Notes on Reinforcement learning - Markov Reward Processes(MRP)

Shree Harsha Satish, Student ID: 201665390, email: sgssatis@liverpool.ac.uk

May 20, 2023

1 Introduction

Dealing with systems that have the Markov property: The probability distribution of states (and rewards) for a given time step only depends on previous time step and action. This already indicates that greedy actions will be preferable and we needn't worry about some potential missed reward later on (which we could have got by not being greedy now) even though we were greedy at each stage in the episode.

An episode is a sequence of states through which the agent travels before terminating. Also called a trajectory, it can be thought of as a sequence of samples from a traversal of the 'game'.

2 Setups that get increasingly complicated

1. A Markov chain is a tuple {S, P}. S is the set of states the agent can be in, while **P** is a model of the world i.e. the **transition probability matrix** describing how the agent can move from one state to another.

2. A Markov random process is a description of the world, outcomes and the agent moving through it. It is another tuple {S, P, R, γ}. S, P are the same as before and **R is the reward/outcome** the agent can obtain after they reach state s ϵ S and play on.

   Precisely, V is the expected sum of discounted future rewards starting in state s. We usually add a discount factor, γ, for future rewards because we wish to prioritize immediate rewards (think of website visits and showing relevant ads right away(why though?)...Anyway).

   $$V(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t\right]$$

   The expectation is over different sequence of states (episodes) the agent can take and different R_t it gets at each such random state.

   The number of time steps we look at is called the **horizon**. It could be finite as well in which case we may not need γ.

   V(s) is called the value function and as an example: V(s4) = 0.33*($R_t(s4) + R_t(s5)$ + $R_t(s4) + R_t(s3)$ + $R_t(s4) + R_t(s2)$); The agent can start in s4 and end up in 3 different states based on P. And we've found the expected value for this setup with V(s).

   There are many ways to compute V(s) if someone gives us R, P of the world around us.

   (a) Through **Monte Carlo** (μC)

   We just randomly sample from a large number of episodes and count the rewards. But we don't make use of P, R and all the markov structure in this way.

   (b) Using the Markov Structure.

   We can write what is called the **Bellman equation** if the system is Markov.
   $V(s) = R(s) + \gamma \sum_{s'} P(s'|s)V(s')$ V(s') is the one from the previous iteration.
   In this form we can compute the value function of the current state using a value function that we may have found previously. Of course, there is no such thing as an episode here because we already have the probabilities provided to us and we don't need to sample and fish for episodes. So, we just loop through the states until convergence after a random initialization.
   Are we guaranteed convergence? Why?

(c) Solve Matrix equation
We can also solve the matrix equation
$V = (I - \gamma P)^{-1} R$

3. A **Markov Decision Process (MDP)** also models our actions into the mix.

   It is also a tuple {S, A, P, R, γ} where A is the set of actions the agent can take. Also the Reward is now the Expectation over state-action sequences instead of just state sequences. This also implies the transition probability matrix is different for different actions a ϵ A.

   We also define a new parameter called a **policy**. A policy π defines what actions to take in each state. It can also be deterministic or random. These actions can still lead to non-deterministic outcomes and random moves to states depending on P and π as well.

   We define $\pi(a|s) = P(a_t = a|s_t = s)$. The policy randomness is different from the same actions leading to different states from the transition probability matrix. It is a mess!

4. A **Markov Reward Process (MRP)** is an MDP + policy.

   We define $R^\pi(s) = \sum_a \pi(a|s) R(s,a)$ as the expected reward over actions under a policy π for a given state s.

   There is also $P^\pi(s'|s) = \sum_a \pi(a|s) P(s'|s,a)$ which is the average or expected transition model from a state s under all actions in π.

   To get to the bellman equation for an MDP:
   $V_k^\pi(s) = r(s, \pi(s)) + \gamma \sum_s' P(s'|s, \pi(s)) V_{k-1}^\pi(s')$.

   Here k indexes how many times we've looped trying to find convergence. This is also called a **bellman backup policy**.

   This is how we evaluate a policy if we have it.

3 MDP Control

The next part is about choosing the best policy once we've *started* evaluating them. This is called MDP control.

