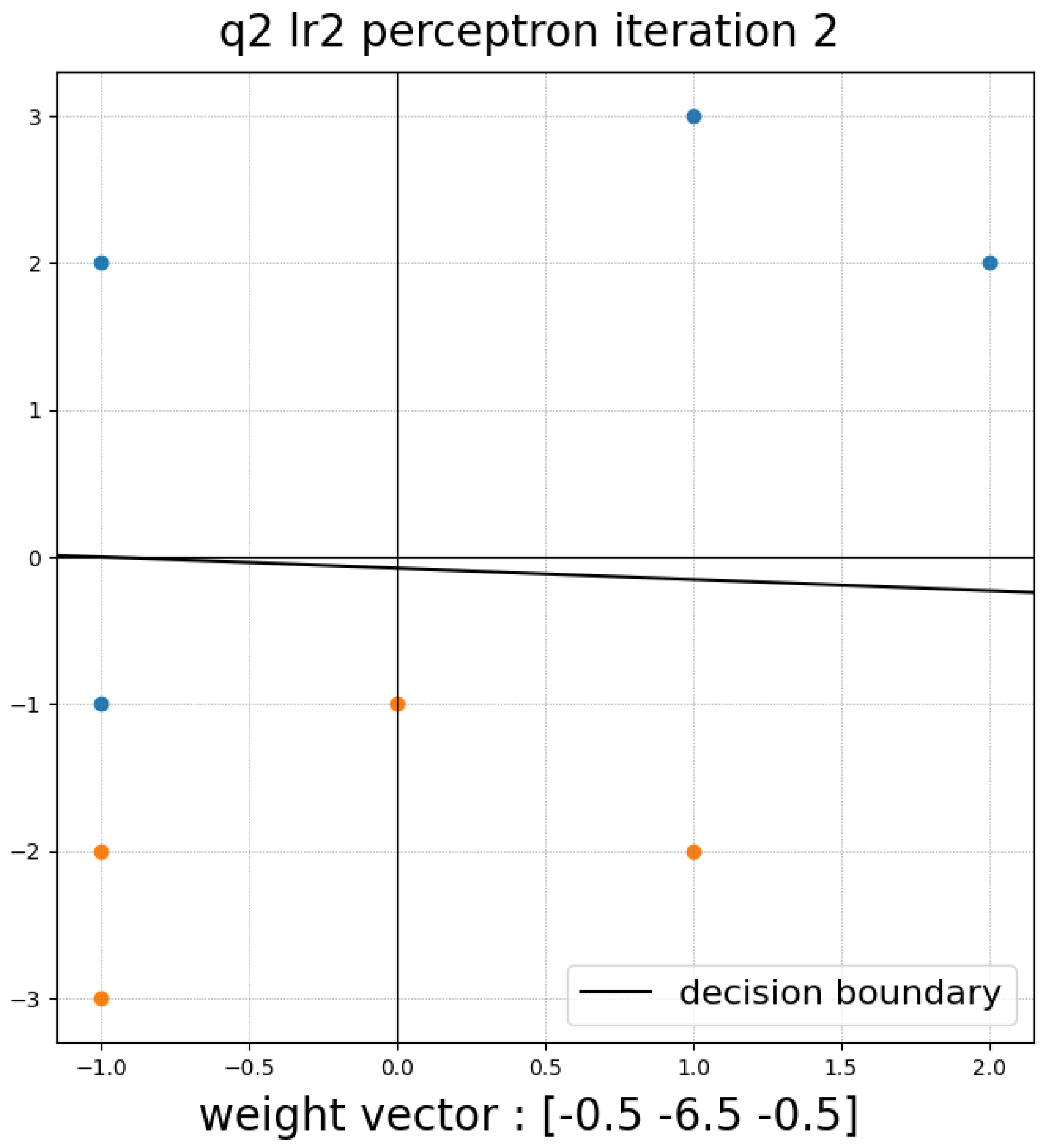
learning rate = 0.5

Iteration 1:



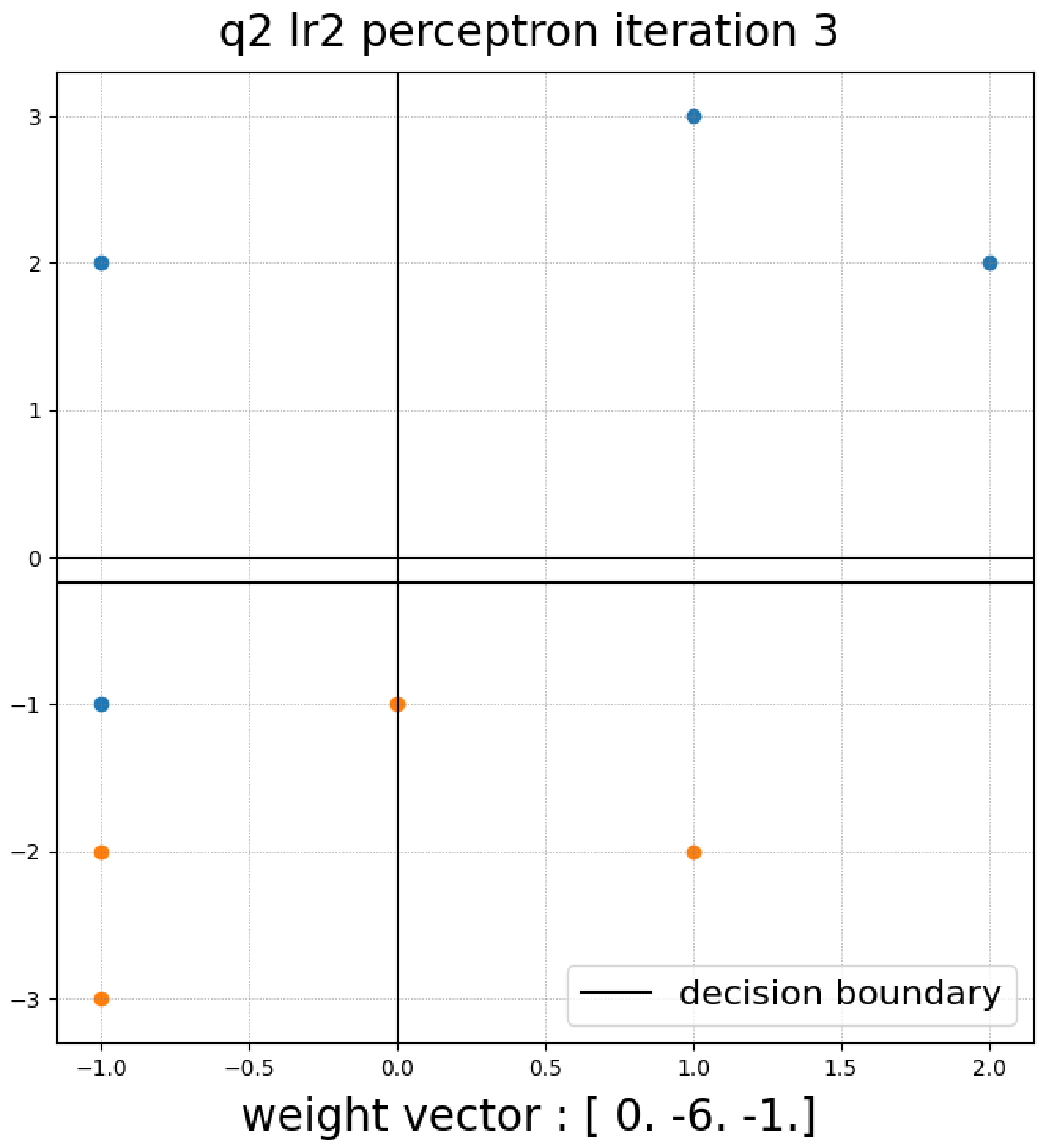
gradient of Jp = [-2 -14 0]
new weight = old weight + learning rate * gradient of Jp
= [0 0 0] + 0.5 * [-2 -14 0]
= [-1. -7. 0.]

Iteration 2:



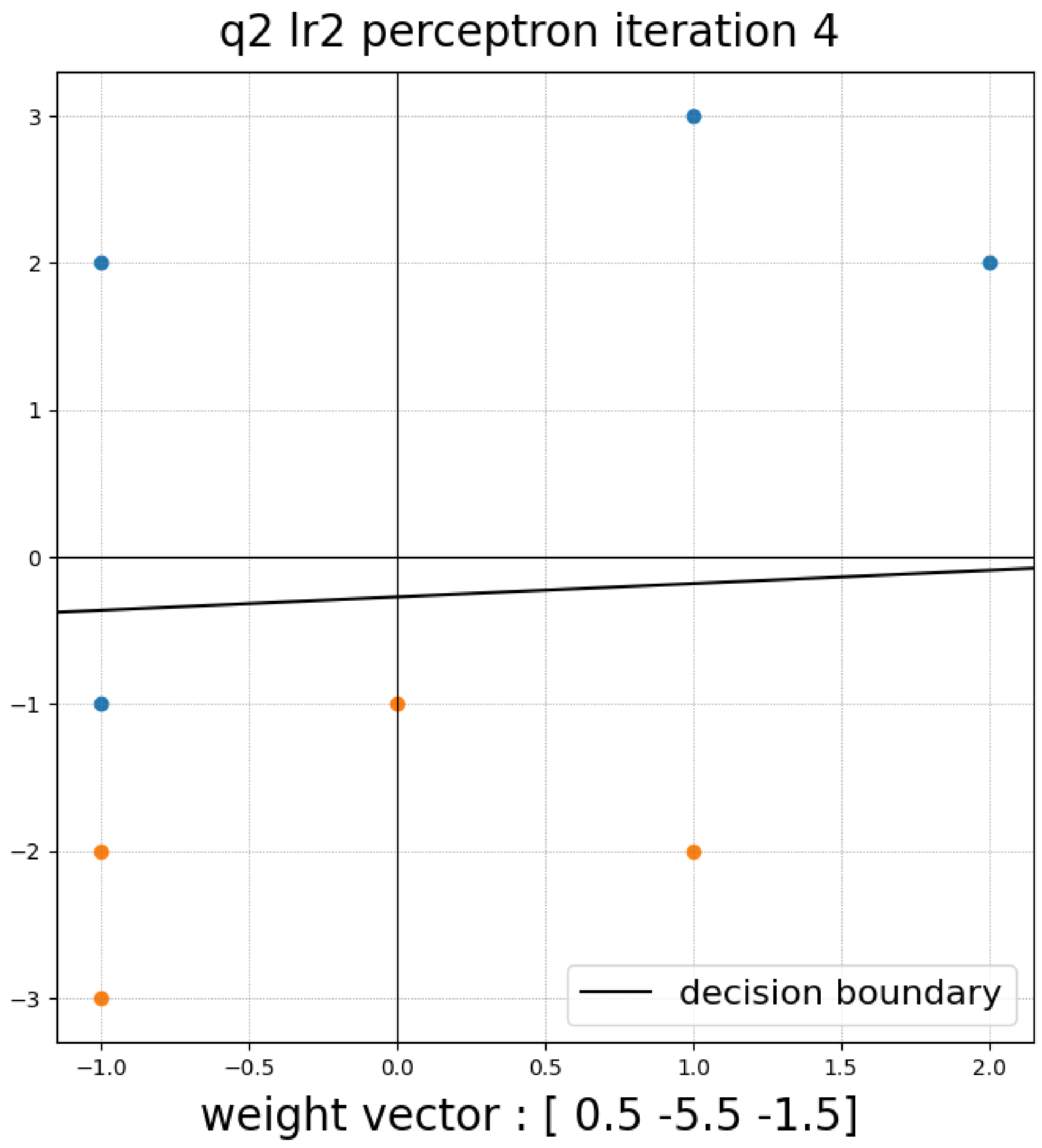
gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [-1. -7. 0.] + 0.5 * [1. 1. -1.]
= [-0.5 -6.5 -0.5]

