
Università di Trieste

Dipartimento di Matematica e Geoscienze

Laurea Magistrale in Data Science
and Scientific Computing

# Briscola Bot

Lorenzo Cavuoti

Relatore:

**Prof: Antonio Celani**

Correlatore:

**Emanuele Panizon**

Candidato:

**Lorenzo Cavuoti**

**ANNO ACCADEMICO 2021/2022**

**ABSTRACT**

Recent developments in reinforcement learning demonstrate that it can be applied in a variety of domains, including card games. In this thesis, we present a reinforcement learning agent that learns to play the Italian card game of Briscola in a one-on-one setting. By utilizing the Proximal Policy Optimization (PPO) algorithm and training a deep neural network to learn the game's policy, our agent achieves performance comparable to that of an average human player. We further investigate the impact of various reward structures on the agent's learning process and identify potential areas of further improvement. To showcase the effectiveness of our agent, we provide a web-based version of the game where users can play against the trained agent, which can be accessed at https://replit.com/@LorenzoCavuoti/BriscolaBot. The code for the agent and the web-based game can be found in the GitHub repository https://github.com/LetteraUnica/BriscolaBot.

# Ringraziamenti

Da scrivere.

# Contents

# Chapter 1

# Background

When a human first plays a game, let it be chess, poker, Minecraft, CS:GO or any other game, it is not given any instructions on how to play. It is not told which moves are good and which are bad, maybe it doesn't even know the rules of the game. By interacting and playing the game, losing to some opponents and starting to beat others, the player gradually learns what moves are best in which situation and starts to understand the dynamics of the game. During all of this the player doesn't have any external feedback telling him which actions to perform in a given situation, but has to learn it by itself, by performing an action and experiencing the outcome. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence [1].

In this chapter we provide an overview of reinforcement learning. First we will discuss Markov Decision Processes (MDPs) that allow us to define an environment (for now we can think of an environment as a game) in mathematical terms, then we will discuss Dynamic Programming, that allows to solve exactly the reinforcement learning problem in case we know everything about the environment. Finally, we will see model-free methods, that allow us to learn even if we know nothing about the environment, these methods learn by interacting directly with the environment, obtaining experience and exploiting it to find the best actions to perform in any situation. This chapter is based on the first part of the Sutton and Barto book [1]

## 1.1 Reinforcement Learning

Reinforcement learning is the field of machine learning that tries to maximize a numerical reward signal. The agent is not said which actions to take, but instead must discover the actions that lead to the highest reward by trying them. Furthermore, actions do not only influence the immediate future reward but also the next situation and consequently subsequent rewards, so

the agent must learn to balance immediate and future rewards.

Reinforcement learning differs from supervised learning because there is no external supervisor, which tells the agent which actions to perform in a given situation, instead the agent must discover the actions that lead to the highest reward by trying them. Reinforcement learning is also different from unsupervised learning, because even though both approaches do not need labels to learn from, their goal is different. In unsupervised learning the goal is to find structure in the data, while in reinforcement learning the goal is to maximize a reward signal.

One important aspect of reinforcement learning is the exploration-exploitation trade-off, which is not present in supervised or unsupervised learning. The agent must weigh the benefits of exploring new actions against the benefits of utilizing the best actions it has already discovered. A focus on exploration alone may lead to the discovery of the best actions, but with infrequent use, while a focus on exploitation alone may prevent the discovery of potentially better actions.

## 1.2 Markov Decision Processes

We can formalize the problem of learning from interaction to achieve a goal with Markov Decision Processes (MDPs). In the MPD framework the learner is called the agent, and the thing it interacts with is called the environment. The agent continually interacts with the environment by performing actions, on each action the environment changes state, and presents a new situation to the agent. The environment is also responsible to give the reward signal the agent must maximize 1.1.

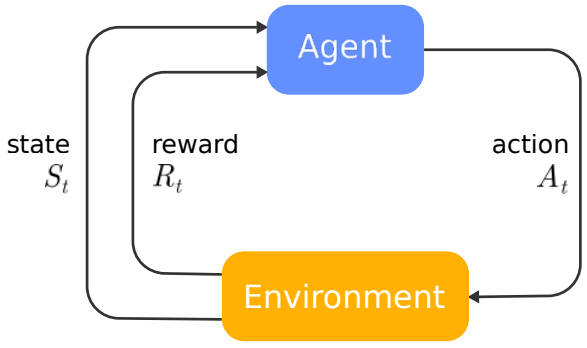


**Figure 1.1:** Illustration of the agent-environment interaction in a Markov Decision Process, image adapted from [1].

More specifically, the agent and environment interact in discrete time steps ($t = 0, 1, 2 \dots$). At each time step $t$ the agent receives an observa-

tion of the environment's state $S_t \in S$ and based on that chooses an action $A_t \in A$. In the next time step $t+1$ the agent receives from the environment a numerical reward $R_{t+1} \in R \in \mathbb{R}$ and a new observation $S_{t+1} \in S$ and so on. This generates a sequence of observations, actions and rewards, which we call trajectory: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$.

In finite MDPs the set of states $S$, actions $A$ and rewards $R$ are finite, so the distribution of next state and reward, given the current state and action is well defined. Furthermore, the distribution of the random variables $R_t$ and $S_t$ follows the Markov property, depending only on the previous state and action, $P(S_{t+1}|S_t) = P(S_{t+1}|S_t, S_{t-1}, \ldots, S_0)$.

We can formalize mathematically an MDP using the following notation:

- $S$ is the set of states
- $A$ is the set of actions
- $R$ is the set of rewards
- $p(s', r|s, a)$ is the dynamics function, which describes the distribution of the next state $s'$ and reward $r$, given the current state $s$ and action $a$, defined as:

$$p(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a] \tag{1.1}$$

- $\gamma \in [0, 1]$ is the discount factor, which determines how much importance the agent gives to future rewards

### 1.2.1 Reward

The goal of the agent is formalized via a reward signal, given from the environment to the agent. At each time step the reward is a single number $R_t \in \mathbb{R}$. The agent goal is to maximize the total reward it receives over time. For example if we want the agent to play a game, the reward signal can be the number of points the agent has scored. If we want the agent to learn to walk, the reward signal can be the distance the agent has traveled.
The reward signal is not a way of telling the agent how to do, if we were to do that the agent will accomplish side tasks without ever completing the main task that we want to be solved. For example in a racing game we might be tempted to give a reward whenever the car moves forward, but this will lead the agent to move forward and backward in a loop, without ever finishing the race. In fact, in this case finishing the race is the worst thing that could happen, because then the agent will stop receiving the reward signal.
The reward signal thus specifies what we want the agent to do, not how to do it.

### 1.2.2 Episodic and Continuing Tasks

We can divide the kinds of problems that reinforcement learning deals with in two classes: episodic problems and continuing ones, which can be both formalized with MDPs 1.2.

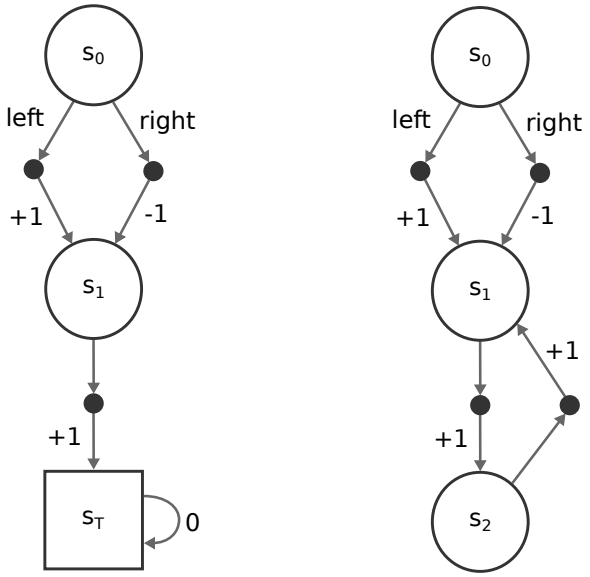


**Figure 1.2:** Empty circles represent the states, while solid circles the actions state-action pairs. The numbers indicate the reward the agent receives if it goes through the corresponding transition. Left: Illustration of an episodic task, the terminal state $s_T$ is represented with a square. If the agent reaches the terminal state it can no longer receive any reward. Right: Illustration of a continuing task, when the agent reaches $s_2$ it comes back to $s_1$, receiving a reward of $r = 1$ in the process, in this MPD the expected return with $\gamma = 1$ would be infinite as the sequence of rewards after the first state is always 1.

In episodic tasks the agent interacts with the environment for a finite (yet random) number of time steps $T$, and then it reaches a terminal state, ending the episode. Examples of episodic tasks is the game of chess, as the episode ends when a player wins or a draw is made.
In continuing tasks, on the other hand, there is no clear notion of episode, as the agent interacts with the environment continually, potentially for an infinite number of time steps.
We can formalize both problems with MDPs, by defining the concept of terminal state: a terminal state is an absorbing state and corresponds to the end of the episode, when the agent reaches the terminal state it can no longer escape it and all future rewards are zero.

### 1.2.3 Goal and Return

In reinforcement learning we formalize the goal of an agent as the maximization of the expected return $\mathbb{E}[G_t]$, where the return, denoted as $G_t$ is the sum of all discounted rewards the agent receives in the future, starting from a given time step $t$:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.2}$$

$\gamma \in [0, 1]$ is the discount factor and is used for two main reasons:

1. $\gamma$ allows tuning how much importance the agent gives to future rewards, so we can make the agent more or less patient. If $\gamma = 0$ the agent will only consider the immediate reward $R_{t+1}$, while if $\gamma = 1$ the agent will consider all future rewards with the same importance.

2. Allows the return to be used also in continuing tasks, because we can set $\gamma < 1$ and keep the sum from diverging.

Returns of subsequent time steps are related to each other in an important way:

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{1.3}$$

Equation (1.3) is also called the one-step return.

### 1.2.4 Policy and value functions

A policy $\pi$ is a mapping from states to actions, which specifies what action the agent should take in a given state. The policy is formally defined as the probability of taking action $a$ in state $s$

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{1.4}$$

Given a policy we can define a value function $v_\pi(s)$ associated to the policy, which gives us a measure of how good it is for the agent to be in a given state when following policy $\pi$. The value function is defined mathematically as the expected return starting from a state $s$ and then following policy $\pi$:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \tag{1.5}$$

We denote with $v_\pi(s)$ the value function of policy $\pi$.

Similarly, we can define the value of selecting an action $a$ in state $s$ under policy $\pi$, denoted $q_\pi(s, a)$, as the expected return starting from $s$, taking the action $a$ and then following policy $\pi$:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{1.6}$$

We call $q_\pi(s, a)$ the action-value function of policy $\pi$.

In practice, it is also convenient to define the advantage, which is the difference between the value of taking action $a$ in state $s$ and the value of the state:

$$A(s, a) = q_\pi(s, a) - v_\pi(s) \tag{1.7}$$

The advantage is a measure of how much better it is to take action $a$ in state $s$, with respect to the other possible actions in state $s$.

### 1.2.5 Bellman Equation

A fundamental property of the value function is that it satisfies a recursive relationship similar to the one of the return (1.3):

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \end{aligned} \tag{1.8}$$

We call equation (1.8) the Bellman equation for $v_\pi$. The Bellman equation expresses the relationship between the value of current state $s$ and the value of successive states $s'$. It tells us that the value of each state must equal the value of the next state plus the reward expected along the way. The Bellman equation forms the basis to compute or approximate $v_\pi$ which we will see in the next section.

It is also possible to define the Bellman equation for the action-value function:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)(r + \gamma v(s')) \\ &= \sum_{s',r} p(s', r|s, a)(r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')) \end{aligned} \tag{1.9}$$

Where in the last step we used the relationship $v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a)$ between the value function and action-value function.

### 1.2.6 Optimal policy and value function

Solving a reinforcement learning problem means finding a policy that achieves a high return. To accomplish this we first need to define an ordering over policies. We define that policy $\pi$ is better than policy $\pi'$ if the expected return is higher for $\pi$ than for $\pi'$.

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in S \tag{1.10}$$

This way we can define the optimal policy as the policy that is at least better or equal to all other policies.

$$\pi^* \geq \pi' \quad \forall \pi' \tag{1.11}$$

Although there may be more than one optimal policy, all of them will have the same value-function $v_*$, so we can denote all the optimal policies with $\pi^*$. We define the optimal state-value function $v_*$ as:

$$v_*(s) = \max_\pi v_\pi(s) \quad \forall s \in S \tag{1.12}$$

Which is the value function that achieves the maximum return from all states $s$. We can also define the optimal action-value function $q_*$ as:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \quad \forall s \in S, \forall a \in A \tag{1.13}$$

This function gives the expected return for taking action $a$ in state $s$ and then following policy $\pi^*$.

Another property of the optimal state-value function is that it satisfies a special form of the Bellman equation (1.8), called the Bellman optimality equation:

$$\begin{aligned} v_*(s) &= \max_a q_*(s,a) \\ &= \max_a \mathbb{E}_{\pi^*}[G_t | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)(r + \gamma v_*(s')) \end{aligned} \tag{1.14}$$

Note that in the final statement we didn't specify any policy, this is because we assume to be following an optimal policy, which is the one that always selects the best action $a_* = \arg\max'_a q_*(s,a') \; \forall s \in S$. Analogously, we can define the Bellman optimality equation for the action-value function:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}_{\pi^*}[G_t | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s',r|s,a)(r + \gamma \max_{a'} q_*(s',a')) \end{aligned} \tag{1.15}$$

Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state.
In the next sections we will see how to solve the Bellman optimality equation both when we have a model of the environment (1.1) and when we don't.

## 1.3 Dynamic Programming

Dynamic programming is a technique that allows us to solve the Bellman optimality equation (1.14), finding the optimal policy $\pi_*$ and value function $v_*$ given a perfect model of the environment, that is, we completely know the dynamics function $p(s', r|s, a)$. We can split the problem of finding a solution to the Bellman optimality equation into two subproblems:

1. **Policy evaluation**: Given a policy $\pi$, find its value function $v_\pi$.
2. **Policy improvement**: Given the value function $v_\pi$, find a better policy $\pi'$.

### 1.3.1 Policy evaluation

A simple way to evaluate a policy $\pi$ would be to solve the linear system of equations described by the Bellman equation (1.8). This could be done with any linear system solution method available in the literature, however, in reinforcement learning iterative solution methods are most suitable.

**Algorithm 1** Policy evaluation

**Require:** A policy $\pi$, a model $p$, a discount factor $\gamma$, a threshold $\theta$
Initialize a candidate value function $\hat{v}_\pi$
**while** True **do**
    $\hat{v}'_\pi = \hat{v}_\pi$
    **for** $s \in S$ **do**
        $\hat{v}'_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma \hat{v}_\pi(s'))$
    **end for**
    **if** $|\hat{v}'_\pi(s) - \hat{v}_\pi(s)| < \theta$ **then**
        **return** $\hat{v}_\pi$
    **end if**
**end while**

Algorithm 1 is called the iteative policy evaluation algorithm and is guaranteed to always converge to the true value function $v_\pi$ with an error below the threshold $\theta$ if $\gamma < 1$, or if $\gamma = 1$ only if termination is guaranteed from all states under the policy $\pi$.
This can be proven mathematically by noting that the operator $B_\pi(v_\pi) =$

$\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s'))$ is contracting, which means that the solution is both unique and is reached no matter how we initialize the value function $\hat{v}_\pi$.

In algorithm 1 we used an additional array to store the updated value function $\hat{v}'_\pi$, however, this is not required, as the algorithm will converge regardless of how we perform the updates, that is, we could have used only one array $\hat{v}_\pi$ and performed the update in line 5 in-place:
$\hat{v}_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma \hat{v}_\pi(s'))$
In practice, the in-place algorithm usually converges faster than the two-array version because it uses updated values as soon as they are available.

### 1.3.2 Policy improvement

Our reason for computing the value function for a policy is to help find better policies. A simple improved version of $\pi$ is a policy that in state $s'$ takes the action that maximizes $q_\pi(s', a)$. This policy will be equal to $\pi$ in all states $s \neq s'$ but will be better in state $s'$, so $\pi' \geq \pi$.
We can take this idea further, by defining $\pi'$ as the policy that takes the greedy action with respect to $q_\pi(s, a)$ in all states, with ties broken evenly. This policy is called the greedy policy and is defined as:

$$\pi'(a|s) = \arg\max_a q_\pi(s, a) = \arg\max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \quad (1.16)$$

The greedy policy always takes the action that looks best in the short term with just one-step look ahead, according to $v_\pi$. It can be shown that the greedy policy is always better or equal to $\pi$ due to the policy improvement theorem, demonstrated on page 78 of the Sutton and Barto's book [1].

### 1.3.3 Policy iteration

A simple way to find the optimal policy is to iterate the policy evaluation and policy improvement steps until the policy doesn't change anymore. This process is guaranteed to converge to the optimal policy and value function with the same conditions that apply to the policy evaluation algorithm 1, that is: convergence is guaranteed to the true value function $v_\pi$ with an error below the threshold $\theta$ if $\gamma < 1$, or if $\gamma = 1$ and the task is episodic.

**Algorithm 2** Policy iteration

**Require:** A model $p$, a discount factor $\gamma$, a threshold $\theta$

Initialize a policy $\pi$
**while** True **do**
    $v_\pi = \text{Policy evaluation}(\pi, p, \gamma, \theta)$ ▷ Algorithm 1
    $\pi' = \arg\max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s'))$ ▷ Policy improvement
    **if** $\pi' = \pi$ **then**
        **return** $\pi, v_\pi$
    **end if**
**end while**

Algorithm 2 is called the policy iteration algorithm. The algorithm iterates the policy evaluation and policy improvement steps until the policy doesn't change anymore.

### 1.3.4 Value iteration

The drawback of the policy iteration algorithm 2 is that it requires an entire policy evaluation for each policy improvement step, requiring multiple sweeps over the state space. In practice, however, we can stop before policy evaluation finished without losing any of the convergence properties of policy iteration 2. One important special case is when policy evaluation is stopped after just one sweep, using the update:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_k(s')) \tag{1.17}$$

This update is called the value iteration update and combines one pass of policy evaluation and policy improvement into one, requiring only 1 sweep over the state space. The update (1.17) is similar to the policy evaluation update (1.8) except that we are using the maximum instead of the expectation.

**Algorithm 3** Value iteration

**Require:** A model $p$, a discount factor $\gamma$, a threshold $\theta$

Initialize a value function $v_0$
**while** True **do**
    $v_{k+1} = v_k$
    **for** $s \in S$ **do**
        $v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_k(s'))$
    **end for**
    **if** $|v_{k+1}(s) - v_k(s)| < \theta$ **then**
        **return** $v_{k+1}$
    **end if**
**end while**

Algorithm 3 usually converges faster than policy iteration 2. Furthermore, like the policy iteration algorithm, this algorithm converges even if we perform all updates in place, without needing to keep in memory both $v_{k+1}$ and $v_k$, thus reducing memory requirements.

The value iteration algorithm 3 has a limitation in that it necessitates a complete examination of the entire state space, which can be an issue if the state space is large. However, the algorithm is still guaranteed to converge to the optimal value function even if updates are performed on certain states more frequently than others, as long as we continue to update all the states. These algorithms are called asynchronous DP algorithms because they update the values of the states in asynchronous fashion.

### 1.3.5 Generalized policy iteration

We can view both the policy iteration algorithm 2 and the value iteration algorithm 3 as special cases of a more general process referred to as generalized policy iteration (GPI). Almost all RL algorithms are described by this process where the policy is improved towards the value function and vice-versa, as shown in figure 1.3. This loop ends only when we reach a policy that is greedy with respect to its own value function, which implies that the value function satisfies the Bellman optimality equation (1.14).

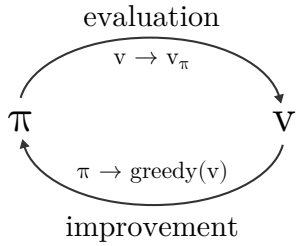


**Figure 1.3:** Generalized policy iteration loop. Image adapted from [1].

Another way to see the process is in terms of two different goals, the two converging lines represented in figure 1.4. The first goal is to compute the value function of a policy $v = v_\pi$, which is achieved by policy evaluation. The second goal is to make the policy greedy with respect to its own value function $\pi = \text{greedy}(v)$, which is achieved by policy improvement. The two goals are complementary and the process converges to the optimal value function when the two goals are achieved simultaneously.

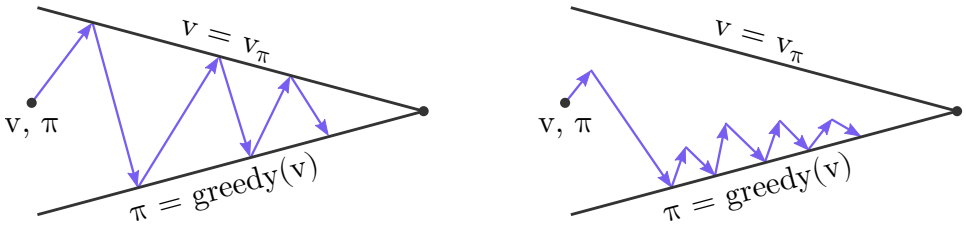


**Figure 1.4:** Another way to see generalized policy iteration. Left: policy iteration with full policy evaluation at each step. Right: policy iteration with truncated policy evaluation at each step. Image adapted from [1].

## 1.4 Monte Carlo methods

Monte Carlo methods are a class of algorithms that learn directly from experience, without the need for a model of the environment. The main idea is to estimate the value function $v_\pi$ by averaging the returns $G_t$ following each state. These methods can only be applied to episodic tasks because we update the value function using $G_t$, which is only available after an episode has finished.
As with dynamic programming we can split the problem of finding the optimal policy into two parts.

1. Monte Carlo prediction: estimate the value function $v_\pi$ corresponding to a given policy $\pi$
2. Monte Carlo control: find the optimal policy $\pi^*$

### 1.4.1 Monte Carlo prediction

Suppose we have a policy $\pi$, and we want to estimate the value function $v_\pi$ given a set of episodes obtained by following policy $\pi$, and passing through state $s$ at time $t$. We call each occurrence of state $s$ in an episode a visit to $s$. Of course, $s$ may be visited multiple times in the same episode, so we can call the first time $s$ is visited in an episode the first visit to $s$. We can then identify two main methods to perform Monte Carlo policy evaluation:

1. First-visit MC method: estimate $v_\pi(s)$ as the average of the returns following first visits to $s$.
2. Every-visit MC method: estimate $v_\pi(s)$ as the average of the returns following all visits to $s$.

For example, in the first visit case we estimate the value function $v_\pi$ as follows:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi(G_t|S_t = s) \\ &= \frac{1}{N}\sum_{i=1}^{N} G_t^{(i)} \\ &= \frac{1}{N}\sum_{i=1}^{N}\sum_{t'=t}^{T} \gamma^{t'} R_{t'+1}^{(i)} \end{aligned} \tag{1.18}$$

Where $G_t^{(i)}$ is the return after visiting state $s$ in the $i$-th episode, $N$ is the number of episodes and $T$ is the episode length.
As the number of visits to state $s$ increases, both first-visit and every-visit Monte Carlo methods converge to $v_\pi(s)$. However, the two methods have different theoretical characteristics. First-visit MC provides independent and identically distributed estimates of $v_\pi(s)$, whereas every-visit MC does not. Despite its less straightforward nature, every-visit MC still maintains the same convergence properties as first-visit MC.

Monte Carlo methods can be utilized to estimate the action-value function $q_\pi(s, a)$ as well, which is critical for control tasks. The concept of visits is redefined to apply to state-action pairs, rather than states alone. A state-action pair $(s, a)$ is considered visited if action $a$ was selected while in state $s$.

### 1.4.2 Monte Carlo control

In the control problem, MC methods draw from the idea of generalized policy iteration (GPI) already presented: alternating between policy evaluation and policy improvement.
In the policy evaluation step we estimate the action-value function $q_\pi$ using the first visit or every visit MC method. However, we need to ensure that all states will be visited, to accomplish this we can generate the episodes with exploring starts, that is, all episodes begin with state-action pairs randomly selected to cover all possibilities.
The policy improvement step, on the other hand, is performed by greedily selecting the action that maximizes the action-value function (1.19), which is the same as selecting the action that maximizes the expected return.

$$\pi'(s) = \arg\max_a q_\pi(s, a) \tag{1.19}$$

A major problem with this approach is the exploring starts assumption, because, if we are not given a model of the environment, we cannot know all state-action pairs beforehand, and even if we did, we would still need

to generate many episodes to ensure that all state-action pairs are visited. Without exploring starts, the only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them. This can be done in two ways:

1. We can use a different policy to generate the episodes, a behavioral policy, different from the one we are trying to improve, the target policy. This is called off-policy learning.

2. We can avoid making the policy completely greedy like in equation (1.19) and instead use an $\epsilon$-soft policy, which is a policy that is not completely greedy, but gives at least probability $\epsilon/|A|$ to each action. An example of such policy is the $\epsilon$-greedy policy, defined as follows:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A|} & \text{if } a = \text{argmax}_{a'} q_\pi(s, a') \\ \frac{\epsilon}{|A|} & \text{otherwise} \end{cases} \tag{1.20}$$

## 1.5 Temporal-Difference learning

Temporal-difference learning, also called TD-Learning, is a class of algorithms that combine both Monte Carlo and dynamic programming ideas. Like Monte Carlo, TD-learning methods learn directly from experience, not requiring a model of the environment, and like DP it updates the value function with already learned estimates, using a technique known as bootstrapping. This allows TD methods to be also applied to continuing tasks because they can update the value of a state as soon as new information is available.
Like we did with Monte Carlo methods, we first focus on the prediction problem, then the control one.

### 1.5.1 TD prediction

TD prediction uses the one-step return, represented as $G_{t:t+1} = R_{t+1} + \gamma G_{t+1}$ (as defined in equation (1.3)), rather than the full return (defined in equation (1.2)) to estimate the value function. To obtain the expected return, which is needed to estimate the value function, we take the expected value of the recursive formula.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi(G_t | S_t = s) \\ &= \mathbb{E}_\pi(R_{t+1} + \gamma G_{t+1} | S_t = s) \end{aligned} \tag{1.21}$$

This expectation (1.21) can be directly estimated by sampling episodes according to the policy $\pi$. The value function $v_\pi$ is then updated as follows:

$$V(S_t) = V(S_t) + \alpha(R_t + V(S_t) - V(S_{t+1})) \tag{1.22}$$

Where we introduced a parameter $\alpha$ which is the learning rate of the algorithm, the learning rate controls how much the new estimate of the value gathered in the episode should count with respect to the previous value estimate $V(S_t)$.
Finally, the quantity in parentheses $\delta = R_t + V(S_t) - V(S_{t+1})$ is known as the TD error, and measures the difference between the estimated value of the current state $V(S_t)$ and the better estimate $R_t + V(S_{t+1})$, given by the episode.
Updating the value function in this way (1.22) introduces bias in the value estimate, however, this also reduces the variance, which is why TD methods usually learn faster Monte Carlo methods.

### 1.5.2 TD control

In TD control we need to estimate the action-value function $q_\pi(s, a)$, rather than a state-value function. There are three main methods to do so, Sarsa, Expected Sarsa and Q-learning.

#### Sarsa

Sarsa is the simplest of the three, it is an on-policy TD control method that uses the following update rule to estimate the action-value function:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{1.23}$$

The method is called Sarsa because in state $S_t$ it selects action $A_t$, then observes reward $R_{t+1}$ and next state $S_{t+1}$, finally it selects the next action $A_{t+1}$.
In Sarsa $Q(S_{t+1}, A_{t+1})$ is a sample of the value of the next state $V(S_{t+1})$, because the next action $A_{t+1}$ is selected according to policy $\pi$.

$$\begin{aligned} v_\pi(s') &= \sum_a q(s', a)\pi(a|s') \\ &= \mathbb{E}_\pi(Q(S_{t+1}, A_{t+1})|S_{t+1} = s') \end{aligned}$$

#### Expected Sarsa

Expected Sarsa, estimates the value of the next state by using directly its expected value $V(S_{t+1})$. The update rule of expected Sarsa is thus the following.

$$\begin{aligned} Q(S_t, A_t) &= Q(S_t, A_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - Q(S_t, A_t)) \\ &= Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \sum_a Q(S_{t+1}, a)\pi(a|S_{t+1}) - Q(S_t, A_t)) \end{aligned} \tag{1.24}$$

This update reduces the variance of the value estimate, but it also increases its computational cost, because to compute the value of the next state we must sum over all actions, $V(S_{t+1}) = \sum_a Q(S_{t+1}, a)\pi(a|S_{t+1})$, which can be computationally expensive when the number of actions is high.

### Q-learning

Q-learning is an off-policy TD control method, that uses the following update rule to estimate the action-value function:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)) \quad (1.25)$$

Q-learning assumes that the next action $A_{t+1}$ is selected according to a greedy policy with respect to the action-value function $Q(S_{t+1}, a)$. In this algorithm, the learned action-value function directly approximates the optimal action-value function $q_*$, independently of the policy being followed. However, the policy $\pi$ still has an important effect because it determines which state-action pairs are visited and is thus called the behavioral policy.

## 1.6 $n$-step methods

$n$-step methods merge the gap between MC and TD methods, so that one can shift from one to the other smoothly. This is important because the best method for a given problem is often intermediate between these two extremes figure 1.5.

$n$-step methods estimate the value function using the $n$-step return, which is the sum of the rewards obtained in the next $n$ steps, plus the value of the state reached after $n$ steps.

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) \quad (1.26)$$

Where the subscript $t : t + n$ indicates that we consider the rewards up to time $t + n$ and approximate subsequent rewards with the value of the state $S_{t+n}$.
We then update the value function as follows:

$$V(S_t) = V(S_t) + \alpha(G_{t:t+n} - V(S_t)) \quad (1.27)$$

If $n$ is greater than the episode length $T$, then the $n$-step return is equal to the full return, and we have Monte Carlo prediction. While, if $n$ is equal to 1, then we have TD prediction.

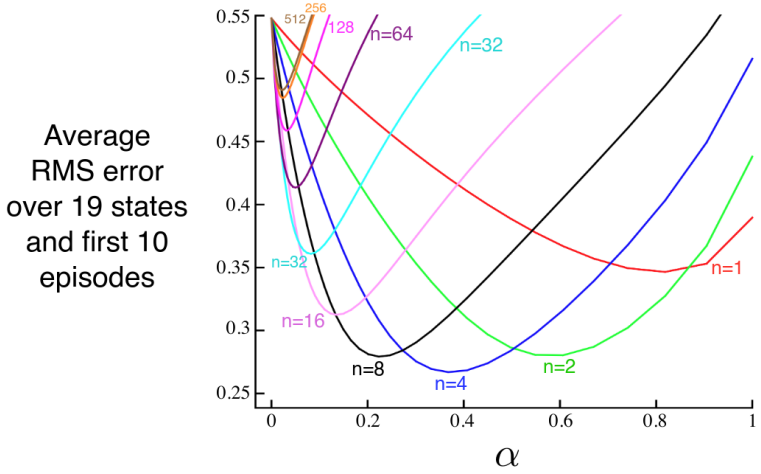


**Figure 1.5:** Comparison between $n$-step methods as a function of $\alpha$ for various values of $n$ in a random-walk task. An intermediate value of $n = 4$ performs best, converging faster than both TD methods ($n = 1$), and MC methods ($n = 512$). Image taken from [1].

It can be shown that all $n$-step TD methods converge to the correct value function $v_\pi$ under appropriate technical conditions. The $n$-step TD methods hence form a family of sound methods, with one-step TD methods and Monte Carlo methods as extreme members.

For control problems, we can generalize one-step TD methods to their respective $n$-step versions by defining the $n$-step return in terms of estimated action values.

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n}) \tag{1.28}$$

The update rule of $n$-step Sarsa is then the following:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(G_{t:t+n} - Q(S_t, A_t)) \tag{1.29}$$

In essence, a new TD control algorithm can be constructed by incorporating a formula for the return into Equation (1.29).
For example, we can define $n$-step Expected Sarsa by defining the $n$-step return in terms of the expected value of state $S_{t+n}$.

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \sum_a Q(S_{t+n}, a)\pi(a|S_{t+n}) \tag{1.30}$$

And $n$-step Q-learning by defining the $n$-step return in terms of the maximum value of state $S_{t+n}$.

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \max_a Q(S_{t+n}, a) \tag{1.31}$$

It can be shown that all $n$-step methods converge to the optimal value function if the leaning and that the convergence rate is independent of $n$.

## 1.7 Discussion

In this chapter we introduced the main concepts of reinforcement learning, we saw that reinforcement learning is a framework for learning in sequential decision-making problems, where the agent interacts with the environment and learns to maximize its reward.
We then presented Monte Carlo methods, which are a class of algorithms that use the full return (1.2) to estimate the value function, and TD methods, which use the one-step return (1.3). TD methods are usually better than MC methods because they can be applied to continuing tasks, and reduce the variance of the value estimate.
Finally, we discussed $n$-step methods, which estimate the value function using the $n$-step return formula (1.26). These methods have been proven to converge to the correct value function $v_\pi$ under specific conditions and represent a family of reliable techniques, with one-step TD methods and Monte Carlo methods being their two ends of the spectrum.

# Chapter 2

# Policy Gradient Methods

So far we focused on action-value methods, those methods learned the action-value function $q$ and used it to derive a policy. In this chapter, we will focus on methods that learn the policy directly, without consulting a value, or action-value function. The value function still has an important role as it's often used to learn the policy, however, it is not required for action selection. This class of methods is called policy gradient methods.

Policy gradient methods can be applied without function approximation. However, in this discussion, we will focus on the function approximation setting, where the policy is expressed as a function of the state $s$ and a vector of parameters $\boldsymbol{\theta}$, expressed as $\pi(a|s;\boldsymbol{\theta})$.

We will begin by covering the basics of function approximation, specifically neural networks, which will serve as the foundation for all the algorithms discussed in this chapter. We will then delve into the policy gradient theorem for discrete action spaces and introduce REINFORCE, a Monte Carlo policy gradient algorithm.
Our main focus will be on actor-critic methods, which have achieved significant success in various domains, such as playing games like Dota 2 [2], StarCraft II [3], Go [4], Minecraft [5, 6] with also applications to language like ChatGPT [7]. We will not provide an exhaustive overview of actor-critic methods, but we will discuss key ones, such as TRPO and PPO. The latter will also be used to in the subsequent chapter to learn to play Briscola.

## 2.1 Function Approximation

The methods discussed so far represent the action-value function $q(s,a)$ as a table and are thus called tabular methods. These methods have been shown to converge to an optimal policy under certain conditions, such as proper selection of the step-size $\alpha$ and the policy $\epsilon$ parameter. However, tabular

methods can be impractical to use in certain scenarios, such as when the state space is extremely large (e.g. in the game of chess) or when the state space is continuous. In these cases, function approximation must be used to represent the value function, action-value function, and policy.

To keep things clear, we will focus on approximating the value function, denoted as $\hat{v}(s; \boldsymbol{w})$, where $\boldsymbol{w}$ is a parameter vector. The same ideas can also be applied to the action-value function or the policy. Furthermore, we assume to perform function approximation with neural networks, as they are most commonly used in practice and are differentiable function approximation methods, which is required by the policy gradient theorem.

Consider the problem of finding the value function corresponding to a policy $\pi$, thus far we implemented the update rule for the value function by moving the value prediction $V(s)$ closer to the return $G_t$, which represents the target:

$$V(s) = V(s) + \alpha(G_t - V(s)) \tag{2.1}$$

With function approximation the idea is the same, moving the prediction closer to the target. But instead of updating the value of the table entry $V(s)$ directly, we need to update the parameter vector $\boldsymbol{w}$. The way we usually update the parameter vector $\boldsymbol{w}$ in neural networks is with gradient descent, that is, we update $\boldsymbol{w}$ in the negative direction of the gradient of a loss function $L$.

$$\boldsymbol{w} = \boldsymbol{w} - \alpha \nabla_w L(\hat{v}(s; \boldsymbol{w}), G_t) \tag{2.2}$$

The loss function measures how far the value prediction $\hat{v}(s; \boldsymbol{w})$ is from the target $G_t$. A natural loss function for learning the value is the *mean squared error loss* denoted as $L_{MSE}$:

$$L_{MSE} = \frac{1}{2}(G_t - \hat{v}(S_t; \boldsymbol{w}))^2 \tag{2.3}$$

The gradient of the loss function (2.3) with respect to $\boldsymbol{w}$, can be quickly computed:

$$\nabla_w \text{MSE} = (\hat{v}(S_t; \boldsymbol{w}) - G_t)\, \nabla_w \hat{v}(S_t; \boldsymbol{w}) \tag{2.4}$$

Where $\nabla_w \hat{v}(s; \boldsymbol{w})$ is the gradient of the output of the function with respect to the parameter vector $\boldsymbol{w}$ and can be quickly found with automatic differentiation techniques.

We can view tabular methods as special case of function approximation where we have one parameter for each state, representing the value of that state. In this setting, the gradient of the loss function is simply the one-hot vector corresponding to state $S_t$:

$$\nabla_w \text{MSE} = (\hat{v}(S_t; \boldsymbol{w}) - G_t)\, \delta_{S_t} \tag{2.5}$$

Where the one-hot vector $\delta_{S_t}$ is a vector of zeros with a one in the position corresponding to the state $S_t$. The update rule (2.2) then becomes:

$$\boldsymbol{w} = \boldsymbol{w} - \alpha \left(\hat{v}(S_t; \boldsymbol{w}) - G_t\right) \delta_{S_t} \tag{2.6}$$

Which is exactly the update rule (2.1) for tabular methods. Function approximation is hence a generalization of tabular methods, where the parameter vector $\boldsymbol{w}$ replaces the table of values.

Note that in the gradient derivation (2.4) we assumed $\nabla_w G_t = 0$, while this is true for Monte Carlo methods, where $G_t = R_{t+1} + \gamma R_{t+2} + \cdots + R_T$, it is not in general true for TD methods, where the one-step return is used $G_t = R_{t+1} + \gamma \hat{v}(S_{t+1}; \boldsymbol{w})$. In this case, the gradient is not a true-gradient but a semi-gradient, because we left out the $\nabla_w G_t = \gamma \nabla_w \hat{v}(S_{t+1}; \boldsymbol{w})$ term. It can be shown that semi-gradient methods have better convergence properties than true-gradient ones, in particular true-gradient methods even in the tabular case won't converge to the true value function $v_\pi$ in stochastic environments. The interested reader can find an example of this behavior in chapter 11.5 of the Sutton and Barto book [1].

### 2.1.1 Policy Approximation

In policy gradient methods, the policy can be parameterized in any way, as long as it is differentiable with respect to the parameter vector $\boldsymbol{\theta}$. Approximating the policy directly has many advantages over approximating the action-value function:

- The approximate policy can approach a deterministic policy, whereas with $\epsilon$-greedy action selection the policy always has a probability of selecting a random action.
- It enables the selection of actions with arbitrary probabilities, for example, in imperfect information games performing always the same action is not optimal, because the opponent can learn the player's strategy and exploit it. Think about the game of poker, the best strategy isn't to always bluff or always check, but to do a mix of both, so that the opponent can't predict the player's behavior.
- The action probabilities change smoothly with the parameter vector $\boldsymbol{\theta}$, whereas with $\epsilon$-greedy action selection the probability can change dramatically for a small change in the action-value function.
- Because of the previous point, stronger convergence guarantees are available for policy-gradient methods.

## 2.2 Policy Gradient Theorem

To understand the policy gradient theorem, we need to define a performance measure of a policy $\pi$ with respect to a parameter vector $\boldsymbol{\theta}$:

$$J(\boldsymbol{\theta}) = v_{\pi_\theta}(s_0) = \sum_a \pi_\theta(a|s_0) q_{\pi_\theta}(s_0, a) \tag{2.7}$$

Where $v_{\pi_\theta}$ represents the true value function for the policy $\pi_\theta$, determined by the parameter vector $\boldsymbol{\theta}$.

The problem with equation (2.7) is that its gradient depends also on the gradient of the action-value function $q_{\pi_\theta}$, which in turn depends on the state distribution $\mu(s)$ which is unknown. Fortunately $\nabla_\theta J(\theta)$ can still be computed via the policy gradient theorem, proven in chapter 13.2 of the Sutton and Barto book [1]:

$$\begin{aligned} \nabla_\theta J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla_\theta \pi_\theta(a|S_t) \\ &= \mathbb{E}_{\pi_\theta}\left[ \sum_a q_\pi(S_t, a) \nabla_\theta \pi_\theta(a|S_t) \right] \end{aligned} \tag{2.8}$$

The proof relies on approximating the state distribution $\mu(s)$ with the empirical distribution of states visited during policy evaluation. This enables us to estimate the gradient of the performance measure (2.7) with respect to the policy parameters without needing the derivative of the state distribution.

The policy gradient theorem is a fundamental result in reinforcement learning that gives a way to compute the gradient of the performance measure with respect to the policy parameters.
We can further simplify expression (2.8) by multiplying and dividing by $\pi_\theta(a|S_t)$, removing the internal sum over actions:

$$\begin{aligned} \nabla_\theta J(\boldsymbol{\theta}) &\propto \mathbb{E}_{\pi_\theta}\left[ \sum_a q_\pi(S_t, a) \nabla_\theta \pi_\theta(a|S_t) \right] \\ &= \mathbb{E}_{\pi_\theta}\left[ q_\pi(S_t, A_t) \frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \right] \\ &= \mathbb{E}_{\pi_\theta}\left[ q_\pi(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t|S_t) \right] \end{aligned} \tag{2.9}$$

Where in the last step we used the equality $\nabla_\theta \pi_\theta(a|s)/\pi_\theta(a|s) = \nabla_\theta \log \pi_\theta(a|s)$, which is true for any differentiable function $\pi_\theta(a|s)$.
This is the most commonly used form of the policy gradient theorem and allows the computation of the policy gradient even with continuous action spaces.

## 2.3 REINFORCE

REINFORCE estimates the action-value function $q_{\pi_\theta}(S_t, A_t)$ of the policy gradient (2.9) using Monte Carlo.

$$q_\pi(S_t, A_t) = \sum_{t'=t}^{T} \gamma^{t'-t} R_{t'+1} = G_t \tag{2.10}$$

Thus the update rule for the policy parameter vector $\boldsymbol{\theta}$ is:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \tag{2.11}$$

This update increases the probability of actions proportional to the return and inversely proportional to the probability of the action. This makes sense because we move towards actions that lead to a higher return and perform a bigger update for actions with a lower probability, which are selected and hence updated less often.

### 2.3.1 REINFORCE with Baseline

The policy gradient theorem (2.8) can be generalized by adding a baseline function $b(s)$ to the action-value function without loss of generality:

$$\begin{aligned}
&\sum_s \mu(s) \sum_a (q_\pi(s,a) - b(s)) \nabla_\theta \pi_\theta(a|S_t) = \\
&= \sum_s \mu(s) \sum_a (q_\pi(s,a)) \nabla_\theta \pi_\theta(a|S_t) - \sum_s \mu(s) b(s) \sum_a \nabla_\theta \pi_\theta(a|S_t) \\
&= \nabla_\theta J(\boldsymbol{\theta}) - \sum_s \mu(s) b(s) \nabla_\theta \sum_a \pi_\theta(a|s) \\
&= \nabla_\theta J(\boldsymbol{\theta})
\end{aligned} \tag{2.12}$$

Where in the last step we used the fact that $\nabla_\theta \sum_a \pi_\theta(a|s) = 0$. The baseline function $b(s)$ can be any function of the state $s$ and is used to reduce the variance of the gradient estimator. However, one natural choice for the baseline function is the value function $v_{\pi_\theta}(s)$, which gives the REINFORCE with baseline update:

$$\theta_{t+1} = \theta_t + \alpha (G_t - \hat{v}_{\pi_\theta}(S_t)) \frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \tag{2.13}$$

Where $\hat{v}_{\pi_\theta}(S_t)$ is estimated with Monte Carlo. The improvement of the REINFORCE with baseline algorithm over the REINFORCE algorithm is shown in Figure 2.1.

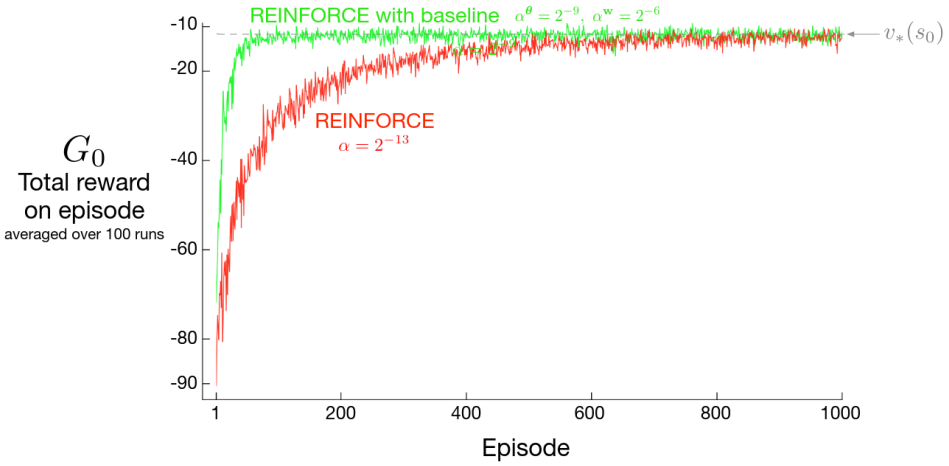


**Figure 2.1:** Comparison of the REINFORCE and REINFORCE with baseline algorithms on the short-corridor gridworld, with the optimal step sizes indicated. The results show that the baseline algorithm outperforms the REINFORCE algorithm, reaching the best return in just 100 steps, while the REINFORCE algorithm takes roughly 1000 steps. Image taken from [1].

## 2.4 Actor-Critic Methods

Actor-Critic methods merge policy gradient and temporal difference methods. The policy gradient is used to update the policy parameter vector $\boldsymbol{\theta}$ and TD-learning is used to update the value function $v_{\pi_\theta}(s)$. This introduces a bias in the policy gradient update, however it also reduces variance and hence accelerates learning.
One-step actor-critic methods updates the policy as following:

$$\theta_{t+1} = \theta_t + \alpha(R_{t+1} + \gamma\hat{v}_{\pi_\theta}(S_{t+1}; \boldsymbol{w}) - \hat{v}_{\pi_\theta}(S_t; \boldsymbol{w}))\frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \tag{2.14}$$

Which is similar to the REINFORCE with baseline update (2.13), but uses the one-step TD target $G_{t:t+1} = R_{t+1} + \gamma\hat{v}(S_{t+1}; \boldsymbol{w})$ instead of the full return $G_t$.
At the same time the value function is updated using the TD error:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha(R_{t+1} + \gamma\hat{v}_{\pi_\theta}(S_{t+1}; \boldsymbol{w}) - \hat{v}_{\pi_\theta}(S_t; \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{v}_{\pi_\theta}(S_t; \boldsymbol{w}) \tag{2.15}$$

These methods are called actor-critic methods because the policy is called the actor and the value function is called the critic. The critic learns the value function and the actor, given the value function, learns the policy.

## 2.5 Trust Region Policy Optimization

# Bibliography


[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd edition. MIT Press, 2018.

[2] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning," *CoRR*, vol. abs/1912.06680, 2019. arXiv: 1912.06680. [Online]. Available: http://arxiv.org/abs/1912.06680.

[3] O. Vinyals, I. Babuschkin, W. Czarnecki, and et al., "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, pp. 350–354, 2019. DOI: https://doi.org/10.1038/s41586-019-1724-z.

[4] D. Silver, A. Huang, C. Maddison, and et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 2016. DOI: https://doi.org/10.1038/nature16961.

[5] B. Baker, I. Akkaya, P. Zhokhov, J. Huizinga, J. Tang, A. Ecoffet, B. Houghton, R. Sampedro, and J. Clune, *Video pretraining (vpt): Learning to act by watching unlabeled online videos*, 2022. DOI: 10.48550/ARXIV.2206.11795. [Online]. Available: https://arxiv.org/abs/2206.11795.

[6] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap, *Mastering diverse domains through world models*, 2023. DOI: 10.48550/ARXIV.2301.04104. [Online]. Available: https://arxiv.org/abs/2301.04104.

[7] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, *Training language models to follow instructions with human feedback*, 2022. DOI: 10.48550/ARXIV.2203.02155. [Online]. Available: https://arxiv.org/abs/2203.02155.