Reinforcement Learning (DRAFT)

Soufiane Fadel[1],Patrick Boily[1]

Abstract
This article is about exploring the power of a hot area of machine learning called "Reinforcement Learning (RL)", that focuses on how you, or how something, might act in an environment in order to maximize some given reward. Reinforcement learning algorithms study the behaviour of subjects in such environments and learn to optimize that behaviour. What distinguishes reinforcement learning from supervised learning is that only partial feedback is given to the learner about the learner's predictions. Reinforcement learning is of great interest because of the large number of practical applications that it can be used to address, ranging from problems in artificial intelligence to operations research or control engineering The tools learned in this section can be applied to game development (AI) , industrial control, Find pages relevant to queries (Search Engines), Robotics, Personalized medical treatment, Quantitative finance, finding optimal architecture in deep learning ...

Keywords
Reinforcement learning, Markov Decision Processes, programmation dynamique, Value functions - bellman function, Dynamic programming, Critic Only, Model-Free Control, Q-Learning, function approximation, Neural network ...

Funding Acknowledgement
Parts of this report were funded by a University of Ottawa grant to develop teaching material in French (2019-2020). These were subsequently translated into English before being incorporated into this document.

[1]Department of Mathematics and Statistics, University of Ottawa, Ottawa
[2]Sprott School of Business, Carleton University, Ottawa
[3]Data Action Lab, Ottawa
[4]Idlewyld Analytics and Consulting Services, Wakefield, Canada
Email: pboily@uottawa.ca

Contents

1. The Reinforcement Learning Introduction

1.1 About Reinforcement learning

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an **agent** from direct interaction with its **environment**, without requiring exemplary supervision or complete models of the environment.

Some AI experts says that Reinforcement learning is the first field to seriously address the computational issues that arise when learning from interaction with an environment in order to achieve long-term goals. Reinforcement learning uses the formal framework of **Markov decision processes** (will be covered in section 1.3) to define the interaction between a learning agent and its environment in terms of **states**, **actions**, and **rewards**. This framework is intended to be a simple way of representing essential features of the artificial intelligence problem.

The concepts of **value** and **value function** are key to most of the reinforcement learning methods that we consider in this report. We take the position that value functions are important for efficient search in the space of **policies**. The use of value functions distinguishes reinforcement learning methods from evolutionary methods that search directly in policy space guided by evaluations of entire policies.

History of Reinforcement Learning

The term "reinforcement" in the context of animal learning came into use well after expression of the **Law of Effect** introduced by Thorndike's, first appearing in this context (to the best of our knowledge) in the 1927 English translation of Pavlov's monograph on conditioned reflexes. Pavlov described reinforcement as the strengthening of a pattern of behavior due to an animal receiving a stimulus a reinforcer in an appropriate temporal relationship with another stimulus or with a response. Some psychologists extended the idea of reinforcement to include weakening as well as strengthening of behavior, and extended the idea of a reinforcer to include possibly the omission or termination of stimulus. To be considered reinforcer, the strengthening or weakening must persist after the reinforcer is withdrawn; a stimulus that merely attracts an animal's attention or that energizes its behavior without producing lasting changes would not be considered a reinforcer.

Pavlov's dog

- ♣ Presenting food to a dog causes salivation
- ♣ Repeated experience: just after presenting the food, we ring a bell
- ♣ The dog ends up salivating at the simple sound of the bell

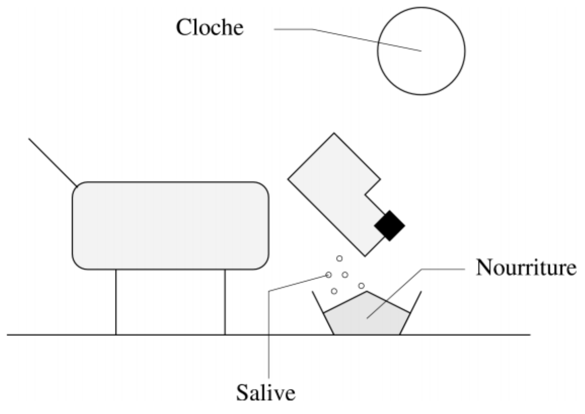


Figure 1. Pavlov's dog experience

Skinner's box

- ♣ Press the lever to deliver food
- ♣ The pigeon learns by trial and error to press the lever so that food is delivered
- ♣ The device can also be complicated to condition the delivery of food by an audible or light signal.

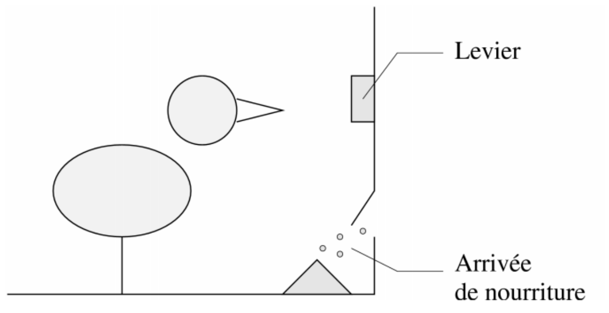


Figure 2. Skinner's box experience

Those experiments illustrate the learning by trial and error of an action possibly conditioned by a situation. we observe in. the second . experiment that as soon as he presses the button, the pigeon is **rewarded** The reward **reinforces** this action. The pigeon learns to press the button more and more often Modifying the reward leads to modifying the behavior that the animal learns.

In fact those experiments of artificial reproduction of animals conditioning are the essence of the modern Reinforcement learning.

1.2 Multi-armed Bandits

In this chapter we study the evaluative aspect of reinforcement learning in a simplified setting, one that does not involve learning to act in more than one situation. We use this problem to introduce a number of basic learning methods which we extend in later chapters to apply to the full reinforcement learning problem. At the end of this chapter, we take a step closer to the full reinforcement learning problem by discussing what happens when the bandit problem becomes *associative*, that is, when actions are taken in more than one situation.

The K-Armed Bandit Problem

The multi-armed bandit problem is a classic problem that well demonstrates the **exploration vs exploitation** dilemma could be defined as the following.

- *If we have learned all the information about the environment, we are able to find the best strategy by even* *just simulating brute-force, let alone many other smart approaches. The dilemma comes from the incomplete information: we need to gather enough information to make best overall decisions while keeping the risk under control. With exploitation, we take advantage of the best option we know. With exploration, we take some risk to collect information about unknown options. The best long-term strategy may involve short-term sacrifices. For example, one exploration trial could be a total failure, but it warns us of not taking that action too often in the future.*

So back to our K-Armed Bandit Problem, imagine you are in a casino facing multiple slot machines and each is configured with an unknown probability of how likely you can get a reward at one play. The question is: What is the best strategy to achieve highest long-term rewards?

