Instituto Superior Técnico

1st Cycle Integrated Project in Applied Mathematics and Computation

Reinforcement Learning

Author:
Vasco Pearson - 97015

Professor:
Conceição Amado

2nd Semester - 2021/2022

Contents

Preface

This project is the result of an investigation into the area of reinforcement learning and deep reinforcement learning within the course "1st Cycle Integrated Project in Applied Mathematics and Computation".

The approach explored in reinforcement learning is much more focused on goal-directed learning from interaction than are any other approaches to machine learning. This, and it's wide range of applications, make reinforcement learning one of the most exciting areas in machine learning. Its application is found in a diverse set of sectors like data processing, robotics, healthcare, manufacturing, recommender systems, energy, games, among others. In the last decade, the combination of reinforcement learning and deep learning have led to incredible advances in problems across several domains and therefore I found it relevant to also discuss this new cutting edge field known as Deep Reinforcement Learning.

I found this project very interesting. The deeper I got into Reinforcement Learning the more captivated I became and that made me eager to learn more. I hope this project can also motivate people to study Machine Learning and Reinforcement Learning as it motivated me.

It is worth mentioning that the main source of informartion I used to study and write this project was the book "Reinforcement Learning: An Introduction." by Richard Sutton and Andrew Barto: [1]. It is a great book that I recommend to anyone interested in this field.

1. Introduction to Reinforcement Learning

When we think about the nature of learning, one of the first ideas that come to mind is probably the idea of interacting with an environment. From an early age we learn to use past positive and negative experiences to guide our behavior, and this continues throughout our life. Whether we are driving a car or having a conversation, we are always aware of our environment and how it responds to our actions, and we seek to influence what happens through our behaviour. Learning through interaction is a foundational idea underlying many theories of learning, such as reinforcement learning.

Reinforcement learning is a subset of Machine Learning where an agent learns from direct interaction with its environment. The fact that the agent does not need a complete model of the environment nor exemplary supervision distinguish reinforcement learning from other types of machine learning such as supervised learning. Reinforcement learning takes a computational approach to understanding and automating goal-directed learning and decision making.

In simpler terms, reinforcement learning is learning what to do (how to map situations/states to actions) in order to maximize a numerical reward. The learning agent must discover which actions lead to higher rewards by trying them, this is done by **trial and error search**. In some cases the actions may not only affect the immediate reward but also the next state, and therefore all subsequent rewards. This is a characteristic of reinforcement learning called **delayed reward**.

A challenge that arises in reinforcement learning is the trade-off between **exploration** and **exploitation**. It is arguably the most significant challenge and it is unique to reinforcement learning. In order to maximize reward the agent must exploit what it has already experienced. On the other hand, it might be missing potential from other unknown actions and therefore it must explore in order to make better action selection in the future. The challenge is in achieving a balance between exploration and exploitation that allows reward to be high while still exploring to find better actions.

Lets consider a few examples that will help us understand how a decision-making agent interacts with its environment (that may not be completely known) in order to achieve a goal.

- A robot trash collector decides whether it shoulg go into a new room to pickup trash or go back to its battery recharging station. The decision is made based

on the amount of battery and on how fast it has been able to find the battery recharging station in the past.

- A reinforcement learning agent can be used to decide whether to buy, sell or hold stock at given predicted price.
- While playing chess, a move is chosen based on planning (anticipating possible replies and counterreplies) and by judgement of the desirability of particular positions and moves.

1.1 Elements of Reinforcement Learning

So far we have discussed the agent, the environment, and the interaction that takes place between them. Beyond this, one can identify four main elements of a reinforcement learning system:

- **Policy**: A policy is a mapping from state space to action space. It is what defines how the agent is going to act, given a state from the environment. In some cases a policy may be a simple function or lookup table, while in others it may be more complex such as a search process. Policies may be stochastic, specifying a probability for each action.
- **Reward Signal**: The reward signal defines the goal of the reinforcement learning system, this means that the agent wants to maximize the positive reward in the long term. At each time step the agent recieves a number from the environment corresponding to the reward, which can be positive, negative or zero. The reward signal is the primary basis for altering the policy. For example if an action is followed by a low reward, we can alter the policy so that the probability of that action being selected is lower.
- **Value Function**: The value function is the total amount of reward expected to be accumulated starting from the current state. We can think of the reward signal as an indicator of what is good in an immediate sense and think of the value function as what is good in the long run. We choose actions based on the value function, not the reward signal, because we are interested in maximizing the reward in the long run.
- **Model of the environment**: A model of the environment is used to mimic the behaviour of the environment and it helps us plan and make inferences about the environment. For example if we have a current state and action, the model may tell us what the next state and reward will be. Not all methods to solve reinforcement learning problems require models to be solved (there are model-free methods and model-based methods).

2. Multi-armed Bandits

Reinforcement learning uses training information that **evaluates** the actions taken. This distinguishes it from other types of learning such as supervised learning where the training information instructs by giving correct actions. This evaluative feedback indicates how good the action was, however we have no clue as to whether it was the best or worst action possible. This creates the need for active exploration, in order to search for the best actions.

In order to dive deeper into the evaluative aspect of reinforcement learning, in this section we will only consider problems where there is only one state (one situation), and therefore we only have to choose an action for that situation.

The evaluative feedback problem that we will explore is a common problem in reinforcement learning literature, called the **k-armed bandit problem**. Imagine a slot machine, or "one arm bandit", but with k leavers instead of one. There is only one state, and in this state you are repeatedly confronted with the choice of pulling one of the k leavers (each leaver represents an action). After each choice you receive a numerical reward chosen from a stationary probability distribution. This reward will depend on the action selected. After defining a certain time period or number of actions to be taken, our objective is to maximize the expected total reward.

