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

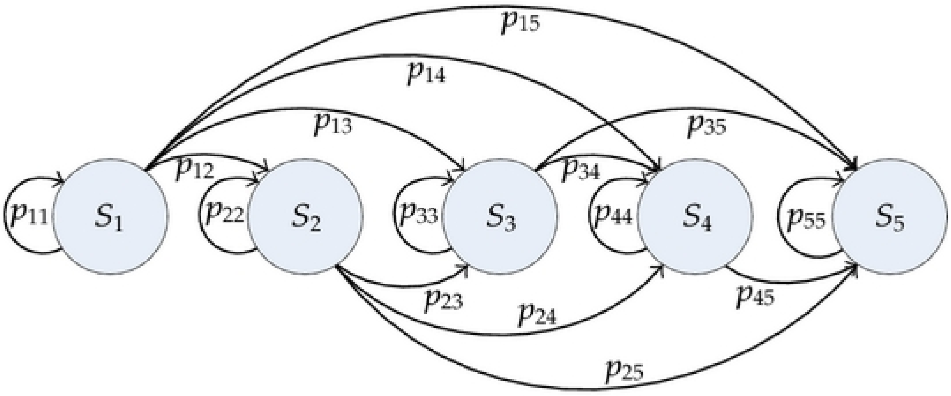
Finally, it is possible to differentiate between prediction and control. **Prediction** is about evaluating the future given a certain policy, while **control** is about finding the best policy to optimize the future.

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.

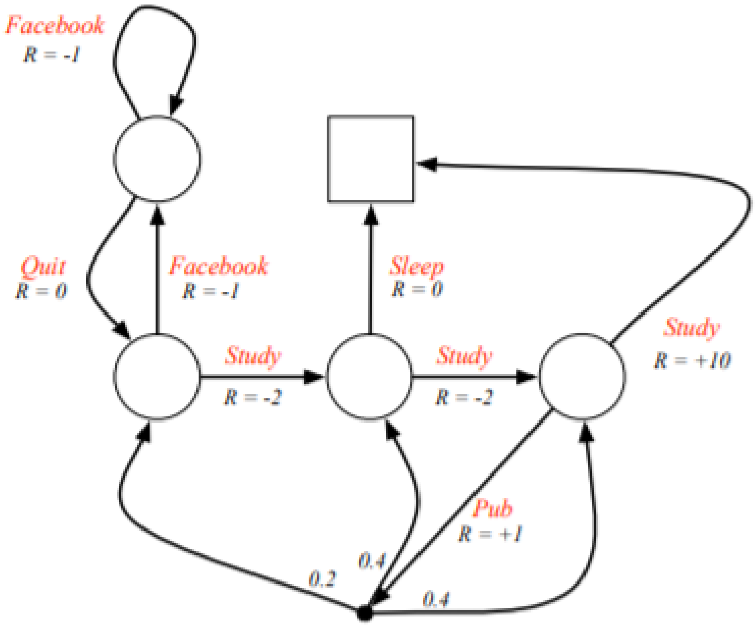


Figure 2.3: Example of a Markov Decision Process

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 $q_*(s, a)$.

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

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 $\pi$
  $i \Leftarrow 0$, $v_0(s) \Leftarrow 0$ 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 $\pi$
  while not converged do
    $v^\pi \Leftarrow$ iterative policy evaluation with $\pi$
    $\pi' \Leftarrow greedy(v_\pi)$
    $\pi = \pi'$
  end while
  return $v_\pi$, $\pi$
```

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

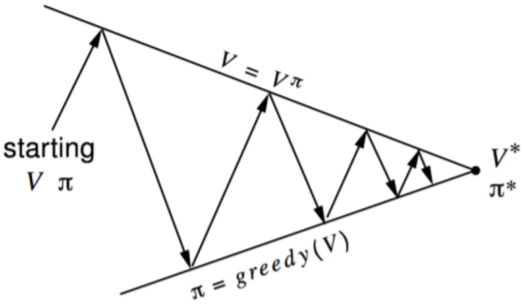


Figure 3.1: Policy Iteration

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 $v_*, \pi_*$
```

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, $v_\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s\right]$. Instead of this *expected* return, MC policy evaluation uses an *empirical mean* return.

Algorithm 4 Iteration in Monte-Carlo methods

```
Require: policy $\pi$
  $\forall s : V(s) \Leftarrow 0, N(s) \Leftarrow 0, S(s) \Leftarrow 0$
  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.

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

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

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 $V(S_t) = V(S_t) + \alpha(G_t^\lambda - V(S_t))$.

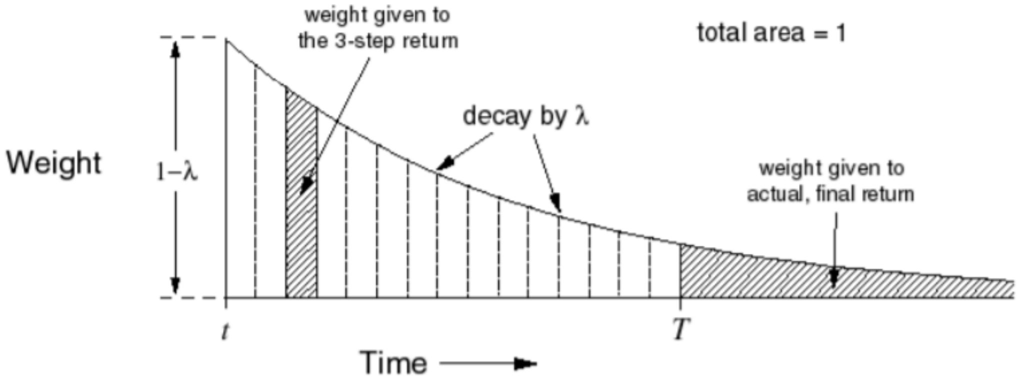

weight given to the 3-step return
total area = 1
decay by λ
Weight
1−λ
weight given to actual, final return
t
T
Time

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 $\pi$
  $\forall s : V(s) \Leftarrow 0, E(s) = 0$
  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

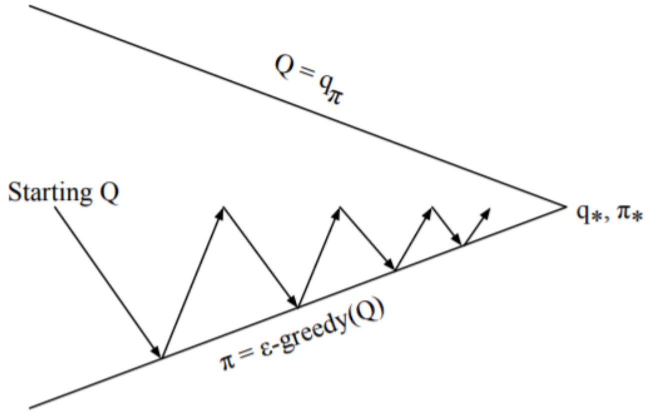


Figure 5.1: Monte-Carlo Control algorithm

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 $Q, \pi$
```

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 $Q, \pi$
```

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.

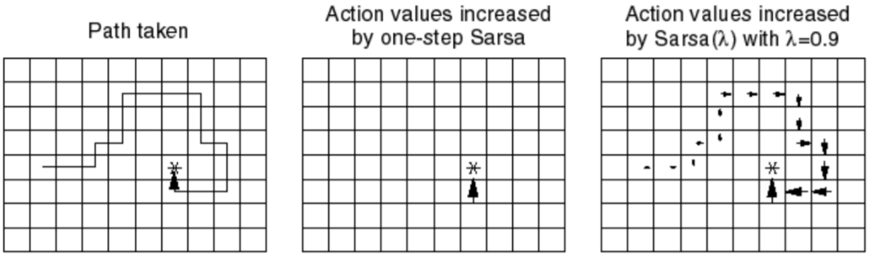

Path taken
Action values increased by one-step Sarsa
Action values increased by Sarsa(λ) with λ=0.9
*
*
*

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 $Q, \pi$
```

Chapter 6

Value Function Approximation

Before this lecture, the value function was always defined per state. But what if the problems are really big? The game of Go for example has 10^{170} states. Some problems even have a *continuous* state space. How is it possible to scale up the previously discussed model-free methods for prediction and control to handle these situations?

