Homework 1: Simple Systems, Complex Behaviors

At least one theme in this course will be how seemingly simple cognitive models can sometimes give rise to complex behaviors. However, we often need computer simulations to help understand how these simple rules will play out. For example, imagine a simple experiment where you roll two die and keep going until a particular pair comes up (say 7,11). What is the distribution of the number of rolls needed to reach this outcome? While we can certainly figure this out analytically, we might also make a simple program that simulates the process (i.e., monte carlo method), and observe the results by having the computer run the game hundreds of thousands of times. In this case, a simulation of the physical process of flipping coins can help us understand the behavior of the system. Once we start trying to understand human cognitive processes or the structure of social behavior, we basically have to use simulations as there are unlikely to be simple, closed-form mathematical expressions that capture the phenomena. In this first homework, we will 'play' with three distinct systems based on simple rules, which are not only difficult to analyze analytically, but give rise to significantly more complex behavior at other levels of analysis than we might expect. The first is the Schelling neighborhood model, the second is Conway's Game of Life, and the third is a neural network model of human memory based on Hebbian Learning. The focus of the homework is simply to interacting with each system to gain some intuition for how they behave, and a sense for how intuition can sometimes fail in understanding even simple models.

If you are an experienced modeler, this is likely to be a somewhat elementary exercise, and at the very worst a little fun. However, if you've never tried thinking through the implications of a complex set of mutual constraining factors it might be more interesting/ informative. The point for this first homework is mostly to get people thinking about the (at least some) kinds of models we will be thinking about at a very broad level (but better than just reading a paper about them).

Part I: Schelling Neighborhood Model

The schelling neighborhood model was demonstrated in class. In the first part of the homework, you will interact with the Schelling model yourself to gain some insight into the system. Download 'hw1.zip' file, and find the schelling.py script and run it. If you have trouble let me know (you'll need python, but it comes installed on mac, and is a simple download for pc). For each of the following steps, you should try to make a guess what will happen using what you know about the rules of the system before you try it, and see if your intuitions are correct.

Step 1:
Starting with the 'pred' and 'pblue' sliders set to their initial value, what is the highest setting of the threshold that leads to segregated neighborhoods where each agent is happy enough that they won't move (i.e., the screen doesn't show any movement after a long time?). You may have to try a couple different randomized initializations of the grid. Does the value of the tolerance (1 = intolerant, 0 = maximally tolerant) surprise you at all?

Step 2:
Adjust the sliders of 'pred' and 'pblue' so there is either less or more blank space in the world. What effect does this have on the threshold parameter that leads to stable segregation?

Step 3:
What happens when one population is bigger than the other (i.e., pred>pblue)? What effect does this have on the threshold that reliably leads to stable segregation?

Step 4:
You can adjust the threshold dynamically while the simulations are running. If you move it to 1.0 then everyone is always unhappy and will move quite a bit, while if you move it lower people will start settling down. Adjust the slider so that you get the biggest neighborhood possible (it may require going low then high again with threshold). The process of raising the "randomness" in the system then slowly lowering it is akin to "annealing" and is a useful solution to some complex optimization problems.

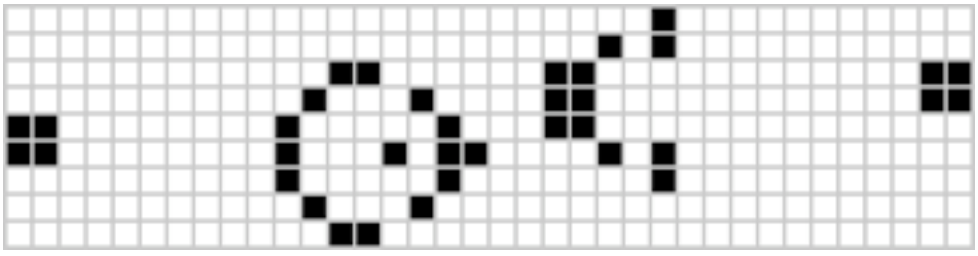
Part II: Conway's Game of Life

A system that is related to Schelling's neighborhood model is Conway's Game of Life. A computer scientist (John Conway) developed Conway's Game of Life in the 1970s while investigating simple cellular automata (see Wolfram reading for more info). The rules of the standard version of Conway's Game of Life are:

1. Any cell with fewer than two "on" neighbors turns "off" (from loneliness)
2. Any cell with more than three "on" neighbors turns "off" (from crowding)
3. Any "on" cell with two or three live neighbors survives
4. Any "off" cell with 3 "on" neighbors turns "on" (comes back to life).

Step 1:
The conway.py script is a version of the conway game. You can enter initial patterns into the grid system and then let the simulation run to completion.

The simulation has built-in commands for some basic patterns (blinkers, gliders, light-weight space ships). However, try entering the following one and watching the implication (be careful to count the spaces, etc...)