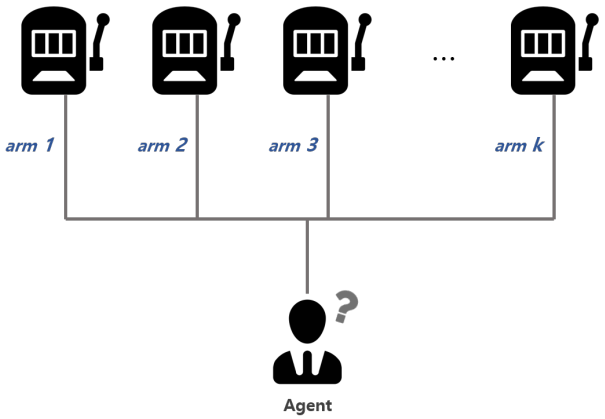


Figure 2.1: k-armed bandit, [4]

The **value** of an arbitrary action a is given by:

$$q_*(a) \doteq \mathbb{E}[R_t|A_t = a]$$

where A_t is the action selected at time t and R_t is the corresponding reward. The value of an action is the expected reward given that that action is selected. However, normally we do not know the value of an action and therefore we use estimates. We

denote the estimated value of action a at time t as $Q_t(a)$.

By maintaining value estimates for each action, we are able to choose actions based on there values. When choosing the action with highest value we are **exploiting** our knowledge in order to obtain higher rewards. When we choose an action with a lower value we are **exploring** in order to obtain better value estimates for our actions.

2.1 Action-value Methods

Methods that obtain value estimates for actions and use those values for decision making are called action-value methods. A simple way of obtaining the value estimates is by averaging the reward obtained for each action. This method is called **sample-average method** and the estimated value of an action a is:

$$Q_t(a) \doteq \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}$$

where $\mathbb{1}_{A_i=a}$ is 1 if $A_i = a$ and 0 otherwise. When the denominator is 0 we define $Q_t(a)$ as some default value and when the denominator goes to infinity $Q_t(a)$ converges to $q_*(a)$.

Given the value estimates, a natural action selection rule is to always select the action with the highest value estimate, as written bellow:

$$A_t \doteq \arg\max_a Q_t(a)$$

This is a **greedy** action selection method in the sense that it always exploits the current knowledge to maximize the expected reward but never samples inferior actions to see if they might really be better. A better approach would be to always behave greedily but choose a random action every once in a while, say with probability ϵ. This way we are able to balance exploration and exploitation in the way we want by tunning the hyperparameter ϵ. These action selection methods are called **ϵ-greedy** methods.

2.2 Incremental Implementation

The value estimates from the sample-average method can be obtained with the incremental formula bellow:

$$Q_{t+1} = \frac{1}{t}\sum_{i=1}^{t} R_i = \frac{1}{t}(R_t + \sum_{i=1}^{t-1} R_i) = \frac{1}{t}(R_t + (t-1)Q_t) = Q_t + \frac{1}{t}(R_t - Q_t)$$

To simplify notation, here we focused on one action. R_i is the reward after the ith selection of the action we are considering and therefore Q_t is the value estimate after $t-1$ selections. In this case, the step-size parameter is $\frac{1}{t}$. This implementation saves a lot of memory and is much more efficient than saving all the reward values until time step t.

2.3 Nonstationary problems

The methods we have discussed so far are only suitable for stationary bandit problems, where the reward probabilities do not change over time. However, if we are dealing with a nonstationary problem (a common scenario in reinforcement learning), the recent rewards are more important and it makes sense to weigh them accordingly. A common way of achieving this is to use a constant step-size parameter. The incremental update rule should now be

$$Q_t \doteq Q_t + \alpha[R_t - Q_t]$$

where $\alpha \in (0, 1]$ is constant.

Sometimes, it is helpfull to vary the step-size parameter according to the time step. When this is the case, we define the step-size parameter as $\alpha(t)$, a function over time.

3. Finite Markov Decision Processes

We will now introduce the formal problem of Markov Decision Processes, which we will attempt to solve during the rest of this project. A Markov decision process (MDP) is a mathematical framework to describe an environment in reinforcement learning. It also involves evaluative feedback, just like the bandit problem, but now we have several states and therefore we also have an **associative** aspect which is choosing different actions in different states. These actions will influence not only the immediate reward but also the subsequent states and therefore future rewards. Thus, in MDP's there is a need to trade off **immediate and delayed reward**.

Due to the associative aspect of the MDP's, it does not make sense to estimate the value $q_*(a)$ of each action as we did in the bandit problem. Instead, we estimate the value $q_*(s, a)$ of each action a in state s, or we estimate the value $v_*(s)$ of state s assuming we always take the optimal action.

In a markov decision process we have an agent who will make the decisions, and everything outside the agent is called the environment. The agent and the environment interact as shown in the figure bellow:

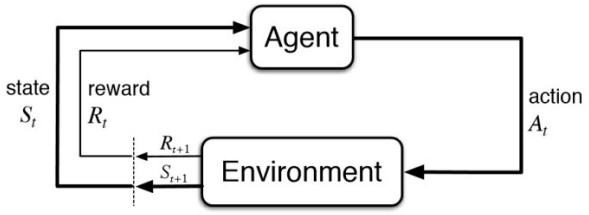


Figure 3.1: Markov decision process, [3]

At each time step t the environment presents the agent with a representation of its current state S_t. The agent then choses the most appropriate action A_t. One time step later, as a consequence of the action chosen, the agent recieves a numerical reward R_{t+1} and finds itself in a new state S_{t+1}.

In a finite MDP, the set of states, actions and rewards all have a finite number of elements. In this case we can define the probability of state s' and reward r occuring at time t, knowing the current state s and the action a taken:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

This equation defines the **dynamics** of the MDP. It is important to note that we assume the state has **Markov property**, meaning that each state includes information about all meaningful past interactions between the agent and the environment.

This is the reason why the probability for each possible value for S_t and R_t depend only on the previous state and action, and not the preceding ones.

3.1 Returns and Episodes

The goal of our reinforcement learning agent is to maximize the **expected return**, where the return (G_t) can be defined in diferent ways depending on the nature of the problem. If the problem has a final time step and the interactions between the agent and the environment can be broken down into subsequences, which we call **episodes**, then we can define the return as the sum of rewards:

$$G_t \doteq R_{t+1} + R_{t+2} + ... + R_T$$

Episodic tasks are tasks where each episode ends in a **terminal state**, followed by a reset to a starting state. **Continuing tasks** on the other hand cannot be broken down into episodes, and continue without limit. In this case, we need to add **discounting** to the return, otherwise it would easily be infinite. Now the agent selects actions that maximize the expected **discounted return**, which is defined bellow:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Discounted return can also be implemented incrementaly:

$$G_t \doteq R_{t+1} + \gamma G_{t+1}$$

Note that discounted returns also work for episodic tasks, we need only define $G_t = 0$, for all $t \geq T$, where T is the final time step. For this reason we will use this notation from here on out, knowing we can be refering to an episodic or continuous task, that can be discounted ou undiscounted ($\gamma = 1$).