The whole time, (action-)value functions were represented using a *lookup table*, where each state had a corresponding value. The problem with large MDP's is that it is too slow to learn the value for each state individually, or even impossible to store all states or actions in memory.

The solution that will be proposed is to estimate the value function using **function approximation**. This means we use parameters w estimate the value functions $\hat{v}(s, w) \approx v_\pi(s)$ or $\hat{q}(s, a, w) \approx q_\pi(s, a)$. There is a really nice benefit to this. After learning w (using for example MC or TD learning), it will also be able to *generalize* to unseen states.

This chapter will focus on *differentiable* function approximators, since they can be updated using gradient-based optimization. Furthermore, the method must be able to handle *non-stationary* and *non-iid* data.

Two different methods of learning will be discussed, incremental methods and batch methods.

6.1 Incremental Methods

This book assumes you are already familiar with the concept of **Gradient Descent**. If you are not, here is a quick explanation. If $J(w)$ is a differentiable function of parameter vector w. The idea is to update the parameters in the direction of the gradient. We obtain $w = w - \alpha \nabla_w J(w)$ where α is a step-size parameter.

The goal of *Value-function approximation* using Stochastic Gradient Descent (SGD), is to find the parameter vector w in order to minimize the mean-squared error between the approximate value $\hat{v}(s, w)$ and $v_\pi(s)$. The cost function to minimize is then $J(w) = \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, w))^2\right]$.

Gradient descent converges to a *local* minimum. SGD will do this by sampling the gradient based on one single sample. The expected update is then equal to the full gradient update.

6.1.1 Prediction algorithms

The state of the problem can be represented by a **feature vector** $x(S)$, where each element is a feature that described the state (preferably independently). The value function is equal to $\hat{v}(S, w) = x(S)^{\intercal} w$.

Table-lookup is actually a special case of linear function approximation. If a state is a one-hot-encoding of all states, then the dot product with parameter vector w is just $v(S_i) = w_i$.

Since it is impossible to use the true value function v_π in the cost function, lets substitute in the targets instead. These are G_t for MC and G_t^λ for TD(λ). The following are more detailed descriptions for these methods

- **MC**

  Return G_t is an unbiased, noisy sample of $v_\pi(S_t)$. For that reason, it is possible to apply supervised learning to the data $(S_1, G_1), (S_2, G_2), ..., (S_T, G_T)$. $\Delta w = \alpha(G_t - \hat{v}(S_t, w))\nabla_w \hat{v}(S_t, w) = \alpha(G_t - \hat{v}(S_t, w))x(S_t)$. Monte-carlo evaluation converges to a local optimum, even when using non-linear function approximation.

- **TD(0)**

  Return $R_{t+1} + \gamma\hat{v}(S_{t+1}, w)$ is a biased sample of $v_\pi(S_t)$. It is possible to apply supervised learning to the data $(S_1, R_2 + \gamma\hat{v}(S_2, w)), (S_2, R_3 + \gamma\hat{v}(S_3, w)), ..., (S_{T-1}, R_T)$. $\Delta w = \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w))x(S_t)$. Linear TD(0) converges close to the global optimum.

- **TD(λ)**

  Return G_t^λ is also a biased sample of $v_\pi(S_t)$. It is again possible to apply supervised learning to the data $(S_1, G_1^\lambda), (S_2, G_2^\lambda), ..., (S_{T-1}, G_{T-1}^\lambda)$.

  - Forward view linear TD(λ)

    $$\Delta w = \alpha(G_t^\lambda - \hat{v}(S_t, w))x(S_t)$$

  - Backward view linear TD(λ)

    $$\begin{aligned} \delta_t &= R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w) \\ E_t &= \gamma\lambda E_{t-1} + \hat{v}(S_t, w) \\ \Delta w &= \alpha\delta_t E_t \end{aligned}$$

> Here, the eligibility traces are defined for every parameter in the function approximator.

The forward and backward view of linear TD(λ) are again equivalent.

6.1.2 Control algorithms

Control, just like prediction, preserves the same intuition from its tabular case. First, *approximate* policy evaluation ($\hat{q}(.,.,w) \approx q_\pi$) will be performed, followed by an ϵ-greedy policy improvement. The goal becomes to minimize the following cost function: $J(w) = \mathbb{E}_\pi\left[(q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, w))^2\right]$. Then, $\Delta w = \alpha(q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, w))\nabla_w \hat{q}(S_t, A_t, w)$.

Now, the state and action are represented by a feature vector $x(S_t, A_t)$. The action-value approximation becomes $\hat{q}(S_t, A_t, w) = x(S, A)^\intercal w$. In this case, $\nabla_w \hat{q}(S_t, A_t, w) = x(S_t, A_t)$.

All algorithms work the exact same way as in the previous section about prediction. All that is different is the swap of v with q. The eligibility traces of backwards TD(λ) are still defined for all parameters.

6.1.3 Algorithm convergence

By using function approximators, the algorithms might not always converge. In some cases they diverge, but in other cases they also chatter around the near-optimal value function. In the case of control, this is because you are not sure if each step is actually improving the policy anymore. The following tables describe the convergence properties of prediction and control algorithms.

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	Yes	Yes	Yes
On-Policy	TD	Yes	Yes	No
On-Policy	Gradient TD	Yes	Yes	Yes
Off-Policy	MC	Yes	Yes	Yes
Off-Policy	MC	Yes	No	No
Off-Policy	Gradient TD	Yes	Yes	Yes

Figure 6.1: Guarantee of convergence properties of Prediction algorithms

As seen before, TD does not follow the gradient of any objective function. For this reason, TD might diverge when off-policy or using non-linear function approximation. **Gradient TD** is an algorithm that exists, which follows the true gradient of the projected Bellman error.

Algorithm	Table Lookup	Linear	Non-Linear
MC Control	Yes	Chatter	No
SARSA	Yes	Chatter	No
Q-Learning	Yes	No	No
Gradient Q-Learning	Yes	Yes	No

Figure 6.2: Guarantee of convergence properties of Control algorithms

Control algorithms are even worse than prediction algorithms. The next section in this chapter will discuss how to address these issues when using Neural Networks.

6.2 Batch Methods

The problem using gradient descent is that it is not sample efficient. It just experiences something once, and then removes it and moves on the the next experience. This means you don't make maximum use of the data to update the model. **Batch methods** seek to find the best fitting value function given the agent's experience.

6.2.1 Least-Squares Prediction

Given $\hat{v}(s,w) \approx v_\pi(s)$ and experience $D = \{(s_1, v_1^\pi), (s_2, v_2^\pi), ..., (s_T, v_T^\pi)\}$, the aim is to find which parameters w give the best fitting value $\hat{v}(s,w)$. **Least-Squares** algorithms aim to find parameter vector w that minimizes the squared error between $\hat{v}(s,w)$ and v_t^π. $LS(w) = \sum_{t=1}^{T}(v_t^\pi - \hat{v}(s_t,w))^2 = \mathbb{E}_D\left[(v^\pi - \hat{v}(s_t,w))^2\right]$.

This can be done by using **Experience Replay**. This method saves dataset D of experience, and repeatedly samples $(s, v^\pi) \sim D$. Then, it applies SGD to the network ($\Delta w = \alpha(v^\pi - \hat{v}(s,w))\nabla_w \hat{v}(s,w)$). Applying this algorithm will find $w^\pi = \arg\min_w LS(w)$.

Deep Q-Networks use experience replay together with **fixed Q-targets**. The algorithm consists of the following steps

- Take an action a_t according to some ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D

- Sample a random mini-batch of size n of transitions $(s, a, r, s') \sim D$
- Compute Q-learning targets with respect to old, fixed parameters w^- (these are the fixed Q-targets)
- optimize MSE between Q-network and Q-learning targets using some variant of SGD

$$LS_i(w_i) = \mathbb{E}_{s,a,r,s' \sim D_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]$$

Experience replay de-correlates the sequence in actions by randomly sampling actions at every step, leading to a better convergence. The *fixed Q-targets* are calculated from w_i^-, which are some old parameters of the network (they are frozen for a defined number of steps before being updated). This is used, because that way we don't bootstrap using our current network. If you do that, it could end up being unstable.

