Sample Responses From a Previous Class

(Response to Chapter 7 of the book's first edition)

What is the advantage of averaging multiple n-step returns (ie, say n=2 and n =4) instead of just using n=4?

The plots of RMS error against alpha in figure 7.2 show that high n and low n get higher error than intermediate n values. I understand why low n would get high error since updating farther back will help (to an extent). But why will high n get high error?

Why are states that have been visited recently more eligible for updating under the eligibility trace method? Logically, I would think that we would want states visited less frequently to be more eligible, since as much information as possible should be obtained when that state is visited since it is unlikely that it will be visited again in the near future.

Are there cases in which Watkins' algorithm would be the "right" algorithm to use? It seems like cutting off the traces at each exploratory action would be really detrimental since this would slow down learning at the beginning when exploratory actions are more likely.

Can you discuss Peng's algorithm some in class? In particular, I'm somewhat confused about the statement that it is neither on-policy or off-policy and the mention of a fictitious final transition.

What's the downside to using replacing traces? Are there cases where it is better to use accumulating traces?

(Response to Chapter 7 of the book's first edition)

In Figure 7.2, why was the RMS error averaged over the first 10 episodes? Why not just average over the 100 runs of the experiment at the 10th episode. Since earlier episodes will have definitely more error, it would have been easier to understand the differences for a single episode.

It also took me some time to completely understand Fig. 7.2. If I understand it correctly, the 1000 step return performs poorly in comparison to say the 8 step return just because of bootstrapping? Even though it performs a 1000 (or < 1000 for terminal) updates to the value function. This was fairly interesting since these were the starting episodes, and until some value backs up to a considerable number of states, bootstrapping should have a minimal effect.

Although the book explicitly states that the forward view is more theoretical and the backward view is good for intuition, but it appears that the forward view has no meaning since you really cant "look forward" for the rewards. Is this what the authors imply as well? Does the forward view only exist because it is a means of proving the convergence guarantees of TD(lambda). So is there a meaning of online and offline in terms of the forward view? Since the backward offline case is same as the forward one, can we not write out the same convergence guarantees for the backward view? Why do we bother with the forward view at all?

The discussion in Sec 7.8 was very interesting, as I had not realized this problem with eligibility traces till this point in the chapter. Example 7.5 was particularly enlightening. However I have some qualms about the modification suggested by Singh and Sutton. I can see how experimentally it might improve learning, but is it completely justified? If we have a domain which does not force this much repetitiveness in state, wont it be better to let the term for other states remain. Both the 19 state problem and the one given in Example 7.5 force repeating states/state-action pairs to get to the final result.

(Response to Chapter 7 of the book's first edition)

Below is my reading response for Ch. 7 - Eligibility Traces. In general, I liked the chapter, which presents some nice mathematical generalizations of previous algorithms, and which appear to empirically work well. The ideas were interesting, and as for the time of writing, it seems there was room for further research to be done - is it still the case today? In addition, I think I might have some answers or alternative solutions to some questions that are presented as unclear / unknown in the text.

A brief summary: Eligibility traces is a generalization that captures or approximates both TD and MC (Monte-Carlo) methods. In the case of offline updates it generalizes both TD and MC, and in the case of online udpates it approximates them.

The first idea was of looking at both TD and MC as two extreme cases of n-step TD methods, which generalized TD and MC. For n==1 an n-step method is just TD from previous chapter and for n==infty it is MC. As n-step returns has the *error reduction property* (equation 7.2) they can be shown to converge to the correct prediction. Empirically, it was shown that in some cases n-step returns does better than both TD and MC, for some n, s.t. 1 < n < infty.

Next, further generalization is introduced, called TD(lambda), which uses a weighted average of all possible n-step returns, for n = 1,...,Infty, where weights usually decrease exponentially for every future step. Here, MC and TD are still two extreme cases of this generalization: for lambda = 1 it is (every-visit) MC and for lambda = 0 it is the one-step TD. Also, an online TD(1) is even more general MC method that we saw in previous chapter - it doesn't wait until an end of episode to update the estimation, but keeps a running estimation that is updated in every step.

What is discussed above is called a "forward view", as we look into rewards in the future. In order to make it practical, a backward view of the same algorithm is presented, in which every state that is visited causes updates in all past states that were visited before it. For these states, the current state is the "future", so it should cause an update in their estimations. For offline updates, the forward and backward views are equivalent, and there is a simple algorithm to compute them, based on the backward view. For online updates, the backward view only approximates the forward view, as state values are updated during an episode, rather then only at the end. However, the approximation is claimed to be pretty good, and in many times it works better than offline updates.

All the above refered to the prediction problem. As usual, we are interested in solving the control problem. Thus, the next step is generalizing Sarsa and Q-learning to Sarsa(lambda) and Q(lambda). For Sarsa(lambda), the generalization is straightforward. For Q-learning, there is a problem, because of the off-policy nature of the algorithm. As the algorithm is off-policy, it computes action-values for policy that it doesn't execute (the optimal one). Therefore, it is not clear what should be an n-step return for n-trace that involves exploratory actions. Three methods try to generalize it, and for me it seemed that Watkins' method should be the more correct one (below), but reported empirical results doesn't seem to support it.

The eligibility traces are extended to actor critic methods as well, in which there are two sets of traces. One trace is for state values, to be used by the critic, and the other is for action values, used by the actor.

