Simulating Language: Lab 7 Worksheet

Download `learning3.py` from the usual place. The simulation explores the *cultural* evolution of a signalling system in a population of agents. In particular, we'll look at the effects on communicative accuracy of:

(i) different learning biases (weight update rules),
(ii) different populations models

The new part of our code starts with a set of parameter declarations; most of their meanings should all be straightforward by now. Note that the `interactions` parameter has two purposes in this simulation:

- to specify the number of utterances produced to create the data.
- to specify the number of times the data is randomly sampled in training.

```
# ----- new code below -----

meanings = 5            # number of meanings
signals = 5             # number of signals
interactions = 100      # both the number of utterances produced and the number
                        # of times this set is randomly sampled for training.
size = 100              # size of population
method = 'replacement'  # method of population update
initial_language_type = 'random' # either 'optimal' or 'random'
rule = [1, 0, 0, 0]     # learning rule (alpha, beta, gamma, delta)
```

Below this, the function `new_agent` creates a new agent. The weights in this agent's network depend on the parameter `initial_language_type`. If this parameter is set to 'random' (or in fact to anything other than 'optima;), every cell in the new agent's signalling matrix is set initially to zero. If the `initial_language_type` parameter is set to 'optimal', then the weights are configured such that the agent's initial weights ensure it signals optimally (m1 is conveyed using s1, m2 is conveyed using s2, etc). The function `new_population` then uses the `new_agent` function to create a population of new agents.

```
def new_agent(initial_language_type):
    system = []
    for row_n in range(meanings):
        row = []
        for column_n in range(signals):
            if (initial_language_type=='optimal') & (row_n==column_n):
                row.append(1)
            else:
                row.append(0)
        system.append(row)
    return system

def new_population(size,initial_language_type):
    population = []
    for i in range(size):
        population.append(new_agent(initial_language_type))
    return population
```

The Simulation

```
def simulation(generations, mc_trials, report_every):
    population = new_population(size,initial_language_type)
    data_accumulator=[]
    for i in range(generations+1):
        print '.', #comment this line out if you don't want the dots
        if (i % report_every == 0):
            data_accumulator.append(ca_monte_pop(population, mc_trials))
        data = pop_produce(population, interactions)
        if method == 'chain':
            population = new_population(size, 'random')
            pop_learn(population, data, interactions, rule)
        if method == 'replacement':
            population = population[1:] #This removes the first item of the list
            learner=new_agent('random')
            pop_learn([learner], data, interactions, rule)
            population.append(learner)
        if method == 'closed':
            pop_learn(population, data, interactions, rule)
    return [population,data_accumulator]
```

The simulation function runs the simulation. It takes three parameters, as follows:

`generations`: the number of generations in the simulation
`mc_trials`: the number of trials used to calculate communicative accuracy
`report_every`: the frequency with which data points (for printing in a graph) are returned

It then runs through the following steps:

1. Initialise the population
2. For each generation:
    a. Evaluate the population's communicative accuracy
    b. Produce some data
    c. Update the population (by adding new agents)
    d. Get (some of) the new population to learn from the data produced in 2a.
3. Output the final state of the population, and the list of communicative accuracy scores

The parameter `method` defines exactly how the population is updated, based on the scheme outlined by Mesoudi & Whiten:

chain: create a completely new population
replacement: remove one agent from the population, and replace with a new agent
closed: do not change the population at all

There are two things to note about the Python code in this function.

Review: slicing a list

The slice (`start : end`) operator allows us to take a slice of sequential elements between **start** and **end** from a list.

As usual in Python, the sequence extracts starts at **start**, and continues up to, *but not including*, **end**. Either the start or end (or both) indexes can be omitted, in which case the start or end of the list is assumed, respectively.

```
>>> x = ['a','b','c','d']
>>> x[1:3]
['b','c']
>>> x[1:]
['b','c','d']
>>> x[:3]
['a','b','c']
```

```
>>> 7 % 3
1
>>> 11 % 4
3
>>> 10 % 2
0
```

Modulus (Remainder)

The `x%y` operator returns the remainder of the division of x by y.

In the `simulation` function, the modulus operator is used to decide whether or not to calculate the value of ca_monte_pop and output it for the graphs. Can you see how it works?

Questions

Everyone should try questions 1-5. Question 6 is more open-ended, and needn't involve coding (but could).

1. Run the simulation for 500 generations, with 1000 mc_trials per generation, outputting communicative accuracy every 10 generations. Plot the values on a graph.
2. Experiment with different learning rules, re-running the simulation and inspecting the output. Which rules *construct* perfect communicative systems from random languages? How many different kinds of output pattern can you find with different rules? Plot them.
3. Change the population update method to 'chain', and re-run the simulation. What happens? Why? Increase the number of interactions by a factor of 100, and reduce the number of generations by a factor of 10. What happens now?
4. Experiment with the 'closed' method as well. What difference does the update method make to the way the simulation works?
5. You can use the same code to test whether a learning rule can *maintain* (rather than construct) a perfect system. Re-test the rules you looked at in answering question 2 above. If a rule fails the construction test, does that mean it always fails the maintenance test? If it passes the construction test, does it always pass the maintenance test?
6. In previous worksheets you have had the opportunity to write and play with code which models genetic transmission, spatial organisation, reinforcement learning, and so on. How would you fit these things in to this iterated learning model? Why might that be an interesting thing to do?