Practical 7

Week 9
Artificial Intelligence
Summer 2019

In this exercise sheet, we shall attempt to understand the working of a perceptron and how they are very effective for problems that are linearly separable. The reason it is important to learn the functioning of perceptrons is because they form the fundamental building blocks of all modern approaches to create more powerful and versatile neural networks.

Recall that perceptrons are the simplest neural networks. We shall begin by implementing a perceptron that is capable of implementing the AND logical function. Within this perceptron, the n binary inputs (usually given as a vector) are combined with a corresponding weight vector as a dot product:

$$z = \sum_{i=1}^{n} W_i x_i = W^T x$$

The result, z is known as pre-activation and to it is added a bias input (x_0) and weight (w_0). Usually, x_0 = +1, therefore even with the inclusion of the bias, the representation would still be:

$$z = \sum_{i=0}^{1} W_i x_i = W^T x$$

The weighted sum is passed onto an activation function, which can be a hard threshold function as follows:

$$\sigma(z) = \begin{cases} 1, z \geq 0 \\ 0, z < 0 \end{cases}$$

Therefore, the way the perceptron functions is that if the value of an activation function is 1, then it signifies that the neuron fires and produces an output and if 0 then the neuron stays dormant and does not produce an output.

Single-layer perceptron:
To implement a perceptron in Python, we start by importing the "`numpy`" package that has support for linear algebra and other mathematical functions. Then a class titled `perceptron` is created within which the number of inputs need to be specified. The "+1" added to `input-size` indicates the presence of a bias.

```python
import numpy as np

class Perceptron(object):
    """Implements a perceptron network"""
    def __init__(self, input_size):
        self.W = np.zeros(input_size+1)
```

Next, the activation function is to be implemented, which would work as explained above:

```python
def activation_fn(self, x):
    return 1 if x >= 0 else 0
```

With the pre-activation and activation functions ready, the next step would be to run the perceptron through the pre-activation and return an output. This is called prediction:

```python
def predict(self, x):
    x = np.insert(x, 0, 1)
    z = self.W.T.dot(x)
    a = self.activation_fn(z)
    return a
```

The Perceptron Learning Algorithm:

We've defined a perceptron as above, but how do perceptrons learn? The basic idea is to run each example input through the perceptron and, if the perceptron fires when it shouldn't have, inhibit it. If the perceptron doesn't fire when it should have, excite it. How to excite or inhibit? By changing the weight vector. The weight vector is the parameter that needs to keep changing until we can correctly classify each of our inputs. This is shown as:

$$w \leftarrow w + \Delta w$$

We need to determine what is a "good" Δw. This is determined by computing the error between the desired output and the predicted output.

$$e \leftarrow d - y$$

In case of binary outputs, when d and y are the same we get 0. When they are different, we can get either 1 or -1. This directly corresponds to exciting and inhibiting the perceptron. This error is multiplied with the input to tell the perceptron to change the weight vector in proportion to the input. This is done as:

$$w \leftarrow w + \eta \cdot e \cdot x$$

The parameter η is the learning rate. It is a scaling factor that specifies by how much should the weight factor be updated with. This is not "learnt" by the perceptron. For perceptrons, the Perceptron Convergence Theorem says that a perceptron will converge (weights will settle), given that the classes are linearly separable, regardless of the learning rate. Therefore, when the error is 0, i.e., the output is what we expect, then we don't change the weight vector at all. When the error is nonzero, we update the weight vector accordingly.

Thus, before the perceptron can begin to learn, we need to add the learning rate and the number of epochs to definition of the perceptron. So edit the `perceptron` class as follows:

```python
def __init__(self, input_size, lr=1, epochs=10):
    self.W = np.zeros(input_size+1)
    # add one for bias
    self.epochs = epochs
    self.lr = lr
```

An epoch is one complete run of the training data through the perceptron. A perceptron usually needs more than one epochs before they are adequately trained on a given training data.

Finally, a function is created that accepts the actual and desired inputs and runs the perceptron learning algorithm. The weights are updated for a number of epochs, and iterated through the entire training set. The bias is inserted into the input when performing the weight update. Then the prediction is created, the error is computed and the update rule is applied.

```python
def fit(self, X, d):
    for _ in range(self.epochs):
        for i in range(d.shape[0]):
            y = self.predict(X[i])
            e = d[i] - y
            self.W = self.W + self.lr * e * np.insert(X[i], 0, 1)
```

With the perceptron ready, the training data can be given to it as follows:

```python
if __name__ == '__main__':
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    d = np.array([0, 0, 0, 1])

    perceptron = Perceptron(input_size=2)
    perceptron.fit(X, d)
    print(perceptron.W)
```

Exercise 1: Combine the different code snippets given above to make the perception functional. Explain the output you obtain. Do you think the perceptron has been successful in classifying the outputs of the AND function? Try to verify it.

Exercise 2: How many epochs were specified within the perceptron learning rule? Do you think the perceptron could be successful in classifying the outputs of the AND function with a lesser number of epochs? Try to determine the minimum number of epochs needed to successfully train the perceptron.

Exercise 3: Change the activation function to a logistic function and observe the behaviour of the perceptron. Is there any difference in its performance in terms of its classification accuracy, learning rate, number of epochs needed etc.?

Exercise 4: Modify the above code to make the perceptron implement the OR function.

Exercise 5: Modify the above code to make the perceptron implement the NOT function.

Exercise 6: Combine the codes generated above and the model given in the lecture to create a multi-layer perceptron that implements the XOR function.

-END-