Vanier College

Reinforcement Learning

J. Goi, D. Hosi, L. Nguyen, A. Hui, J. Trinh, A. Julsain, J. Kalsi

28 May 2020

Abstract

Reinforcement Learning is the science of making sequences of decisions with the goal of obtaining maximum reward. The decision maker could be any agent equipped with the ability to influence its environment. One branch of Reinforcement learning is based on algorithms which iteratively estimate the Value Function of States (V) or the Action-State Value Function (Q). Dynamic Programming algorithms based on the Bellman are able to compute the optimal Value Function and also optimal policies. The incorporation of neural networks in Reinforcement Learning-based machines provides them the capability to learn good decision making policies and allows efficient approximation of parametrized Policy Functions. The Reinforcement Learning community has proposed algorithms such as REINFORCE and Proximal Policy Optimization to solve very complicated sequential decision making problems in an efficient fashion. In this article we will present the foundational concepts of reinforcement learning and illustrate some of the most popular algorithms.

Contents

1 Markov Decision Processes, Policies and Value Functions

1.1 Finite Markov Decision Processes

Reinforcement learning trains the machine into learning to take action in order to maximize the rewards, special numerical values. A specific way to do so is through the finite Markov decision processes, or simply named finite MDPs. The main concept of the latter is the agent-environment interaction. As illustrated in *Reinforcement Learning* by Richard S. Sutton and Andrew G. Barto, figure 1 shows an agent, which is the body that learns and makes decisions, acting with the purpose to obtain the best rewards in the long-term run. This agent constantly interchanges with its environment, which is everything that surrounds the agent. But to be more precise, anything that is outside the agent's power to modify randomly is considered its environment. For instance, if the agent is a person, by looking at the interaction between a person and his environment, everything is considered the environment except the person's mind which is the agent. The action the agent takes at time t, $A_t \in \mathrm{A(s)}$, leads it in a state, a situation given by the environment, $S_t \in \mathrm{S}$, and will give it a reward depending on A(s). These actions are anything the agent decides it wants to learn how to produce, and the states are all useful information that can help to make the product. The agent then processes S_t to act once again, A_{t+1}, and will receive in return R_{t+1}, while also ending up in a new state, S_{t+1}. The agent, following this MDP, generates the following sequence, so-called trajectory [6]:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$$

where the subscripts indicate the time step where the state, the action and the reward happen. The time steps can not only be fixed intervals of real time, but can also be arbitrary consecutive stages of decision making and acting.

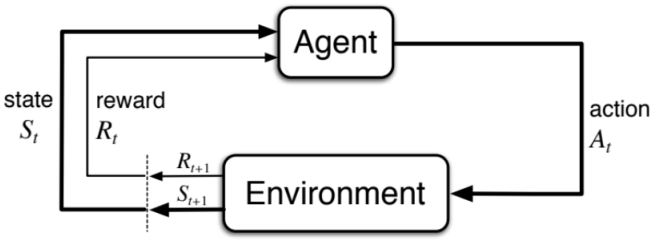


Figure 1: Interaction between the agent and its environment in a Markov decision process [6]

The states (S), actions (A), and rewards (R) are sets of finite number of elements. At time t, R_t and S_t are randomized sets composed of randomized

values, $s' \in S$ and $r \in R$, where each of the latter have a probability of happening depending only on the previous state, S_{t-1}, and action, A_{t-1}:

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

for all s, $s \in S$, $r \in R$, and $a \in A(s)$ as given by Richard S. Sutton and Andrew G. Barto in *Reinforcement Learning* just like the following equations in chapters 1 and 2. Note that by only considering S_{t-1}, the agent's strategy of act will not be hindered since S_{t-1} should include enough information about all critical aspects of the past agent-environment interactions. In fact, such state possesses the Markov property.

The environment's dynamics of the MDP is given by this conditional fundamental probability function p of four arguments: $S \times R \times S \times A$ that gives a result $\in [0, 1]$. To be more specific, this function portrays the probability distribution caused by the agent's choice of s and a[6]:

$$\sum_{s' \in S} \sum_{r \in R} p(s', r|s, a) = 1, \tag{2}$$

for all $s \in S$, $a \in A(s)$.

