Reinforcement Learning
Winter 2017
Assignment 5 - Policy Learning

Note: the codes proposed in this assignment are a modification of the codes proposed here:

http://ai.berkeley.edu/reinforcement.html#Welcome

There is no autograder function for this assignment, you will ned to report your observations.

Please have an environment-friendly policy, avoid printing these sheets !

1. In this last assignment, we will finally turn to a basic policy gradient algorithm based on a TD-based actor-critic. We will use a softmax policy, where the policy distribution is built as:

$$\pi_\theta(a\,|\,s) \propto \exp\left(\sum_i \phi_i(s,a)(s)\,\theta_i\right)$$

having the score function:

$$\nabla_{\theta_j} \log \pi_\theta(a\,|\,s) = \phi_j(s,a) - \mathbb{E}_{\pi_\theta}[\phi_j(s,.)] \tag{1}$$

You will need to download the modified code available on the course webpage, under the page "Organisation, examination and schedule", section "Home assignements".

We will start with implementing an exact policy learning algorithm. In the exact case, each state-action pair receives a parameter θ, and a trivial feature function ϕ that takes the value 1 when the pair is visited and 0 otherwise. In order for your code to run decently fast in the exact case, we will skip the eligibility traces.

The pseudo-code performing the policy learning in the exact case and without eligibility traces is very simple and will look like this:

```
For given state $s$, action $a$, next state $s_+$, reward $r$, do
1: $\delta \leftarrow r + \gamma V(s_+) - V(s)$
2: $\theta(s,a) \leftarrow \theta(s,a) + \alpha\delta \frac{\mathrm{d}}{\mathrm{d}\theta(s,a)} \log \pi_\theta(a\,|\,s)$
3: $V(s) \leftarrow V(s) + \alpha\delta$
```

Here we will only need three methods in the class "PiLearningAgent" in the file "pilearningAgents.py":

a Method **getPiValue** has to return the probability of a state-action pair under the policy distribution π_θ attributed to the self. You will need to handle the case where the policy does not exist yet. What probability should you return then?

b Method **getAction** will

- for a given state and for $\varepsilon > 0$ (training phase), draw an action from the policy distribution $\pi_\theta(a\,|\,s)$. The Python function **random.choice** will be very useful here, as the instruction:

np.random.choice(*List of actions* , p = *Distribution*)

will draw in the provided *List of actions* an element with probability given in the numeric list *Distribution*. Of course, these two lists must have the same length, and the list *Distribution* must sum up to 1. Make sure that your getAction method is purely based on the method getPiValue.

- for a given state and for $\varepsilon = 0$ (game phase) return the max-likelihood action:

$$a = \arg\max_a \pi_\theta (a \mid s)$$

Hint: this code can be very simple if you store your distribution as a Counter with the actions as keys and their probability as corresponding values. You can then extract the list of action via the .keys() method and the list of probabilities via the .values() method. The max-likelihood action is then obtained via the method .argMax().

c Method **update** which will be at the core of our policy learning. You will need to create and keep track of the value function and the policy parameter function.

- Think carefully of what the score function $\nabla_\theta \log \pi_\theta (a \mid s)$ will be. What will be the term $\mathbb{E}_{\pi_\theta} [\phi_j (s, .)]$ in the exact case ?

Note that the parameter ε is used here as a mean to detect if we are in the learning phase or the testing phase. We do not actually use this parameter in the algorithm.

Test your code by running:

- python gridworld.py -a q -k 100. The GUI will display the evolution of your policy π_θ on the grid. After 100 games, you ought to see something *similar* to this (at least on the "well-travelled" path):

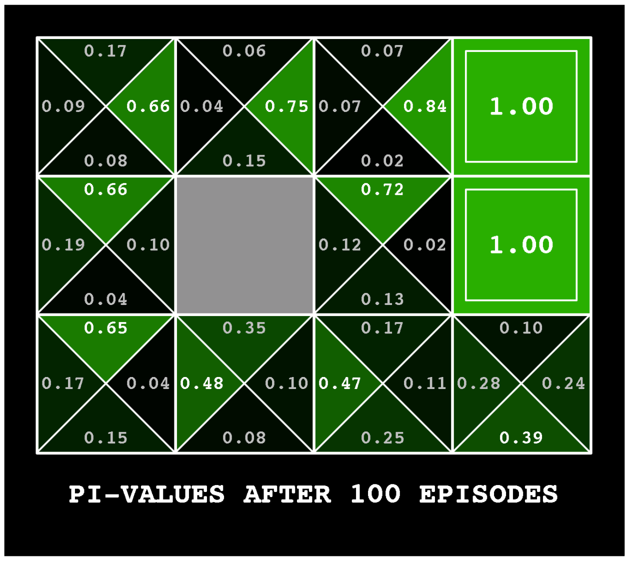

Figure 1: $\pi_\theta (a \mid s)$ function of the maze game.

