MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034—Artificial Intelligence
Recitation 9, Friday, November 9, 2001

Agenda

1. Genetic Algorithms: Reproduce or Die, Crazy Frank
2. How to Perceptron, re Harry Perceptron (or, Harry Potted and the Goblet of Vodka)

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 in this course. (Surely there is additional, independent evidence for this fact.)

The amount of time required will, of course, be rather longer than the time you'll spend here at MIT. So we make the task a bit easier: we want a monkey to type randomly until it generates the string *sixothreefour*. However, real monkeys are difficult to get hold of. We will do the next best thing: decide to use a Scheme program to simulate the monkey's random typing.

We will use this example to illustrate how a "genetic algorithm" attempts to simulate (part of) the process of evolution by natural selection (or, as Darwin actually called it, "descent with modification"). There are three (evidently necessary and sufficient) components to this approach (which Darwin actually recorded in his Notebooks in 1827, nearly 30 years before publishing "Origin of Species"):

- Heredity (or information transmission): children must resemble their parents more closely on average than other, unrelated children;
- Variation (aka "diversity"): not all individuals are identical
- Differential representation in the next generation: some parents leave more offspring than others (one ordinarily says that individuals who leave more offspring — whose genes are differentially represented more highly in the next generation — are *fitter* than those who are less highly represented. (So if an organism has no offspring, its fitness is zero.)

Variation is the jet fuel that evolution by natural selection "burns" (no variation, then nothing can change; a classic theorem by R. Fisher, 1930, is that the rate of evolution is directly proportional to the amount of variation in a population).

To model this process by computer, we simply simulate each of these components:

[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.

1. The representation of populations in terms of individual strings
2. Copying these strings as a way of simulating reproduction, and heritable transmission of information (heredity)
3. The notion of *mutation* by introducing random error into the copying process
4. The notion of *fitness* as a way to test that some strings are "closer" to the target string (in this case "intelligent") than others, or the notion of differential reproduction

In the Winston book, and in classical genetic algorithms, there is an added twist that adds in more variation: one also models the notion of *two* strands of bits (DNA), corresponding to the pairs of chromosomes in diploid (eukaryotic) cells. This permits the mechanism of *crossover*, where sections of each bit string may be exchanged, shuffled from one of the pairs of chromosomes to another. (In actual practice, though this simulates the process of sexual recombination, and though this mechanism has been preserved through about a billion years, this may not help a genetic algorithm very much, and it's not clear at all why this variation has been required in real life, because the immediate consequence of sexual reproduction is to *cut* the proportion of one's own genes in the next generation by one-half. Just cloning yourself would work far better – it would have twice the genetic representation, or twice the fitness, from the get go.)

Intuitively, a genetic algorithm (GA) does a kind of beam search via local gradient ascent: each of the individuals is trying to climb to the top of a fitness peak, all starting out in different locations, or different string patterns (the length of the string is the number of dimensions). Thus, all the problems with local gradient ascent apply!

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 probability that X won't happen. We start with one $l = 15$ string where each letter is chosen randomly (with equal probability) from the $c = 26$ alphabet.

One can now show that is very unlikely that one could arrive at exactly the target string by chance alone, without some measure of "fitness" that accumulates good changes. Then, we will carry out, step by step, the calculations that are done to "simulate" evolution. This method has a number of components that we cover in turn.

Code

The code `ga-runs-faster.scm` is a simple genetic algorithm (GA) that evolves character strings, with fitness being determined by the number of characters that match with a target string. We give an example of output from the algorithm immediately below.

