Introduction to Reinforcement Learning

Based on Silver [2015]

Lars Quaedvlieg
Larsquaedvlieg@outlook.com

October 2020

Contents

Chapter 1

Intro to Reinforcement Learning

1.1 The Reinforcement Learning Problem

What makes reinforcement learning different from other machine learning paradigms?

- There is no supervisor, only a **reward** signal
- Feedback is delayed, not instantaneous
- Time really matters (sequential, non i.i.d data)
- Agent's actions affect the subsequent data it receives

Rewards are scalar feedback signals. Reinforcement Learning is based on the **Reward Hypothesis**, meaning all goals can be described by the maximization of expected cumulative reward. This can be hard, since actions can have long-term consequences and reward can be delayed.

A state is **Markov**, if and only if it holds the **Markov Property**, meaning $\mathbb{P}(S_t + 1|S_t) = \mathbb{P}(S_t + 1|S_1, ..., S_t)$. This means the probability of future states solely depends on the current state, and not on any previous states. I.e. the history is a sufficient statistic of the future.

Let S_t^a be the state of the agent at any time t and S_t^e be the state of the environment on any time t. If the environment is **fully observable**, then $S_t^a = S_t^e$. This means that the Markov property holds, so formally it is a Markov Decision Process.

However, when the environment is **partially observable**, the agent indirectly observes the environment. Now, $S_t^a \neq S_t^e$. Formally, this is called a partially observable Markov decision process (POMDP). The agent must construct it's own state representation S_t^a. For example:

- Complete history: $S_t^a = H_t$
- Beliefs of environment state: $S_t^a = (\mathbb{P}[S_t^e = s^1], ..., \mathbb{P}[S_t^e = s^n])$
- Recurrent Neural Network: $S_t^a = \sigma(S_{t-1}^a W_s + O_t W_o))$

1.2 Components of an RL Agent

An RL agent may include one or more of these components:

- **Policy**: agent's behaviour function
- **Value function**: how good is each state and/or action
- **Model**: agent's representation of the environment

A **policy** describes the agent's behavior. It maps states to actions. You can have deterministic ($a = \pi(s)$) and stochastic policies ($\pi(a|s) = \mathbb{P}(A_t = a|S_t = s)$). Often, π is used to denote a policy.

A **value function** is a prediction of future reward of a given state. You can use it to determine if a state is good or bad. This means you can use it to select actions. It can be computed by $v_\pi(s) = \mathbb{E}_\pi(G_t|S_t = s)$, where G_t is the **return** (or total reward). The return is defined as $G_t = R_1+\gamma R_2+\gamma^2 R_3+... = \sum_{i=t+1}^{\infty} \gamma^{i-t-1} R_i$ for some $\gamma \in [0, 1]$. This gamma is the **discount factor**, and it influences how much the future impacts return. This is useful, since it is not known if the representation of the environment is perfect. If it is not, it is not good to let the future influence the return as much as more local states. So, it is discounted.

Finally, a **model** predicts what the environment will do next. We let $P^a_{ss'} = \mathbb{P}(S_t + 1 = s'|S_t = s, A_t = a)$ and $R^a_s = \mathbb{P}(R_t + 1|S_t = s, A_t = a)$. P (**Transition model**) is the transition probability to a next state given an action, while R is the probability of obtaining a certain reward when taking an action in some state.

Category	Properties
Value based	No Policy (implicit), Value function
Policy based	Policy, No Value function
Actor Critic	Policy, Value function
Model Free	No Model
Model based	Model

Figure 1.1: Types of RL agents

RL Agents can be categorized into the categories that are listed in 1.1. These can require different approaches that will be discussed later.

There are two fundamental problems in **sequential decision making**.

- **Reinforcement Learning**
  - The environment is initially unknown
  - The agent interacts with the environment
  - The agent improves its policy
- **Planning** (e.g. deliberation, reasoning, introspection, pondering, thought, search)
  - A model of the environment is known
  - The agent performs computations with its model (without any external interaction)
  - The agent improves its policy

It is important for an agent to make a trade-off between exploration and exploitation as well. Depending on the choice in this trade-off, agents will be more or less flexible and may or may not find better actions to perform.

- **Exploration** finds more information about the environment
- **Exploitation** exploits known information to maximize reward

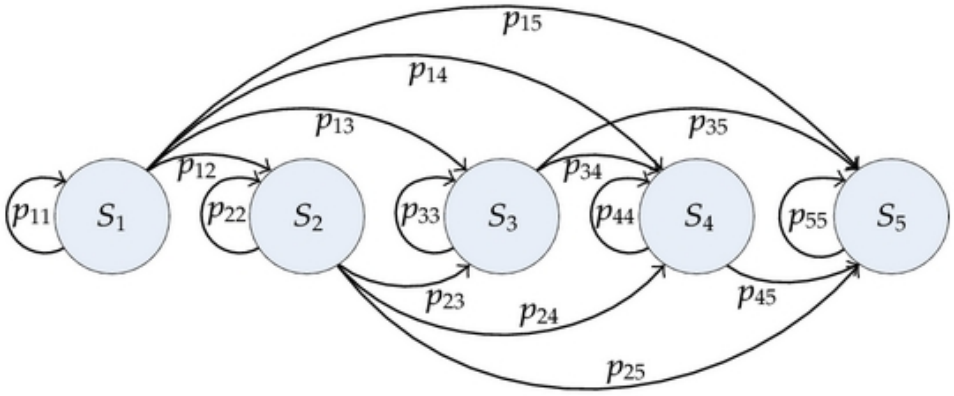
Finally, it is possible to differentiate between prediction and control. **Prediction** is about evaluating the future given a certain policy, while **control** is about finding the best policy to optimize the future.

