Name:

Assignment # **10** Due: **25 Nov 97** 6.034 Item # **27**

Please indicate the section you were assigned to, or if you do not remember, put a tick mark on the last line.

Time	Room	Instructor	TA	Tick
10–11	34-302	Randall Davis	Brad Banks	
10–11	34-303	Tomás Lozano-Pérez	Lisa Tucker-Kellogg	
11–12	34-302	Randall Davis	Brad Banks	
11–12	36-156	Tomás Lozano-Pérez	Lisa Tucker-Kellogg	
12–1	34-301	Paul Viola	Constantine Papageorgiou	
12–1	34-302	Robert Berwick	Adrian Danieli	
1–2	24-407	Paul Viola	Constantine Papageorgiou	
1–2	34-303	Robert Berwick	Adrian Danieli	
2–3	34-303	David Cliff	Oded Maron	
3–4	34-303	David Cliff	Oded Maron	
—	—	—		

Refined Guidelines for Homework Solutions

What to Hand In

The Cover Sheet: Please detach the cover sheet from each homework handout, fill it in, and attach it *securely* to your solutions. In addition, please put your name on every page of your problem set.

Code: If you are modifying or extending existing code, you should include in your solutions only the code that you have modified or extended, not the entire program. Please clearly mark the changes or additions that you have made.

Sample Outputs: For most problems, sample outputs of your runs will be required. When the test cases are not fully specified in the homework, choose a few pertinent cases that best illustrate the full behavior of your code. Even if your algorithm is correct, we cannot give you full credit for the problem without outputs demonstrating that your code works correctly across the range of relevant cases.

Commenting Your Code

Comments are a critical part of the code that you will write. For each abstraction that you write or modify (programs, functions, data structures, rules, objects, etc.), we expect you to include an English summary of *what* it does (including input–output behavior, if appropriate); *how* it does what it does; and if there were alternative implementations, *why* you chose the one you chose.

More generally, we expect that you will place comments within the body your code to *bridge the gap* between what you are trying to do and what your code actually does. Comments are especially important to us when we assign partial credit.

Collaboration

We ask for no vow of silence; you can talk with others about the problems and learn from each other. *You can show each other your finished code, and you can ask for help with debugging if you are having difficulty.*

On the other hand, you are not to copy the work of others; everything you hand in must be your own work. *More specifically, you are expected to do your own programming. Jointly prepared programs are not acceptable.*

If you do collaborate with someone, specify the name of that person(s) and the nature of the collaboration.

Massachusetts Institute of Technology

6.034 Artificial Intelligence

Assignment # **10** Due: **25 Nov 97** 6.034 Item # **27**

This problem set was handed out on Tuesday, November 18th and will be due on November 25th.

Goals

- You learn about training neural nets, and how to interpret what the weights and the hidden units "mean".
- You learn about the effects of various modifications to the backpropagation algorithm and to initial conditions for the algorithm, and how to overcome them in part.
- You learn about simulated evolution, using Genetic Algorithms (GAs), including the strengths and limitations of GAs.

Note: The description and problems for the genetic algorithm portions of this problem set follow all the neural network problems.

Code

- The Scheme code for the neural network portion of this problem set is in `/ps10/back.scm`.
- The Scheme code for the genetic algorithm portion of this problem set is in `/ps10/ga.scm`.

Background: Neural networks

You quit school to take a job as a political consultant that is eager to use neural nets to provide advice to political candidates. They expect you to be an expert, noting that you were taking 6.034 when you dropped out.

On the first day, they ask you to demonstrate the power of neural nets by training a neural net on some focus-group data. Basically, they have determined, from the focus-group data, that for a certain population, that when married people belong to different parties, they do not vote, but when they belong to the same party, they do vote, and they want your neural net to reach the same conclusion on the basis of training data.

Accordingly, you start to fool around with a neural net program taken from a 6.034 problem set.

Overview: Neural networks

The file `back.sc` contains the basic neural net learning program. It works by repeatedly changing weights so as to make the network a better predictor of outputs given in a set of input-output samples.

The backpropagation algorithm is used to determine how weights change. That is, the program computes partial derivatives, multiplies them by a rate constant, and adds the product to the existing weights.

