Instructions for Creating and Running your own Neural Network Model
Cognitive Science, Fall 2010
Professor Blank, CS371

In this lab, we will use the Conx neural network package written in Python.

1) Log into one of the computers in the Computer Science Linux network with the password provided in class.
2) Start a Terminal (Menu → Applications → System Tools → Terminal)
3) At the Terminal prompt, type:

   export PYTHONPATH=/home/dblank

   Match the case of letters and spaces exactly. If you make a mistake, press the UP-ARROW and edit the command (or just re-type it).
4) At the Terminal prompt, type:

   idle

You are now running IDLE, a Python environment for running and editing programs. First, we will load the neural network package.

5) At the Python >>> prompt in IDLE, enter:
   from pyrobot.brain.conx import *

A transcript follows. The items that you type are in bold.

In this example, we will build a 2-2-1 feed-forward neural network that can learn based on seeing input patterns and associated teacher-provided "targets". We will learn one of the hardest problems for neural networks to learn, Exclusive-Or. Exclusive-Or says: "given two inputs, output a 1 if one (and only one) of the inputs is a 1; otherwise output a 0". As a truth table, that looks like:

	Input 1	Input 2	Target
Pattern 1	0	0	0
Pattern 2	0	1	1
Pattern 3	1	0	1
Pattern 4	1	1	0

The network will will build is a 2-2-1 network: it has 2 nodes in the input later, 2 nodes in the hidden layer, and 1 node in the output layer. The layers will be called "input", "hidden", and "output" and are created with the net.addLayers() command. The net.train() command will automatically begin to adjust the weights.

First, let's build an 2-2-1 network, and set the training inputs and targets.

```
IDLE 2.6
>>> from pyrobot.brain.conx import *
Conx, version 2571 (psyco enabled)
>>> net = Network()
Conx using seed: 1284985091.03
>>> net.addLayers(2, 2, 1)
>>> net.setInputs( [[0, 0], [0, 1], [1, 1], [1, 0]])
>>> net.setTargets([[0], [1], [0], [1]])
>>> net.train()
Epoch #    25 | TSS Error: 1.0051 | Correct: 0.0000 | RMS Error: 0.5013
Epoch #    50 | TSS Error: 1.0130 | Correct: 0.0000 | RMS Error: 0.5033
...
Epoch #  1225 | TSS Error: 0.7489 | Correct: 0.2500 | RMS Error: 0.4327
Epoch #  1250 | TSS Error: 0.5588 | Correct: 0.7500 | RMS Error: 0.3738
Final #  1261 | TSS Error: 0.4393 | Correct: 1.0000 | RMS Error: 0.3314
```

That's it! The network found a set of weights (just numbers) that, when given the input patterns, will output the correct output (eg, it is close to the given target). It took 1,261 “epochs” or sweeps through the data. That is, it had to see each pattern 1,261 times.

Let's actually see the details of how it works. Let's turn learning off, and interactivity on, and then sweep through the data.

```
--------------------------------------------------------------------------------
>>> net.learning = False
>>> net.interactive = True
>>> net.sweep()
----------------------------------Pattern # 1
Display network 'Backprop Network':
==============================
Layer 'output': (Kind: Output, Size: 1, Active: 1, Frozen: 0)
Target    :  0.00
Activation:  0.21
==============================
Layer 'hidden': (Kind: Hidden, Size: 2, Active: 1, Frozen: 0)
Activation:  0.42 0.03
==============================
Layer 'input': (Kind: Input, Size: 2, Active: 1, Frozen: 0)
Activation:  0.00 0.00
--More-- [<enter>, <q>uit, <g>o]
<Enter>
 ----------------------------------Pattern # 3
Display network 'Backprop Network':
==============================
Layer 'output': (Kind: Output, Size: 1, Active: 1, Frozen: 0)
Target    :  1.00
```

```
Activation:  0.66
==============================
Layer 'hidden': (Kind: Hidden, Size: 2, Active: 1, Frozen: 0)
Activation:  0.06 0.25
==============================
Layer 'input': (Kind: Input, Size: 2, Active: 1, Frozen: 0)
Activation:  1.00 0.00
--More-- [<enter>, <q>uit, <g>o]
<Enter>
 ----------------------------------Pattern # 4
Display network 'Backprop Network':
==============================
Layer 'output': (Kind: Output, Size: 1, Active: 1, Frozen: 0)
Target    :  1.00
Activation:  0.65
==============================
Layer 'hidden': (Kind: Hidden, Size: 2, Active: 1, Frozen: 0)
Activation:  0.06 0.25
==============================
Layer 'input': (Kind: Input, Size: 2, Active: 1, Frozen: 0)
Activation:  0.00 1.00
--More-- [<enter>, <q>uit, <g>o]
<Enter>
 ----------------------------------Pattern # 2
Display network 'Backprop Network':
==============================
Layer 'output': (Kind: Output, Size: 1, Active: 1, Frozen: 0)
Target    :  0.00
Activation:  0.37
==============================
Layer 'hidden': (Kind: Hidden, Size: 2, Active: 1, Frozen: 0)
Activation:  0.00 0.76
==============================
Layer 'input': (Kind: Input, Size: 2, Active: 1, Frozen: 0)
Activation:  1.00 1.00
--More-- [<enter>, <q>uit, <g>o]
<Enter>
(0.41809742054942256, 4, 4, {'output': [4, 4, 4, 4]})
>>>
```

You press the <Enter> key between each pattern. You can see the activation flow from bottom to top for each presentation of the pattern. The above example showed patterns 1, 4, 3, 2.

Now, let's test generalization: how well does the neural network do on input patterns that it hasn't seen before? We use the net.propagate() command:

```
>>> net.propagate(input=[0.1, 0.1])
array([ 0.368965],'f')
```

```
>>> net.propagate(input=[0.1, 0.1])
array([ 0.368965],'f')
>>> net.propagate(input=[0.1, 0.2])
array([ 0.44656304],'f')
>>> net.propagate(input=[0.1, 0.3])
array([ 0.5145424],'f')
>>> net.propagate(input=[0.1, 0.4])
array([ 0.56921721],'f')
>>> net.propagate(input=[0.4, 0.4])
array([ 0.65306389],'f')
>>> net.propagate(input=[0.4, 0.4])
```

For Wednesday, design your own neural network, and training patterns.

Some possible questions:

1) Does the size of the hidden layer affect learning? Speed of learning? Accuracy of learning?
2) How does the network generalize on patterns that you didn't train it on? How would you determine if it generalizes in an intelligent fashion?
3) You can try adjusting two other parameters in learning: the learning rate, and momentum:

```
>>> print net.epsilon
0.1
>>> net.setEpsilon(.2)
>>> net.momentum
0.9
>>> net.setMomentum(.8)
```

---

4) How does changing epsilon and momentum affect the learning?