CogSci C131: Computational Models of Cognition
UC Berkeley, Spring 2014

Problem Set #4
Due: April 3rd, 2014 at 2:00pm

The problem sets in this class will be done using Matlab. You may need to use the computers in the EECS labs to get access to Matlab. If you have problems accessing these computers, contact the GSIs at `gsi-cogsci131@lists.berkeley.edu`.

Collaboration Policy

Class policy is that collaboration is permitted when solving problems, but each student must: (1) write his or her own code; (2) write the document containing his or her answers independently; and (3) list the people he or she worked with on his or her problem set.

Submission Instructions

Problems will require you to either turn in Matlab code, written answers, or both. All of the files you need are provided for you – you just need to fill them in with your answers. **Make sure to fill in your name and collaborators for each part of the homework.**

All parts that include Matlab code come with a validation script called `validatePN.m` which you can run by calling `validatePN()` from Matlab (replace `N` with the problem number, e.g. for Problem 1, run `validateP1()`). Make sure you run these functions for each part of the homework before submitting your assignment. **If your code does not pass the validation script, you will NOT receive credit.** Note, however, that this is not a grading script; it is entirely possible for incorrect answers to pass the validation script.

You will be submitting your homework on bSpace. For each problem set, you will submit a single zip file containing your filled-in files from each problem. The directory structure within your submission should be the same as when the problem set was released. The zip file should be named:

```
YourLastName_YourFirstName_ProblemSetNumber.zip
```

For example, if your name is Jane Doe, your submitted file should be named `Doe_Jane_N.zip`, where `N` is the problem set number. You can update your submission until the due date. If you make any changes to your submission after the due date, we will regard your problem set as having been turned in at the last time it was changed.

If you use Matlab on the computers in Cory 105 or log in with remote desktop, please note that files saved to the Desktop are removed when you log out. Instead, save any files to your user directory, `U:/`. The `U:/` drive can also be accessed through a shortcut on the Desktop.

Late Problem Set Policy

This problem set is due on **April 3rd, 2014 at 2:00pm**. You get **THREE LATE DAYS** total for the entire semester, to account for foreseeable minor crises. If you turn in a problem set late after using these late days, 20% of the total points on the problem set will be deducted for each 24 hours (or portion thereof) that it is late.

1 Hebbian Learning (6 points)

One of the simplest neural network learning algorithms is Hebbian learning, in which the weight between two nodes is increased when those nodes take on the same value, and decreased when they take on different values. If $\mathbf{x} = (x_1, \dots x_n)^T$ and $\mathbf{y} = (y_1, \dots, y_m)^T$ are $n \times 1$ and $m \times 1$ vectors representing the inputs and outputs to a neural network respectively, the weights of the neural network can be expressed in a $m \times n$ matrix $\mathbf{W}$. The networks predicted output $\hat{\mathbf{y}}$ is then:

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} \tag{1}$$

We train the neural network by providing it with a set of input-output pairs, $(\mathbf{x}, \mathbf{y})$. Hebbian learning adjusts the weights using the following equation for each input-output pair:

$$\Delta\mathbf{W} = \eta\mathbf{y}\mathbf{x}^T \tag{2}$$

That is, the change in the weight matrix $\mathbf{W}$ is determined by the outer product of the output and input vectors, multiplied by the learning rate η. Then, the updated weight matrix equals the old weight matrix plus $\Delta\mathbf{W}$.

$$\hat{\mathbf{W}} = \mathbf{W} + \Delta\mathbf{W} \tag{3}$$

Part A

Assume we want to learn what noises animals make based on their appearance. We might list four input features: chasing sticks, liking water, having whiskers, and being furry , and then represent dogs with $\mathbf{x}_{\text{dog}} = (1, 1, 1, 1)^T$ and cats with $\mathbf{x}_{\text{cat}} = (-1, -1, 1, 1)^T$. Likewise, we could have four output features: hissing, barking, neighing, and growling, with $\mathbf{y}_{\text{dog}} = (-1, 1, -1, 1)^T$ and $\mathbf{y}_{\text{cat}} = (1, -1 - 1, 1)^T$. Then $(\mathbf{x}_{\text{dog}}, \mathbf{y}_{\text{dog}})$ would be one possible input-output pair. Note that 1 indicates a logical `TRUE` and -1 indicates a logical `FALSE`.