The rate constant is bound to the global variable `*rate*`. The following is a representative example of a rate constant suited to the particular problem you will work on:

```
(define *rate* 2.0)
```

The input-output samples are bound to `*samples*`. The following is a representative example of a sample set for a two-input, one-output net that you decide to work with. The first element represents sample inputs; the second represents sample outputs. The first element of each sample input represents one partner's party (1 means republican, 0 means democrat); the second represents the other partner's party (again, 1 means republican, 0 means democrat). The first and only element of each sample output represents whether the partners vote (1 means yes, 0 means no).

```
                        *-- Inputs (two)
                        |     *-- Output (just one)
                        |     |
                        v     v
(define *samples*  '(((1 0) (0.0))
                      ((0 1) (0.0))
                      ((0 0) (1.0))
                      ((1 1) (1.0))))
```

The structure of the net, and its current state, is determined by an expression bound to `*weights*`. The following is a representative example, showing how structure is encoded for a two layer net. The first element in the list describes the weights between the two inputs and the two neurons in the first layer of neurons. The second element describes the weights between the neurons in the first layer and the second layer.

Each list of weights contains one particular weight, the first, that is taken to be the weight connected to a permanent `-1` value. Accordingly, that weight embodies a neuron's threshold value. The values shown below are those values in the code filed, rounded-off.

```
(((0.26 -0.70  0.25)     <-- between inputs and neuron 1,  layer 1
  (0.40  0.91  0.36))    <-- between inputs and neuron 2,  layer 1
 ((0.13 -0.54 -0.58))))  <-- between layer 1 and neuron 1, layer 2
```

Normally, you would initialize a neural net by using a random number generator buried in the function `initialize-weights`. So as to assist in debugging, and grading, you are usually asked to work from a given set of initial weights, bound to `*initial-weights*`.

```
(define *weights* *initial-weights*)
```

Training is initiated by calling the `train` function. The argument specifies the maximum number of training epochs. For example, the following adjusts the weights bound to `*weights*` for 600 epochs.

```
(train 600)
```

In each epoch, the weights are adjusted once for each training sample. Every 25 epochs, `train` prints information about how the training is going in terms of rms error.

The `train` function stops when that maximum is reached or when the root-mean-square error of the net, when working with the training samples, goes below 0.1.

Problems on neural networks and backpropagation

Problem 1 Network picture

Using the space below, sketch the net pictured by the weights bound to `*initial-weights*`:

Problem 2

Run `train` on the training data. Set initial weights, `*weights*`, bound to `*initial-weights*`. Use a rate factor of 2.0. Sketch the progress of training on the graph below. You can draw the plots by computer or do them by hand. (Save the values you get for `*weights*` for future use in parts A and B of Problem 4. In order to avoid spurious Scheme interrupts, you will find it more convenient to break the 600 epochs into units of 100 or 50 each: that is, simply run `(train 100)`, and do so 6 times total; each time you will start the next epoch with the `*weights*` retained from the previous epoch. Be patient! However, if you find this Scheme on Athena totally brain-dead, consult the TAs for how to run Scheme-to-C.)

```
 *-- Average rms error
 |
 v
0.5 *---------*---------*---------*---------*---------*---------*
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    *---------*---------*---------*---------*---------*---------*
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
0.0 *---------*---------*---------*---------*---------*---------*
    0                  200                 400                 600 <-- Epoch
```

Problem 3 What are hidden units?

Part A

By modifying `*initial-weights*`, increase all the initial weights on the first input hidden unit to 100. The code for `*initial-weights*` should now be as follows:

```
(define *initial-weights*
      '((((5000.0 5000.0 5000.0)
         (0.3975036 0.906125069 0.360087872))
        ((0.129490018 -0.5424665 -0.582934))))
```

Important: keep all other parameter values the same. Now re-run `train` and sketch the progress of training on the graph below. Describe in one sentence or two the behavior you observe.

```
 *-- Average rms error
 |
 v
0.5 *---------*---------*---------*---------*---------*---------*
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    *---------*---------*---------*---------*---------*---------*
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
0.0 *---------*---------*---------*---------*---------*---------*
    0                  200                 400                 600 <-- Epoch
```

In a few sentences, please describe the behavior:

Part B

In terms of the backpropagation gradient algorithm, explain in simple mathematical terms *why* you see this behavior. (Note: the behavior is not a problem due to "round-off", and this is not the explanation of the phenomenon in question.)

