CS 395

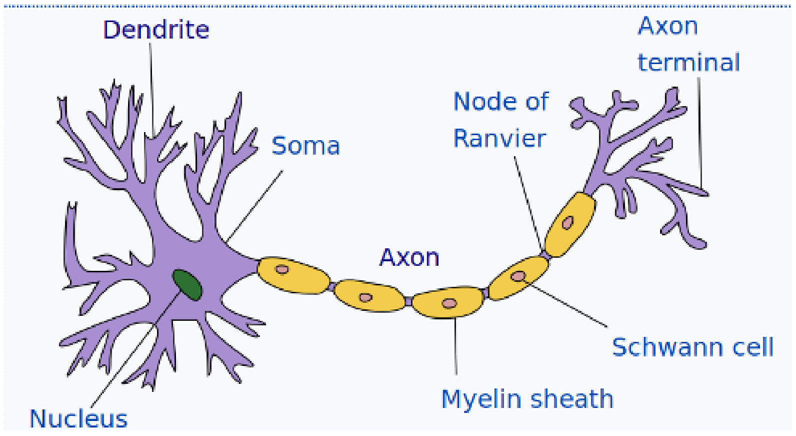
Homework 2 – Perceptrons

18 Points Total

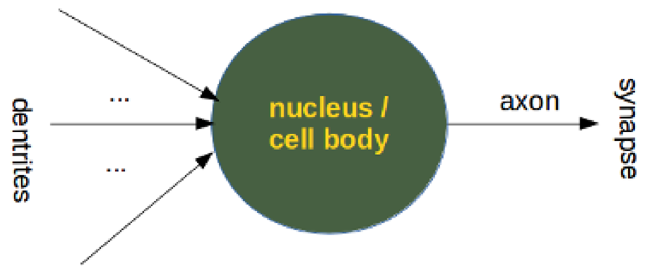
Due in Canvas by 11:59 PM on Saturday, January 26, 2019

Answer the following questions. This is an individual assignment. Show all code and outputs.

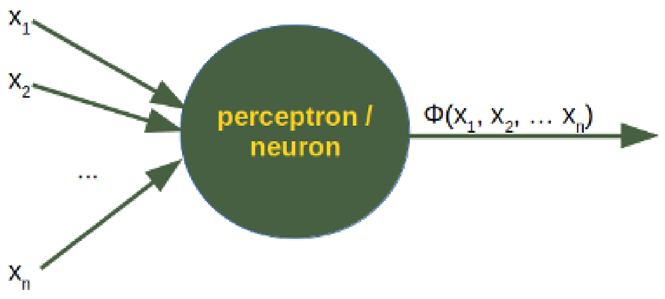
Recall from last week's lecture, the brain's peripheral nervous system looks like the following.



We can make this already abstract model (above) even more abstract:



This looks surprisingly similar to a perceptron:

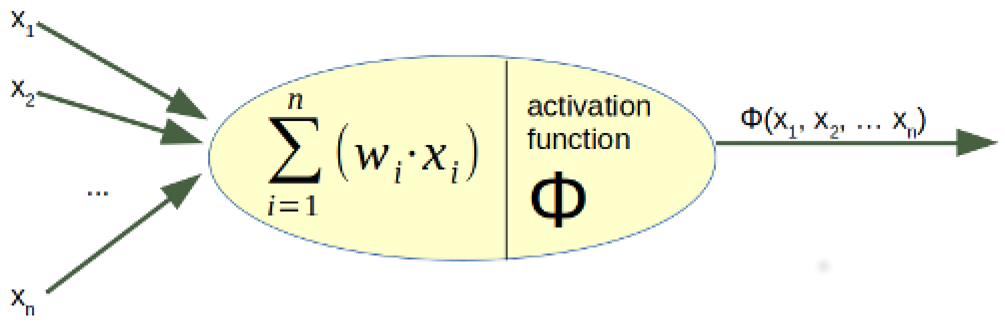


Let’s build and test a perceptron!

