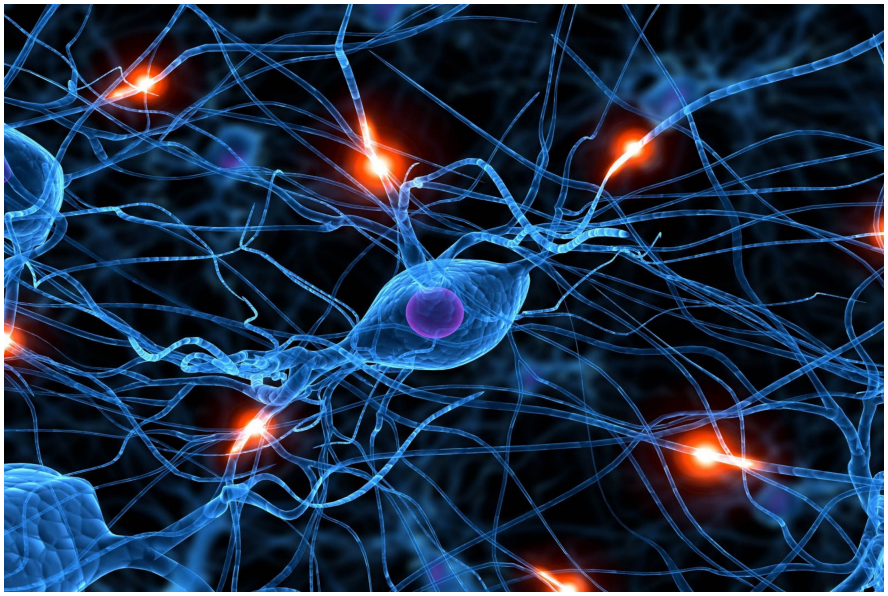
CS30 - Neural Nets in Python



We will experiment with neural networks using a simple software package written in Python. You can find the package at:

`http://www.cs.pomona.edu/~dkauchak/classes/cs30/assignments/assign6/cs30neural.txt`

Create a new file in Wing, copy the contents of this file and save it as `cs30neural.py`.

1 Usage

It is easy to use the package. First, make sure to import the contents of the package:

```
from cs30neural import *
```

Here is a simple example of creating, training, and testing a network that has two input nodes, five hidden nodes, and one output node.

```
nn = NeuralNet(2, 5, 1)
nn.train(xor_training_data)  #notice that xor_training_data is defined below
print nn.test(xor_training_data) # we can test on training, though it's not ideal
```

During training, we want to give the network labeled examples. An example, consists of a tuple of two things:

```
(input, output)
```

All input and output values are specified as *lists*. For example,

```
([1.0, 1.0], [0.0])
```

represents an input output pair mapping input 1, 1 to output 0.

The training data is then a *list* of input-output pairs. For the classic XOR example, the training data would be as follows:

```
xor_training_data = [([0.0, 0.0], [0.0]),
                     ([0.0, 1.0], [1.0]),
                     ([1.0, 0.0], [1.0]),
                     ([1.0, 1.0], [0.0])]
```

The testing data is also presented in the same format.

If you apply the model to test data, you will get back a list of *triples:*

```
(input, desired-output, actual-output)
```

This format that is convenient for further analysis by computer, but it is difficult to read when printed directly. You might want to print it one triple to a line:

```
for triple in nn.test(testing_data):
    print triple
```

If you unpack the triple, you can ask more interesting questions, for example to print only the differences between the desired and actual outputs:

```
for triple in nn.test(testing_data):
    (input, desired, actual) = triple # unpack the triple
     # remember outputs are lists, so we need to get the first item from the list
     print desired[0] - actual[0]
```

If you want to evaluate just one input, you may do it with the `evaluate` function. To see the output for a particular list of inputs, for example, you would write the following.

```
print nn.evaluate(input_list)
```

2 The Neural Network Class

The software we're playing with represents a neural network as an *object* of the `NeuralNet` *class.* We've already seen instances of this, for example, *lists* are a class of objects.

The main difference between a `list` object and our neural network object is that there is no special way for constructing neural network objects (to create lists we use the square brackets, `[]`). Instead, there is a special function called a *constructor* that takes some number of arguments (in our case, three) and creates a neural network object. Once you have that object, you can call methods on it just like you would any other object.

In the following summary, the neural network object is named `nn`. You may, of course, give your networks any name you like.

`nn = NeuralNet(num_input, num_hidden, num_output)` this calls the constructor and creates a network with the specified number of nodes in each layer. The initial weights are random values between -2.0 and 2.0. Notice that all of the other methods below are called *on* a neural network object that has been created.