In this section, we will only discuss the setting of having an infinite number of trials. The restriction on a finite number of trials introduces a new type of exploration problem. For instance, if the number of trials is smaller than the number of slot machines, we cannot even try every machine to estimate the reward probability (!) and hence we have to behave smartly w.r.t. a limited set of knowledge and resources (i.e. time).

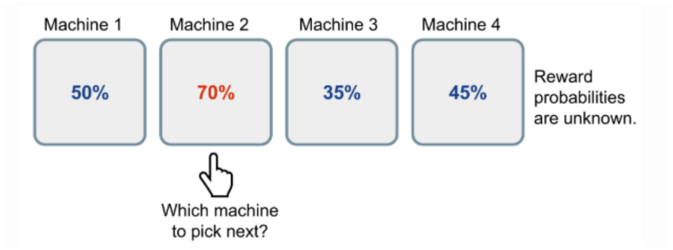

Machine 1
Machine 2
Machine 3
Machine 4
50%
70%
35%
45%
Reward probabilities are unknown.
Which machine to pick next?

Figure 3. An illustration of how a Bernoulli multi-armed bandit works. The reward probabilities are **unknown** to the player

A naive approach can be that you continue to playing with one machine for many many rounds so as to eventually estimate the "true" reward probability according to the law of large numbers. However, this is quite wasteful and surely does not guarantee the best long-term reward.

Definition

Now let's give it a scientific definition.
A Bernoulli multi-armed bandit can be described as a tuple of $\langle \mathscr{A}, \mathscr{R} \rangle$, where:

- We have **K** machines with reward probabilities, $\{\theta_1, \ldots, \theta_K\}$.
- At each time step t, we take an action a on one slot machine and receive a reward r.
- $\mathscr{A}$ is a set of actions, each referring to the interaction with one slot machine. The value of action a is the expected reward, $Q(a) = \mathbb{E}[r|a] = \theta$ If action a_t at the time step t is on the i-th machine, then $Q(a_t) = \theta_i$
- $\mathscr{R}$ is a reward function. In the case of Bernoulli bandit, we observe a reward r in a stochastic fashion. At the time step t, $r_t = \mathscr{R}(a_t)$ may return reward 1 with a probability $Q(a_t)$ or 0 otherwise.

It is a simplified version of *Markov decision process* (see next section), as there is no state $\mathscr{S}$.

The goal is to maximize the cumulative reward $\sum_{t=1}^{T} r_t$. If we know the optimal action with the best reward, then the goal is same as to minimize the potential regret or loss by not picking the optimal action.
The optimal reward probability θ^* of the optimal action a^* is:

$$\theta^* = Q(a^*) = \max_{a \in \mathscr{A}} Q(a) = \max_{1 \le i \le K} \theta_i$$

Our loss function is the total regret we might have by not selecting the optimal action up to the time step T:

$$\mathscr{L}_T = \mathbb{E}\Big[\sum_{t=1}^{T} \big(\theta^* - Q(a_t)\big)\Big]$$

Bandit Strategies

Based on how we do exploration, there several ways to solve the multi-armed bandit.

- **Greedy algorithm :** No Exploration.
- **ϵ-Greedy algorithm :** Random Exploration.
- **Upper Confidence Bounds algorithm :** Smart Exploration.

The ϵ-Greedy algorithm

The ϵ-Greedy algorithm takes the best action most of the time, but does random exploration occasionally. The action value is estimated according to the past experience by averaging the rewards associated with the target action a that we have observed so far (up to the current time step t):

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{\tau=1}^{t} r_\tau \mathbb{1}[a_\tau = a]$$

where $\mathbb{1}$ a binary indicator function and $N_t(a)$ is how many times the action a has been selected so far, $N_t(a) = \sum_{\tau=1}^{t} \mathbb{1}[a_\tau = a]$

According to the ϵ-greedy algorithm, with a small probability ϵ we take a random action, but otherwise (which should be the most of the time, probability 1-ϵ) we pick the best action that we have learnt so far: $\hat{a}_t^* = \arg\max_{a \in \mathscr{A}} \hat{Q}_t(a)$

It is easy to devise incremental formulas for updating averages with small, constant computation required to process each new reward. Given Q_n and the n-th reward, r_n, the new average of all n rewards can be computed by :

$$
\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} r_i \\
&= \frac{1}{n}\left(r_n + \sum_{i=1}^{n-1} r_i\right) \\
&= \frac{1}{n}\left(r_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} r_i\right) \\
&= \frac{1}{n}(r_n + (n-1)Q_n) \\
&= \frac{1}{n}(r_n + nQ_n - Q_n) \\
&= Q_n + \frac{1}{n}[r_n - Q_n]
\end{aligned}
$$

The Pseudocode for a complete bandit algorithm using incrementally computed sample averages and "ϵ-greedy action selection is shown in the algorithm below. The function bandit(a) is assumed to take an action and return a corresponding reward.

Algorithm 1: ϵ-greedy algorithm

```
1 Input: Initialize, for $a = 1$ to $k$
2 $Q(a) \leftarrow 0$
3 $N(a) \leftarrow 0$
4 while True do
5     $A \leftarrow \begin{cases} \text{argmax}_a Q(a) & \text{with probability } 1-\varepsilon \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
      $R \leftarrow$ bandit$(A)$
6     $N(a) \leftarrow N(a)+1$
7     $Q(A) \leftarrow N(A) + \frac{1}{N(A)}[R - Q(A)]$
8 end
9 Sortie: $Q(A)$
```

Notice that : for $\epsilon = 0$ the algorithm become the normal Greedy algorithm (No Exploration).