From the fundamental dynamics function, many other equations can be derived which offer the necessary information one desires, such as the state-transition probabilities, a three-argument function p : $S \times S \times A$ giving results in the interval [0,1] :

$$p(s'|s, a) \doteq Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in R} p(s', r|s, a); \tag{3}$$

the expected rewards $\in \mathbb{R}$ for state-action pairs, a two-argument function r : $S \times A$:

$$r(s, a) \doteq \mathbb{E}[\ R_t|S_{t-1} = s, A_{t-1} = a] \ = \sum_{r \in R} r \sum_{s' \in S} p(s', r|s, a); \tag{4}$$

and the expected rewards for state-action-next-state triples $\in \mathbb{R}$, a three-argument function r : $S \times A \times S\times$:

$$r(s, a, s') \doteq \mathbb{E}[\ R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] \ = \sum_{r \in R} r \frac{p(s', r|s, a)}{p(s'|s, a)}. \tag{5}$$

Rewards are outside of the agent's control, in its environment. However, the agent is capable to figure out its rewards to a certain limit as they can be computed from a function of its actions and states in which they are taken; the agent is not totally clueless about its environment, it just possesses a limited absolute control over its environment. In practice, when the agent has decided

its main task by deciding on its states, actions, and rewards, a boundary between the agent and the environment is then set.

In summation, the MDP is basically a real situation put in an simplified, but very informative, abstract form to explain the processes of how to achieve a goal through the continuous interactions between the agent and its environment with the use of three signals: one for the decisions taken by the agent (the actions), one that informs the agent of the circumstances surrounding its choices (the states), and one to announce the goal of the agent (the rewards).

Let's consider an example taken from the same reference as the figures used in chapters 1 and 2, which will also be the case for the following examples in the first two chapters of this work: a pick-and-place robot. Using reinforcement learning with MDP, the dynamics of a robot arm will be controlled as it does its pick-and-place task in a looping manner. Thus, the agent is the specific system of the robot that controls the robotic arm. Moreover, the ideal motion of the arm is to be fast and smooth. To do so, the agent will require information about its current state, meaning its current positions and velocities of the mechanical linkages, and will then act accordingly by applying the required amount of voltage into its joints, for instance. Hence, whenever the arm acts the ideal way, the robot will acquire a positive reward, +1 for instance. However, if the agent ends up doing the opposite, it will obtain a negative reward, which will in return encourage the robot arm to not act this way.

1.2 Goal, maximization of returns and episodes

As was mentioned above, the goal of the agent is to maximize the total amount of rewards it can obtain from its environment. This can be done by adding all the rewards it has accumulated in the long run. The reward hypothesis states that the agent's purpose is to maximize the expected values of all the accumulated sum of received rewards. This method can be demonstrated through the example of a recycling robot collecting empty cans in a mobile environment. By programming the robot to receive a +1 reward only when it collects a can, it will learn to find and collect empty cans to recycle. It is also possible to give the robot negative rewards in a way that every time it bumps into an obstacle or when it picks up the wrong object, it will receive a reward of -1. At the end, the agent eventually learns to maximize its rewards. However, it does not mean it will always accomplish the goal that we intended it to achieve. The use of a reward signal is important in order to communicate what is our objective to the robot. In fact, what we want is to set up the robot's reward in a way that in maximizing them, it will also attain what we want accomplished.

Now, we must formally define this concept in by looking at what exactly we want to maximize. Let the rewards received after time t be denoted by $R_{t+1}, R_{t+2}, R_{t+3}...$ Then, if we are looking to maximize the return where the

return is denoted by G_t, we get an equation where the return, G_t is equal to the sum of the rewards[6].

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{6}$$

where T is the final time step. The agent's objective is to maximize its total rewards in each episode, which is when the agent-environment interaction breaks naturally into subsequences. An episode ends when the interaction enters a terminal state and restarts at a starting state. Each episode starts and ends independently from the previous ones. Therefore, the episodes can result in different rewards for the different outcomes. Episodic tasks are task with episodes of this kind. In episodic tasks, the set of all non-terminal state is designated by S and the set of terminal states is denoted by S^T. The time of termination T usually varies from episode to episode.

