# Reinforcement Learning Notes

jongman@gmail.com

June 29, 2016

**Abstract**

The usual disclaimers: this is intended to be a personal note. I did try, but not too hard, to make it correct or understandable, so do not trust everything blindly.

# 1 Introduction

$t$ represents time, $s$ state, $a$ action. $r$ represents the reward. A policy $\pi$ is a function which returns the probablity of doing action $a$ given state $s$: $\pi(a|s)$. The world is represented by a probability distribution for next state $s'$ and the one-step reward, given a state and action: $p(s_{t+1}, r_{t+1}|a_t, s_t)$. Note you receive $r_{t+1}$ after you perform $a_t$ - mind the indices! This convention might be unintuitive, but used in Sutton. $\gamma$ is the discounting factor. The objective of reinforcement learning is to find a policy that maximizes the rewards it receives from the environment.

## 1.1 Prediction and Control Problems

Generally, there are two classes of problems in reinforcement learning. The prediction, or estimation problem asks us to evaluate the given policy - how good is it? The control problem asks us to find the optimal policy.

# 2 Value Based Reinforcement Learning

## 2.1 Value Functions

A *value function* is a central concept to reinforcement learning. It estimates how good is it for a given policy to be in the given state. Concretely, it estimates the expected discounted future rewards:

$$V_\pi(s_t) \sim \mathbb{E}_{s_i, a_i|\pi}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots\right]$$

The goal of reinforcement learning is maximizing this expected discounted future rewards. Given an optimal policy $\pi^*$, the value function $v_{\pi^*}(s)$ will satisfy a fundamental property, called the *Bellman equation*, which is second nature for anyone who understands dynamic programming. Read my book if you don't... :-p

$$v_{\pi^*}(s) = \sum_a \pi^*(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_{\pi^*}(s')\right]$$

Sometimes, just the value function is not enough. In many RL algorithms we define *action value functions*, which estimates the discounted future rewards following a given action from a given state:

$$Q_\pi(s_t, a_t) \sim \mathbb{E}_{s_i, a_i|\pi, a_t}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots\right]$$

Note that given an action value function, we can trivially come up with a policy: always choose the action that maximizes $Q_\pi$. This is how many value-based RL algorithms work: it iteratively updates $Q$, while the policy is implicitly defined by being greedy with regards to $Q$.

## 2.2 Dynamic Programming

### 2.2.1 Policy Evaluation

Given a policy *and a model of the world (i.e.* $p(s', r|a, r)$), dynamic programming can always solve both prediction and control problems. The solution to the prediction problem is an iterative DP algorithm - we can start from a vector of random values $V_\pi^0$, and iteratively obtain new values according to the Bellman equation to find the value function for the given policy. Concretely:

$$V_\pi^t(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)\left[r + \gamma V_\pi^{t-1}(s')\right]$$

This means the value function is *self consistent*. A lot of RL algorithms start with garbage values for $V$, and iteratively updates it trying to make it self consistent.

### 2.2.2 Policy Iteration

The solution to the control problem is more interesting. Let's start with a random policy $\pi_{random}$, and estimate its value function $V$ by the above algorithm. Now given a state $s_t$, consider the following scenario: we take action $a$, and follow $\pi_{random}$ (the original policy) thereafter. We have everything we need to estimate the expected reward of this scenario: we know the one step reward from the environment knowledge, and we know what will happen thereafter from the value function. Concretely, the total expected discounted reward for doing action $a$ at this state is:

$$f(s, a) = \mathbb{E}_{s', r|a}\left[r + \gamma V(s')\right] = \sum_{s', r} p(s', r|s, a)\left[r + \gamma V(s')\right]$$

Now, consider a policy $\pi'$ that always takes the action that maximizes $f$:

$$\pi'(s) = \text{argmax}_a \sum_{s', r} p(s', r|s, a)\left[r + \gamma V(s')\right]$$

This is proven to take the policy *towards* to the optimal policy, and is called *policy improvement*. Alternating these two procedure is the first way to solve the RL problem on a finite MDP: this is called *policy iteration*.

### 2.2.3 Value Iteration

Policy iteration is often too slow - as policy evaluation itself is a slow procedure, being iterative. We improve it by interleaving the evaluation step and the improvement step together; which is called *value iteration*, and is generally much more efficient. Instead of repeatedly updating $V_\pi$ following the Bellman equation until convergence, we update to the value of a greedy action. So we now have:

$$V_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V_k(s')\right]$$

instead of:

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V_k(s')\right]$$

This is also proven to converge on the optimal policy! Now we have solved the control problem too. End of reinforce learning! Easy right?

## 2.3 Monte Carlo Methods

Of course, life is not that easy. Having a model of the world is a huge assumption. Even for some trivial environments (e.g. a game of blackjack), deriving state transitions might be a daunting task. *Model-free methods* work without this model of the environment, and just learns by observing what we did, and the rewards we have received. Monte Carlo, as well as Temporal Difference methods, are examples of model free methods.