The Upper Confidence Bounds algorithm

Random exploration gives us an opportunity to try out options that we have not known much about. However, due to the randomness, it is possible we end up exploring a bad action which we have confirmed in the past (bad luck!). To avoid such inefficient exploration, one approach is to decrease the parameter ϵ in time and the other is to be optimistic about options with high *uncertainty* and thus to prefer actions for which we haven't had a confident value estimation yet. Or in other words, we favor exploration of actions with a strong potential to have a optimal value.
The Upper Confidence Bounds (UCB) algorithm measures this potential by an upper confidence bound of the reward value, $\hat{U}_t(a)$ so that the true value is below with bound $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$ with high probability. The upper bound $\hat{U}_t(a)$ is a function of $N_t(a)$; a larger number of trials $N_t(a)$ should give us a smaller bound $\hat{U}_t(a)$.
In UCB algorithm, we always select the greediest action to maximize the upper confidence bound:

$$a_t^{UCB} = argmax_{a\in\mathcal{A}}\hat{Q}_t(a) + \hat{U}_t(a)$$

Now, the question is *how to estimate the upper confidence bound ?*

If we do not want to assign any prior knowledge on how the distribution looks like, we can get help from **'Hoeffding's Inequality'** — a theorem applicable to any bounded distribution.
Let $X_1, \ldots, X_t$ be i.i.d. (independent and identically distributed) random variables and they are all bounded by the interval $[0, 1]$. The sample mean is $\overline{X}_t = \frac{1}{t}\sum_{\tau=1}^{t} X_\tau$ Then for u > 0, we have :

$$\mathbb{P}[\mathbb{E}[X] > \overline{X}_t + u] \leq e^{-2tu^2}$$

Given one target action a, let us consider:

- $r_t(a)$ as the random variables,
- $Q(a)$ as the true mean,
- $\hat{Q}_t(a)$ as the sample mean,
- And u as the upper confidence bound, $u = U_t(a)$

Then we have,

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

We want to pick a bound so that with high chances the true mean is blow the sample mean + the upper confidence bound. Thus $e^{-2tU_t(a)^2}$ should be a small probability. Let's say we are ok with a tiny threshold p:

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

One heuristic is to reduce the threshold p in time, as we want to make more confident bound estimation with more rewards observed. set $p = t^{-4}$ we get **UCB1** algorithm:

$$U_t(a) = \sqrt{\frac{2\log t}{N_t(a)}} \text{ and } a_t^{UCB1} = \arg\max_{a\in\mathcal{A}} Q(a) + \sqrt{\frac{2\log t}{N_t(a)}}$$

Python Implementation : TO BE DONE!

The plot of time step vs the cumulative regrets for :

- Greedy algorithm
- ϵ-Greedy algorithm
- The Upper Confidence Bounds algorithm (**UCB1**)

1.3 Basic Reinforcement learning terminology

As we define at the beginning **Reinforcement learning** is a branch of artificial intelligence that deals with an agent that perceives the information of the environment in the form of state spaces and action spaces, and acts on the environment thereby resulting in a new state and receiving a reward as feedback for that action. This received reward is assigned to the new state. Just like when we had to minimize the cost function in order to train our neural network, here the reinforcement learning agent has to maximize the overall reward to find the the optimal policy to solve a particular task.

How this is different from supervised and unsupervised learning?

In supervised learning, the training dataset has input features, $\mathbf{X}$, and their corresponding output labels, $\mathbf{Y}$. A model is trained on this training dataset, to which test cases having input features, $\mathbf{X}^{'}$, are given as the input and the model predicts $\mathbf{Y}^{'}$.

In unsupervised $\mathbf{X}$ learning, input features, $\mathbf{Y}$, of the training set are given for the training purpose. There are no associated Y values. The goal is to create a model that learns to segregate the data into different clusters by understanding the underlying pattern and thereby, classifying them to find some utility. This model is then further used for the input features $\mathbf{X}^{'}$ to predict their similarity to one of the clusters.

Reinforcement learning is different from both supervised and unsupervised. Reinforcement learning can guide an agent on how to act in the real world. The interface is broader than the training vectors, like in supervised or unsupervised learning. Here is the entire environment, which can be real or a simulated world. Agents are trained in a different way, where the objective is to reach a goal state, unlike the case of supervised learning where the objective is to maximize the likelihood or minimize cost.

Reinforcement learning agents automatically receive the feedback, that is, rewards from the environment, unlike in supervised learning where labeling requires time-consuming human effort. One of the bigger advantage of reinforcement learning is that phrasing any task's objective in the form of a goal helps in solving a wide variety of problems. For example, the goal of a video game agent would be to win the game by achieving the highest score. This also helps in discovering new approaches to achieving the goal. For example, when AlphaGo became the world champion in Go, it found new, unique ways of winning.

Terminologies and conventions

- **Agent:** This we create by programming such that it is able to sense the environment, perform actions, receive feedback, and try to maximize rewards.
- **Environment:** The world where the agent resides. It can be real or simulated. State: The perception or configuration of the environment that the agent senses. State spaces can be finite or infinite.
- **Rewards:** Feedback the agent receives after any action it has taken. The goal of the agent is to maximize the overall reward, that is, the immediate and the future reward. Rewards are defined in advance. Therefore, they must be created properly to achieve the goal efficiently.
- **Actions:** Anything that the agent is capable of doing in the given environment. Action space can be finite or infinite.
- **SAR triple:** (state, action, reward) is referred as the SAR triple, represented as $(\mathbf{s},\mathbf{a},\mathbf{r})$.
- **Episode:** Represents one complete run of the whole task.

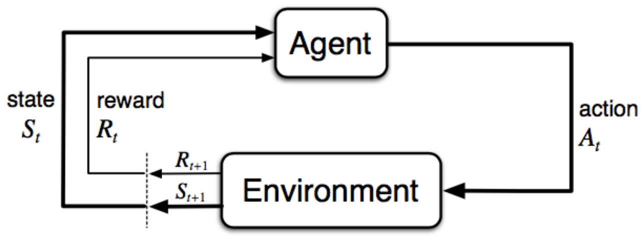


Figure 4. The agent-environment interaction in reinforcement learning

Every task is a sequence of SAR triples. We start from state S_t, perform action A_t and then the Agent receive a reward R_{t+1},and land on a new state S_{t+1}. The current state and action pair gives rewards for the next step. Since, S_t and A_t results in S_{t+1}, we have a new triple of (current state, action, new state), that is (S_t,A_t,S_{t+1}) or (s,a,s')

Optimality criteria

The optimality criteria are a measure of goodness of fit of the model created over the data. For example, in supervised classification learning algorithms, we have maximum likelihood as the optimality criteria. Thus, on the basis of the problem statement and objective optimality criteria differs. In reinforcement learning, our major goal is to maximize the future rewards. Therefore, we have two different optimality criteria, which are:

- ♣ **Value function:** To quantify a state on the basis of future probable rewards
- ♣ **Policy:** To guide an agent on what action to take in a given state

