EE194/290S: ML for Sequential Decision under Uncertainty **Fall 2018**

Lecture 25: Introduction to Reinforcement Learning II

Lecturer: Anant Sahai, Vidya Muthukumar *Scribes: Yagna Patel, Lakshya Jain*

25.1 Introduction

25.1.1 Recap

Discounting

In the previous lecture, we understood reinforcement learning as thinking of it as approximate dynamic programming. We saw how we can use a dynamic programming approach to computationally solve for the optimal policy. Furthermore, we brought up the concept of discounting, and how we can view discounting from a drug dealer's perspective.

The Drug Dealer perspective: There is some probability that the drug dealer finds himself dead. This perspective is useful because it helps connect discounting to time horizons, i.e. if you expect an algorithm to find a policy for some problem, and you get rewards after 100 time steps, and you discounting is 0.9, then you shouldn't expect this to work. Discounting of 0.9 means that your expected duration to live is 10 time steps. You can't expect someone to plan for 100 time steps if they are dead in 10 time steps.

Now, if you are learning from simulated data, you can simulate this drug dealer dying effect by augmenting a terminal state (in which you are dead) to your simulated system, e.g. by tossing a coin.

Value/Policy Iteration

We also talked about value iteration and policy iteration:

- *Value Iteration:* We want to estimate the optimal cost-to-go function by first guessing the cost-to-go function and applying the bellman operator repetitively. Value iteration takes the maximum value of possible actions from our current node and will converge to the optimal value for each node. It is determined by the equation $V(s) = \max_a \sum_{s \in S} p(s^{'}|s,a)[r(s^{'},a) + \gamma V^{'}(s^{'})]$ and is performed iteratively for each state until the optimal values V converge. The loop for value iteration is:

```
initialize V(s) arbitrarily
loop until V(s) converges for all states s
      for all states s
           for all actions a
                Q(s,a) = Σ_{s'∈S} p(s'|s,a)[R(s,a,s') + γV'(s')]
           V(s) = max_a Q(s,a)
```

- *Policy Iteration:* We have a policy for which we will evaluate the cost-to-go function. We then assume that we follow that policy in the future and determine the action we will take given that policy. We then apply the bellman operator, and if the policy performed well, then it becomes the new policy. Policy iteration is found via an iterative fashion, where you begin with a policy π and modify it. The loop for policy iteration is:

```
initialize π(s) arbitrarily
loop until π(s) is unchanged
      for all states s
            a' = arg max_a R(s,a,s') + Σ_{s'∈S} p(s'|s,a)[R(s,a,s') + γV_π(s')
            Update the action taken by policy π at the current state to be
            a' and continue iteration through the states until end of loop.
```

25.2 Learning from Data

When dealing with value iteration and policy iteration, we must know how to calculate the expectations in an uncertain environment. In these uncertain environments, we must look and learn through data. One strategy we can use is to explore then commit.

25.2.1 Explore then Commit Strategy

1. *Simulator:* The first step is to get data. We assume that we have access to a simulator from which we can extract data samples of what would happen if we were in some state and we took some action. We assume that this simulator has two functionalities:
   (a) *Trace*
      i. *Trace with Interaction:* At each state, we choose the action we want to take. Through these interactions we gather data.
      ii. *Trace with Policy:* At each state, we allow a predefined policy to take actions. Through the actions made by this policy we gather data.
   (b) *Restart*[1]
2. *Get Data:* Now that we have a simulator to get data from, the next step is to get the data. In order to do this, we have to interact with the simulator by providing it with some actions. The simplest approach is to use an explore-only random policy, i.e. taking actions uniformally at random.
3. *Transition Probabilities and Rewards:* After getting this data, we can now estimate transition probabilities and rewards, from which we get $\hat{J}$, the estimated optimal cost-to-go. We get $\hat{J}$ by repeatedly applying the bellman operator using the estimated transition probabilities and rewards. Upon that, we can find $\hat{\pi}$, the optimal policy. We then commit to this policy $\hat{\pi}$.
4. *Validation:* After finding the optimal estimated policies, we should add one more step for validation (to check correctness). We validate $\hat{\pi}$ by using the same simulator.

This essentially predicts an optimal (policy, cost) pair answering the question of "What is the optimal policy? And what does it cost?". We can test this on the simulator by running the policy several times and averaging the costs.

There is still a big problem, however: the estimation step might not get you your optimal policy, because it might give you the wrong transition probabilities. We could overexplore boring states and underexplore interesting states. Thus, the logical thing to question is our approach of exploring and *then* finding policies; perhaps its best to first estimate an optimal policy and then use that to explore. This can be viewed as policy iteration's counterpart of Stochastic Gradient Descent, where we begin with a random policy, use it to explore, and iteratively improve the policy until it converges.