In some applications, the agent-environment interaction is not cut into well defined episodes and continues without limit. Tasks that are made of one never-ending episode are called continuing tasks. In addition, another important concept is the one of discounting where the agent chooses actions that will maximize the sum of its discounting rewards it receives over the future. By choosing A_T, it maximizes the expected discounted return[6].

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k} \tag{7}$$

with a discount rate γ that is $0 \leq \gamma \leq 1$. The discount rate determines the present value of future rewards. A reward is worth less when received immediately than in the future. If $\gamma < 1$, the infinite sum in the equation above has a finite value only if the reward sequence is bounded. If $\gamma = 0$, the agent is only concerned in optimizing the immediate reward. However, in general, performing to maximize the immediate reward limit the future rewards which can lead to less return. As γ approaches 1, the agent also considers the future reward more frequently.

$$G_t \doteq R_{t+1} + \gamma R_{t+1} \tag{8}$$

This equation functions for all time steps $t < T$ if we define G_T=0,and it can easily be used to calculate the returns from reward sequences. It is important to note that even if the return is a sum on an infinite number of terms, it will still remain finite if the reward is nonzero and constant.

Let's take the example of the balancing pole. The goal in this task is to keep the pole, placed vertically on the cart, from falling over while the latter is in movement. It is considered a failure once the pole falls more than a given angle or when the cart goes off track. When this happens, the pole is reset to start vertically. This could be treated as an episodic task where each episode consists of the repeated attempts to maintain the pole vertical. In this case, a +1 reward for each time step would be given when it is successful, and the return at each

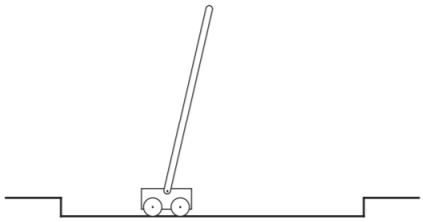


Figure 2: Pole-Balancing example

time would be the amount of steps until failure. Therefore, it means a continual success would give a return of infinity. On the other hand, if it is treated as a continuing task with discounting, a reward of -1 would be attributed for each failure and 0 for the rest of the time. Consequently, the return would be related to $-\gamma^{k-1}$. In both cases, the maximization of the return is only attainable by having the pole balanced as long as possible.

1.3 Combination of episodic and continuing task

The previous concepts discussed about the episodic task and continuing task separately, but there is a way to talk about both cases simultaneously. To do so, we must make some changes on what we know for episodic tasks. Instead of taking one long sequence of time steps, we must consider series of episodes where each episode is a finite sequence of time steps. For each episode, we start the steps at 0. Thus, we must refer to $S_{t,i}$, which is the state representation at time t of episode i, although in practice, it is normally written as S_t. However, we usually do not look at each episodes individually when using this way. Instead, what is generally considered is a specific episode or something that applies to all of the episodes. In addition, we have discussed the return of a sum of finite as well as infinite amount of terms. These two can be combined if we consider that the termination of the episode leads to an absorbing state that always goes back to itself and generates a reward of 0. Thus, it is possible to define return without considering episodes that are unnecessary and having the possibility of $\gamma = 1$ if the sum stays defined. Subsequently, the combination of episodic and continuing tasks can be written in the following way[6]:

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{9}$$

that includes the possibility of $T = \infty$ or $\gamma = 1$.

1.4 Policies and Value Function

The policy is a mapping from some states to the probabilities of selecting each possible action given that state. An agent following π policy at time t will give $\pi(a|s)$, which is the probability that $A_t = a$ is obtained if we get $S_t = s$ where $a \in A(s)$ for each $s \in S$. Value functions follow some policy. They are functions of states that measures the overall expected reward. Let's designate v_π as the value function of a state s following a policy π, which is also the expected return when starting in s and following π afterwards. v_π is defined mathematically as:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \tag{10}$$

for all $s \in S$ where $E\pi[]$ represent the expected return of some variable only if the agent follows policy π, and t is in any time step. v_π is the state-value function for policy π. Similarly, let's designate $q_\pi(s, a)$ as the value of taking action a in state s under a policy π, which can also be described as the expected return from state s, taking action a and following policy π afterwards[6].

