Psych 186B, Winter, 2014
HW 6
Due 11:59pm February 26, 2014

20 points

In this model you will create a neural network simulation that performs a parity (odd/even) judgment on the input units: if the # of input units turned on is even then the output is 1, otherwise the output is zero.

The steps that you should take are as follows.

1. Set up the parameters of the model:
learning rate (lrate): for this problem, a learning rate of around 1 should be fine
number of input units: 8
number of hidden units: 3
number of output units: 1

2. Set up the patterns that the network will be trained on. This should be a matrix of ones and zeros, with the number of rows equal to the number of input units. The number of columns represents the number of patterns that the network will be trained on; for this exercise, 8 patterns should be sufficient. You can create these patterns by rounding random uniform variates created using rand().

3. Determine the desired output for each pattern, which is 1 if the number of ones in the input is even and zero if the number is odd. Hint: you can use the mod() function to determine whether the number of on-elements in odd or even.

4. Create two matrices for the weights that connect the input to hidden units (w_fg), and hidden units to the output unit (w_gh). Fill these matrices with uniform random numbers between –0.5 and 0.5.

5. Create a loop that will continue to iterate the model until one of two conditions is met:
- the sum of squared error between the observed and desired output is less than .01
- the number of epochs is greater than 1000 (an epoch is similar to an iteration, except that one epoch is one pass through all the input patterns)

Each loop through the model should perform the following operations (we provide pseudocode for some of the equations):

For each input pattern:

a) Pass the activation from the input units (which is simply the input pattern) to the hidden units:

```
input_to_hidden = w_fg * pattern
```

b) Determine hidden unit activation by passing the input to the hidden units through the activation function:

```
hidden_activation = activation_fn(input_to_hidden)
```

You should create a separate function (in its own file) called activation_fn that takes in a matrix of activation values and returns a sigmoid on those values:

```
function f=activation_fn(x)
  f=1./(1+exp(-x));
```

c) Pass the activation from the hidden units to the output units:

```
input_to_output = w_gh * hidden_activation
```

d) Determine output activity by passing the input to the output units through the activation function:

```
output_activation = activation_fn(input_to_output)
```

e) Compute the output error:

```
output_error = desired_output – output_activation
```

f) Determine the weight changes (dw) for each layer of connections.

g) Apply the weight changes:

```
w_fg = w_fg + dw_fg
w_gh = w_gh + dw_gh
```

Do steps a-g for all input patterns. After all patterns have been presented to the network once, one epoch has been completed.

h) Next, we want to compute the sum of squared errors (SSE) over all input patterns for the current epoch, after the weights have been updated. We can do this by repeating steps a-e, except with all input patterns at once rather than one pattern at a time. This will give us a matrix of the output errors for all patterns, and we can calculate the SSE as follows:

```
sse = trace(output_errors'*output_errors)
```

trace() takes the sum of the diagonal elements in a square matrix; if you are not familiar with this, you may want to confirm for yourself that this operation does indeed compute the sum of squares. Also, create a variable that will save the SSE value for each epoch, so that these values can be plotted later.

i) Print a brief report every 10 epochs, listing the epoch number and the SSE value.

Once the model has converged (SSE < .01), print two figures:
- a plot of the SSE as it changes over epochs
- a display of the matrices of desired and obtained outputs as images, using the imagesc() function. Show the images next to one another in a 1x2 subplot.

You will notice that on some occasions the model may not converge upon a solution to the problem, but instead will get stuck in a local minimum; if the model goes through 1000 epochs without converging (i.e., the SSE is still > .01), then the model should exit with a warning that it did not converge. When this happens, re-run the model (with a new set of random weights) until it converges.

After the model has converged, test the model's generalization abilities by creating a new set of patterns (again using rounded random numbers), and running those patterns through the model by performing steps a-e using the new patterns (note that you should not apply weight changes during the test). Does the model generalize well to the new stimuli? **Please quantify your answer!**