studywolf

a blog for things I encounter while coding and researching neuroscience, motor control, and learning

NOV 25 2012

61 COMMENTS

BY TRAVISDEWOLF LEARNING, PROGRAMMING, PYTHON, REINFORCEMENT LEARNING

Reinforcement learning part 1: Q-learning and exploration

We've been running a reading group on Reinforcement Learning (RL) in my lab the last couple of months, and recently we've been looking at a very entertaining simulation for testing RL strategies, ye' old cat vs mouse paradigm. There are a number of different RL methods you can use / play with in that tutorial, but for this I'm only going to talk about Q-learning. The code implementation I'll be using is all in Python, and the original code comes from one of our resident post-docs, Terry Stewart, and can be garnered from his online RL tutorial (https://github.com/tcstewar/ccmsuite). The code I modify here is based off of Terry's code and modified by Eric Hunsberger, another PhD student in my lab. I'll talk more about how it's been modified in another post.

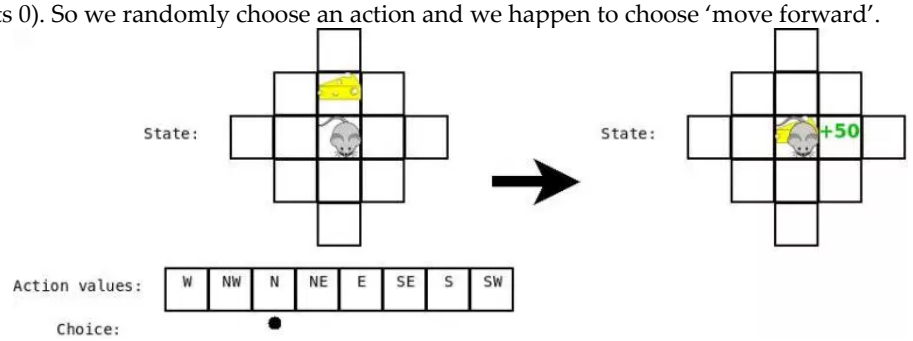
Q-learning review

For those unfamiliar, the basic gist of Q-learning is that you have a representation of the environmental states `s`, and possible actions in those states `a`, and you learn the value of each of those actions in each of those states. Intuitively, this value, `Q`, is referred to as the state-action value. So in Q-learning you start by setting all your state-action values to 0 (this isn't always the case, but in this simple implementation it will be), and you go around and explore the state-action space. After you try an action in a state, you evaluate the state that it has lead to. If it has lead to an undesirable outcome you reduce the `Q` value (or weight) of that action from that state so that other actions will have a greater value and be chosen instead the next time you're in that state. Similarly, if you're rewarded for taking a particular action, the weight of that action for that state is increased, so you're more likely to choose it again the next time you're in that state. Importantly, when you update `Q`, you're updating it for the *previous* state-action combination. You can only update `Q` *after* you've seen what results.

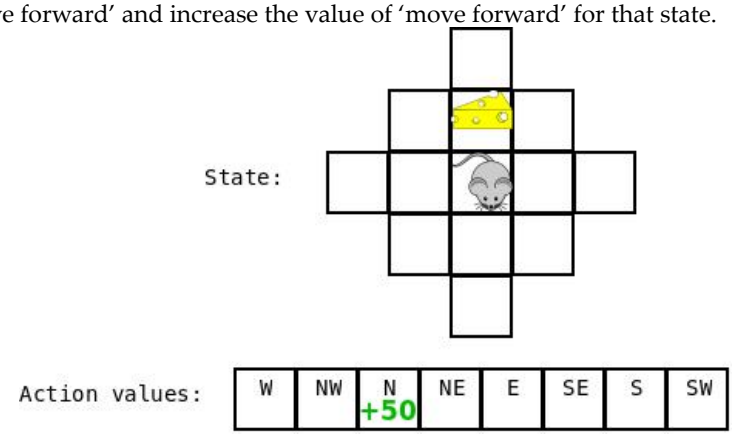
Let's look at an example in the cat vs mouse case, where you are the mouse. You were in the state where the cat was in front of you, and you chose to go forward into the cat. This caused you to get eaten, so now reduce the weight of that action for that state, so that the next time the cat is in front of you you won't choose to go forward you might choose to go to the side or away from the cat instead (you are a mouse with respawning powers). Note that this doesn't reduce the value of moving forward when there is no cat in front of you, the value for 'move forward' is only reduced in the situation that there's a cat in front of you. In the opposite case, if there's cheese in front of you and you choose 'move forward' you get rewarded. So now the next time you're in the situation (state) that there's cheese in front of you, the action 'move forward' is more likely to be chosen, because last time you chose it you were rewarded.

Now this system, as is, gives you no foresight further than one time step. Not incredibly useful and clearly too limited to be a real strategy employed in biological systems. Something that we can do to make this more useful is include a look-ahead value. The look-ahead works as follows. When we're updating a given `Q` value for the state-action combination we just experienced, we do a search over all the `Q` values for the state the we ended up in. We find the maximum state-action value in *this* state, and incorporate that into our update of the `Q` value representing the state-action combination we just experienced.