The basic procedure for the genetic algorithm is the function `ga`, with arguments as follows.

```
(ga popsize repros target mrate)
```

Where `popsize` is the population size (number of offspring); `1000` is the number of reproductions (that is, the number of "generations"), `target` is the target string; and `mrate` is the mutation rate. We always start from a random initial string of the given length. For instance, for our problem set we might call `ga` as follows:

```
(ga 10 1000 "intelligent" 0.01)
```

This says to have each generation produce 10 offspring; compute 1000 generations; start at a random string, and introduce copying errors with a rate of 1% per letter per generation (that is, out of 100 copying operations, 1 will go wrong, introducing another randomly drawn letter). (This is MUCH GREATER than the actual mutuation rate in organisms, which is about 1 every million "digits" or so — there are a variety of error-correcting devices that organisms have for copying, itself a fascinating topic.)

You shouldn't call `ga` directly, however. Because genetic algorithms are inherently probabilistic, we generally carry out multiple runs of the same system; this is done via the function `ga-multi`, which calls `ga` repeatedly, as follows:

```
(define (ga-multi nruns popsize repros target)

Example:

(ga-multi 10 10 1000 "intelligent")
```

This runs `ga` over and over, 1000 generations each time, 10 times total, with 10 random draws for the starting point.

Note that `ga-multi` **fixes** the mutation rate as `1/(length target)` (in our examples, the target length is 11, so the mutation rate is approximately 0.09). You can easily change this in the code. `Ga-multi` is useful because it will automatically compute the mean and standard deviation of the results of a number of complete runs.

Here is some sample output from the algorithm. We call the algorithm for 1000 generations, a mutation rate of about 0.09, 6 times over. The program generates **a lot** of output.

```
(load "ga-runs-faster.scm")
;Loading "ga-runs-faster.scm" -- done
;Value: ga-multi

(ga-multi 10 6 1000 "intelligent")

;; Note that we START with the random string efeeekscaan, which
;; happens to have 1 letter in the right spot matched against intelligent
;; (the fourth letter e), so the FITNESS is 1

Experiment 0
Repro=0 Current best: (efeeekscaan . 1)
Repro=1 Current best: (efeeekscaan . 1)
Repro=2 Current best: (efeeekscaan . 1)
...
```

```
Repro=17 Current best: (efeeekscaan . 1)
Repro=18 Current best: (efeeeksoean . 2)
...
Repro=62 Current best: (ebzelksoean . 3)
Repro=63 Current best: (ebzelksoean . 3)
...
..
;; After 1000 generations, the system has got all but 1 letter
;; correct, so the fitness is 10.
Repro=998 Current best: (intellxgent . 10)
Repro=999 Current best: (intellxgent . 10)

Finished after 1000 reproductions

;; Here is one run in this series that came out perfectly
Experiment 2
Repro=0 Current best: (uminyuindiy . 1)
...
...
Repro=843 Current best: (intellicent . 10)
Repro=844 Current best: (intellicent . 10)
Repro=845 Current best: (intelligent . 11)

Finished after 846 reproductions
```

Why do some runs not converge?

Basic string probability calculations

Starting with one $l = 15$ string where each letter is chosen randomly (with equal probability) from the $c = 26$ alphabet:

How many possible such strings are there?

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

Answer: The probability of an incorrect character at any one position in the string is $\frac{25}{26}$. Let C be the number of correct characters in a string.
Then $P(C \geq 1) = 1 - P(C = 0) = 1 - (25/26)^{15} \simeq 0.4447$

What is the probability that exactly one of the 15 characters is correct? *(Remember: each character is generated independently and with equal likelihood).*

Answer: This requires one character correct and $l - 1 = 14$ characters wrong, thus:
$P(C = 1) = (\frac{25}{26})^{14} \cdot \frac{1}{26} \simeq 0.0222$

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?

Answer: Let N be the number of copies of a totally incorrect string that have at least one correct character. Then $P(N \geq 1) = 1 - P(N = 0)$, and for 10 copies $P(N = 0) = P(I = 0)^{10}$ where I is the number of correct characters in one copy of the original string. Now at each of the l incorrect characters in the original there is a $\frac{99}{100}$ chance of it being copied without a mutation, and a $\frac{1}{100}$ probability of mutation. If the mutation occurs then there is a $\frac{24}{25}$ probability of the original incorrect character mutating to one of the 24 remaining incorrect characters. So, for each location on the string there is a probability $i = \frac{99}{100} + \frac{1}{100} \cdot \frac{24}{25} = 0.9996$ that the copy's character at that location will still be incorrect. Thus $P(I = 0) = i^l = 0.9996^{15} \simeq 0.9940$ and hence $P(N \geq 1) = 1 - P(I = 0)^{10} \simeq 0.0582$.