Part C

Suppose the "fan in" to a middle unit is, say, 4, or, in general, k. If the inputs to the unit are all normalized, propose a simple mathematical formula in terms of k for an initial weighting that avoids the behavior you observed in the part just above (your formula should take just one line at most):

Problem 4 What do the hidden units do?

If we think of each unit in the network we have designed as computing a Boolean logical combination, there are in fact two solutions that can generate the output z from the two inputs x and y. Here, $\wedge$ means "Boolean logical and"; $\vee$ means "Boolean logical or"'; and $\neg$ means "Boolean complement" (converts 1 to 0 and vice-versa):

Type 1: $(x \vee y) \wedge \neg(x \vee y)$

Type 2: $(x \wedge \neg y) \vee (y \wedge \neg x)$

Part A

Classify the solution you get from training the net in Problem 2, part A as TYPE 1 or TYPE 2, and circle your answer below:

- TYPE 1
- TYPE 2

Part B

Try running the network with two very different sets of initial weights than the two given above (the initial set and the altered set), and classify your resulting network as Type 1 or Type 2. If you get only weights that yield Type 1 networks (or vice-versa), then explain why, and suggest what possible weights would yield a Type 2 network (or vice-versa).

Part C Extra credit: Only if you use Scheme-to-C

Now run the network 100 times with random initial weights, while holding all other parameters — the thresholds, momentum, learning rate, etc. constant. You will have to write a small piece of scheme code to do this, making use of `another-random-number`. Make sure whatever method you use to generate random starting intial weights covers a wide enough range — you might consider drawing from the range ± 20 for each weight, for instance. (For this problem, you *will* have to use Scheme-to-C, in order to avoid spurious Scheme interrupts. Please see the TAs for how to do this. When you are done (this may take a long time!), and you can tabulate your results, please answer the following question:

What is the relative frequency of Type 1 vs. Type 2 solutions?

Please display your answer below as a fraction Type 1/Type 2 and graphically, as a histogram showing the proportion of Type 1 to Type 2 answers, scaled in percent.

Relative frequency Type 1/Type 2, and histogram:

Part D

What does this suggest about the regions of the weight space for each of the "hidden units" that implement each type of solution? Your answer should take no more than a few sentences.

Problem 5 Faster, Better, Learning

Professor Hacker drops by and comments that the "learning rate" parameter, described by the variable `*rate*`, dictates "how fast the network will learn — the higher the learning rate, the better." Inspired by this comment, you re-set the learning rate from its current value, 2.0, to a much higher value, 2000.00 — "that will increase the learning rate 1000 times," the Professor cries. You leave all the other parameters at the values they had initially in Problem 2, Part A (i.e., their values as set initially by `*initial-weights*` in the original code).

Part A

What is the result of your experiment? Does increasing the learning rate assist learning or hinder learning? Describe your results in terms of the behavior of the network learning progress, recorded in the graph below.

```
 *-- Average rms error
 |
 v
0.5 *---------*---------*---------*---------*---------*---------*
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    *---------*---------*---------*---------*---------*---------*
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
    |         |         |         |         |         |         |
0.0 *---------*---------*---------*---------*---------*---------*
    0                  200                 400                 600 <-- Epoch
```

Part B

Briefly explain *why* the network with the changed learning rate behaves as it does. Hint: look at the "error deltas" — the difference between what the network is predicting as the output vs. the actual, true output desired, as one proceeds from epoch to epoch.

Part C

Propose a solution to this problem by means of adjusting an initially high learning rate `*rate*` in the face of such behavior. Describe your proposed solution in mathematical and/or geometric terms.

Problem 6 Nets and Neighbors

Much to your dismay, Professor Hacker returns. Pondering your network picture from Problem 1, the Professor insists that this is a particularly *bad* benchmark problem on which to investigate the ability of neural networks to *generalize* beyond the training data they are given. What is it about the nature of the problem this particular network is trying to solve that justifies the Professor's remarks? A few sentences, or at most one (short) paragraph will suffice to answer this question.

