Reinforcement Learning (State-of-the-Art)

Soufiane Fadel[1],Patrick Boily[1]

Abstract
Reinforcement learning is an area of machine learning that focuses on how an agent might act in an environment in order to maximize some given reward. Reinforcement learning algorithms study the behaviour of agents in such environments and learn to optimize their behaviour. What distinguishes reinforcement learning from supervised learning is that only partial feedback is given to the learner about its predictions.

Reinforcement learning is of great interest due to the large number of practical applications that it can be used to address, ranging from problems in artificial intelligence to operations research and statistical quality monitoring in control engineering.

In this report, we introduce tools that can be applied to game development (A.I.), industrial control, search engines, robotics, personalized medical treatment, quantitative finance, finding optimal architecture in deep learning networks, etc.

Keywords
Reinforcement learning, Markov decision processes, value functions, Bellman equation, model-free reinforcement learning, Q—learning, function approximation.

[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. Introduction

Reinforcement learning (RL) is a computational approach used to to understand and automate goal-directed learning and decision making [1]. Unlike other algorithmic approaches, RL puts the emphasis on learning through the direct interaction of an **agent** with its **environment**, without requiring full supervision of the agent or "complete" models of its environment.

1.1 About Reinforcement Learning

According to (some) experts, RL is the first field to seriously address the computational issues arising from agent-environment interactions to achieve long-term goals. It uses the formal framework of **Markov decision processes** (see Section 3) to define the interactions between a learning agent and its environment in terms of

- **states**
- **actions**
- **rewards**

This framework is intended to be a simplified representation of the essential features of the **artificial general intelligence problem**, namely, to build a computer which can "understand or learn any task that a (generic) human being can" [5].[1]

The concepts of **value** and **value function** are key to many RL methods, providing the tools for efficient searching in the space of **policies** (or strategies), and helping to distinguish RL methods from **evolutionary methods**, where searches are conducted directly in the policy space, guided by evaluations of entire policies.

History of Reinforcement Learning

Historically, the term "reinforcement" (in the context of animal learning) seems to have first appeared in an early 20th century English translation of Pavlov's work on conditioned reflexes, where reinforcement is described 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 [1].

In pyschological circles, reinforcement can sometime includes weakening of behaviour, and omission/termination of stimulii could also be considered to be reinforcers, as long as the change in behaviour remains once they are removed from the situation.

Two famous classical experiments in animal behaviour reinforcement form the basis of modern RL, in the sense that agents are expected to mimic the behaviour of **Pavlov's dog** and **Skinner's pigeon** when faced with reinforcers.

[1]Unsurprisingly, this problem is quite difficult to solve: experts cannot even agree on what constitutes an intelligence, although most experts agree that a so-called intelligence should be able to

- reason and use strategy to solve puzzles and make judgements under uncertainty;
- represent knowledge (commonsense or not);
- plan;
- learn;
- communicate in a natural language, and
- integrate all these skills towards common goals [5].

Other capabilities may include the ability to see, sense, move and manipulate objects in the universe where intelligent behaviour takes place, form emergent mental images and concepts, and exhibit autonomous behaviour. While some systems can perform some of these tasks, none can do so at human levels yet, at least not according to various tests that have been proposed:

- the **imitation game** (Turing) – fairly consistently fooling a human observer into thinking that a hidden interlocutor is human/intelligent;
- the **coffee test** (Wozniak) – entering an average home and figuring out how to brew a pot of coffee, from scratch;
- the **robot college student test** (Goertzel) – enrolling into a university program, taking and passing (occasionally failing?) classes as a human student would, in order to graduate with a degree;
- the **employment test** (Nilsson) – joining the workforce and performing economically useful tasks, at least as well as (some) humans do in the same position [5], and
- the **stand-up comedy test** (Ebert) – making a reasonable proportion of humans laugh, somewhat consistently.

In the first experiment, Pavlov noticed that presenting a dog with food causes the animal to salivate. He set up a repeated experiment: whenever food is presented to a dog, a bell rings. Over time, the bell ringing becomes a reinforcer, so that the dog ends up salivating upon hearing the bell, whether food is presented simultaneously or not.

In the second experiment, Skinner set up a box in which a lever can be pressed in order to drop some birdfeed. A pigeon learned by trial-and-error that pressing the lever will provide it with food (note that the device can be made more complicated to condition the delivery of food *via* an audible or visual signal; the results are the same).

In the first experiment, there is no trial-and-error interaction: the dog learns a new behaviour through repeated exposure over which it has no control. In the second experiment, the pigeon is **rewarded** by pressing the lever; the food reward **reinforces** the action (pressing the lever), which is subsequently performed more often in the hopes of further rewards. Additionally, the pigeon's behaviour can be modified by modifying the reward.

1.2 Multi-Armed Bandits

We start by studying the evaluative aspect of reinforcement learning in a simple setting, which is used to introduce a number of basic learning methods, eventually leading to the **full reinforcement learning problem**.

The $K-$Armed Bandit Problem

The **multi-armed bandit problem** is a classic problem that demonstrates the **exploration/exploitation** dilemma central to RL.

On the one hand, if we have learned the full environment information, we can find the best action to take in any situation simply by exploring the space of consequences, using a brute force approach.

In practice, however, only incomplete information is (usually) available, so that any action taken cannot be guaranteed to be the best one: we need take into account the risk of making a poor decision due to less than optimal information.

