CS 287 Fall 2015 Problem Set #2: Function Approximation and LQR

Deliverable: pdf write-up by Wednesday September 23rd, 11:59pm, submitted through Gradescope. Note that some of the starter code will generate plots. Don't include all of them, include whatever is sufficient to show that you correctly solved the problem. Whenever including a figure, make sure it is accompanied by a discussion of that figure.

Your pdf write-up should be typeset as follows: 1(a) an page 1; 1(b) on page 2; 1(c) on page 3; 1(d) on page 4; 1(e) (i) on page 5; 1(e) (ii) on page 6; 1(e) (iii) on page 7; 1(e) (iv) on page 8; 1(e) (v) on page 9; 2(a) on page 10; 2(b) on page 11; 2(c) on page 12; 2(d) on page 13; 2(e) on page 14. Note that for most of these you wouldn't need the entire page, so frequently the bottom part of a page will be left empty. Also note that you only need to solve three out of five subquestion of 1(e) — please insert a blank page for the one you skip over. Thanks!

Refer to the class webpage for the homework policy.

Various starter files are provided on the course website.

You are free to use any linear programming solver of your choice for this assignment. 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. Note that sometimes CVX is not able to solve a problem with one solver, but is able to solve it with another solver, so it can be worthwhile trying out the different solver options.

1. Function Approximation

We will study the application of linear programming with function approximation to building a control policy for the game of tetris. The file main_ALP_starter.m illustrates the use of the starter files available to you. There should be no need to investigate other files, though you might have some fun with main_play_and_collect_data.m.

The game of tetris has a very large state space. We will use a linear function approximator with the standard set of 22 features. 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 $V(b) = \theta^\top \phi(b)$. As this function approximation ignores the current block k, we are going to use a slightly altered Bellman equation:

$$\begin{aligned} V(b) &= \frac{1}{K}\sum_{k=1}^{K} V(b,k) \\ V(b,k) &= \max_a R(b'(b,k,a)) + \gamma V(b'(b,k,a)) \end{aligned}$$

where $k = 1, \ldots, K$ indexes over the different blocks, $V(b,k)$ is the value of board configuration b with current block k, a indexes over all possible actions (the product space of 4 rotations and 10 translations), and b' is the next board configuration achieved when starting from b with block k and taking action a.

(a) Write out the approximate linear program you obtain with this new formulation.

(b) Implement the approximate linear programming (ALP) approach. Per the constraint sampling: your starter file loads a log of a couple of games Pieter played. Investigate how well the ALP approach works as a function of 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] 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.

(c) Discuss performance as a function of the choice of discount factor. Same evaluation metric as above.

(d) For this subquestion only, change the reward function. The default reward function gives 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. Change it to reward of 1 for every block being placed, a reward of zero when the "game-over" state has been reached. With the default set of game states I provided you with, you should see that this did not work–how can you tell from the values you found as the solution of the LP? Would there be a choice of states that would still enable learning with this simpler reward function? Discuss why you believe so, but no need to implement with such set of states.

(e) Investigate three of the following five extensions:

  (i) **Smoothed ALP.** The ALP formulation asymmetrically considers errors in the Bellman update equation. The "smoothed ALP" has been proposed to address this. 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.)

  (ii) **Bootstrapping the ALP.** 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. Discuss your findings. Note this isn't very well understood yet. You might also want to read about it here: `http://web.stanford.edu/~bvr/pubs/tetrischapter.pdf`.

  (iii) **Bootstrapping with Policy Iteration.** After solving the ALP to initialize, use the greedy policy w.r.t. the obtained value function to sample state, next block, and action triplets (s, k, a). Then modify the ALP formulation to compute the value of the policy that was used to produce the (s, k, a) triplets (i.e., perform policy evaluation on the samples). You will have to use slack variables to account for nonequality of the Bellman equation and penalize the ℓ_1 norm of the vector of slack variables. This should have considerably fewer constraints than the regular ALP for the same number of samples. Then use the greedy policy w.r.t. the value of the policy evaluated in order to sample more (s, k, a) triplets and iterate. Note that for good performance, you might need to initialize the policy with the SALP solution instead of the ALP solution. In addition, a trust region constraint will be useful. The magnitude of θ's last (22nd) component is typically much larger than the rest of θ. To ensure the trust region treats each component equally, let $\theta' = \theta$ except $\theta'(22) = 1/L * \theta(22)$ for some large

[1] On a 2013 MacBook Pro, it took 200s for CVX to parse and solve (parsing took most of the time) the ALP when sampling every 5'th board configuration for the constraint set, and 68s when sampling every 10'th board.