The value function for optimality

Agents should be able to think about both immediate and future rewards. Therefore, a value is assigned to each encountered state that reflects this future information too. This is called value function. Here comes the concept of delayed rewards, where being at present what actions taken now will lead to potential rewards in future. $V(s)$, that is, value of the state is defined as the expected value of rewards to be received in future for all the actions taken from this state to subsequent states until the agent reaches the goal state. Basically, value functions tell us how good it is to be in this state. The higher the value, the better the state. Rewards assigned to each $(s,a,s^{'})$ triple is fixed. This is not the case with the value of the state; it is subjected to change with every action in the episode and with different episodes too. One solution comes in mind, instead of the value function, why don't we store the knowledge of every possible state? The answer is simple: **it's time-consuming and expensive**, and this cost grows exponentially. Therefore, it's better to store the knowledge of the current state, that is, $V(s)$:

$$V_t(s) = \mathbb{E}[\text{all future rewards discounted } |S_t = s]$$

More details on the value function will be covered in chapter *The Markov Decision Process*

The policy model for optimality

Policy is defined as the model that guides the agent with action selection in different states. Policy is denoted as π. π is basically the probability of a certain action given a particular state:

$$\pi(a,s) = p(A_t = a|S_t = s)$$

Thus, a policy map will provide the set of probabilities of different actions given a particular state. The policy along with the value function create a solution that helps in agent navigation as per the policy and the calculated value of the state.

1.4 Reinforcement learning Taxonomy

In this section we are going to make a global vision on all the reinforcement learning algorithms and understanding of some of them, such as Q-learning and A3C algorithms. (Figure *)

Model-Free vs Model-Based RL

One of the most important branching points in an RL algorithm is the question of whether the agent has access to (or learns) a model of the environment. By a model of the environment, we mean a function which predicts state transitions and rewards.
The main upside to having a model is that it allows the agent to plan by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. Agents can then distill the results from planning ahead into a learned policy.
Algorithms which use a model are called **model-based** methods, and those that don't are called **model-free**. While model-free methods forego the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune. As of the time of writing this paper (April 2020), **model-free** methods are more popular and have been more extensively developed and tested than model-based methods.

Links to Algorithms in Taxonomy

- ♣ `Q-learning` : classic paper by Tsitsiklis and van Roy.
- ♣ `A2C/A3C` : (Asynchronous Advantage Actor-Critic): Mnih et al, 2016
- ♣ `PPO` : (Proximal Policy Optimization): Schulman et al, 2017
- ♣ `TRPO` : (Trust Region Policy Optimization): Schulman et al, 2015
- ♣ `DDPG` : (Deep Deterministic Policy Gradient): Lillicrap et al, 2015
- ♣ `TD3` : (Twin Delayed DDPG): Fujimoto et al, 2018
- ♣ `SAC` : (Soft Actor-Critic): Haarnoja et al, 2018
- ♣ `DQN` : (Deep Q-Networks): Mnih et al, 2013
- ♣ `C51` : (Categorical 51-Atom DQN): Bellemare et al, 2017
- ♣ `QR-DQN` : (Quantile Regression DQN): Dabney et al, 2017
- ♣ `HER` : (Hindsight Experience Replay): Andrychowicz et al, 2017
- ♣ `WorldModels` : Ha and Schmidhuber, 2018
- ♣ `I2A` : (Imagination-Augmented Agents): Weber et al, 2017
- ♣ `MBMF` : (Model-Based RL with Model-Free Fine-Tuning): Nagabandi et al, 2017
- ♣ `MBVE` : (Model-Based Value Expansion): Feinberg et al, 2018
- ♣ `AlphaZero` : Silver et al, 2017

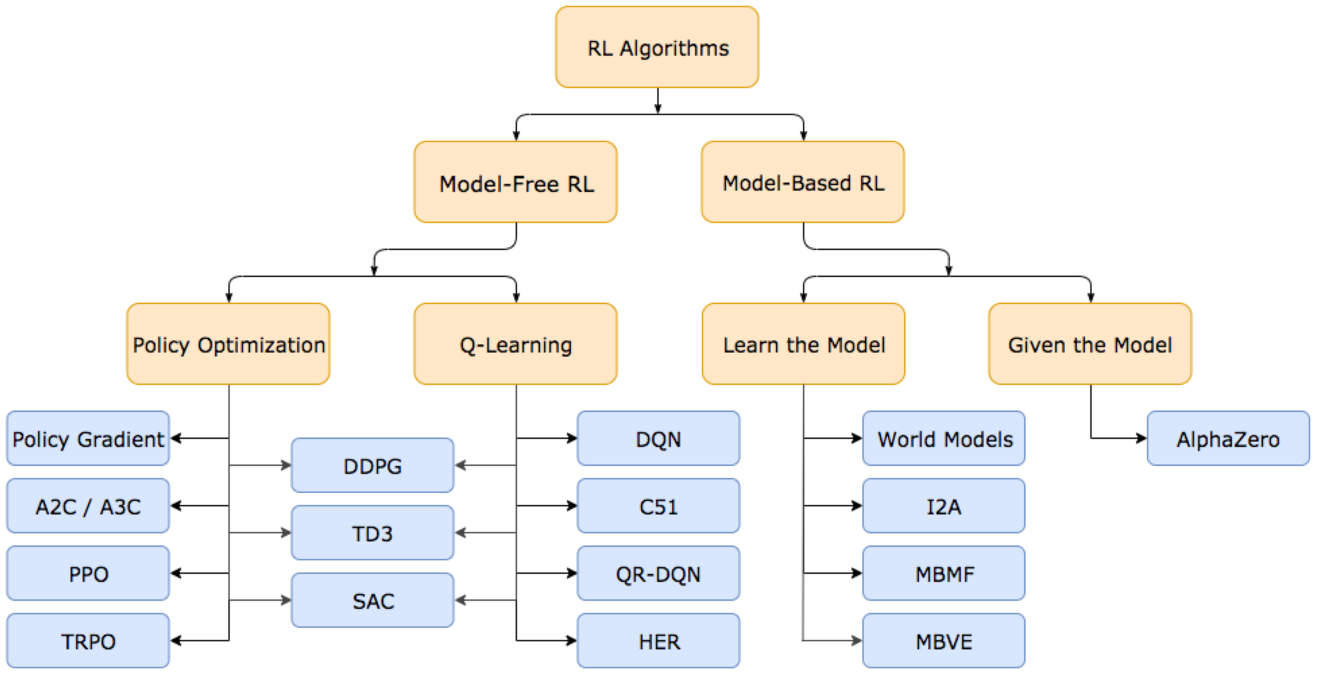


Figure 5. Basic Reinforcement learning Taxonomy.

