CS 287, Fall 2009
Problem Set #2 : Reinforcement Learning

Deliverable: 4-10 page pdf write-up by Friday November 20, 23:59pm. For complete credit you should complete three (out of four) exercises and a total of 6* of extensions. If you go beyond that, you will get extra credit (i.e., you can go > 100) accordingly.

NOTE: Please refer to the class webpage for the homework policy.

Various starter files are provided on the course website.

When making your write-up, make sure to include and discuss plots (and if helpful, snippets of code) which are helpful in demonstrating that your system works and in answering the questions. However, there is no need to include all plots being generated, similarly for code you write. (In fact, including all on both accounts would make for a pretty poor write-up.) Your writing should be self-contained, though it can reference other material (just like a paper would).

1. Approximate linear programming and approximate value iteration

You are free to use any linear programming solver of your choice for this question. If you don't have in-depth experience with a particular solver already, I recommend you try out CVX. CVX can solve a broad class of convex optimization problems, including linear programs, but also quadratic programs (which you will need for the imitation learning question!). The file cvx_intro.m provides download/installation instructions, as well as a few examples of problems being solved with CVX.

We will study the application of approximate linear programming and approximate value iteration to building a control policy for the game of tetris. The file main_play_and_collect_data_and_computer_play.m illustrates the use of the starter files available to you. There should be no need to investigate other files.

The game of tetris has a very large state space. We will use a linear function approximator with the standard set of 22 features. The features are covered in the lecture notes, and a function computing them from a board configuration is provided (tetris_standard_22_features.m). Our resulting value function approximation for a board configuration b is of the form $V(b) = \theta^\top \phi(b)$.

1. Note that the state of the system comprises both the board configuration *and* the current block to be placed, with each of the seven blocks appearing with equal probability. The standard Bellman equation contains the value of the entire state, not of just the board configuration. Write out a proper Bellman equation which only contains $V(b)$ (and no value function which depends on the entire state).

2. Implement the approximate linear programming (ALP) approach. Per the constraint sampling: you could code up a simple baseline policy, you could play games yourself, or you could load tetris_game_log.mat, which contains a log of a game (or two) I played. Investigate how well the ALP approach works as a function of: (a) The number of constraints being sampled (when sub-sampling, it would be most natural to subsample equally spaced, maximizing the diversity of board configurations),[1] (b) The choice of discount factor, and (c) The choice of reward function: One choice is a reward of 1 for every block being placed,

[1]On my laptop, it took 200s for CVX to parse (most of the time) and solve the ALP when sampling every 5'th board configuration for the constraint set, and 68s when sampling every 10'th board.

a reward of zero when the "game-over" state has been reached; Another choice is a reward of 20 − maxheight, i.e., the distance of the highest filled square from the top of the board, and also zero for game-over. Max height is feature $\phi(20)$ in the standard list of 22 features.

As your evaluation metric for each of the settings, run the resulting policy (i.e, the greedy policy w.r.t. the found value function) 20 times and report the mean and standard error of the number of blocks placed. To make your comparisons more meaningful, compare on the same 20 sequences of blocks.

3. Implement value iteration with linear function approximation. Use the same board configurations as for the ALP approach. Use least squares for the function approximation step. Compare the performance of the resulting policy (i.e., the greedy policy w.r.t. the value function obtained) with the results from the ALP approach. Note, as covered in lecture, this approach need not converge and this question is here simply to make sure you have tried out and understand perhaps the simplest approximate dynamic programming approach.

Each of the following improvements are possible extensions:

1. (**) **Bootstrapping.** After solving the ALP use the greedy policy w.r.t. the obtained value function to sample states. Then re-solve the ALP with the newly sampled states and iterate.
2. (**) **Feature selection.** Generate a very large vocabulary of features. Investigate how the ALP performs with an increasing number of features. (You might want to add some form of regularization to avoid overfitting.)
3. (**) **Smoothed ALP.** Recently the "smoothed ALP" has been proposed. Modify your code to use this smoothed version and compare performance with the original formulation. (See, http://web.mit.edu/ vivekf/www/papers/salp.pdf for details on the smoothed ALP.)

