A Comprehensive Guide on Learning Step-by-Step: Q-Learning and Deep Q-Learning

German Nova

Abstract

This tutorial paper provides an accessible introduction to the fundamental concepts of Reinforcement Learning, with a focus on Q-learning and its extension into Deep Q-Learning. Reinforcement Learning is a branch of Machine Learning that focuses on optimal sequential decision-making. This tutorial starts by introducing the basic concepts of Reinforcement Learning, such as states, actions, rewards, and policies. Then, Q-learning is introduced. This algorithm is known for its step-by-step update process for estimating action-values. Building upon Q-learning's principles, the next topic of the tutorial is Deep Q-learning, which employs Deep Learning to approximate the Q-function and handle high-dimensional state spaces. For both Q-learning and Deep Q-learning, illustrative examples of code implementation in Python are presented. Lastly, useful references are recommended for further reading.

Contents

1 Introduction

Reinforcement Learning offers a unique framework for training an agent in an environment, so that it can optimize its decision-making. With the introduction of Deep Learning, Deep Reinforcement Learning algorithms have attracted significant attention due to their potential to autonomously tackle complex, real-world problems. In this tutorial paper, an accessible introduction to its fundamental concepts is presented. Both Q-learning and Deep Q-learning algorithms, which are among the most relevant in Reinforcement Learning, are also explored. Subsequently, a discussion of their strengths and weaknesses, along with references for further study, is provided.

2 The Basics of Reinforcement Learning

2.1 Sequential Decision-Making

Reinforcement Learning is a branch of Machine Learning that focuses on optimal **sequential decision-making**. The entity taking **actions**, is called the **agent**. It resides in an **environment** from which it can obtain information about the current situation, known as the **state**, and observe the **reward** and **next state** it obtains for performing actions under this current state. The objective of the agent is to maximize the reward. However, the reward is not seen solely in terms of immediate utility but under the consideration that an action the agent takes will affect the future, and consequently its future reward. This interaction at each time step $t \in [0, T]$ between the agent and the environment is summarized in the following graph:

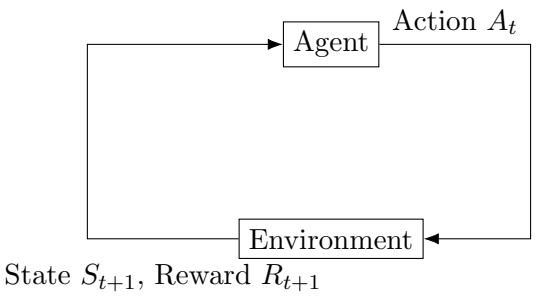


Figure 1: Interaction between Agent and Environment.

With the agent's objective being to maximize the future reward:

$$R_{t+1} + R_{t+2} + \cdots + R_T \tag{1}$$

This future reward, however, can follow stochastic dynamics, so we say that the agent's objective is to maximize the expectation of the future reward, or the **expected future reward**:

$$\mathbb{E}[R_{t+1} + R_{t+2} + \cdots + R_T] \tag{2}$$

Notice that the decision-making process of the agent is sequential, in the sense that there is an initial state S_0 that the agent observes, and then, takes an action A_0. Next, it observes its reward R_1 and the next state S_1. Then the same process repeats under the new current state S_1. This sequential interaction with the environment can be summarized by a **Markov Decision Process**. This process has an important property called the **Markov property**,

which states that the current state contains all the information that the agent can use to make decisions about the future.

2.2 Discounting Future Rewards

Another aspect to consider is the case of maximizing the expected future reward indefinitely. For this, we need to introduce a concept known as **discounting**, which aims to capture the idea that when we anticipate future rewards, we typically prioritize those that are closer in time. It is a concept commonly applied to human behavior. The extent of priority an individual assigns to rewards as they become more distant from the present varies. An impatient individual may assign significantly more priority to immediate rewards, whereas someone with a future-oriented mindset might consider rewards that take time as equally valuable as immediate ones. This notion can be expressed mathematically in a straightforward manner.