Problems on Genetic Algorithms

There is a common saying that, given enough time, a monkey bashing on a typewriter could produce all the works of Shakespeare.[1] Presumably, given enough time, the same monkey could a produce all the 6.034 problem-sets you get on this course.

The amount of time required will, of course, be rather longer than you'll spend here at MIT. So let's make the task a bit easier: we want a monkey to type randomly until it generates the string 'sixohthirtyfour'. Real monkeys are difficult to get hold of. Instead, you decide to use a Scheme program to simulate the monkey's random typing.

The target string is 15 characters long, so your program should generate strings of the same length $l = 15$ using the 26 lower-case characters 'a' to 'z'. Use l to represent the length of the strings and c to represent the number of characters in the alphabet. Recall that the probability of an event X happening can be calculated from 1.0 minus the probabilty that X wont't happen. Starting with one $l = 15$ string where each letter is chosen randomly (with equal probability) from the $c = 26$ alphabet:

Part A

How many possible such strings are there?

Part B

What is the probability that at least one of the 15 characters correctly matches the corresponding character in the target string?

[1] These problems are based on those given in Chapter 1 of John Maynard Smith's *Evolutionary Genetics* (1989) which are in turn based on an example in Chapter 3 of Richard Dawkins' *The Blind Watchmaker: Why the evidence of evolution reveals a universe without design* (1986). Despite coming from the biology literature, they are still very relevant for studying artificial evolution and genetic algorithms.

Part C

What is the probability that exactly one of the 15 characters is correct? *(Remember: each character is generated equi-probably and indepedently).*

The original string is copied ten times, but errors creep in. For each letter copied, there is a 99% chance that everything goes OK, but 1% of the copy operations "mutate" the letter, changing it to any one of the other characters in the alphabet, with equal probability.

Part D

If none of the characters in the original string are correct, what is the probability that at least one character does match in at least one of the ten copies?

The 'fitness' of each copy is calculated by counting the number of characters that correctly match the corresponding character in the target string. The copy with the highest fitness value is chosen as the *elite*. If there are two or more copies with the same highest fitness value, one of them is chosen at random to be the elite. The elite copy is then used as the 'parent' of the next 'generation': ten copies are made of the elite, with mutations, in the same way as before.

Part E

Approximately how many generations will pass before a string with a nonzero fitness value is created, if the original string had a fitness of zero? *(Hint: in these questions, you can estimate the number of generations before an event X occurs by calculating the reciprocal of the probability of X occuring on any one generation).*

Part F

If you wait long enough, a string with a fitness value of 14 will be created. What is the probability that at least one of the ten 'children' of such a string is correct at all 15 sites?

Part G

Roughly how many generations will it take to progress to a copy with all 15 characters correct, starting from a string with a fitness of 14.

Part H

Give a symbolic expression that can be used to estimate the number of generations it will take to evolve from a string of fitness f to at least one 'child' having fitness $f+1$, where the the child differs from the parent by only a single mutated character.

Part I

Use the expression you just derived to estimate how many generations it will take to complete the whole process, from zero fitness to 15.

Part J

If the mutation rate is increased from 1% to 10%, would it take less time? Why?

The file `ga.scm` has Scheme code for a simple genetic algorithm (GA) that evolves character strings, with fitness being determined by the number of characters that match with a target string. Read the comments at the start of the file, and run the code a few times to get a feel for how it works.

Part K

The function that generates random characters is currently specified to generate all characters equiprobably. Yet some letters occur more frequently than others: in English, the twelve most frequent characters are e, t, a, o, i, n, s, h, r, d, l, and u. Alter the code so that these characters are twice as likely as any others. Write your results (number of successes, followed by the mean and standard deviation of the number of reproductions per success) here:

Part L

Alter the code again so that the random-character function now only produces the characters in the target string, with each character having a probability determined by its frequency in the target string 'sixohthirtyfour': e.g., 'i' occurs twice, so its frequency is 2/15. Write your results (number of successes, followed by the mean and standard deviation of the number of reproductions per success) here:

Part M

The current code shows only *asexual* reproduction, where copies of the best individual are made with mutation. Alter the code so that each `kid` is generated from `mom`, the best individual in the population, and `dad`, the second-best individual in the population. Run the code and write your results (number of successes, followed by the mean and standard deviation of the number of reproductions per success) here: