Reinforcement Learning
Winter 2017
Assignment 4 - TD-learning

Note: this assignment is an extended version of the assignment 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. **TD-Learning** In this assignment, we will ichange our Q-learning codes towards TD-learning.

   We will focus on backward TD-learning, which is often the most efficient approach to TD-learning, as well as the simplest to implement. You will make these modifications in the same structure as before, i.e. in the file **qlearningAgents.py** (make sure you keep a copy of your first Q-learning implementation).

   Deploying TD-learning will require some modifications. You will need to create an eligibility trace for your Q-value function (both for the exact and approximate version of your codes). When working on the exact learning (i.e. where the Q-function covers the full state-action pairs) our TD-learning update will have at its core the following SARSA pseudo-code:

For given state S, action A, next state-action pair S_+, A_+, reward R, do

$\delta \leftarrow R + \gamma Q(S_+, A_+) - Q(S,A)$
$E(S,A) \leftarrow E(S,A) + 1$
for all s' and legal a' **do**
    $Q(s', a') \leftarrow Q(s', a') + \alpha \delta E(s', a')$
    $E(s', a') \leftarrow \gamma \lambda E(s', a')$
end for

   Select A_+ using an ε-greedy policy based on Q. You will have to think a bit how and where to store A_+ between moves in the game.

   Do not forget to **reset your eligibility trace** at the end of each episode!! Within the update method, you can simply detect the end of an episode by testing whether your next state is the string TERMINAL_STATE.

   Important: think *very carefully* about how you should handle your eligibility trace!! More specifically, think about the meaning of the previous remark and what it entail in terms of storage!! This can make a substantial difference on the running time of your TD code!!

   Test your code by running:

   python gridworld.py -a q -k 50

   Note that your λ parameter ought to be defined internally in your learning functions. Check the behavior of your agent for $\lambda = 0$, it should be identical to the Q-learning you tested previously. You should get in the end something *similar* to

Figure 1: Q-value function of the maze.

Observe carefully what happens in the early stages of the TD-learning, in comparison to classic Q-learning for e.g. $\lambda = 0.5$. You can run your agent in manual mode using e.g.

python gridworld.py -a q -k 4 -m -n 0

in order to experiment with it at a slow pace. Try to drive your agent using consistently “north” twice followed with “east” twice. You see obtain this:

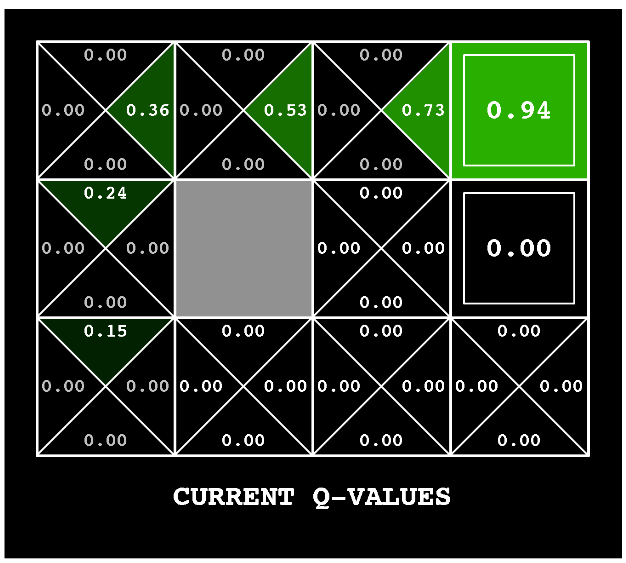

Figure 2: Q-value function of the maze after four games using consistently “north” twice followed with “east” twice.

2. Let’s test our new learning algorithm on the Pacman game!! Run the training with $\lambda = 0.5$:

python pacman.py -p PacmanQAgent -x 2000 -n 2100 -l smallGrid -q

Note: run this test at night, it will take forever. Your Pacman should win all the games.

3. We will now develop an approximate TD-learning algorithm, as an evolution of your approximate Q-learning algorithm.

   When doing approximate TD-learning, the pseudo-code is similar to the exact TD-learning, but has some striking differences. Your eligibility trace will no longer keep track of the state-action pairs visited recently, but of the strength of the features visited recently. The pseudo-code of the approximate TD learning reads as:

For given state S, action A, next state-action pair S_+, A_+, reward R, do

$Q = \sum_i w_i \phi_i (S,A)$ and $Q_+ = \sum_i w_i \phi_i (S_+, A_+)$
$\delta \leftarrow R + \gamma Q_+ - Q$
for all i enumerating features **do**
    $E_i \leftarrow \lambda E_i + \phi_i (S,A)$
    $w_i \leftarrow w_i + \alpha \delta E_i$
end for

---

   where $\phi_{1,\ldots,n}$ is your set of feature function, and $w_{1,\ldots,n}$ the corresponding weights. We will use again the set of feature functions built-in for the Pacman environment, in the class **featureExtractors.py**. You can interrogate the class via

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

   Important: to reset your eligibility traces in the approximate Pacman game, you will need to use a different condition than state = TERMINAL_STATE! You can detect the end of the game by e.g. using reward$\neq -1$ instead!

   You can test your code by running:

   python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid

   where your agent should always win. Test it further on the full Pacman game:

   python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic

   A typical outcome is that your Pacman typically wins the game at least 90% of the time.

4. Try now to make a Q-learning version of the TD approach, i.e. use:

$$\delta \leftarrow R + \gamma \max_a Q(S_+, a) - Q(S,A) \quad \text{and} \quad Q_+ = \max_a \sum_i w_i \phi_i (S_+, a) \tag{1}$$

   in your codes. What happens?

Questions

- Comment on the running time you get with your exact TD-learning algorithm, compared to the one you had with your exact Q-learning algorithm. Why is that so?
- By running our exact TD-learning algorithm for different values of λ on the smallGrid, i.e. running:

  python pacman.py -p PacmanQAgent -q -x 2000 -n 2010 -l smallGrid

  for different settings of λ, we have observed an instance of the following learning progressions:

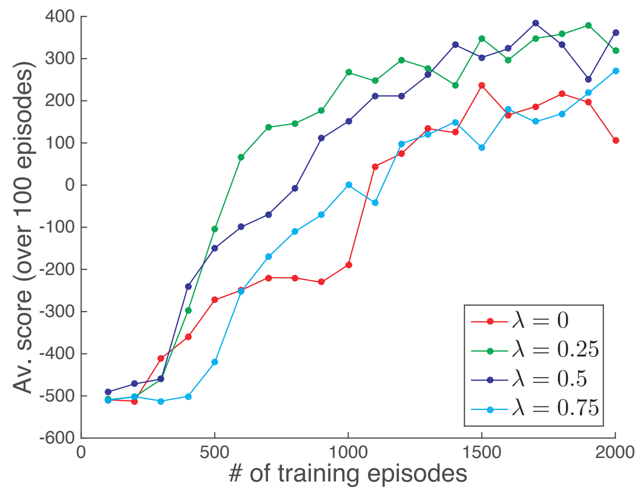

Figure 3: Progression of the average scores of the Pacman agent (ϵ-greedy policy) on the small grid with exact TD-learning for different values of λ.

  Comment on this observation. What fundamental trade-off in the learning process is addressed by the λ parameter?

- What happens to your approximate TD-learning agent for different values of λ?
- In what case using $\lambda = 1$ would be in theory justified?
- What difference do you see between deploying SARSA(λ) and using a TD approach on Q-learning?