In the function definition provided in `problem1/calculateWeights.m`, write code that takes in the current weight matrix $\mathbf{W}$, a matrix of training data inputs, and a matrix of training data outputs, and returns a matrix with the updated weights using Hebbian learning (Equation 2) on the given training data.

The matrix of training data inputs should have training instances as its columns. For example, if there were two training instances, $\mathbf{x}_{\text{dog}}$ and $\mathbf{x}_{\text{cat}}$, the matrix would have two columns: $[\mathbf{x}_{\text{dog}}, \mathbf{x}_{\text{cat}}]$. The matrix of training data outputs should have two columns corresponding to output instances: $[\mathbf{y}_{\text{dog}}, \mathbf{y}_{\text{cat}}]$.

Set the learning rate of $\eta = 0.25$ inside `problem1/calculateWeights.m`.

Part B

The file `problem1/Problem1.m` has the input and output feature vectors for dogs and cats defined for you. In the appropriate space in this file, call your `calculateWeights` function to find out the new weights after seeing one dog and one cat, given that the initial matrix $\mathbf{W}$ starts with all zeros. Save the new weights in the variable `updatedW`.

Using the value of $\mathbf{W}$ computed above (after seeing one instance of a cat and one of a dog), calculate the network's predicted output $\hat{\mathbf{y}}$ for inputs $\mathbf{x}_{\text{dog}}$ and $\mathbf{x}_{\text{cat}}$ (see Equation 1). In the appropriate spot in `Problem1.m`, assign the predicted values to the variables `yhatDog` and `yhatCat`.

Hint: To double-check that your code is functioning correctly, these predicted outputs should be equal to their respective training data outputs.

Part C

Now, imagine you saw an animal that was definitely furry and had whiskers, but you weren't sure about whether it liked water or chased sticks. We could represent the features of this animal as

$$\mathbf{x}_{\text{unknown}} = \alpha \mathbf{x}_{\text{dog}} + (1 - \alpha)\mathbf{x}_{\text{cat}} \tag{4}$$

for some value of α between 0 and 1.

Using the value of $\mathbf{W}$ computed in part (a), what is the predicted network output $\hat{\mathbf{y}}$ for different inputs $\mathbf{x}_{\text{unknown}}$ set by Equation 3 with $\alpha = 0.2$, 0.5, and 0.8? In the appropriate space in the provided file `Problem1.m`, compute and store these predictions in the variables `yhatAnimalX2`, `yhatAnimalX5`, and `yhatAnimalX8`.

Part D

Interpret these results in terms of the noises the animal might make. How does the kind of solution the neural network produces differ from the kind of thing we might expect from an account based on rules and symbols? Write your responses in the appropriate spot in the provided file `problem1/writtenAnswersPartD.txt`.

2 Layers and logic (4 points)

Part A

You will first train a one-layer feedforward neural network (a "perceptron") to implement a logical OR function. In order to use the neural network library we've chosen (`http://www.cs.cmu.edu/afs/cs/academic/class/15782-f06/matlab/`), you need to make a "training set" of input and output pairs for the function you want the network to learn. Your training set needs to be specified using two matrices: one will contain the "inputs" to the neural network, the other the "output". You will be training a network with two inputs and one output. The inputs and output can all take the values -1 or 1, where -1 stands for *false* and 1 stands for *true*, and you want to train the network to produce the output 1 if any of the inputs are 1, and -1 otherwise.

Letting `inputs` be the matrix for the training inputs (with one row for each input, and one column for each piece of training data we give to the network) and `ANDoutput` be the matrix for the training outputs (one row per output, one column for the desired output value for each column in the input), an appropriate training set for the logical AND function would be

```
inputs = [ -1 -1  1  1;
           -1  1 -1  1 ];
ANDoutput = [ -1 -1 -1  1 ];
```

You may refer back to the first problem set for more information on truth tables, but keep in mind that *false* values are represented with -1 rather than a 0 for this neural network.

Under Part A in the file `problem2/Problem2.m` define the training set matrices `inputs` and the output vector `ORoutput` based on the truth table for the logical OR function. We'll also go ahead and define what output we would expect from the logical XOR function, which we will use later in this problem. Using the same set of inputs, define the output vector `XORoutput` based on the truth table for the logical XOR function.