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{11}$$

where q_π is named action-value function for policy π.

It is possible to estimate v_π and q_π from experience as empirical averages. Monte Carlo method is an estimation method that consists of taking the average of many random samples of actual returns. As an example, if we take an agent following policy π, where, for each state it encounters, it keeps an average of the actual returns that have followed that state, then there will be a convergence of the average to the state's value, v_π(s), as the amount of times the state that comes across the agent approaches infinity. Similarly, if we have separate averages for each action from each state, these averages will also converge to the action value, $q_\pi(s, a)$. If there are a lot of states, instead of taking separate averages for each state, the agent would have to keep v_π and q_π as parameterized functions as well as to modify the parameters to fit with the observed returns.

2 Optimal policies and the Bellman equation

2.1 The Bellman equation

For all policies and states, a consistency condition can be found, through some computations of the definition of $v_\pi(s)$, between the value of the current state, s, and its possible following states, s' :

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[\ G_t | S_t = s] \\
&= \sum_a pi(a|s) \sum_{s',r} p(s', r|s, a)[\ r + \gamma v_\pi(s')]
\end{aligned} \tag{12}
$$

for all $s \in S$, and where $a \in A(s)$, $s' \in S$, or S^+ if the agent is going through episodic tasks, and $r \in R$. In fact, Equation (12) is named the *Bellman equation*, for v_π, which is the only solution to this equation. The Bellman equation is a summation of all values of the variables, a, s', and r; the computation of the probabilities of each set of three different variables, $\pi(a|s)p(s', r|s, a)$; a product of the latter with the result obtained in the brackets; and lastly, a summation of all possibilities to finally obtain an expected value. The Bellman equation can be further explained with the use of backup diagrams, diagrams that showcase the correlations of its constituents and portray algorithms and models in reinforcement learning. These correlations are the basis of the backup operations which are at the core of understanding reinforcement learning methods as they send back value functions from successor states, or state-actions pairs, to a state, or a state-action pair.

Consider figure 3:

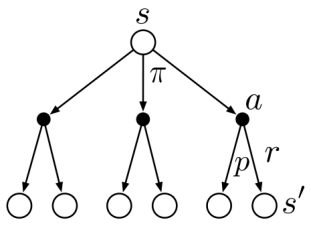


Figure 3: Backup diagram for v_π

The open circles are states whilst the solid ones are state-action pairs. Actions are represented by arrows starting from state s. Figure 3 suggests that from state s, the agent, following a policy π, will pick an action a of A(s) - only three are illustrated - that the environment will respond with a next state, $s' \in S$, with a reward, r, that depends on its dynamics, as suggested by the function p.

A backup diagram for the Bellman equation for action values will look like this[6]:

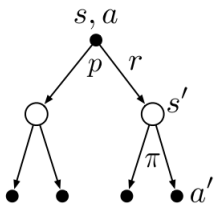


Figure 4: Backup diagram for q_π

2.2 Optimal Policies and Optimal Value Functions

As mentioned right at the beginning of chapter 1, reinforcement learning is used for the purpose to maximize the amount of rewards in the long-term run, and it can be concluded that to do so, the agent has to follow the right policy, the optimal policy. But more precisely, an optimal policy is a policy π whose expected returns are greater or equal to a policy π' for all states, and always exists. All optimal policies, as there may be more than one, are denoted all together as π_*. They possess the same state-value function, also known as an optimal state-value function v_* :

$$v_*(s) \doteq \max_\pi v_\pi(s), \tag{13}$$

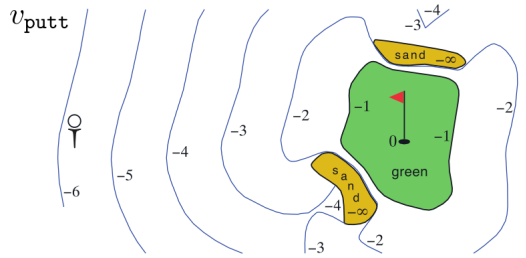
for all $s \in S$; as well as the same optimal action-value function q_* :