General view of the Q-learning approach

Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations. For any *finite Markov decision process (FMDP)*, Q-learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning is an attempt to learn the value $Q(s,a)$ of a specific action given to the agent in a particular state. Consider a table where the number of rows represent the number of states, and the number of columns represent the number of actions. This is called a Q-table. Thus, we have to learn the value to find which action is the best for the agent in a given state.

The Pseudocode that ilustrate steps involved in Q-learning is shown in the algorithm below.

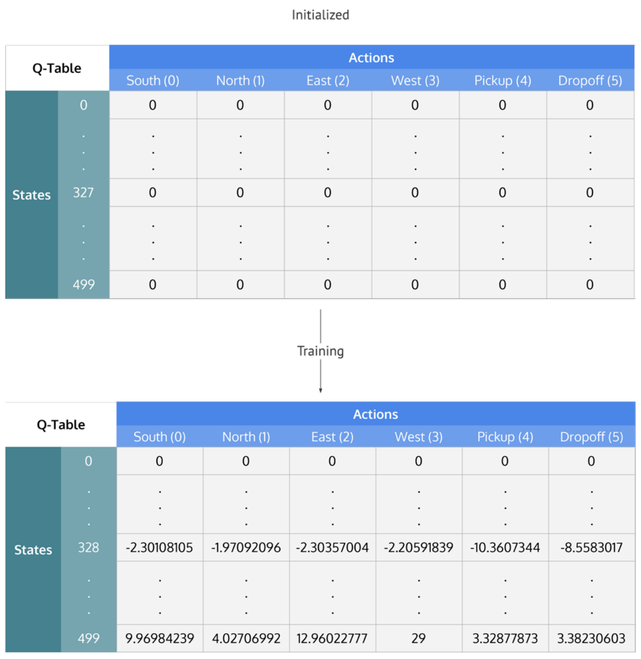
Algorithm 2: Steps involved in Q-learning

```
1 Input: $\gamma$, $\alpha$
2 Initialize the table of $Q(s,a)$ with uniform values
    (say, all zeros).
3 Observe the current state, $s$
4 Choose an action, $a$, by epsilon greedy or any other
    action selection policies, and take the action
5 As a result, $a$ reward, $r$, is received and a new state,
    $s^{'}$, is perceived
6 Update the $Q$ value of the $(s,a)$ pair in the table by
    using the following Bellman equation:
        $Q(s,a) = r + \gamma\left(\max\left(Q\left(s',a'\right)\right)\right)$
    where $\gamma$ is the discounting factor
7 Set the value of current state as a new state and
    repeat the process to complete one episode, that is,
    reaches the terminal state
8 Run multiple episodes to train the agent
9 Sortie: $Q - table$
```

To simplify, we can say that the Q-value for a given state, s, and action, a, is updated by the sum of current reward, r, and the discounted (γ) maximum Q value for the new state among all its actions. The discount factor delays the reward from the future compared to the present rewards. For example, a reward of 100 today will be worth more than 100 in the future. Similarly, a reward of 100 in the future must be worth less than 100 today. Therefore, we will discount the future rewards. Repeating this update process continuously results in Q-table values converging to accurate measures of the expected future reward for a given action in a given state.

When the volume of the state and action spaces increase, maintaining a Q-table is difficult. In the real world, the state spaces are infinitely large. Thus, there's a requirement of another approach that can produce $Q(s,a)$ without a Q-table. One solution is to replace the Q-table with a function. This function will take the state as the input in the form of a vector, and output the vector of Q-values for all the actions in the given state. This function approximator can be represented by a **neural network** to predict the Q-values. Thus, we can add more layers and fit in a deep neural network for better prediction of Q-values when the state and action space becomes large, which seemed impossible with a Q-table. This gives rise to the **Q-network** and if a deeper neural network, such as a convolutional neural network, is used then it results in a **deep Q-network** (DQN).

More details on Q-learning and deep Q-networks will be covered in , Q-Learning and Deep Q-Networks.


Initialized
Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	327	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	0	0	0	0	0	0

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Figure 6. Q-Learning Matrix Initialized and After Training

General view of the Asynchronous advantage actor-critic approach

The **A3C** algorithm was published in June 2016 by the combined team of *Google DeepMind* and *MILA*. It is simpler and has a lighter framework that used the *asynchronous gradient descent* to optimize the deep neural network. It was **faster** and was able to show good results on the multi-core CPU instead of GPU. One of A3C's big advantages is that it can *work on continuous as well as discrete action spaces*. As a result, it has opened the gateway for many new challenging problems that have complex state and action spaces.

We will discuss it at a high note here, but we will dig deeper in the next chapters. Let's start with the name, that is, **asynchronous advantage actor-critic (A3C)** algorithm and unpack it to get the basic overview of the algorithm:

♣ **Asynchronous:** In DQN (deep Q-learning), you remember we used a neural network with our agent to predict actions. This means there is one agent and it's interacting with *one environment*. What A3C does is *create multiple copies of the agent-environment* to make the agent learn more efficiently. A3C has a global network, and *multiple worker agents*, where each agent has its own set of network parameters and each of them interact with their copy of the environment simultaneously without interacting with another agent's environment. **The reason this works better than a single agent is that the experience of each agent is independent of the experience of the other agents.** Thus, the overall experience from all the worker agents results in diverse training.

♣ **Actor-critic:** Actor-critic combines the benefits of both **value iteration** and **policy iteration**. Thus, the network will estimate both a value function, $V(s)$, and a policy, $\pi(s)$, for a given state, s. There will be *two separate fully-connected layers* at the top of the function approximator neural network that will output the *value* and *policy* of the state, respectively. The agent uses the **value**, which acts as a **critic** to update the **policy**, that is, the intelligent **actor**.

♣ **Advantage:** Policy gradients used discounted returns telling the agent whether the action was good or bad. Replacing that with Advantage not only quantifies the the good or bad status of the action but helps in encouraging and discouraging actions better (we will discuss this deeper in next chapters).

2. Application of Reinforcement Learning Using OpenAI Gym

The OpenAI Gym provides a lot of virtual environments to train your reinforcement learning agents. In reinforcement learning, the most difficult task is to create the environment. This is where OpenAI Gym comes to the rescue, by providing a lot of toy game environments to provide users with a platform to train and benchmark their reinforcement learning agents.

In other words, it provides a playground for the reinforcement learning agent to learn and benchmark their performance, where the agent has to learn to navigate from the start state to the goal state without undergoing any mishaps.

