Neural Networks in Python

COMP_SCI 295

The goal of this assignment is to give an introduction to using neural networks. Neural networks are a machine learning concept used for tasks such as facial recognition, translation, speech generation and more.

We want to give you a chance to play around with and get a sense of how neural nets work. We are **NOT** asking you to code a neural network. Neural nets are an incredibly complicated concept with many varieties. For this assignment we'll focus on the most standard of these implementations, a multilayer network with feed forward nodes (don't worry if this sounds like gobbeldeygook we'll discuss it as we get to it).

Usage

First here is an example of creating, training and testing a network using the class we've provided. The neural net has two input nodes, five hidden nodes, and one output node (if you've used neural nets before we're only doing single hidden layer networks though feel free to have as many nodes in that layer as you'd like).

```
nn = NeuralNet(2, 5, 1)
# notice that xor_training_data & xor_testing_data are defined further below
nn.train(xor_training_data)
print(nn.test(xor_testing_data))
```

The training data is a list of input-output tuples (pairs). Each item in this tuple - the input and output separately are specified as *lists*. In this case, our input is a list of two elements and our output is a list of one element. Here is an example set of training data:

```
# each row is an (input, output) tuple
xor_training_data = [
      # input      # output
    ([0.0, 0.0],    [0.0]), # [0, 0] => 0
    ([0.0, 1.0],    [1.0]), # [0, 1] => 1
    ([1.0, 0.0],    [1.0]), # [1, 1] => 1
    ([1.0, 1.0],    [0.0])  # [1, 0] => 0
]
```

Once training has finished there are several methods to actually use our net. The three methods are `test`, `test_with_expected` and `evaluate`. Here's a little more detail on each (for a lengthier description see `The Neural Network Class` section further below:

- `evaluate(self, inputs: List[Any]) -> List[float]` - takes a single input (which remember is a list itself) and returns the output as a list (even if the output is only one element). For our `xor` example we might use this method like so: `trained_net.evaluate([0.0, 1.0]) # should return [1.0]`. The two other methods in this list each use `evaluate` in their implementation.
- `test(self, data: List[List[Any]]) -> List[Tuple[List[Any], List[Any]]]` - takes a list of inputs and returns a list of (input, output) tuples
- `test_with_expected(self, data: List[Tuple[List[Any], List[Any]]]) -> List[Tuple[List[Any], List[Any], List[Any]]]:` - takes a list of (input, expected_output) tuples and returns a list of (input, expected_output, actual_output) triples. It might seem weird to pass in expected outputs

but this sort of method can be helpful when trying to evaluate the accuracy of your model to compare expected and actual outputs.

Testing data is also provided as a list of inputs but here you can choose whether or not to pass in a list of (input, expected_output) tuples for testing or just a list of inputs depending on which method you use (see the list above for more detail)

The output of either testing function can be difficult to read. You can make it easier to read if you print each result on its own line:

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

or if you're using `test_with_expected` print only the differences between the desired and actual outputs:

```
print([expected[0] - actual[0] for _, expected, actual in nn.test(testing_data)])
```

The Neural Network Class

The neural network class represents a neural network with three layers - input, hidden, output.

- `__init__(self, n_input, n_hidden, n_output)`: creates a network with the specified number of nodes in each layer. The initial weights are random values between -2.0 and 2.0.
- `evaluate(self, input)`: returns the output of the neural network when it is presented with the given input. Remember that the input and output are *lists*.
- `compute_one_layer(self, curr_layer, num_next_layer, weights, is_hidden_layer)`: calculates feed forward propagation over one layer of the neural net.
- `test(self, 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 pairs which can be used in further evaluation.
- `test_with_expected(self, data, expected)`: Identical to `test` except the `data` input is now a list of tuples. Each tuple is an `(input, expected_result)` pair. Here `input` is itself a list of values, i.e. for `XOR` one of the `input` lists might look like `[0, 1]`. `expected_result` is the expected value given that input (essentially ground truth). This format is useful when testing as one would like to see how the model compares to the actual correct values. The result is a list of triples which can be used in further evaluation.
- `train(self, data, learning_rate, momentum_factor, iters, print_interval)`: carries out a training cycle. `data` is the only required argument here, all others have defaults. 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_factor` - 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 `print_interval` 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.
- `back_propagate(self, inputs, desired_result, learning_reate, momentum_factor)`: This is where the magic happens. This function performs forward propagation on the input to determine error then adjusts weights on each node in the network to reduce error. This is a massive oversimplification of the heart of a neural net, if you'd like some more detail here are a few different resources - a video, an article (which actually references the video) and a longer form online book if you have the time/interest.
- `get_ih_weights(self)`: 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.

- `get_ho_weights(self)`: returns a list of lists representing the weights between the hidden and output layers. If there are u hidden nodes and v output nodes, the result will be a list containing u lists of length v.
- `switch_activation(self)`: changes the activation function from the sigmoid function to a commonly used alternative, the hyperbolic tangent function.

Theory and Implementation Details

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.

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

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 $W^{IH}_{i,j}$ is the weight assigned to the link between input node i and hidden node j, and $W^{HO}_{j,k}$ 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 $C^{IH}_{i,j}$ and $C^{HO}_{j,k}$, 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 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

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

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

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

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" 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 error 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 error at hidden node j is

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

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 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 code recreates a standard neural net, there are many variations, including the following:

- 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 forming a `fully connected` neural net. Some neural nets omit some connections between nodes. 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.
- Many (if not most) neural nets use more than one hidden layer.
- Sometimes there are edges from the output layers back to the input or hidden layer. These are then called `recurrent` neural networks. These add cycles which give the network a form of "memory" in which one calculation is influenced by the previous calculation.