Chapter 2

Markov Decision Processes

As said in chapter 1, fully observable environments in Reinforcement Learning have the Markov property. This means the environment can be represented by a **Markov Decision Process** (MDP). This means that the current state completely characterizes the process. MDP's are very important in RL, since they can represent almost every problem.

2.1 From Markov Chains to MDP's



$$P = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} & p_{15} \\ 0 & p_{22} & p_{23} & p_{24} & p_{25} \\ 0 & 0 & p_{33} & p_{34} & p_{35} \\ 0 & 0 & 0 & p_{44} & p_{45} \end{bmatrix} \text{ where } p_{ij} \geq 0 \text{ and } \sum_j p_{ij} = 1$$

Figure 2.1: Example of a Markov Chain

Here, P is the **Transition Matrix**. It contains all transition probabilities of the model. A **Markov Chain** (or Markov Process), is a tuple (S, P), where S is a set of states and P is the transition matrix. A sequence $S_0, ..., S_n$ is called an **episode**.

From this Markov Chain, we can create a **Markov Reward Process**. This is defined as an extension of the tuple, with the elements (S, P, R, γ). S and P mean the same thing as before. However, now we also have a reward function (recall that $R_s = \mathbb{E}[R_{t+1}|S_t = s]$) and a discount factor $\gamma \in [0, 1]$. How to then turn 2.1 into a MRP? Simply add reward values to each state in the Markov Chain. The rewards of each state can then be computed with the reward function.

Now let's introduce the **Bellman Equation** for MRP's. It is possible to rewrite the formula of the value function in the following way.

$$\begin{aligned} v(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2}, \gamma R_{t+3}, ...)|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1})|S_t = s] \\ &(= R_s + \gamma \sum_{s' \in S} P_{ss'} v(s')) \end{aligned} \tag{2.1}$$

Figure 2.2: Bellman equation of the value function in a MRP

Now it is possible to see that the value of a state is dependent on the immediate reward, and the return of neighboring states. This can then be computed using the transition matrix ($P_{ss'} > 0 \iff$ you can go from s to s'). The bellman equation can then be written in matrix form, namely $v = R + \gamma P v$. This can be rewritten as $v = (I - \gamma P)^{-1} R$. This equation can be solved in $O(n^3)$, since a matrix inverse is necessary. For large MRP's, there are several iterative methods to solve this equation. For example,

- Dynamic programming
- Monte-Carlo evaluation
- Temporal-Difference learning

These methods will be explained in later chapters.

2.2 Markov Decision Processes

Finally, let's talk about the **Markov Decision Process**. This is an extension of the MRP, with properties (S, A, P, R, γ). It introduces a new set, defining the actions that can be taken. The transition matrix P^a and R^a now also conditionally depend on the action.

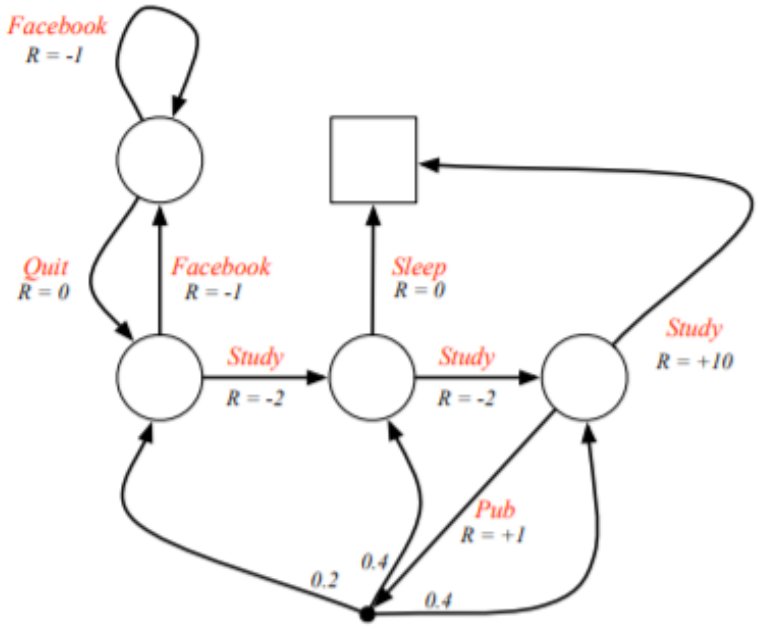


Figure 2.3: Example of a Markov Decision Process

2.3 is an example of a Markov Decision Process. In the image you see that the rewards are now dependent on the actions, as previously explained. Now it is possible to talk about the behavior of an agent. In chapter 1, policies were mentioned. These determine the behavior of agents (like which action to take in which state).

Given an MDP and a policy π. The state sequence $S_0, S_1, ...$ is a Markov Process (S, P^π) and the state and reward sequence $S_0, R_1, S_1, R_2, ...$ is a Markov Reward Process $(S, P^\pi, R^\pi, \gamma)$ where $P^\pi_{s,s'} = \sum_{a \in A} \pi(a|s) P^a_{s,s'}$ and $R^\pi_s = \sum_{a \in A} \pi(a|s) R^a_s$ (Law of Total Probability).