For this, we need to have a methodology for the following puzzle: Given policy $\hat{\pi}$, how

[1]One can imagine certain situations, e.g. video game playing, where the concept of restarting is much more complex. For example, one may have a simulator that allows to restart from any previous configuration an agent may have encountered, or even restarting from any arbitrary state.

do we estimate its cost $J_{\hat{\pi}}$? We can write it as $J_{\hat{\pi}} = \frac{1}{M}\sum_{m=1}^{M} C_{\hat{\pi}}(x, m^*)$, where the term inside the summation represents the total cost of trace m from the time we saw state x until the end of the trace. This averaging can be written in a more iterative fashion, however; it can be written as

$$J_{\hat{\pi}}^m = J_{\hat{\pi}}^{m-1}(x) + \gamma_m(C_{\hat{\pi}}(x, m) - J_{\hat{\pi}}^{m-1}(x))$$

Note that this is an iterative process of approximating J_π that is just like stochastic gradient descent with stepsize γ_m. If we set $\gamma_m = \frac{1}{m}$, we will actually achieve the exact value of the summation above! In general, we can play around with the value of γ and run it for more traces to achieve better or worse approximations of J_π experimentally.

25.3 Q-Learning

Q-learning essentially decouples your knowledge of what we should be doing from what we actually did. Here we will consider the un-discounted case. We define the Q-factor for every state-action pair, (x, a), as

$$Q^*(x, a) = \sum_{j=1}^{n} P_{x\to j}(a)\left[r(x, a, j) + J^*(j)\right]$$

Now, applying Bellman's equation, we get that

$$Q^*(x, a) = \sum_{j=1}^{n} P_{x\to j}(a)\left[r(x, a, j) + \min_{a'} Q^*(j, a')\right]$$

Now, if we have multiple, possibly infinite, such states, then the Q-factor is updated using a step-size $\gamma > 0$, while the other Q-factors are unchanged, i.e.

$$\begin{aligned} Q_{\text{new}}(x, a) &= (1-\gamma)Q_{\text{old}}(x, a) + \gamma\left[r(x, a, j) + \min_{a'} Q_{\text{old}}(j, a')\right] \\ &= Q_{\text{old}} + \gamma\left[r(x, a, j) - Q_{\text{old}}(x, a) + \min_{a'} Q_{\text{old}}(j, a')\right] \end{aligned}$$

Therefore, the Q-learning algorithm is:

```
initialize Q(s, a) arbitrarily
loop
      initialize s
      repeat until s is terminal
           select action a from s based on Q(s, a) value
           take a and observe reward r and new state s'
           Q(s, a) ← (1 − γ)Q(s, a) + γ [r(s, a, s') + min_a' Q(s', a')]
```

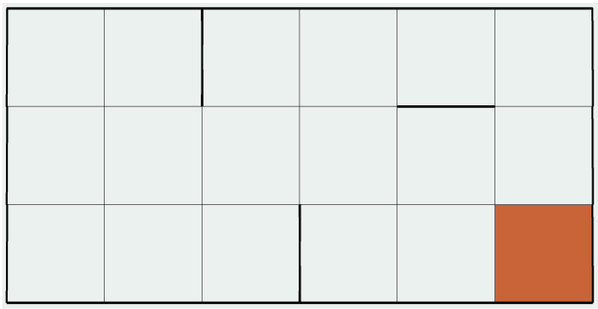
Now, consider the following theorem:

Theorem 25.1 *If we choose $\gamma_m = \frac{c}{m}$, where c is a constant, then as we visit all (x, a) pairs infinitely often, $Q \to Q^*$.*

Notice that we can think of γ_m as a learning rate. Let's see this theorem in action.

25.3.1 Demo[1]

Context. Here we consider a maze where the goal of the agent is to navigate through the maze. There are four actions to move: up, down, right, and left. The agent will get 0 reward for reaching the goal state, and will receive a negative reward for hitting a wall in the maze. A depiction of the maze is as follows:



Here we will apply the Q-learning algorithm to train an agent to solve the maze optimally. We will consider to different approaches. We will evaluate the performances of the approaches by computing a score:

$$\text{Score}(\pi) = \sigma \|V^{\pi}(s) - V^{*}(s)\|$$

where π is the policy we are evaluating for, π^* is the optimal policy, $V^{\pi}(s)$ is the true value for using π starting from state s, and $V^{*}(s) = V^{\pi}(s)$.

Approach 1: Constant Learning Rate

In this approach the learning rate is set to 0.1 throughout all the episodes. In Figure 25.1 we see the effect of this on the score as we go through episodes. We see the score converging down to a value close to 0, but oscillating quite a bit.

