Homework Assignment # 5

Due: Thursday, April 27, 2016, 11:59 p.m.
Total marks: 100

Question 1. [100 MARKS]

Programming question: Mountain Car

The objective of this question is to solve the mountain-car problem by implementing on-policy Sarsa(λ) with tile coding and replacing traces.

[Part one: 75 marks]

The first thing to do is implement the mountain car problem described in the text book (example 9.2). 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. Make sure you environment code contains an start function called at the beginning of every episode. The start function 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. 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 use the tile C coding software, available here:

(http://incompleteideas.net/rlai.cs.ualberta.ca/RLAI/RLtoolkit/tilecoding.html)
and described here:

(http://incompleteideas.net/rlai.cs.ualberta.ca/RLAI/RLtoolkit/tiles.html).

I have personally tested the C implementation and it works well.

Do not write your own tile coding software.

To start with, use the following parameters:

- numTilings = 4
- 9 x 9 tiles
- $\lambda = 0.9$
- $\alpha = 0.05/numTilings$
- $\epsilon = 0$
- initial parameter vector (θ_0): random numbers between 0 and -0.01
- $\gamma = 1$ (**cannot be changed**)

Your on-policy Sarsa(λ) agent should implement the following equations:

$$A_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \text{argmax}_a \theta^\top \phi(S_t, a), & \text{otherwise} \end{cases}$$

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha \delta_t \mathbf{e}_t \\ \delta_t &= R_{t+1} + \gamma \theta_t^\top \phi(S_{t+1}, A_{t+1}) - \theta_t^\top \phi(S_t, A_t) \\ \mathbf{e}_t &= \max(\gamma \lambda \mathbf{e}_{t-1}, \phi(S_t, A_t)) \text{(replacing traces)} \end{aligned}$$

where θ_t is an n-component parameter vector, ϕ 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 e separately.

How do we use the tile coder with actions? 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 active tile indices. These are numbers between 0 and 4x9x9-1. If the action is 0, then these four are the places where ϕ 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 ϕ is 1. And if the action is 2 then you add twice that. Basically, you are shifting the active indices into a unique third of the feature vector depending on which action is specified. This will pick out a different third of the θ vector for learning about each action.

To implement the equations above you may want to follow the strategy of the boxed algorithm for semi-gradient Sarsa(λ) with binary features (covered in our lecture on eligibility traces—check the slides).

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 around 200 steps long or less and produce returns from the starting state of less than 300. After good performance is reached, make a 3D plot of minus the learned state values. That is, plot

$$-\max Q_\theta(s, a) = -\max \theta^\top \phi(s, a)$$

as a function of the state, over the range of allowed positions and velocities as given in the book. That means you will need to sample valid position and velocity values, tile code them and then query the negative max of your learned q-function. Take a look the sample python code provided in the file plot.py to get an idea how to make your 3D plot.

Now add an outer loop and run 50 independent runs of 200 episodes each, with the parameter θ reset at the beginning of each run (e.g., agent_init). Produce a graph of the average (over runs) of the number of steps to goal, versus episode number—a learning curve like you have made in previous assignments.

What to turn in: (1) turn all code used to produce your graphs, (2) your 3D plot, and (3) your learning curve.

Part two: 25 marks

Experiment with changing the parameters from the values listed above (e.g., ϵ, λ etc.) 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 (like dutch traces you will have to do some research on the internet to learn how they are defined for Sarsa with linear function approximation) or the tile-coding strategy (number of tilings, size and shape of the tiles). As an overall measure of performance on a run, use the sum of all the rewards received in the first 200 episodes. 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 or equal to 50 (note that larger numbers of runs will tend to make the standard errors smaller).

What to turn in: (1) 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. (2) A learning curve based on 500 runs of 200 episodes with your parameter settings.

Homework policies:

Your assignment will be submitted as a single pdf document and a zip file with code, on canvas. The questions must be typed; for example, in Latex, Microsoft Word, Lyx, etc. or must be written legibly and scanned. Images may be scanned and inserted into the document if it is too complicated to draw them properly. All code (if applicable) should be turned in when you submit your assignment. Use the RL-glue framework available on the course webpage (your code will be in c/c++), and any language of choice for plotting the results (learning curves).

Policy for late submission assignments: Unless there are legitimate circumstances, late assignments will be accepted up to 5 days after the due date and graded using the following rule:

on time: your score 1

1 day late: your score 0.9

2 days late: your score 0.7

3 days late: your score 0.5

4 days late: your score 0.3

5 days late: your score 0.1

For example, this means that if you submit 3 days late and get 80 points for your answers, your total number of points will be $80 \times 0.5 = 40$ points.

All assignments are individual work, no exceptions. All the sources used for problem solution must be acknowledged, e.g. web sites, books, research papers, personal communication with people, etc. You are expected to solve problems from scratch. That means, if you are asked to code something, do not find code online and modify it. Write it from scratch yourself. Academic honesty is taken seriously; for detailed information see Indiana University Code of Student Rights, Responsibilities, and Conduct.

Good luck!