Under an **exploitation** approach, we pick the best option available given the information at our disposal; under an **exploration** approach, we could follow a sub-optimal action path if doing so could help us gather more information about the environment (and thus make better decisions down the road) – the best long-term strategy may involve short-term sacrifices.

For instance, a crossword solver who only ever puts down an answer when they are certain that it is the right answer (exploitation approach) will make few mistakes, but they may not always be successful in the long-term goal of completing the puzzle, whereas a cross-word solver who sometimes puts down an answer even when they are not

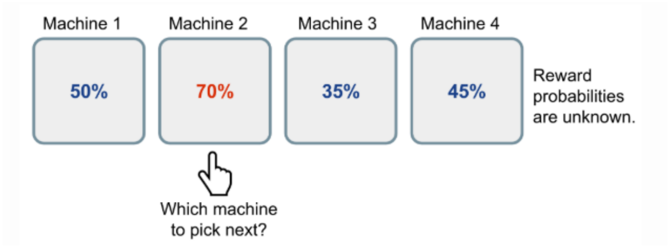

Machine 1
Machine 2
Machine 3
Machine 4
50%
70%
35%
45%
Reward probabilities are unknown.
Which machine to pick next?

Figure 1. An illustration of a Bernoulli 4—armed bandit. The reward probabilities are **unknown** to the player.

certain that it is the right answer (exploration trial) will definitely make more mistakes (and their filled grids will be messier!), but they are likely to have a higher success rate overall as those additional answers that have been laid down with less than 100% certainty may still open up answers for other clues.

In the K—**Armed Bandit Problem**, we find ourselves in a casino facing K slot machines[2], each of which is configured with an unknown constant probability of rewarding the player after one turn. The probabilities may be different from one slot machine to the next. Assuming that all machines are free and that we can switch from one to another after any turn, what strategy should we use to achieve the highest long-term reward (see Figure 1)?

The situation is easier to analyze if we assume that we can continue playing indefinitely, as the restriction of a finite number of trials introduces a different type of exploration problem.[3]

The naîve approach is to continue playing with one machine for multiple rounds in order to use the **law of large numbers** to estimate its "true" reward probability, then to move on to another machine and estimate its "true" reward probability, then to move on to another machine and so on.

This approach has the double disadvantage of being quite wasteful, while not even guaranteeing that it will provide the best long-term reward.

Formalism

A **Bernoulli** K**—armed bandit** is a tuple $\langle A, R, \theta\rangle_K$, where:

- the K reward probabilities are $\theta = \{\theta_1, \dots, \theta_K\}$;
- at each time step t, the action $a_t \in A$ consists in picking a machine $i \in \{1, \dots, K\}$ and receiving a reward $r_t \in R$;
- A is a set of actions, each referring to the interaction with a specific machine – the **value** of action a is the expected reward, $Q(a) = \mathrm{E}[r|a] = \theta$; if we take action $a_t \in A$ at time step t on the ith machine, then $Q(a_t) = \theta_i$, and
- R is a reward function $R : A \to [0, 1]$ – in the case of a Bernoulli bandit, we observe a reward r in stochastic fashion, that is, at the time step t,

$$r_t = R(a_t) = \begin{cases} 1 & \text{with probability } Q(a_t) \\ 0 & \text{otherwise} \end{cases}$$

Such a process is an instance of a **Markov decision process** with an empty set of states $S = \varnothing$ (see Section 3).

The goal of the K—armed bandit is to maximize the cumulative reward

$$\text{cumulative reward} = \sum_{t=1}^{T} r_t, \quad T \in \mathbb{N} \cup \{\infty\}.$$

This problem is equivalent to minimizing the **potential regret** (or loss) of not picking the optimal action.

The optimal reward probability θ^* of the optimal action a^* is thus:

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

The **loss function** is the total regret we have, on average, by not selecting the optimal strategy/action up to time step T:

$$\mathscr{L}_T = \mathrm{E}\left[\sum_{t=1}^{T}\left(\theta^* - Q(a_t)\right)\right].$$

Bandit Strategies

Based on how we explore the action space, there are several ways to solve the multi-armed bandit problem:

- the **greedy algorithm** leaves no room for exploration;
- the ε—**greedy algorithm** requires some random exploration, while
- the **upper confidence bounds algorithm** uses smart exploration.

The ε—Greedy algorithm

The ε—**greedy algorithm** usually takes the best available action, but will occasionally use random exploration. The value of a target action a is estimated according to past experience, by averaging the rewards associated with a when it was taken at previous time steps:

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

where $\mathbb{1}$ is a **binary indicator function** and $N_t(a)$ records how often the action a has been selected so far, i.e.

$$N_t(a) = \sum_{\tau=1}^{t} \mathbb{1}[a_\tau = a].$$

When using the greedy algorithm, the target action with the largest value $\hat{Q}_t(a)$ is ultimately selected at time step t.

[2]Which are known, in the vernacular, as *one-armed bandits*.

[3]If the number of allowed turns is smaller than the number of slot machines, for instance, we cannot even try every machine to estimate the reward probabilities and we have to behave intelligently with respect to limited knowledge and resources.

With small probability ε, the ε—greedy algorithm will instead select a random action; with probability $1-\varepsilon$ (which should turn out to be most of the time), it will revert to the greedy algorithm action $\hat{a}_t^* = \arg\max_{a\in A}\hat{Q}_t(a)$ (note that the random action could turn out to be a^*).

