Mountain Car Programming Project (Python)

Policy: This project can be done in teams of up to two students (all students will be responsible for completely understanding all parts of the team solution)

In this assignment you will solve the mountain-car problem by implementing on-policy Expected Sarsa(λ) with tile coding and replacing traces. Start with your existing code for running episodic Expected Sarsa(0) that you wrote in p1 and for tile coding that you wrote in p2 (copy these files to a new directory and edit them into new versions). You may also use the starter code in learning-starter.py. The code for the Mountain Car problem is available in the dropbox folder as mountaincar.py. The three actions (decelerate, coast, and accelerate) are represented by the integers 0, 1, and 2. The states are represented by tuples of two doubles corresponding to the position and velocity of the car.

mountaincar.py provides two functions:

- `mountaincar.init()`, which takes no arguments and returns the initial state. In this case, the initial position is randomly chosen from $[-0.6, -0.4)$ (near the bottom of the hill) and the initial velocity is zero.
- `mountaincar.sample(`S, A`) --> (`R, S'`)`, which returns a tuple of a reward and a next state, corresponding to taking action A in state S. Arrival in the terminal state is indicated by S' = None (as opposed to -1 for blackjack). In mountain car the rewards and transitions are deterministic. For this project we are using a modified version in which the reward is -1 for the coast action and -1.5 for accelerating or decelerating.

You will need to change your tile coder so that it covers the 2D state space for the car position and velocity as given in the textbook (Section 9.4). To start with, use the following parameters:

- numTilings = 4
- shape/size of tilings = 9 x 9, scaled to that an 8x8 subset exactly fills the allowed state space
- $\lambda = 0.9$
- $\alpha = 0.5/\text{numTilings}$
- $\varepsilon_\mu = \varepsilon_\pi = \varepsilon = 0$ (on-policy)
- initial parameter = random numbers between 0 and -0.01

Note that $\gamma = 1$ in this formulation of the problem and cannot be changed.

Your program should implement the following equations:

$$A_t = \begin{cases} \text{random action} & \text{with probability } \varepsilon \\ \text{arg}\max_a \boldsymbol{\theta}_t^\top \boldsymbol{\phi}(S_t, a) & \text{otherwise.} \end{cases}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \delta_t \mathbf{e}_t$$

$$\delta_t = R_{t+1} + \sum_a \pi(a|S_{t+1}) \boldsymbol{\theta}_t^\top \boldsymbol{\phi}(S_{t+1}, a) - \boldsymbol{\theta}_t^\top \boldsymbol{\phi}(S_t, A_t)$$

$$\mathbf{e}_t = \max[\gamma \lambda \mathbf{e}_{t-1}, \boldsymbol{\phi}(S_t, A_t)] \qquad \text{(replacing traces)}$$

where $\boldsymbol{\theta}_t$ is an n-component parameter vector, $\boldsymbol{\phi}$ is a feature function, returning n-component feature vectors every component of which is either 0 or 1, and $\mathbf{e}_t$ is an n-component eligibility trace vector. In the last equation above, the max is taken component-wise, that is, on each component of $\mathbf{e}$ separately. The number of components, n, is the total number of features, which is, with the standard parameters above, 4 x (9 x 9) x 3 (numTilings x tilesPerTiling x numActions). Basically, you call your tile coder on the state to get a list of four tile indices. These are numbers between 0 and 4 x 9 x 9 − 1. If the action is 0, then these four are the places where $\boldsymbol{\phi}$ is 1 (elsewhere 0). If the action is action 1, then you add 4 x 9 x 9 to these numbers to get the places where $\boldsymbol{\phi}$ is 1. And if the action is 2 then you add twice that. Basically, you are shifting the 1 indices into a unique third of the feature vector depending on which action is specified. This will pick out a different third of the $\boldsymbol{\theta}$ vector for learning about each action.

To implement the equations above you may want to follow the strategy of the boxed algorithm for Q-learning in Figure 9.9. (Modified to Expected Sarsa.)

Once your code is working, try a run of 1000 episodes. The initial episodes will be quite long, but eventually a good solution should be found wherein episodes are 100-200 steps long or less and produce returns from the starting state of less than 200. After good performance is reached, make a 3D plot of minus the learned state values. That is, plot

$$-\max_a \hat{q}(s, a, \boldsymbol{\theta}) \;=\; -\max_a \boldsymbol{\theta}^\top \boldsymbol{\phi}(s, a)$$

as a function of the state, over the range of allowed positions and velocities as given in the book. Use may use the provided function writeF and the file plot.py to make the 3D plot.

Now add an outer loop and run 50 independent runs of 200 episodes each, with the parameter $\boldsymbol{\theta}$ reset at the beginning of each run. Use Excel or some other plotting package to produce a graph of the average (over runs) of the return and of number of steps, versus episode number. Alternatively, if you have gnuplot installed, then you may use the provided plotreturns.gnuplot to make this plot.

What to turn in. Turn in your modified versions of learning.py and Tilecoder.py, your 3D plot, and your learning curve.

Extra Credit

Experiment with changing the parameters from the values listed above (but keep $\varepsilon_\mu = \varepsilon_\pi = \varepsilon$) to see if you can get faster learning or better final performance than is obtained with the original parameter settings. You can also change the kind of traces used (if you want to use dutch traces you will have to do some research on the internet to learn how they are defined for linear function approximation) and the tile-coding strategy (number of tilings, size and shape of the tiles). As an overall measure of performance on a run, use the average of all the rewards received in the first 200 episodes. If you can find a set of parameters that improves this performance measure by two standard errors, then you will earn an extra 8 points (out of 72 total on the project). To show the improvement, you must do many runs with the standard parameters and then many runs with your parameters, and measure the mean performance and standard error in each case (a standard error is the standard deviation of the performance numbers divided by the square root of the number of runs). If the difference between the means is greater than 2.5 times the larger of the two standard errors, then you have shown that your parameters are significantly better. It is permissible to use any number of runs greater than or equal to 50 (note that larger numbers of runs will tend to make the standard errors smaller).

To collect your extra credit, report the alternate parameter settings (or other variations) that you found, the means and standard errors you obtained for the two sets of parameters, and the number of runs you used in each case.

Extra Extra Credit (for 366 students only)

Finally, once you have found your favorite set of parameters, make a learning curve based on 500 runs of 200 episodes with them. Provide a printout of this curve, along with its single average performance number and its standard error on these 500 runs. This will be used to compare your team's performance with that of the other teams. The top three teams will receive additional extra credit:

- 1st place: +12 points on the project
- 2nd place: +8 points
- 3rd place: +4 points