Thus, in this chapter, we will be learning to understand and use environments from OpenAI Gym and trying to implement basic Q-learning and the Q-network for our agents to learn.

2.1 OpenAI Gym with Frozen-Lake environment

To understand the basics of importing Gym packages, loading an environment, and other important functions associated with OpenAI Gym, the detail of the example of a **Frozen Lake** environment is given in the this `Notebook`

Understanding an OpenAI Gym environment with Frozen-Lake example

The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.

Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend. The surface is described using a grid like the following:

- **SFFF:** (S: starting point, safe)
- **FHFH:** (F: frozen surface, safe)
- **FFFH:** (H: hole, fall to your doom)
- **HFFG:** (G: goal, where the frisbee is located)

The episode ends when you reach the goal or fall in a hole. You receive a reward of 1 if you reach the goal, and zero otherwise.

2.2 Programming an agent for Frozen Lake Game with Q-Learning

Now, let's try to program a reinforcement learning agent using Q-learning. Q-learning consists of a Q-table that contains Q-values for each state-action pair. The number of rows in the table is equal to the number of states in the environment and the number of columns equals the number of actions. Since the number of states is 16 and the number of actions is 4 , the Q-table for this environment consists of 16 rows and 4 columns.

As we described in the algorithm 2 (chapter 1), the steps involved in Q-learning are :

1. Initialize the Q-table with zeros (eventually, updating will happen with a reward received for each action taken during learning).
2. Updating of a Q value for a state-action pair, that is, $Q(s,a)$ is given by:

$$Q(s,a) \longleftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q\left(s',a'\right) - Q(s,a)\right]$$

with:

- s : current state
- a : action taken (choosing new action through epsilon-greedy approach)
- $s^{'}$: resulted new state
- $a^{'}$: action for the new state
- r : reward received for the action a
- α : learning rate, that is, the rate at which the learning of the agent converges towards minimized error
- γ : discount factor, that is, discounts the future reward to get an idea of how important that future reward is with regards to the current reward

3. By updating the Q-values as per the formula mentioned in step 2, the table converges to obtain accurate values for an action in a given state.

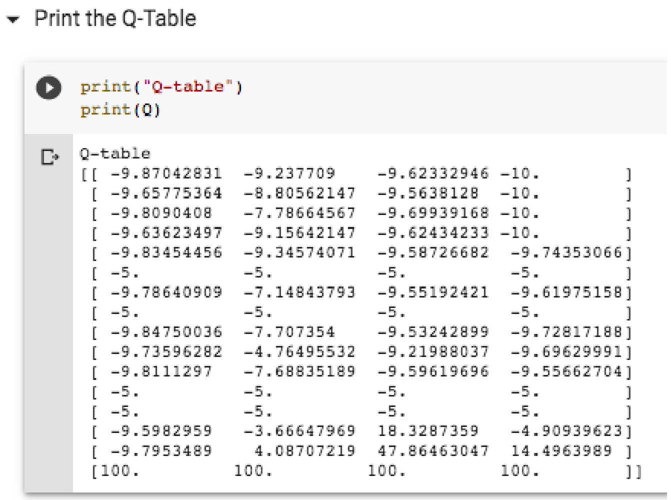
The **Epsilon-Greedy** that we have seen the the first chapter (Multi-armed bandit section) is a widely used solution to the explore-exploit dilemma. Exploration is all about searching and exploring new options through experimentation and research to generate new values, while exploitation is all about refining existing options by repeating those options and improving their values.

The Epsilon-Greedy approach was described in the algorithm 1 (chapter 1).

Eventually, after several iterations, we discover the best actions among all at each state because it gets the option to explore new random actions as well as exploit the existing actions and refine them.

Let's try to implement a basic Q-learning algorithm to make an agent learn how to navigate across this frozen lake of 16 grids, from the start to the goal without falling into the hole.

This Notebook contain the implementation code.

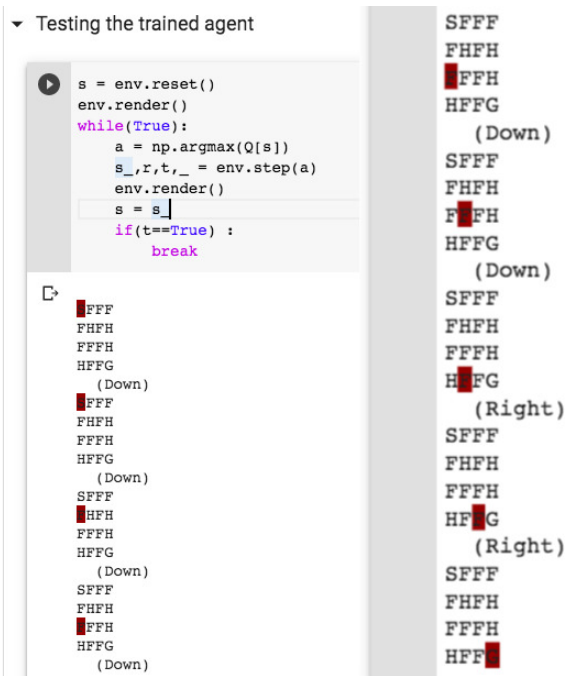


Print the Q-Table

```
print("Q-table")
print(Q)
```

```
Q-table
[[ -9.87042831  -9.237709    -9.62332946 -10.        ]
 [ -9.65775364  -8.80562147  -9.5638128  -10.        ]
 [ -9.8090408   -7.78664567  -9.69939168 -10.        ]
 [ -9.63623497  -9.15642147  -9.62434233 -10.        ]
 [ -9.83454456  -9.34574071  -9.58726682  -9.74353066]
 [ -5.           -5.           -5.           -5.        ]
 [ -9.78640909  -7.14843793  -9.55192421  -9.61975158]
 [ -5.           -5.           -5.           -5.        ]
 [ -9.84750036  -7.707354    -9.53242899  -9.72817188]
 [ -9.73596282  -4.76495532  -9.21988037  -9.69629991]
 [ -9.8111297   -7.68835189  -9.59619696  -9.55662704]
 [ -5.           -5.           -5.           -5.        ]
 [ -5.           -5.           -5.           -5.        ]
 [ -9.5982959   -3.66647969  18.3287359   -4.90939623]
 [ -9.7953489    4.08707219  47.86463047  14.4963989 ]
 [100.         100.         100.         100.        ]]
```

Figure 7. Q-Learning Table After Training



Testing the trained agent

```
s = env.reset()
env.render()
while(True):
    a = np.argmax(Q[s])
    s_,r,t,_ = env.step(a)
    env.render()
    s = s_
    if(t==True) :
        break
```

```
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
```

Figure 8. Agent test after training

2.3 Programming an agent for Frozen Lake Game with Q-Network approach

Maintaining a table for a small number of states is possible but in the real world, states become infinite. Thus, there is a need for a solution that incorporates the state information and outputs the Q-values for the actions without using the Q-table. This is where **neural network** acts a function approximator, which is trained over data of different state information and their corresponding Q-values for all actions, thereby, they are able to predict Q-values for any new state information input. The neural network used to predict Q-values instead of using a Q-table is called **Q-network**.