Part B

A one-layer feedforward neural network (a "perceptron") can be created and trained by calling the function `trainPerceptron()`. `trainPerceptron()` takes three arguments:

- `X` – the training input matrix
- `Y` – the training output vector
- `bias` – a weight which you can think of as the weight for an extra input that always takes the value 1 (i.e. it determines the baseline activation of the output when receiving no input).

`trainPerceptron` returns a vector of weights; the final element in this vector is the learned bias weight.

Using the variables `inputs` and `ORoutput` that you defined in Part A and `bias = 0`, train a one-layer feedforward neural net and assign it to the variable `ORweights`. MATLAB will spawn a figure indicating the four points and superimpose a plot of a linear separator as learned by the perceptron.

Recall that in this one-layer network, the output of the network is

$$y = \tanh(w_1 x_1 + w_2 x_2 + b) \tag{5}$$

where x_1 and x_2 are the values of the inputs, w_1 and w_2 are the values of the corresponding weights, b is the bias weight. tanh() is a hyperbolic tangent function which produces positive outputs when its input is greater than 0.5 and negative outputs when its input is less than 0.5.

You can find the values of w_1, w_2, and b from the `ORweights` variable that represents weights in the trained network. w_1 and w_2 correspond to the first two elements and b to the third. Compute the expected output using the above equation for each of the four truth-value pairs (x_1,x_2) in `inputs`, using the weights learned by the network. Save these values in the vector `ORprediction`.

Part C

Can we learn XOR using the same network architecture? Train a new one-layer feedforward neural net using the variables `inputs`, `XORoutput`, and `bias = 0` and assign the result to the variable `XORweights`. MATLAB will again spawn a figure indicating the four points to be learned and superimpose a plot a linear separator based on the current state of the perceptron.

Again, we would like to see what outputs the network produces after training. Compute the expected output using the above equation for each of the four truth-value pairs (x_1,x_2) in `inputs`, using the weights learned by this network (those you just stored in `XORweights`). Save these values in the vector `XORpredition`.

Was the single-layer perceptron able to learn the XOR function? Explain why or why not. How does this compare to the behavior of the perceptron used to learn the OR function? Please write your answer in `problem2/writtenAnswersPartC.txt`.

Part D

Can we learn XOR using a two-layer feedfordward network? (see Figure 1; you can find examples of diagrams like this, and accompanying explanations, in the Rumelhart reading on learning internal representations). We can train a two-layer feedforward network with the function `trainMultilayerPerceptron()`.

`trainMultilayerPerceptron()` takes four arguments:

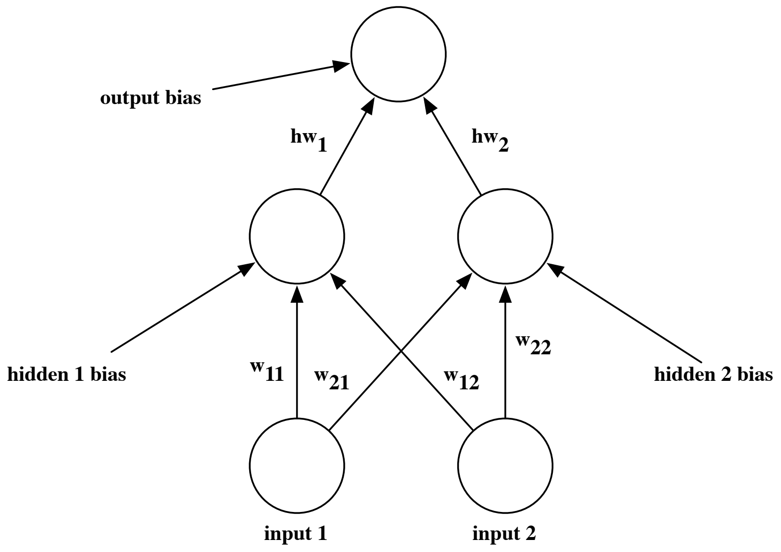

Figure 1: Network structure with one hidden layer.

