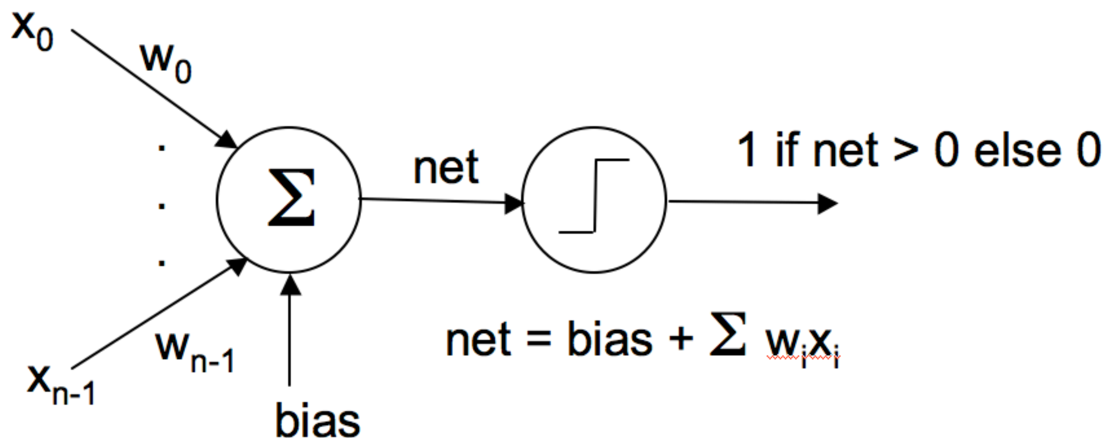
CS 42 Fall 2012
Python Exercise, 17 December
Artificial Neurons: Circuits that Can Learn
Robert M. Keller

In this exercise, we will explore the use of Python to create simple neural models and show how these objects can learn. We expect to be using Python objects, inheritance, lists, lambda expressions, and higher-order functions map and reduce. **The last page provides questions for summarizing your results and lists references.**

A Basic Neuron Model

The figure below shows how a neuron can be modeled using "threshold logic". This was one of the very first formal models, as put forth by McCulloch and Pitts (1943). This model was the basis for Rosenblatt's Perceptron(1958), the first work on learning, and also inspired Kleene's original work on Regular Expressions (1956).

Although the inputs and outputs to this model are in set {0, 1}, the operation of the neuron is based on real-valued *weights* that are attached to each input. There is an additional input called the *bias*. To compute the output of a neuron, we first compute the *net value*, which is defined to be the bias plus the weighted sum of the inputs. If the net value is positive, then the neuron is said to "fire" and the output is 1. If not, the output is 0. The figure diagrams a basic neuron with n inputs x_i, weights w_i, and bias b.



A linear neuron

Class LinearNeuron

The first step of this exercise is to implement a Python class representing the neuron described above. Although we call it LinearNeuron, this is slightly a

misnomer, as the output function is a nonlinearity. Nonetheless, we use this term to distinguish it from QuadraticNeuron defined later.

#1. Create a file **neuron.py**, which is used to define this class. It begins as follows:

```
#!python

class LinearNeuron:

    def __init__(self, inputs, learning_rate):        # constructor/initializer
        """ create an instance of LinearNeuron, which has
            a specified number of inputs,
            a single output,
            and a learning rate"""
        self.inputs = inputs
        self.learning_rate = learning_rate
        self.weights = [0 for i in range(self.inputs)]
        self.bias = 0

    def display(self):
        """ display the weights and bias of this LinearNeuron"""
        print self.weights, self.bias, 'learning rate = ', self.learning_rate
```

The **__init__** method serves to initialize aspects of the neuron. It remembers the number of inputs and initializes a weight list to all 0's. The learning_rate argument will be explained in the next section.

The ***self*** arguments are Python's way of establishing the object to which the method is applied, sort of like ***this*** in Java. When the method is called on an object, the self argument is omitted.

I am providing a file **main.py** that will have tests in it that use your methods. The file begins something like the following, and you can see the object being constructed at the top.

```
from neuron import LinearNeuron

# create a 2-input LinearNeuron neuron2
neuron2 = LinearNeuron(2, 0.1)
```

A second file **cancer.py** is also provided, and we will use that later on.