3.2 Policies and Value Functions

Value functions are functions that estimate how good it is to be in a certain state, or state-action pair. This is done by estimating the future reward, as we have seen before. In order to estimate the future rewards, the value function must take into consideration how the agent acts, which is defined by the **policy**. A policy is a mapping from state space to action space and it is responsible for the action choice. If an agent is following a policy $\pi : A(s) \rightarrow [0, 1]$ then $\pi(a|s)$ is the probability of picking action a when in state s.

Given a state s and a policy π, a **state-value function** $v_\pi(s)$ is the expected return

when starting in state s and following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \text{ for all s} \in \text{S}$$

Similarly, given a state s, an action a and a policy π, an **action-value function** $q_\pi(s, a)$ is the expected return when starting in state s, taking action a and following a policy π:

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

3.3 Bellman equation

The state-value function also satisfies a recursive relationship similar to what we have seen before:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_t = s]\big] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\big[r + \gamma v_\pi(s')\big] \end{aligned} \tag{3.3.1}$$

This equation is called the **Bellman equation** for v_π. It expresses a relationship between a value of a state and a value of its successor states.

3.4 Optimal Policies and Value Functions

When comparing policies, we compare the expected reward obtained from following that policy. Given two policies π_1 and π_2, if $v_{\pi_1}(s) \geq v_{\pi_2}(s)$, for all $s \in S$, then π_1 is a better policy than π_2. A policy that is better than or equal to all other policies is called the **optimal policy**. We denote all the optimal policies by π_* (there may be more than one). All optimal policies have the same **optimal state-value function** v_*, defined as:

$$v_*(s) \doteq \max_\pi v_\pi(s), \text{ for all } s \in S$$

and the same **optimal action-value function** $q_*(s, a)$, defined as:

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a), \text{ for all } s \in S \text{ and } a \in A(s)$$

We can write q_* in terms of v_* in the following way:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a]$$

Since v_* is a value function we can write its Bellman equation as we did for a state value function in the previous section. However, since v_* is the optimal value function, we can write this equation in a way that highlights the fact that the value of a state under the optimal policy must equal the expected return for the best action from that state. This equation, called the **Bellman optimality equation**, is shown bellow:

$$\begin{aligned} v_*(s) &= \max_{a \in A(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t|S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \end{aligned} \tag{3.4.1}$$

The Bellman optimality equation for q_* is:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a)|S_t = s, A_t = a] \\ &= \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')] \end{aligned} \tag{3.4.2}$$

The bellman optimality equations are special consistency conditions that the optimal value functions must satisfy and than can, in principle, be solved for the optimal value functions, from which we can determine an optimal policy.

4. Dynamic programming

Dynamic programming is used in problem of complete knowledge. This means that we have a perfect model of the environments dynamics. This is quite rare in reinforcement learning and that is why dynamic programming is of limited utility in reinforcement learning. However, the concepts approached in this section will be used in later sections, and thus have theoretical importance.

We will focus on **Generalised Policy Evaluation (GPI)**, which is an algorithm used to solve **finite MDPs**. The key idea will be to compute the state value function, and then use it to search for the optimal policy.

4.1 Policy evaluation (prediction)

Here we take an arbitrary policy π and we want to compute the state-value function v_π. We do this by taking an initial aproximation v_0, which is chosen arbitrarily (except that the terminal state must be given value zero), and we build a succesive aproximation using the Bellman equation for v_π as follows:

$$v_{k+1}(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \big[r + \gamma v_k(s')\big]$$

If we look at the Bellman equation for v_π, we know that $v_k = v_\pi$ is a fixed point for this update rule. This sequence can be shown to converge to v_π as $k \to \infty$, as long as $\gamma < 1$ or the final time-step is finite. We call this algorithm **iterative policy evaluation**. This way of updating the value of each state is called an **expected update**, because the updates are based on an expectation over all possible next states rather than on a sample next state.

4.2 Policy improvement

Policy improvement consists of computing an improved policy, given the value function for that policy. For this we will use the **policy improvement theorem**. Let π and π' be any pair of deterministic policies such that, for all $s \in S$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

then the policy improvement theorem tells us that

$$v_{\pi'}(s) \geq v_\pi(s)$$

meaning that policy π' must be as good as, or better than policy π.

If we now consider a greedy policy that at each state selects the action that appears best according to $q_\pi(s, a)$, given by

$$\begin{aligned} \pi'(s) &\doteq \arg\max_a q_\pi(s, a) \\ &= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{4.2.1}$$

this policy meets the conditions of the policy improvement theorem, therefore the policy π' is as good as, or better than, the original policy. This process of improving upon an original policy in this manner is called **policy improvement**.

4.3 Policy iteration

We obtain policy iteration by combining policy evaluation and policy improvement, untill the process converges to the optimal policy and the optimal value function in a finite number of iterations (this is possible because a finite MDP has only a finite number of policies). Bellow we have the sequence of improving policies and value functions, where I denotes a policy improvement and E denotes a policy evaluation:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} ... \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

The term **Generalised policy evaluation** is used to describe the idea of letting policy evaluation and policy improvement interact. Almost all reinforcement learning methods are well described as GPI.

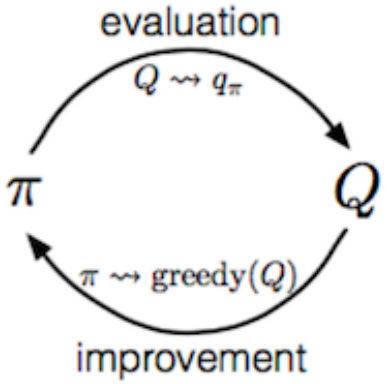


Figure 4.1: Generalised policy evaluation, [1]

5. Monte Carlo Methods

In this section we no longer assume a complete model of the environment, which makes Monte Carlo methods much more useful in real life situations where it is hard to provide a perfect model of the environments dynamics. Monte Carlo methods learn from experience (sequences of states, actions and rewards from interacting with the environment), which is striking because no previous knowledge of the environment is required in order to achieve optimal behaviour.