6.2.2 Linear Least-Squares Prediction

When using a linear value function approximation, it is possible to solve the cost function directly. This might be more efficient, since experience replay might take many iterations to find the optimal parameters. It is fairly simple.

$$\begin{aligned}
\mathbb{E}_D\left[\Delta w\right] &= 0 \\
\alpha \sum_{t=1}^{T} x(S_t)(v_t^\pi - x(S_t)^\intercal w) &= 0 \\
\sum_{t=1}^{T} x(S_t) v_t^\pi &= \sum_{t=1}^{T} x(S_t) x(S_t)^\intercal w \\
w &= \left(\sum_{t=1}^{T} x(S_t) x(S_t)^\intercal\right)^{-1} \sum_{t=1}^{T} x(S_t) v_t^\pi
\end{aligned}$$

For n features, the time complexity of solving this equation is $O(n^3)$. Since we do not know v_t^π, we can again use the estimates G_t, $R_{t+1} + \gamma \hat{v}(S_{t+1}, w)$, or G_t^λ. Respectively, these algorithms are called **LSMC**, **LSTD** and **LSTD(λ)**. For *off-policy* LSTD with a linear function, it will now also converge to the optimal value function, whereas standard TD does not guarantee this.

6.2.3 Least-Squares Control

An idea for control would be to do policy evaluation using *least-squares Q-Learning*, and then follow that with a greedy policy improvement. Approximate $q_\pi(s,a) \approx \hat{q}(s,a,w) = x(s,a)^\intercal w$. We want to minimize the squared error between those from experience generated by π. The data would consist of $D = \{(s_1, a_1, q_1^\pi), (s_2, a_2, q_2^\pi), ..., (s_T, a_T, q_T^\pi)\}$.

For control the policy aims to be improved. However, the experience has been generated by many different policies. So, to evaluate $q_\pi(s,a)$, learning must happen off-policy. The same idea as Q-learning can be used. First, use experience generated by the old policy $S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{old}$. Then, consider a successor action $A' = \pi_{new}(S_{t+1})$. $\hat{q}(S_t, A_t, w)$ should then be updated towards the alternative action $R_{t+1} + \gamma\hat{q}(S_t, A', w)$.

The **LSTDQ** algorithm does this. The calculation of the parameters can be derived in a similar way as linear least-squares prediction (Using linear Q-target $R_{t+1} + \gamma\hat{q}(S_{t+1}, \pi(S_{t+1}), w)$). After doing this derivation, the result becomes

$$w = \left(\sum_{t=1}^{T} x(S_t, A_t)(x(S_t, A_t) - \gamma x(S_{t+1}, \pi(S_{t+1})))^\intercal\right)^{-1} \sum_{t=1}^{T} x(S_t, A_t) R_{t+1}$$

Now, it is possible to use LSTDQ for evaluation in Least-Squares Policy Iteration (**LSPI-TD**).

Algorithm 10 LSPI-TD

```
Require: $D$, $\pi_0$
  $\pi' \Leftarrow \pi_0$
  while $\pi' \not\approx \pi$ do
    $\pi \Leftarrow \pi'$
    $Q \Leftarrow LSTDQ(\pi, D)$
    for all $s \in S$ do
      $\pi' \Leftarrow \arg\max_{a \in A} Q(s,a)$
    end for
  end while
  return $\pi$
```

The algorithm uses LSTDQ for policy evaluation. Then, it repeatedly re-evaluates experience D with different policies. THE LSPI algorithm always converges when doing table lookup, and chatters around the near-optimal value function when using linear approximation. Non-linear approximation is not possible, since LSTDQ uses a linear target.

Chapter 7

Policy Gradient Methods

The goal of this chapter is to discuss methods that optimize for policy directly, and not by acting greedily to a value function. The goal is to sample experience, and from that learn in which direction to move the policy to improve it. This will be done by considering the *gradient* of the policy.

For this, the policy will be parametrised by θ, so $\pi_\theta(s, a) = \mathbb{P}\left[a|s, \theta\right]$. This could be, for example, a neural network that predicts the probability of actions given a certain state.

Now, it is possible to talk about more approaches to Reinforcement Learning.

- Value Based

  The value function is learned, and there is an explicit policy

- Policy Based

  The policy is learned, but there is no value function

- Actor-Critic

  Both the value function and the policy are learned

The *advantages* over Policy-Based RL are the following: Better convergence properties, effective in high-dimensional/continuous action spaces, and can learn stochastic policies. However, there are also *disadvantages.* Evaluating a policy is typically inefficient and has high variance. Also, it usually converges to a local rather than a global optimum.

In POMDPs, a stochastic policy may be optimal in certain problems. But how to measure the quality of a policy? There are different options.

1. Start value (Episodic environments, start state)

   $J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}\left[v_1\right]$. This means it is a measure of reward until the end of an episode with current parameters θ.

2. Average value (Continuous environments)

   $J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s_1)$ where $d^{\pi_\theta}(s)$ is the stationary distribution (probability of being in a state) of the Markov chain for π_θ. It is basically an averaged value over all states.

3. Average reward per time-step

   $J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) R_s^a$. This is a weighted average over the possible immediate rewards for each state and action pair.

However, the policy gradient of all of these are the same. Policy-based RL is an optimization problem to find the θ that maximizes $J(\theta)$. Methods that do not use a gradient (e.g. Hill climbing) can be used. However, it is usually much more efficient to use gradients (e.g. Gradient Descent, Conjugate Gradient, Quasi-Newton).

7.1 Finite Difference Policy Gradient

Policy gradient algorithms search for a local maximum in $J(\theta)$ by ascending the gradient with respect to θ. This can be represented by $\Delta\theta = \alpha \nabla_\theta J(\theta)$.

Computing gradients by finite differences means to numerically compute them. For each dimension $k \in [1, n]$, estimate the k-th partial derivative by numerical approximation. $\frac{\delta J(\theta)}{\delta \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon}$, where u_k is the a unit vector with a 1 in the k-th component.

It is noisy and inefficient, but can sometimes be effective. An advantage is that it even works with non-differentiable policies.

7.2 Monte-Carlo Policy Gradient

The goal now is to compute the policy gradient *analytically.* Assume that π_θ is differentiable when it is non-zero and gradient $\nabla_\theta \pi_\theta(s,a)$ is known.

Likelihood Ratios exploit the identity $\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} = \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a)$. Here, the **score function** is defined by $\nabla_\theta \log \pi_\theta(s,a)$. This is useful, since it will make computing expectations much easier.

An example is the *Softmax Policy* $\pi_\theta(s,a) \propto e^{x(s,a)^\intercal \theta}$. For this, the score function is $\nabla_\theta \log \pi_\theta(s,a) = x(s,a) - \mathbb{E}_{\pi_\theta}\left[x(s,.)\right]$.

Another one is a *Gaussian Policy.* These are common in continuous action spaces. Here, the mean is a linear combination of state features $\mu(s) = x(s)^\intercal \theta$. The same can be done for the variance, with different parameters. Then, $a \sim N(\mu(s), \sigma^2)$. The score function is $\nabla_\theta \log \pi_\theta(s,a) = \frac{(a - \mu(s)) x(s)}{\sigma^2}$.

In terms of MDPs, the policy gradient can be derived in the following manner.

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta}\left[Q^{\pi_\theta}(s,a)\right] \\ &= \sum_{s\in S} d(s) \sum_{a\in A} \pi_\theta(s,a) Q^{\pi_\theta}(s,a) \\ \nabla_\theta J(\theta) &= \sum_{s\in S} d(s) \sum_{a\in A} \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a) \\ &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)\right] \end{aligned} \tag{7.1}$$

This is called the **Policy Gradient Theorem**. The theorem applies to all 3 of the previously discussed objective functions.

The algorithm for **Monte-Carlo Policy Gradient (REINFORCE)** uses the return G_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$ and this theorem to update the parameters using stochastic gradient descent.

Algorithm 11 MCPG (REINFORCE)

```
Require: $\theta$
  for all episodes $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do
    for $t \Leftarrow 1, ..., T-1$ do
      $\theta \Leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) G_t$
    end for
  end for
  return $\theta$
```

7.3 Actor-Critic Policy Gradient

The algorithms presented before are sadly slow and have a high variance. The following ideas are to speed of these algorithms and use a similar idea as before. To reduce this variance (but introduce some bias), the idea of function approximation can be re-used on the estimate of $Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$.

The idea consists of two components

- **Actor**: Update action-value function parameters w
- **Critic**: Update policy parameters θ, in the direction of suggestion of the critic.

These algorithms follow an *approximate* policy gradient $\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)\right]$.

The critic is solving the problem of policy evaluation. So, previously discussed methods can be applied for solving it. An example algorithm below is an *Action-Value Actor-Critic* with the critic using TD(0) for policy evaluation, and the actor using policy gradient.

Algorithm 12 QAC

```
Require: $s$, $\theta$
  $a \sim \pi_\theta$
  for all steps do
    $r \Leftarrow R_s^a, s' \sim P_s^a$
    $a' \sim \pi_\theta(s', a')$
    $\delta \Leftarrow r + \gamma Q_w(s', a') - Q_w(s, a)$
    $\theta \Leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
    $w \Leftarrow w + \beta \delta x(s, a)$
    $s \Leftarrow s', a \Leftarrow a'$
  end for
```

Since there is bias, usually the algorithms will end up in a local optimum. However, if the approximation of the value function is chosen carefully, the policy gradient is exact. **(Compatible Function Approximation Theorem** says that the policy gradient is *exact* ($\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)\right]$), when two conditions are satisfied.

1. $\nabla_w Q_w(s, a) = \nabla_\theta \log \pi_\theta(s, a)$ (value function approximator is *compatible* to the policy)

2. $\epsilon = \mathbb{E}_{\pi_\theta}\left[(Q^{\pi_\theta}(s, a) - Q_w(s, a)^2)\right]$ (value function parameters minimize the MSE)

To reduce the variance on the method, it is possible to subtract a **baseline function** $B(s)$ from the policy gradient. This can reduce variance without affecting the expectation. The proof is as follows

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a) B(s)\right] &= \sum_{s \in S} d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s, a) B(s) \\
&= \sum_{s \in S} d^{\pi_\theta}(s) B(s) \nabla_\theta \sum_a \pi_\theta(s, a) \\
&= \sum_{s \in S} d^{\pi_\theta}(s) B(s) \nabla_\theta 1 \\
&= 0
\end{aligned}$$

So, as long as the baseline does not include the action, it will not modify the expectation. An example of a good baseline function is $B(s) = V^{\pi_\theta}(s)$. The policy gradient can then be rewritten as the **advantage function** $A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s)$. $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a) A^{\pi_\theta}(s, a)\right]$. This can significantly reduce variance, since the advantage function is now the distance from the mean value of a state.

The critic is in charge of estimating both of these value functions. There are multiple ways to do this.

- Using two separate parameter vectors: $V_v(s) \approx V^{\pi_\theta}(s)$ and $Q_w(s) \approx Q^{\pi_\theta}(s,a)$. Then, $A(s,a) = Q_w(s,a) - V_v(s)$.

- Using the TD error

  The TD error for $V^{\pi_\theta}(s)$ equals $\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$. Then,

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\delta^{\pi_\theta}|s,a\right] &= \mathbb{E}_{\pi_\theta}\left[r + \gamma V^{\pi_\theta}(s')|s,a\right] - V^{\pi_\theta}(s)\\ &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)\\ &= A^{\pi_\theta}(s,a)\end{aligned}$$

  So, the TD error can be used to compute the policy gradient. $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\delta^{\pi_\theta}\right]$. In practice, $\delta_v = r + \gamma V_v(s') - V_v(s)$ is used instead. This approach only requires one set of critic parameters v.

The value function for the critic can be estimated at different time-scales, as seen in previous lectures. Ideas like MC, TD(0), and TD(λ) still work. The same thing holds for actors. Monte-Carlo policy gradient can be applied, as well as one-step TD for Actor-critic. In summary, this means the actor and critic's choices of algorithms do not influence each other.

It is also possible to apply policy gradient with eligibility traces for TD(λ). $\Delta\theta = \alpha(v_t^\lambda - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$, where $v_t^\lambda - V_v(s_t)$ is a biased estimate of the *advantage function.*

$$\begin{aligned}\delta &= r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)\\ e_{t+1} &= \lambda e_t + \nabla_\theta \log \pi_\theta(s,a)\\ \Delta\theta &= \alpha\delta e_t\end{aligned}$$

Similar to prediction using TD(λ) with a value function approximation, the eligibility traces are defined for each parameter and updated using the scores it encounters. The gradient with respect to the parameters becomes $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\delta e\right]$.

Gradient ascent algorithms can follow any ascent direction. Following a good ascent direction can massively speed up convergence. Another problem is that a policy can often be reparametrised without changing action probabilities. The vanilla gradient is sensitive to these reparametrisations.

The benefit of the **Natural Policy Gradient** is that it is parametrisations independent. It finds the ascent direction that is closest to the vanilla gradient when changing the policy by a small fixed amount.

$$\nabla_\theta^{nat}\pi_\theta(s,a) = G_\theta^{-1}\nabla_\theta\pi_\theta(s,a)$$

Here, G_θ is the *Fisher information matrix.* $G_\theta = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a)^\intercal\right]$. Using compatible function approximation, $\nabla_\theta^{nat} J(\theta) = w$. The actor parameters are updated in the direction of the critic parameters.

Chapter 8

Integrating Learning and Planning

The goal of this lecture is to provide as an introduction of *learning a model* of an MDP directly from experience, and use *planning* to construct a value function or policy. These two are then combined into a single architecture. The difference between Model-Free and Model-Based RL are the following:

- Model-Free

  No model, learn value function (and/or policy) from experience

- Model-Based

  Learn model from experience, plan value function (and/or policy) from model

8.1 Model-Based RL

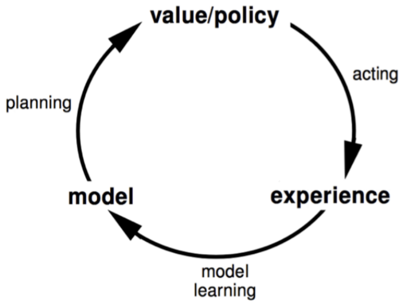


Figure 8.1: Model-Based RL cycle

The advantages of Model-Based RL are that models can be efficiently learned using *supervised learning* methods. It is also possible to then reason of model uncertainty. A disadvantage, however, is that first the model must be learned. Only then can the value function be constructed. This also means there are two sources of approximation error, and the accuracy of the value function is limited to the accuracy of the model.

A *model* M is a representation of an MDP parameterized by η. In this case, it is assumed that the state and action spaces are known (for simplification). So then, a model $M = (P_\eta, R_\eta)$ represents state transitions $P_\eta \approx P$ and rewards $R_\eta = R$. These can be used to estimate rewards and state transitions:

$$
\begin{aligned}
S_{t+1} &\sim P_\eta(S_{t+1}|S_t, A_t)\\
R_{t+1} &= R_\eta(S_{t+1}|R_t, A_t)
\end{aligned}
$$

Typically, a conditional independence between state transitions and rewards are assumed $(P(S_{t+1}, R_{t+1}|S_t, A_t) = P(S_{t+1}|S_t, A_t)P(R_{t+1}|S_t, A_t))$.

So, the goal is to estimate M_η from experience $\{S_1, A_1, R_2, ..., S_T\}$. This is a supervised learning problem. More specifically, it is usually best to split up the two tasks of learning R_η and P_η. $S_t, A_t \Rightarrow R_{t+1}$ is a *regression problem*, while $S_t, A_t \Rightarrow S_{t+1}$ is a *density estimation* problem.

There are a lot of ways to create a model. This chapter focusses on the *Table-Lookup Model*.

8.1.1 Table-Lookup Model

The idea here is to count the visits $N(s, a)$ to each state-action pair. Then, similar to what TD learns, the idea is to construct the maximum-likelihood Markov model from this. It can be constructed in the following manner.

$$
\begin{aligned}
\hat{P^a_{ss'}} &= \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} 1(s^k_t, a^k_t, s^k_{t+1} = s, a, s')\\
\hat{R^a_s} &= \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} 1(s^k_t, a^k_t = s, a) r^k_t
\end{aligned}
\tag{8.1}
$$

Alternative, it is possible to record an experience tuple $(S_t, A_t, R_{t+1}, S_{t+1})$ at each time-step t. In order to then sample the model, uniformly randomly pick a tuple matching $(S_t, A_t, ., .)$.

8.1.2 Planning with a Model

Given a model $M_\eta = (P_\eta, R_\eta)$, the goal is to solve the MDP (S, A, P_η, R_η) using a planning algorithm (e.g. Value iteration, Policy iteration, Tree search, ...).

Algorithms like Value iteration are quite expensive, since they go over the entire state(-action) space. **Sample-Based** planning uses the model to generate samples. This is often much more efficient. It also means more frequent states will be sampled more often. This can be a good property to have.

The sampling happens in the same way as in equation 8.1. At each step, model-free RL can be applied to the samples (e.g. Monte-Carlo control, SARSA, Q-Learning, ...).

Given an imperfect model $(P_\eta, R_\eta) \neq (P, R)$, the performance of model-based RL is limited to the optimal policy of approximate MDP (S, A, P_η, R_η). So, when the model is inaccurate, planning will compute a sub-optimal policy. It is always possible to switch to model-free RL. However, there are ways of reasoning about the *uncertainty* in the model. This can be done using for example Bayesian modelling.

8.2 Integrated Architectures

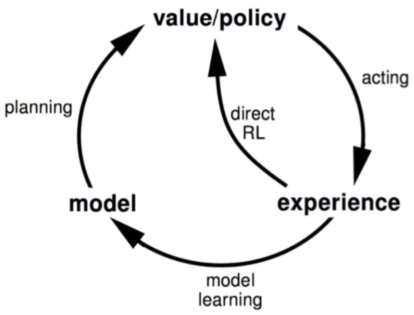


Figure 8.2: Dyna architecture cycle

It is possible to learn and plan the value function (and/or policy) from real and simulated experience. This architecture is referred to as **Dyna**.The algorithm below uses Dyna with Q-Learning.

Algorithm 13 Dyna-Q

```
Require: $Q$, $M$, $n$ (number of planning steps)
  $S \Leftarrow$ current state
  $A \Leftarrow \epsilon$-greedy$(S, Q)$
  Execute $A$; observe $R, S'$
  $Q(S,A) \Leftarrow Q(S,A) + \alpha\,[R + \gamma \max_a Q(S',a) - Q(S,A)]$
  $M \Leftarrow R, S'$
  for $i = 1, ..., n$ do
    $S, A \Leftarrow$ random previously taken state-action pair
    $R, S' \Leftarrow M(S,A)$
    $Q(S,A) \Leftarrow Q(S,A) + \alpha\,[R + \gamma \max_a Q(S',a) - Q(S,A)]$
  end for
```

Here, the Q-function is learned from both real experience and planned experience.

8.3 Simulation-Based Search

Forward search algorithms select the best action by *lookahead.* They build a search tree with current state s_t at the root. The idea is to use a *model* of the MDP to perform lookahead. Then, only a subsection of the entire MDP must be solved.

Simulation-Based search simulates episodes of experience from the current state with the model. So, from current state s_t, K episodes are sampled:

$$\{s_t^k, A_t^k, R_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim M$$

Then, model-free RL is applied to the simulated episodes. Monte-carlo control gives rise to **Monte-Carlo search**, while SARSA gives rise to **TD search**.

Simple Monte-Carlo search uses the mean return to construct Q-values $Q(s_t, a) = \frac{1}{K}\sum_{k=1}^K G_t$ for each action $a \in A$. This estimates $q_\pi(s_t, a)$, where π is the simulation policy. The real action is then taken using $a_t = \arg\max_{a \in A} Q(s_t, a)$.

8.3.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) works differently. It has two policies, a *Tree policy* and a *Default policy.* The tree policy picks the actions to maximize $Q(S, A)$ and the default policy is fixed and takes random actions. So, the tree policy improves over time.

At each simulation, it first uses the tree policy to find with state to simulate from. Then, the simulation (also referred to as rollout) happens using the default policy. The Q-values are then the mean return of episodes.

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{u=t}^{T} 1(S_u, A_u = s, a) G_u$$

After the planning has finished, the real action is again taken using policy $a_t = \arg\max_{a \in A} Q(s_t, a)$. MCTS is MC-control applied to simulated experience. It converges on the optimal search tree, $Q(S, A)$ becomes $q_*(S, A)$.

There are several advantages to using MCTS

- The best-first search is highly selective
- Evaluates states dynamically, unlike DP
- The use of sampling breaks the curse of dimensionality
- It works for black-box models, all it needs are samples
- Computationally efficient, anytime and parallelisable

8.3.2 Temporal-Difference Search

The idea is to use TD instead of MC, by bootstrapping. It applies SARSA to the sub-MDP. It can be helpful for the following reasons

- TD search reduces variance, but increases bias
- TD search is usually more efficient than MC search
- TD(λ) can be much more efficient than MC search

TD search simulates episodes from the current state s_t. It estimates the value function $Q(S, A)$. At each step of the simulation, $\Delta Q(S, A) = \alpha(R + \gamma Q(S', A') - Q(S, A))$. Actions are selection ϵ-greedily with respect to the Q-values. A function approximator can also be used for Q.

There is another idea called **Dyna-2**. The agent stores two sets of feature weights: *Long-term memory* and *Short-term (working) memory.*

The long-term memory is updated from real experience with the environment. It is perceived as general knowledge that applies to any episode. On the other hand, the short-term memory is updated from simulated experience using TD search. This can figure out specific local knowledge about the current situation.

The value function is then a sum of the long- and short-term memories.

Chapter 9

Exploration and Exploitation

The trade-off between exploration and exploitation is a fundamental problem in Reinforcement Learning and online decision-making. It provides a trade-off between making a decision based on the current beliefs, or gather more of this information.

In general, the long-term is preferred over short-turn. This means the best strategy might include short-term sacrifices. The foal is to gather enough information to make the best overall decisions.

This chapter will first explain exploration in a special forms of an MDP, called bandit problems. Finally, these ideas are generalized to the general MDP.

There are lots of *principles* to make sure to explore. There include but are not limited to the following

- **Naive Exploration**: Add noise to greedy policy (e.g. ϵ-greedy)
- **Optimistic Initialization**: Assume the best until proven otherwise
- **Optimism in the face of uncertainty**: Prefer actions with uncertain values
- **Probability Matching**: Select actions based on probability their are best
- **Information State Search**: Look-ahead search incorporating value of information

9.1 Multi-Armed Bandits

A multi-armed bandit is a tuple (A, R). Here, A is a known set of actions (arms) and $R^a(r) = \mathbb{P}\left[r|a\right]$ is an unknown probability distribution over rewards. At each step t, the agent chooses an action $a_t \in A$. Then, the environment generates a reward $r_t \sim R^{a_t}$. The goal is to maximize the cumulative reward $\sum_{T=1}^{t} r_T$.

The goal is to maximize this reward. However, this can otherwise be states as *minimizing* the total **regret**. The action-value is the expected reward of an action ($Q(a) = \mathbb{E}\left[r|a\right]$). The optimal value $V* = Q(a^*) = \max_{a \in A} Q(a)$.

The regret is the measured opportunity loss for one step: $l_t = \mathbb{E}\left[V^* - Q(a_t)\right]$. The *total regret* is then equal to the total opportunity loss $L_t = \mathbb{E}\left[\sum_{T=1}^{t} V^* - Q(a_t)\right]$.

It is possible to count regret. The *count* $N_t(a)$ is the expected number of selection for action a. For that action, there is a *gap* in difference of value between a and a^*. This gap equals $\Delta_a = V^* - Q(a)$. The regret can then be rewritten to the following

$$\begin{aligned} L_t &= \mathbb{E}\left[\sum_{T=1}^{t} V^* - Q(a_t)\right] \\ &= \sum_{a \in A} \mathbb{E}\left[N_t(a)\right]\left(V^* - Q(a_t)\right) \\ &= \sum_{a \in A} \mathbb{E}\left[N_t(a)\right]\Delta_a \end{aligned}$$

So, the regret is a function of the gaps and counts. A good algorithm will ensure small counts for large gaps. There is a problem however; the gaps are not known.

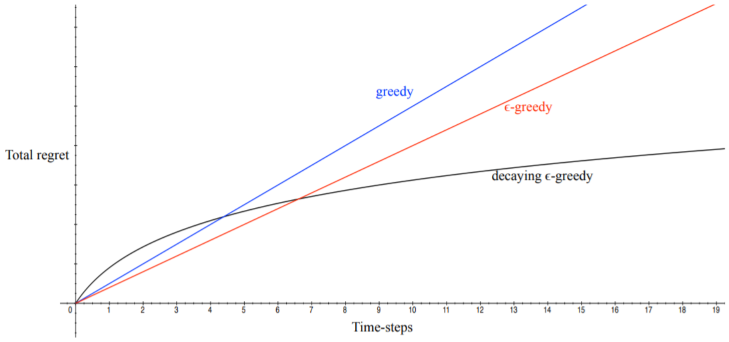


Figure 9.1: Total regret over time

From 9.1, the following observations can be made. If an algorithm *never-* or *forever* explores, the total regret will be *linear*. This is not good, since the goal is to minimize regret while exploring. The idea of *decaying* ϵ-greedy is interesting, since it will converge to a logarithmic regret.

Lets estimate $Q(a) \approx \hat{Q}_t(a)$. The value of each action can be evaluated by MC-evaluation: $\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{t=1}^{T} r_t 1(a_t = a)$.

Since the *greedy* algorithm always selects the action with the highest value, it can lock onto a sub-optimal action forever, thus receiving a linear total regret.

Using an ϵ-*greedy* algorithm, the constant ϵ ensures that the minimum regret $l_t \geq \frac{\epsilon}{A} \sum_{a \in A} \Delta_a$. Thus, the total regret is also linear.

A simple but powerful idea is to use **Optimistic Initialization**. Initialize $Q(a)$ to a very high value and from there on update the action-value by MC evaluation starting with $N(a) > 0$. It encourages exploration early on, but can still lock onto sub-optimal actions. In practice however, this generally works really well.

The *decaying ϵ-greedy* algorithm works by picking a decaying schedule for $\epsilon_1, \epsilon_2, \ldots$. Imagine the following schedule

$$\begin{aligned} & c > 0 \\ [\ & d = \min_{a|\Delta_a>0} \Delta_i \\ & \epsilon_t = \min\{1, \frac{c|A|}{d^2 t}\} \end{aligned}$$

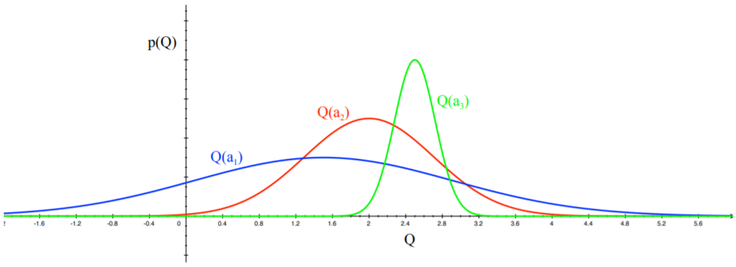
There is a problem with the schedule though. It uses advance knowledge about gaps. So, the goal is to find an algorithm with sub-linear regret for any multi-armed bandit without using knowledge of R.

The performance of any algorithm is determined by the *similarity* between the optimal arm and other arms. Hard problem have similar-looking arms with different means. This can be described as the size of the gap, and the similarity in distributions (measure by KL-divergence).

The following **Lower Bound** is a theorem. It means the *asymptotic total regret is at least logarithmic in the number of steps.*

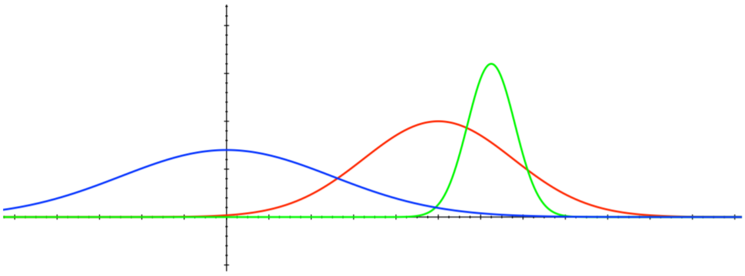
$$\lim_{t\to\infty} L_t \geq \log t \sum_{a|\Delta_a>0} \frac{\Delta_a}{KL(R^a||R^{a^*})}$$

9.1.1 Upper Confidence Bounds



Take a look at the figure above. The idea of **Optimism in the Face of Uncertainty** is to pick the action with the highest potential. This would be

a_1, since the tail of the distribution is the best one of them all. Say you pick this action. Then, the distribution gets updated.



Now, we are more certain about the actual value of $Q(a_1)$ and thus more likely to pick another action until the optimal one has been discovered.

Another way to estimate uncertainty is through **Upper Confidence Bounds** (UCB). An upper confidence $\hat{U}_t(a)$ can be calculated for each action-value, such that $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$ with high probability. The action maximizing the UCB is then selected.

By **Hoeffding's Inequality**, $\mathbb{P}\left[\mathbb{E}\left[X\right] > \bar{X}_t + u\right] \leq e^{-2tu^2}$ for $\bar{X}_t = \frac{1}{T}\sum_{t=1}^{T} X_t$, where X_t is an i.i.d. r.v. in $[0,1]$. When applying this to the bandit problem, the following can be derived

$$\mathbb{P}\left[Q(a) > \hat{Q}_t(a) + U_t(a)\right] \leq e^{-2N_t(a)U_t(a)^2}$$

Now, pick a probability p that the true value exceeds the UCB.

$$p = e^{-2N_t(a)U_t(a)^2}$$
$$U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

Since the goal is to reduce the probability as more rewards are observed, set e.g. $p = t^{-4}$. Then ensures the optimal action is selected when $t \to \infty$. The final equation becomes

$$U_t(a) = \sqrt{\frac{2\log t}{N_t(a)}}$$

This leads to the **UCB1** algorithm

$$a_t = \arg\max_{a\in A} Q(a) + \sqrt{\frac{2\log t}{N_t(a)}}$$

Theorem: the UCB algorithm achieves logarithmic asymptotic total regret

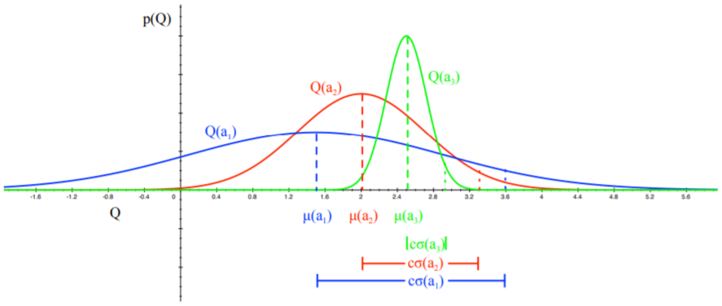
$$\lim_{t\to\infty} L_t \leq 8 \log t \sum_{a|\Delta_a>0} \Delta_a$$

9.1.2 Bayesian Bandits

So far, no assumptions were made about the reward distribution R, except for bounds on rewards. **Bayesian Bandits** exploit prior knowledge of rewards, $p\left[R\right]$. They compute the posterior distribution of reward $p\left[R|h_t\right]$, where $h_t = a_1, r_1, ..., a_{t-1}, r_{t-1}$ is the history. The posterior distribution can then be used to guide exploration.

The following is an example. Assume that the reward distribution is Gaussian, so $R_a(r) = N(r; \mu_a, \sigma_a^2)$. The Gaussian posterior over μ_a and σ_a^2 can be computed using Bayes' law: $p[\mu_a, \sigma_a^2; h_t] \propto p[\mu_a, \sigma_a^2] \prod_{t|a_t=a} N(r_t; \mu_a, \sigma_a^2)$.

Then, pick the action that maximizes the standard deviation of $Q(a)$, so $a_t = \arg\max_{a\in A} \mu_a + \frac{c\sigma_a}{\sqrt{N(a)}}$. This is a Bayesian way to compute Upper Confidence Bounds.



Here, the blue action would be taken. This is because the mean and standard deviation summed up is the largest (assuming an equal number of visits).

9.1.3 Probability Matching

Probability Matching selects an action a according to the probability that a is optimal. It can be calculated using

$$\pi(a) = \mathbb{P}\left[Q(a) = \max_{a'} Q(a') | R_1, ..., R_{t-1}\right]$$

This concept is optimistic in the face of uncertainty, since uncertain actions have a higher probability of being the maximum. A disadvantage however is that it can be difficult to be computed analytically from the posterior.

Thompson Sampling is sample-based probability matching. The policy can be computed using

$$\pi(a) = \mathbb{E}\left[1(Q(a) = \max_{a'} Q(a'))|R_1, ..., R_{t-1}\right]$$

Then, use Bayes' law to compute the posterior distribution $P_w(Q|R_1, ..., R_{t-1})$. First sample an action-value $Q(a)$ from the posterior. Then, choose action $A_t = \arg\max_{a \in A} Q(a)$.

9.1.4 Information State Search

The goal of exploration is to gain more information about the environment. If it is possible to quantify how much this information is worth. When this value is known, the optimal trade-off between exploration and exploitation is known.

The idea is to look at bandits as *sequential* decision-making processes instead of *one-step*. At each step, there is an **information state** $\bar{s} = f(h_t)$. So, the state is a summary of all history that has been accumulated so far. Each action a then causes a transition to a new information state $\bar{s}'$, with probability $\bar{P}^a_{\bar{s}\bar{s}'}$.

This defines the MDP $\bar{M}$ in augmented information state space $\bar{M} = (\bar{S}, A, \bar{P}, R, \gamma)$.

An example of this are Bernoulli Bandits, so lets say $R^a = Bernoulli(\mu_a)$. The goal is to find the arm with the highest μ_a. The information state space $\bar{s} = (\alpha, \beta)$, where α_a and β_b count when the reward was 0 or 1 respectively. This MDP can then be solved by previously discussed Reinforcement Learning algorithms. There is an alternative approach, referred to as **Bayes-adaptive RL**. For Bernoulli bandits, it finds the Bayes-optimal exploration/exploitation trade-off with respect to the prior distribution.

Imagine a problem of drug assignment. There are two drugs 1 and 2 that can be used; $\bar{s} = (\alpha_1, \beta_1, \alpha_2, \beta_2)$. First, start with a $Beta(\alpha_a, \beta_a)$ prior over reward function R^a. Each time a is selected, the distribution gets updated accordingly, since a success will increase beta, and a failure increases alpha. Here, each state transition corresponds to a Bayesian model update.

The solution of this problem is known as the *Gittens* index, and can be computed by solving the MDP (by using for example MCTS for planning).

9.2 Contextual Bandits

A **Contextual Bandit** is a tuple (A, S, R). Instead of the bandit problem, there is now also a state space, which provides *context* to the problem. $S =$

$\mathbb{P}[s]$ is an unknown distribution over the contexts. Then, the reward $R_s^a(r) = \mathbb{P}[r|s,a]$ is an unknown probability distribution of the rewards. At each time step, the environment generates $s_t \sim S$. Then, an agent selects $a_t \in A$, to which the environment generates $r_t \sim R_{s_t}^{a_t}$. The goal is to maximize $\sum_{T=1}^{t} r_T$.

This section explains the concept of **Linear UCB**. The action-value function needs to be estimated: $Q(s,a) = \mathbb{E}[r|s,a] \approx Q_\theta(s,a) = x(s,a)^\intercal\theta$. The parameters of this linear function can be estimated by least-squares. The solution is

$$\theta_t = \left(\sum_{T=1}^{t} x(s_T, a_T)x(s_T, a_T)^\intercal\right)^{-1} \sum_{T=1}^{t} x(s_T, a_T)r_T$$

Least squares regression estimates $Q_\theta(s,a)$. However, it can also be used to estimate the variance of the action-value $\sigma_\theta^2(s,a)$. The idea is to add a bonus for uncertainty, $U_\theta(s,a) = c\sigma$, where the UCB is the number of standard deviations above the mean defined by c.

For least squares regression, parameter covariance is $A^{-1} = \left(\sum_{T=1}^{t} x(s_T, a_T)x(s_T, a_T)^\intercal\right)^{-1}$. The action-value is linear in features, so

$$\begin{aligned}
\sigma_\theta^2(s,a) &= x(s,a)^\intercal A^{-1} x(s,a) \\
a_t &= \arg\max_{a \in A} Q_\theta(s_t, a) + c\sigma_\theta(s_t, a) \\
a_t &= \arg\max_{a \in A} Q_\theta(s_t, a) + c\sqrt{x(s,a)^\intercal A^{-1} x(s,a)}
\end{aligned}$$

9.3 MDPs

All previously described principles for exploration/exploitation can apply to MDPs with slights modifications.

- Optimistic Initialization
  - Model-Free RL

    The idea is to initialize $Q(s,a)$ to $\frac{r_{max}}{1-\gamma}$. This encourages systematic exploration of states and actions.
  - Model-Based RL

    Here, an optimistic model of the MDP is constructed. For example, transitions to a terminal state are initialized with r_{max} reward. An example is the RMax algorithm (Brafman and Tennenholtz).
- Optimism in the Face of Uncertainty

- Model-Free RL

  The goal is to maximize UCB on the action-value function $Q^{\pi}(s, a)$. $a_t = \arg\max_{a \in A} Q(s_t, a) + U(s_t, a)$. It estimates uncertainty in policy evaluation. However, it completely ignores uncertainty in the policy improvement step.

  For this reason, maximize UCB on $Q^*(s, a)$ instead. $a_t = \arg\max_{a \in A} Q(s_t, a) + U_1(s_t, a) + U_2(s_t, a)$. This estimates uncertainty in both policy evaluation and policy improvement. However, estimating policy improvement uncertainty is hard to do.

- Model-Based RL

  Bayesian Model-Based RL maintains a posterior distribution over MDP models. Both transitions and rewards are estimated $p[P, R|h_t], h_t = s_1, a_1, r_2, ..., s_t$. This posterior can then be used to guide exploration by for example Bayesian UCB or Thompson Sampling.

- Probability Matching

  Thompson sampling implements probability matching

$$\begin{aligned}\pi(s, a|h_t) &= \mathbb{P}[Q^*(s, a) > Q^*(s, a'), \forall a' \neq a|h_t] \\ &= \mathbb{E}_{P,R|h_t}\left[1(a = \arg\max_{a \in A} Q^*(s, a))\right]\end{aligned}$$

  Bayes' Law can be used to compute posterior distribution $p[P, R|h_t]$. Then, sample an MDP P, R from the posterior. Then, solve this MDP using a favored planning algorithm to obtain $Q^*(s, a)$. Then, select the optimal action from the sample MDP by $a_t \arg\max_{a \in A} Q^*(s_t, a)$.

- Information State Search

  MDPs can also be augmented to include an information state. The augmented state becomes $(s, \hat{s})$, where s is the original state within the MDP, and $\hat{s}$ is accumulated information about the history. Each action a causes a transition to s' with probability $P^a_{ss'}$. However, a new information state $\hat{s}'$ is also created.