Similarly to the state-value function $v_\pi(s)$, let's now define the **action-value** function $q_\pi(s, a) = \mathbb{E}_\pi[G_t | S = s, A = a]$. This is the expected return

starting from state s, taking action a, and then following policy π. In a RL problem, this is what we care about. The goal is to select the best action in a given state, in order to maximize the reward. If we have access to the q-values, it is the solution to the problem.

After deriving the Bellman equations for the MDP's, we end up with the following formulas.

$$\begin{aligned} v(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v(S_{t+1})|S_t = s] \\ &(= \sum_{a\in A} \pi(a|s) q_\pi(s,a)) \\ &(= \sum_{a\in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a v_\pi(s') \right)) \end{aligned} \tag{2.2}$$

Figure 2.4: Bellman equation of the state value function in a MDP

$$\begin{aligned} q(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma q(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \\ &(= R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a v_\pi(s')) \\ &(= R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a \sum_{a'\in A} \pi(a'|s') q_\pi(s',a')) \end{aligned} \tag{2.3}$$

Figure 2.5: Bellman equation of the action value function in a MDP

The second-last form of formula (2.5) can be intuitively thought of as the following. Imagine we are taking a certain action in a state. Then, the environment might bring us into different successor states even if we take the action. So, we have to sum over all these possible successor states and use the law of total probability again to compute the action-values.

Now we can talk about optimality. The **optimal state-value** and **optimal action-value** functions are defined as $v_*(s) = \max_\pi v_\pi(s)$ and $q_*(s,a) = \max_\pi q_\pi(s,a)$. We can say $\pi \geq \pi'$ if $\forall s : v_\pi(s) \geq v_{\pi'}(s)$. The optimal value function specifies the best possible performance in the MDP. An MDP is "solved" when we know the optimal value. For any MDP, there always exist

an optimal policy that is better or equal to all policies. This policy achieves both the optimal value function and the optimal action-value function.

An **optimal policy** can be found by maximizing over $q_*(s, a)$.

$$\pi_*(a|s) = \begin{cases} 1, & \text{if a} = \text{argmax}_{a \in A}\, q_*(s, a) \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

Figure 2.6: Equation for an optimal policy

Thee optimal value functions are recursively related to the **Bellman Optimality Equations**. $v_*(s) = \max_a q_*(s, a)$ and $q_*(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')$. Solving for these equations is difficult, since the Bellman Optimality Equation is non-linear. Generally, there is no closed form solution. However, there exist iterative methods to approximate the solution. Example of this are

- Value Iteration
- Policy Iteration
- Q-learning
- Sarsa

2.3 Advanced: Extensions to MDPs

2.3.1 Infinite and continuous MDPs

- Countably infinite state and/or action spaces
  - Straightforward
- Continuous state and/or action spaces
  - Closed form for linear quadratic model (LQR)
- Continuous time
  - Requires partial differential equations
  - Hamilton-Jacobi-Bellman (HJB) equation
  - Limiting case of Bellman equation as time-step approaches 0

2.3.2 Partially observable MDPs

A Partially Observable Markov Decision Process is an MDP with hidden states. It is a **hidden Markov model** with actions. A POMDP is a tuple $(S, A, O, P, R, Z, \gamma)$. We now also have O representing a finite state of observations and Z being an observation function $Z^a_{s'o} = \mathbb{P}[O_{t+1} = o | S_{t+1} = s', A_t = a]$.

Let's define a **history** H_t being a sequence of actions, observations and rewards ($H_t = A_0, O_1, R_1, ..., A_{t-1}, O_t, R_t$). A **belief state** $b(h)$ is a probability distribution over states conditioned on the history h. $b(h) = (\mathbb{P}[S_t = s^1 | H_t = h], ..., \mathbb{P}[S_t = s^n | H_t = h])$.The history and belief state both satisfy the Markov property. A POMDP can be reduced to an (infinite) history tree and an (infinite) belief state tree.

Chapter 3

Planning - Dynamic Programming

Dynamic Programming (DP) breaks one problem down into smaller sub-problems in order to solve them. Then, you can combine the solutions of the sub-problem to answer the problem.

DP is used for planning within an MDP. This means the method assumes there is full knowledge of the MDP. It is technically not Reinforcement Learning yet, since we don't discover an initially unknown environment. It can be used for both *prediction* (the input is the MDP and policy, returns value function) and *control* (input only the MDP, returns optimal policy and value function).

3.1 Prediction

Iterative policy evaluation can be used to perform prediction in an MDP (evaluate how good a policy is). On a high level, it works by backing up the Bellman expectation from the states in which rewards are observed. The pseudocode below shows how the algorithm works.

Algorithm 1 Iterative policy evaluation

Require: the MDP, policy π
  $i \Leftarrow 0$, $v_0(s) \Leftarrow 0$ for each state s
  while not converged **do**
    for each state s **do**
      $v_{i+1}(s) \Leftarrow \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_i(s') \right)$
    end for
    $i \Leftarrow i + 1$
  end while
  return v_i

When this algorithm converges, we have obtained the value function v_π.

3.2 Control

Now that we have a way to evaluate how good a certain policy is, we can start improving it. There are two main ways of performing control using DP. The first method that will be discusses is called **Policy Iteration**.