Step 2 (more advanced):
Inside the python script there is a function called 'updateagent()' (around line 112) that is the logic for each cell. The code should be pretty straight-forward. The first part gets the current state of the grid location, the second part gets the value of the current cell's neighbors, and the final part sets up the rules for when to turn on, and when to turn off. The standard Conway's Game of Life is known by short hand as 23/3 since cells stay ON if either 2 or 3 neighbors are on (otherwise turn OFF), or turn back ON if exactly 3 neighbors are ON. Thus, anther rule might be 23/36 which means "stay ON of 2 or 3 neighbors are on (otherwise turn off), or turn back ON if 3 or 6 neighbors are ON". Try changing the rules of your life script to the following set, and observe the result (you may have to play with a couple different grid arrangements to "feel" out the behavior of the system.

Rules:

/234

1/1 - also known as "Gnarl"

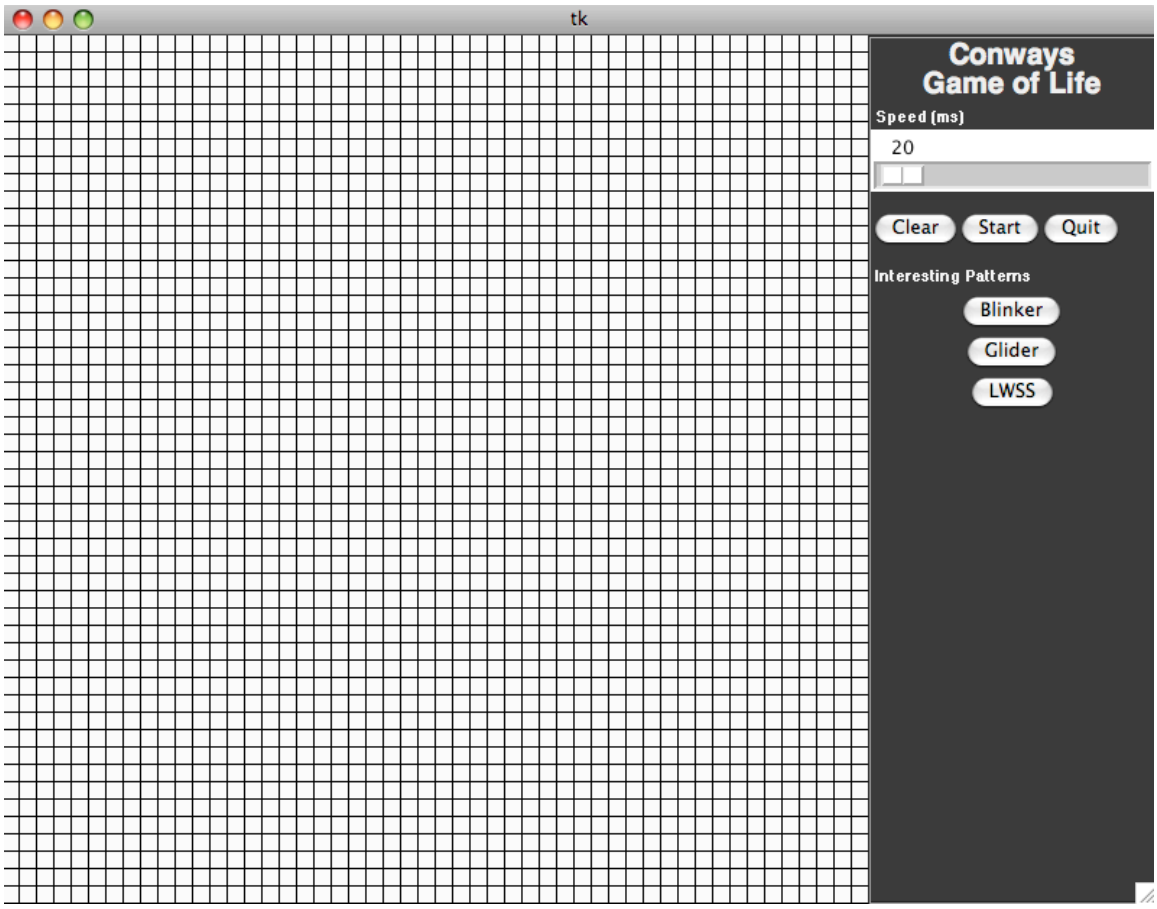
12345/3 - Maze

1358/357 - Ameoba

23/36 - High Life

As you can see, after playing with these different rule systems, not all sets of simple rules give rise to the same complex sets of behavior.