We use a fixed parameter $\gamma \in [0, 1)$ in the following form:

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

The smaller the value of γ, the more impatient the agent will be. It can be shown that this sum is finite. Therefore, by adding the concept of discounting, we are able to view the expected future return, which extends indefinitely, as a finite sum.

Exercise 1. Given that the reward at each step is finite, show that the sum in (3) is finite.

2.3 Policies

Now, the agent must determine how to act. Essentially, it needs to find a way to optimize its strategy, known as a **policy** π, so that it maximizes its expected future reward, considering the impact of actions on its situation many steps ahead. To understand how to find this optimal policy π^*, we must consider that initially, the agent lacks knowledge about which action to take. Then, it must **explore** by trying different actions and evaluating which one performs better given the current state. After it has gathered sufficient information, it can then **exploit** the best actions for the given state.

So far, we have seen the agent's policy as the strategy that the agent must follow given the current state it's observing. The Markov property also has a repercussion on these policies, in the sense that they cannot be defined as something that depends on past decisions. For example, any strategy that alternates between actions or acts conditionally on past actions does not make sense, as the agent must build its strategy based on the current state. So, what should a policy look like?

As the agent acts based on the observed current state, it follows that given this observed state, the policy must return an action. We have two types of policies. One called **deterministic policy**, which is a strategy that relates a state solely to one action, although many states can return the same action. The other type is a **stochastic policy**. This relates a state to a set of probabilities assigned to each possible action. Notice that the deterministic policy is a special case of the stochastic policy.

Exercise 2. Answer if each of the following policies satisfies the Markov Property. In case its a valid policy, determine its type of policy.

- If the robot has jumped 10 times in a row, then it must crawl.
- If the floor is red, the robot must jump. If the floor is blue, the robot must crawl.
- The robot throws a dice, and jumps if it gets an odd number; otherwise, it crawls.

2.4 Value Functions

The next step is to understand how an agent can find an optimal policy that maximizes its expected future reward. First, we must recognize that the agent observes its current state and firmly aims to maximize the expected future reward. Thus, its useful for the agent to compare different policies. Given this, we can infer that the agent would be highly interested in estimating the expected future reward given its current state following a certain policy. This conditional expectation is called **state-value function**, expressed as:

$$V_\pi(S) = \mathbb{E}_\pi[\sum_{i=0}^{\infty} \gamma^i R_{t+1+i} | S_t = S] \tag{4}$$

For its purpose, the agent might also find it useful to estimate the expected future reward given its current state and a certain action it desires to take following a certain policy. This conditional expectation is called **action-value function**, which we express as:

$$Q_\pi(S, A) = \mathbb{E}_\pi[\sum_{i=0}^{\infty} \gamma^i R_{t+1+i} | S_t = S, A_t = A] \tag{5}$$

Recall that an optimal policy π^* is the one that maximizes the expected future reward. So ideally, we would like to estimate this maximization, as it will allow us to retrieve the optimal policy. For this to be possible, we must define two main tasks that can help our agent fulfill its main objective. The first task is to evaluate our policies, or **policy evaluation**, so that we can retrieve the value function for each policy we want to test. The second task, called **policy control**, is to improve these policies so that we can help the agent maximize the expected future reward.

In general, a Reinforcement Learning algorithm needs to execute a policy under a certain state to observe its reward and next state. This is called the **behavior policy**. Then the agent uses this information to improve its policy, called the **target policy**. We also have two different ways of doing policy evaluation and control. The **off-policy algorithms** intend to have separate behavior and target policies. In this case, the behavior policy is used to select the actions the agent will perform, while the target policy is the one that is being improved and evaluated. In contrast, in the **on-policy algorithms**, the behavior policy is the same as the target policy. So, in this case, the policy that is being improved and evaluated is the same as the one that is used for selecting the actions.

2.5 Exploration and Exploitation

About how the agent must behave, it is summarized by the concepts of **exploration** and **exploitation**. The agent must explore to obtain information about which actions give better

rewards. Then, when it is considered that enough information has been collected, the agent must exploit this information by using the actions that give the best reward. However, the agent must decide to what extent it will explore versus how much it will exploit, as there must be a balance between them. Too much exploration will cause problems because the best action cannot be used to increase the reward, while too much exploitation will be problematic as the agent is not amplifying its set of knowledge about which actions are better. So, we say that there is a trade-off between exploitation and exploration. This trade-off is something a good behavior policy must consider. One common policy that encapsulates this notion is the **ϵ-greedy policy**:

$$\pi = \begin{cases} \text{random action } A & \text{with probability } \epsilon \\ \arg\max_A Q(S, A) & \text{with probability } 1 - \epsilon \end{cases} \tag{6}$$

This way, an action is performed randomly with probability $\epsilon \in [0, 1]$, and a greedy action, of the form $\arg\max_A Q(S, A)$, is performed with probability $1 - \epsilon$.

3 Q-Learning

3.1 Updating Step-by-Step

There are many methods in Reinforcement Learning to learn value functions. One of the most relevant ones is **Temporal-Difference Learning**. It updates the value function at each step without the need for a model of the environment, as its estimations are based on the experience it collects.

We will focus our attention in one of the most important Temporal-Difference Learning algorithms called **Q-learning** (Watkins, 1989) used for policy control. That is, for estimating an optimal policy. With this algorithm, we focus on updating the action-value function as follows:

$$Q(S_t, A_t) = (1 - \alpha)Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1})] \tag{7}$$

Where $\alpha \in (0, 1]$. This parameter is a **learning rate** if we rearrange (7):

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{8}$$

Notice we can view $R_{t+1} + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1})$ as a target, in the sense that we have seen R_{t+1} and we are estimating the rest of the expected future reward by using the maximum action-value for the next state using the experience we have collected so far. Therefore, when we subtract our current estimate $Q(S_t, A_t)$ from this target, we are deciding to update it according to the difference between the target and the current estimate, while controlling the size of this update with the learning rate α.

3.2 The Algorithm

In the Q-learning algorithm, the behavior policy, which can be an ϵ-greedy policy, will allow the agent to update the values of the states and actions that are visited. So, it's essential that with the behavior policy we ensure that all possible state and action combinations are visited. It's

useful to think of these values as stored in a look-up table, commonly called **Q-table**. Then, we use this information stored in the table to compute $\max_{A_{t+1}} Q(S_{t+1}, A_{t+1})$.

The Q-learning algorithm is defined as follows:

Algorithm 1 Q-Learning Algorithm

1: Initialize Q-table $Q(S, A)$ arbitrarily
2: Initialize state S_0
3: **repeat**
4: Select action A_t using the behavior policy (e.g., ϵ-greedy)
5: Take action A_t, observe reward R_{t+1} and new state S_{t+1}
6: Update $Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
7: **until** termination condition is met

The termination condition can vary. A common termination condition is to finish after a certain number of iterations.

You may ask whether Q-learning is on-policy or off-policy. In this case, the target policy is based on the estimation of the optimal policy rather than from the behavior policy. Since the target policy differs from the behavior policy, Q-learning is considered an off-policy algorithm.

Exercise 3. What happens in the Q-learning algorithm if the behavior policy does not ensure enough exploration? What happens if it does not ensure enough exploitation?

3.3 Q-learning in Python

In this implementation, we are going to create our own environment for trading a stock. The game is simple: there will be two states, one if the stock goes up and another one if the stock goes down. There will also be just three actions: sell (0), buy (1) or do nothing (2). At each step, the state will change with probability p. The reward of the agent will be given in terms of the number of good trades, that is, the agent will receive $+1$ if it buys and the stock goes up or if it sells and the stock goes down. If the agent do nothing, the reward is 0. In the other cases, it will receive -1. The class `StockGame` defines this environment.

There will also be a class for the Q-learning `Agent`. It will allow to perform actions based on a ϵ-greedy behavior policy and to update the Q-table.

Finally, we define the parameters and use a `for loop` to iterated according to the number of episodes. Each episode will have a defined number of steps. The example will also run the experiment many times, so that we can understand the effects of exploration and exploitation better. How many times it run the game is defined by the number of simulations.

Exercise 4. Try running the algorithm with $\epsilon = 0$ and $p = 0$, $p = 1$. For the other parameters use the ones given on the code. Observe the total reward at each episode. What are the actions that maximize the action-value functions across the simulations? Comment on the results.

Exercise 5. Try running the algorithm wit $p = 0.55$ and $\epsilon = 0.1$, $\epsilon = 0.7$. For the other parameters use the ones given on the code. What are the actions that maximize the action-value functions across the simulations? Compute the mean reward across the simulations at each step and take its mean across all steps. Comment on the results.

Exercise 6. Try running the algorithm wit $p = 0.55$ and $\epsilon = 0.1$, $\epsilon = 0.7$, and add a decay parameter of 0.995 for ϵ. This way it will slightly decrease after each episode. For the other

parameters use the ones given on the code. What are the actions that maximize the action-value functions across the simulations? Compute the mean reward across the simulations at each step and take its mean across all steps. Comment on the results.

4 Deep Q-Learning

4.1 Large Number of States

What if the number of states is too large? In this case, thinking about using a look-up table for storing the values is not feasible or even possible in the case of infinitely many states. Here, we can think of building a parametric representation of the action-value function, and it is in this situation where introducing Deep Learning is useful. One of the most relevant algorithms that does this is Deep Q-Networks (Mnih et al., 2015), which was used for learning how to play Atari games using Reinforcement Learning. Interestingly, another advantage of introducing Deep Learning that can be viewed with this example is that, by using Convolutional Neural Networks, we are now able to learn from grid-like structured data.

At a more general level, the implementation of Deep Learning involves an approximation of the action-value function that is updated through **gradient descent**. For the gradient step to be performed, a **random minibatch** is used from the **experience memory**. There is a separate network for updating the target and a user-fixed parameter that decides after how many steps the parameters of this network are updated to the ones used from the network that is being trained to approximate the action-value function.

4.2 The Algorithm

Algorithm 2 Deep Q-Learning

Initialize experience memory D with size N
Initialize the action-value function with random weights θ
Initialize target action-value function with weights $\theta^- = \theta$
for episode = 1 to M **do**
  Initialize state S_0
  repeat
    Select action A_t using the behavior policy (e.g., ϵ-greedy)
    Take action A_t, observe reward R_{t+1} and new state S_{t+1}
    Store experience $(S_t, A_t, R_{t+1}, S_{t+1})$ in D
    Sample a random minibatch of size M from D
    Set target for Q-learning update:

$$y_i = \begin{cases} R_{i+1} & \text{if episode terminates at step } i+1 \\ R_{i+1} + \gamma \max_{A_{i+1}} Q(S_{i+1}, A_{i+1}; \theta^-) & \text{otherwise} \end{cases}$$

    Perform a gradient descent step on $(y_i - Q(S_i, A_i; \theta))^2$ with respect to the action-value function parameters θ
    Every C steps, update target action-value function: $\theta^- = \theta$
  until termination condition is met
end for

4.3 Deep Q-Learning in Python

For this example, we will use the Gymnasium package (Towers et al., 2023) that provides useful Reinforcement Learning environments. To highlight the utility of Deep Q-Learning, the following implementation will use an environment with a continuous set of states but a discrete set of actions called the Cart Pole environment. The objective of the agent is to balance a pole that is attached to a car. There are only two actions: to move left or right. The state is defined by four continuous variables: the cart position and velocity, and the pole angle and angular velocity. The reward at each step is +1 if the pole is still upright. The episode ends if the pole loses balance or if the cart reaches the edge of the display. There is also a truncation ending if 500 steps are achieved.

For the action-value function approximation, we will use PyTorch (Paszke et al., 2019). The class `ActionValue` define the network. The input size of the network is the number of states, while its output size is the number of actions. The network will have one hidden layer. Both first and hidden layer will use a ReLU activation function. The final hidden layer will use the identity function.

Next the `Agent` class will implement the behaviour policy, the gradient step and the update of the target parameters. Therefore, it will store the experience memory, both networks, and the algorithms parameters. Notice that we still use the parameter α as in Q-learning even though in algorithm 2 this parameter is not seen. This is because, α is the learning rate of the optimizer. In this case, we use Adam (Kingma and Ba, 2015).

After we have defined our classes, the next step is to create the environment and initialize the Agent. The function `process state` converts the state to a `torch.tensor`. After that, we run a `for loop` based on the number of episodes. Inside it, we use a `while loop` until the termination criteria is satisfied.

Exercise 7. Run the code and observe its performance. What do you observe?

5 Algorithms Related to Q-learning

Q-learning is part of the Temporal-Difference Learning algorithms. This algorithms have the advantage that they allow learning from experience rather than requiring some type of model from the environment. Also, they are characterized for using bootstrapping, which essentially means that their current state estimations depend on the next state estimations. Moreover, these algorithms can update their functions at each step, which allows quicker learning.

As we have Q-learning as an off-policy algorithm for estimating an optimal policy. There is also SARSA (Rummery and Niranjan, 1994), which is an on-policy Temporal-Difference Learning algorithm with the same intent. Between them, there is no algorithm that is better in every scenario, so the selection of them might depend on the environment settings.

We also have that Deep Q-Networks allow extending Q-Learning to environments with many possible states, although some other algorithms have been introduced to address some of its limitations. Double Q-Learning (Hasselt, Guez, and Silver, 2015) was introduced to alleviate the overestimations that Deep Q-Networks present under certain settings. Another algorithm, Deep Deterministic Policy Gradient (Lillicrap et al., 2016), is able to include a continuous action domain.

6 Further Study

For further study, there are many great books about Reinforcement Learning. The following books are recommended:

- For Reinforcement Learning: This tutorial paper is mostly inspired by "Reinforcement Learning: An Introduction" (Sutton and Barto, 2018), which has a great way of explaining the theoretical concepts of Reinforcement Learning. Another excellent book is "Algorithms for Reinforcement Learning" (Szepesvári, 2010).
- For Deep Reinforcement Learning: "Deep Reinforcement Learning - Frontiers of Artificial Intelligence" (Sewak, 2019) and "An Introduction to Deep Reinforcement Learning" (François-Lavet et al., 2018).
- You are also invited to visit the papers and other articles that have been discussed throughout the tutorial.

7 Solutions

7.1 Exercise 1.

Given that the reward at each step is finite, show that the sum in (3) is finite.

Define $\bar{R} \geq R_t$, $\forall t$. Then we can establish the inequality:

$$\sum_{i=0}^{\infty} \gamma^i R_{t+1+i} \leq \bar{R} \sum_{i=0}^{\infty} \gamma^i \tag{9}$$

And as $|\gamma| < 1$ we have that it is a converging geometric series

$$\sum_{i=0}^{\infty} \gamma^i R_{t+1+i} \leq \bar{R} \sum_{i=0}^{\infty} \gamma^i = \frac{\bar{R}}{1-\gamma} \tag{10}$$

7.2 Exercise 2.

Answer if each of the following policies satisfies the Markov Property. In case its a valid policy, determine its type of policy.