In total, there are $|A|^{|S|}$ policies possible ($|A| actions for every state$) but the optimal policy is not necessarily unique.

Our goal is to find $\pi^*(s) = argmax_\pi V^\pi(s)$ i.e. that policy which maximizes V^π, the state value function.

One method is policy iteration instead of just testing all policies. To better understand this, we will define: **state-action value or Q function**: $Q^\pi(s,a)$ which states that the agent is following policy π, however it will only follow it from the next step after taking action a in state s. This action a can be part of π or not. This is where we can alter our policy in each step this way.

So, $Q^{\pi_i}(s,a) = R(s,a) + \gamma \sum_s' P(s'|s,a) V^{\pi_i}(s')$

So the policy after every step is $\pi_{i+1}(s) = argmax_a Q^{\pi_i}(s,a)$ for all s.

We know this is going to be an improvement because one of the tested actions for a is from π_i, and suppose we went back to our original policy π_i after a, we cannot be worse off than before i.e. $max_a Q^{\pi_i}(s,a) \geq V_i^\pi(s)$

But from then on, we are sticking with π_{i+1} and this is supposed to be even better because the policies are monotonically better for all states which is because of the Markov nature of the game. By monotonically better, I mean that if $V^{\pi_{i+1}}(s) \geq V^{\pi_i}(s)$ for some s, then it is better for all s ϵ S.

The proof is because of the bellman expansion which uses the Markov nature:

We know $V^{\pi_i}(s) \leq max_a Q^{\pi_i}(s,a)$

$\leq max_a R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^{\pi_i}(s')$

$\leq R(s, \pi_{i+1}(s)) + \gamma \sum_{s'} P(s'|s, \pi_{i+1}(s)) V^{\pi_i}(s')$

Here, because of the Markov nature, we can re-expand $V^{\pi_i}(s')$ after using maxQ in place:

$\leq R(s, \pi_{i+1}(s)) + \gamma \sum_{s'} P(s'|s, \pi_{i+1}(s)) max_a Q^{\pi_i}(s', a)$

$\leq R(s, \pi_{i+1}(s)) + \gamma(\sum_{s'} (P(s'|s, \pi_{i+1}(s))(max_a R(s', a) + \gamma \sum_{s''} P(s''|s', a) V^{\pi_i}(s'')))$

where s" represents subsequent states.

And so on, if the policy doesn't change once for all states s, then it will never change again.

So our strategy is to choose the greedy best option in each state. It will obviously be different if the Markov property doesn't hold true where being greedy at a current state may miss out on a larger reward in the distant future. The expansion we did is called the Bellman operator and it is a contraction operator. For $\gamma < 1$ or for finite episodes with $\gamma = 1$, the operator squeezes the vectors down from original size and we are guaranteed of the proof and obtaining the best policy.

The algorithm to update:

Algorithm 1 Dynamic Programming

```
1: Initialize:
     1. $V(s) \leftarrow$ an arbitrary state-value function
2: Repeat:
     1. $\Delta \leftarrow 0$
     2. For each $s \in S$:
        (a) $v \leftarrow V(s)$
        (b) $V(s) \leftarrow \max_a \{R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')\}$
        (c) $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
     3. Until $\Delta < \theta$ (a small positive number)
3: Output a deterministic policy, $\pi \approx \pi^*$, such that
4: $\pi(s) = \arg\max_a \{R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')\}$
```

4 Model free policy evaluation

What if we don't have access to R and P about the world? How to evaluate a policy and choose the best. We cannot use the Dynamic Programming approach from the previous section because of no knowledge of R, P.

Well there's always μC and it is an unbiased estimator of V^π and Q^π and with enough episodes the variance also $\rightarrow 0$. This also has 2 variations: first visit and every visit.

In first visit, we only update our state value function the first time we encounter the state (I don't know why, seems like last visit could also be useful?) and in every visit we update the state value function every time we encounter that state in the episode.