If, for each action a, we keep track of rewards and values at every time step where it is used, we can easily devise incremental formulas that update the reward average with very little computational power. Given an action value $Q_{t_n}(a)$ with corresponding reward $r_{t_n}(a)$, and assuming that action a has been taken $n-1$ times in the previous time steps, at times $t_1,\ldots,t_{n-1}$, the (new) average of all rewards for a up to time step t_n is

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n} r_{t_i}(a) &= \frac{1}{n}\left(r_{t_n}(a)+\sum_{i=1}^{n-1} r_{t_i}(a)\right)\\ &= \frac{1}{n}\left(r_{t_n}(a)+(n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} r_{t_i}(a)\right)\\ &= \frac{1}{n}\left(r_{t_n}(a)+(n-1)Q_{t_n}(a)\right)\\ &= \frac{1}{n}\left(r_{t_n}(a)+nQ_{t_n}(a)-Q_{t_n}(a)\right)\\ &= Q_{t_n}(a)+\frac{1}{n}\left[r_{t_n}(a)-Q_{t_n}(a)\right]\end{aligned}$$

The pseudocode for a complete ε—greedy K—armed bandit algorithm using incrementally computed sample averages is shown below; the function $bandit(a)$ takes an action $a\in\{l,\ldots,K\}$ and returns the corresponding reward.

Algorithm 1: The ε—Greedy Algorithm

```
1  Input: a threshold $T$ for time steps, a set of actions
     $A=\{1,\ldots,K\}$
2  Initialize: for $a \in A$
3  $Q(a) \leftarrow 0$
4  $N(a) \leftarrow 0$
5  for $1 \le t < T$ do
6      $a \leftarrow \begin{cases} \arg\max_{a\in A}\{Q(a)\} & \text{with probability } 1-\varepsilon \\ 1 & \text{with probability } \varepsilon/K \\ \vdots & \vdots \\ K & \text{with probability } \varepsilon/K \end{cases}$
7      $R \leftarrow bandit(\alpha)$
8      $N(\alpha) \leftarrow N(\alpha)+1$
9      $Q(\alpha) \leftarrow Q(\alpha)+\frac{1}{N(\alpha)}[R-Q(\alpha)]$
10 end
11 Output: $Q(1),\ldots,Q(K)$
```

Note that for $\varepsilon = 0$, the algorithm collapses to the regular greedy algorithm, in which no exploration is conducted.

The Upper Confidence Bounds Algorithm

Random exploration provides an opportunity to try options that we may not know as much about; however, we may end up exploring avenues with no exits.[4]

In order to reduce the risk of inefficient exploration, we may reduce the value of the parameter $\varepsilon = \varepsilon_t$ over time, to reflect growing confidence in the "tried, tested, and true"; another approach could be to remain (artificially) optimistic about options for which the action value has not been estimated with confidence yet – we might want to favour "exploration of actions with a strong potential to have a optimal value." [1]

The **upper confidence bounds** (UCB) algorithm measures this potential *via* an upper confidence bound on the action/reward value, $\hat{U}_t(a)$, so that the true value is bounded by

$$Q(a) \le \hat{Q}_t(a)+\hat{U}_t(a)$$

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 yield a smaller bound $\hat{U}_t(a)$.

The UCB algorithm selects the greediest action which maximizes the upper confidence bound:

$$a_t^{UCB} = \arg\max_{a\in A}\left\{\hat{Q}_t(a)+\hat{U}_t(a)\right\}.$$

How is the estimate $\hat{U}_t(a)$ computed? If no prior knowledge on the distribution of rewards is assumed, we can use **Hoeffding's Inequality** to obtain the estimate.

Let $X_1,\ldots,X_t$ be independent and identically distributed (i.i.d.) random variables on the interval $[0,1]$, with sample mean

$$\overline{X}_t = \frac{1}{t}\sum_{\tau=1}^{t} X_\tau.$$

Then, for $u>0$, we have:

$$P\left[\mathrm{E}[X] > \overline{X}_t + u\right] \le e^{-2tu^2}.$$

In our context, given a target action a, we use:

- $X_\tau = r_\tau(a)$ are the random variables;
- $E[X_\tau] = Q(a)$ is the true mean of the action value;
- $\overline{X} = \hat{Q}_t(a)$ is the sample mean, and
- $u = U_t(a)$ is the upper confidence bound.

Then Hoeffding's Inequality becomes

$$P\left[Q(a) > \hat{Q}_t(a)+U_t(a)\right] \le e^{-2tU_t(a)^2}.$$

If $e^{-2tU_t(a)^2}$ is small, then $Q(a) < \hat{Q}_t(a)+U_t(a)$ with large probability. For a tiny threshold p,

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

[4]"Barking up the wrong tree," as they say in English.

As we may want to reduce the threshold $p = p_t$ in time in order to become more confident in the UCB as more rewards are observed, we could set $p_t = t^{-\ell}$, $\ell \in \mathbb{N}$. For $\ell = 4$, we obtain the **UCB1** algorithm:

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

$$a_t^{UCB1} = \arg\max_{a\in A}\left\{Q(a) + \sqrt{\frac{2\log t}{N_t(a)}}\right\}.$$

1.3 Basic Reinforcement Learning Terminology

Recall that RL is a branch of AI that deals with agents that perceive their environment through the **state space** and act on it through the **action space**, resulting in a (potentially) different state and receiving a reward as feedback for that action, which is assigned to the new state. In supervised learning, we train models by minimizing a cost function; in RL, the agent maximizes the overall reward to find the **optimal task-solving policy**.