With just the display method, you should be able to get output by calling

```
neuron2.display()
```

You may use the IDLE or execute the file from the command line:

python –i main.py

#2. Next add to neuron.py another method:

```
def operate(self, input):
    """ present list of inputs to LinearNeuron, returns output 0 or 1"""
```

With this you should be able to operate neuron2 by giving it selected inputs:

```
print neuron2.operate([0, 0])
print neuron2.operate([0, 1])
print neuron2.operate([1, 0])
print neuron2.operate([1, 1])
```

Because the weights are initialized to 0's, the result should be 0 in each case.

Learning

One of Rosenblatt's contributions was to describe how an artificial neuron can learn. To do this, we need the notion of **desired** value and **error**. The *desired* value is the value the neuron should produce for a given input. The error is the desired value minus the actual output. The idea of learning is to teach the neuron to produce the desired value by modifying the weights.

The **Perceptron Learning Rule** says that to learn, we present a **training set** which consists of inputs paired up with desired values. We operate the perceptron for an input, observe the output and compute the error, which will be in {0, 1, -1}. Then if the error is non-zero we **modify the weights and bias** according to a formula: for each index i:

$w_i = w_i + \text{learning_rate} * \text{error} * x_i$,
$bias = bias + \text{learning_rate} * \text{error}$

Note that bias is treated like a weight that goes with a constant input value of 1.

#3. Next implement the method that learns according to the above formula.

```
def learn(self, input, desired):
    """ this LinearNeuron learns based on input and desired"""
```

For example, if we wanted our neuron to behave as an **and** gate, we might use the following method calls, which you can use to test your neuron.

```
neuron2.learn([0, 0], 0)
neuron2.learn([0, 1], 0)
neuron2.learn([1, 0], 0)
neuron2.learn([1, 1], 1)
```

If you display the result at the end, the weights and bias should have changed to 0.1

```
neuron2.learn([1, 1], 1)
neuron2.display()
```

If you then run

```
neuron2.operate([1, 1])
```

you should get output of 1, which is correct for an **and** gate. However, you will also get that result for

```
neuron2.operate([0, 1])
```

which is obviously not correct. In order to train the neuron, you need to present the same inputs and desired value multiple times, as described next.

Training

Training means to present the set of input and desired output values, using **learn**, over and over again, until it produces correct values.

#4. Implement the method

```
def train(self, inputs, outputs, epochs, verbose):
        """ train this LinearNeuron based on a list of inputs and
            a list of corresponding outputs
            over a specified number of epochs """
```

One *epoch* is the presentation of all training samples to the learn method, one at a time. In this case, the **inputs** is a lists of 2-element lists, and **outputs** are the corresponding desired output values. The verbose argument is a boolean (True or False), indicating whether to show the number of errors for the epoch at the end of the epoch. This can be useful for debugging.