It is amazingly simple, what is going on inside the body of a perceptron or neuron. The input signals get multiplied by weight values, i.e. each input has its corresponding weight. This way the input can be adjusted individually for every x_i. We can see all the inputs as an input vector and the corresponding weights as the weights vector.

When a signal comes in, it gets multiplied by a weight value that is assigned to this particular input. That is, if a neuron has three inputs, then it has three weights that can be adjusted individually. The weights usually get adjusted during the learn phase.
After this the modified input signals are summed up. It is also possible to add additionally a so-called bias b to this sum. The bias is a value which can also be adjusted during the learn phase.

Finally, the actual output has to be determined. For this purpose an activation or step function Φ is applied to weighted sum of the input values.



The simplest form of an activation function is a binary function. If the result of the summation is greater than some threshold s, the result of Φ will be 1, otherwise 0.

$$\Phi(x) = \begin{cases} 1 & \mathrm{wx + b > s} \\ 0 & \text{otherwise} \end{cases}$$

We will write a very simple Neural Network implementing the logical "And" and "Or" functions.

Let's start with the "And" function. It is defined for two inputs:

Input1	**Input2**	**Output**
0	0	0
0	1	0
1	0	0
1	1	1

Before we begin we need to install the matplotlib library

```
conda install -c conda-forge matplotlib
```

or for other environments

https://matplotlib.org/faq/installing_faq.html

Once we get this installed successfully, let's build an AND perceptron.

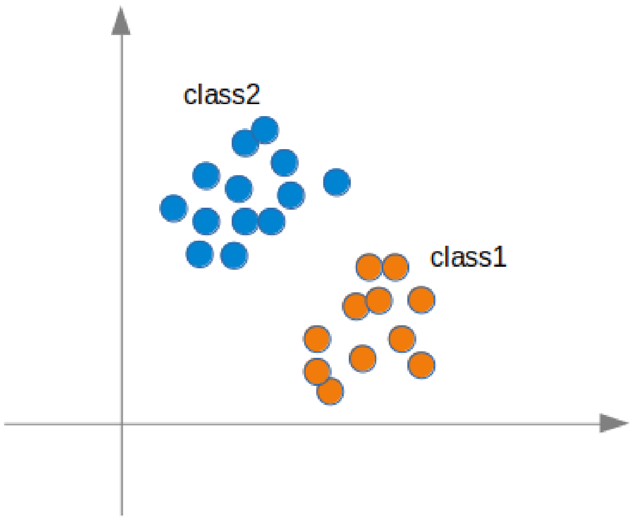
```
import numpy as np
class Perceptron:

    def __init__(self, input_length, weights=None):
        if weights is None:
            self.weights = np.ones(input_length) * 0.5
        else:
            self.weights = weights

    @staticmethod
    def unit_step_function(x):
        if x > 0.5:
            return 1
        return 0

    def __call__(self, in_data):
        weighted_input = self.weights * in_data
        weighted_sum = weighted_input.sum()
        return Perceptron.unit_step_function(weighted_sum)

p = Perceptron(2, np.array([0.5, 0.5]))
for x in [np.array([0, 0]), np.array([0, 1]),
          np.array([1, 0]), np.array([1, 1])]:
    y = p(np.array(x))
    print(x, y)
```

We get the following result:

```
[0 0] 0

[0 1] 0

[1 0] 0

[1 1] 1
```

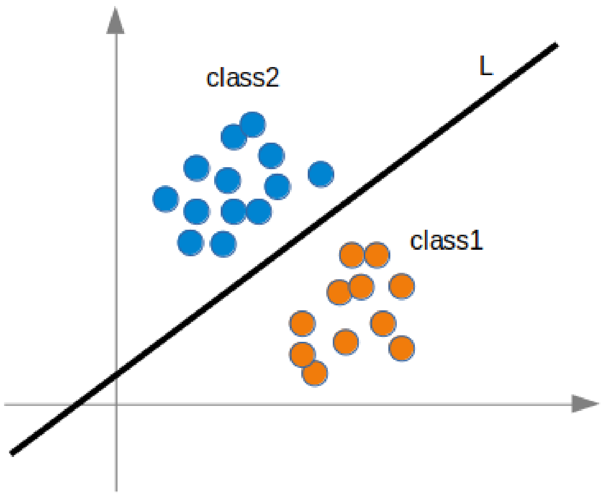
1. Run the code above and show the result you received. (1 point)
2. Based on the above, build an OR perceptron. (4 points)