Reinforced vs. Supervised/Unsupervised Learning

In **supervised learning** (SL), the training set

$$\mathbf{Z} = [\mathbf{X}, \mathbf{Y}] \in \mathbb{M}_{n,p} \times \mathbb{M}_{m,p}$$

has **input features** $X = \{X_1, \ldots, X_p\}$ and corresponding **output labels** $Y = \{Y_1, \ldots, Y_m\}$.

A model f is trained on the training set $\mathbf{Z}$ and predictions $\mathbf{Y}^*$ are obtained for the test dataset $\mathbf{Z}' = [\mathbf{X}', \mathbf{Y}']$, such that $\mathbf{Y}^* = f(\mathbf{X}')$; good models are those for which the predictions $\mathbf{Y}^*$ and the true output labels $\mathbf{Y}'$ are "close".[5]

In **unsupervised learning** (UL), only the input values $\mathbf{X}$ of the training set are used for training purposes: there are no associated output label values $\mathbf{Y}$.

As an example, the goal could be to build a model that learns to segregate the data into different **clusters** by identifying natural groups found in the data. This model could then further be used on test cases $\mathbf{X}'$ to predict their similarity (and eventual probability of assignment) to the clusters.[6]

RL takes a different path, as it guides agents on how to act in their environment; the interface includes more than simply training vectors (as is the case in both SL and UL). The training objective is for agents to **reach a target state**, not to maximize a likelihood or minimize a cost as in SL.

RL agents automatically receive feedback, in the form of **rewards from the environment**, and activities such as labeling (which are time-consuming for humans) are not necessary.

[5]This can be done in a multitude of ways [7]. A model's validity also depends on the dataset: no model can consistently outperform random predictions when it comes to predicting coin tosses, for example. Note too that the model f may contain stochastic elements.

[6]The absence of labels to "verify" the cluster assignments, say, creates a fair amount of ambiguity, which is compounded by the rich variety of algorithms and measures of similarity that can be used in practice [8].

One of the main advantages of RL over SL/UL is that **phrasing any task's objective in the form of a goal** helps to solve a wide variety of problems. As an example, the goal of a video game agent might be to win the game by achieving the highest possible score, by reaching the finish line first, or by discovering new ways to achieve these goals.[7]

Terminologies and Conventions

We have used various terms to describe (rather vaguely) a number of concepts; we will now define the terms as they will be used in the rest of this work:

- an **agent** is programmed to "sense" its environment, perform actions (selected from an action space), receive feedback (in the form of a numerical reward or penalty), and to attempt to maximize rewards or minimize penalties;
- the agent's **environment** is the world in which it resides; it can be real (as in robotics) or simulated (virtual);
- the agent senses its environment by reading its **state**, which is to say, the configuration in which the environment finds itself at any particular moment (parameters, position of objects, occupied locations, etc.); the space of the environment's states can be finite (and small, such as in the game of tic-tac-toe, or large, such as in the games of Go and chess), or infinite (such as the position of a Turing machine scanning head);
- anything that the agent is capable of doing in its environment is called an **action**; available actions at any given time may depend on the state of the environment; the space of actions can be finite (as the list of available moves in tic-tac-toe, Go, or chess would be) or infinite (such as selecting a value in the interval $[0, 1]$ and adding it to an initialized state value to obtained a new state);
- the feedback that the agent receives based on the action it has taken is given in the form of a numerical **reward** or **penalty**; the agent's objective is to maximize (resp. minimize) the **overall reward** (resp. penalty), that is, the immediate and the future reward; rewards are defined before the agent is let loose into the environment, and must be created properly in order for the agent to achieve its goal(s) efficiently;
- the combination of a state, an action, and a reward, is referred to as an **SAR triple**, and is represented by the tuple $(\mathbf{s}, \mathbf{a}, \mathbf{r})$, and, finally,
- an agent's complete run for the task (from initialization to completion, assuming that the task can be completed) is a RL **episode**.

[7]When AlphaGo, an A.I. system designed by DeepMind, beat Lee Sedol, one of the world's best Go player, in a best-of-five series in 2016, it found new and unorthodox ways of winning. Interestingly, so did Sedol who won one game against AlphaGo using a "miraculous" move dubbed "God's Touch" by players around the world. It would seem that humans and A.I. agents can (still) learn from one another in the world of RL [9].

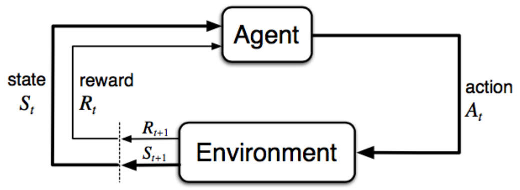


Figure 2. The agent-environment interaction in RL [1].

Every RL task can thus be represented as a sequence of SAR triples: starting from a state S_t, the agent performs action A_t yielding the new state S_{t+1}, for which it receives a reward R_{t+1}.[8]

Since, S_t and A_t result in S_{t+1}, we could also represent the steps as a SAS triple (current state, action, new state), i.e. (S_t, A_t, S_{t+1}) or (s, a, s') (see Figure 2).

Optimality Criteria and Validation

Optimality criteria are measures of goodness-of-fit for the model created over the data. In SL models of classification/regression, for instance, the maximum likelihood is often used as an optimality criterion.