- If the robot has jumped 10 times in a row, then it must crawl. *Not valid because the policy depends on past actions.*
- If the floor is red, the robot must jump. If the floor is blue, the robot must crawl. *Valid, its a deterministic policy.*
- The robot throws a dice, and jumps if it gets an odd number; otherwise, it crawls. *Valid, its a stochastic policy.*

7.3 Exercise 3.

What happens in the Q-learning algorithm if the behavior policy does not ensure enough exploration? What happens if it does not ensure enough exploitation?

If the behavior policy in the Q-learning algorithm does not ensure enough exploration, the agent may fail to discover the optimal or near-optimal actions in the environment. This lack of exploration can lead to the agent getting stuck in suboptimal policies.

On the other hand, if the behavior policy does not ensure enough exploitation, the agent may not fully take advantage of the actions that are known to be good based on its current estimates.

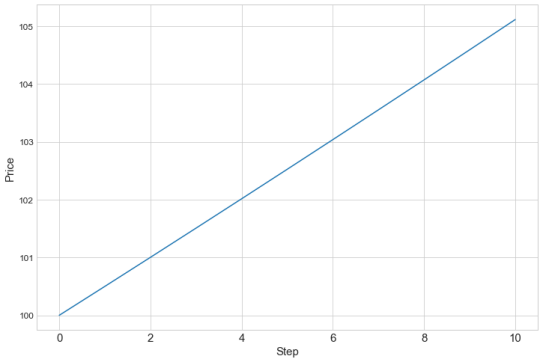
7.4 Exercise 4.

Try running the algorithm with $\epsilon = 0$ and $p = 0$, $p = 1$. For the other parameters use the ones given on the code. Observe the total reward at each episode. What are the actions that maximize the action-value functions across the simulations? Comment on the results.

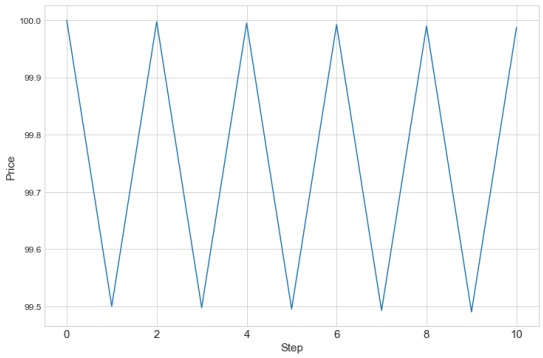
When $p = 0$ there is no probability of changing the state, so the algorithm identifies across all simulations correctly that when the simulation start with an upward trend, the best actions is to buy. Similarly, if the simulation start with a downward trend, the best action is to sell.

When $p = 1$ there is always a change of the state, so the algorithm correctly identifies across all simulations that it must buy when the current state is down and to sell when the current state is up.

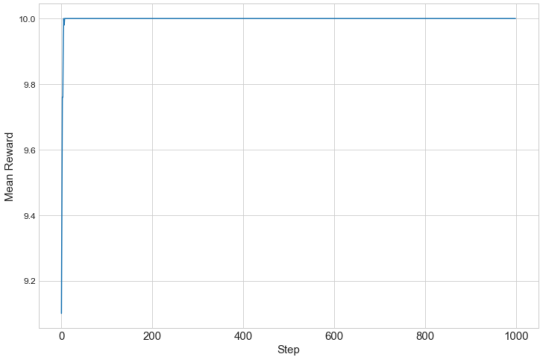
Notice that this happens regardless of the value of ϵ so in this deterministic cases there is no use in exploring.