Still, all methods work with value functions. We start with a random value function. and iteratively improve it until convergence.

### 2.3.1 Monte Carlo Estimation

In Monte Carlo method, we let the policy interact with the environment without doing anything, and try to improve the policy from what we observed. These can be either simulated or real. Suppose we have observed an episode of interaction: we started in state $s_1$, chose action $a_1$, received reward $r_1$ and ended up in state $s_2$, chose action $a_2$, .. and so on. Now, how good is it to be in $s_1$? One estimate is to take the total reward we received after being in state $s_1$. So let's move $V(s)$ *toward* that value!

Formally, for each $s_t$, if this is the first time we encountered this state, do the following update:

$$V(s_t) = V(s_t) + \alpha\left(\sum_{i=t+1}^{n} r_i - V(s_t)\right)$$

where $\alpha$ is the learning rate, and $n$ is the number of states in the episode (this means we can apply MC only to episodic problems: with non-episodic problems, $n$ is not defined.)

Blackjack is a good toy example where Monte Carlo is easier to pull off than full DP. Given current hand of value 15, what's the probability of getting a 6 next? It's actually hard to answer but blackjack is absolutely trivial to simulate! In many cases, drawing and working with samples is easier than working with the distribution.

### 2.3.2 Monte Carlo Control

Monte Carlo Control is just acting greedily with the value function. However, a state value is not enough here: since we do not have the $p(s', r|a, s)$, we won't know which state we will end up given an action, nor what rewards will we receive. Thus, we learn state-action value functions instead: $q(s, a)$. The learning itself is trivial, and you can now easily maximize with respect to the action.

### 2.3.3 Exploration and $\epsilon$-greedy

The big problem of MC method is that some of the states might be abandoned; they never get visited, thus have zero expected value, so never get visited by the greedy policy, repeat. *Exploring starts method* start simulating in every state, but probably it's not so practical. Having a *soft* policy is a more realistic solution; ensure $\pi(a|s) > 0 \; \forall s, a$. An $\epsilon$-greedy policy satisfies this criteria; it chooses greedily with respect to a value function *most of the time*; with a probability $\epsilon$ it chooses an action randomly. It is proven that improving on soft policies still converge to the optimal value function.

### 2.3.4 Off-Policy Prediction

MC methods can learn from histories from a different policy: this is called off-policy learning. Say we have a policy $\pi$ and we are trying to learn from trajectory by another policy, $\mu$. The trick is to weight each sample by the *importance-sampling ratio:*

$$\rho_t^T = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{\mu(A_k|S_k)}$$

Intuitively, this makes sense: if $\rho$ is high, $\mu$ is unlikely to take this action, but $\pi$ is very likely. Since this example is important and we won't be getting many samples, it's good to weight it highly. Although, Silver notes that off-policy in MC learning is almost always too noisy in practice and thus never works. However, importance-sampling ratio is an important idea and shows up in a lot of places - for example, TRPO.

## 2.4 Temporal Difference Learning

TD learning is the hybrid between DP and MC methods. It learns from actual histories, but at the same time *bootstraps* (uses values from earlier iterations), and can be applied online (i.e. applied at every step, not necessarily at the end of the episode). This makes TD applicable to non-episodic problems. The simplest TD method, TD(0) updates its estimate of the value function after each step.

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

Quite intuitive, actually. It is effectively using observed probabilities instead of the model as in $p(s', r|a, s)$ - it is proven that TD(0) finds the correct for the *maximum-likelihood* model of the Markov process. Nice.

### 2.4.1 On-Policy TD Control: Sarsa

**S**tate **A**ction **R**eward **S**tate **A**ction: SARSA! :-p The only difference is now we estimate/update $Q$ values.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

where $a_{t+1}$ is chosen from your current policy from $s_{t+1}$.

### 2.4.2 Off-Policy: Q-learning

With $\epsilon$-greedy, we are actually performing off-policy learning all the time. Because after training is done, we don't want to use epsilon greedy anymore, we just want to act greedily! Q-learning exactly does this:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right]$$

### 2.4.3 MC vs TD

As MC uses realized rewards for all future steps, it has a high variance. On the other hand, it has zero bias and good convergence property, even when we use function approximation. On the other hand, TD has low variance and high bias. It also might not converge well with function approximations, and is more sensitive to initial values than TD.

A fundamental difference of TD is that it actively exploits Markov property: so it is usually more efficient when the environment is more Markovian.

## 2.5 Eligibility Trace and TD($\lambda$)

What is 0 in TD(0)? Is there some TD($x$) where $x > 0$? Of course it is! Generalized $n$-step TD prediction updates use the following $n$ step return:

$$G_t^{t+n} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_n + \gamma^n V_t \left(s_{t+n}\right)$$

However, we can never say a specific $n$ is better than all other values. So we take averages of different returns - in particular, TD($\lambda$) takes the EMA of future returns, called the $\lambda$-return:

$$L_t = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{t+n}$$

We now realize that MC and TD(0) are both special cases of this algorithm, with $\lambda = 1$ and 0 respectively (although, TD(1) is only *roughly* equivalent to an every-visit MC). As can be expected, there is usually a "sweet spot" between the two algorithms, and in some problems TD($\lambda$) with the correct $\lambda$ can result in superior performance.

### 2.5.1 Backward Implementation

Implementing $\lambda$-return naively requires us to simulate infinite histories before making any progress, which doesn't make too much sense in non-episodic problems. A backward-looking implementation uses *eligibility traces*; which represents how much a given state pair is *responsible* for current state of the agent. The eligibility trace $E_t\left(s\right)$ is increased whenever we visit state $s$, and exponentially decays every time when the state changes. There are different schemes of updating $E_t\left(s\right)$, but they are not probably terribly important. :-p

Silver introduces recent results that shows the equivalence of forward and backward implementations.

# 3 Policy-based Reinforcement Learning

## 3.1 What and Why?

So far, all policies were derived from value functions or action value functions. Policy-based RL methods (alternatively, actor-only methods) work with the policy directly. Typically, policy is parameterized by a parameter vector $\theta$. So $\pi\left(s, a, \theta\right) = \mathbf{P}\left[a_{t+1} = a | s_t = s, \theta\right]$. Softmax policy is a good example.

The main reason we do policy-based RL is dealing with continuous action spaces. With continuous action space, value-based RL agent need to solve an optimization problem at every step $\arg\max_a Q\left(s, a\right)$,

which might not be trivial. With policy based learning, you just generate a random action given the distribution. Better convergence property is another reason. Silver also brings up examples where stochastic policies are inherently superior than deterministic policies (e.g. adversarial setup). What's the downside? Naive policy-based methods typically converge to a local maximum, and can be inefficient and high variance than naive value-based RL.

## 3.2 Policy Gradients

Policy based learning works by defining an objective that is expressed in terms of the policy parameter $J(\theta)$, and running numerical optimization to maximize it. The typical objective functions include the start value ($V^{\pi_\theta}(s_1)$), the average value, or average reward per time step. This optimization problem can be solved in a myriad of ways - we typically resort to gradient based methods, which are vastly more efficient. SGD and variants go towards the local optimum by following the gradient of the objective function w.r.t. $\theta$

$$\Delta\theta = \alpha \nabla_\theta J(\theta)$$

where $\nabla_\theta J(\theta) = \left\{ \frac{\partial J(\theta)}{\partial \theta_i} \right\}$ is called the *policy gradient* and $\alpha$ is the learning rate.

### 3.2.1 Estimating Policy Gradient

The objective function is an expected value of some form of reward (discounted or average), and we cannot calculate it directly without the distribution of the states. Silver introduces two methods to estimate the gradient. One is finite difference - this is brute. Another method is the likelihood ratio method. This exploits the following identity

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} = \pi_\theta(s,a) \nabla \log \pi_\theta(s,a) = \mathbb{E}[\nabla \log \pi_\theta(s,a)]$$

The gradient is the expected value of this quantity! Now, we can sample the quantity$\nabla \log \pi_\theta(s,a)$ as an unbiased estimator of the gradient. This is so significant it has a separate name; it is called the score function.

### 3.2.2 Policy Gradient Theorem

Consider a non-episodic (never-ending) environment. Then with $d(s) = \sum_{t=0}^{\infty} \gamma^t p(x_k = x | d_0, \pi)$ as the discounted state distribution under policy $\pi_\theta$, we can write $J$ as:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] = \sum_s d(s) \sum_a \pi_\theta(s,a) \cdot Q^{\pi_\theta}(s,a)$$

Using the likelihood ratio method, the gradient is:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \sum_s d(s) \sum_a \nabla \pi_\theta(s,a) \cdot Q^{\pi_\theta}(s,a) \\
&= \sum_s d(s) \sum_a \pi_\theta(s,a) \nabla \log \pi_\theta(s,a) \cdot Q^{\pi_\theta}(s,a) \\
&= \mathbb{E}\left[\nabla \log \pi_\theta(s,a) \cdot Q^{\pi_\theta}(s,a)\right]
\end{aligned}$$

This is called the policy gradient theorem. The most popular actor-only algorithm, REINFORCE, is a Monte-Carlo policy gradient which uses exactly this. $Q^{\pi_\theta}(s,a)$ is, of course, estimated using actual episodes.

### 3.3 Actor-Critic Methods

Main downfall of vanilla policy-based ("actor-only") RL is the high variance of estimated gradients: $Q^{\pi_\theta}(s,a)$ above has a very high variance, due to the same reason as Monte Carlo. Actor-critic RL is a hybrid between value-based and policy-based RL. It introduces a "critic" which gives you a low-variance estimate of $Q^{\pi_\theta}(s,a)$. Of course, we know exactly how to solve this - TD or MC learning. So in AC methods, we do TD learning alongside the actor learning: and instead of $Q^{\pi_\theta}$, we use the estimated value function.

## 4 Planning

### 4.1 Interweaving Planning/Acting/Learning

Planning is getting a policy from a model. Learning means translating experience to a model. Dyna-Q is an algorithm which interleaves these three aspects of RL into a single algorithm. It is based off of Q-learning; it maintains a crude model of the world, and at each stage runs backup for some states chosen randomly. This seems kind of arbitrary.

### 4.2 When The Model Is Wrong

Typically, when the environment becomes hostile to the policy, RL methods will quickly realize and change its behavior. It is hard when environment changes for the better - unless the algorithis is explicitly encouraged for exploration, the new action might not be picked up, even with $\epsilon$-greedy policies.

### 4.3 Choosing States to Back Up

Vanilla Dyna-Q randomly chooses states and update their value functions. This is suboptimal because of several reasons. 1. The states that may never come up in real life can get chosen. 2. The value function for a state might only change when it precedes a state with changed value. Several alternatives are chosen; trajectory sampling (only update along experience), prioritized sweeping (do a backwards search with priority being changes in state value), etc.

### 4.4 Heuristic Search

Do a classic state-space search over multiple levels, and take value function at the leaves of the searched portion of the tree. TD-Backgammon uses this.

## 5 Function Approximation

### 5.1 Stochastic Gradient Descent

Tabular value function is infeasible when the state space is continuous. Then, it is a natural approach to take each backup as an input-output pair which gets fed into an function approximation algorithm. SGD is a natural fit for these kind of algorithms, because it allows for piecemeal updates to the approximation.

### 5.2 Eligibility Traces with Function Approximation

Previously we introduced eligibility trace as a mapping from state to a real number. This is impossible anymore, as there are infinite states. How do we implement eligibility trace? Say the function approximation is represented by a weight vector $\mathbf{w}_t$. Then, eligibility trace is a vector, with one element for each component of $\mathbf{w}_t$. The update is:

$$\mathbf{e}_t = \gamma\lambda\mathbf{e}_{t-1} + \nabla\hat{v}\left(S_t, \mathbf{w}_t\right)$$

which is the decaying sum of recent gradients to the value function. This makes some intuitive sense; if a dimension had a high gradient on recent states, it can be updated to greatly change the value of the states; it is somehow *responsible* for them.

### 5.3 Convergence Properties

Perhaps not surprisingly, we can't always get convergence guarantees when using function approximation. For linear approximators and on-policy scenarios, things look good, but outside of that (nonlinear approximators, or off-policy scenarios), all bets are off. Silver notes that in practice linear approximator does work well even without theoretical guarantees. However, for nonlinear approximator controls, special care must be taken to prevent the coefficients from blowing up. Gradient TD is one such algorithm.

## 6 Continuous Control

### 6.1 Deterministic Policy Gradient

## 7 Recent Improvements

### 7.1 Replay Buffer and Fixed Q Targets

Silver mentions the bag of tricks (ooh, juicy) Atari-solving DQN employed. There are two big tricks: experience replay and fixed Q-targets.

Experience replay avoids feeding transitions in the order they happen to avoid feeding correlated inputs. This is super related to financial markets as well. Concretely, it stores all transition $(s_t, a_t, r_{t+1}, s_{t+1})$ instead of applying SGD update and throwing them away. After each step, randomly pick a small batch (10s~) of such updates from the library of transitions and apply them.

Fixed Q-targets mean avoiding a moving target; TD learning uses $\max_a Q\left(s_{t+1}, a\right)$ to update $Q\left(s_t, a_t\right)$ - with nonlinear approximators, the update can cause the target to move, and this can spiral out of control. Instead, you keep an old policy towards which we converge to. We periodically swap out this *reference* policy.

### 7.2 Double Q-Learning

Q-learning has a tendency to overestimate action values, because we take $\max_a \mathbb{E}\left[Q\left(s_{t+1}, a\right)\right]$ instead of $\mathbb{E}\left[\max_a Q\left(s_{t+1}, a\right)\right]$. Double Q-learning tries to fix this by having two $Q$ functions: whenever we observe a transition $(s_t, a_t, r_{t+1}, s_{t+1})$, we update only one of the two functions. And we use the value from the other function to calculate the update target.

## 7.3 Dueling Network Architecture

## 7.4 Trust Region Policy Optimization

State of the art policy gradient method. Less data efficient than regular value-based learning methods.