`nn.evaluate(input)` returns the output of the neural network when it is presented with the given input. Remember that the input and output are *lists.*

`nn.train(training_data)` carries out a training cycle. As specified earlier, the training data is a list of input-output pairs. There are four optional arguments to the `train` function:

`learning_rate` defaults to 0.5.

`momentum_ractor` defaults to 0.1. The idea of momentum is discussed in the next section. Set it to 0 to suppress the affect of the momentum in the calculation.

`iterations` defaults to 1000. It specifies the number of passes over the training data.

`print_interval` defaults to 100. The value of the error is displayed after `printInterval` passes over the data; we hope to see the value decreasing. Set the value to 0 if you do not want to see the error values.

You may specify some, or all, of the optional parameters by name in the following format.

```
nn.train(training_data,
         learning_rate=0.8,
         momentum_factor=0.0,
         iterations=100,
         print_interval=5)
```

`nn.test(testing_data)` evaluates the network on a list of examples. As mentioned above, the testing data is presented in the same format as the training data. The result is a list of triples which can be used in further evaluation.

`nn.get_IH_weights()` returns a list of lists representing the weights between the input and hidden layers. If there are t input nodes and u hidden nodes, the result will be a list containing t lists of length u.

`nn.get_HO_weights()` returns a list of lists representing the weights between the input and hidden layers. If there are u hidden nodes and v output nodes, the result will be a list containing u lists of length v.

`nn.use_tanh()` changes the activation function from the sigmoid function to a commonly used alternative, the hyperbolic tangent function.

3 Examples

The standard XOR example.

inputs		output
0	0	0
0	1	1
1	0	1
1	1	0

Testing for "greater than." The data come in two parts: the positive facts and the negative ones.

inputs		output
0.2	0.0	1
0.4	0.2	1
0.6	0.4	1
0.8	0.6	1
1.0	0.8	1

inputs		output
0.0	0.2	0
0.2	0.4	0
0.4	0.6	0
0.6	0.8	0
0.8	1.0	0

Squaring a number.

input	output
0.0	0.00
0.2	0.04
0.3	0.09
0.4	0.16
0.6	0.36

Voter preferences. This is training information. Voters were asked to rate the importance of some issues and then to identify their party affiliation. The inputs are, from left to right, the importance of budget, defense, crime, environment, and social security. They are ranked on a scale of 0.0 to 1.0. The outputs are 0.0 for Democrat and 1.0 for Republican.

inputs					output
0.9	0.6	0.8	0.3	0.1	1.0
0.8	0.8	0.4	0.6	0.4	1.0
0.7	0.2	0.4	0.6	0.3	1.0
0.5	0.5	0.8	0.4	0.8	0.0
0.3	0.1	0.6	0.8	0.8	0.0
0.6	0.3	0.4	0.3	0.6	0.0

Unidentified. Can you determine the underlying rule?

inputs						output
1	0	1	0	0	0	1
1	0	1	1	0	0	1
1	0	1	0	1	0	1
1	1	0	0	1	1	1
1	1	1	1	0	0	1
1	0	0	0	1	1	1
1	0	0	0	1	0	0
0	1	1	1	0	1	1
0	1	1	0	1	1	0
0	0	0	1	1	0	0
0	1	0	1	0	1	0
0	0	0	1	0	1	0
0	1	1	0	1	1	0
0	1	1	1	0	0	0

4 Theory and Implementation Details

For those that are curious, this section gives more detail on exactly how the neural network learning algorithm works. This material goes beyond what I expect you to know for the class, however, I've included it here in case you're curious and would like to learn more.

Data values. Suppose that we have a network with t input nodes, u hidden nodes, and v output nodes. The state of the network is represented by the values below.

$$\begin{array}{llll} I_0 & I_1 & \ldots & I_{t-1} \quad I_t \\ H_0 & H_1 & \ldots & H_{u-1} \quad H_u \\ O_0 & O_1 & \ldots & O_{v-1} \end{array}$$

Notice that there is an extra input node and an extra hidden node. These are the *bias* nodes; I_t and H_u always have the value 1.0. The bias nodes are handled automatically; you do not have to account for them in the code you write.

There are two matrices for weights. The value $W_{i,j}^{IH}$ is the weight assigned to the link between input node i and hidden node j, and $W_{j,k}^{HO}$ is the weight between hidden node j and output node k. The two matrices are of size $(t+1) \times (u+1)$ and $(u+1) \times v$, respectively.

The change matrices keep track of the most recent changes to the weights. The values are $C_{i,j}^{IH}$ and $C_{j,k}^{HO}$, with notation analogous to the weights.

Activation function. The most commonly used activation function is the sigmoid function, defined by $\sigma(x) = 1/(1+e^{-x})$. Its derivative given by the formula $D_\sigma(x) = \sigma(x)(1-\sigma(x))$. The values of the sigmoid function are always between 0.0 and 1.0.