Iteration 3:



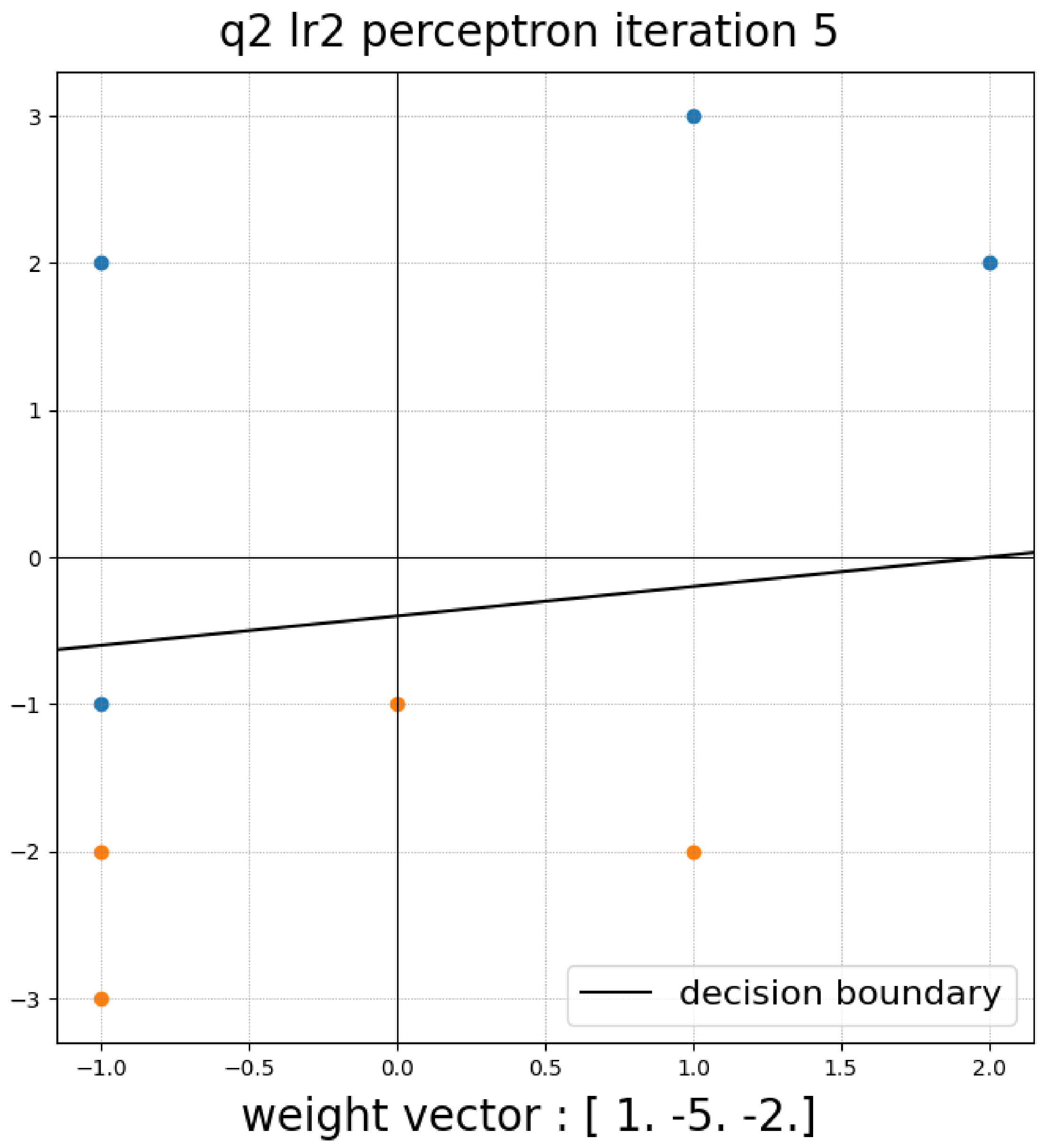
gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [-0.5 -6.5 -0.5] + 0.5 * [1. 1. -1.]
= [0. -6. -1.]

Iteration 4:



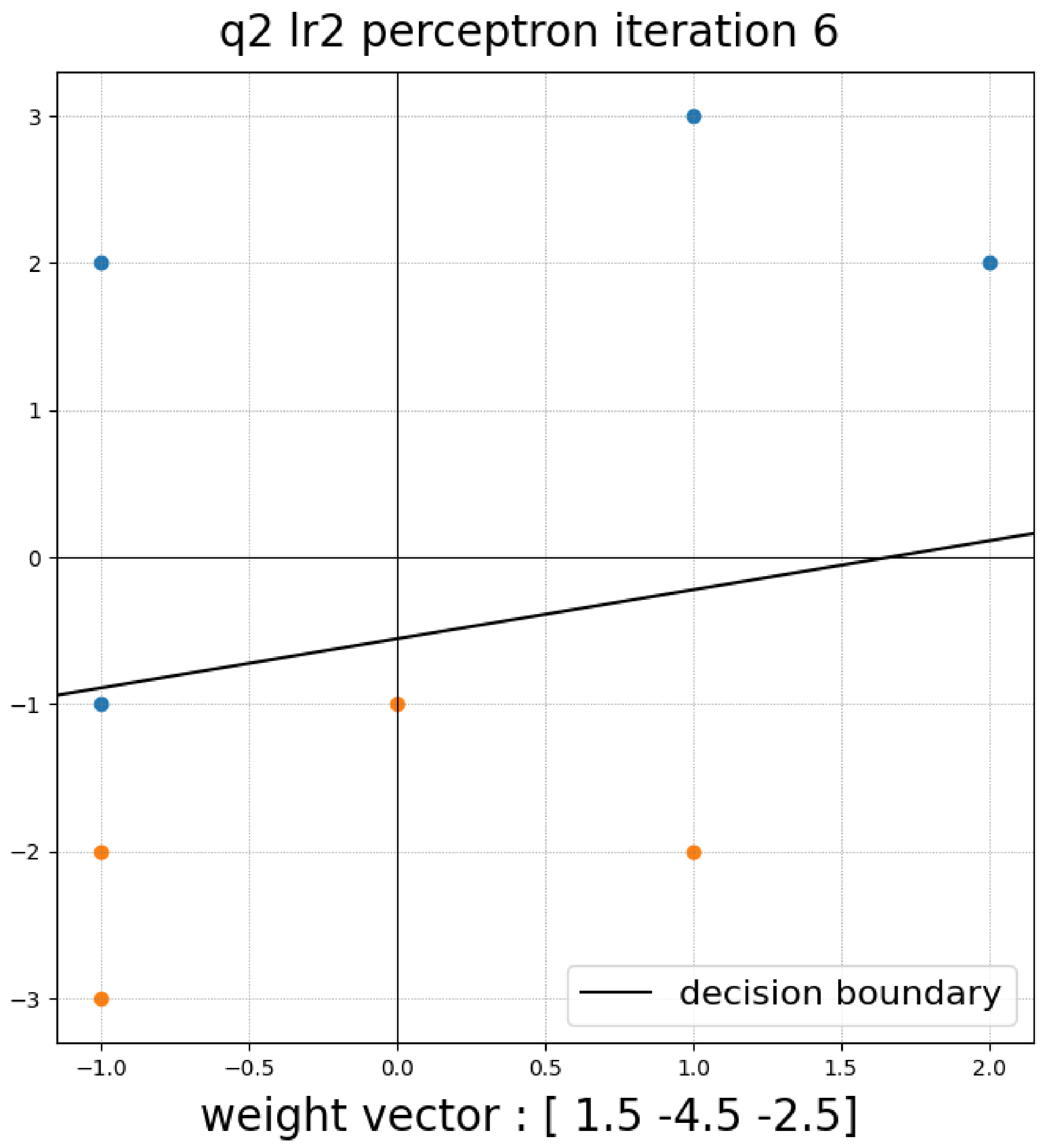
gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [0. -6. -1.] + 0.5 * [1. 1. -1.]
= [0.5 -5.5 -1.5]

Iteration 5:



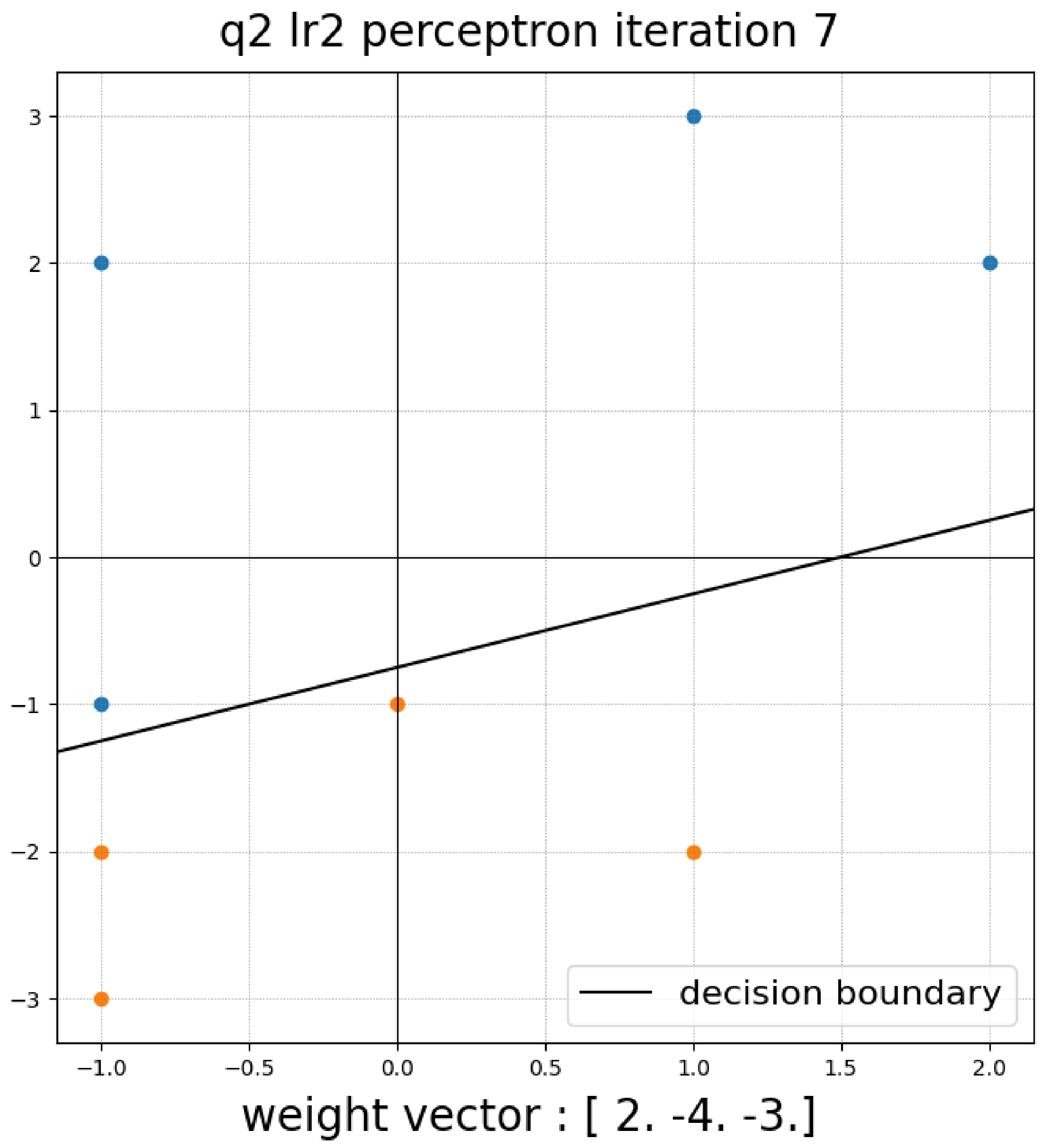
gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [0.5 -5.5 -1.5] + 0.5 * [1. 1. -1.]
= [1. -5. -2.]

Iteration 6:



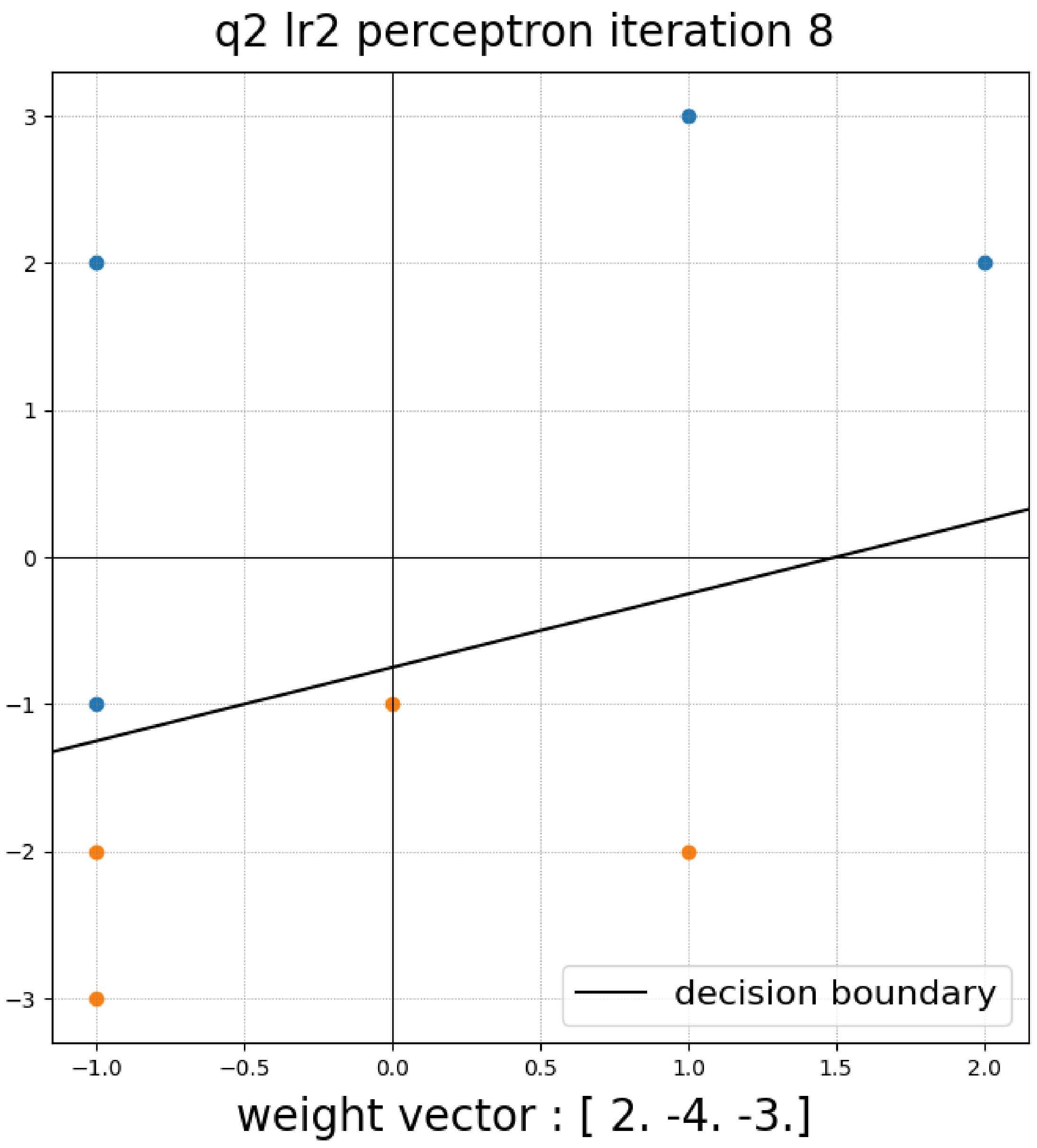
gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [1. -5. -2.] + 0.5 * [1. 1. -1.]
= [1.5 -4.5 -2.5]

Iteration 7:



gradient of Jp = [1. 1. -1.]
new weight = old weight + learning rate * gradient of Jp
= [1.5 -4.5 -2.5] + 0.5 * [1. 1. -1.]
= [2. -4. -3.]

Iteration 8:



gradient of Jp = [0. 0. 0.]
new weight = old weight + learning rate * gradient of Jp
= [2. -4. -3.] + 0.5 * [0. 0. 0.]
= [2. -4. -3.]
As there is no significant change in the new weight vector after this iteration, let us stop the training process.

Conclusion

Learning Rate	Number of Iterations Required
0.01	7
0.5	7

We would normally expect the number of iterations required to be different for different learning rates but surprisingly we got the same number of iterations for both. This is because of the fact that the initial weight vector is zero. Because of this, the gradient of Jp (summation of misclassified y) will be the same in the 1st iteration. New weights w <– w + learning rate * Jp. Clearly, Jp is same for both a nd only the learning rate differs. So, we can say that w at iter 1 with lr1 = w at i ter 1 with lr2 * some constant. (the constant is in fact lr1/lr2 = 0.01/0.5 = 0.02) Since one weight vector is the scaled version of the other, we can say that both represent the same decision boundary.
Now, the same thing happens at iteration 2 and again w at iter 1 with lr1 = w at iter 1 with lr2 * (lr1/lr2) ==> both w represent the same decision boundary. This happens at all iterations and the resultant weights at corresponding iterations of the different learning rates essentially represent the same decision boundary. However, in case of some random initial weight vector instead of a 0 vector, varying the learning rate will bring in differences.

Question 3

In the given I set of images from poly1.png to poly14.png, let poly1 to poly 7 belong to class 1 and poly 8 to poly 14 belong to class 2. Assume that all the weights of the perceptron are initialized as 0 with the learning rate of 0.01.
* Identify two discriminant features x1 and x2 for the two target classes w={w1,w2}. Here, w1 - class 1 and w2 - class 2. * Generate an input feature vector X for all the images mapping them to a corresponding taget classes wi, where i belongs to (1,2). * Train a **single perceptron and SVM** to learn the feature vector X mapping to w. * Plot and draw the final decision boundary separating the two classes.

Choosing and extracting features:

Since there are more green pixels in class 1 than class 2, let us choose x1 as the fraction of "greeinsh pixels" where a "greenish pixel" is defined as a pixel an array of (r,g,b) values where the green value is more than that of both red and blue values. Similarly we can choose another feature, "reddish pixels" as x2 for the classification purpose.

After calculating, the values are as follows:

x1	x2	class
0.0696443	0.0	0
0.22008099	0.0	0
0.12225877	0.0	0
0.12225877	0.0	0
0.08680556	0.0	0
0.08841766	0.0	0
0.06721722	0.0	0
0.0	0.11133429	1
0.0	0.11133429	1
0.0	0.13755403	1
0.0	0.07324398	1
0.0	0.12326172	1
0.0	0.0656937	1
0.0	0.0656937	1

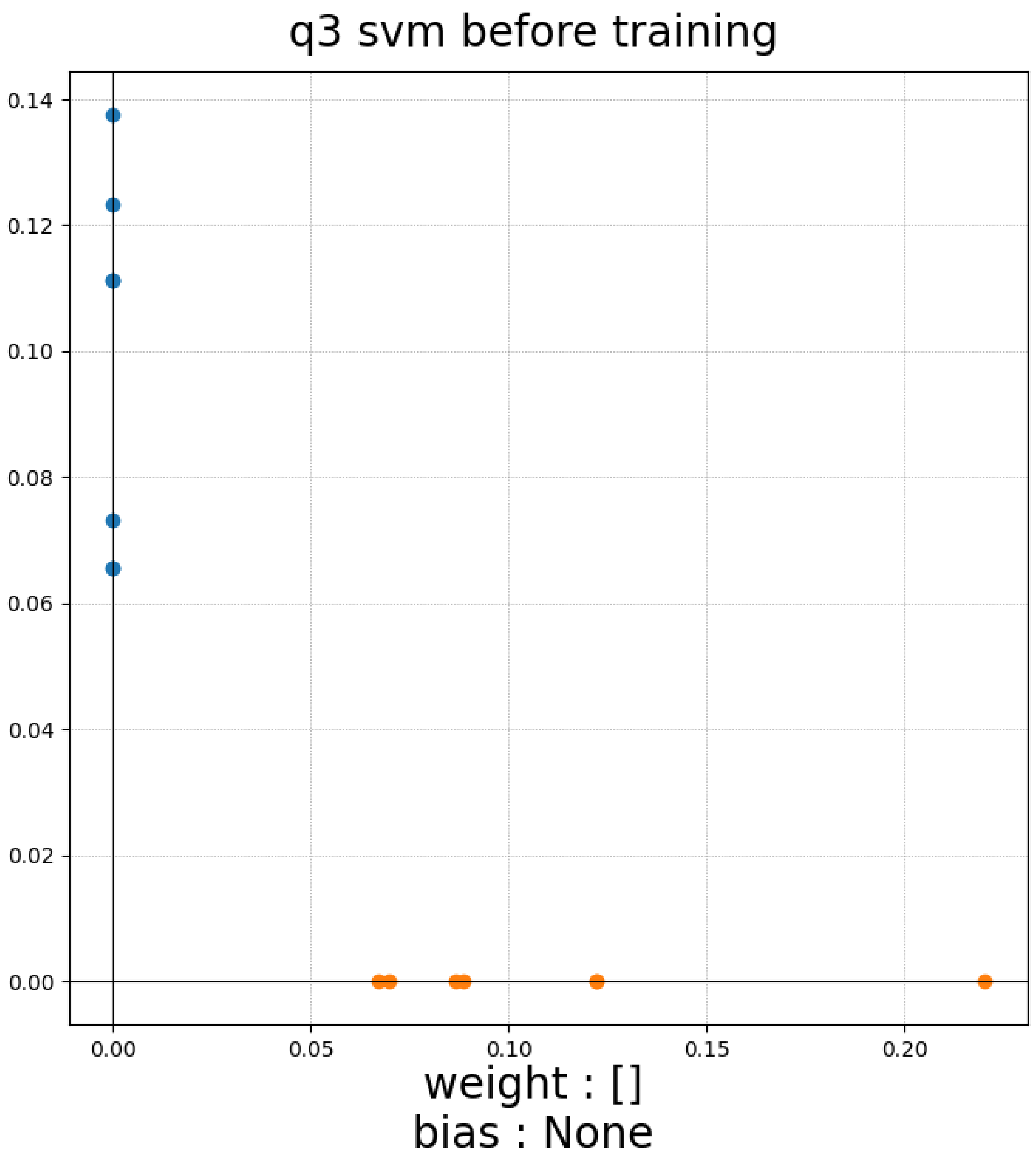
where 0 corresponds to w1 and 1 corresponds to w2.

code

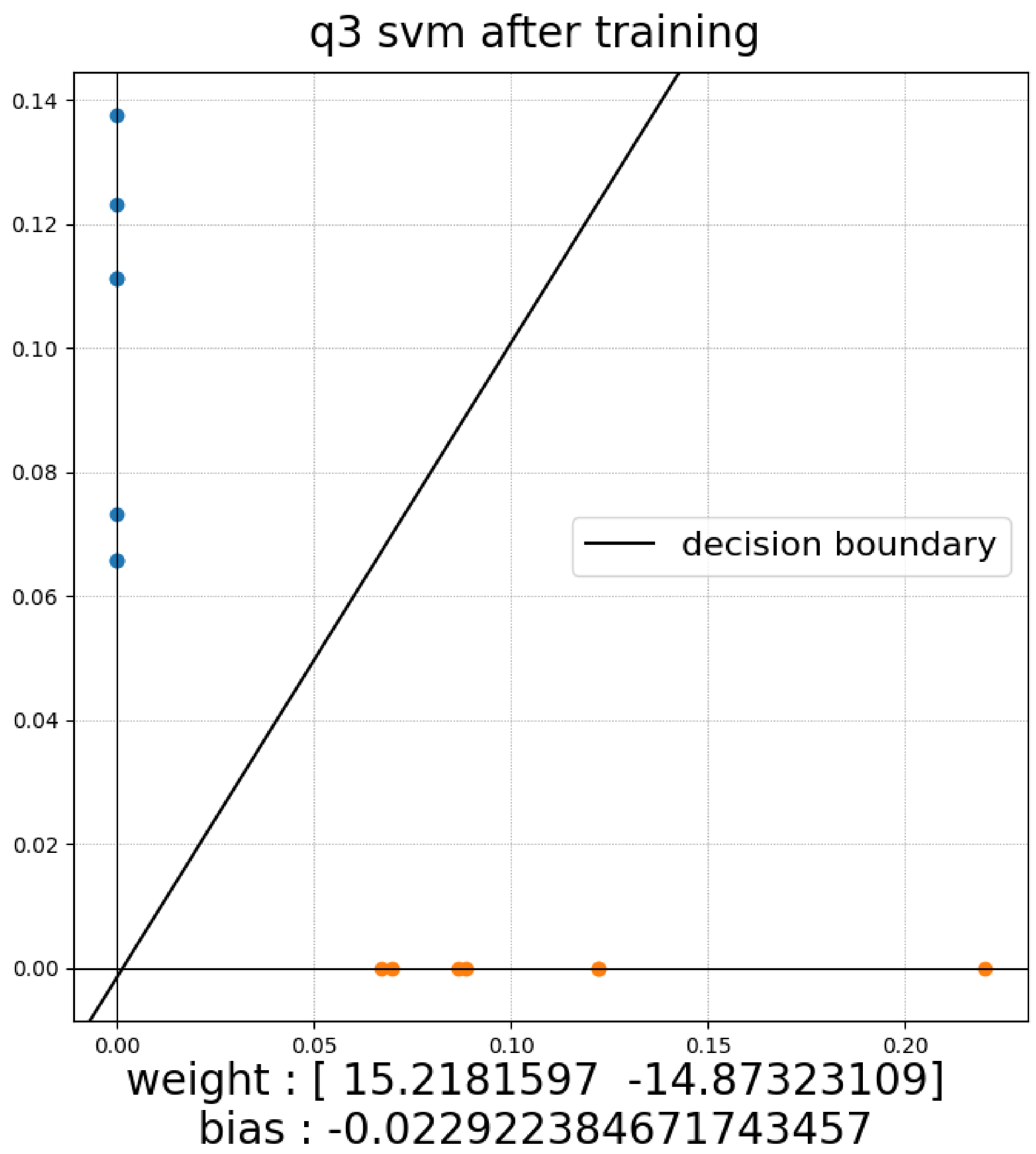
```
import numpy as np, cv2, svm, perceptron

# reading images
images=[]
for i in range(1,15):
    images.append(cv2.imread(f'poly_images/poly{i}.png'))

# choosing features:
def greenish_pixels(image):
    X,Y=image.shape[:2]
    value=0
    for x in range(X):
        for y in range(Y):
            b,g,r=image[x,y]
            # if int(g)>int(b)+int(r):
            if g>b and g>r:
                value+=1
    return value/X/Y

def reddish_pixels(image):
    X,Y=image.shape[:2]
    value=0
    for x in range(X):
        for y in range(Y):
            b,g,r=image[x,y]
            # if int(g)>int(b)+int(r):
            if r>b and r>g:
                value+=1
    return value/X/Y

x1=[]
x2=[]

# extracting features:
for image in images:
    x1.append(greenish_pixels(image))
    x2.append(reddish_pixels(image))

# training
X=np.array(list(zip(x1,x2)))
Y_perc=np.array([1,1,1,1,1,1,1, 0, 0, 0, 0, 0, 0, 0])
Y_svm =np.array([1,1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1])

svm.demo(X,Y_svm,plot_title="q3")

perceptron.demo(X,Y_perc,learning_rate=0.01,plot_title="q3")
```

output

SVM



Feature vectors before training



weight : [15.2181597 -14.87323109]
bias : -0.022922384671743457

After training, w = [15.22 -14.87] and bias = -0.023 approximately

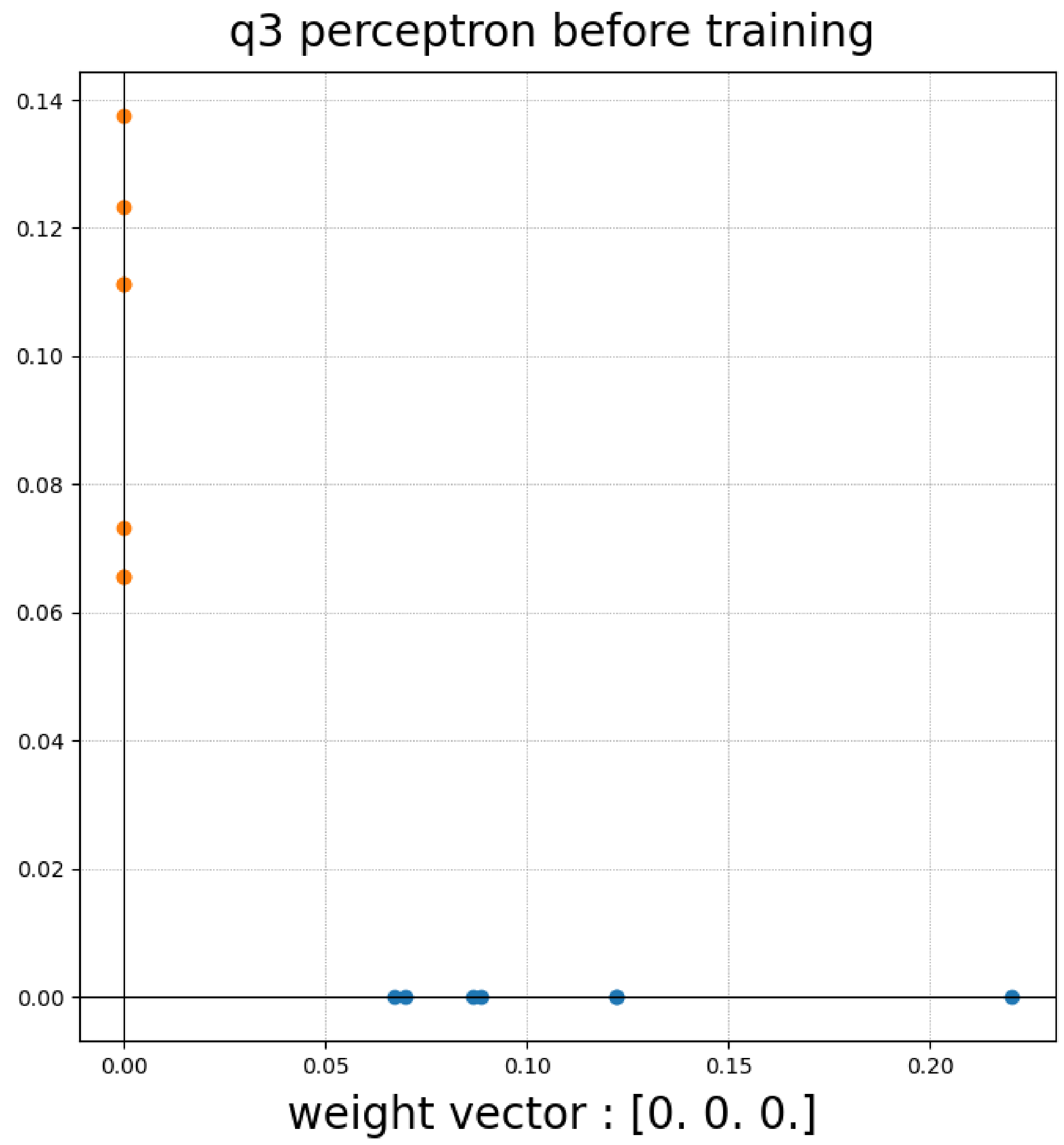
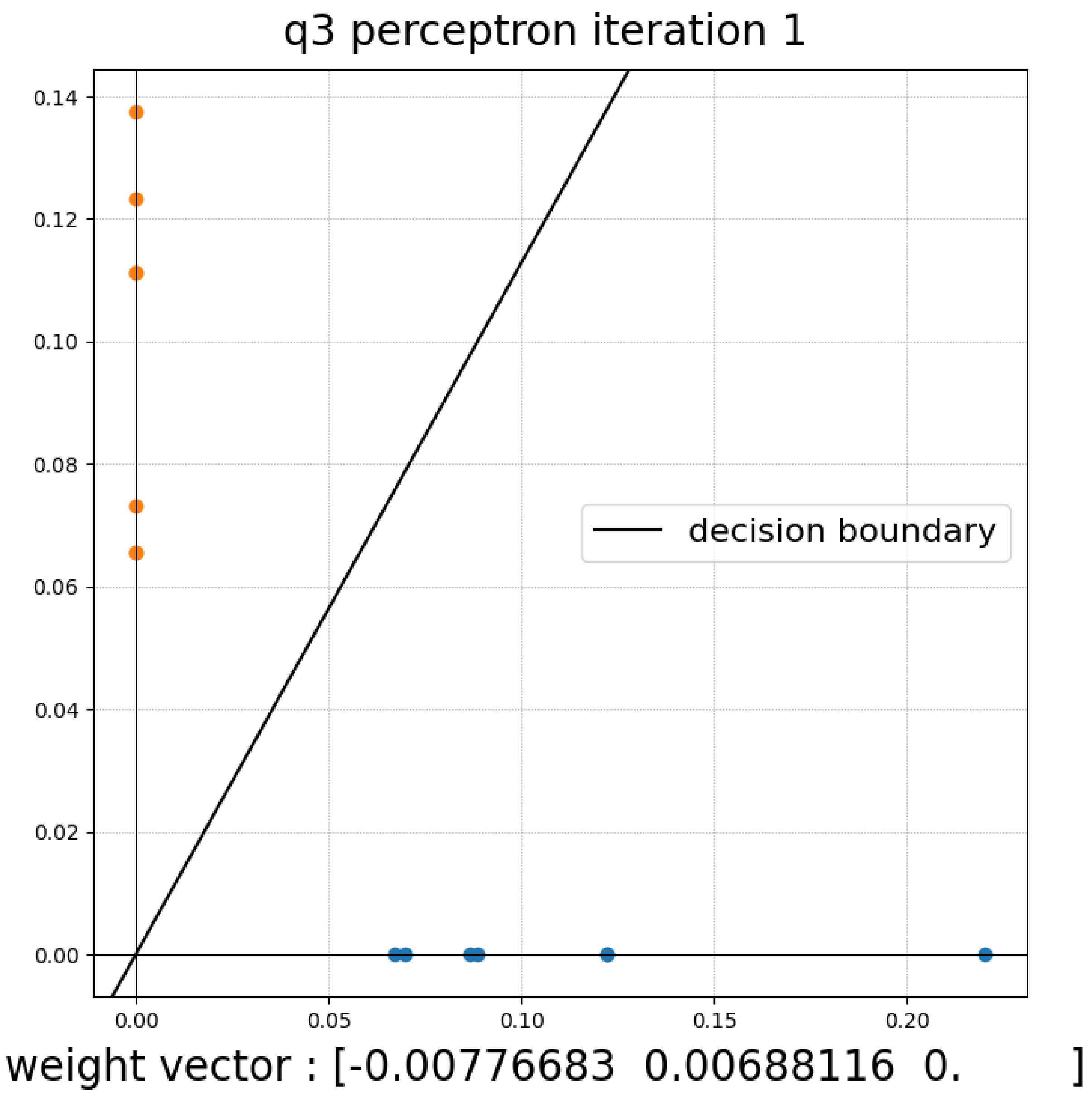


Figure 6: q3_perceptron_before_training.p

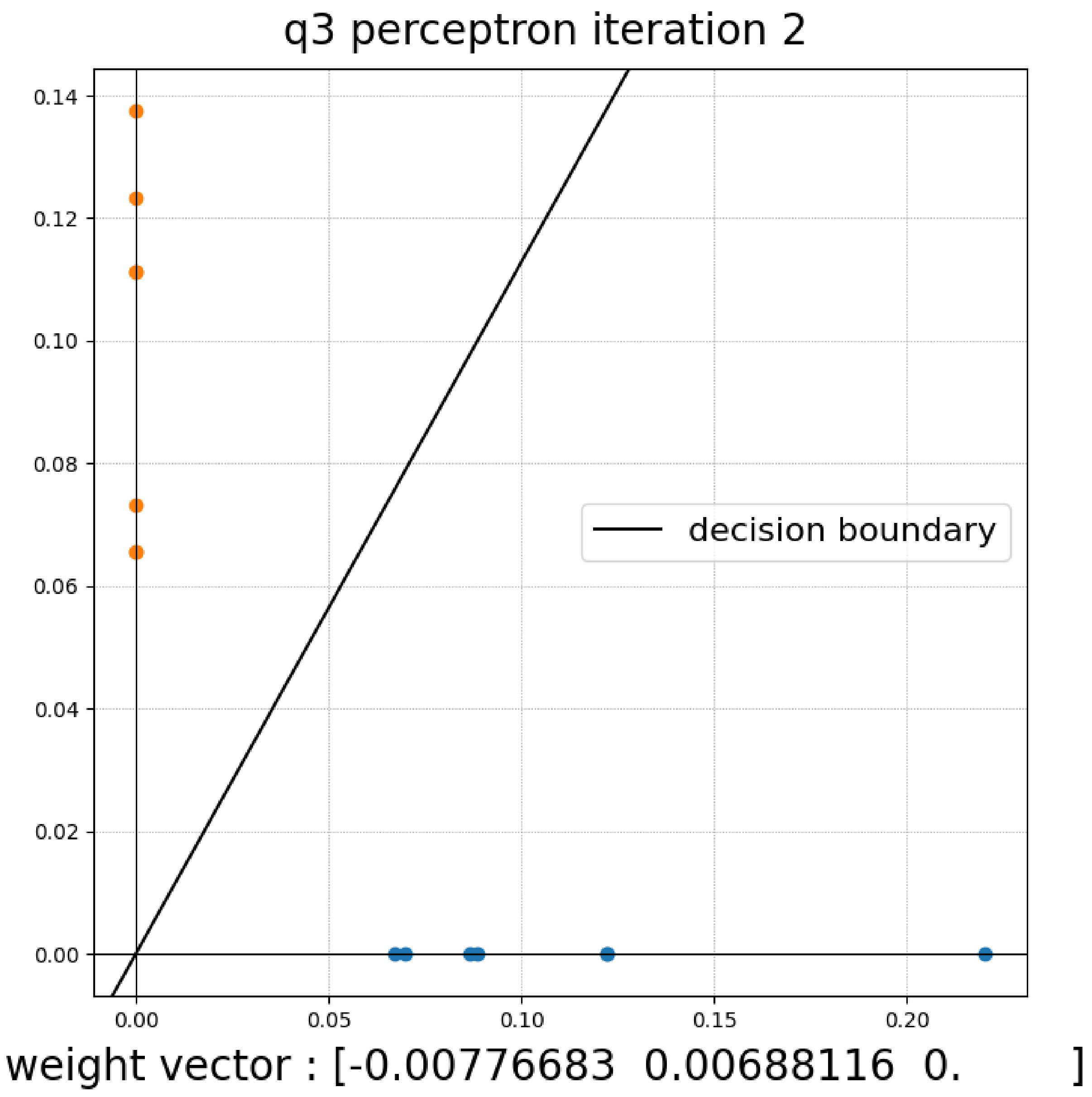
Perceptron

Iteration 1:



gradient of Jp = [-0.77668328 0.68811571 0.]
new weight = old weight + learning rate * gradient of Jp
= [0. 0. 0.] + 0.01 * [-0.77668328 0.68811571 0.]
= [-0.00776683 0.00688116 0.0]

Iteration 2:



gradient of Jp = [0. 0. 0.]
new weight = old weight + learning rate * gradient of Jp
= [-0.00776683 0.00688116 0.] + 0.01 * [0. 0. 0.]
= [-0.00776683 0.00688116 0.]
As there is no significant change in the new weight vector after this iteration, let us stop the training process in the 2nd iteration itself.