At each iteration, this method consists of two components: **policy evaluation** and **policy improvement**. The whole algorithm works in the following way

Algorithm 2 Iterative policy evaluation

Require: the MDP, policy π
  while not converged **do**
    $v^\pi \Leftarrow$ iterative policy evaluation with π
    $\pi' \Leftarrow greedy(v_\pi)$
    $\pi = \pi'$
  end while
  return v_π, π

The image below shows how this algorithm works on a higher level.

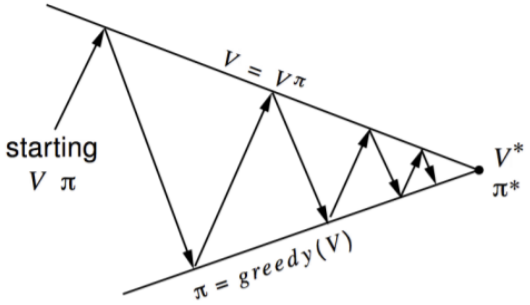


Figure 3.1: Policy Iteration

Why does acting greedily with respect to the obtained v_π improve the policy? There is a relatively simple proof for this. Say we choose a deterministic policy $\pi(s)$. Acting greedily would mean we create a new policy $\pi'(s) = \arg\max_{a \in A} q_\pi(s, a)$. This means that $q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$. So we know $q_\pi(s, \pi'(s)) \geq v_\pi(s)$. Therefore, $v_{\pi'}(s) \geq v_\pi(s)$.

This means if the improvement stops, we must have satisfied the Bellman optimality equation.

Policy iteration can be generalized. Before we performed policy evaluation until we obtained v_π. However, this is not necessary. The next DP control algorithm, **Value Iteration**, is a special variant of policy iteration, updating the policy after every step (so not after it converges to v_π). You can basically use **any** policy evaluation and policy improvement algorithm to perform policy iteration.

Lets now start talking about Value Iteration. This method works because of the *Principle of Optimality*, which states the following: An optimal policy can be subdivided into two components

- An optimal first action A_*
- An optimal policy from successor state S'

This means that if we know the solution to $v_*(s')$, we can find $v_*(s)$ by performing the following one-step lookahead. This is possible due to the Bellman optimality equations.

$$v^*(s) = \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \right)$$

The intuition is that you can start with the final rewards and work your way backwards.

Algorithm 3 Value iteration

Require: the MDP
  $i \Leftarrow 0$, $v_0(s) \Leftarrow 0$ for each state s
  while not converged **do**
    for each state s **do**
      $v_{i+1}(s) \Leftarrow \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_i(s') \right)$
    end for
    $i \Leftarrow i + 1$
  end while
  $v_* \Leftarrow v_i, \pi_*(s) \Leftarrow \arg\max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')$
  return v_*, π_*

Observe that the policy is extracted by acting greedily with respect to the computed q-values for any state. For any intermediate value functions, this

might not be true. These in-between value function might not correspond to any policy.

The previously discusses algorithms all share a runtime complexity of $O(n^2m)$ per iteration when computing v, n being the number of states and m the number of actions. However, the same process can be applied to compute the q-values. This would be $O(n^2m^2)$ per iteration.

3.3 Extensions to DP

So far all described DP methods are *synchronous*. This means all states are updated in parallel. However, these algorithms can also be implemented in an *asynchronous* manner. Since it uses more updated information, it generally converges a lot faster than the synchronous variant. It is also guaranteed to converge as long as all states are continued to be visited. The following three ideas are simple ideas for asynchronous DP with Value Iteration.

- In-place DP

  Instead of using different value functions v_i at each iteration, we only use one value function. $v_{i+1}(s) = \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_i(s') \right)$ will then become $v(s) = \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s') \right)$.

- Prioritized sweeping

  Instead of iterating over all states in one order, it is possible to use the magnitude of Bellman error to guide the selection of the next state to evaluate. This bellman error can be expressed as

  $$\left| \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s') \right) - v(s) \right|$$

  The idea is to backup the state with the largest remaining Bellman error and update the Bellman error of the affected states after. This requires knowledge of the reverse dynamics of the MDP, since we are working backwards. It can be very simply implemented using a priority queue.

- Real-time DP

  The idea here is to only update states that are relevant to the agent. The agent's experience can guide the state selection. So, after each

time step, we observe S_t, A_t and R_{t+1}. We back-up the state S_t by $v(S_t) = \max_{a \in A} \left(R^a_{S_t} + \gamma \sum_{s' \in S} P^a_{S_t s'} v(s') \right)$.

The DP approach that was discussed this chapter is already good. It is effective for problems containing millions of states. However, since it uses full-width backups, each successor state and action is considered. For large problems this causes DP to suffer from the curse of dimensionality; the number of states grow exponentially with the number of state variables. As proposed in further chapters, this problem is approached using sample backups.

Chapter 4

Model-Free Prediction

Last chapter was about planning by dynamic programming. This was an approach to solving a known MDP. In many cases, however, we do not have access to the full MDP. In this case, we are talking about unknown MDPs. In this chapter, estimating the value function of an unknown MDP will be discussed.

4.1 Monte-Carlo RL

Sampling methods like Monte-Carlo methods are about learning from episodes of experience. This means they don't need to have any knowledge of the dynamics of the MDP (transitions and rewards). This makes them model-free methods. The point of MC methods is to run until the end of each episode. This means it can only work if episodes always terminate. It backtracks the experience that was generated during the episode to estimate the value function. They are based on one simple idea: **value = mean return**.