Here we assume we are dealing with an episodic task (experience is divided into episodes and each episode has a clear and defined ending). Monte Carlo methods estimate the value function and discover the optimal policy be **averaging sample returns**. Only upon the completion of an episode can the value estimates change and the policy be updated.

5.1 Monte Carlo prediction

A simple way to learn the state value function for a given policy is to average the returns observed after visits to that state (remember that the state-value function is the expected return starting from that state). There are different ways to do this but we will focus on the **first visit MC method** which only averages returns following the first visit to a state s.

So suppose we want to estimate the value of $v_\pi(s)$, the value of state s when following policy π. We do this by generating an episode and keeping the return obtained after visiting state s for the first time. When we have done this for several episodes we average the returns and that will be the value of the state s we are considering. This procedure is shown bellow.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow G + R_{t+1}$
        Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append G to $Returns(S_t)$
            $V(S_t) \leftarrow$ average$(Returns(S_t))$

Figure 5.1: First visit MC method, [1]

This method will converge to $v_\pi(s)$, as the number of first visits to s goes to infinity. It is important to note that Monte Carlo methods, unlike Dynamic Programming, do not bootstrap, this means that the value estimate for one state does not build up on the estimate of another state.

5.2 Monte Carlo Estimation (Action Values)

Monte Carlo methods are often used when a model of the environment is not available. In this case it is more useful to estimate **action values** that state values, because without a model state values may not be sufficient. Therefore, what we want to do is use the first visit MC method to estimate action-values instead of state values. This is done exactly as described beforem however now we consider visits to a state-action pair rather than to a state (a state-action pair s, a is visited when a state s is visited and action a is taken).

The problem that arises is that many state-action pair may never be visited. This is the general problem in reinforcement learning of **maintaining exploration**. A solution to this is to start the episode in a random state-action pair, where every state-action pair has a nonzero probability of being selected. We call this the assumption of **exploring starts**. An alternative to the assumption of exploring starts is to consider only policies that are stochastic with a nonzero probability of selecting all actions in each state.

5.3 Monte Carlo control

Monte Carlo control methods are designed as explained in the previous section, following the generalised policy iteration (GPI) schema. Recall that GPI involves interacting processes of policy evaluation and policy improvement in order to cause both policy and value function to approach optimallity. Policy evaluation is done as described above, while policy improvement is done by making the policy greedy with respect to the current value function (action value function in our case).

This can be achieved by defining π_{k+1} as the greedy policy with respect to q_{π_k}, because we can then apply the **policy improvement theorem** to π_k and π_{k+1}, since

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg\max_a q_{\pi_k}(s, a)) \\ &= \max_a q_{\pi_k}(s, a) \\ &\geq q_{\pi_k}(s, a) \geq v_{\pi_k}(s) \end{aligned} \tag{5.3.1}$$

A lot can be said about Monte Carlo methods, however in this project the focus will be mostly on temporal-difference learning and Q-learning (next section), so we will keep this section short and invite anyone who is interested in learning more about Monte Carlo methods to read chapter 5 of Richard Sutton and Andrew Barto's book [1].

6. Temporal-Difference learning

Temporal-difference (TD) learning is a central idea to reinforcement learning. It combines the ideas of Monte Carlo methods and Dynamic Programming. They are similar to Monte Carlo methods in the sense that they also do not need a model of the environment's dynamics, which is a great advantage since a complete model is hard to come by in a lot of problems. Like Dynamic Programming, TD methods **bootstrap**, meaning that they update estimates based on other estimates, without waiting for the final outcome.

Dynamic Programming, TD, and Monte Carlo methods all use some sort of generalized policy evaluation (GPI), however the main differences between these methods is their approach to the prediction problem.

6.1 TD prediction

Just like the Monte Carlo methods, TD also takes experience as the approach to solve the prediction problem. The difference is that while Monte Carlo methods must wait untill an episode ends to increment the value of $V(S_t)$ (because only then is the return (G_t) known), TD methods need only wait untill the next time step. The simplest TD method makes use of the reward R_{t+1} observed at time setp $t+1$, and the estimate V_{t+1} to form a target for the following update

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

The TD method described above is called **TD(0)** or **one-step TD** because it is a special case of TD(γ) or n-step TD methods. In the box bellow we show the procedure followed for TD(0).

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 6.1: TD(0), [1]

So the target for TD(0) is $R_{t+1}+\gamma V(S_{t+1})$, while the target in a Monte Carlo method would be G_t (the return after one episode) and the update would be

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

We refer to Monte Carlo and TD updates as sample updates because they involve looking ahead to a successor state or state action pair and using its value and the reward along the way to compute a backed-up value, and then updating the value of the original state. So TD methods combine the **sampling** of Monte Carlo methods with the **bootstraping** of DP, with the objective of obtaining the advantages of both methods.

6.2 On-policy vs off-policy

As we now try to solve the control problem, we are faced with a dilema. We want our agent to learn action values conditional on subsequent optimal behaviour, but we also want it to behave non-optimally in order to explore all actions. There are two approaches to solve this problem. The **on-policy** approach is to attempt to evaluate or improve the policy that is used to make decisions. This on policy approach is a compromise, because the agent learns action values not for the optimal policy, but for a nearly optimal policy that still explores. A more straightforward approach would be to use two policies, which is what is done in the **off-policy** approach. In this approach, a **target policy** is learned about and becomes the optimal policy, while a more exploratory **behaviour policy** is used to generate behaviour.

6.3 Q-learning: On-policy TD Control

As usual, we will follow the generalized policy iteration (GPI) pattern to solve the control problem. Approaches to solve the exploration/explotation problem can fall into two categories, just as discussed above, which can be on-policy and off-policy. The most common on-policy approach is the Sarsa algorithm, however in this section we will focus on an off-policy approach called Q-learning [2].