Part III on the next page (skip step 3 for Part III, and the stuff about "Bonus point" in the game of life)



Bonus Points

Add a button that sets up the "glider gun" shown on the Wikipedia entry for Conway's Game of Life: http://en.wikipedia.org/wiki/Conway's_Game_of_Life

---

Part III: Hebbian Learning

The above systems are based on simple rules that give rise to interesting spatial patterns. However, neurons in our brains appear to have similar properties. For example, scientists have developed a strong understanding of the way the neurons communicate through "firing". Learning appears to change the propensity of one neuron to cause another to fire. In this final part of the lab we will explore a "memory" system that operates on this principal.

Donald Hebb first noticed a form of neuronal learning in 1949. In his experiments, he found that when two neurons repeatedly fire together the association between them seems to strengthen such that activating one of the pair would often cause the

other to fire. In effect the paired firing of two neurons appears to "link" them together. In his words, "When neuron A repeatedly participates in firing neuron B, the strength of the action of A onto B increases".

In this lab we will investigate this principal using a neural network called an auto-associator network. The basic idea is that there are a set of input neurons and a set of output neurons. Association weights connect each input unit to each output unit. During learning, changes are made to the strength of the weights. The goal of the network is to re-create its input on the output. While this might seem a pointless task, we will show how this type of system can create a relatively powerful memory.

We assume that the input units are a single list of values. Neurons can either be ON (1) or OFF (-1). Output units are activated by the input units according to the following equation:

$$y_j = \sum_i w_{ij} \cdot x_i$$

which simply says the output of neuron 'j' is equal to the sum of all the input units (x_i) times the value of the weight connecting input using x_i to output unit y_j.

Learning takes the form of the hebbian association rule:

$$\Delta W_{ij}(t) = \eta \cdot y_j \cdot x_i$$

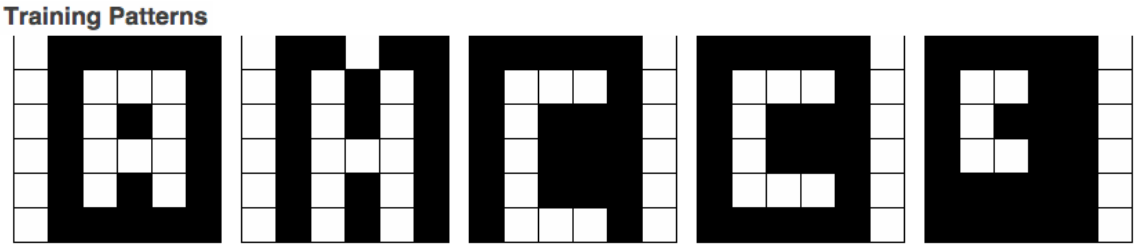
Which says the change to the weight connecting input unit I and output unit j is equal to the value of the input pattern unit I x_i and output unit y_i times a learning rate parameter (eta).

Step 1

What happens when you make two of the input patterns very similar to one another? What happens when you make all of the patterns in a different section of the screen (so there is little overlap)?

Step 2

Set PATSIZE=6 (so each input/output pattern is a 6x6 grid). Then input the following training patterns.



Note the column of "on" pixels along either the left or right edge. You can think of this as the "category label". Here we have two categories, A and C. Train your network for ~40 blocks. Now perform the following tests:

1. Input the large c (middle pattern) but turn OFF the column along the right (i.e., remove the category label). Test the pattern with the network? What happens?
2. Input the smaller "A" (first pattern) also without its "category label" or name. What happens here?
3. Now input only the category label (i.e. vertical column on left or right) with no pattern. What does the network do? Compare this to the prototype model from Lab 1.
4. Now create 5 new input patterns. Put two patterns in category 'A' (associated with the left column all being set to ON) and three in category 'B' (associated with the right column all begin set to ON). However, make it so that each pattern doesn't overlap with each other either within or between categories (patterns can be simple lines or something). Train the network as above, and test the "categorization performance" of the network. Does the network "fill-in" the category label when it is missing?

Step 3

One advantage of models like this is that they can be resistant to damage (called fault-tolerance). Add a new button to the program called "Lesion" which, when clicked, will randomly select ~1% of the memory weights in the model and reset them to zero (imagine this is the same as inducing "lesions" in a rat, but obviously much less painful!!). The basic outline of where you should write this function has been provided as a function called "lesion". Repeat your experiment above. However, after testing (to be sure the network learned the training patterns), press the lesion button. Describe how well the model does. Keep pressing the lesion button until performance drops considerably (i.e., the output pattern looks nothing like the input pattern). How fault-tolerant is the network in your opinion?

Step 4

At the top of the script you see that input is encoded such that OFF=-1, ON = 1. What happens if you change OFF =0 instead? Consider the equation above first, then try it in the code by making the change.

Step 5

What are some limitations to this simple model? Besides the situation implied by the last question, can you think of an example set of pictures that the model will perform poorly on but humans are likely to do well? One way to think about this

problem is with respect to the last example. Why does the model do poorly in this situation? Show a picture of the patterns you chose and describe what happens.

The lab write up will be due Wednesday April 23rd by 12pm.