Introducing mutation in copying

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.

Fitness

We can now measure the "fitness" of each copy by calculating 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.

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 occurring on any one generation. You will also need the result from the previous question.)*

Answer: From the hint, and the answer to the previous question, if $G_{F>0}$ is the number of generations it takes to generate an individual string with non-zero fitness, starting from an individual with zero fitness, then $G_{F>0} = 1/P(N \geq 1) \simeq 17.168$; but the number of generations must be an integer, so 17 or 18.

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.

Answer: If f is the parent's fitness, then $G_{F=f\rightarrow f+1} = 1/P(N_{f+1} \geq 1)$ where $P(N_{f+1} \geq 1)$ is the probability that one or more children have fitness $f+1$.
Using $P(N_{f+1} \geq 1) = 1 - P(N_{f+1} = 0)$, and $P(N_{f+1} = 0) = (1 - c_{f+1})^{10}$. Here c_{f+1} is the probability of any one child having a fitness of $f+1$ as a result of a single-character mutation when copying the parent. This requires that:

- The f correct characters are not mutated: the chance of this happening are $(\frac{99}{100})^f$.
- One of the incorrect characters mutates to be correct: the chance of this is $\frac{1}{100} \cdot \frac{1}{25} = \frac{1}{2500}$.
- The remaining $l - f - 1$ incorrect characters are not mutated to correct characters. The chance of this is $(1 - \frac{1}{2500})^{l-f-1}$.

Thus $c_{f+1} = (\frac{99}{100})^f \cdot \frac{1}{2500} \cdot (\frac{2499}{2500})^{l-f-1}$. And so:

$$G_{F=f\rightarrow f+1} = \frac{1}{1 - (1 - \frac{1}{2500}(\frac{99}{100})^f(\frac{2499}{2500})^{l-f-1})^{10}} \tag{1}$$

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

Answer: The expression above gives estimates shown in the table below, so the total is around 4000 generations.

f	$G_{F=f\rightarrow:f+1}$	Cumulative total
0	251.85	251.85
1	254.29	506.15
2	256.75	762.90
...	...	...
13	285.46	3756.65
14	288.22	4044.87

Mutation rate

Change the mutation rate, `mrate`, in `ga-multi`, from 0.09 to 0.01. Do you expect it take more or less time to find an optimally fit string? Why or why not?

The limits of genetic algorithms

In what **single major** way does the evolutionary model described here *differ* from biological evolution (Hint: one does not need to be a biologist to know the answer.)

Acessibility

Consider a (2-D) square matrix 26 x 26, X and Y axes labeled with the letters of the alphabet. These represent all 2-letter sequences in English.

Assume this kind of evolutionary system starts with an *arbitrary* two-letter English word in the form consonant-vowel (C-V), or vowel-consonant (V-C), such as *do*, *by*, *at*, *in* (but **not** *yb* or *oo*; in fact there are no legitimate V-V or C-C words in English — take our word for it. The English vowels comprise *a, e, i, o, u, y*; consonants are all the other letters.).

Note that this corresponds to a kind of "viability" constraint on the evolutionary process (the organisms in between generations must all "live").

Question: Given this constraint, is the evolution of "do" into "be" possible (given enough time)? Similarly, given this viability constraint, is it possible for *any* legitimate two-letter English word to evolve into *any other* two-letter English word? Why or why not?

As it turns out, we will get two separate clusters: one for vowel-consonants (like "AT") and another cluster for "DO", etc. We *can't* get IN→AT because there are no meaningful vowel-vowel sequences in English.