Walking through episodes of a problem using policy π yields the information $S_1, A_1, R_1, ..., S_k \sim \pi$. The return is the total discounted reward, computed by $G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$. Then, $v_\pi(s) = \mathbb{E}_\pi [G_t | S_t = s]$. Instead of this *expected* return, MC policy evaluation uses an *empirical mean* return.

Algorithm 4 Iteration in Monte-Carlo methods

Require: policy π
  $\forall s : V(s) \Leftarrow 0, N(s) \Leftarrow 0, S(s) \Leftarrow 0$
  for $t \Leftarrow 0, ..., T$ **do**
    $A_t, R_{t+1}, S_{t+1} \sim \pi$
    $N(S_t) \Leftarrow N(S_t) + 1$
    $S(S_t) \Leftarrow S(S_t) + G_t$
    $V(S_t) \Leftarrow \frac{S(S_t)}{N(S_t)}$
  end for
  return V

By the law of large numbers

$$\lim_{N(s)\Rightarrow\infty} V(s) = v_\pi(s)$$

This algorithm uses an *every-visit* approach, meaning it will execute an iteration every time S_t has been encountered in an episode. There is a second approach called *first-visit*. This works by only doing the iteration for state S_t at most once per episode, when it is encountered for the first time.

Imagine that at time t you observe S_t, but the same state is observed at time $t+2$. Then, $S_t = S_{t+2}$. The first-visit approach will not do the iteration at $t+2$, but the every visit will.

There is also a way of computing the mean incrementally.

$$\begin{aligned} V_t(S) &= \frac{S_t(S)}{N_t(S)} \\ &= \frac{1}{N_t(S)}(G_t + S_{t-1}(S)) \\ &= \frac{1}{N_t(S)}(G_t + (N_t(S)-1)V_{t-1}(S)) \\ &= V_{t-1}(S) + \frac{1}{N_t(S)}(G_t - V_{t-1}(S)) \\ V(S_t) &= V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)) \end{aligned}$$

This way you can see how it works intuitively. The current perception of the value is updated using a difference in the observed return and current perception of the value, weighted by the number of times a state is visited.

However, this means that we will never completely forget old experience, since we are using this moving average. In non-stationary problems, it may be useful to track the running mean to forget old episodes. This can be achieved by for example replacing $\frac{1}{N(S_t)}$ by a constant α. The new equation will become $V(S_t) = V(S_t) + \alpha(G_t - V(S_t))$.

4.2 Temporal-Difference RL

One problem with Monte-Carlo methods is that we need episodes to terminate. TD-Learning on the other hand, does not need this. Like MC, they learn directly from episodes of experience. They are also model-free. The

key difference is that TD-Learning uses a concept called **bootstrapping** to learn from incomplete episodes. They basically update towards a guess of G_t.

The simplest TD-algorithm is called TD(0). It attempts to update the value towards the estimated return $R_{t+1} + \gamma V(S_{t+1})$, which it uses as a guess for the expected return G_t. This value is often referred to as the TD target. Updating the value function will then become $V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) = V(S_t) + \alpha\delta_t$ where δ_t is referred to as the TD error.

4.3 Comparison of methods

One big difference between TD and MC is that TD can learn *online* after every step and without the final outcome. MC must wait until the episode is over, not being able to learn from incomplete/non-terminating sequences.

We know that the return $G_t = R_{t+1} + \gamma V_\pi(S_{t+1})$ is an *unbiased* estimate of $v_\pi(S_t)$. TD uses a *biased* estimate of the value function, since it uses $R_{t+1} + \gamma V(S_{t+1})$ to bootstrap. The positive thing about this is that the TD target has a much lower variance than the return. This is because the return depends on many random actions/transitions/rewards, while the TD target only depends on one random action/transition/reward.

Both MC and TD converge to the true value function for a policy as we get infinite experience. However, for finite batches of experience that are repeatedly sampled, they will not produce the same results.

Monte-Carlo methods converge to the solution to the best fit of the observed return. It has the objective function

$$\min \sum_{k=1}^{K} \sum_{t=1}^{T_k} (G_t^k - V(s_t^k))^2$$

Temporal-Difference methods on the other hand converge to the solution of the *maximum likelihood Markov model*. It constructs and solves (given π) the MDP $(S, A, \hat{P}, \hat{R}, \gamma)$ that best fits the data.

$$\hat{P_{ss'}^a} = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} 1(s_t^k, a_t^k, s_{t+1}^k = s, a, s')$$

$$\hat{R_s^a} = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} 1(s_t^k, a_t^k = s, a) r_t^k$$

From this you can see TD exploits the Markov property. Standard MC does not do this. For this reason, TD is usually more effective in Markov environments and MC in non-Markov environments.

4.4 TD(λ)

As we saw before, TD(0) looks one step into the future, by calculating $R_{t+1} + \gamma V(S_{t+1})$. Can we look more steps into the future? The answer is yes. For example, a 2-step TD target can be calculated as $R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})$. It is interesting to notice that when we look ∞ steps into the future, we converge to the Monte-Carlo method!

In general, the n-step return $G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$. To perform n-step TD, you just use $G_t^{(n)}$ as your estimation of G_t.

We can even average different n-step returns to create a more sound estimate of G_t. For example, $\frac{1}{2}G^{(2)} + \frac{1}{2}G^{(4)}$. The idea that rises is whether we can efficiently combine information from all time-steps.

This is actually what TD(λ) does. It defines the λ-return G_t^λ that combines all n-step returns. The method uses weighting $(1-\lambda)\lambda^{n-1}$. The reason for this weighting is that it has nice convergence properties that allow us to calculate this return in the same time-complexity as TD(0).