The Q-learning algorithm was considered one of the early breakthroughs is reinforcement learning. It is an off-policy TD control algorithm. It is model free, meaning it does not need a model of the environments dynamics. We solve the prediction problems by learning the action-value function rather than the state-value function.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

The learned action-value function Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. This is because the update is done always assuming we pick the next action a as the one that will maximize $Q(S_{t+1}, a)$. So we have this optimistic view for the update that all hence-forth action selections from every state should be optimal, but in reality we have no constraints on how the next action will be selected, allowing us to have a more exploratory policy that determines which state-action pairs are visited and updated.

All that is required for the function Q to converge to q_* with probabillity 1 is that all action-value pairs continue to be updated, and that the learning rates approach zero, but not to fast (the sum of the learning rates must diverge, but the sum of their squares must converge). Proof is shown in [2].

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 6.2: Q-learning procedure, [1]

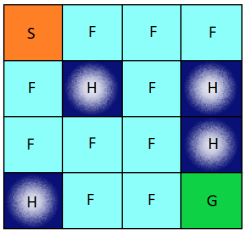
7. Practical example - Q-learning

Let's now try to solve an actual problem using what was introduced in the previous sections. The most interesting algorithm to try and implement would be Q-learning, since it is one of the most used algorithms in reinforcement learning and up to this day variants of Q-learning (such as Deep Q-learning) are achieving groundbreaking results in several areas.

7.1 OpenAI Gym - FrozenLake environment

OpenAI is an AI research and deployment company. They provide a toolkit, called Gym [5], for developing and comparing reinforcement learning algorithms. I recomend anyone who is learning reinforcement learning to have a look at the toolkit and try to solve some environments with simple algorithms.

The environment we will solve is the FrozenLake environment. Here, our agent controls the movement of a character in a grid world. Some tiles of the grid world are walkable, while others are holes in the lake wich lead to the agent falling into the water. The agent is rewarded when it reaches the goal tile/state without falling in the water. The surface is described using a grid like the following:



S: starting point, safe
F: frozen surface, safe
H: hole, fall
G: goal

So each tile in 4x4 grid corresponds to a state and in each tile or state, the agent has 4 possible actions: left, right, up or down. The actions and states are numbered as shown bellow:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Left = 0
Down = 1
Right = 2
Up = 3

Note that we cannot leave the grid, so for example if we are in state 4 and take action 0 (left) we remain in state 4.

Our agent only obtains reward when it reaches the goal state, therefore our discount rate γ will have to be quite high in order to focus on future rewards.

7.2 Implementing Q-learning

The Q-learning algorithm was implemented just as described in the previous section. Our Q-table was initialized as a 16x4 table filled with zeros, where each line corresponds to a state and each column to an action (the indexes are the ones stated above, the first row and column are index 0). The algorithm is shown bellow.

In [1144]:
```
rewards_all_episodes = []
exploration_rates = []

#Q-learning algorithm
for episode in range(num_episodes):
    state = env.reset() #start state
    done = False
    rewards_current_episode = 0

    for step in range(max_steps_per_episode):

        #Exploration exploitation trade off
        exploration_rate_threshold = random.uniform(0,1)
        if exploration_rate_threshold < exploration_rate:
            action = env.action_space.sample() #random action
        else:
            action = np.argmax(q_table[state, :]) #action that maximizes Q(s, a)

        #Take action and see new state, reward and if we have reached the goal state
        new_state, reward, done, info = env.step(action)

        #Update Q-table for Q(s,a)
        update = learning_rate * (reward + discount_rate * np.max(q_table[new_state, :]) - \
                                                    q_table[state, action])
        q_table[state, action] += update

        #Data for plots
        if ((episode,q_table[state, action]) not in dict_action_val[(state, action)]):
            dict_action_val[(state, action)].append((episode,q_table[state, action]))

        #Update new state and rewards
        state = new_state
        rewards_current_episode += reward

        if done == True:
            break;

    #Exploration rate decay
    exploration_rate = min_exporation_rate + \
    (max_exploration_rate - min_exporation_rate) * np.exp(-exploration_decay_rate*episode)

    rewards_all_episodes.append(rewards_current_episode)
    exploration_rates.append(exploration_rate)
```

We ran for 7000 episodes, with a maximum of 100 steps per episode. We used a learning rate of 0.1 and a discount rate of 0.95. The discount rate had to be a high value because the agent only recieves a reward signal when it reaches the goal state. Recall that a low discount rate means the agent will focus more on immediate rewards, while a higher discount rate makes it focus on potential future rewards (see section 3).

To change the exploration rate we used exponencial decay, so that after the necessary initial exploration, the exploration rate would decrese in order to maximize the reward. The exploration decay rate used was 0.0005 because values bigger than this would not provide enough exploration and the values of the Q-table would remain 0. The minimum exploration rate was set to 0.01, so the agent will always act randomly at least 1% of the time.

```
In [1143]: num_episodes = 7000
           max_steps_per_episode = 100

           learning_rate = 0.1
           discount_rate = 0.95

           exploration_rate = 1
           max_exploration_rate = 1
           min_exporation_rate = 0.01
           exploration_decay_rate = 0.0005
```

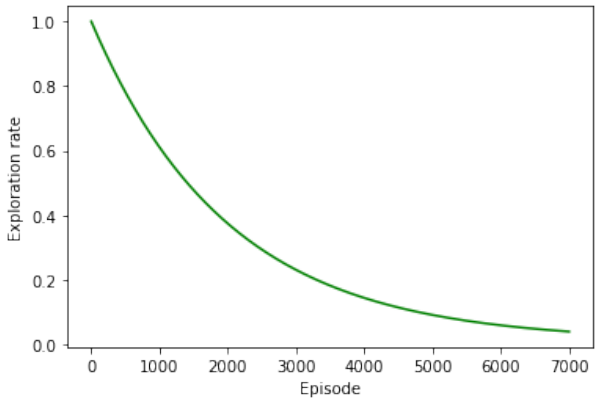
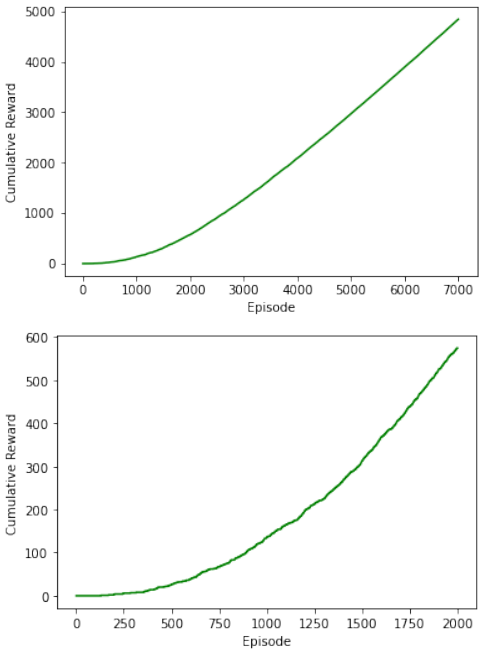


Figure 7.1: Exploration rate as a function of the episode number, [1]

7.3 Results

Let's start by looking at the rewards obtained throughout the 7000 episodes.