  The posterior distribution over the MDP model is an information state $\hat{s_t} = \mathbb{P}[P, R|h_t]$. An augmented MDP over $(s, \hat{s})$ is called a **Bayes-Adaptive MDP**.

  Solving this MDP gives rise to the optimal exploration/exploitation trade-off. However, the Bayes-Adaptive MDP is typically enormous. Simulation-based search has been proven effective to this problem.

Chapter 10

RL in Classic Games

Why are classic games such a meaningful area to apply Reinforcement Learning to? The main point is that these games usually have quite simple rules, but very deep concepts. They also provide for a meaningful IQ test, and encapsulate real world issues. This makes it a really interesting field for Reinforcement Learning.

10.1 Game Theory

In order to get a sense of optimality in multiplayer games, the field of Game Theory is often used. The aim is to find the optimal policy π^i for the i-th player. If all other players fix their policies (π^{-i}), the **best response** $\pi^i_*(\pi^{-i})$ is the optimal policy against those policies.

There is a problem with this, because it basically learns the optimal policy for *one* fixed policy of the opponents. Instead, the goal is to find the optimal policy against all policies opponents can have. The most well-known solution to this is referred to as the **Nash equilibrium**. This is a joint policy for all players.

$$\pi^i = \pi^i_*(\pi^{-i})$$

Here, every player's policy is a best response. This means no player would choose to deviate from this equilibrium.

- Best Response is a solution of a single-agent RL problem.

  Other players become part of the environment, and the game is reduced to an MDP. The best response is then an optimal policy of this MDP.

- Nash equilibrium is a fixed-point of self-play RL

  Experience is generated by playing games between agents $a_1 \sim \pi^1, a_2 \sim \pi^2, ...$ Each agent learns the best response to other players. One player's policy determines another player's environment. This means all players are adapting to each other.

To keep things relatively simple, this chapter restricts itself to **two player**, **zero sum** games. Zero sum games have equal and opposite rewards $R^1 + R^2 = 0$. Methods for finding Nash equilibria in these games are discussed.

10.2 Minimax Search

In these types of games, the value function defines the expected total reward given joint policies (π^1, π^2)

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

A *minimax* value function maximizes player 1's expected return while minimizing the one from player 2. The optimal value function becomes

$$v_*(s) = \max_{\pi^1} \min_{\pi^2} v_\pi(s)$$

A *minimax* policy (π^1, π^2) is a joint policy which achieves the minimax values. For two-player zero sum games, there exists a unique minimax value function. In this case, a minimax policy is a Nash equilibrium.

There is a problem though. The search tree grown exponentially at every level. This means it is impractical to search all the way to the end of the game. A simple solution to this problem uses a value function approximation $v(s, w) \approx v_\pi(s)$ at the leaf nodes of the search tree. Then, it is possible to run minimax to some fixed depth with respect to the leaf values.

10.3 Self-Play RL

Similar to this fixed function approximation in the last section, it is possible to apply value-based RL algorithms to games of self-play. Any algorithm like $MC, TD(0), TD(\lambda), ...$ can be used. For deterministic games, it is sufficient to estimate $v_*(s)$. The idea uses the following short proof

$$q_*(s, a) = \sum_{s'} P^a_{ss'} v_*(s)$$
$$q_*(s, a) = v_*(s')$$

Actions are selected by min- or maximizing the value of the next state

$$A_t = \arg\max_a v_*(S'_t) \text{ for the maximizing player}$$
$$A_t = \arg\min_a v_*(S'_t) \text{ for the minimizing player}$$

This improves the joint policy for both players.

10.4 Combining RL and Minimax Search

The ideas of both these methods can be combined. This section focuses on some ideas to combine both.

10.4.1 Simple TD

As previously seen, the idea of TD is to update the value function towards the successor value. The idea of *Simple TD* is to first learn the value function in TD learning. After learning, this value function can be used in minimax search (without any learning).

$$v_+(S_t, w) = minimax_{s \in leaves(S_t)} v(s, w)$$

An important question arises when using this approach: can the value function be learning directly from minimax search values? This could increase performance more quickly, since it can be hard to find a terminal state without search.

10.4.2 TD Root

The idea of *TD Root* is to update the value towards the successor search value. Imagine the tree is at root node S_t. After taking an action, S_{t+1} is explored next. TD would update the value of S_t towards the value of S_{t+1}. The value of S_{t+1} equals the value of the minimax search backup, so $v(S_{t+1}, w) = v(l(S_{t+1}), w)$, where $l(s)$ is the leaf node achieving minimax value from s.

This was actually the first ever TD learning algorithm applied to a checkers program that learned by self-play. It was able to defeat an amateur human player. This algorithm was then used and extended to perform better.

10.4.3 TD Leaf

One idea is to think about the whole search. Instead of updating the value of S_t, the value of the leaf $l(S_t)$ should be updated. This is the case, because that is the node that contributed to the minimax value $v(S_t)$. This results in the back-up $v(l(S_t), w) \leftarrow v(l(S_{t+1}), w)$, instead of $v(S_t, w) \leftarrow v(l(S_{t+1}), w)$ as seen in TD Root.

Knightcap used TD leaf in chess. It started by only knowing standard piece values, and learned weights using TD leaf. The search used $\alpha\beta$-pruning to speed up the minimax search. It ended up achieving master-level play after a small number of games. However, it was not effective in self-play or without starting from good weights. In checkers however, a similar algorithm called Chinook was able to play at superhuman level.

10.4.4 TreeStrap

There is still something to optimize. The previously algorithms use the values obtained from the minimax search only once. *TreeStrap* updates search values towards deeper search values. Minimax search values are computed at *all* nodes in the search. The backup $\forall s \in nodes(S_t) : v(s, w) \leftarrow v(l(s), w)$ is performed.

Meep, a chess algorithm using TreeStrap, was able to beat 13/15 international masters starting from random initial weights. It is both effective in self-play and random initial weights.

10.4.5 Simulation-Based Search

As you have seen now, self-play reinforcement learning can replace search. Next to the idea of minimax, it is also possible to simulate games of self-play from root state S_t, and then apply RL to this simulated experience.

Previously, a concept using Monte-Carlo Control called *MCTS* was discussed. The most effective variant of this search uses the UCT-algorithm to guide exploration/exploitation in every node. It actually turns out self-play UCT converges on minimax values in zero-sum, 2-player games.

10.5 RL in Imperfect-Information Games

In imperfect information games, players have different information states and therefore separate search trees. In this tree, there is one node for each information state. These nodes summarize what a player knows (e.g. the cards they have seen).

There may be many real states that share the same information state, but there may also be aggregate states (e.g. those with a similar value).

There are multiple ways to solve for these information-state game trees.

- Iterative forward-search methods

  An example of this is Counterfactual Regret Minimization

- Self-play Reinforcement learning

> Here, a variant of MCTS, called Smooth UCT is discussed briefly. Applying the method to the game of Poker resulted in 3 silver medals in two- and three-player Poker. It outperformed massive-scale forward-search agents.

10.5.1 Smooth UCT

Using Monte-Carlo Tree Search in fully observable games has shown promising. However, improvements must be made to let it perform well in imperfect information games. In Poker for example, naive MCTS diverges, since its approach is just not well-fitted for these types of games.

However, a variant of UCT, inspired by a game-theoretic concept called **Fictitious Player** was implemented. The idea is that the agent learns against and to respond to opponents' *average* behavior. This gives a much robust approach.

The average strategy can be extracted from nodes' action counts, $\pi_{avg}(a|s) = \frac{N(s,a)}{N(s)}$. Then, at each node, pick actions according to

$$A \sim \begin{cases} UCT(S), & \text{with probability } \epsilon \\ \pi_{avg}(.|S), & \text{with probability } 1-\epsilon \end{cases}$$

Empirically, in variants of Poker, smooth UCT converged to the Nash equilibrium while naive MCTS diverged.

Bibliography

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