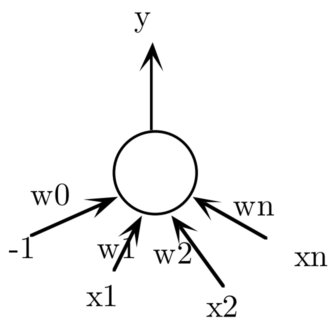
Note that for all THREE letters sequences, the space is fully connected. (This is easy to show).

For all FOUR letter sequences, one can show (it's hard!) that the space is not (in fact it is not for n¿ 3, in English). So, for "GENE"–¿"WORD" e.g., this isn't always doable, for all possible English 4-letter word pairs....

Perceptrons

A perceptron is a single-layer neural net. The output $y = 1$ if the weighted sum on the inputs is greater than 0; it is 0 otherwise. (The idea is to simulate a single neuron.)

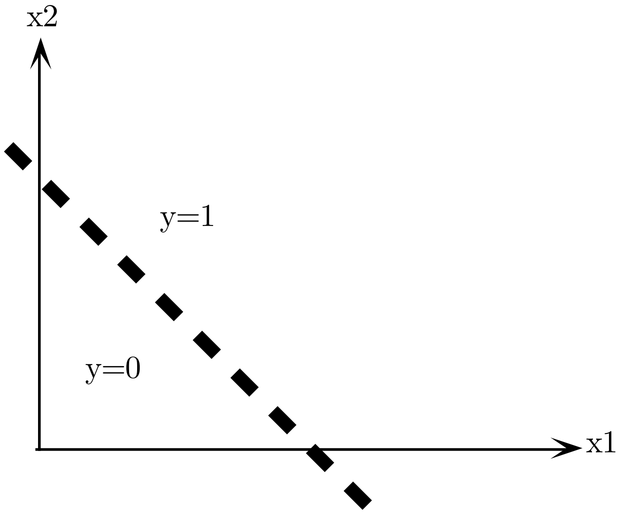
The picture looks like this:



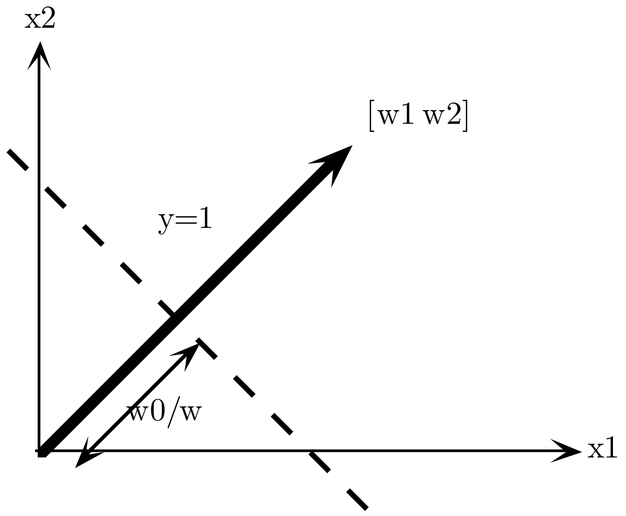
We can understand this mathematically as the equation of a line in a plane (hyperplane). For two variables,

$$w_1x_1 + w_2x_2 = w_0 = 0$$

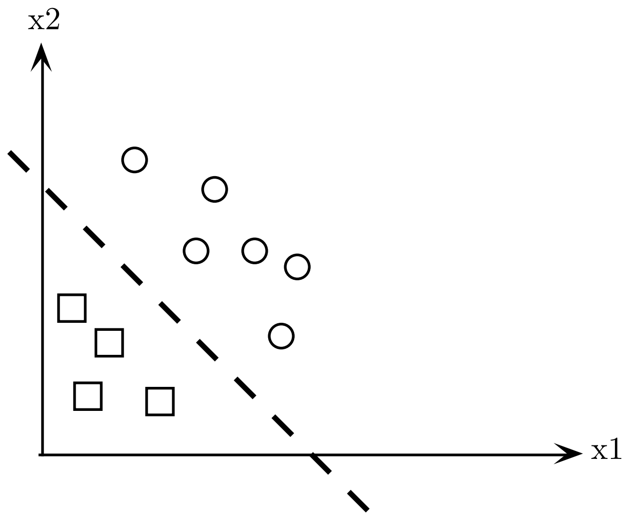
The equation in the plane looks like this:

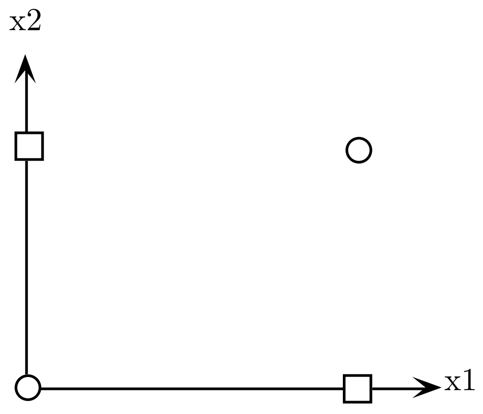


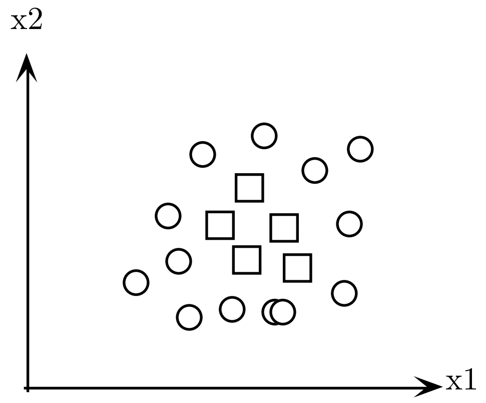
where x_i are the input variables and w_i are the weights. Alternatively, as a matrix equation, we have that the dot product of $[x_1s_2]$ and $[w_1w_2]$ is equal to the constant 0. This is true of the points on the line perpendicular to the vector $[w_1w_2]$. If we let w be the length of the vector $[w1w2]$, i.e., $\sqrt{w_1^2 + w_2^2}$, then we can write this as:



We can use a single layer net (perceptron) to make decision boundaries that are *linearly separable*:







Each perceptron unit can define ONE line; the inputs are directly coupled to the outputs.

If there are MORE than two variables, then the perceptron defines a PLANE (or HYPERPLANE) in 3, 4, ... dimensions. So that's all we can do with it. We CANNOT intersect lines or hyperplanes.

Perceptron Learning rule

Because outputs are *directly* tied to inputs, the perceptron learning rule is easy: change the weights of inputs that are non-zero in the direction of the error, where the *error* is the difference between the true output and the output the perceptron with the current weight gives. (If the error is negative, i.e., the weight is too large for that example, then lower the weight; if the error is positive, then raise the weight. Ignore weights that are zero — they are contributing nothing.)

The learning equation to change the weights is:

$$\Delta w = \alpha(y^* - y)x$$

where y^* is the *correct* output, y is the current output with the current weights, and α is the learning rate (amount of allowed change in weights, usually small so we don't overshoot). Note that the difference between the correct and computed values can only be $-1, 0, +1$ in this case.

This method is guaranteed to converge for linearly separable problems, and is chaotic otherwise.

Representing line equations: we use $ax + by - c = 0$ since that lets us represent vertical lines.

To separate regions that are not split by a single line, we must "project" into a higher-dimensional space by using "hidden" units (called hidden because they are between the input and the output). (Effectively, we add an extra hidden layer of units to draw each separate line. — Let's go through the recitation assignment on this one.)

For next time: the backpropagation learning algorithm. Intuition: to find out how to twiddle the weights on the hidden units, we imagine the network like a spider web: if we tap an output unit, which strings vibrate the most, going backwards? (Depends on the weights and topology — gradient ascent.)