In the following program, we train a neural network to classify two clusters in a 2-dimensional space. We show this in the following diagram with the two classes: class1 and class2. We will create those points randomly with the help of a line, the points of class2 will be above the line and the points of class1 will be below the line.



We will see that the neural network will find a line that separates the two classes. This line should not be mistaken for the line, which we used to create the points.

This line is called a **decision boundary**.



```
import numpy as np
from collections import Counter
class Perceptron:

    def __init__(self, input_length, weights=None):
        if weights==None:
            self.weights = np.random.random((input_length)) * 2 - 1
        self.learning_rate = 0.1

    @staticmethod
    def unit_step_function(x):
        if x < 0:
            return 0
        return 1

    def __call__(self, in_data):
        weighted_input = self.weights * in_data
        weighted_sum = weighted_input.sum()
        return Perceptron.unit_step_function(weighted_sum)

    def adjust(self,
               target_result,
               calculated_result,
               in_data):
```

```
        error = target_result - calculated_result
        for i in range(len(in_data)):
            correction = error * in_data[i] *self.learning_rate
            self.weights[i] += correction

def above_line(point, line_func):
    x, y = point
    if y > line_func(x):
        return 1
    else:
        return 0

points = np.random.randint(1, 100, (100, 2))
p = Perceptron(2)
def lin1(x):
    return  x + 4
for point in points:
    p.adjust(above_line(point, lin1),
             p(point),
             point)
evaluation = Counter()
for point in points:
    if p(point) == above_line(point, lin1):
        evaluation["correct"] += 1
    else:
        evaluation["wrong"] += 1
print(evaluation.most_common())
```

We would get something like the following (remember we are using random numbers, so we could get different values)

```
[('correct', 100)]
```

3. Run the above code and show your results. (1 point)
4. Do we use an activation function in our code? Describe the activation function used. (3 points)

5. What effect does changing the learning rate have on the code? Remember, since random values are being generated for points, it may be hard to deterministically evaluate this empirically. (2 points)

The decision boundary of our previous network can be calculated by looking at the following condition

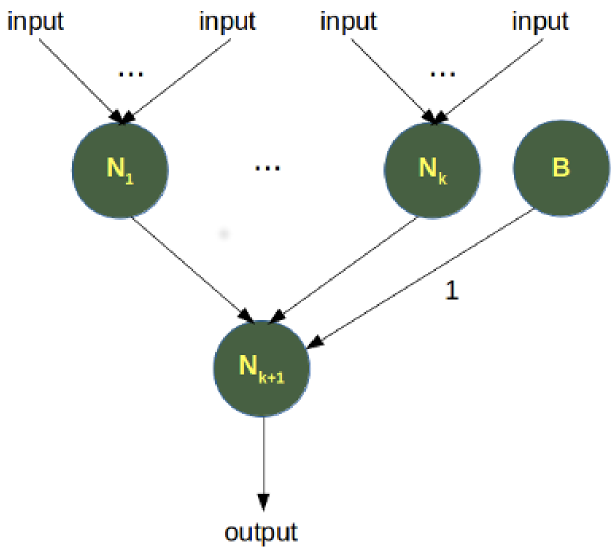
$$x_1 w_1 + x_2 w_2 = 0$$

We can change the equation into

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

Single Layer with Bias

As the constant term bb determines the point at which a line crosses the y-axis, i.e. the y-intercept, we can see that our network can only calculate lines which pass through the origin, i.e. the point (0, 0). We will need a bias to get other lines as well, i.e. lines which don't go through the origin. A neural network with bias nodes can look like this:



Now, the linear equation for a perceptron contains a bias:

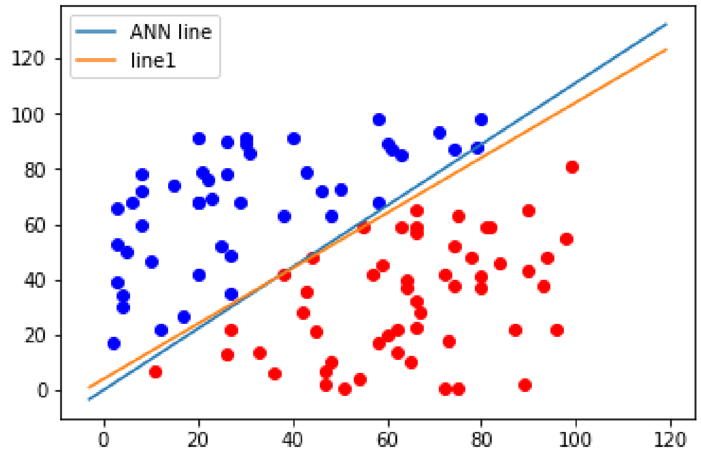
$$b + \sum_{i=1}^{n} x_i \cdot w_i = 0$$

We add now some code to print the points and the dividing line according to the previous equation:

```
from matplotlib import pyplot as plt
cls = [[], []]
for point in points:
    cls[above_line(point, lin1)].append(tuple(point))
colours = ("r", "b")
for i in range(2):
    X, Y = zip(*cls[i])
    plt.scatter(X, Y, c=colours[i])

X = np.arange(-3, 120)

m = -p.weights[0] / p.weights[1]
print(m)
plt.plot(X, m*X, label="ANN line")
plt.plot(X, lin1(X), label="line1")
plt.legend()
plt.show()
```

We might get something like the following, which is the slope of our line (m):

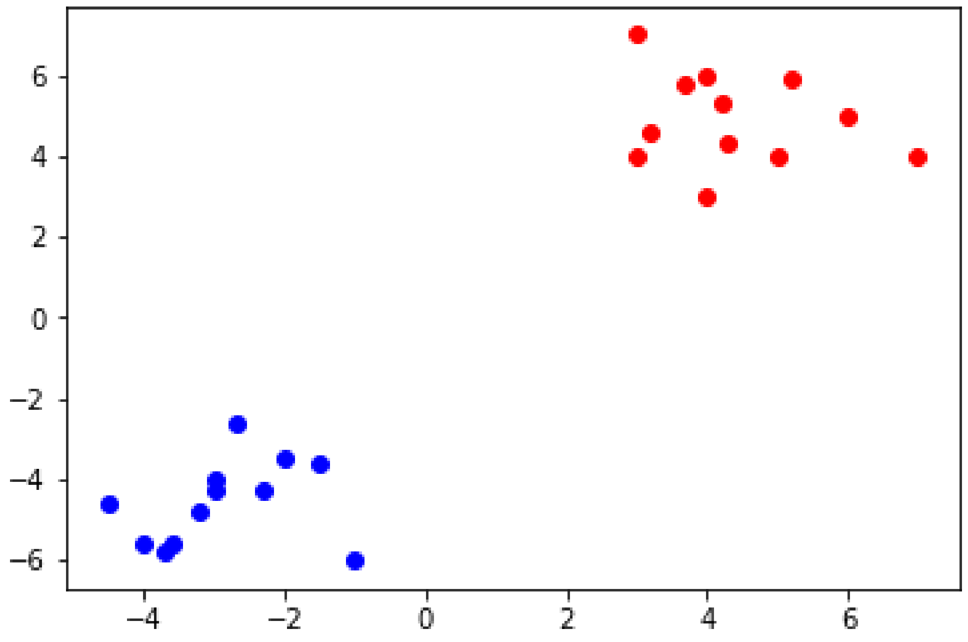
```
1.11082111934
```



6. Run the above code (don’t’ forget to download matplotlib first!) and show your code and results. (1 point)
7. From the code, how is the line determined? What in the code sets the slope and intercept of the line (2 points)