L ($\in [100, 1000]$) and then the trust region constraint is the form ($||\theta'_k - \theta'_{k-1}||_2 \leq C$ for some C). You can define θ' within a cvx program as a function of θ. Experiment with different values of C, L, number of sampled state triplets per iteration, and slack penalty. Discuss your findings.

(iv) **Cross Entropy.** An alternate approach to learning a control policy for Tetris uses the cross entropy method, which directly considers the performance under the current parameter vector θ rather than the Bellman errors. Modify your code to use cross entropy to search across values of θ for the best policy. Experiment with different values of the hyperparameters: the discount factor, N, Z, and ρ (see the paper). Evaluate performance by plotting the average number of blocks placed by the mean policy after each iteration. (See `https://hal.inria.fr/inria-00418930/document` for implementation details.) Using the standard list of 22 features, after 15 iterations (which takes 15 minutes total on a current desktop) the mean policy is able to place 800+ blocks. After 21 iterations (34 minutes total), it is able to place 1400+ blocks. This used a discount factor of 0.9 and the same hyperparameters as in the cited paper. Mean policy performance was computed by averaging over 5 runs.

(v) **A different extension.** This could be something you read / heard about somewhere, or an original idea of your own.

2. LQR

(a) **LQR for a Linear System.** In this question you will implement and evaluate LQR for a linear system. See p_a_starter.m for detailed instructions.

(b) **LQR-based Stablization for a Nonlinear System, Cartpole.** In this question you will implement and evaluate the application of LQR to stablization of a nonlinear system. See p_b_starter.m for detailed instructions.

(c) **LQR-based Stabilization of a Helicopter in Hover.** In this question you will implement and evaluate the application of LQR to stablization of a helicopter in hover. See p_c_starter.m for detailed instructions.

(d) Consider the following optimal control problem, which considers a linear system with additive noise and quadratic cost:

$$\begin{array}{ll} \min_{x,u} & \mathrm{E}[\sum_{t=0}^{T-1} x_t^\top Q x_t + u_t^\top R u_t] + \mathrm{E}[x_T^\top Q x_T] \\ \text{s.t.} & x_{t+1} = A x_t + B u_t + w_t, \quad \forall t = 0, 1, 2, \ldots, T-1 \end{array}$$

with w_t independent random vectors with $\mathrm{E}[w_t] = 0$, and $\mathrm{E}[w_t w_t^\top] = \Sigma_w$.

Find an LQR-like sequence of matrix updates that computes the optimal cost-to-go at all times and the optimal feedback controller at all times. Describe the expected cost incurred in excess of the expected cost in the case when there is no noise.

(e) Consider the following optimal control problem, which considers a linear system with multiplicative noise and quadratic cost:

$$\begin{array}{ll} \min_{x,u} & \mathrm{E}[x_T^\top Q x_T] \\ \text{s.t.} & x_{t+1} = Ax_t + (B + W_t)u_t, \quad \forall t = 0, 1, 2, \ldots, T-1 \end{array}$$

Here $Q \in \mathbb{R}^{n_X \times n_X}, A \in \mathbb{R}^{n_X \times n_X}, B \in \mathbb{R}^{n_X \times n_U}$ are given and fixed. $W_t \in \mathbb{R}^{n_X \times n_U}, t = 0, 1, \ldots, T-1$ are independent random matrices with $\mathrm{E}[W_t] = 0$. Higher-order expectations involving W_t will show up. These higher-order expectations are *not* assumed to be zero, and you should just keep these expectations around—no need to try to simplify these (and not possible anyway unless additional assumptions were made).

Find an LQR-like sequence of matrix updates that computes the optimal cost-to-go at all times and the optimal feedback controller at all times.