Episodes	Average Reward
0-500	0.052
500-1000	0.220
1000-1500	0.354
1500-2000	0.522
2000-2500	0.670
2500-3000	0.722
3000-3500	0.806
3500-4000	0.834
4000-4500	0.870
4500-5000	0.894
5000-5500	0.914
5500-6000	0.934
6000-6500	0.942
6500-7000	0.954



Looking at the data, it is obvious that the agent has solved this environment. In the last 500 episodes the average reward is 0.954, meaning that the agent reached the goal 477 times out of 500. Recall that the exploration rate is always higher than 0.01 so at least 1% of the time the agent is choosing randomly, which is why the average reward will never be 1. We will se later that if we removed exploration at the end, the agent would always reach the goal state. It is also worth mentioning that most of the learning seems to be happening in the first 2000/3000 steps. Lets now examine the Q-table values to verify this and to see how and what the agent learned.

Recall that we initialized the Q-table with zeros, that is why we needed a high initial exploration rate to reach the goal. So at episode 0 the Q-table looks like the one bellow on the left. After 132 episodes the agent finaly reaches the goal for the first time and the value of $Q(14, 2)$ (going right on the 14^{th} state) is updated.

```
Episode: 0, Total reward: 0.0, Eps: 1
s:  0 a:  3
  (Up)
SFFF
FHFH
FFFH
HFFG
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

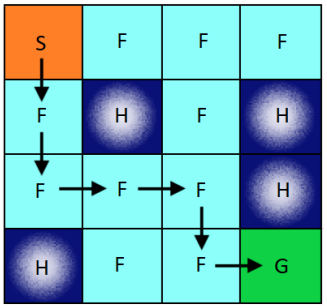
```
Episode: 132, Total reward: 1.0, Eps: 0.9372330562920316
s:  0 a:  0
  (Left)
SFFF
FHFH
FFFH
HFFG
[[0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.1 0. ]
 [0.  0.  0.  0. ]]
```

After a few more episodes (bellow we see the Q-table for episodes 178 and 210) a couple more states are updated.

```
Episode: 178, Total reward: 2.0, Eps: 0.9161500796286376
s:  1 a:  3
  (Up)
SFFF
FHFH
FFFH
HFFG
[[0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.0095 0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.0095 0.19   0.    ]
 [0.     0.     0.     0.    ]]
```

```
Episode: 210, Total reward: 4.0, Eps: 0.9017670494328143
s:  4 a:  1
  (Down)
SFFF
FHFH
FFFH
HFFG
[[0.00000007 0.00000147 0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.00000077 0.00000815 0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.0009025  0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.00008574 0.00000077]
 [0.         0.         0.0009025  0.        ]
 [0.         0.0266     0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.025745   0.        ]
 [0.         0.0095     0.3439     0.        ]
 [0.         0.         0.         0.        ]]
```

What is most impressive is that if the agent were to stop exploration by episode 210 and follow a greedy policy where it always picks the action a that maximizes $Q(s, a)$ for the state it is in, it would always reach the goal state. Take a look at the Q-table above, corresponding to episode 210. The values underlined are the maximum in their rows and therefore their column number coresponds to the action that should be taken, leading to the path shown bellow.



The exploration rate at episode 210 is still 0.90 (value of *Eps* in the figure above) and therefore the agent will continue to explore and may find beter paths. By episode 2000 we have the following results, very simillar to the final Q-table results (after 7000 episodes) shown on the right.

```
Episode: 2000, Total reward: 574.0, Eps: 0.37438279261577706
s:  4 a:  1
  (Down)
SFFF
FHFH
FFFH
HFFG
[[0.73509189 0.77378094 0.76975601 0.73509189]
 [0.73509189 0.         0.81291319 0.72478715]
 [0.67872204 0.85700548 0.32281055 0.72382941]
 [0.58336142 0.         0.07862958 0.07736439]
 [0.77378094 0.81450625 0.         0.73509189]
 [0.         0.         0.         0.        ]
 [0.         0.90242648 0.         0.68071919]
 [0.         0.         0.         0.        ]
 [0.81450625 0.         0.857375   0.77378094]
 [0.81450397 0.9025     0.90249974 0.        ]
 [0.85228535 0.95       0.         0.84673489]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.90249921 0.95       0.85737262]
 [0.90249931 0.949999   1.         0.90248394]
 [0.         0.         0.         0.        ]]
```