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \tag{14}$$

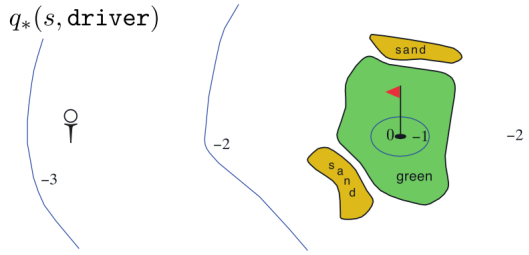
for all $s \in S$ and $a \in A(S)$. Put into words, this latter function gives an expected return after acting a in state s, and then following an optimal policy for a state-action pair (s,a). From this, the function can be rewritten with respect to v_*:

$$q_*(s,a) = \mathbb{E}[\ R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]\ . \tag{15}$$

Let's consider an example of golf [6].



(a)



(b)

Figure 5: The state-value function for using a putter (a) and the optimal action value function for using a driver (b).

Before learning about optimal value functions, to develop a simulation of golf as a reinforcement learning task, we start by giving a -1 reward for each hit that does not lead to the ball going into the hole. Using the backup diagram as reference, the state at the top of the diagram would be the emplacement of the ball. The state is followed by actions, which are the selection of the club and the way the ball is hit. Let's examine the situation by considering the first action mentioned, assuming that the club could either be a putter or a driver. Figure 5(a) shows a state-value function v_putt(s) for a policy that is constantly using a putter. If we the value of the state is given by the negative number of hit needed to reach the hole from its current emplacement, then the closer it is to the hole, the closer the number is from zero. Therefore, if the ball is found in the -6 zone, it signifies that it would take 6 hits to get from its current location to the hole. If the ball reaches a sandbank, it returns a value of $-\infty$.

After going through optimal value functions, the same problem can be tackled more cleverly to achieve a higher reward after reaching the terminal state. In others words, the agent will now include in its consideration an optimal action-value function $q_*(s, driver)$ as figure 5(b) shows; the state values and their contours are different from figure 5(a) as the agent will now start off with the driver and will then act later on as it sees fit, meaning by using the driver or

the putter. Both of these actions have their own advantages and inconveniences. For instance, by using the driver, one will have less accuracy but can hit the golf ball further than when one uses the putter. Therefore, at certain states, the agent will favor using the driver more than the putter, that is when it is far from the hole. For example, if the agent is at the tee and decides to putt instead of driving for its first stroke, at its next state, it will require minimum three strokes to arrive at the terminal state while it will require minimum two strokes to reach its goal if he uses the driver at first. In addition, the golf ball will barely approach the hole if it chooses to act this way. To sum up, the optimal policy for the agent is to use the driver the first two tries, and then finish off his game with the putter, all in all with three strokes.

Speaking of optimal state-value function, the Bellman equation for v_π (equation 12) can also be rewritten for v_*, an optimal value function, as the latter, by definition, $\in v_\pi$, and more specifically it is the value function of an optimal policy. Moreover, it can be rewritten in two forms that do not need to inquire any specific policy as this concerns an optimal value function[6]:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[\ R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)[\ r + \gamma v_*(s')]\ . \end{aligned} \tag{16}$$

This is called the *Bellman optimality equation* which suggests that the value of a state under an optimal policy has to be equal to the expected return that results from the best act of that state.

The Bellman optimality equation exists also for q_* :

$$\begin{aligned} q_*(s, a) &= \mathbb{E}[\ R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[\ r + \gamma \max_{a'} q_*(s', a')]\ . \end{aligned} \tag{17}$$

Just like how the Bellman equations for v_π and q_π can be expressed with backup diagrams, the same goes for the Bellman equations for v_* (Equation 16) and q_* (Equation 17):

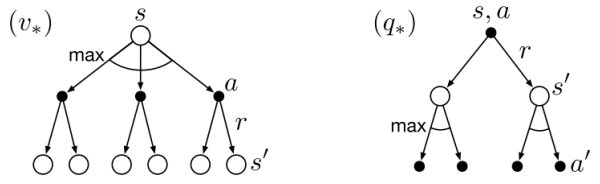


Figure 6: Backup diagrams for v_* and q_* [6]

However, as it can be observed in the optimal cases, there are presence of arcs whenever the agent has to choose an action starting from state s. In fact, these arcs represent that the agent will choose the actions, over the action of non-optimal policy, that will bring the maximum state value, the maximum reward, at its successor states. Having said that, if solely the maximum actions can occur, that the other non-maximum actions have no probability of occurring during a policy, then this policy π is an optimal policy π_*.