We obtain $G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$. **Forward view TD($\lambda$)** will then be $V(S_t) = V(S_t) + \alpha(G_t^\lambda - V(S_t))$.

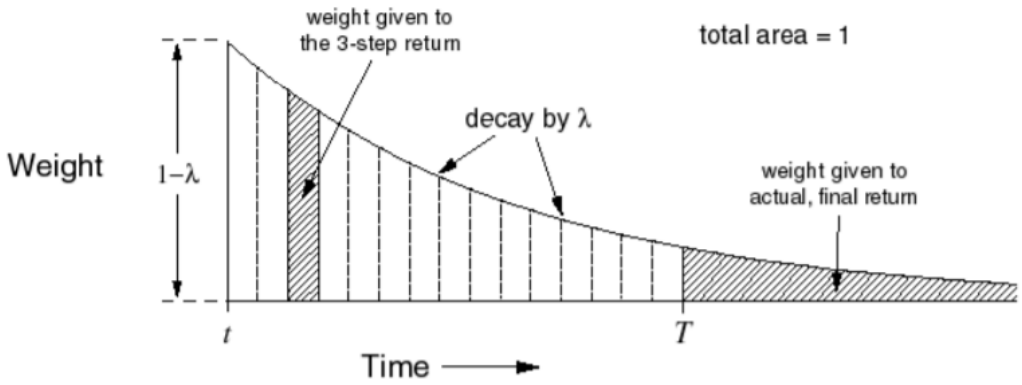


Figure 4.1: Influence of weighting on different returns $G^{(n)}$

As seen in 4.1, more recent states get a higher weighting in TD(λ). The weight decays as we look more into the future.

There is a problem with this forward-view algorithm. Just like MC, the estimated return can only be computed once the episode terminates. Luckily, there is another view for TD(λ) called the **Backward View**. This algorithm allows for updating online, on every step from incomplete sequences.

To understand Backward View TD, lets introduce a concept called **Eligibility traces**. Imagine you're in a situation where you just got electrocuted. This happened right before you heard a bell right. Before that bell even rang, light flashed three times. What do you think caused the shock? The light flashes or the bell?

The idea of this is that this is controlled by **Frequency heuristics** and **Recency heuristics**. We can combine both of these heuristics to form Eligibility Traces. Let $E_t(S)$ be the eligibility trace of state s at time t. Initially, $E_0(s) = 0$. We create the recursive relationship $E_t(s) = \gamma\lambda E_{t-1}(s) + 1(S_t = s)$. Observe that at time t, if we're in state s, a value of 1 will be added to $E_t(s)$. However, for all other previously visited states, their trace only gets decayed. This corresponds to the intuition of figure 4.1.

For Backward View TD(λ), the idea is the following

- Keep an eligibility trace for all states s
- Update value V(s) for every state s
- In proportion to TD-error δ_t and $E_t(s)$, we say $V(s) = V(s) + \alpha\delta_t E_t(s)$

Intuitively, this means you are constantly decaying and updating the values of previously observed states. When $\lambda = 0$, we end up with TD(0). When $\lambda = 1$, the credit is deferred until the end of the episode. This means that *over the course of an episode*, the total update for TD(1) is the same as the total update for every-visit MC.

Algorithm 5 Iteration of TD(λ)

Require: policy π
  $\forall s : V(s) \Leftarrow 0, E(s) = 0$
  for $t \Leftarrow 0, ..., T$ **do**
    $A_t, R_{t+1}, S_{t+1} \sim \pi$
    $\delta_t \Leftarrow R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$
    $E(S_t) \Leftarrow E(S_t) + 1$
    for all unique previously occurred s **do**
      $V(s) \Leftarrow V(s) + \alpha \delta_t E(s)$
      $E(s) \Leftarrow \gamma \lambda E(s)$
    end for
  end for
  return V

Chapter 5

Model-Free Control

This chapter several methods for Model-Free control will be discussed. Algorithms like Monte-Carlo and Temporal-Difference learning are often used for prediction, but how can these concepts be used for control? There are two different ways of learning when sampling, these are

- **On-policy** learning

  The goal is to learn about policy π from experience that is being sampled by π

- **Off-policy** learning

  The goal is to learn about policy π from experience that is being sampled by a policy μ, $\mu \neq \pi$.

From generalized policy iteration, any valid *policy evaluation* algorithm can be followed by any valid *policy improvement* algorithm and iterated in order to find the optimal policy. The first question that should come up in your mind is "Can we just use the algorithms from the previous chapter for the policy evaluation step?". Initially, there are two problems with this.

1. Greedy policy improvement over $V(s)$ requires a model of the MDP, since $\pi'(s) = \arg\max_{a \in A} R_s^a + P_{ss'}^a V(s')$. We do not have access to the rewards and the transition probabilities.

   We know this is equal to $\pi'(s) = \arg\max_{a \in A} Q(s, a)$. So learning the q-value function instead of the value function will be *model-free*.

2. Being greedy using sampling methods does not ensure we cover the entire state space.

   Exploration vs. exploitation is a whole problem on its own in Reinforcement Learning, that will be discussed in a later chapter. For now we can consider the easiest way to ensure continual exploration. Instead of being greedy, lets be ϵ-**greedy**. The policy becomes the following

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon, & a^* = \arg\max_{a \in A} Q(s, a) \\ \epsilon/m, & \text{otherwise.} \end{cases} \tag{5.1}$$

This means that there is a $1-\epsilon$ probability to choose the greedy action and an ϵ probability to choose any other random action, wtih m being the number of actions.