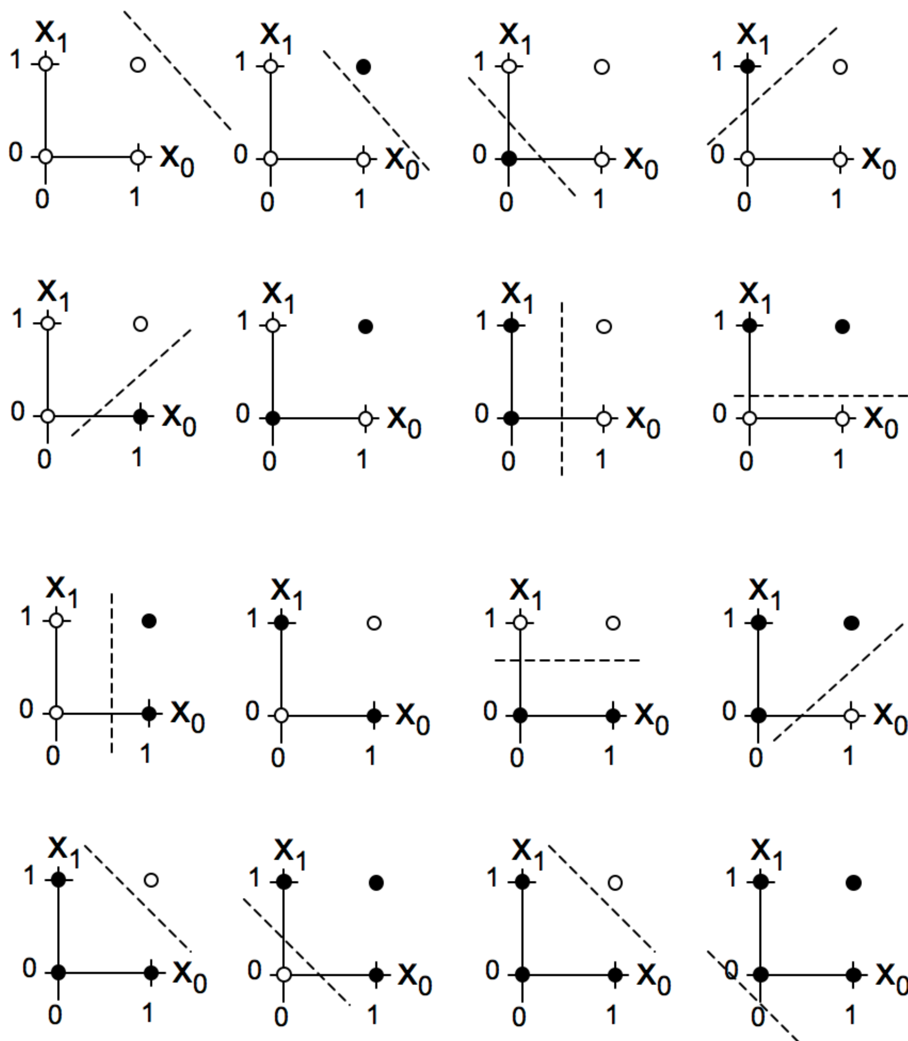
#5. Then implement the method

```
def train_and_assess(self, label, inputs, outputs, epochs, verbose):
        """ train based on a list of inputs and outputs,
            over a specified number of epochs,
            then perform a final assessment """
```

#6. Once you have this method work, try training the same neuron on all 16 2-argument Boolean functions. In other words, we will be reusing the same neuron, adjusting its weights differently for each function. The file **main.py** has the calls to do all of these. You only need to uncomment them.

Linear Separability

You may notice that a couple of the functions do not train in the prescribed number of steps. If we plot the output values for the 16 functions, using one dimension for each variable, we see the following plots. You will notice that in most cases we can draw a line that separates the combinations with output 1 (black circles) from those with output 0 (white circles). When such a line can be drawn, the function's training set is said to be *linearly separable*. Theory tells us that the points are linearly separable if, and only if, there is a linear neuron that can implement that function. Furthermore, if the points are linearly separable, then the neuron can be trained in a finite number of steps to establish the weight values for the function using the training method we have described.