To find the optimal policy, one can use the Bellman optimality equation for v_* (Equation 16) as it possesses a unique solution independent of the policy. The solution to this equation is found through a system of equations composed of n, and thus of n unknowns, and with the help of the dynamics p of the environment, and also the techniques used to solve systems of nonlinear equations. The same idea can be used to solve for the Bellman optimality equation for q_*.

In summation, if the agent finds the optimal value function, v_*, only maximum actions will compose its path, and it will undoubtedly follow an optimal policy, since v_*. Thus, even if one doesn't look too far into the future of the consequences of its actions, it would not succumb to any negative impact as v_* already planned its future actions and rewards. That is, a one-step-ahead search approach, in reinforcement learning, in this case, will result in the long-term optimal actions.

In addition, if q_* was found, the agent will not even need to go through a one-step-ahead search as it needs to just play out the action that maximizes $q_*(s, a)$ in all states. Basically, it can be worthwhile to go through the trouble to find $q_*(s, a)$ as the agent can then ignore its environment's dynamics, meaning the possible successor states and their values, as it just chooses the optimal actions placed in front of its view.

Easier said than done, seeking the Bellman optimality equation's solution, which will certainly lead the agent to find the optimal policy easily, might be in fact a pointless thing do to. It requires one to look at the numerous future possibilities, to compute the probabilities for the latter to occur,and as well as to determine how rewarding they are. On top of this, this solution can only work by making three assumptions that barely holds any truth in practice[6]:

1. information about the dynamics of the environment is accurate;
2. we have the necessary amount of data to compute the solution for reinforcement learning;
3. the Markov Decision Processes.

If even one assumption is violated, solving the Bellman equation for v_*, and even more so for q_*, with our current technology can take thousands of years. That is the case for the game of backgammon that has about 10^{20} states, but

a lack of computational resources prevents the finalization of a solution for the optimal policy. Therefore, in reinforcement learning, approximation of solutions cannot usually be avoided, and there exists many decision-making methods to solve for the Bellman optimality equation whilst using approximation such as the heuristic search methods and the methods of dynamic programming. In this event, actual experienced transitions are used instead of expected transitions to solve for the equation.

2.3 Agent's constraints and approximation

In practice, optimal policies only rarely happens as it requires extreme computational costs. Although the ideal is to have an agent able to learn an optimal policy, the closest we can get is an approximation. As mentioned above, it is generally not possible to compute an optimal policy by solving the Bellman optimality equation. The major problem that the agent faces is the computational power available, specifically the number of computations it can do in a single time step. For example, in chess, it requires a bit of human experience, and even a custom-designed computers would not be able to compute the optimal moves. In addition, having a large amount of memory is necessary to compute approximation of value functions, policies, and models. When tasks have small state sets, we can form the approximations with the use of tables and arrays with one entry for each state that we named tabular case. If there are too many states compared to the possible entries in the table, the cases are approximated. In reinforcement learning, approximations is the best option. A key property to solving MDP's that differentiates reinforcement learning from other approaches is its ability to approximate optimal policies in ways that put more effort into learning to choose the better option for states that occur more times, and spend less effort for infrequently encountered states.

3 Dynamic Programming

Please explain the distinction between model-based (Dynamic Programming) and model-free (Reinforcement Learning) algorithms.

3.1 MDP

Markov decisions process is usually defined with the following variables S is the state space of an environment A is the set of actions the agent can choose between R(s,a) is a function that returns the reward received for taking action an in a state S T a transition state

MDP is a process that attempt to predict the outcome of a model with the information about the current state, the probability distribution of every possible action (state transition probability function) and the reward function.