Now, all that is left to be done is prove any ϵ-greedy policy π' with respect to q_π actually is an improvement to π. The proof is rather simple:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_{a \in A} \pi'(a|s) q_\pi(s,a) \\ &= \epsilon/m \sum_{a \in A} q_\pi(s,a) + (1-\epsilon) \max_{a \in A} q_\pi(s,a) \\ &\geq \epsilon/m \sum_{a \in A} q_\pi(s,a) + (1-\epsilon) \sum_{a \in A} \frac{\pi(a|s) - \epsilon/m}{1-\epsilon} q_\pi(s,a) \\ &= \sum_{a \in A} \pi(a|s) q_\pi(s,a) = v_\pi(s) \end{aligned}$$

Then, from policy improvement theorem, $v_{\pi'}(s) \geq v_\pi(s)$.

We have now tackled the problems that prevented us to use the idea of generalized policy iteration. The methods we have previously seen can now be applied to solve the problem.

5.1 On-Policy methods

Last chapter Monte-Carlo and TD-Learning were discussed for policy evaluation. This section will show how to apply these algorithms for control tasks.

5.1.1 Monte-Carlo Control

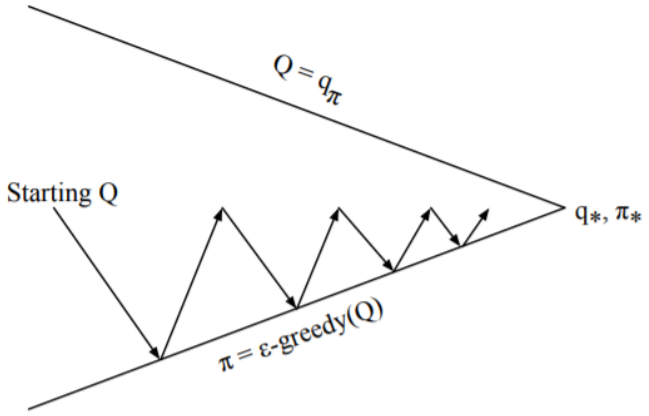


Figure 5.1: Monte-Carlo Control algorithm

The idea of 5.1 is similar to policy iteration. In order to speed up convergence, it is not necessary to evaluate the policy until q_π is obtained. A better approach is to perform Monte-Carlo policy evaluation 8 until the end of the episode, and then perform an ϵ-greedy policy improvement step with respect to the computed q-values.

There is a theorem called *Greedy in the Limit with Infinite Exploration* (GLIE), which ensures you converge to the optimal policy (which is always greedy). The following two rules must apply

1. $\lim_{k \Rightarrow \infty} N_k(s,a) = \infty$
2. $\lim_{k \Rightarrow \infty} \pi_k(a|s) = 1(a = \arg\max_{a' \in A} Q_k(s,a'))$

Here, the ϵ-greedy policy would reduce to greedy when there is infinite experience. For example, ϵ-greedy is GLIE if ϵ reduces to zero at the rate $\epsilon_k = \frac{1}{k}$. Lets construct an algorithm based on this knowledge.

Algorithm 6 One iteration of GLIE Monte-Carlo Control

for $t \Leftarrow 0, ..., T$ **do**
  $A_t, R_{t+1}, S_{t+1} \sim \pi$
  $N(S_t, A_t) \Leftarrow N(S_t, A_t) + 1$
  $Q(S_t, A_t) \Leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$
end for
$\epsilon \Leftarrow 1/k, \pi \Leftarrow \epsilon$-greedy$(Q)$
return Q, π

5.1.2 TD-Control (SARSA)

TD-Learning has several advantages over MC such as a lower variance, online updating, and being able to learn from incomplete sequences. It would be a natural idea to use TD instead of MC in the control loop that was previously presented. This method is referred to as **SARSA**. This yields the following algorithm

Algorithm 7 One iteration of SARSA

Require: $\pi \Leftarrow \epsilon$-greedy(Q)
  for $t \Leftarrow 0, ..., T$ **do**
    $A_t, R_{t+1}, S_{t+1}, A_{t+1} \sim \pi$
    $Q(S_t, A_t) \Leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$
  end for
  $\epsilon \Leftarrow 1/k, \pi \Leftarrow \epsilon$-greedy$(Q)$
  return Q, π

SARSA converges to the optimal action-value function under the following conditions:

1. GLIE sequence of policies $\pi_t(a|s)$
2. **Robbins-Monro** sequence of step-sizes α_t ($\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$)

In practice however, this is only very rarely taken into account (like with a constraint learning rate α). It does not seem to pose that much of a threat.

Just like n-step TD and TD(λ), there exists **n-step SARSA** and **SARSA(λ)**. SARSA(λ) also has the same forward- and backward-view algorithms. These are the exact same concepts. The only difference is SARSA is for action-values. For this reason, I will just give you the formulas for it. The intuition is the same as in the previous chapter.