Question 4

From the iris dataset, choose the 'petal.length', 'sepal.width' for setosa, versicolor and virginica flowers. Learn decision boundary for the two features using asingle perceptron and SVM. Assume that all the weights of the perceptron are initialized as 0 with the learning rate of 0.01. Draw the decision boundary.

This is a multi class problem. Let us try to make the models learn the decision boundary between *class i* and *not class i* for each class.

code

```
import numpy as np, multi_class_perceptron,multi_class_svm

# extract required data from file
data = np.genfromtxt('../Assignment 2/Iris_dataset.csv', delimiter=',',dtype=str)

pl_index = np.where(data[0] == 'petal.length')[0][0]
sw_index = np.where(data[0] == 'sepal.width')[0][0]
variety_index = np.where(data[0] == 'variety')[0][0]

train_data=data[1:,[pl_index,sw_index]].astype(np.float)

train_label=data[1:,[variety_index]]
unique_labels = np.unique(train_label)
d={}
for i,label in enumerate(unique_labels):
    d[label]=i
train_label_encoded=np.array([d[label[0]] for label in train_label])

multi_class_perceptron.demo(train_data,train_label_encoded,plot_title='q4')
multi_class_svm.demo(train_data,train_label_encoded,plot_title='q4')
```

output

Perceptron

Since this is a much complicated problem, let us look at iterations 100, 500 and 1000 respectively.

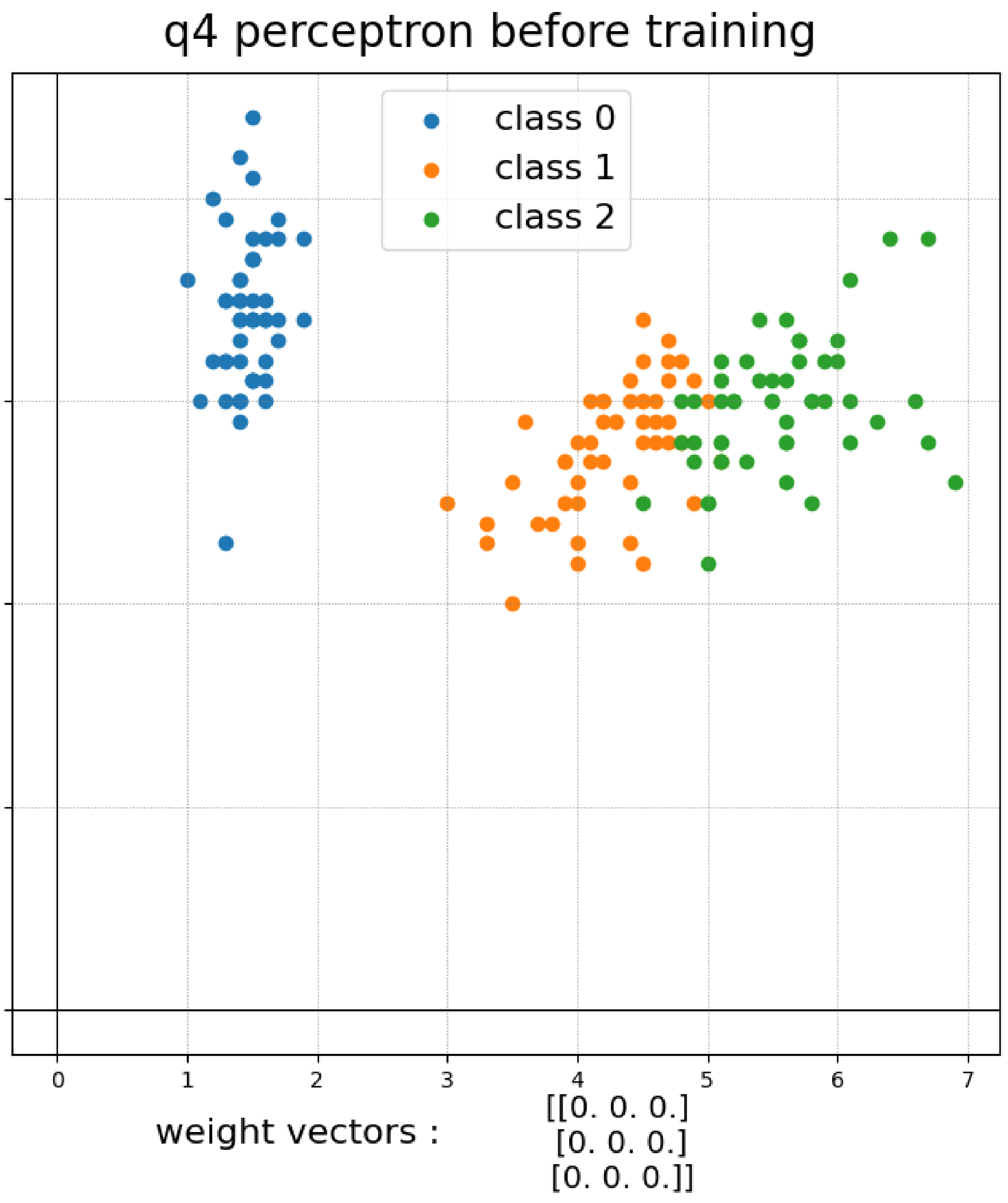
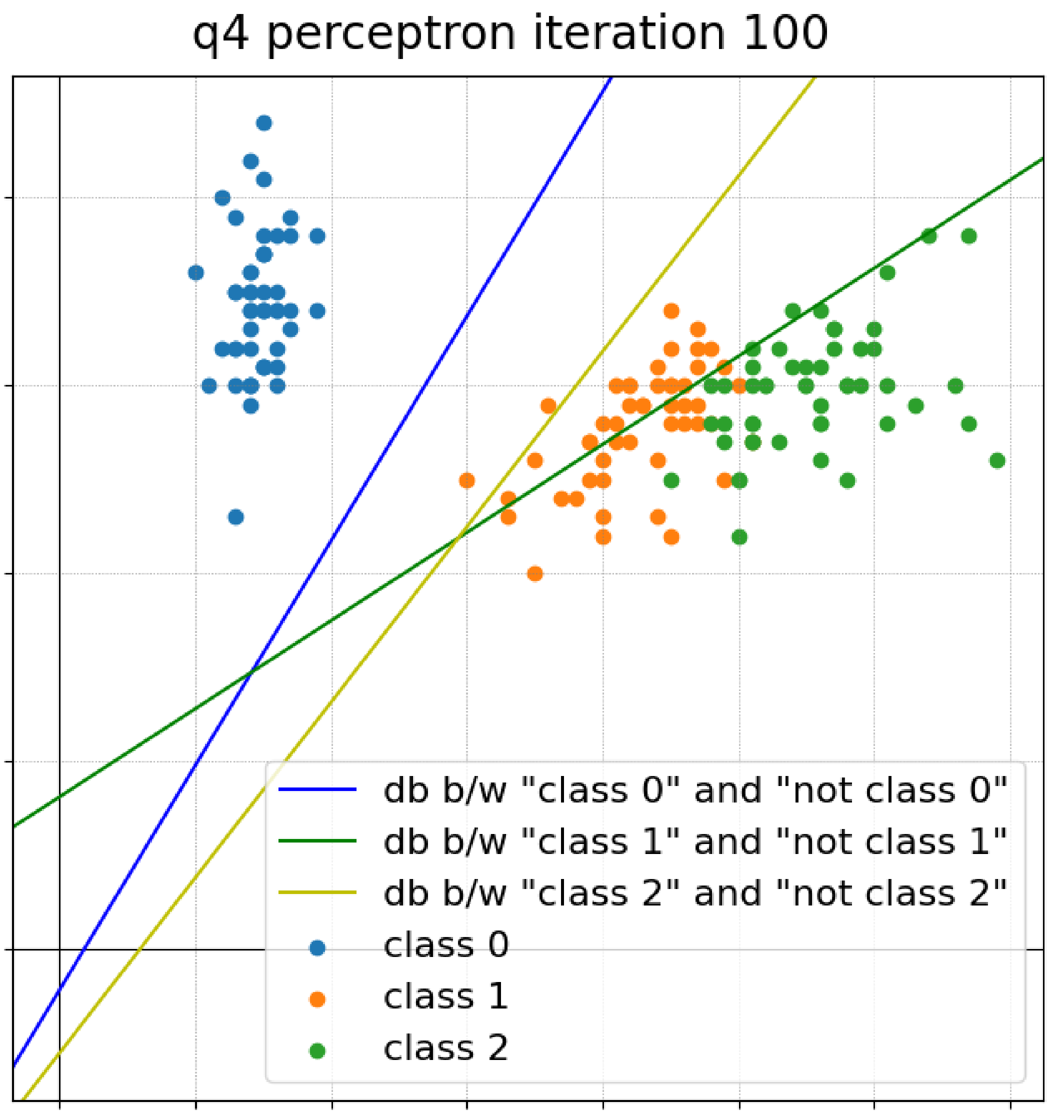