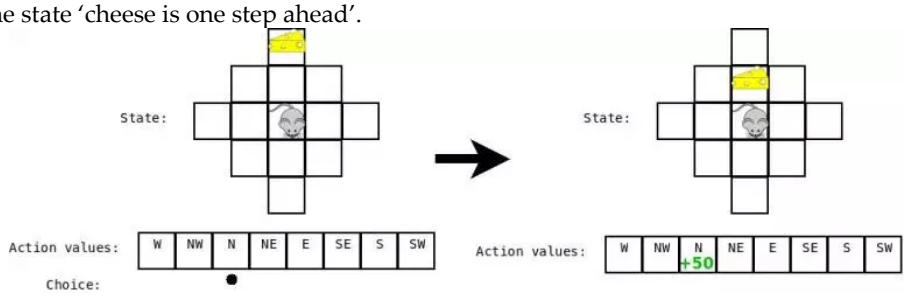
For example, we're a mouse again. Our state is that cheese is one step ahead, and we haven't learned anything yet (blank value in the action box represents 0). So we randomly choose an action and we happen to choose 'move forward'.



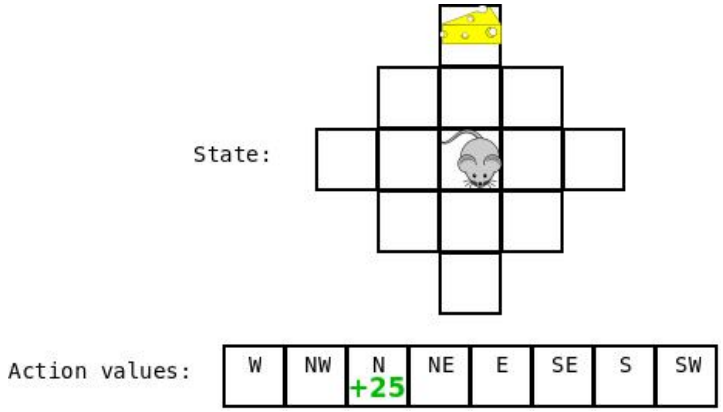
(https://studywolf.files.wordpress.com/2012/11/q-move-into-cheese.jpeg)
Now, in this state (we are on top of cheese) we are rewarded, and so we go back and update the `Q` value for the state 'cheese is one step ahead' and the action 'move forward' and increase the value of 'move forward' for that state.



(https://studywolf.files.wordpress.com/2012/11/q-cheese-ahead-updated.jpeg)
Now let's say the cheese is moved and we're moving around again, now we're in the state 'cheese is two steps ahead', and we make a move and end up in the state 'cheese is one step ahead'.



(https://studywolf.files.wordpress.com/2012/11/q-move-near-cheese.jpeg)
Now when we are updating the `Q` value for the previous state-action combination we look at all the `Q` values for the state 'cheese is one step ahead'. We see that one of these values is high (this being for the action 'move forward') and this value is incorporated in the update of the previous state-action combination.



(https://studywolf.files.wordpress.com/2012/11/q-cheese-near-updated.jpeg)
Specifically we're updating the previous state-action value using the equation: `Q(s, a) += alpha * (reward(s,a) + max(Q(s') - Q(s,a))` where `s` is the previous state, `a` is the previous action, `s'` is the current state, and `alpha` is the discount factor (set to .5 here).

Intuitively, the change in the Q-value for performing action `a` in state `s` is the difference between the actual reward (`reward(s,a) + max(Q(s'))`) and the expected reward (`Q(s,a)`) multiplied by a learning rate, `alpha`. You can think of this as a kind of PD control, driving your system to the target, which is in this case the correct Q-value.

Here, we evaluate the reward of moving ahead when the cheese is two steps ahead as the reward for moving into that state (0), plus the reward of the best action from that state (moving into the cheese +50), minus the expected value of that state (0), multiplied by our learning rate (.5) = +25.

Exploration
In the most straightforward implementation of Q-learning, state-action values are stored in a look-up table. So we have a giant table, which is size `N x M`, where `N` is the number of different possible states, and `M` is the number of different possible actions. So then at decision time we simply go to that table, look up the corresponding action values for that state, and choose the maximum, in equation form:

```
def choose_action(self, state):
    q = [self.getQ(state, a) for a in self.actions]
    maxQ = max(q)
    action = self.actions[maxQ]
    return action
```

Almost. There are a couple of additional things we need to add. First, we need to cover the case where there are several actions that all have the same value. So to do that, if there are several with the same value, randomly choose one of them.

```
def choose_action(self, state):
    q = [self.getQ(state, a) for a in self.actions]
    maxQ = max(q)
    count = q.count(maxQ)
    if count > 1:
        best = [i for i in range(len(self.actions)) if q[i] == maxQ]
        i = random.choice(best)
    else:
        i = q.index(maxQ)

    action = self.actions[i]
    return action
```