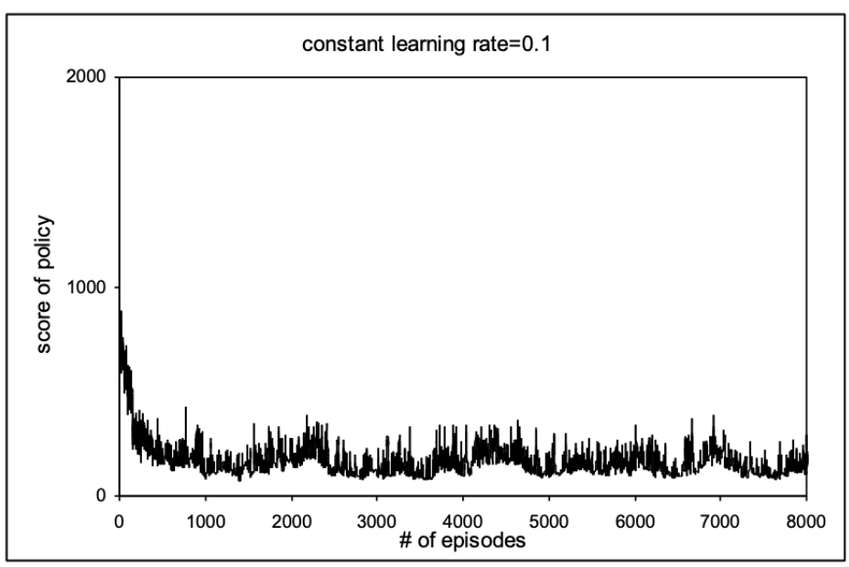


Figure 25.1: Policy score for constant learning rate

Approach 2: Decaying Learning Rate

In this approach, we decay the learning rate over the episodes. In Figure 25.2 we can see the rate of decrease of the learning rate. As we increase the number of episodes, the learning rate decays. Now, according to the theorem, we should expect our score to converge to 0,

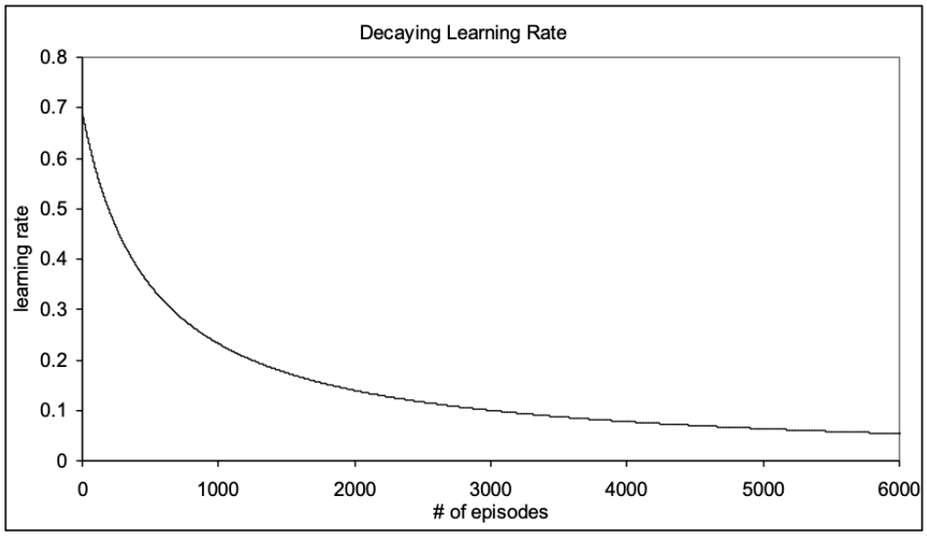


Figure 25.2: Decaying learning rate

i.e. an optimal Q function given that we run our algorithm for infinitely many state-action pairs. If we look at the resulting graph, in Figure 25.3, we see the exact behavior described in the theorem. After around the 2000th episode, we see the score converging to 0.

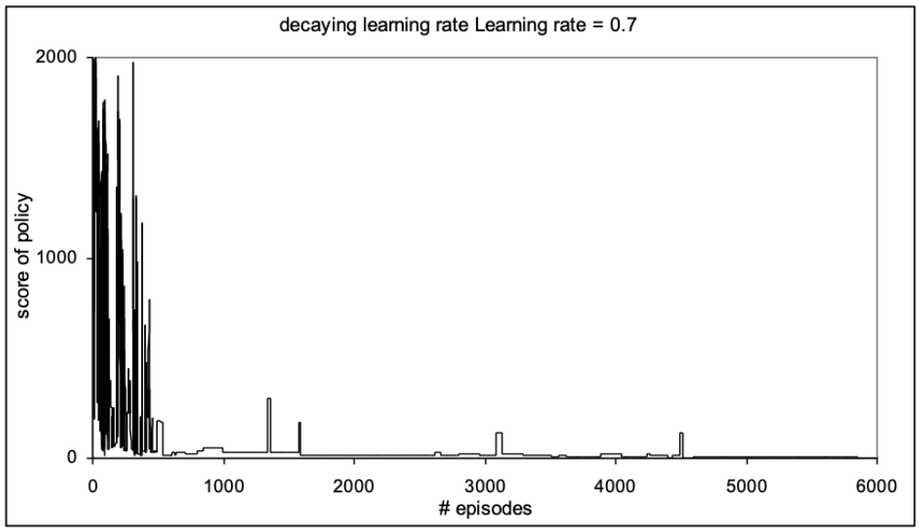


Figure 25.3: Policy score for decaying learning rate

Bibliography

[1] Vivek Mehta, Rohit Kelkar. *Simulation and Animation of Reinforcement Learning Algorithm* . 2005.