Project P4 — One More Thing

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)

The objective of this final project is to take something that you have learned in the course and extend it by adding one new thing. For example, you could take a method that you have learned about in class and apply it to a new problem. Or you could take an algorithm learned in the course and try to extend it in some way to perform better or on a wider class of problems. Or you could do something new by combining two ideas that you have seen in the course, such as double learning and Sarsa. The key is not to introduce more than one new thing. That would make drawing conclusions from your results harder, as well as possibly making it difficult to finish the project in the small amount of time still available. An exception that I would allow is if the one new thing that you want to investigate requires both a simple new problem and a simple new algorithm. For example, if you wanted to investigate bandit algorithms that work well on non-stationary problems you would presumably introduce both the new non-stationary problem and a new algorithm.

Throughout the course we came across a number of opportunities for final projects. Here are some of those that I can recall, or that other students have done, which could be exemplary:

1. Design a new bandit algorithm suitable for non-stationary problems, and test it on a version of the 10-armed testbed modified to be non-stationary. For the problem, one might allow the true action values to drift continually, by incrementing them on every step by a small amount of mean-zero normally-distributed noise. Then you would modify one of the existing methods, perhaps just by changing from sample averages to exponential moving averages. Finally, you could compare performance to that of other methods in Chapter 2.

2. Apply LSTD(λ) to the mountain-car problem. This would presumably require a smaller tile coding, to keep the number of features down such that a quadratic complexity algorithm is feasible. Still, just to implement and apply LSTD(λ), and get reasonable results, is a good goal. One complication is that it is less clear how to apply LSTD(λ) for control. So, perhaps, you should just make this a prediction problem. That is, you might build in a reasonable policy and then learn its value function. Here's one possibility: the policy that always accelerates in the direction the car is already moving in. You would probably want to compare with, say, Sarsa(λ). You could compare to Sarsa(λ) with the same feature representation, which you should be able to beat, and to Sarsa(λ) with a larger feature representation such that its overall computational complexity was approximately the same as that of LSTD(λ)

3. Implement a basic policy-gradient algorithm on a small, natural problem and

show that it works. Perhaps this could be done on one of the existing problems in the book: cliff-world, mountain-car, queue-admission, ...

4. Implement a policy-gradient method with a single continuous action, normally distributed, and show that it can learn both a good mean and a good standard deviation. Maybe this would be best as a single state problem, a bandit with a continuous action. The action values $q_*(a)$ could also be a normal distribution. Pick its mean $N(0, 1)$ and with variance 1. Then produce rewards as $q_*(A_t) + N(0, 1)$. Then you should see the learned mean and variance go to reasonable things (maybe increasing in variance if the variance starts out small) and then the variance fall to zero in the long run.

5. Apply gradient-TD or emphatic-TD methods to any problem (but ideally a very simple one or one that we have already seen in the course) and report on what happens and what it suggests to you.

6. Apply any method you have learned in the course to learn to optimally control a simple physical system such as a robot.

7. Apply Monte Carlo Tree Search to a new game. Compare to another planning method such as dynamic programming or conventional minimax search.

8. I expect to add further ideas to this list as I think of them and talk to various students.

I would like to stress again that you should choose a small, manageable project. I would much rather see a simple project done well than a larger project never finished or that we aren't sure was done properly and whose results we can't trust to be correct.

Finally, you will want to prepare a short writeup on your project. I would like you to keep this as short as possible consistent with explaining your work (but no shorter). You should not repeat anything that is well presented in the book or even well presented elsewhere in the public literature, but rather should just cite that presentation. There are a few important things that you have to include in your writeup, and a few key guidelines to its structure:

- You have to fully explain the whole project such that I could give your report to one of your classmates and they could reproduce the whole of it and get the same results (excepting possibly that they would have a different sequence of random numbers).

- You should clearly separate the presentation of the *problem* from the presentation of the *solution methods* that you apply to the problem. To do this, I find that a good approach is to alternate between problem and solution becoming successively more specific. For example, you might start in an introduction with the vague or general idea of an issue in solving some problem (e.g., non-stationarity in bandits). Then switch to an algorithmic idea that might address the issue and explain the algorithm (e.g., a running-average algorithm). Then switch back and explain a particular task (e.g., a non-stationary version of the 10-armed testbed). Now you switch two more times, first giving all the details of the algorithm, including the precise range of algorithm-parameter values used in your experiment, and then the details of number of episodes, runs, performance measures, etc. At this point everything is explained and you are ready to present your results, discuss them, and reach any conclusions that seem valid.

- You should clearly separate your presentation of the results on the problem from the conclusions that you draw from those results. An easy way to do this is by switching tense, as discussed below.
- You should use present and past tense properly. Algorithms and problems both just are, and they should both be described in present tense. Your experiment, on the other hand, is something that you did. It and the results you obtained should be described in past tense. Finally, any conclusions that you draw from the results should be described in present tense again. For example, an algorithm maintains (present tense) an approximation to the action-value function, and a problem has (present tense) a 3-dimensional state space with 4 discrete actions, and the actions affect (present tense) the state is such and such a way. In the experiment, you applied (past tense) 10 instances of the algorithm, each with a different value of the step-size parameter, to the problem. Each algorithm instance was initialized (past tense) with a particular weight vector, and then run (past tense) for 100 episodes. Then the whole thing was repeated (past tense) for 30 runs. The random seed was initialized (past tense) to the same value for all algorithm instances at the beginning of each group of 30 runs. For each run, the total reward on each episode was recorded (past tense) and averaged (past tense) over runs to produce the learning curves shown in Figure 1. I conclude that the differences between algorithm A and algorithm B at episode 100 are (present tense) statistically significant because they are more than twice the standard error in both means. Thus, on this problem, algorithm A clearly performs (present tense) better than algorithm B, though it remains unclear whether it performs (present tense) better in general. Further experiments on other problems would be needed to make that conclusion.

"You can lead a horse to water, but you can't make it drink"

Some other guidelines in choosing implementing your project:

- You may want to use either use the tile-coding software you wrote in Project 3, or the standard tile-coding software that you can find on the internet. In the latter case, you may have to figure out how to call C code from Python. This is straightforward, and it would be good for you to figure out how to do it, and how to use the standard code, which provides several useful features that your implementation does not. Using the standard tile coding software will satisfy the requirement for this project that you introduce one additional challenge.

- You likely will need to vary parameters to see how well each algorithm can do. To show that one algorithm is better than another, you normally would have to show that it is not difficult to find parameter values for the winning algorithm that enable it to perform better than the losing algorithm does at any setting of the losing algorithm's parameter values.
- In presenting your results, you will probably want to have a sequence of graphs. Typically, early graphs present more detail and validate the basic idea of the experiment, and then you progress to more summary and all-encompassing graphs, just as, in Project 2, you first did learning curves, then a summary number for each learning curve as a function of the step-size parameter. In general, it is helpful to the reader to present your results incrementally, gradually adding complexity and summarizations.

"Extra credit"

The extra credit on this course is not course credit at all, but a chance for small fame, by having your course project immortalized as a section in the second edition of the RL textbook. As you know, the textbook is full of small examples of implemented algorithms on small problems. In my mind, an ideal project is one that could be an additional such example, and if you allow it and your project is suitable, then the small fame would be to have your example in the second edition.

Due Date

The project is officially due on December 7, the last day of class. You should size your project and reign in your ambitions consistent with that date. However, if your good plan for a project runs into unforeseen trouble, there will be no penalty for missing this date, as I expect to be out of town at that time. Please submit your projects, writeup and code as appropriate, as a zip file on eclass.