- `X` – the training input matrix
- `Y` – the training output vector
- `num_hidden` – the number of hidden units
- `num_iter` – the number of iterations to run

`trainMultilayerPerceptron()` returns a cell array of matrices. The first cell array corresponds to connection weights form the inputs to the nodes in the hidden layer; the second cell array from the hidden layer to the output. Within the matrices, the entry corresponds to the bias node. Using the variables `inputs` and `XORoutput` that you just used in Part C, `num_hidden = 2` and num_iter = 20000, train a two-layer feedforward neural net (a network with one hidden layer) and assign the output to the variable `XORweights_mp`. Note that this uses a slightly different implementation and does not include a graphing routine.

Then calculate the predicted output values given the input and the trained model using the function `predictMultilayerPerceptron()`. `predictMultilayerPerceptron()` is provided because the calculation is somewhat more complex than the analogous operation with single-layer feedforward networks that we asked you to write in Parts B and C. `predictMultilayerPerceptron()` takes two arguments:

- `net` – the weights from the trained multi-layer perceptron (the output from `trainMultilayerPerceptron()`)
- `input` – the training input matrix

`predictMultilayerPerceptron()` returns a vector of output values given the trained net and the input. Assign the output values to the variable `XORprediction_mp`.

Was the multi-layer perceptron able to learn the XOR function? Explain why or why not. How does this compare to the behavior of the single-layer perceptron on the same input data? Please write your answer in `problem2/writtenAnswersPartD.txt`.

3 Neural networks and language (7 points)

One of the most successful (and controversial) applications of neural networks has been as models of human language. You will test whether a simple neural network is capable of learning the rule underlying a context-free language.

The language $a^n b^n$, being the set of all strings containing a sequence of a's followed by a sequence of b's of the same length is a simple example of a language that can be generated by a context-free grammar but not a finite-state grammar. Human languages exhibit similar long-range constraints – for example, a plural noun at the start of a sentence affects conjugation of a verb at the end, regardless of what intervenes. Some criticisms of applications of neural networks to human languages are based upon their apparent reliance on local sequential structure, which makes them seem much more similar to finite-state grammars than to context-free grammars. An interesting question to explore is thus whether a recurrent neural network can learn to generalize a simple rule characterizing a long-range dependency, such as the rule underlying $a^n b^n$.

The dataset `abdata.mat` contains two variables: `traindata`, a vector containing the sequence used to train the network, and `testdata`, the sequence used to test the network. In both `traindata` and `testdata` a 1 represents an a and a 0 represents a b.

The training set was constructed by concatenating a randomly ordered set of strings of the form $a^n b^n$, with n ranging from 1 to 11. Frequency of sequences with a given value of n in the training set is inversely proportional to n, being `ceil(50/n)`. `ceil(x)` returns the smallest integer greater or equal to `x`, e.g. `ceil(3)` is 3, but `ceil(3.1)` is 4. The test set contains an ordered sequence of strings of the form $a^n b^n$, with n increasing from 1 to 18 over the length of the string.

Recall that an "Elman" network, as discussed by Elman (1990), is a recurrent network, where the activation of the hidden units on the previous timestep is used as input to the hidden units on the next timestep. This type of network representation is used to allow the network to learn about sequential dependencies in the input data.

We want to see if such a network can learn an $a^n b^n$ grammar. Learning the grammar can be seen as equivalent to correctly predicting what the next item in the sequence should be, given the rules of the grammar. Therefore, the output node represents the networks's prediction for what the next item in the sequence (the next input) will be – it outputs a 1 if it thinks the current input will be followed by an a, and outputs a 0 if it thinks the current input will be followed by a b.

Part A

In order to train your network, you will need both training input and training output. That is, you need a sequence of inputs of the form $a^n b^n$, and a corresponding sequence with the correct output for each item in the input sequence.

For this problem we're going to use `traindata(1:end-1)` as the input training sequence, and `traindata(2:end)` as the output training sequence (where `end` is a Matlab keyword corresponding to the index of the last entry in the vector).

In the space provided in `writtenAnswersPartA.txt` explain why this is an appropriate input sequence and an appropriate output sequence. If you're confused by what the sequences `traindata(1:end-1)` and `traindata(2:end)` look like, try creating them in Matlab, and comparing them to `traindata`.