There is also a section about "replacing traces". I believe that a forward view of this algorithm means that, from a state s, we look into the future for n-returns only until the next visit in s. I have a few comments about it below.

The weighted average of traces above used exponentially decreasing weights. There are other methods to give weightes to n-step returns, which were relatively new at the time of writing.

Questions / comments (according to the order in the text):

-- pg 166, equation 7.2: I got a little confused here. From the equation it seems that for n-step return, the higher n the faster the convergence. So it seems that the best thing is to use an n-step return with n as high as possible.

Clearly this is not the case here, as TD(lambda) give higher weights to lower n's, and still gives better results then, say monte-carlo. What am I missing...?

-- pg 172, last paragraph before exercise 7.4: They say that TD(lambda) is not clearly better or worse then n-steps return for some n. But I think a main advantage of it is that one doesn't need to wait n-steps before value estimations are updated - is it correct?

-- Section 7.4, pg 176: It's not too important, but I think the proof could be much simpler if they had shown equivalence for every *s_t*, rather then for every *s*. The math should become simpler, as we don't need to account for all the visits in s at once. Rather, we would only focus on s_t, and later show that combining all the visits of s gives the required result.

-- For Sarsa(lambda) and Q(lambda), I didn't see a discussion of whether updates are done online or offline. Does it matter? are there comparison results?

-- Regarding the different versions of Q(lambda): None of the three versions (Watkins', Peng's, Naive) seemed to give satisfactory generalization. - For me it seemed that Watkins' version should be the better one: This is because in later section they mentioned that practically, the weights of n-step returns quickly decrease, and so the n-step returns have significant weights only for small n's. Therefore it seemed that Watkins' algorithm shouldn't introduce a problem by the fact that it usually looks only into the close future(?). - Do we know more today about the correct way to generalize to Q(lambda)?

-- Regarding Actor-critic methods (also from last chapter) How important / common are they today?

-- Page 186-188, replacing traces - It seems that the forward view of it is that, from a given state s, account for the n-step returns only until the next visit to s (or end of episode if comes first). Is it correct?

- In page 188 they mention the relationship between replace-trace method and first-visit MC. They say it is not known how it extends to the discounted case. I think I have an extension for the discounted case, as follows. I think that all we should do is change equation 7.13 (pg 186), such that the second condition sets the trace to 1 *only in the first visit to s_t*. I think this should give a first-visit MC for the discounted, non-episodic case.

- Figure 7.18: It seems this example show a problem with eligibility traces, at least as they were defined so far (or I am missing something), and I think I might have alternative solutions, in addition to the "replacing traces" method, as follows. What we usually want is to compute the expected return from a state (or state-action). However, if the value of the "wrong" action in this example can be higher than the corresponding "right", then what is being computed is not any kind of average of the returns following a visit. This is because in a discounted case, every "wrong" action that is previous to the "right" action whould have a lower return than the "right" action's return, as its final reward is further in the future, and thus discounted more. Consequently, any average of the "wrong" actions should also be lower then the average return of "right" actions. So it seems there is some double (or more) counting of events somewhere(?). I think that one possible way to solve the "right-wrong" problem is by setting the eligibility to 1 only in the *first* visit, and later decrease it as in the eligibility traces case (that woule be a "first-visit" eligibility trace). Another way to do that is to make sure somehow that the final value of a state would be some kind of an average of all samples, which can be done in different ways.

-- pg 190, regarding variable lambda: It seems that n-step returns worked best for some n > 1 (e.g. 9 in the random-walk experiment). - Following that example, it seems it is worth trying to set small weights for very small and very large n's and higher weights for n's that are somewhere "in the middle". Was it tried before? - Another option is to look no more than n-steps into the future - was it tried before? (after implementing it in code, I guess everyone does that, from efficiency reasons?) - What do we know today about methods with variable lambda? is any of them better then the methods presented in the chapter?

-- In general: I wasn't sure why eligibility traces were specifically considered as a mechanism for temporal credit assignment (pg 165). Isn't every algorithm we discussed so far solves the credit assignment problem implicitely?

-- Finally, do we know today more about the comparison of eligibility traces / MC / one-step TD methods?

Experiments:

I implemented sarsa(lambda) and compared it with Q-learning, sarsa and MC, on the race-track example that I used in previous chapters. When I implemented sarsa(lambda) according to the pseudo-code from the book without optimizations, run-time was way too long... I then increasingly optimized it until it started to run in reasonable time. The current optimized implementation keeps eligibility traces, i.e. e(s,a), only when the value is > 0.1. For lambda = 0.9 this means the last 22 visited states, and for lambda = 0.8 this means the last 10 visited states. I used in my experiments the lambda = 0.8.

This set of experiments was a little more constrained then previous chapters. I ran two main experiments, for track1 and track2 from pg 128 of Sutton and Barto.. In each experiment I compared the optimized sarsa(lambda) with the other algorithms, using alpha=0.1, gamma=1, epsilon=0.1, 50,000 iterations. As in previously, I used epsilon-greedy policies. Each experiment is an average of 10 runs.

For both tracks (named grid1 here), the optimized Sarsa(lambda) converged faster then Q-learning and Sarsa, but still slower than Monte-Carlo.

The learning curves are attached as png files. All the code and results is in the attached tar file: -- raceTrack.py contains all the code -- The results/ directory contains all the results of all runs of the two experiments. -- A README file explains how results can be reproduced.