Optimality criteria differ on the basis of the problem statement and the task objective(s). In RL, the main goal is to maximize the future rewards (or minimize the future penalties). As such, there are two requirements:

- we need a **value function** in order to quantify states on the basis of future probable rewards, and
- a **policy** in order to guide an agent on what action to take in a given situation (state).

Value Function Smart agents should be able to "think" both about immediate rewards and about future rewards. A value needs to be assigned to each encountered state to reflect this future information, with the help of the **value function**. This is where the concept of **delayed rewards** comes into play.

The **value** $V_t(s)$ of the state s at time t is the expected value of future rewards for all actions taken on the state s until the agent reaches the target state (i.e. the objective has been met). States with high values are states from which it is more likely that the target state can eventually be reached.

While the reward assigned to each SAS triple (s, a, s') is fixed, that is not the case for $V_t(s)$; it is subject to change with every action within the episode and from one episode to the next, since $V_t(s)$ depends on t as well as s.

Consequently, it might seem preferable to store the knowledge of every possible state, rather than the value function, but that is not a practical idea for anything but the simplest RL systems: that approach is both **time-consuming** and expensive, with the cost growing exponentially with the problem complexity.

[8]The current state-action pair gives rewards for the next step.

Formally, we have

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

More details on the value function and on discounts are provided in the other sections.

Policy Model A **policy** is a model that guides the agent when the time comes for action selection in different states. The policy is typically denoted by π, the probability of a certain action being taken in a given state:

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

A **policy map** must thus provide the set of probabilities for the different actions given a particular state.

Using an appropriate policy, together with the value function, will help the agent navigate its environment in an efficient manner.

1.4 Reinforcement Learning Taxonomy

Recently, a number of RL algorithms and approaches have become more popular, such as Q–learning and the A3C algorithm (a taxonomy is provided in Figure 3). We discuss some of the highlights in the rest of this section.

Model-Free RL vs. Model-Based RL

One of the most fundamental differentiators for various types of RL algorithms is whether the agent has access to (or learns) a model of the environment on the way to meeting its objective(s), or whether it acts on its environment without trying to **predict** state transitions and rewards (or penalties).

The main benefit of having access to 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 the available options, eventually distilling the results into a learned policy.

Algorithms which use such a model are called **model-based** methods, while those that do not are **model-free**. Model-free methods tend to be less efficient, as they do not allow for "thinking" ahead, but they also tend to be easier to implement and fine-tune. As of the writing of this report (July 2020), model-free methods remain more popular and have been more extensively developed and tested than model-based methods. The references provided in [10] can be found at the end of this report.

Q–Learning Overview

Q–**learning** [11] is a model-free RL algorithm, whose goal is to **learn a policy**, namely, how an agent should act under a set of circumstances. It does not require a model of the environment, and it can handle issues with stochastic transitions and rewards, without requiring adaptation.

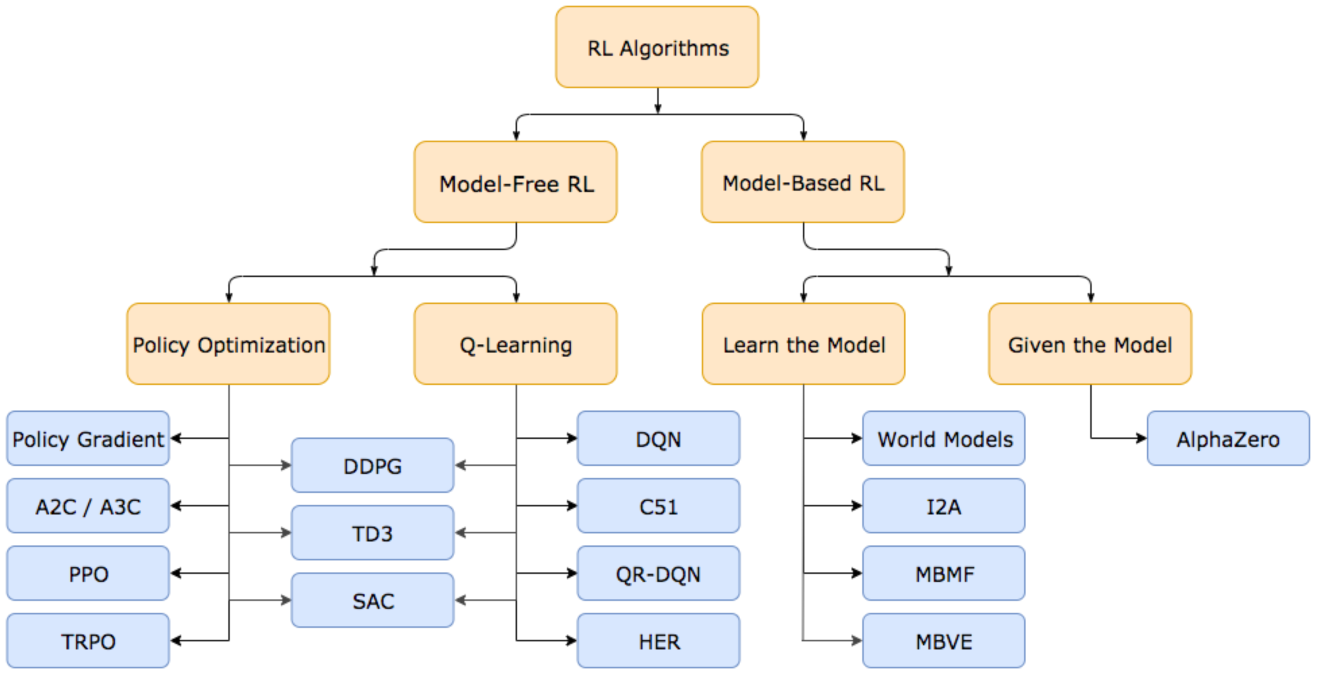


Figure 3. Basic RL taxonomy [10].

