CS7642 Notes

Eric Huan

Summer 2022

1 Reinforcement Learning Review

1.1 Introduction

Before the dive into decision making and reinforcement learning, we first introduce the three main types of machine learning and how they can be differentiated from each other:

- Supervised Learning - Given x and y, find f such that $y = f(x)$
- Unsupervised Learning - Given x, find f that characterizes x in the form of $f(x)$
- Reinforcement Learning - Given x and z, create f to calculate optimal values $y = f(x)$

Some examples of the above are as follows:

- Supervised Learning - Regression, Splines, etc.
- Unsupervised Learning - Clustering
- Reinforcement Learning - MDPs

1.2 Markov Decision Process

One way that RL data scientists like to see the world is through something called the Markov decision process. The world quoted in this context is the space for which our problem of interest is defined. The framework for formulating such a world is defined as follows:

1. States : S - State space S containing all possible states s
2. Actions : $A(s)$ - The set of actions for a given state s
3. Model : $T(s, a, s') \sim Pr(s'|s, a)$ - The transition model from s to s' given action a
4. Reward : $R(s), R(s, a), R(s, a, s')$ - The scalar value obtained from being in a certain state s. The three listed forms are mathematically equivalent, but may be more useful to use one over the other in certain circumstances.
5. Policy : $\pi(s) \rightarrow a$ - A function that when given a state, returns an action. The policy defines a solution to the problem that is setup.

The States, Actions, Model, and Reward combined together defines/characterizes the problem. With reinforcement learning, we aim to produce an **optimal** Policy π^* that will maximize the long-term expected reward.

As clued in by its name, because we will model the process as having the Markov property, the transition probability from one state to another is only influenced by the **prior** state. Another key assumption that we will make is that the transition probabilities are **stationary**. That is, as the world evolves/progresses, the transition probability for any given pair of states do not change. This assumption may be relaxed further into the course.

1.3 More About Rewards

One aspect that creates a distinction between supervised learning and reinforcement learning is the concept of delayed rewards. Sometimes, large rewards cannot be achieved until a significant number of steps later. Being able to solve for and identify key pivotal steps that will lead to achieving the goal defines the Temporal Credit Assignment Problem. For example, consider a game of chess where say a certain opening could prove to lead to immediate loss in a hundred moves if the opponent plays optimally. Being able to identify and penalize such an opening (state + action) would prove to be very beneficial when designing an AI agent for chess so as to avoid such a pitfall.

1.4 Sequence of Rewards

As part of our formulation of the problem, we made a series of assumptions in relation to stationarity. Now we will look at a few ideas pertaining these assumptions:

1.4.1 Infinite Horizons

Through the assumption of stationarity, we implicitly assumed that we have infinite horizon. If we assign a finite horizon, then our policy function must be impacted by time since it now becomes a variable. Thus, our actions then lose stationarity in the policy function (i.e. $\pi(s)$ becomes $\pi(s,t)$).

1.4.2 Utility of Sequences

Now consider the concept of utility with regard to a sequence of states in the form of a utilty function $U(S_0, S_1, ...)$. By assumption we have that if $U(S_0, S_1, S_2, ...) > U(S_0, S'_1, S'_2, ...)$, then $U(S_1, S_2, ...) > U(S'_1, S'_2, ...)$. This is concept of the stationarity of preferences, that as states evolve our preferences would not change. A more formalized definition resulting from the above assumption is defined below:

$$U(S_0, S_1, S_2, ...) = \sum_{t=0}^{\infty} R(S_t) \tag{1}$$

However, the above definition can easily converge to infinity. To tackle this issue, we can penalize future rewards based on its distance with respect to the current state. The new formula which gives us discounted rewards is defined as follows:

$$U(S_0, S_1, S_2, ...) = \sum_{t=0}^{\infty} \alpha^t R(S_t), \quad 0 \leq \alpha < 1 \tag{2}$$

Note that this equation looks a lot like a geometric series. If we consider $R_{max} = \max_t R(S_t)$, then

$$\begin{aligned} \sum_{t=0}^{\infty} \alpha^t R(S_t) &\leq \sum_{t=0}^{\infty} \alpha^t R_{max} \\ &= \frac{R_{max}}{1-\alpha} \end{aligned}$$

Hence, we have an upper bound on the utility of the sequence. This discounting of future rewards is a method that would allow us to retain stationarity while only having to really only valuing a subset of the states.

1.5 Policies

With the above pieces finally cleared up, we can now formulate an equation for our optimal policy. The optimal policy π^* is defined as follows:

$$\pi^* = \arg\max_{\pi} \mathrm{E}\left[\sum_{t=0}^{\infty} \alpha^t R(S_t) \;\middle|\; \pi\right] \tag{3}$$

From this, we can then define the utility function with respect a particular state and policy:

$$U^{\pi}(s) = \mathrm{E}\left[\sum_{t=0}^{\infty} \alpha^t R(S_t) \;\middle|\; \pi, S_0 = s\right] \tag{4}$$

One point that I want to make clear is that $U^{\pi}(s) \neq R(s)$. $R(s)$ represents the immediate reward from achieving a state while $U^{\pi}(s)$ pertains to the long-term expected reward of a state.

Now we can combine the above equation into an equation for the optimal utility of a given state. First consider:

$$\pi^*(s) = \arg\max_{a} \sum_{s'} T(s, a, s') U(s') \tag{5}$$

Then we can derive the **Bellman Equation**:

$$U^{\pi^*}(s) = R(s) + \alpha \max_{a} \sum_{s'} T(s, a, s') U(s') \tag{6}$$

1.6 Finding Policies

Given the above Bellman equation, how do we put it into use? We do not have the utility right off the bat and the equation is not like a system of linear equations that can be solved. The approach to tackling this problem is to use an algorithm. The algorithm is called **value iteration** and goes as follows:

1. Start with initializing the utility to arbitrary values
2. Update utilities based on neighbours
3. Repeat 1 and 2 until convergence

Neighbours in the above algorithm refers to any state that it can reach, defined by the transition matrix. Now finding optimal utility values using value iteration may not be the most effective, as in fact we care more about policies than optimal utility values. We don't necessarily need to find the optimal utility values to find the optimal policies. Consider the following algorithm:

1. Initialize: Start with π_0 (Guess)
2. Evaluate: Given π_t, calculate $U_t = U^{\pi_t}$
3. Improve: $\pi_{t+1} = \arg\max_a \sum_{s'} T(s, a, s')U_t(s')$

Calculating U_t boils down to the non-max version of the Bellman Equation, defined as follows:

$$U_{t+1}(s) = R(s) + \alpha \sum_{s'} T(s, a, s')U_t(s') \tag{7}$$

Now that this equation no longer has a max function, our equation effectively becomes linear. With it, we have obtained a clear-cut method of deriving the optimal policy function.

1.7 The Bellman Equation

Consider an equation of the form:

$$U(s) = \max_a (R(s, a) + \alpha \sum_{s'} T(s, a, s')U(s')) \tag{8}$$

This is mathematically equivalent to the previously derived Bellman equation for the optimal utility. By expanding this formula and refactoring it, we can derive a similar equation:

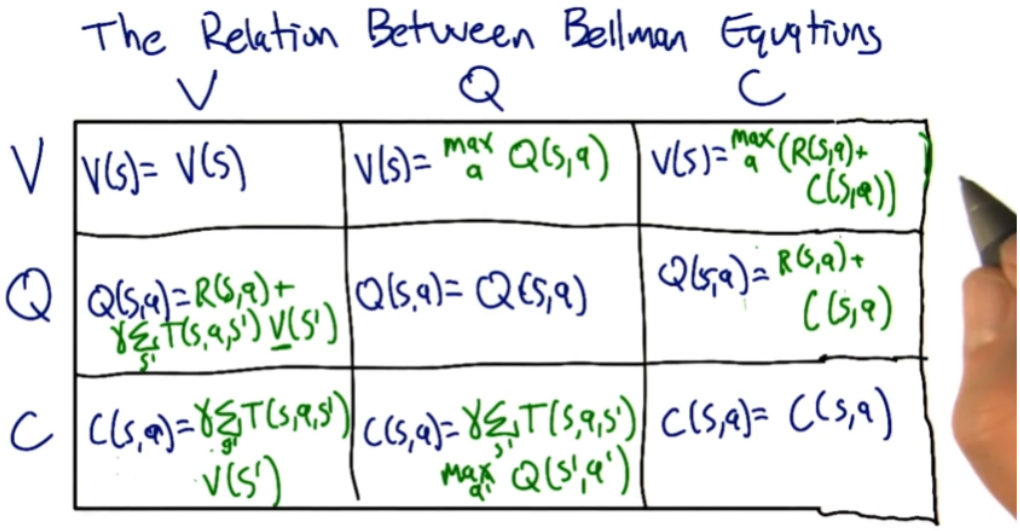
$$Q(s, a) = R(s, a) + \alpha \sum_{s'} T(s, a, s') \max_{a'} Q(s', a') \tag{9}$$

We will denote (9) as the **quality** Bellman equation while we (8) is the **value** Bellman equation. The reason why we defined the quality Bellman equation is because this form

is useful in the context of Reinforcement Learning, where calculating the expectation of $Q(s, a)$ does not necessarily need the Reward function or the transition function since we can extract the value from experienced data. Finally, we can also declare a third Bellman equation:

$$C(s, a) = \alpha \sum_{s'} T(s, a, s') \max_{a'} (R(s', a') + C(s', a') \tag{10}$$

Equation (10) is referred to as the **continuation** Bellman equation, useful sometimes for when the reward function gets really complicated since it leaves off the reward for later. These three Bellman equations can also be defined in terms of each other, according to the below diagram:



2 Reinforcement Learning Basics

2.1 Introduction

The first concept to understand when dealing with reinforcement learning is understanding that a lot of it takes place through conversations between an agent and its environment. The environment reveals itself to the agent in the form of state S. The agent then can influence the environment through a set of actions A. This updates the state and the agent receives a reward for this action, after which the next state reveals itself. This leads to a feedback loop of State, Action, Reward, as represented through the MDP.

2.2 Behavior Structures

An agent's way of interaction with the environment is called its behavior, and is equivalent to the idea of a policy. Our goal is to teach the agent behaviors such that it will interact with the environment to obtain optimal rewards. There are different ways to think about

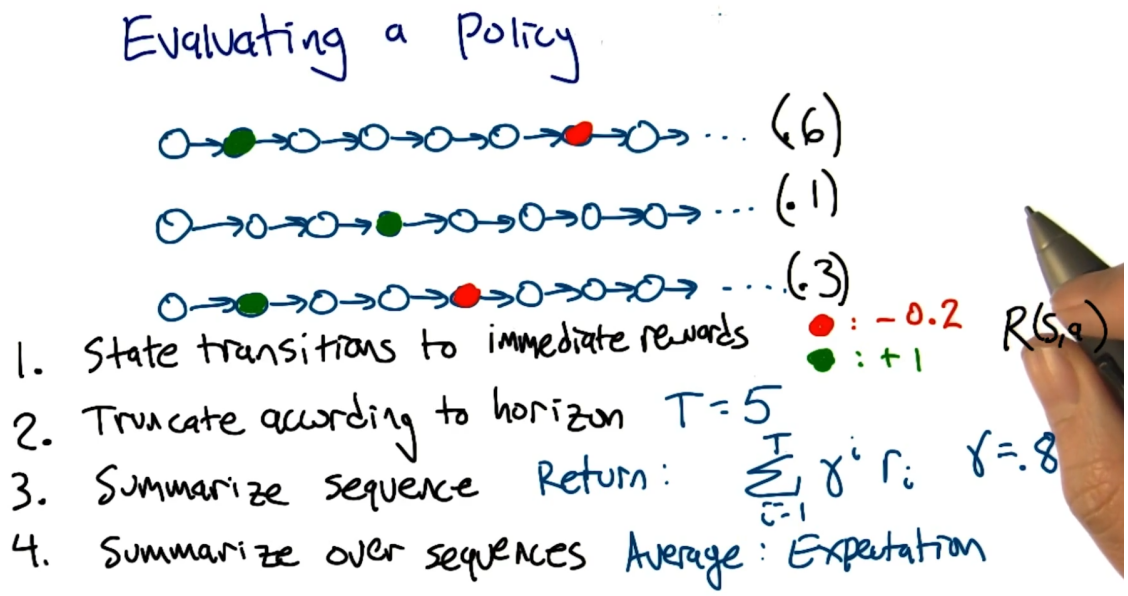
the behavior (i.e. the framework to its behavior) depending on what kind of behavior that is desired. The behavior types are the following:

- Plan - Fixed sequence of actions
- Conditional Plan - Plan with conditional statements
- Stationary Policy/Universal Plan - Mapping from state to action

Both the plan and conditional plan types of behavior are deterministic with the actions decided at the onset. We can make these two more flexible by implementing something called **dynamic replanning**, which is to follow the fixed plan until the state deviates from the plan after which a new plan is created. The universal plan is what we had seen in prior examples regarding MDPs, and offers the greatest flexibility in terms of modelling stochastic states. By comparison, it is very large (Due to stochastic considerations) but there is **always** an optimal universal plan for any MDP.

2.3 Evaluating a Policy

To evaluate a policy, there is a sequence of steps that we can follow to derive its policy value. Consider the below screenshot:



The idea is to separate each sequence of states and sum their weighted reward and taking an expectation over the weighted rewards of all sequence of states.

2.4 Evaluating a Learner

Policy functions are simply the output of learners that generate them as it observes the environment. To evaluate the learner, there are a number of metrics that we can look at:

- Value of returned policy
- Computational complexity
- Sample complexity (Data/iterations needed)

While space complexity is a domain that we can evaluate, it is often not interesting since we are typically not constrained by space.

3 Temporal Difference Learning

3.1 Introduction

Before we begin the conversation regarding Temporal Difference Learning, we need to first provide some reinforcement learning context. In particular, we have already gone over how in RL we take sequences of state, action, rewards tuples puts them through some algorithm, and generates a policy function. This RL algorithm can be classified into three main families:

- Model-based
- Value-function based (Model-free)
- Policy Search

As you will see below, the above algorithms becomes more direct as we go down the list or more supervised vice versa.

3.1.1 Model-based Algorithm

The one that is most like what has been discussed so far is the model-based algorithm. The model-based algorithm looks like

$$< s, a, r >^* \rightarrow \boxed{\text{Model Learner}} \leftrightarrow \text{T / R} \rightarrow \boxed{\text{MDP Solver}} \rightarrow Q^* \rightarrow \boxed{\text{arg max}} \rightarrow \pi \quad (11)$$

In model-based algorithms, the algorithm takes the sequence of tuples and puts it through a model learner to generate the transitions and rewards. Then, it utilizes an MDP solver to derive the optimal value function. Using the optimal value function we can take an arg max over the actions to get the optimal policy.

3.1.2 Value-function based Algorithm

The value-function based algorithm looks like

$$< s, a, r >^* \rightarrow \boxed{\text{Value Update}} \leftrightarrow Q \rightarrow \boxed{\text{arg max}} \rightarrow \pi \quad (12)$$

The difference between the above algorithm and model-based algorithms is that it skips the middle man of transitions and rewards and directly iterates back and forth between a value function updater and the value function itself.

3.1.3 Policy Search Algorithm

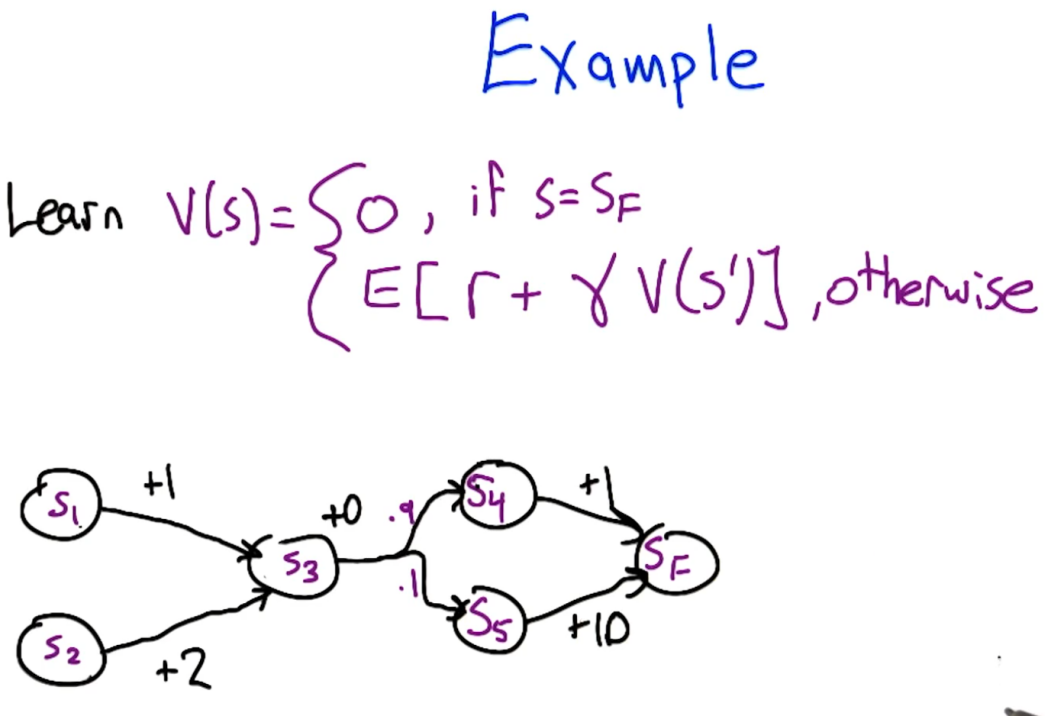
The policy search algorithm looks like

$$< s, a, r >^* \rightarrow \boxed{\text{Policy Update}} \leftrightarrow \pi \tag{13}$$

Policy search algorithms further shortens the chain such that the policy updater generates a new policy and based on this feedback iteratively generates new policies. This algorithm demonstrates a much more direct relationship between the inputs and the policies.

3.2 Temporal Difference Learning

The algorithm that we will examine to study temporal difference learning is from Sutton and is called TD-Lambda or TD(λ). This algorithm is a value-function based algorithm and is about learning to make predictions that take place over time. First let's start with a concrete example. Consider the markov chain below. We define the value function as given.



Using the formula, we can easily derive the value of each particular node. Now, let us consider how we can estimate these values with using only data. One way would be to generate samples from simulations of the Markov chain and get the average. By law of large numbers, as sample size increases, we would eventually converge upon the expected value. To demonstrate, let's consider the above Markov chain with the following simulated results.

1. $S_1 \to S_3 \to S_4 \to S_F$
2. $S_1 \to S_3 \to S_5 \to S_F$
3. $S_1 \to S_3 \to S_4 \to S_F$

Suppose we want to know what $E(S_1)$ is. The total rewards for each sample is 2, 11, and 2 respectively. By taking the average we get that our estimate is 5. With this idea in mind, we can define a formula for computing estimates incrementally (i.e. as we iterate over each sample)

$$V_T(s) = V_{T-1}(s) + \alpha_T(R_T(s) - V_{T-1}(s)) \tag{14}$$

Where T is the enumerate position of current sample, R_T is the reward for sample T, V_T is the estimate for state s with T samples, and $\alpha_T = \frac{1}{T}$. Note the diminishing coefficient α_T and that we can consider $R_T(s) - V_{T-1}(s)$ the error of the estimate (we move in the direction of the error a little bit when updating the estimate).

3.3 Properties of Learning Rates

With regard to the above formula, we can have that:

$$\lim_{T\to\infty} V_T(s) = V(s) \tag{15}$$

Where $V(s)$ is the true expectation/average of the value of state s. However, there are two conditions that must be satisfied with respect to the learning rate α_T.

- $\sum_{T=0}^{\infty} \alpha_T = \infty$
- $\sum_{T=0}^{\infty} \alpha_T^2 < \infty$

The intuition is that the learning rate must be large enough such that we can actually move to the true estimate no matter where we start but not too large that the estimates become too sensitive to noise. As a rule with regard to infinite series, for series of the form $\sum_{T=0}^{\infty} \frac{1}{T^p}$, it will converge for $p > 1$, otherwise it will diverge (go to infinity)

3.4 $TD(\lambda)$ Rule

With this, we can now formalize the $TD(\lambda)$ rule. At epoch T,

$\forall s$, $e(s) = 0$ at start of episode and $V_T(s) = V_{T-1}(s)$.
After $s_{t-1} \xrightarrow{r_t} s_t$,
$\forall s$,
$V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(s) - V_{T-1}(s_{t-1}))e(s)$
$e(s) = \lambda\gamma e(s)$

* Leave Explanation Here *

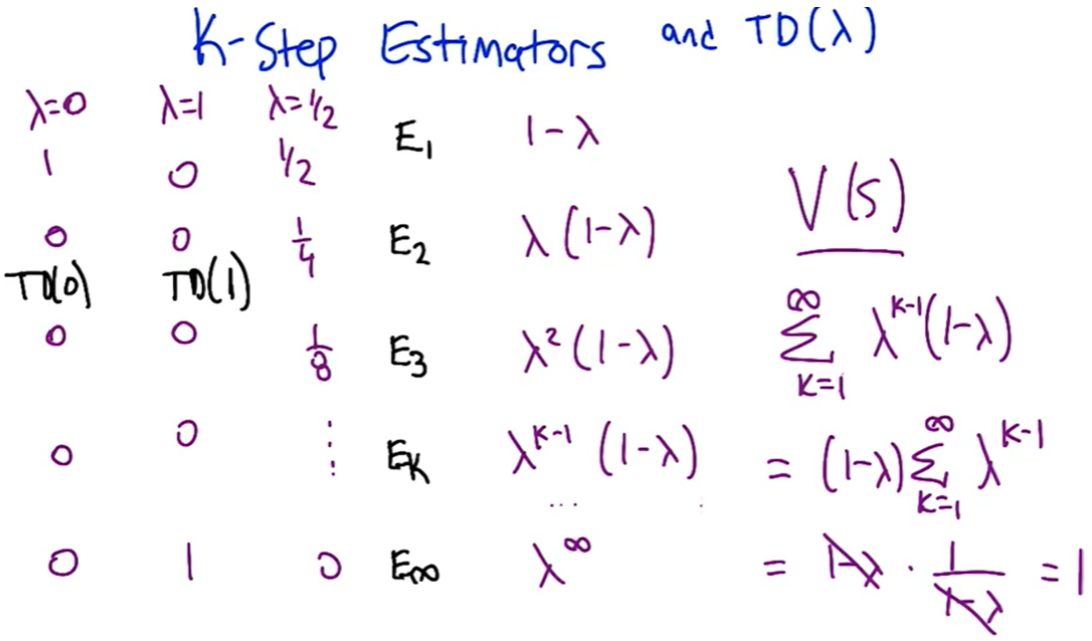
3.5 K-Step Estimators

A K-Step Estimator is defined as follows:

$$E_k : V(S_t) = V(S_t) + \alpha_t(r_{t+1} + ... + \gamma^{k-1} r_{t+k} + \gamma^k V(S_{t+k}) - V(S_t)) \tag{16}$$

Notice that for $k = 1$, we get $TD(0)$ while for $k = \infty$, we get $TD(1)$.

We can use the K-step estimator to study the $TD(\lambda)$ rule by using a weighted sum of the K-step estimators (From 1 to ∞). The weights are determined according to the below table.



By using the weighted K-step estimators, we can intuitively see the impact of using certain lambdas (i.e. how it is distributing the weighting across the estimators).