1. n-step SARSA

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha \left[q_t^{(n)} - Q(S_t, A_t)\right]$$
$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n})$$

2. (Forward View) SARSA(λ)

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha \left[q_t^{\lambda} - Q(S_t, A_t)\right]$$
$$q_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}$$

Algorithm 8 Iteration of (backward view) SARSA(λ)

Require: $\pi \Leftarrow \epsilon$-greedy(Q), Q
  $E(s,a) = 0$
  Initialize S_0, A_0
  for $t \Leftarrow 0, ..., T$ **do**
    $R_{t+1}, S_{t+1}, A_{t+1} \sim \pi$
    $\delta_t \Leftarrow R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$
    $E(S_t, A_t) \Leftarrow E(S_t, A_t) + 1$
    for all unique previously occurred (s,a) pairs **do**
      $Q(s,a) \Leftarrow Q(s,a) + \alpha\delta_t E(s,a)$
      $E(s,a) \Leftarrow \gamma\lambda E(s,a)$
    end for
  end for
  return V

The eligibility traces for the backward view algorithm are, just like the value function, now taken into account for all state-action pairs.

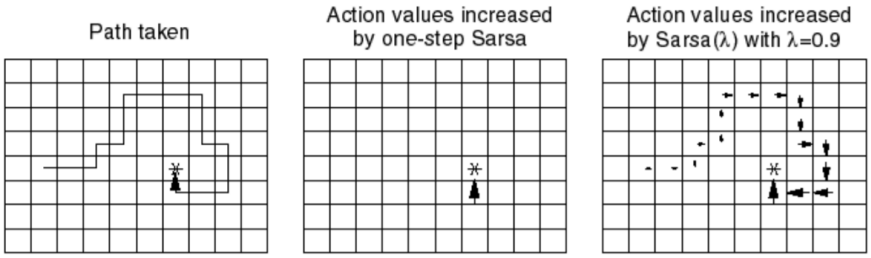


Figure 5.2: The role of λ in SARSA(λ)

In 5.2, you can see the role of the lambda parameter. When receiving reward (in the image only at the end), the combination of the recency and visitation heuristic made into the eligibility trace makes sure there is a decay in propagating the reward backwards. This is all one iteration of the algorithm.

5.2 Off-Policy methods

The goal of off-policy learning is to evaluate the target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s,a)$. All of this happens while following a different policy $\mu(a|s)$. This is referred to as the *behaviour policy*. The most well-known

use of this is learning about the optimal policy while following a *exploratory* policy.

Importance Sampling is a way of estimating the expectation of a different distribution. The underlying idea is the following

$$\mathbb{E}_{X\sim P}\left[f(X)\right] = \sum f(X)P(x) \quad = \sum f(X)\frac{P(X)}{Q(X)}Q(X) = \mathbb{E}_{X\sim Q}\left[f(X)\frac{P(X)}{Q(X)}\right]$$

From this derivation, notice that it is possible to estimate the expectation of sampling distribution P while sampling from Q, using a simple division $\frac{P(X)}{Q(X)}$.

5.2.1 Importance Sampling for Off-Policy MC / TD

The idea of this simple division can be used in the Monte-Carlo return and TD target. This means we could follow policy μ, while learning policy π by just dividing π by μ for every time-step (as seen in the proof).

$$G_t^{\pi/\mu} = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}\frac{\pi(A_t+1|S_t+1)}{\mu(A_t+1|S_t+1)}...\frac{\pi(A_T|S_T)}{\mu(A_T|S_T)}G_t$$

The value will then be updated towards the corrected return: $V_{k+1}(S_t) = V_k(S_t)+\alpha\left(G_t^{\pi/\mu} - V_k(S_t)\right)$. There are several downsides to using this method. The first one is that it can not be used if $\mu = 0, \pi \neq 0$. More importantly, it can dramatically increase variance. This is of course something to avoid when possible.

For TD, the update rule becomes

$$V_{k+1}(S_t) = V_k(S_t) + \alpha\left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}(R_{t+1} + \gamma V_k(S_{t+1})) - V_k(S_t)\right)$$

The benefit to this is that it will have a much lower variance than the MC approach. It means the policies only need to be similar over a single step instead of the whole episode chain.

5.2.2 Q-Learning

Now lets consider off-policy learning for action-values $Q(s,a)$. The idea of **Q-Learning** is to choose the next action using a behaviour policy $A_{t+1} \sim \mu(.|S_t)$, but an alternative successor action $A' \sim \pi(.|S_t)$ is also considered. $Q(S_t, A_t)$ will then be updated towards the value of the alternative action. $Q_{k+1}(S_t, A_t) = Q_k(S_t, A_t) + \alpha\left(R_{t+1} + \gamma Q_k(S_{t+1}, A') - Q_k(S_t, A_t)\right)$.

Now allow both policies to be able to improve. $\pi(S_{t+1}) = \arg\max_{a'} Q(S_{t+1}, a')$ and $\mu = \epsilon$-greedy(Q). In this case, the Q-learning target simplifies to the following

$$\begin{aligned} & R_{t+1} + \gamma Q(S_{t+1}, A') \\ =& R_{t+1} + \gamma Q(S_{t+1}, \arg\max_{a'} Q(S_t, a')) \\ =& R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') \end{aligned} \tag{5.2}$$

The algorithm of Q-Learning is then

Algorithm 9 One iteration of Q-Learning

Require: $\pi \Leftarrow \epsilon$-greedy$(Q), Q$
  for $t \Leftarrow 0, ..., T$ **do**
    $A_t, R_{t+1}, S_{t+1}, A_{t+1} \sim \pi$
    $Q(S_t, A_t) \Leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$
  end for
  return Q, π

Bibliography

David Silver. Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/, 2015.