Figure 7: q4_perceptron_before_training.p

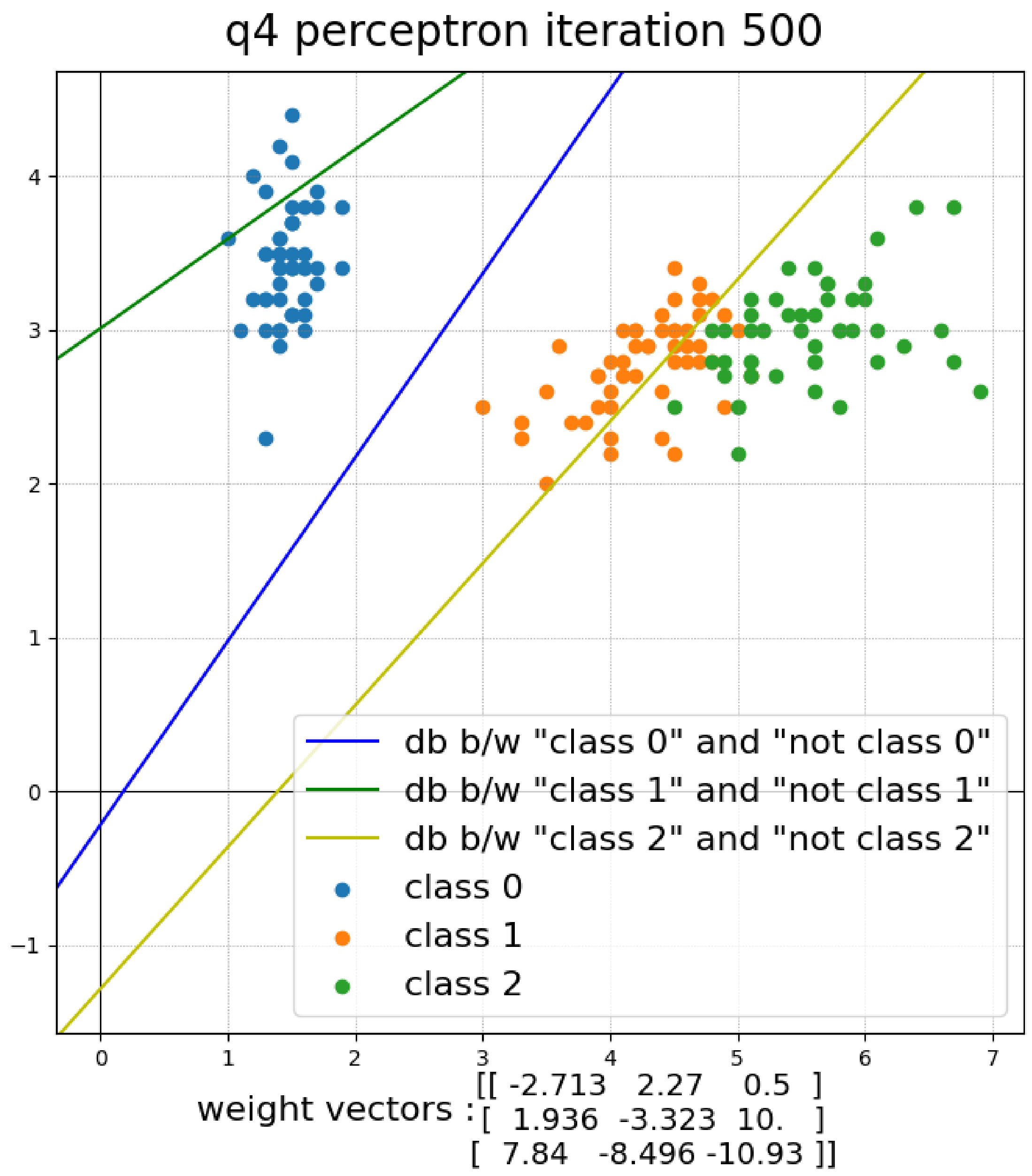
Iteration 100



```
weight vectors : [[-2.713  2.27   0.5  ]
                  [ 1.228 -2.619  2.12 ]
                  [ 8.088 -8.665 -4.81 ]]
```

The decision boundary, db0 between class 0 and not class 0 and the decision boundary between class2 and not class 2 look fine but they can be better. The decision boundary, db1 between class 1 and not class 1 does not look good.

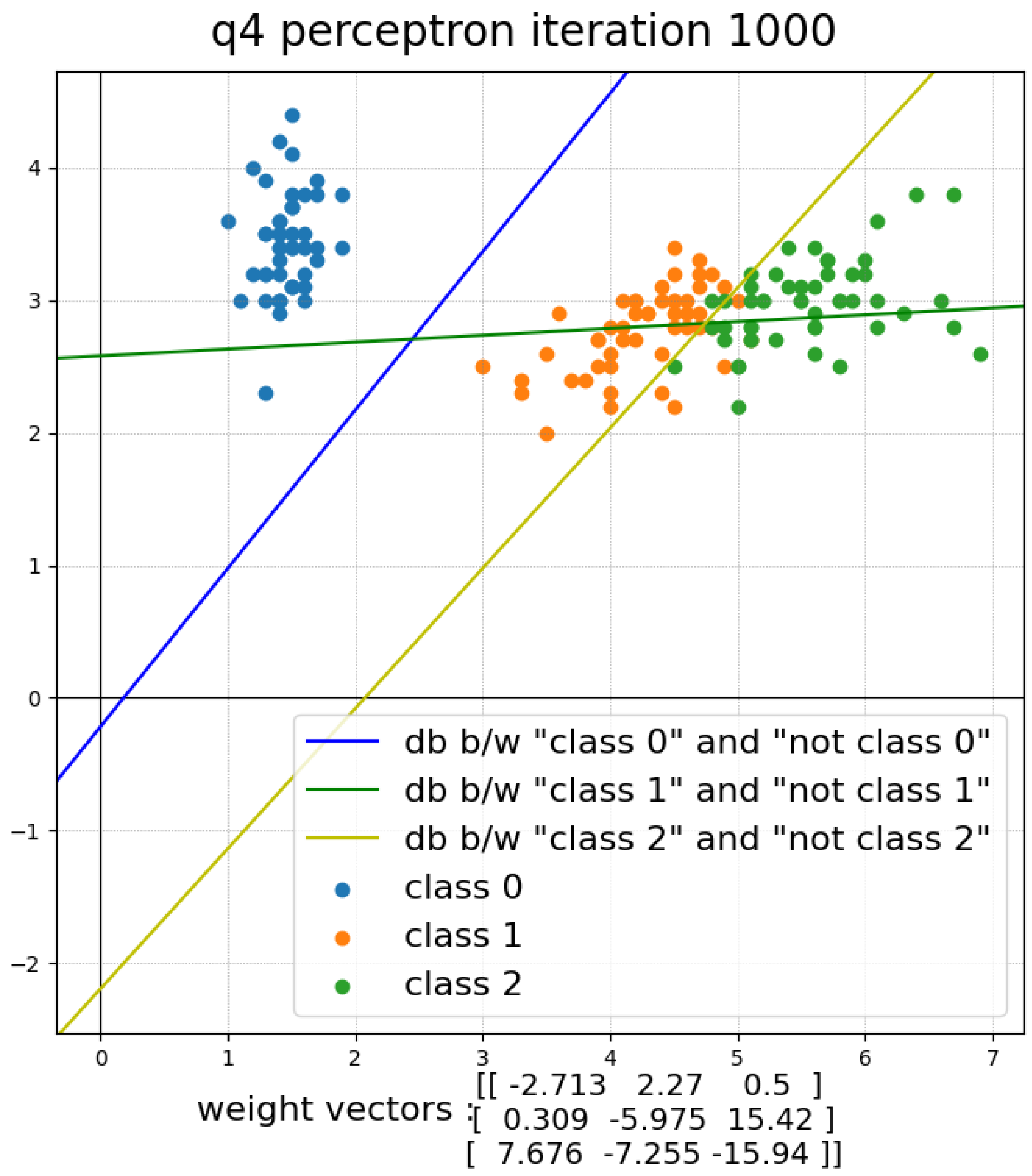
Iteration 500



```
                    [[ -2.713   2.27    0.5  ]
weight vectors :     [  1.936  -3.323  10.   ]
                    [  7.84   -8.496 -10.93 ]]
```

The decision boundaries db0 and db2 look much better now and separate all of the samples in the test case. The boundary d1 still does not look good.

Iteration 1000



```
[[ -2.713   2.27    0.5  ]
 [  0.309  -5.975  15.42 ]
 [  7.676  -7.255 -15.94 ]]
```

Now, after 1000 iterations, the decision boundaries db0 and db2 look ever so slightly better. But the extra 500 iterations it took to get here might not be worth it after all since it was already close to perfect in the previous case.

We can see that even after 1000 iterations, db1 does not look good at all. We can also see that it is not moving towards a direction but it is moving in a random direction everytime. This because the boundary between class 1 and not class 1 cannot be linear. (i.e) this region of class 1 is not linearly separable. Since the perceptron algorithm is based on linear algebra, we cannot arrive at a definite positive answer with the perceptron algorithm for this case.