Let's use a single neural network that takes state information as input, where state information is represented as a one hot encoded vector of the $1 \times$ number of states shape (here, 1×16) and outputs a vector of the $1 \times$ number of actions shape (here, 1×4). The output is the Q-values for all the actions. With the options of adding more hidden layers and different activation functions, a Q-network definitely has many advantages over a Q-table. Unlike a Q-table, in a Q-network, the Q-values are updated by minimizing the loss through **backpropagation**. The loss function is given by:

$$\begin{aligned} \text{Loss } &= \sum \left(Q_{\text{target}} - Q_{\text{predicted}}\right)^2 \\ Q(s,a)_{\text{target}} &= r + \gamma \max_{a'} Q(s',a') \end{aligned}$$

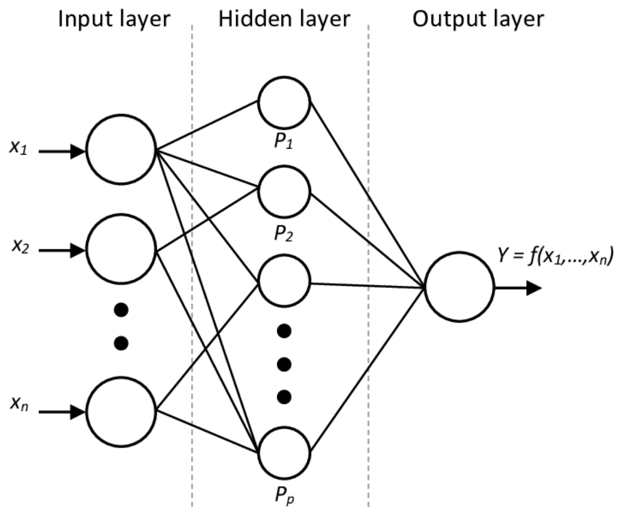


Figure 9. The neural nets architecture, n = 16

2.4 Comparison between Q-Table and Q-Network approaches

There is a cost of stability associated with both Q-learning and Q-networks. There will be cases when with the given set of hyperparameters of the Q-values are not converge, but with the same hyperparameters, sometimes converging is witnessed. This is because of **the instability** of these learning approaches.

In order to tackle this, a better initial policy should be defined (here, the maximum Q-value of a given state) if the state space is small. Moreover, hyperparameters, especially learning rate, discount factors, and epsilon value, play an important role. Therefore, these values must be initialized properly.

In general, Q-networks provide more flexibility compared to Q-learning, owing to increasing state spaces. A deep neural network in a Q-network might lead to better learning and performance. As far as playing Atari using Deep Q-Networks.

3. Markov Decision Processes

3.1 Definitions

A Markov decision process is a tuple $(S, A, \{P_{sa}\}, \gamma, R)$ where:

- S : is a set of **states**. (For example, in autonomous helicopter fight, S might be the set of all possible positions and orientations of the helicopter).
- A : is a set of **actions**. (For example, the set of all possible directions in which you can push the helicopter's control sticks.)
- $T(s, a, s^{'}) \sim P_{\text{sa}}$: are the state transition probabilities. For each state $s \in S$ and action $a \in A$, P_{sa} is a distribution over the state space. We'll say more about this later, but briely, P_{sa} gives the distribution over what states we will transition to if we take action a in state s
- $\gamma \in [0, 1)$, is called the **discount factor**.
- $R : S \times A \mapsto \mathbb{R}$ is the **reward function**. (Rewards are sometimes also written as a function of a state S only, in which case we would have $R : S \mapsto \mathbb{R}$
- $\pi(s) \rightarrow a$ is a function that takes the state as an input and outputs the action to be taken.

General idea on the dynamics of an MDP

We start in some state s_0, and get to choose some action $a_0 \in A$, as a result of our choice, the state of the MDP randomly transitions to some successor state s_1, drawn according to $\mathbb{P}_{s_0 a_0}$. Then, we get to pick another a_1, and so on $\cdots$

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \ldots$$

Upon visiting the sequence of states $s_0, s_1, \cdots$ with actions $a_0, a_1, \cdots$ our total payoff is giving by:

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots$$

So our main goal in reinforcement learning is to choose actions over time so as to maximize theexpected value of the total payoff:

$$\mathrm{E}\left[R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots\right]$$

3.2 Assumptions

The sequence of rewards play an important role in finding the optimal policy for an MDP problem, but there are certain assumptions that unveil how a sequence of rewards implements the concept of delayed rewards.

The infinite horizons

The first assumption is the infinite horizons, that is, the infinite amount of time steps to reach goal state from start state. Therefore,

$$\pi(s) \rightarrow a$$

The policy function doesn't take the remaining time steps into consideration. If it had been a finite horizon, then the policy would have been,

$$\pi(s, t) \rightarrow a$$

where t is the time steps left to get the task done. Therefore, without the assumption of the infinite horizon, the notion of policy would not be stationary, that is, $\pi(s) \rightarrow a$, rather it would be $\pi(s, t) \rightarrow a$

Utility of sequences

The utility of sequences refers to the overall reward received when the agent goes through the sequences of states. It is represented as $U(s_0, s_1, s_2 \cdots)$ where $s_0, s_1, s_2 \cdots$ represents the sequence of states.