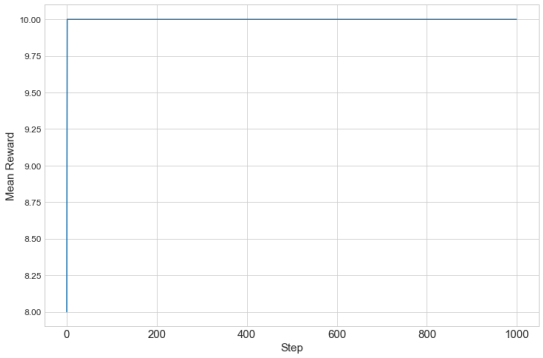
(a) Price $p = 0$



(b) Price $p = 1$



(c) Mean Reward Across Simulations $p = 1$



(d) Mean Reward Across Simulations $p = 0$

Figure 2: Sample of Prices and Mean Reward for $p = 0$, $p = 1$

7.5 Exercise 5.

Try running the algorithm wit $p = 0.55$ and $\epsilon = 0.1$, $\epsilon = 0.7$. For the other parameters use the ones given on the code. What are the actions that maximize the action-value functions across the simulations? Compute the mean reward across the simulations at each step and take its mean across all steps. Comment on the results.

Now, we have that the probability of a change in the state is slightly higher than the probability of remaining on the same state. Therefore, we would like to see across the simulations that the algorithm tends to buy when the market is down and to sell when the market is up, so that it can have a positive reward in the long term.

For $\epsilon = 0.1$ the best actions across the simulations are:

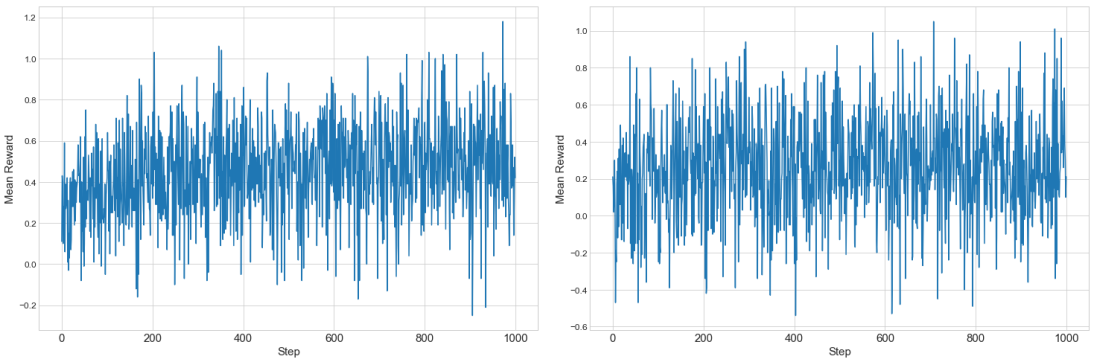
- Buy (1) when market is down, do nothing (2) when market is up (1,2): 48 simulations.
- Do nothing (2) when market is down, sell (0) when market is up (2,0): 43 simulations.
- Buy (1) when market is down, sell (0) when market is up (1,0): 9 simulations.
- Mean of the rewards across steps: 0.45.

When $\epsilon = 0.1$ we find that across the simulations the algorithm does not find the optimal policy as only 3 simulations of the 100 have the correct action-value estimations. In this case, the algorithm is not obtaining new valuable experience because is centered in exploiting.

For $\epsilon = 0.7$ the best actions across the simulations are:

- Buy (1) when market is down, sell (0) when market is up (1,0): 86 simulations.
- Do nothing (2) when market is down, sell (0) when market is up (2,0): 5 simulations.
- Buy (1) when market is down, do nothing (2) when market is up (1,2): 8 simulations.
- Mean of the rewards across steps: 0.26.

When $\epsilon = 0.7$ we find that across the simulations the algorithm have now identified overall the optimal policy. However we have that for this to be achieved we are mostly exploring instead of using the experience collected.



(a) Mean Reward Across Simulations $\epsilon = 0.1$ (b) Mean Reward Across Simulations $\epsilon = 0.7$

Figure 3: Mean Reward for $\epsilon = 0.1$, $\epsilon = 0.7$

7.6 Exercise 6.

Try running the algorithm wit $p = 0.55$ and $\epsilon = 0.1$, $\epsilon = 0.7$, and add a decay parameter of 0.995 for ϵ. This way it will slightly decrease after each episode. For the other parameters use the ones given on the code. What are the actions that maximize the action-value functions across the simulations? Compute the mean reward across the simulations at each step and take its mean across all steps. Comment on the results.