For any *finite Markov decision process* (FMDP, see Section **??**), Q—learning finds an optimal policy which maximizes the expectation of the total reward over all successive steps, beginning with the current one.

Q—learning attempts to learn the value $Q(s,a)$ of a specific action given to the agent in a particular state, by constructing a Q—**table**, an $|S|\times|A|$ matrix with one row per state and one column per action. The Q—table can then be used to determine which action is best for an agent in a given state (see pseudocode in Algorithm 2).

Simply speaking, the Q—value for a given state-action pair (s,a) is updated by adding the current reward r to the discounted (with factor γ) maximum Q—value for the new state s' (resulting from action a on the current state s) among all possible actions; this sum is then modified by the learning rate $\alpha$and added to a modified version of the old Q—value (see step 6 in the algorithm).

The **discount factor** $\gamma\in[0,1]$ balances the impact of future and present rewards; small values of γ make the agent "short-sighted" (favouring short-term rewards) while large values make it stride for "long-term rewards." While taking the longer-term view is usually a good approach, conceptually-speaking, there are technical difficulties associated with too large of a discount factor.

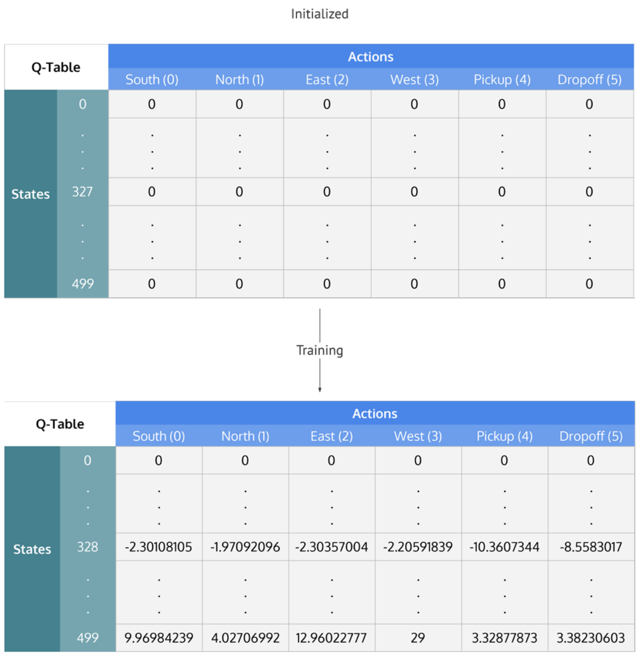
The **learning rate** $\alpha\in(0,1]$ determines what effect old information has on the updating process; small values of α make the agent run an exploitation process (focusing mainly on past knowledge, which would be good for stochastic problems), while large values make it run an exploration process (focusing mainly on the most recent information, which would be good for deterministic problems).

Algorithm 2: Q—Learning Steps

```
1  Input: actions $A$, states $S$, discount factor $\gamma\in[0,1]$,
     learning rate $\alpha\in(0,1]$, initial condition $\beta\in[0,1]$
2  Initialize: $Q(s,a)\leftarrow\beta$
3  while current state $s$ is not a final state do
4      Select an action $a\in A$ using the $\varepsilon$—greedy
         method (or any other action selection policy),
         and apply the action to $s$
5      Record the reward $r$ and the resulting state $s'$
6      Update the $Q$—value for the $(s,a)$ pair via the
         Bellman equation:
         $Q(s,a)\leftarrow(1-\alpha)Q(s,a)+\alpha\left[r+\gamma\cdot\max_{a'}\{Q(s',a')\}\right]$
7      Set $s=s'$
8  end
9  Run multiple episodes to train the agent
10 Output: $Q$—table
```

The **initial condition** β also has an effect; large values of β correspond to "optimistic initial conditions" and encourage exploration. Various strategies exist to **reset** the initial conditions, such as setting $Q(s,a)=r$ the first time an action a is taken.[9]

[9]Note that $\gamma=\gamma_t$, $\alpha=\alpha_t$ and $\beta=\beta(s,a)$ need not be constant.


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 4. Illustration of Q—table updating: pre- and post-training [28].

Multiple episodes may be required since, in any given episode, it is conceivable that an action-state pair (s, a) will not be encountered by the agent, which would leave the Q—table without an updated value for this pair.

Seeing as the Bellman equation also requires knowledge of a full Q—table row when updating $Q(s, a)$, running multiple episodes help guarantee that the Q—table values converge to accurate representations of the expected future reward for a state-action pair (s, a).

For continuous state and action spaces, $|S| = |A| = \infty$; even for discrete systems, maintaining a Q—table becomes very costly when $|S|$ and $|A|$ are large. **Surface fitting** (or function learning) can produce the Q—values without a Q—table.

Any reasonable surface fitting method can be used to approximate $Q(s, a)$; when $Q(s, a)$ is approximated by a neural network, we speak of a Q—**network**; when $Q(s, a)$ is approximated by a **deeper neural network**, such as a convolutional neural network (CNN) [27], the algorithm is called a **deep** Q—**network** (DQN) [18–20].