2. Imitation learning—behavioral cloning

Play a few tetris games (or load the provided game-log tetris_game_log.mat). Then use the game log as supervised learning examples to learn a value function using the max-margin method (with a fixed margin of 1) covered in lecture. Evaluate how well the learned policy performs as a function of the number of training examples. Make sure to have the slack penalty variable C vary over a reasonable range of values for each training set size and pick the value for C that results in the best performance for each training set size.

Possible extensions:

1. (**) **Probabilistic/Logistic formulation.** Implement and evaluate the probabilistic/logistic formulation (see lecture notes for the equations) which encodes the assumption that the demonstrator is using a softmax policy. Compare performance with the max-margin version.
2. (**) **Margin scaling.** In structured prediction tasks, it is customary to have the margin not simply always be one, but have the margin depend on how different two board situations are. Come up with two reasonable distance metrics and use them as your scaled margin rather than simply using the constant margin of one for all constraints. Evaluate its performance.

3. (**) **Combine ALP and behavioral cloning.** Find a way to combine the behavioral cloning optimization problem and the ALP into a single optimization problem. Investigate its performance.

3. Policy iteration with TD learning Policy iteration algorithms repeatedly iterate over policy evaluation and policy improvement. In this problem, we will consistently use the same policy improvement step, namely, the new policy becomes the greedy policy w.r.t. the value function estimate from the latest policy evaluation step. Evaluate the following two classes of policy evaluation: (In each case, use the standard 22 features.)

1. TD learning: Run n games under the current policy. Use TD learning to learn the value function of the current policy. After the n games have finished, use the current estimate of the value function parameter θ to define the policy for the next set of n games. Try various settings for the learning and exploration rate. Feel free to fix $n = 20$. No need to tinker with feature scaling, but comment on why this might help?

2. LSTD learning: Similar to TD learning (and similarly, you can fix $n = 20$), but now use LSTD to estimate the value function after the n games have been played. Note that LSTD estimates θ as $A^{-1}b$ for appropriate A and b computed from the execution traces. Investigate what happens if you build A and b just from the last n runs, versus from all past data, versus from all past data where older data gets discounted. (The latter worked best for me.) A and b can be built up incrementally. Hence there is no need to keep all experience traces around. Make sure to include in your write-up how you build up A and b as data comes along.

Briefly discuss your results. Make sure to include learning curves showing the number of bricks successfully placed as a function of the trial number. Also evaluate how well the final policy does and compare to the previously studied approaches. Note: the graph showing the learning curve does not necessarily show the performance of the final policy, as during learning deliberate randomness is introduced.

For your reference: in my experience, TD learning was quite hard to get to learn much at all (but you might have better luck). If your experience is similar, make sure to describe why you think this is the case (especially contrast TD with LSTD) and which variety of settings you tried. Using LSTD for policy evaluation resulted in a policy that enabled it to place on the average 1800 blocks after 1000 games. My epsilon greedy had $\epsilon = 0.1\frac{20}{20+\text{trialnumber}}$. It was fairly robust to choices of n. I used as reward function distance from the highest filled square to the top of the board; I used a discount factor of 0.9 for the rewards; I used a discount factor of .99 for discounting the old data when incorporating the last n runs of data.

Possible extensions:

1. (*) **Updating the inverses directly.** Rather than keeping track of A itself, describe updating the inverse of A directly. (No need to implement.)

2. (*) **Importance sampling.** Describe how you could leverage importance sampling to re-use data from runs under previous policies in a proper way in LSTD. (Rather than simply discounting.) Would it still be possible to incrementally build A and b in a similar way? (No need to implement.)

4. Policy search

The likelihood ratio method provides us with the following episodic estimate of the gradient:

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)};\theta)(R(\tau^{(i)}) - b).$$

In this question we will compare a few instantiations of the likelihood ratio method as applied to tetris. We will use the following policy class:[2]

$$\pi_\theta(b'|b,p) = \frac{\exp(\theta^\top \phi(b'))}{\sum_{b''}\exp(\theta^\top \phi(b''))}$$

where b is the current board situation, p is the current piece to be placed, b' is a possible next-time board situation we could choose, and b'' indexes over all possible next-time board situations. *In this question we will use a feature function ϕ with only two features, as provided in tetris_2_features.m.*

While in practice you would most likely want to optimize your choice of m for your problem at hand, in this question feel free to run all your experiments for $m = 5$.