Monte Carlo:

Algorithm 2 First-Visit Monte Carlo Policy Evaluation

```
1: **Initialize:**
    1. $\pi \leftarrow$ policy to be evaluated
    2. $V(s) \leftarrow$ an arbitrary state-value function
    3. Returns($s$) $\leftarrow$ an empty list, for all $s \in S$
2: **Loop forever (for each episode):**
    1. **Generate an episode following $\pi$:** $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    2. $G \leftarrow 0$
    3. **Loop for each step of episode,** $t = T-1, T-2, \ldots, 0$**:**
        (a) $G \leftarrow \gamma G + R_{t+1}$
        (b) **Unless $S_t$ appears in $S_0, S_1, \ldots, S_{t-1}$:**
            i. Append $G$ to Returns($S_t$)
            ii. $V(S_t) \leftarrow$ average(Returns($S_t$))
```

Algorithm 3 Every-Visit Monte Carlo Policy Evaluation

```
1: **Initialize:**
    1. $\pi \leftarrow$ policy to be evaluated
    2. $V(s) \leftarrow$ an arbitrary state-value function
    3. Returns($s$) $\leftarrow$ an empty list, for all $s \in S$
2: **Loop forever (for each episode):**
    1. **Generate an episode following $\pi$:** $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    2. $G \leftarrow 0$
    3. **Loop for each step of episode,** $t = T-1, T-2, \ldots, 0$**:**
        (a) $G \leftarrow \gamma G + R_{t+1}$
        (b) **Append $G$ to Returns**($S_t$)
        (c) $V(S_t) \leftarrow$ average(Returns($S_t$))
```

Temporal Difference:

But we can still make use of the Markov nature even if we don't know R, P. This is by combining bootstrapping and episodic estimates of Monte Carlo. We have to wait for each episode to finish

We did $V^\pi(s) = V^\pi(s) + \alpha(G - V^\pi(s))$ and we had to wait for G to be completely calculated i.e. going through the entire episode. The trick here is to use Bellman to update even more quickly. So instead of G we will use $R_t + \gamma V^\pi(s')$ i.e. we use previous estimates and get faster convergence.

Algorithm 4 Temporal Difference (TD) Learning

```
1: Initialize:
     1. $V(s) \leftarrow$ an arbitrary state-value function
2: Loop forever (for each episode):
     1. Initialize $S$
     2. Loop (for each step of episode) until $S$ is terminal:
        (a) $A \leftarrow$ action given by $\pi$ for $S$
        (b) Take action $A$; observe reward $R$ and next state $S'$
        (c) $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
        (d) $S \leftarrow S'$
```

This is also biased and also no guarantee of the variance going to 0 but is much faster in practice.

5 Continuous Spaces and Value Function Approximation (VFA)

So far, we've dealt with spaces where the set of actions and states is discrete and easily stored on to memory. But what happens if there are too many states to enumerate/tabulate or if they're continuous?

Well, we would need a different representation that can capture the relation between many states and actions succinctly. Like a function...

And this is where DNNs and other function approximater (or approximator?) like trees etc. will be used. These approximations are parameterized which can be tuned using supervised techniques to capture the relationship by minimizing the error between our approximation and the actual function. The function could be V, Q, R or even P and we could try to estimate all of these.

Some caveats to this: If we restrict the complexity of the approximation (by say, using a simple network or just a quadratic polynomial) we limit ourselves with the different policies we can explore and the final rewards we are able to collect. But the advantage is that it is much better for memory, computation etc. compared to listing all states and tabulating.

For policy evaluation under π:

Our loss functions for this task are usually the expected MSE over different states under a single policy π: $\mathbb{E}_\pi[V^\pi(s) - V^*(s, \lambda)]$ where V* is our approximation parameterized by λ.

With this, we can reuse the old algorithms, μC **and TD** but we put in the V* training part inside our loops. Training V* requires sampling the environment to get state reward mappings and we represent the many (possibly infinite) states we have now in a vector space. This representation of the state space needs to be *good and appropriate* because it ultimately affects our final outputs.

However, we still don't have $V^\pi(s)$ to plug in to the MSE loss, if we did, why would we need to estimate it? So, we need an approximate value for the ground truth values of the function that we are trying to approximate! That is why these problems are hard I guess.

So, we just sample the environment and use what we get in place. When we walk through the game for an episode and get a reward G_t, we use that in place of V^π. Because after all, V^π is the expected value of G_t over different state sequences but we just use samples because we are trying to find the expectation. This is similar to feeding in different samples from your "training data".

We can also do MDP control with VFAs.