We create a new dataset for our next experiments:

```
from matplotlib import pyplot as plt
class1 = [(3, 4), (4.2, 5.3), (4, 3), (6, 5), (4, 6), (3.7, 5.8),
          (3.2, 4.6), (5.2, 5.9), (5, 4), (7, 4), (3, 7), (4.3, 4.3) ]
class2 = [(-3, -4), (-2, -3.5), (-1, -6), (-3, -4.3), (-4, -5.6),
          (-3.2, -4.8), (-2.3, -4.3), (-2.7, -2.6), (-1.5, -3.6),
          (-3.6, -5.6), (-4.5, -4.6), (-3.7, -5.8) ]
X, Y = zip(*class1)
plt.scatter(X, Y, c="r")
X, Y = zip(*class2)
plt.scatter(X, Y, c="b")
plt.show()
```



We run the following next:

```
from itertools import chain
p = Perceptron(2)
```

```
def lin1(x):
    return  x + 4
for point in class1:
    p.adjust(1,
             p(point),
             point)
for point in class2:
    p.adjust(0,
             p(point),
             point)

evaluation = Counter()
for point in chain(class1, class2):
    if p(point) == 1:
        evaluation["correct"] += 1
    else:
        evaluation["wrong"] += 1

testpoints = [(3.9, 6.9), (-2.9, -5.9)]
for point in testpoints:
    print(p(point))

print(evaluation.most_common())
```

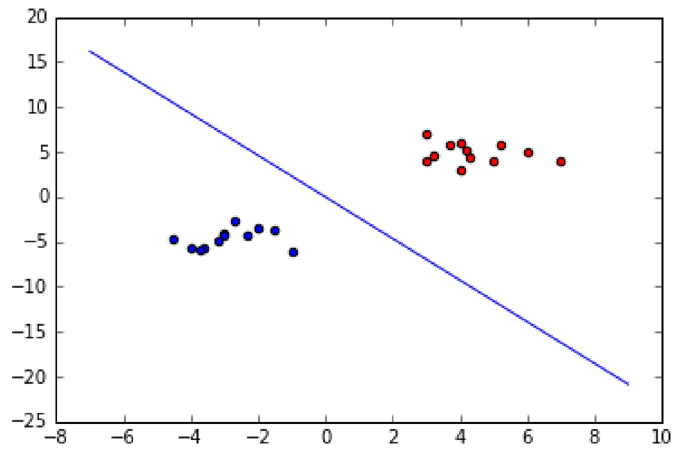
We get the following (yours should look the same):

```
1
0
[('correct', 12), ('wrong', 12)]
```

Now we write and execute the following code:

```
from matplotlib import pyplot as plt
X, Y = zip(*class1)
plt.scatter(X, Y, c="r")
X, Y = zip(*class2)
```