```
******Q-table******

[[0.7351 0.7738 0.7738 0.7351]
 [0.7351 0.     0.8145 0.7655]
 [0.7579 0.8574 0.4563 0.786 ]
 [0.6916 0.     0.1321 0.168 ]
 [0.7738 0.8145 0.     0.7351]
 [0.     0.     0.     0.    ]
 [0.     0.9025 0.     0.7505]
 [0.     0.     0.     0.    ]
 [0.8145 0.     0.8574 0.7738]
 [0.8145 0.9025 0.9025 0.    ]
 [0.8571 0.95   0.     0.8569]
 [0.     0.     0.     0.    ]
 [0.     0.     0.     0.    ]
 [0.     0.9025 0.95   0.8574]
 [0.9025 0.95   1.     0.9025]
 [0.     0.     0.     0.    ]]
```

There is very little difference between these two tables, which again points to the ideia that most of the learning is happening in the first 2000 episodes. Bellow, to make the results clearer, we have two color maps representing the value of each state action pair (left) and the value of each state (right). Recall from section 3 that under the optimal policy

$$v_*(s) = \max_{a \in A(s)} q_{\pi_*}(s, a)$$

therefore the value of each state s is the max value along row s of the Q-table.

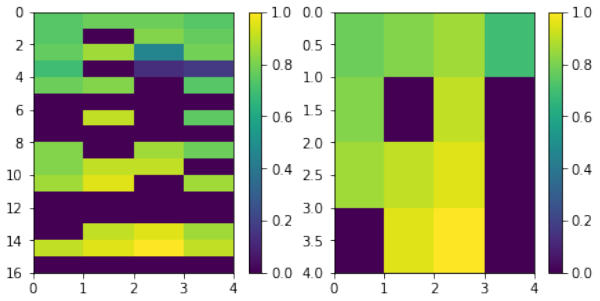


Figure 7.2: Value of each state-action pair (left) and of each state (right)

Note also that if we follow the greedy policy

$$\pi(s) = \arg\max_a Q(s, a)$$

where $Q(s, a)$ are the values from the final Q-table (after 7000 episodes), there are three paths that lead us to the goal.

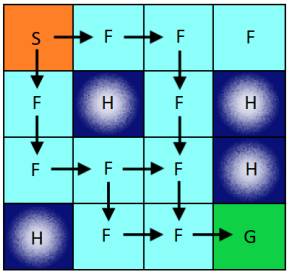
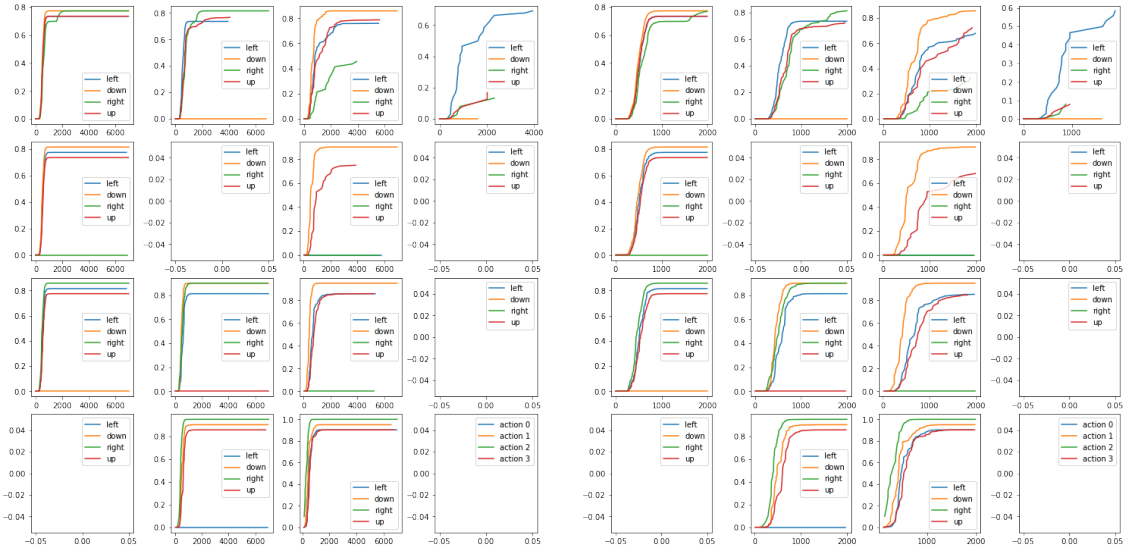


Figure 7.3: Final paths followed by the greedy policy

It is also interesting to compare the action-values within each state. Bellow on the left we have 16 plots representing the 16 states and in each plot we see the value of each action in that state. As we can see, most of the values converge before episode 2000. Therefore on the right we have have the same 16 plots but only untill episode 2000, for a more clear visualization. The plots that are empty correspond to holes or the goal state. We can see that the states that are near the goal state are updated before the initial states, which is expected. Also, the state-action pairs that are part of the optimal path are updated faster that the others, which is also the expected behaviour.



8. Applications

During the 1970s and the 1990s disappointing progress in the areas of machine learning and artificial intelligence led to a decrease of interest in these areas. This is very much not the case right now. In the past decade we have seen amazing accomplishments in artificial intelligence which have sparked excitement and interest in the area. Some areas of recent success are language translation, image recognition and game-playing systems.

Reinforcement Learning, being an area of machine learning, has taken advantage of the recent spike in interest and has been responsible for some major advances in several areas. We discuss some of these bellow.

8.1 Deep reinforcement learning

Deep reinforcement learning is the intersection of reinforcement learning and deep learning to solve problems where it is necessary to deal with incredibly large state sets. In these cases, it is impossible for the agent to experience every state, and therefore it needs to be able to generalise from experience so that in new states, it can make decisions based on previous similar states. One approach to this is to pass the input (state) through an artificial neural networks (ANNs) and using the output for the reinforcement learning model. The latest advances in training deeply-layered ANNs (deep learning) is responsible for some of the most impressive abilities of machine learning systems, including reinforcement learning systems.

8.1.1 Autonomous driving

Various papers have proposed Deep Reinforcement Learning for autonomous driving. The paper [7] provides an overview of RL applications for autonomous driving problems and discusses challenges in deploying RL for real-world autonomous driving systems. Creating systems that are capable of autonomous driving is a very complex task because there are a lot of aspects and restrictions that need to be taken into account. Some of these are speed limits at various places, drivable zones, avoiding collisions, traffic, and many others.

Autonomous driving tasks where RL could be applied include: path planning and trajectory optimization, development of high-level driving policies for complex navigation tasks, intersections, merges and splits, learning of policies that ensures safety and perform risk estimation, and others mentioned in [7].

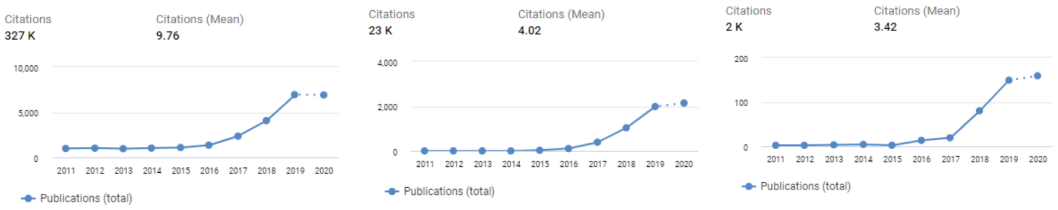


Figure 8.1: Trend of publications for keywords 1. "reinforcement learning", 2."deep reinforcement", and 3."reinforcement learning" AND ("autonomous cars" OR "autonomous vehicles" OR "self driving"), [7]

An interesting aplication is the use of **Q-learning** (in this case a variant of Q-learning called **Deep Q-learning**) to solve the lane changing task, as done in [8]. Deep Q-Learning replaces the regular Q-table with a neural network. Rather than mapping a state-action pair to a q-value, a neural network maps input states to (action, Q-value) pairs, then choosing between several actions based on their Q-values.

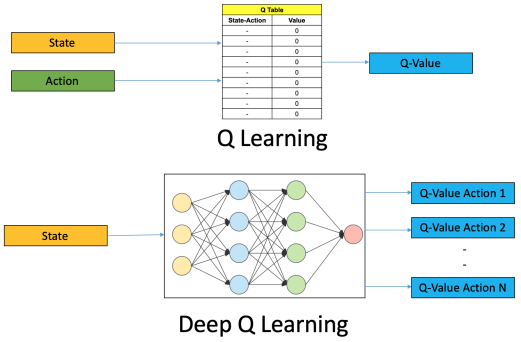


Figure 8.2: Difference between Deep Q-learning and Q-learning, [9]

8.1.2 Gaming

In 2013, a paper [10] by the Google Deepmind team explored using Deep Q-learning to learn to play Atari games. The goal was to present a model that could deal with the high dimensional input data and achive great results en several games without needing to be modified.