The second assumption is that if there are two utilities, $U(s_0, s_1, s_2 \cdots)$ and $U(s_0^{'}, s_1^{'}, s_2^{'} \cdots)$ such that the start state for both the sequences are the same and,

$$U(s_0, s_1, s_2 \cdots) > U(s_0^{'}, s_1^{'}, s_2^{'} \cdots)$$

then

$$U(s_1, s_2 \cdots) > U(s_1^{'}, s_2^{'} \cdots)$$

This assumption is called **the stationary of preferences**, and the following equation satisfies this assumption.

$$U(s_0, s_1, s_2 \cdots) = \sum_{t=0}^{\infty} y^t R(s_t)$$

3.3 The Bellman equations

Since the optimal π^* policy is the policy that maximizes the expected rewards, therefore,

$$\pi^* = \text{argmax}_\pi\, E\left[\sum_{\infty}^{t=0} \gamma^t R(s_t) | \pi\right]$$

where $E\left[\sum_{\infty}^{t=0} \gamma^t R(s_t) | \pi\right]$ means the expected value of the rewards obtained from the sequence of states agent observes if it follows the π policy. Thus argmax_π outputs the policy π that has the highest expected reward.

Similarly, we can also calculate **the utility of the policy of a state**, that is, if we are at the s state, given a policy, then, the utility of the policy for the s state, that is, $U^\pi(s)$ would be the expected rewards from that state onward:

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi, s_0 = s\right]$$

The immediate reward of the state, that is, $R(s)$ is different than the utility of the $U(s)$ state) (that is, the utility of the

optimal policy of the $U^{\pi}(s)$ state) because of the concept of delayed rewards. From now onward, the utility of the $U(s)$ state will refer to the utility of the optimal policy of the state, that is, the $U^{\pi^*}(s)$ state.
Moreover, the optimal policy can also be regarded as the policy that maximizes the expected utility. Therefore,

$$\pi^* = \text{argmax}_a \sum_{s'} T\left(s,a,s'\right) U\left(s'\right)$$

where, $T(s,a,s^{'})$ is the transition probability, that is, $P(s^{'}|s,a)$ and $U(s^{'})$ is the utility of the new landing state after the a action is taken on the s state.
$T\left(s,a,s'\right) U\left(s'\right)$ refers to the summation of all possible new state outcomes for a particular action taken, then whichever action gives the maximum value of $T\left(s,a,s'\right) U\left(s'\right)$ that is considered to be the part of the optimal policy and thereby, the utility of the 's' state is given by the following **Bellman equation**,

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T\left(s,a,s'\right) U\left(s'\right)$$

where, $R(s)$ is the immediate reward and
$\max_a \sum_{s'} T\left(s,a,s'\right) U\left(s'\right)$ is the reward from future, that is, the discounted utilities of the s state where the agent can reach from the given s state if the action, a, is taken.

Solving the Bellman equation to find policies

Value iteration :
Say we have some n states in the given environment and if we see the Bellman equation,

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T\left(s,a,s'\right) U\left(s'\right)$$

we find out that n **states** are given; therefore, we will have n **equations** and n unknown but the max_a function makes it **non-linear**. Thus, we cannot solve them as linear equations.
Therefore, in order to solve:

1. Start with an arbitrary utility
2. Update the utilities based on the neighborhood until convergence, that is, update the utility of the state using the Bellman equation based on the utilities of the landing states from the given state.

Iterate this multiple times to lead to the true value of the states. This process of iterating to convergence towards the true value of the state is called **value iteration**.

For the terminal states where the game ends, the utility of those terminal state equals the immediate reward the agent receives while entering the terminal state.
Policy iteration :

The process of obtaining optimal utility by iterating over the policy and updating the policy itself instead of value until the policy converges to the optimum is called policy iteration. The process of policy iteration is as follows:

1. Start with a random policy' π_0
2. For the given π_t policy at iteration step t, calculate $U_t = U_t^{\pi}$ by using the following formula:

$$U_t(s) = R(s) + \gamma \sum_{s'} T\left(s, \pi_t(s), s'\right) U_{t-1}\left(s'\right)$$

3. Improve the π_{t+1} policy by:

$$U_t(s) = R(s) + \gamma \sum_{s'} T\left(s, \pi_t(s), s'\right) U_{t-1}\left(s'\right)$$

3.4 Training the FrozenLake-v0 environment using MDP

This is about a gridworld environment in OpenAI gym called FrozenLake-v0, discussed previously. We implemented Q-learning and Q-network to get the understanding of an OpenAI gym environment. Now, we will implement **value iteration** to obtain the utility value of each state in the FrozenLake-v0 environment. Let the state representation be as follows:

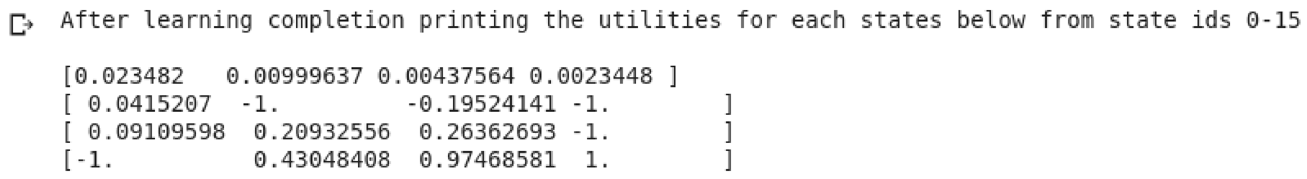
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

This Notebook contain the implementation code.
results: The start state of our agent is 0. Let's start from $s = 0$, so when $U[s = 0] = 0.023482$, the action can be either UP, DOWN, LEFT, or RIGHT.
At $s = 0$ if:

- action UP is taken, the $s_{new} = o$, therefore, $u[s_{new}] = 0.023482$
- action DOWN is taken, $thes_{new} = 4$, therefore, $u[s_{new}] = 0.0415207$
- action LEFT is taken, the $s_{new} = o$, therefore, $u[s_{new}] = 0.023482$
- action RIGHT is taken, the $s_{new} = 1$, therefore, $u[s_{new}] = 0.00999637$

The max is $u[s_{new} = 15] = 1.0$, therefore, the action taken is RIGHT and $s_{new} = 15$.

Therefore, our policy contains DOWN, DOWN, RIGHT, DOWN, RIGHT, and RIGHT to reach from $s = 0$(start state) to $s = 15$(goal state) by avoiding hole states $(5,7,11,12)$.



```
After learning completion printing the utilities for each states below from state ids 0-15

[0.023482   0.00999637 0.00437564 0.0023448 ]
[ 0.0415207  -1.         -0.19524141 -1.        ]
[ 0.09109598  0.20932556  0.26362693 -1.        ]
[-1.          0.43048408  0.97468581  1.        ]
```

Figure 10. The utilities for each states

References

[1] Richard S. Sutton and Andrew G. Barto [2018], Reinforcement Learning: An Introduction, *MIT Press, Cambridge, MA*.