Simulating Language: Lab 6 Worksheet

Download `learning2.py` from the usual place. This simulation extends the previous model of learning to allow for different ways of updating weights. In our initial model of learning we simply increased the weight of connections between co-occurring signals and meanings. In our new model we will be adjusting the weights in four different circumstances:

1. Where meaning and signal are both active (we'll call the change to be made in this condition 'alpha').
2. Where only the meaning is active (we'll call this beta).
3. Where only the signal is active (gamma).
4. Where neither meaning nor signal is active (delta)

This gives us a set of possible learning rules, expressed as a list of four numbers, corresponding to the weight changes in each of these circumstances: [alpha, beta, gamma, delta].

Learning rules

We have changed the **learn** function from `learning1.py` so that it takes an extra parameter which specifies the weight-update rule to be used during learning.

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

def learn(system, meaning, signal, rule):
    for m in range(len(system)):
        for s in range(len(system[m])):
            if m == meaning and s == signal: system[m][s] += rule[0]
            if m == meaning and s != signal: system[m][s] += rule[1]
            if m != meaning and s == signal: system[m][s] += rule[2]
            if m != meaning and s != signal: system[m][s] += rule[3]
```

As before, this function takes a signalling system, a meaning, and a signal and modifies the values in the cells of the signalling matrix (or the connection weights in the network, depending on how you look at it). The function potentially changes the values of every cell in the matrix. Can you see how it is doing this?

```
>>> s = [[0, 0], [0, 0]]
>>> learn(s, 0, 0, [1, 0, 1, -1])
>>> s [[1, 0], [1, -1]]
```

Enter the code in the box and try it out. This trains a 2x2 matrix on a single utterance (meaning 0 paired with signal 0) with a particular (slightly weird) weight-update rule.

Try different rules. Which seem to make sense? Which corresponds to the one we used for the last worksheet?

Learning in a population

The next part of the code allows us to go from a single agent to a population (if we wish). **pop_learn** takes a list of signalling systems, a list of utterances (i.e. [meaning, signal] pairs), some number of interactions, and a learning rule. For the number of interactions specified, it trains a random individual in the population on a random utterance picked from the list of data.

```
def pop_learn(population, data, no_learning_episodes, rule):
    for n in range(no_learning_episodes):
        ms_pair = random.choice(data)
        learn(random.choice(population), ms_pair[0], ms_pair[1], rule)
```

The advantage of this might not be immediately obvious, but will be clear when we come to the next worksheet. For the time being, you can choose to use this function to train a single agent by simply building a population that has a single agent in it. Alternatively, you can use this to look at whether two or more agents may end up speaking similar languages when exposed to utterances picked at random from a set of training data.

Why are there three square brackets at the start of the variable "p"?

```
>>> p = [[[0, 0], [0, 0]]]
>>> pop_learn(p,[[0,0],[1,1]],100, [1,0,0,0])
>>> p [[[48, 0], [0, 52]]]
```

Try different learning rules and different data. How can we use this way of training to model different frequencies of different types of utterance?

We have a way for a population to learn from some data, but how about getting them to produce data, in order to evaluate how well they have learnt? **pop_produce** carries out this function. It takes a population and a required number of productions, and returns a list of utterances (meaning-signal pairs) generated by individuals picked randomly from the population:

```
def pop_produce(population, no_productions):
    ms_pairs = []
    for n in range(no_productions):
        speaker = random.choice(population)
        meaning = random.randrange(len(speaker))
        signal = wta(production_weights(speaker, meaning))
        ms_pairs.append([meaning,signal])
    return ms_pairs
```

Try generating data from a population that contains a single agent with a matrix made up of all zeros as weights. Now try training another similar agent with the data that your first agent created. What kind of data does the new agent produce after learning? The answer should depend on what your learning rule is.

Finally, we've added a population-based version of our Monte Carlo measure of communicative accuracy: **ca_monte_pop.** This takes a population and a number of trials, and return an estimate of

the chance that a random communication between members of the population will be successful - note that it just returns a single value, rather than a list of values (which was the case for previous implementations of Monte Carlo evaluation).

```
def ca_monte_pop(population, trials):
    total = 0.
    for n in range(trials):
        speaker = random.choice(population)
        hearer = random.choice(population)
        total += communicate(speaker, hearer, random.randrange(len(speaker)))
    return total / trials
```

Questions

1. Which weight-update rules "work" as a model of learning in terms of output data being similar to input data? Try this with an optimal language and a sub-optimal language.

2. What effect do the differences in weight-update rules have on *generalisation*? To find this out, try holding some data back and see what the agents do for unseen meanings/signals.

3.Which weight-update rules lead to better communication in the population?

4. In answering questions 1-3, you have probably been training agents on data that you provided. In a proper model of language learning, where would this data come from? Could you use the code above to model this?

5. Similarly, this code allows us to stipulate the weight-update rule that an agent or population of agents uses. In a complete model, where would this weight-update rule come from? How could you model this?