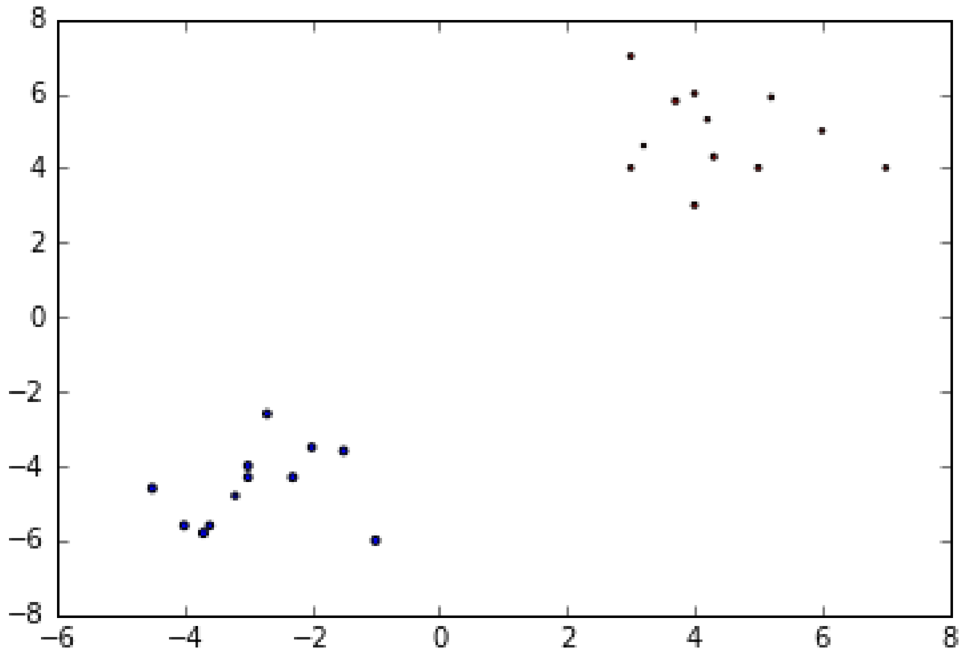
```
plt.scatter(X, Y, c="b")
x = np.arange(-7, 10)
y = 5*x + 10
m = -p.weights[0] / p.weights[1]
plt.plot(x, m*x)
plt.show()
```



8. Run your code. Show your results. (1 point)
9. Does your line look the same? Why or why not (1 point)

```
from matplotlib import pyplot as plt
class1 = [(3, 4, 3), (4.2, 5.3, 2.5), (4, 3, 3.8),
          (6, 5, 2.7), (4, 6, 2.9), (3.7, 5.8, 4.2),
          (3.2, 4.6, 1.9), (5.2, 5.9, 2.7), (5, 4, 3.5),
          (7, 4, 2.7), (3, 7, 3.1), (4.3, 4.3, 3.8) ]
class2 = [(-3, -4, 7.6), (-2, -3.5, 6.9), (-1, -6, 8.6),
          (-3, -4.3, 7.4), (-4, -5.6, 7.9), (-3.2, -4.8, 5.3),
          (-2.3, -4.3, 8.1), (-2.7, -2.6, 7.3), (-1.5, -3.6, 7.8),
          (-3.6, -5.6, 6.8), (-4.5, -4.6, 8.3), (-3.7, -5.8, 8.7) ]
X, Y, Z = zip(*class1)
plt.scatter(X, Y, Z, c="r")
X, Y, Z = zip(*class2)
plt.scatter(X, Y, Z, c="b")
plt.show()
```

We get the following:



Linearly Separable and Inseparable Neural Networks

If two data clusters (classes) can be separated by a decision boundary in the form of a linear equation

$$\sum_{i=1}^{n} x_i \cdot w_i = 0$$

they are called linearly separable.

Otherwise, i.e. if such a decision boundary does not exist, the two classes are called linearly inseparable. In this case, we cannot use a simple neural network.

We will come back now to our initial example with the random points above and below a line. We will rewrite the code using a bias value.

First we will create two classes with random points, which are not separable by a line crossing the origin.

We will add a bias b to our neural network. This leads us to the following condition

