HW 8, Reinforcement Learning and Ants

Ben Sapp

Due: Tuesday, 17th before class

1 Background

It will probably help to skim the corresponding lecture notes from
`http://www.seas.upenn.edu/~cis099/cis99-rl.pdf`
It is a useful recap and lays out the notation we'll reuse in this assignment.

2 Q-learning

Everything we saw in lecture completely specifies how to estimate a value function $V(s)$ by having an agent wander about the world and adjust weights to fit its experience. But we left out one important detail in lecture (a little bit for simplicity's sake, but mostly because I ran out of time): suppose I have estimated $V(s)$ accurately, how do I decide which action to take? The answer is:

$$a^\star = \arg\max_a \sum_{s'} P(s \xrightarrow{a} s')V(s').$$

so once again we run into the problem of needing a model of the transition probabilities! One of the main points of doing temporal difference learning instead of value iteration was to deal with not knowing $P(s \xrightarrow{a} s')$.

It turns out we can again get around this by slightly reformulating the definitions. Instead of a *value function* $V(s)$ which models the value of every state, we're going to consider a *quality function* $Q(s, a)$ which models the value of (state, action) pairs.

It turns all the math is essentially the same, the Bellman equation still holds, the temporal differencing method still works. Similar to $V(s)$, $Q(s, a)$ can be interpreted as "the immediate reward of being in state s and issuing action a, plus the best expected discounted reward over all the future (state, action) pairs."

Once we've estimated $Q(s, a)$ via a temporal differencing approach, the best action is easy to compute:

$$a^\star = \arg\max_a Q(s, a)$$

The new algorithm is called "Q-learning" instead of "Temporal Difference learning," but the weight update is basically the same:

$$w_i \leftarrow w_i + \alpha(R(s) + \gamma \max_{a'} Q_w(s', a') - Q_w(s, a))f_i(s) \tag{1}$$

where we again use features to represent the quality function much like the value function before:

$$Q_w(s,a) = \sum_{i=1}^{n} w_i f(s,a)$$

In fact, this is the value function form. Whether you call it a "value" or "quality" function, whether you use a V or a Q and whether the action plays a role in the features and value/quality calculation is somewhat arbitrary. The only reason to make the distinction now is so that you can talk to other people and read textbooks and understand their terminology; a reinforcement learning *lingua franca*. So "value function" means $V(s)$ means Temporal Difference learning, and "quality or q-function" means $Q(s,a) \equiv V(s,a)$ means Q-learning. Annoying, right?

For completeness, I present to you

Q-learning

Repeat:

1. In current state s, determine an action a based on either exploration (a = random action) or exploitation ($a = \arg\max_{\tilde{a}}(Q(s,\tilde{a}))$
2. Execute action a and get to a new state s'
3. Adjust your belief of the value of state s by altering the weights

$$w_i \leftarrow w_i + \alpha(R(s) + \gamma \max_{a'} Q_w(s',a') - Q_w(s,a))f_i(s)$$

3 QLearnBot [35 points]

For this assignment, we want you to finish implementing Q-learning in the skeleton in `qlearner.py`, such that it beats `GreedyBot`.

That's it! The comments will guide you, but here is everything you must consider, along with pro-tips. Training can go haywire if you pick bad values of R, γ, α, etc:

- **Weight update.** This part is straightforward, just implement Equation 1 given the inputs in `QLearnBot::update_weights`.
- **Rewards.** You need to assign a numerical reward value for different events. Do this in `QLearnBot::get_reward`. We modded the localengine to inform your bot in the current turn what reward events happened in the previous turn. We give you fine-grained event details—the fraction of food the ant ate, the fraction of death the ant dealt, and whether it was just killed. You need to map these to reward values, as well as decide on a default, living reward. This is a somewhat delicate balance. If the living reward is positive, it will make no progress (why?). If the living reward is too negative, it will also make no progress (why?). You don't have to get too fancy, but make sure the rewards are reasonably scaled—how much is a death worth versus a food eaten? Is the reward for eating food big enough to make it worthwhile to live long enough to want to get it?

- **Step-size α.** Remember α is the step size to walk down the bowl of the learning optimization function $J(V)$. If your step is too big, you're going to go right past the bottom of the bowl and end up somewhere on the other side, farther away than when you started. Repeat this enough and your weight values will explode. A good rule of thumb is to err on the small side. If your weight values are blowing up, either your rewards are too optimistic or your step size is too big.