This was the paper that introduced Deep Q-learning to the world and it demonstrated its ability to master difficult control policies for Atari 2600 computer games, using only raw pixels as input. In the paper the approach gave state-of-the-art results in six of the seven games it was tested on, with no adjustment of the architecture or hyperparameters.

8.1.3 Trading and finance

Supervised time series models can be used for predicting future sales as well as predicting stock prices. However, these models cannot decide for themselves what action to take at different stock prices. This is where reinforcement learning is needed. A reinforcement learning agent can help make these decisions and benefit any person or company.

The paper [11] gives us a deep reinforcement learning approach for time series by playing idealized trading games. It shows that deep reinforcement learning is promising to discover the optimal strategies to act on time series input. Companies like IBM use systems that make financial trades using the power of reinforcement learning. In this case the model trains on the historical stock price data and calculates the reward function based on the profit or loss for each trade.

8.2 Healthcare

Dynamic treatment regimes provide a new way to automate the process of developing new effective treatment regimes for individual patients with long-term care. A dynamic treatment regimes is composed of a sequence of decision rules to determine the course of actions (treatment type, drug dosage, or reexamination timing) at a time point according to the current health status and prior treatment history of an individual patient.

The design of DTRs can be viewed as a sequential decision making problem that fits into the RL framework well. The series of decision rules in DTRs are equivalent to the policies in RL, while the treatment outcomes are expressed by the reward functions. The inputs in DTRs are a set of clinical observations and assessments of patients, and the outputs are the treatments options at each stage, equivalent to the states and actions in RL, respectively.

More about DTRs and other applications of reinforcement learning methods in healthcare (e.g., clinical resource allocation and scheduling) are discussed in this paper [12].

9. Conclusion

The machine learning world (reinforcement learning included) has seen tremendous progress in the last decade which captured the attention of lots of people who can have very different opinions on the matter. Very often artificial intelligence is seen in a negative way and sometimes there have been failures in AI systems that give people reasons to not trust the technology. However, these failures are rare and AI systems are tested extensively before being put in practice in order to benefit people in a safe and ethical way. Arguing against AI is to argue against safer cars than will have fewer accidents and against better medical diagnoses when people are sick, potentially saving lives. Of course the advancements in AI will have some downsides like the loss of jobs in some sectors and we will have to come up with solutions for these problems, but in my opinion the good outweighs the bad and it is important that this technology is constantly regulated so that it is safe and people trust it.

Reinforcement learning has benefited tremendously from the growth of the machine learning field, and by being combined with deep learning has produced some of the most groundbreaking results in artificial intelligence in the past decade. These results are built from years of research by some of the worlds brightest minds.

In this project a simple introduction to reinforcement was made, and even though we talked about Q-learning and Monte Carlo methods which are commonly used, topics such as planning and eligibility traces were not discussed. I found the topics of Q-learning and deep Q-learning very interesting and the breakthroughs achieved be applying deep Q-learning to different areas are really amazing.

I hope this project motivates the reader to look into reinforcement learning and machine learning and all the amazing work being done in these areas. I enjoyed learning more about these areas, being up to date with the state of the art research and looking into what may come next. I invite anyone interested in learning more about reinforcement learning to take a look at the references bellow.

10. Bibliography

- Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. The MIT Press, 2020. [1]
- Watkins, C.J.C.H. and Dayan, P. (1992) Q-Learning. Machine Learning, 8, 279-292. [2]
- https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html. [3]
- https://velog.io/@taejinjeong/Reinforcement-Learning-Multi-Armed-Bandit-Problem. [4]
- https://gym.openai.com. [5]
- https://gsverhoeven.github.io/post/frozenlake-qlearning-convergence/. [6]
- B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yo- gamani, and P. Pe´rez, "Deep reinforcement learning for autonomous driving: A survey," IEEE Transactions on Intelligent Transportation Systems, 2021. [7]
- Wang, G., Hu, J., Li, Z., and Li, L. (2019a). Cooperative Lane Changing via Deep Reinforcement Learning. arXiv preprint arXiv:1906.08662. [8]
- https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/. [9]
- Mnih, V., et al., Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013. [10]
- Gao, Xiang. (2018). Deep reinforcement learning for time series: playing idealized trading games. Arxiv preprint. [11]
- Yu, C.; Liu, J.; Nemati, S. Reinforcement learning in healthcare: A survey. arXiv 2019, arXiv:1908.08796. [12]