PGML Library Documentation

Brad Chapman

Last Updated: 29 July 01

Contents

Chapter 1

Module Documentation

1.1 Neural Networks

1.1.1 What are Neural Networks?

Neural Networks are a machine learning approach to mapping a set of inputs to some kind of output. Basically, I think of them as a factory that converts one thing into another. The nice thing of neural networks, as opposed to mathemetical functions that you learn about in school, is that you don't need to know the exact formula that is used to convert the inputs into the outputs. Instead, the neural network is an architecture which can be used to learn the relationship between inputs and outputs.

Setting up a neural network requires having a set of training examples which the neural network will use to "learn" the input/output relationship. Based on what you're trying to accomplish, you will have to decide on what you inputs and outputs are for each training example. For instance, it you were trying to predict whether a stretch of DNA sequence is a promoter, you would have your DNA sequences as input and your output would be a prediction about whether or not the sequence is a promoter. This is the basic idea, the specifics of doing this will be covered just a little later on.

This documentation will only work through how to use neural networks in this python implementation. If you want to learn all the nitty gritty of neural networks, there are tons of books and courses out there that will teach them. One book I've found which specifically describes how neural networks are used is **Neural Networks and Genome Informatics** by Wu and McLarty (http://www1.fatbrain.com/asp/bookinfo/bookinfo.asp?theisbn=0080428002).

1.1.2 Back-Propagation Neural Networks

This library implements back-propagation neural networks. This is one of the simpler neural network architectures and one of the first taught in books and classes (which is why I decided to use it). This section describes the basic setup of back-propogation neural network, and how to encode it.

Almost all back-propogation neural networks are set up with three layers. The input layer is what you are putting into the neural network. The output layer is the stuff you want to get out. The third layer is the hidden layer (I like to think of it as a "magic layer."). This layer allows you to do much more complicated analyses, and basically any problem which you can do usefully with neural networks should be able to be done with these three layers.

Each layer has nodes, which are the individual subcomponents making up the network. The input and output layers have node numbers fixed by the problem you are working on. Based on your data you'll have certain inputs that you need to put in, and you'll be getting out specific outputs. By contrast the hidden layer can have a variable number of nodes, and varying this can help you play around with how well your neural network works.

Figure 1.1 shows what we've been talking about in a graphical form. This is a neural network with 2 input nodes, 3 hidden nodes, and 2 output nodes. The arrows show the idea that the nodes are fully connected together.

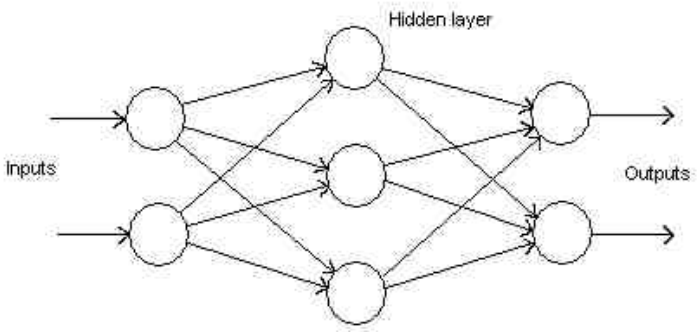

Figure 1.1: Example of a back-propagation neural network with three layers: input, hidden and output

Creating this type of network with this library is pretty straightforward. To set up a neural network corresponding to that shown in figure 1.1, we would need to do:

```
from Bio.PGML.NeuralNetwork.BackPropagation import Layer
from Bio.PGML.NeuralNetwork.BackPropagation.Network import BasicNetwork

output = Layer.OutputLayer(2)
hidden = Layer.HiddenLayer(3, output)
input = Layer.InputLayer(2, hidden)

network = BasicNetwork(input, hidden, output)
```

This will give us a neural network we're ready to work with. All that needs to be done now is to train this neural network and it'll be ready to work.

1.1.3 Organizing your training examples

Before we jump ahead with actually training our neural network, we first need to step back and think about how we are going to deal with our training examples. Training examples are very precious since they are the only thing that is going to help you make your neural network useful and test how well it is running.

Getting a TrainingExample

When you start setting up a neural network, you'll have a set of training data. To use a slightly biological example, let's pretend we are trying to learn to classify DNA sequences as being a GC rich motif. We'll define a sequence as being GC rich if it has greater than 60 percent Gs and Cs, and if it is longer than 10bp. Additionally, we'll define a second item of iterest, whether or not the sequence comes from a CpG island (a region with a high number of CG dinucleotides). This is a really simplified example, since we would definitely need more than just two inputs to accomplish our goal, but we're just demonstrating.

With this type of problem description, we might have inputs like:

Sequence	GC precent	Length	Is GC rich?	Is a CpG island?
CGCGCGCGCGCG	100	12	1	1
AAAAAATTTT	0	10	0	0
GCGCGAG	86	7	0	0
GGGGAAGGGGG	82	11	1	0

In reality we'd also have a lot more training examples, but you get the point. To get these four training examples set up for use in our neural network, we would just do the following:

```
from Bio.PGML.NeuralNetwork.Training import TrainingExample

examples = []
examples.append(TrainingExample([100, 12], [1, 1]))
examples.append(TrainingExample([0, 10], [0, 0]))
examples.append(TrainingExample([86, 7], [0, 0]))
examples.append(TrainingExample([82, 11], [1, 0]))
```

The `TrainingExample` class is just a simple class to hold our inputs and outputs. The first argument is the list of inputs, and the second is the list of output. Doing the above gives us a list of training examples we can now work with.