$$x_1 w_1 + x_2 w_2 + b w_3 = 0$$

And therefore we get

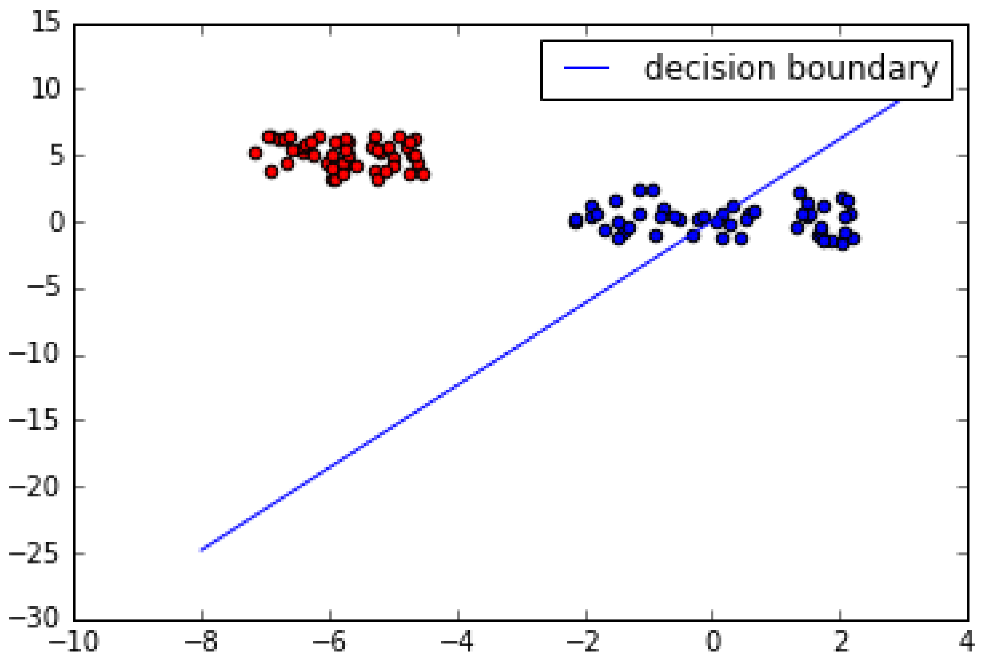
$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_3}{w_2} b$$

```
import numpy as np
from matplotlib import pyplot as plt
npoints = 50
X, Y = [], []
# class 0
X.append(np.random.uniform(low=-2.5, high=2.3, size=(npoints,)) )
Y.append(np.random.uniform(low=-1.7, high=2.8, size=(npoints,)))
# class 1
X.append(np.random.uniform(low=-7.2, high=-4.4, size=(npoints,)) )
Y.append(np.random.uniform(low=3, high=6.5, size=(npoints,)))
learnset = []
for i in range(2):
    # adding points of class i to learnset
    points = zip(X[i], Y[i])
    for p in points:
        learnset.append((p, i))
colours = ["b", "r"]
for i in range(2):
    plt.scatter(X[i], Y[i], c=colours[i])
```

```
import numpy as np
from collections import Counter
class Perceptron:

    def __init__(self, input_length, weights=None):
        if weights==None:
            self.weights = np.random.random((input_length)) * 2 - 1
        self.learning_rate = 0.1

    @staticmethod
    def unit_step_function(x):
        if x < 0:
            return 0
        return 1

    def __call__(self, in_data):
        weighted_input = self.weights * in_data
        weighted_sum = weighted_input.sum()
        return Perceptron.unit_step_function(weighted_sum)

    def adjust(self,
               target_result,
               calculated_result,
               in_data):
        error = target_result - calculated_result
        for i in range(len(in_data)):
            correction = error * in_data[i] *self.learning_rate
            self.weights[i] += correction



p = Perceptron(2)
for point, label in learnset:
    p.adjust(label,
             p(point),
             point)
evaluation = Counter()
for point, label in learnset:
    if p(point) == label:
        evaluation["correct"] += 1
    else:
        evaluation["wrong"] += 1
print(evaluation.most_common())
colours = ["b", "r"]
for i in range(2):
    plt.scatter(X[i], Y[i], c=colours[i])
XR = np.arange(-8, 4)
m = -p.weights[0] / p.weights[1]
print(m)
plt.plot(XR, m*XR, label="decision boundary")
plt.legend()
plt.show()
```

Here is what I got. Your answer might differ:

```
[('correct', 77), ('wrong', 23)]
3.10186712936
```



It is not possible to find a solution with one neuron and without a bias node. The reason is that the class of the blue data points spread around the origin. Without bias nodes we get only lines going through the origin as we have mentioned earlier. It is easy to see that no line going through the origin can separate the blue from the red data.