We have that the best actions across the simulations are:

- Buy (1) when market is down, sell (0) when market is up (1,0): 58 simulations.
- Do nothing (2) when market is down, sell (0) when market is up (2,0): 22 simulations.
- Buy (1) when market is down, do nothing (2) when market is up (1,2): 19 simulations.
- Mean of the rewards across steps: 0.67.

With the decay parameter the algorithm is able to achieve better results as it will explore at the start while slightly increasing its exploitation rate after each epoch.

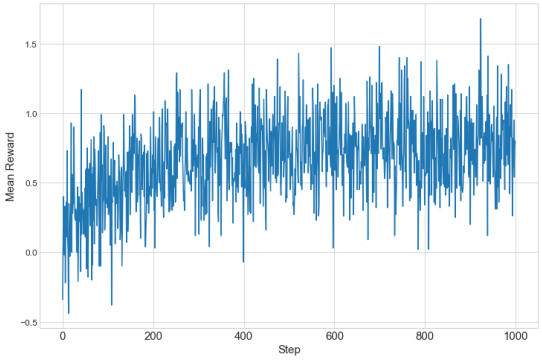


Figure 4: Mean Reward for a Decay Parameter of 0.995

7.7 Exercise 7.

Run the code and observe its performance. What do you observe?

After some epochs, the algorithm is able to increase the time the pole is balanced. It is worth experimenting with some parameters to check if the performance improves.

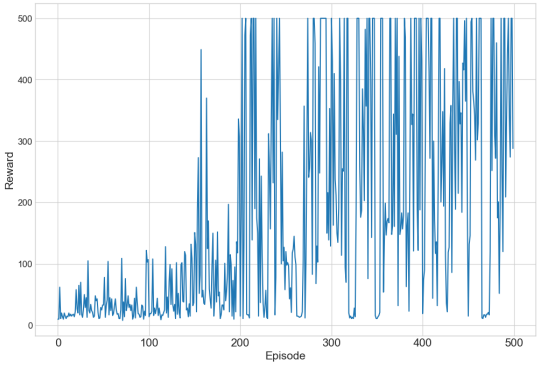


Figure 5: Accumulated Reward per Episode

References

François-Lavet, V. et al. (2018). "An Introduction to Deep Reinforcement Learning". In: *CoRR* abs/1811.12560. arXiv: 1811.12560. URL: http://arxiv.org/abs/1811.12560.

Hasselt, H. van, A. Guez, and D. Silver (2015). "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461. arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

Kingma, D. and J. Ba (2015). "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations (ICLR)*. San Diega, CA, USA.

Lillicrap, T. P. et al. (2016). "Continuous control with deep reinforcement learning." In: ed. by Y. Bengio and Y. LeCun. URL: http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15.

Mnih, V. et al. (Feb. 2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

Paszke, A. et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *CoRR* abs/1912.01703. arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703.

Rummery, G. and M. Niranjan (Nov. 1994). "On-Line Q-Learning Using Connectionist Systems". In: *Technical Report CUED/F-INFENG/TR 166*.

Sewak, M. (2019). *Deep Reinforcement Learning - Frontiers of Artificial Intelligence*. Springer, pp. 1–203. ISBN: 978-981-13-8284-0.

Sutton, R. S. and A. G. Barto (2018). *Reinforcement Learning: An Introduction*. Second. The MIT Press. URL: http://incompleteideas.net/book/the-book-2nd.html.

Szepesvári, C. (2010). *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan Claypool Publishers. ISBN: 978-3-031-01551-9.

Towers, M. et al. (Mar. 2023). *Gymnasium*. DOI: 10.5281/zenodo.8127026. URL: https://zenodo.org/record/8127025 (visited on 07/08/2023).

Watkins, C. J. (1989). "Learning from delayed rewards". PhD thesis. King's College, Cambridge.