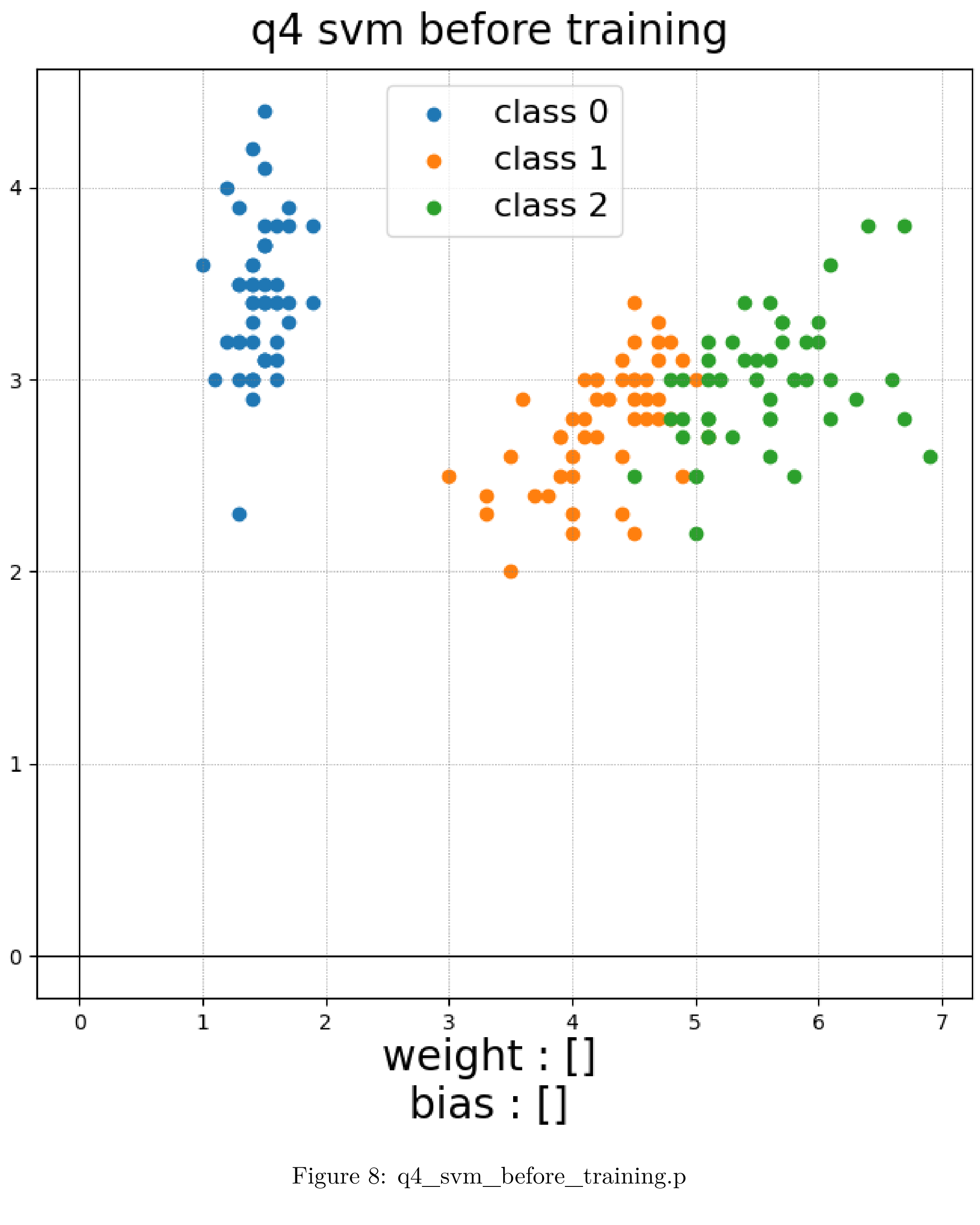
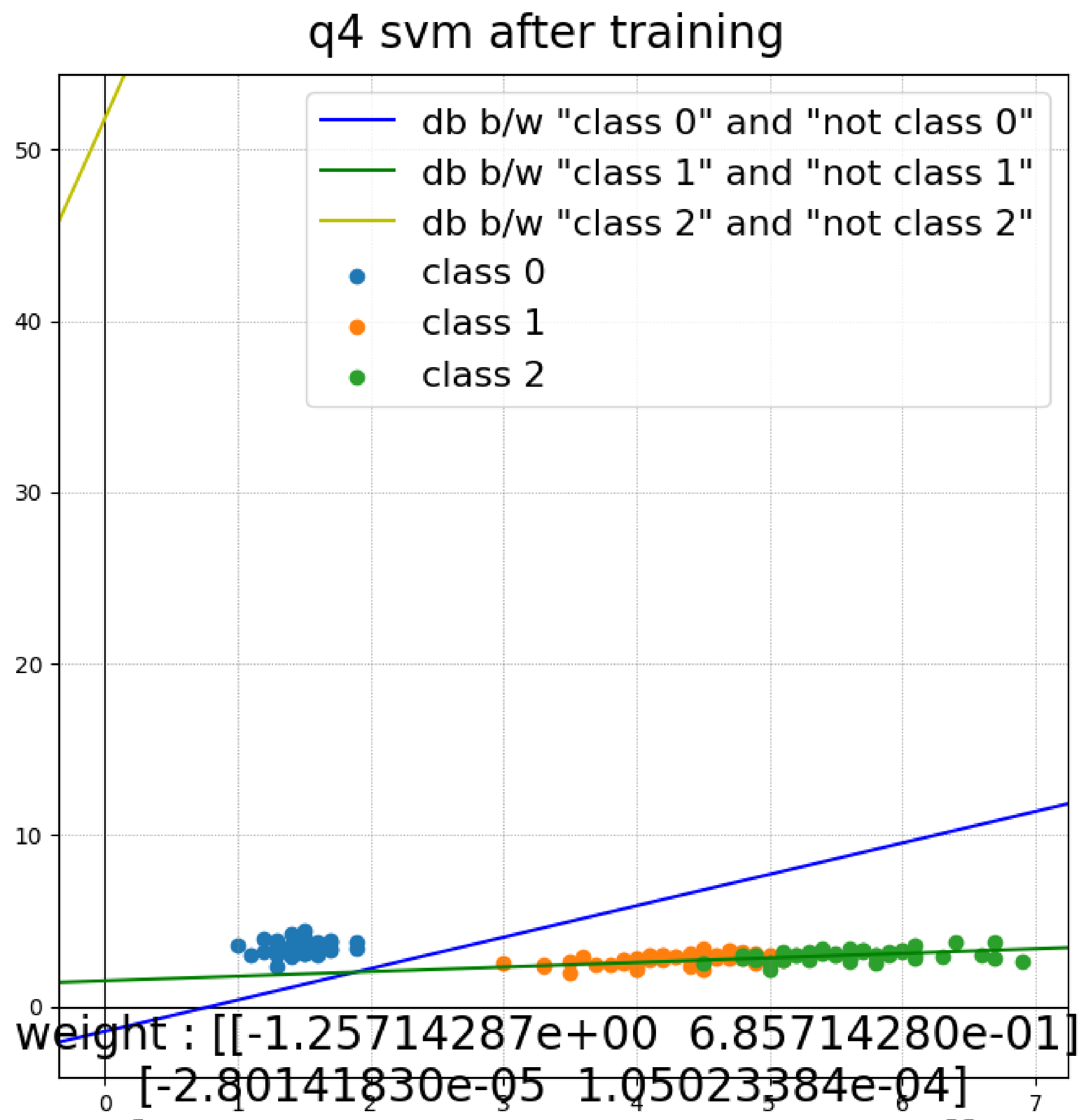


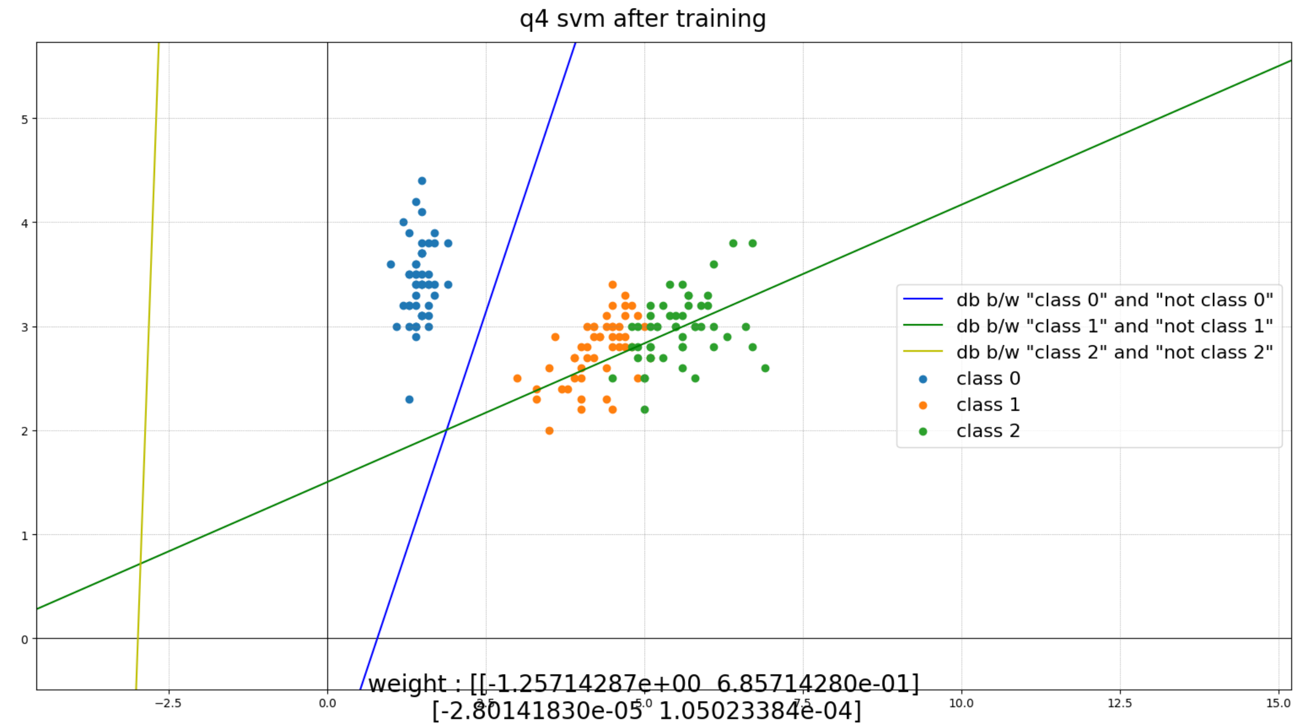
Figure 8: q4_svm_before_training.p

SVM



```
weight : [[-1.25714287e+00  6.85714280e-01]
 [-2.80141830e-05  1.05023384e-04]
 [-6.62803650e-05  3.81469727e-06]]
as : [ 1.00000000e+00 -1.57546997e-04 -1.97434425e-0
```

Again, since the dcision boundary between class 1 and not class 1 is not linear, the boundary is inaccurate for the same reasons. The other two boundaries are correctly predicted by the svm.



```
weight : [[-1.25714287e+00  6.85714280e-01]
 [-2.80141830e-05  1.05023384e-04]
 [-6.62803650e-05  3.81469727e-06]]
bias : [ 1.00000000e+00 -1.57546997e-04 -1.97434425e-04]
```

Here is a better version of the same plot (manually zoomed and panned).