Partitioning the training examples

Once the training examples are set up for input into the neural network, we next need to decide how we are going to split up the training examples. In general, these examples will be divided up into three categories:

1. Training Examples – These data are used for the actual training of the network.
2. Validation Examples – These data are used to validate the network during the training process. By using these data independently of the training examples, we can be sure the network is trained independently of noise in the training examples.
3. Testing Examples – These data are used to test the network after it is trained. Since we know the expected results for these examples, we can see how well the network will do with completely new data it has never seen.

In this implementation, the training examples are organized using an ExampleManager. This manager takes all of the examples and partitions them into the three different categories so they can be used. By default, the manager will set 40 percent of the examples for training, 40 percent for validation, and the remaining 20 percent for testing. This can be changed by setting the `training_percent` and `validation_percent` arguments to the initializer. So we can organize our training examples from above by:

```
from Bio.PGML.NeuralNetwork.Training import ExampleManager

manager = ExampleManager(training_percent = 0.4, validation_percent = 0.4)
manager.add_examples(examples)
```

This will randomly partion the examples we add into three attributes of the manager:

- `train_examples`
- `validation_examples`
- `test_examples`

The examples are partitioned according to the percentages set above. Now that we've got our examples ready to go, we can go ahead and use the network.

1.1.4 Running the Neural Network

Training

Before using a network for actually predicting things, we first need to teach the network about our data by training it. The `train` function of our neural network will be used for this, but first we need to look at the arguments that should be passed to this function:

- `training_examples` – This is a list of examples that will be used to training the network. If you are using the ExampleManager, this should be `manager.train_examples`.
- `validation_examples` – This is a list of examples to be used for validating the network. With the ExampleManager, this would be `manager.validation_examples`.
- `stopping_criteria` – This is a function that will be used to tell the network when to stop training. This should be a reference to a callable object (ie. the name of a function), and the object should take 3 arguments: the number of iterations, the current validation error and the current training error. The function should then make a decision based on these whether or not to stop training, and return a 0 or 1 appropriately.
- `learning_rate` – How quickly the network should learn. The lower the number, the slower the network will "learn." A common value for this might be 0.5. This value can be used to tune how the network learns.
- `momentum` – How much information the network should use from a previous iteration and apply to the current iteration. A common value might be 0.1. Like the learning rate, this can be used to tune the network.

Now that we've got all of the arguments we need to pass to the trainer in order, here's an example of training a network:

```
def stop_function(num_iterations, validation_error, training_error):
    """Simple stop function that stops after 10 iterations."""
    if num_iterations >= 10:
        return 1
    else:
        return 0

network.train(manager.train_examples, manager.validation_examples,
              stop_function, 0.5, 0.1)
```

When the network training stops, we'll have an appropriately trained network that we can now think about using.

Predicting

Once the network had been trained, we can use it for what we wanted it for: predicting things. Normally the first thing to do after training a network is to test out its predictions on your `test_examples` and make sure it is predicting things properly.

Getting predictions out of the trained network is very easy – you just need to do:

```
prediction = network.predict(inputs)
```

The `inputs` should be a list of inputs corresponding to the number of input nodes (something like `[65, 15]` for the example we've been following so far. The returned predictions will also be a list corresponding the number of output nodes (something like `[1, 0]`).

So in order to test whether or not our network is predicting properly on our test examples, we would do:

```
for test_example in manager.test_examples:
    prediction = network.predict(test_example.inputs)
    print "for %s, expected %s, got %s" % (test_example.inputs,
                                           test_example.outputs, prediction)
```

Once you are satisfied that the network is predicting correctly, you can then set it loose on your unknown inputs and let it predict!

1.1.5 Encoding Biological Sequences for input into Neural Networks

Once gaining an understanding of how neural networks work in general, the next big question is how they can be applied to information of interest to biologists. The NeuralNetwork library contains modules that help represent biological sequences as inputs into NeuralNetworks.

The major problem with using DNA sequences in neural networks is that they don't readily fit into the traditional thinking about how neural networks are organized. In neural networks, you have a fixed number of inputs into the system. However, the initial impulse for encoding a biological sequence is to encode each residue in the sequence as a neural network input. This won't work since the sequences you are dealing with are often different in size. Additionally, this probably won't work right since even if sequences are the same size, the "important" residues will not always be in the same absolute position along the sequence.

These problems require us to rethink how we should encode biological sequences. The approach that is taken is this library is to encode a sequence by counting the number of patterns that occur in the sequence. Each input into the neural network is thus a count of a particular pattern. So, sequences are thus encoded as a sum of the patterns that they contain.

The NeuralNetwork libraries contain three different ways to represent a sequence:

1. Motif – A motif is a simple sub-sequence; for instance, the sequence `GATCG` has the 4-residue length motifs `GATC` and `ATCG`.

2. Schema – A schema is a motif that can contain ambiguity characters. The ambiguity characters could be something generic like a `*` to represent any base, or more specific, like R to represent any purine (A or G). The advantage of schemas over motifs is that they often represent biological information better; in some cases only certain residues will be important and others will be allowed to vary. The added flexibility of schemas thus helps in representing your data.

3. Signature – A signature represents two motifs separated by arbitrary sequence. For instance, if you had the two sequences `AAAGGGTTT` and `AAACCCCCCCTTT`, they both contain the signature `AAA-TTT` (three As, an arbitrary number of other residues, then three Ts). The advantage of this representation is that it can represent things like insertions in DNA that maintain reading frame, or insertion of a variable loop in a protein structures.

First we'll discuss the general classes that are useful for encoding any biological sequence. Additionally, all of the encoding classes share some general features, which will be detailed. Finally, we'll discuss the nitty gritty specifics of each encoding type.

Generally useful classes for help in encoding

The overall structure of encoding classes

Motif specific information

Schema specific information

Signature specific information

1.2 Genetic Algorithms

1.3 Hidden Markov Models

1.4 Creating Pretty Graphics

Chapter 2

Cookbook Examples