A3C Overview

The **asynchronous advantage actor-critic** (A3C) [12] approach is a recent, simple, and fast framework that uses asynchronous gradient descent to optimize the underlying deep neural network. One of its main advantages is that it can work both on continuous and on discrete action spaces, opening a gateway for many new challenging problems that have complex state and action spaces.

We provide a deeper discussion of the algorithm in later Section 2; for now, we simply unpack the algorithm's name in order to get a basic overview of what it does [29].

At it's core, A3C is **asynchronous**, meaning that it creates multiple copies of the agent and its environment, but the agents do not interact with each other; each agent uses a different DQN (or the same network but with different parameters) to learn the Q—function $Q(s, a)$. Since all agents are independent of one another, the overall experience results in diverse training, and procedures such as bagging or ensemble learning can be used to provide better Q—function estimates.

A3C is also **actor-critic**, combining the benefits of both **value iteration** and **policy iteration**. The underlying DQN estimates both a value function $V(s)$, and a policy $\pi(s)$ for a given state s. It contains *two separate and fully-connected layers* at the top of the function approximator neural network; these layers output $\hat{V}(s)$ and $\hat{\pi}(s)$, respectively. The agent then uses the **value**, which acts as a **critic**, to update the **policy** of an intelligent **actor**.

Finally, the **advantage** of an action a on state s is the difference between the Q—value $Q(s, a)$ for the state-action pair and the value of the state $V(s)$. This metric allows not only for the quantification of an action as "good" or "bad," but it provides a way to encourage (or discourage, as the case may be) actions better than the discounted returns of policy gradient methods.

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

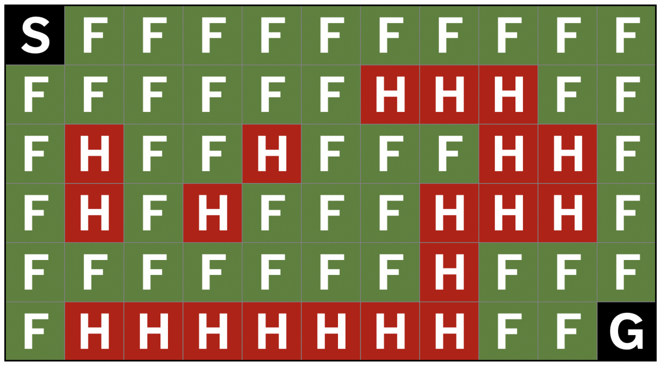
The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding 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. [6]

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.



S	F	F	F	F	F	F	F	F	F	F
F	F	F	F	F	F	H	H	H	F	F
F	H	F	F	H	F	F	F	H	H	F
F	H	F	H	F	F	F	H	H	H	F
F	F	F	F	F	F	F	H	F	F	F
F	H	H	H	H	H	H	H	F	F	G

Figure 5. FrozenLake

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

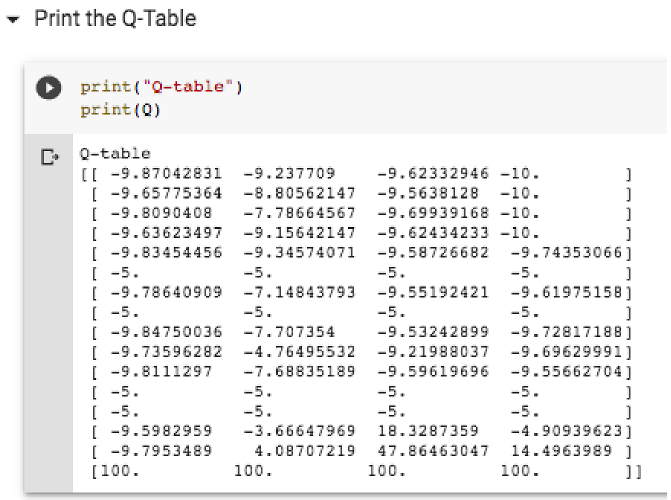
3. By updating the Q-values as per the formula mentioned in step 2, the table converges to obtain accurate values for an action in a given state.

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 (section 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.

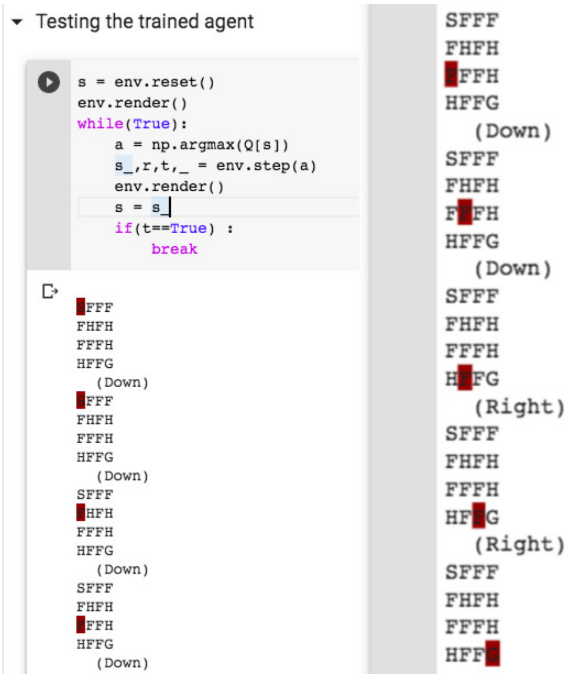
This Notebook contain the implementation code.