The activation function is used to compute the value at the hidden and output nodes. Its derivative is used to compute adjustments to the weights.

An alternative to the sigmoid function is the hyperbolic tangent, given by the formula $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$ and its derivative is given by the formula $D_{\tanh}(x) = (1 - \tanh^2(x))$. The values of the hyperbolic tangent range between -1.0 and 1.0.

Computing the output values. When presented with t input values, the network is evaluated by computing the hidden values and then the output values. The input values are assigned to I_0 through I_{t-1}. (Remember that I_t is always 1.0.) The jth hidden value is

$$\begin{aligned} H_j &= \sigma\left(I_0 W_{0,j}^{IH} + I_1 W_{1,j}^{IH} + \ldots + I_t W_{t,j}^{IH}\right) \\ &= \sigma\left(\textstyle\sum_{i=0}^{t} I_i W_{i,j}^{IH}\right). \end{aligned}$$

Once we know all the hidden values, we can similarly compute the output values. The kth output value is

$$\begin{aligned} O_k &= \sigma\left(H_0 W_{0,k}^{HO} + H_1 W_{1,k}^{HO} + \ldots + H_u W_{u,k}^{HO}\right) \\ &= \sigma\left(\textstyle\sum_{j=0}^{u} H_j W_{j,k}^{HO}\right). \end{aligned}$$

Training the network. We start with a collection of known input and output values. For a single input-output pair, we evaluate the input to obtain the *actual* output values O_0 through O_{v-1}. We compare these values with the *desired* output values Q_0 through Q_{v-1} to obtain the "error" or "required change" at each output node. The change at output node k is

$$\delta_k^O = (Q_k - O_k) D_\sigma(O_k).$$

Notice that the change is multiplied by the derivative of the activation function at the actual output value.

We apportion the "responsibility" for the error at an output node to the hidden nodes according to the weights. The change at hidden node j is

$$\begin{aligned}\delta_j^H &= \left(W_{j,0}^{HO}\delta_0^O + W_{j,1}^{HO}\delta_1^O + \ldots + W_{j,v-1}^{HO}\delta_{v-1}^O\right) D_\sigma(H_j) \\ &= \left(\sum_{k=0}^{v-1} W_{j,k}^{HO}\delta_k^O\right) D_\sigma(H_j).\end{aligned}$$

The next step is to adjust the weights between the input and hidden layer, using the errors in the hidden layer. The weight $W_{i,j}^{IH}$ becomes

$$W_{i,j}^{IH} + L\delta_j^H I_i + MC_{i,j}^{IH}.$$

The weight is adjusted by two terms. One involves the *learning rate, L*. It is usually a value between 0 and 1. In our software, the default value is 0.5. A higher rate may make the learning go faster, but it may also take too large steps causing the weights to gyrate and never settle down.

The other term involves the *momentum factor, M*. It tries to capitalize on the idea that if we moved a weight in one direction the last time, it may speed up the learning if we moved it a little further in the same direction this time. The momentum factor is usually rather small; the default value in our software is 0.1. In order to use the momentum factor, we have to remember the previous change. We record it in the change matrix after updating $W_{t,u}^{IH}$, by setting $C_{i,j}^{IH}$ to

$$\delta_j^H I_i.$$

Finally, we update the weights and changes between the hidden and output layers in the same manner. The weight $W_{j,k}^{HO}$ becomes

$$W_{j,k}^{HO} + L\delta_k^O H_j + MC_{j,k}^{HO},$$

and the change $C_{j,k}^{HO}$ becomes

$$\delta_k^O H_j.$$

Computing the error. The error for one evaluation is defined by

$$\left((Q_0 - O_0)^2 + (Q_1 - O_1)^2 + \ldots + (Q_{v-1} - O_{v-1})^2\right)/2$$

The whole training cycle described above—evaluate the output, compute the errors, and adjust the weights and changes—is executed for each element in the training data. The error for one pass over the training data is the sum of the individual errors. The formulas for adjusting the weights are designed to minimize the sum of errors. If all goes well, the error will decrease rapidly as the process is repeated, often with thousands of passes across the training data.

Variations. Our software works with only the most common type of neural network. There are many variations, including these:

- Some formulations omit the bias node in the hidden layer; others omit both bias nodes.
- Our software connects each node in one layer to each in the succeeding layer; sometimes some of the connections are omitted. One can argue that it is not necessary to leave out a connection; ideally, the learning process will discover that the weight is zero. But if we know something about the network and the significance of the nodes, we may know *a priori* that the weight is zero, and omitting the connection will speed up the learning process.
- Occasionally, there is more than one hidden layer.
- Sometimes there are edges from the output layers back to the input or hidden layer. These add cycles which give the network a form of "memory" in which one calculation is influenced by the previous calculation.