- **Discount γ.** When I got my implementation to work, I didn't really play with γ too much. Just set it to a reasonable value. (In the given code it is not set to a reasonable value).

- **Explore or exploit?** This is also tricky to get right. Make sure you explore a lot and your step size is small. Consider the following example: let's say you have a feature `close_to_food`. A majority of the time, being close to food is a good thing. But `GreedyBot` also likes food, and sometimes getting close to food means running into `GreedyBot` and dying. So realistically, let's say 60% of the time that `close_to_food` is active, the ant dies soon after. Given enough trials, the ant will eventually learn that the feature is good (positive weight), but if you play only a few games and then stop exploring, it is somewhat likely that your ant will be afraid of food. If at this point the trainer switches over to favoring exploitation over exploration, it may avoid food as much as possible, and it would be a very very long time before it hits a piece of food by accident and might change it's notion that food is good. A lot of exploration will help prevent this problem, as will a small α, which means that any individual experience will not affect value beliefs as much, and thus avoid fluctuation.

  As discussed previously, we can decide to explore with probability something like ϵ/t, but I found it much easier to debug by making the explore decision a step function of the number of games played. Example: if number of games is less than k, explore. If number of games is between k and K, explore with probability 0.5. If number of games is greater than K, exploit. Then you just decide values for k and K. This is easier to watch while debugging, otherwise you'll be wondering—did my ant just do that because my features are buggy or because he decided to explore a random action?

- **Features** Here are the basic features supplied, and the weights I learned with them:

```
Moving Towards Closest Enemy = 0.00258695
Moving Towards Closest Food = 0.0788598
Moving Towards Friendly = 0.0387113
Friendly adjacent = -0.144241
Closest food 1 away = 0.0906998
Closest food 2 away = 0.035803
Closest food 3 away = 0.0150907
Closest food 4 away = 0.00659417
Closest food > 4 away = -0.0737943
Closest enemy 1 away = -0.0162851
Closest enemy 2 away = -0.0225222
Closest enemy 3 away = -0.0226736
Closest enemy 4 away = -0.0341454
Closest enemy >4 away = -0.127516
```

3.1 What you need to do

We provide you with some basic weights in `saved_bots/qbot.json`. Simply load the file (the code should already be setup for you to do this).

We also composite these features with themselves to end up with 210 total features (some are redundant or impossible, but that doesn't really matter). The largest positive weight I learned was for `Moving Towards Closest Food AND Closest food 1 away` = 0.100307, and the largest negative weight was for `Friendly adjacent` = −0.144241, go figure.

These features are still fairly weak, and my learned bot only beats `GreedyBot` some of the time. First, try to see if you can learn with these features and if the weights make sense. Second, add at least one Basic feature of your choice to the `BasicFeatures` class in `src/features.py`. Note that the things you need to ad to this class is to append a feature name to the `feature_names` vector and then append the calculation for the binary result to the feature vector in the `extract` method. Refer to the file for examples. What were your most extreme positive and negative weights? Do these feature weights make sense, and why or why not? Is your bot able to beat GreedyBot, how often of 10 games can it beat greedybot? What new feature did you add, did it help? Why or why not? Let us know in your `README.txt`

3.2 Training

In the main function of `qlearner.py`, we've setup a training protocol which plays 50 games in sequence with random maps versus `GreedyBot`. I pretty much hardcoded everything, so you can train with

```
python run_qlearner.py
```

You can choose whether to step, play continuously (good for debugging) or play without the gui (much faster) by setting `PLAY_TYPE` accordingly in `qlearner.py` and you can modify the number of games to train by modifying `num_games` value in `run_qlearner.py`. The bot saves its weights by default to `saved_bots/qbot.json`, but it also saves a human readable list of features and weight values to `saved_bots/qbot-game-<game #>.txt`

3.3 Testing

Once you're happy with your training, you can use your learned weights to drive `ValueBot` by running

```
python valuebot.py
```

will load up the weights and feature definitions from `saved_bots/qbot.json` and play one game against `GreedyBot`.

4 Turnin Instructions

You should turn in the following:

- A `README.txt` text file in which you list what you chose for α, γ, $R(s)$, what features you used and what were the most negatively and positively weighted features, and finally your exploration-exploitation strategy. You can be as brief as possible. One word or one sentence answers are fine.
- Any code files you modified or created as you felt necessary. However, make sure calling `qlearner.py` as specified above will work in your submission.