- python pacman.py -p PacmanPIAgent -x 2000 -n 2010 -l smallGrid. After 600-800 training sessions, your agent should start earning positive scores. At the end of the learning, your Pacman agent should win every game.
- python pacman.py -p PacmanPIAgent -x 2000 -n 2010 -l mediumGrid. What happens? Why?

Use the option -q if you want to have fast results (no display of the games).

2. Reference? Let's now turn to an approximate policy learning algorithm. We will use the same feature functions as in the approximate Q-learning assignment, i.e. you can interrogate the set of feature functions via the command:

   Features = self.featExtractor.getFeatures(state,action)

   for any state-action pair, and receive a dictionary of features with their respective score.

   For the policy and value function, we will use:

$$\pi_\theta\left(a\,|\,s\right) \propto e^{\sum_i \phi_i(s,a)\theta_i} \qquad \text{and} \qquad V_w(s) = \mathbb{E}_{\pi_\theta}\left[\sum_i w_i\phi_i\left(s,.\right)\right] \tag{2}$$

   What will be your score function $\nabla_\theta \log \pi_\theta\left(a\,|\,s\right)$ now?

   You will need to write the code in the class "ApproximatePIAgent" of the file "pilearningAgent.py". You will have to edit the following two methods:

   - Method **getPiValue** will return for a given state-action pair the associated probability $\pi_\theta\left(a\,|\,s\right)$. Similarly as before, you will need to handle the case where the policy does not exist yet.
   - Method **getVValue** will return for a given state the associated value function V_w.
   - Method **update** will be at the core of our policy learning algorithm. It will work similarly to the previous question, with some modifications due to the fact that we use now non-trivial feature functions. Similar remarks as before apply.

   Test your code using $\lambda = 0.75$ by running (use the option -q if you don't want to see the games):

   - python pacman.py -p ApproximatePIAgent -a extractor=SimpleExtractor -x 50 -n 60 -l smallGrid, your Pacman agent should win 65-80% of the games, with typically about 500 points.
   - python pacman.py -p ApproximatePIAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid, your Pacman agent should win every game, typically with scores above 520 points.
   - python pacman.py -p ApproximatePIAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic, your Pacman agent will quite consistently win 80%-95% of the games with scores above 1300.

   The pseudo-code performing the policy learning with eligibility traces will look like this: Think carefully about

```
For given state $s$, action $a$, next state $s_+$, reward $r$, do
1: $\delta \leftarrow r + \gamma V\left(s_+\right) - V\left(s\right)$
2: $E_V \leftarrow \lambda E_V + \mathbb{E}_{\pi_\theta}\left[\phi\left(s,.\right)\right]$
3: $E_\pi \leftarrow \lambda E_\pi + \nabla_\theta \log \pi_\theta\left(a\,|\,s\right)$
4: $\theta \leftarrow \theta + \alpha\delta E_\pi$
5: $V \leftarrow V + \alpha\delta E_V$
```

   the dimension of each object in this code.

Questions

- Derive the expression (1).
- Can you provide an intuition of the pseudo-code described in Question 1?
- What is the meaning of the numbers you observe in Figure 1? How does this relate to your previous experiments with e.g. Q-learning?
- Comment on how the trade-off exploration-exploitation is approached in the policy-learning context.

- Why do we use a max-likelihood policy during the playing phase ? What happens if we keep the stochastic policy ?
- Can you explain/justify the validity of the choices (2), reasoning from their validity for supporting the construction of the approximate Q-value function?
- Why does your Pacman agent lose 20% of the time on the small and large grid and not on the medium one? What should we do to fix that ?
- What happened with your exact agent on mediumGrid (end of question 1)? Why? How can you fix it? With what drawback?
- How does this approach compare to Q-learning?