The policy search proceeds by iteratively performing m runs under the current policy, and performing a policy gradient update of the form:

$$\theta \leftarrow \theta + \alpha\hat{g} \tag{1}$$

where α is a step-size parameter. The choice of α can greatly affect the efficiency of the optimization procedure. For this question you may choose α for a given procedure by eye-balling whether it looks like the parameters are shooting back and forth (α too large), or they are taking a very long time going in the same direction (α likely too small). In my experiments, all my choices of α fell in the $[0.01, 1]$ range.

Evaluate and discuss the following approaches: (about 200 policy gradient updates, which would make for 1000 total runs, should suffice to observe the salient properties)

1. Use a zero baseline, and perform the updates as in Eqn. (1).
2. Use a zero baseline, and perform the following update instead:

$$\theta \leftarrow \theta + \alpha\frac{\hat{g}}{\|\hat{g}\|_2} \tag{2}$$

   This update ensures all updates are of size α in the parameter space.

3. Use an estimate of the value under the current policy as the baseline. You could use an exponentially decaying average of the value obtained in past runs.
4. Natural gradient with the same baseline. When using the natural gradient, the renormalization through division by the norm of the gradient (or natural gradient) is undesirable: doing so would make the stepsize dependent on the choice of parameterization. Find a renormalization which makes the stepsize independent of the parameterization and use this renormalization in your implementation.

[2]The following equality might inspire you to get a numerically better conditioned computation: $\frac{\exp(x)}{\exp(x)+\exp(y)}) = \frac{\exp(x-\max\{x,y\})}{\exp(x-\max\{x,y\})+\exp(y-\max\{x,y\})}$. Note this equality fails to hold when computing with finite precision, and that's why you might want to favor one side of the equality over the other side.

Possible extensions:

1. (**) **Exploit temporal structure and TD.** We saw in lecture that we could exploit the temporal structure by using the following expression for the gradient estimate:

   $$\hat{g} = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H}\nabla_\theta \log \pi_\theta(u_t|s_t)(\sum_{k=t}^{H} R(s_k, u_k) - b_k).$$

   Investigate the performance of this version. For the time-varying baseline, use the following expression:

   $$b_k = \hat{V}(s_k) - \gamma \hat{V}(s_{k+1})$$

   where $\hat{V}$ is a value function estimate obtained by simultaneously running (R)LSTD.

2. (**) **Finite difference policy search.** Use the θ learned in one of the previous questions (using 22 features!) as a starting point for a finite-difference based policy search. Compare performance when fixing and not fixing the random seed during the optimization. [Note: for the final evaluation, make sure to evaluate on a set of random seeds which was not used during the optimization!]

Other possible extensions:

1. (**/****) Application of one of the foredescribed exercises to another system that is sufficiently interesting. You get ** if you apply an extension only, you get **** for the basic component of an exercise. I included a very nice acrobot simulator in the starter code (thanks to Woody Hoburg!), and it could be fun and interesting to learn to swing it up through TD-style learning or policy search. In case of policy search, the RTRL of PS1 is one particular case of policy search. For this problem set you should investigate one of the methods covered in the scope of this problem set. From a practical point of view, I think you would be more likely to swing up the acrobot by running sysID, then RTRL (or some sequential quadratic programming) in simulation, and then close a feedback loop around it (designed, e.g., using iLQR) when running on the real system. Still, investigating the methods on the acrobot could be a worthwhile intellectual exercise.

2. (**/****/******) Ideas of your own which are related and you want to try out—contact me to make sure I consider it sufficiently interesting. Number of *'s depends on what you suggest to try out.

5. Feedback (Voluntary, no credit involved)

Towards improving this assignment for next year: if you had one or two constructive suggestions, that would be helpful. E.g., in question X it would have been nice had you provided the code for Y b/c it took a really long time compared to the amount of learning involved; in question Z it would have been a great learning experience if we had to do W ourselves instead of being given the code; it would be great if the lecture slides had a bit more detail on V.