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 6. 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
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
```

Figure 7. 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:

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

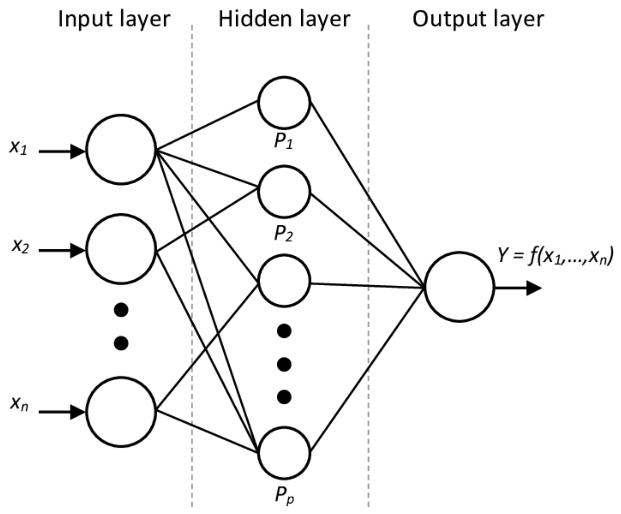


Figure 8. The neural nets architecture, n = 16

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 $R : S \mapsto \mathbb{R}$
- $\pi(s) \to a$ 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,

$$\pi(s) \to a$$

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,

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

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, $\pi(s) \to a$, rather it would be $\pi(s, t) \to a$

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:

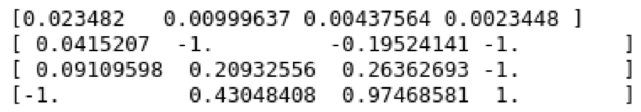
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

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, $s_{new} = 4$, 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 9. The utilities for each states

References

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

[2] Silva et. al, Reinforcement Learning, *UCL*.

[3] White et. al, Fundamentals of Reinforcement Learning, *University of Alberta*.

[4] Sayon Dutta - Reinforcement Learning with TensorFlow A beginner's guide to designing self-learning systems with TensorFlow and OpenAI Gym-Packt Publishing (2018).

[5] **Artificial general intelligence** on Wikipedia.

[6] **openai-gym** official gym-openai library description.

[7] Aggarwal, C.C. (ed.) [2015], *Data Classification*, CRC Press.

[8] Aggarwal, C.C., Reddy, C.K. (eds.) [2015], *Data Clustering*, CRC Press.

[9] Wood, G. [2016], **In Two Moves, AlphaGo and Lee Sedol Redefined the Future**, Wired Magazine.

[10] **Spinning Up**, on OpenAI.com.

[11] `Q-learning`: classic paper by Tsitsiklis & van Roy.

[12] `A2C/A3C` (Asynchronous Advantage Actor-Critic): Mnih et al, 2016

[13] `PPO` (Proximal Policy Optimization): Schulman et al, 2017

[14] `TRPO` (Trust Region Policy Optimization): Schulman et al, 2015

[15] `DDPG` (Deep Deterministic Policy Gradient): Lillicrap et al, 2015

[16] `TD3` (Twin Delayed DDPG): Fujimoto et al, 2018

[17] `SAC` (Soft Actor-Critic): Haarnoja et al, 2018

[18] `DQN` (Deep *Q*—Networks): Mnih et al, 2013

[19] `C51` (Categorical 51—Atom DQN): Bellemare et al, 2017

[20] `QR-DQN` (Quantile Regression DQN): Dabney et al, 2017

[21] `HER` (Hindsight Experience Replay): Andrychowicz et al, 2017

[22] `WorldModels`: Ha and Schmidhuber, 2018

[23] `I2A`: (Imagination-Augmented Agents) Weber et al, 2017

[24] `MBMF` (Model-Based RL with Model-Free Fine-Tuning): Nagabandi et al, 2017

[25] `MBVE` (Model-Based Value Expansion): Feinberg et al, 2018

[26] `AlphaZero`: Silver et al, 2017

[27] Patel, S., Pourhasan, R., Boily, P. [in preparation], *Deep Learning and Applications*, Data Science Report Series, Data Action Lab.

[28] Kansal, S., Martin, B., **Reinforcement Q-Learning from Scratch in Python with OpenAI Gym**, LearnDataSci.com.

[29] Phadnis, A. [2018], **A3C – What is it, and What I Built**, on GoodAudience.com.

[30] Szita, I. [2012], *Reinforcement Learning in Games*, in Reinforcement Learning: State-of-the-Art, Wiering, M and can Otterlo, M. (eds.), Springer Verlag.

[31] Heinz, S. [2019], **Using Reinforcement Learning to Play Super Mario Bros. on NES Using TensorFlow**, on towardsdatascience.com.

[32] Jung, A. [2016], **AI playing Super Mario World with Deep Reinforcement Learning**, on YouTube.com.

[33] Zychlinski, S. [2019], **Qrash Course: Reinforcement Learning 101 & Deep Q Networks in 10 Minutes**, on towardsdatascience.com.

[34] Mahmood, R., Korenkevych, D., Komer, B., Bergstra, J. [2018], **Setting up a Reinforcement Learning Task with a Real-World Robot**, on YouTube.com. Paper at **https://arxiv.org/abs/1803.07067**.

[35] [2008], **Reinforcement learning for a robotic soccer goalkeeper**, by YouTube user africlean.