Part B

In the provided function `anbnLearner.m`, train an "Elman" network with two hidden units using the `trainElman` (remember to use the input and output sequences from Part A). Train the network for 100 iterations, and assign the output to the variable `net`, i.e.

```
net = trainElman(...); % replace ... with your arguments
```

The `trainElman` command takes four arguments:

- `input` – the training input sequence
- `output` – the training output sequence
- `num_hidden` – the number of hidden units
- `num_iters` – the number of training iterations

It will create a network with one input node, the specified number of hidden units, and one output node, and then train it on the training data for the specified number of iterations. The network sees the training data one input at a time (in our case, it sees a single 1 or 0 per time step).

Once the network is trained, you can try it on a new set of sequences, and look at its predictions to see how well it has learned the grammar. Run the trained network on the sequences in `testdata` using the following:

```
predictions = predictElman(net, testdata);
```

Part C

The mean squared error (MSE) of a vector of n predictions $\mathbf{p}$ compared to n target values $\mathbf{y}$ is

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(p_i - y_i)^2 \tag{6}$$

which is just the average of the squared difference between the predicted and target values.

Using the provided template `meanSquaredErrors.m`, complete the function which takes in an array of test data and an array of predictions. The function should return a vector with the MSE for each of the predictions of the network compared to the test set. Remember that the predictions refer to the *next* item in the sequence (e.g. `predictions(:, 1)` should be compared to `testdata(:, 2)`) and the last prediction should be an a.

Note that since we are asking for the MSE of *each* of the predictions of the network compared to the test set that you do not need to divide by n, since n in this case is 1.

Part D

Write code in the provided function `plotMSE` to plot a bar graph of the mean squared error. You can use a command like `bar(mse)`, where `mse` is a vector of mean squared errors. Make sure to also label the x and y axes

Once you have implemented `plotMSE`, run the script `anbnError.m`. This calculates the MSE values for your network predictions using `meanSquaredErrors`, and plots these errors with `plotMSE`.

If you have difficulty interpreting this graph, you may want to examine `testdata`, `predictions`, and your MSE for a few values to see how they are related.

Save this bar graph as as a JPEG named `barGraph.jpg` using the File → Save As menu in the figure.

Part E

Earlier we said that we can evaluate whether the network has learned the grammar by looking at the predictions it makes. If the network has learned the $a^n b^n$ grammar, in what cases should it make correct predictions? When should it make incorrect predictions? How does this correspond to the MSE values you calculated above (i.e. when should the MSE be high, and when should it be low)?

Write your answers to these questions in the appropriate space provided in `writtenAnswersPartE.txt`

Part F

In the space provided in `writtenAnswersPartF.txt`, briefly discuss the implications of this analysis for using neural networks as models of human language.

4 Dimensionality Reduction (3 points)

Dimensionality reduction is a general technique to find low-dimensional representations for high-dimensional stimuli. In this problem, we'll examine the classical approach called *principal component analysis* (PCA).

Part A

Run the script `generateData` to generate three datasets. Each dataset will be saved in a separate `.mat` file and plotted in a separate figure so you can see what they each look like. Note that the colors of the points in the figures have no meaning; they're only colored to help you visualize how the points are distributed in space. Use the "Rotate 3D" tool (it looks like this:) in the Matlab figure window to rotate the figures for datasets B and C to see them from multiple angles. This will help you better visualize the points.

In the space provided in `writtenAnswersPartA.txt`, report how many dimensions each dataset is embedded in (i.e., how many dimensions does the data have *as is*), and how many dimensions each dataset actually varies along (i.e., what is the *minimum* number of dimensions you need to describe the data).

Part B

Run the script `plotPCAvectors`, which calls the function `PCA_proj` to perform the core PCA algorithm, and then plots the datasets again, but with some additional information this time.

In the space provided in `writtenAnswersPartB.txt`, describe the figures produced by `plotPCAvectors`. What do the red and green vectors represent? What is the difference between the red and the green vectors in the figures for datasets B and C? What special property do the red and green vectors have with respect to each other, and why? You may find it useful to look at the code called in the function `PCA_proj.m`.

Part C

Run the script `plotPCAdata`, which plots each of the datasets in their low-dimensional coordinates found by the PCA algorithm. How many dimensions are each of the datasets projected onto? For each dataset, explain whether or not PCA works well, and why.

Write your answers to these questions in the appropriate space provided in `writtenAnswersPartC.txt`.