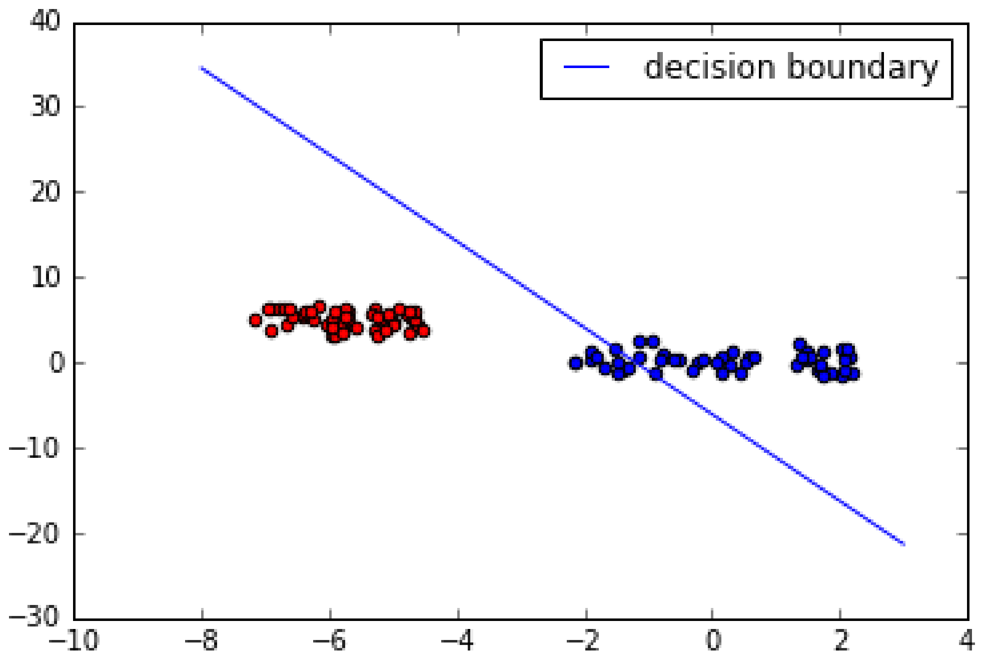
The following class uses bias nodes and solves this problem:

```
import numpy as np
from collections import Counter
class Perceptron:

    def __init__(self, input_length, weights=None):
        if weights==None:
            # input_length + 1 because bias needs a weight as well
            self.weights = np.random.random((input_length + 1)) * 2 - 1
        self.learning_rate = 0.05
        self.bias = 1

    @staticmethod
    def sigmoid_function(x):
        res = 1 / (1 + np.power(np.e, -x))
        return 0 if res < 0.5 else 1

    def __call__(self, in_data):
        weighted_input = self.weights[:-1] * in_data
        weighted_sum = weighted_input.sum() + self.bias *self.weights[-1]
        return Perceptron.sigmoid_function(weighted_sum)

    def adjust(self,
               target_result,
               calculated_result,
               in_data):
        error = target_result - calculated_result
        for i in range(len(in_data)):
            correction = error * in_data[i]  *self.learning_rate
            #print("weights: ", self.weights)
            #print(target_result, calculated_result, in_data, error,
correction)
            self.weights[i] += correction
        # correct the bias:
```

```
        correction = error * self.bias * self.learning_rate
        self.weights[-1] += correction



p = Perceptron(2)
for point, label in learnset:
    p.adjust(label,
             p(point),
             point)
evaluation = Counter()
for point, label in learnset:
    if p(point) == label:
        evaluation["correct"] += 1
    else:
        evaluation["wrong"] += 1
print(evaluation.most_common())
colours = ["b", "r"]
for i in range(2):
    plt.scatter(X[i], Y[i], c=colours[i])
XR = np.arange(-8, 4)
m = -p.weights[0] / p.weights[1]
b = -p.weights[-1]/p.weights[1]
print(m, b)
plt.plot(XR, m*XR + b, label="decision boundary")
plt.legend()
plt.show()
```

We get the following (your answer may vary):

```
[('correct', 90), ('wrong', 10)]
-5.07932788718 -6.08697420041
```



10. Run the code above several times. Do you get the same results? (2 points)