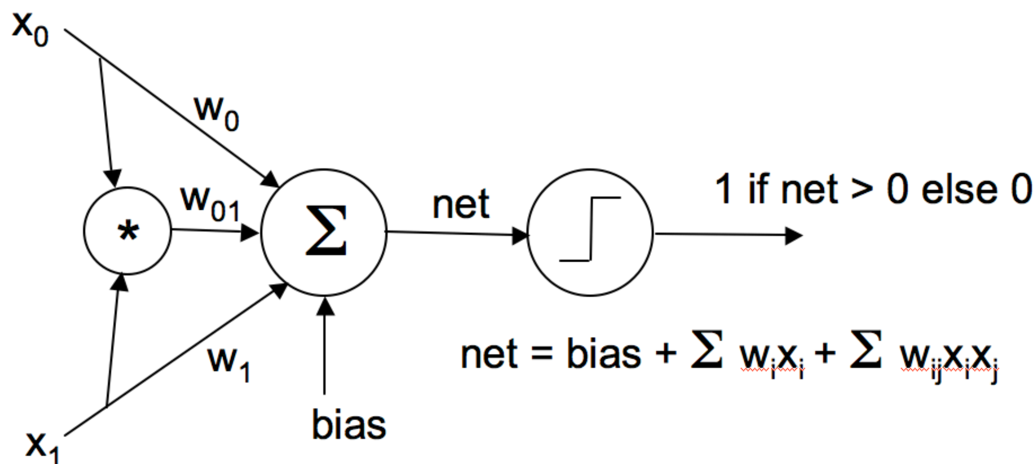


Plots of the 16 2-argument Boolean functions, showing separations

Quadratic Neurons

In order to handle the case of non-linearly-separable sets of points, we need additional capabilities beyond the linear neuron. One possibility is a "quadratic" neuron. This neuron adds derived inputs consisting of all *products* of all of the inputs two at a time with each other, and forms a linear combination of those. For example, if there are two inputs x_0 and x_1, then the term $x_0 x_1$ would be added as an *additional* input to the linear neuron. If there are three inputs x_0, x_1,and x_2, then we would add x_0x_1, x_0x_2, and x_1x_2.

The diagram below shows the case for two inputs.



A quadratic neuron

#7. Implement the class QuadraticNeuron, by deriving it from LinearNeuron. Most of the methods in LinearNeuron are probably reusable. It is only necessary to implement the ones shown below. These headers show the way to refer to the base class LinearNeuron as well. (*For simplicity*, we can use *all* products of inputs with inputs, which would include square terms and duplicate cross terms, for a total of n+n*n inputs to the summation.)

```
class QuadraticNeuron(LinearNeuron):

    def __init__(self, inputs, learning_rate):        # constructor/initializer
        LinearNeuron.__init__(self, inputs + inputs*inputs, learning_rate)

    def learn(self, input, desired):
        """ this QuadraticNeuron learns based on input and desired desired"""

    def train(self, inputs, outputs, epochs, verbose):
        """ train this Quadratic based on a list of inputs and corresponding outputs
            over a specified number of epochs """
```

```
def train_and_assess(self, label, inputs, outputs, epochs, verbose):
    """ train based on a list of inputs and outputs,
        over a specified number of epochs,
        then perform a final assessment """
```

#8. Once you have implemented this new class, see if a neuron based on it can learn the functions corresponding to non-linearly separable training sets.

Using Neurons as Classifiers

A common use for the technology we have developed is to *classify* vectors, such as results of clinical data. In this role, the input values can be reals, rather than just {0, 1}. The output can still be {0, 1}, indicating that a given input vector is either in or not in the class in question.

For example, in the file **cancer_data.py** are 683 samples of from clinical tests for breast cancer from the University of Wisconsin. Each sample consists of a list of 9 numbers representing the results of specific tests and a value 0 or 1 indicating a negative or positive diagnosis. If a neuron can classify these samples with reasonably high accuracy, it can serve as an advisory aid to a physician.

The Python variables **cancer_input** and **cancer_output** are derived from the samples:

cancer_input is a list of lists of the test outcomes
cancer_output is corresponding diagnoses

#9. Try training a 9-input LinearNeuron with this data. Use a learning rate of 0.05 and about 100 epochs, then assess the number of errors at the end of training.

#10. Then try training a QuadraticNeuron with the same data. A lower learning rate, 0.01, and 500 epochs are suggested. Does the QuadraticNeuron improve the performance of the classifier?

If you train in verbose mode, you will notice that the number of errors in successive epochs may fluctuate rather then decrease monotonically. If time permits, modify your training method to reserve the weights corresponding to the epoch with the fewest errors, and use those weights as the final result.

Summarize your results here:

1. Names of 2-input functions that were not learned perfectly with LinearNeuron:

2. Names of 2-input functions that were not learned perfectly with the QuadraticNeuron:

3. Errors in the cancer training set using LinearNeuron:

4. Errors in the cancer training set using QuadraticNeuron:

5. Errors in the cancer training set using QuadraticNeuron with the best weights reserved:

References:

Links to files and this document:
http://www.cs.hmc.edu/courses/2012/fall/cs42/pythonExercise.html