This helps us out of that situation, but now if we ever happen upon a decent option, we'll always choose that one in the future, even if there is a way better option available. To overcome this, we're going to need to introduce exploration. The standard way to get exploration is to introduce an additional term, `epsilon`. We then randomly generate a value, and if that value is less than `epsilon`, a random action is chosen, instead of following our normal tactic of choosing the max `Q` value.

```
def choose_action(self, state):
    if random.random() < self.epsilon: # exploration action = random.choice(self.actions) else: q =
            best = [i for i in range(len(self.actions)) if q[i] == maxQ]
            i = random.choice(best)
        else:
            i = q.index(maxQ)

       action = self.actions[i]
    return action
```

The problem now is that we even after we've explored all the options and we know for sure the best option, we still sometimes choose a random action; the exploration doesn't turn off. There are a number of ways to overcome this, most involving manually adjusting `epsilon` as the mouse learns, so that it explores less and less as time passes and it has presumably learned the best actions for the majority of situations. I'm not a big fan of this, so instead I've implemented exploration the following way: If the randomly generated value is less than `epsilon`, then randomly add values to the `Q` values for this state, scaled by the maximum `Q` value of this state. In this way, exploration is added, but you're still using your learned `Q` values as a basis for choosing your action, rather than just randomly choosing an action completely at random.

```
def choose_action(self, state):
    q = [self.getQ(state, a) for a in self.actions]
    maxQ = max(q)

    if random.random()  1:
        best = [i for i in range(len(self.actions)) if q[i] == maxQ]
        i = random.choice(best)
    else:
        i = q.index(maxQ)

    action = self.actions[i]

    return action
```

Very pleasingly, you get results comparable to the case where you perform lots of learning and then set `epsilon = 0` to turn off random movements. Let's look at them!

Simulation results

So here, the simulation is set up such that there is a mouse, cheese, and a cat all inside a room. The mouse then learns over a number of trials to avoid the cat and get the cheese. Printing out the results after every 1,000 time steps, for the standard learning case where you reduce `epsilon` as you learn the results are:

```
10000, e: 0.09, W: 44, L: 778
20000, e: 0.08, W: 39, L: 617
30000, e: 0.07, W: 47, L: 437
40000, e: 0.06, W: 33, L: 376
50000, e: 0.05, W: 35, L: 316
60000, e: 0.04, W: 36, L: 285
70000, e: 0.03, W: 33, L: 255
80000, e: 0.02, W: 31, L: 179
90000, e: 0.01, W: 35, L: 152
100000, e: 0.00, W: 44, L: 130
110000, e: 0.00, W: 28, L: 90
120000, e: 0.00, W: 17, L: 65
130000, e: 0.00, W: 40, L: 117
140000, e: 0.00, W: 56, L: 86
150000, e: 0.00, W: 37, L: 77
```

For comparison now, here are the results when `epsilon` is not reduced in the standard exploration case:

```
10000, e: 0.10, W: 53, L: 836
20000, e: 0.10, W: 69, L: 623
30000, e: 0.10, W: 33, L: 452
40000, e: 0.10, W: 32, L: 408
50000, e: 0.10, W: 57, L: 397
60000, e: 0.10, W: 41, L: 326
70000, e: 0.10, W: 40, L: 317
80000, e: 0.10, W: 47, L: 341
90000, e: 0.10, W: 47, L: 309
100000, e: 0.10, W: 54, L: 251
110000, e: 0.10, W: 35, L: 278
120000, e: 0.10, W: 61, L: 278
130000, e: 0.10, W: 45, L: 295
140000, e: 0.10, W: 67, L: 283
150000, e: 0.10, W: 50, L: 304
```

As you can see, the performance now converges around 300, instead of 100. Not great.
Now here are the results from the alternative implementation of exploration, where `epsilon` is held constant:

```
10000, e: 0.10, W: 65, L: 805
20000, e: 0.10, W: 36, L: 516
30000, e: 0.10, W: 50, L: 417
40000, e: 0.10, W: 38, L: 310
50000, e: 0.10, W: 39, L: 247
60000, e: 0.10, W: 31, L: 225
70000, e: 0.10, W: 23, L: 181
80000, e: 0.10, W: 34, L: 159
90000, e: 0.10, W: 36, L: 137
100000, e: 0.10, W: 35, L: 138
110000, e: 0.10, W: 42, L: 136
120000, e: 0.10, W: 24, L: 99
130000, e: 0.10, W: 52, L: 106
140000, e: 0.10, W: 22, L: 88
150000, e: 0.10, W: 29, L: 101
```

And again we get that nice convergence to 100, but this time without having to manually modify `epsilon`. Which is pretty baller.

Code
And that's it! There is of course lots and lots of other facets of exploration to discuss, but this is just a brief discussion. If you'd like the code for the cat vs mouse and the alternative exploration you can access them at my github: Cat vs mouse – exploration (https://github.com/studywolf/blog/tree/master/RL/Cat%20vs%20Mouse%20exploration).
To alternate between the different types of exploration, change the `import` statement at the top of the `egoMouseLook.py` to be either `import qlearn` for the standard exploration method, or `import qlearn_mod_random as qlearn` to test the alternative method.
To have `epsilon` reduce in value as time goes, you can uncomment the lines 142-145.