Value function pi(a—s) This function determines the value of the state the agent is currently in.

3.2 Model-based programming

Model-based programming usually infers that the agent bases his learning process upon a prediction based on an already established model. A model-based algorithm has an agent trying to find the optimal policy in order to maximize the reward equations of the. In RL, we have a state called S and an action called A. Just like mentioned in chapter 1, at time t, we are in a state S_t which is in S. If we decide to execute an Action At which is in A, we are now in a transition state $S_{t+1} = f(s_t, a_t)$. This follows the dynamical model

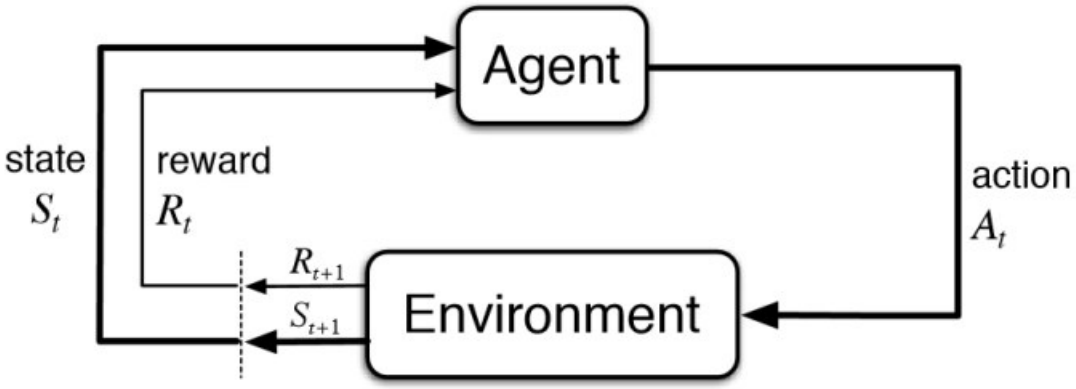


Figure 7: Caption

Model-based RL tries to overcome the issue by enabling the agent to construct an approximation of its environment. Model-based emphasizes planning. Each different programming methods have there different pros and cons. Model-based programming requires way fewer samples to be effecting. This is because model-based programming focuses on planing rather than reacting to its environment. Model-based algorithms can also be used to simulate situations without needing an actual environment. Model base RL can also be more flexible than its counterpart. If you can predict the dynamics of one specific environment, you can usually generalize this model with a small few tweaks to accommodates a variety of other similar environments

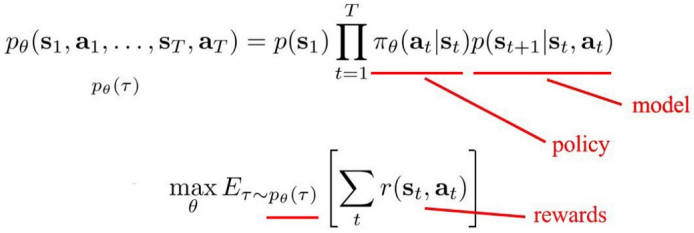


Figure 8: Caption

Reinforcement training consistent of maximizing the rewards which also dictates the most optimal policy. Model-Based reinforcement learning focuses only on which actions to take depending on its current states. One of the main assumptions of model-based learning is that all assumptions about the problem are regrouped into the form of a model. The learn how the environment works. The optimizations of reinforcement learning consists of trying to achieve the highest while assuring that the transitions states are viable.

The main con of a model-based algorithm is that it requires a lot of assumptions about the environment in order for it to be effective. These assumptions allow us to diminish the number of trials necessary but always limits the variety of situation which can be used in

A simple way to see if a Reinforcement Algorithm is the model base is to see if after learning, the agent is able to make predictions about the nest state/reward. If it can, it is model base, otherwise, it is model-free.

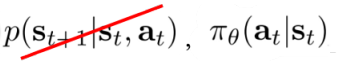


Figure 9: Caption

3.3 Model-free programming

Model-free reinforcement learning is quite different from it's model-based counterpart. Model-free reinforcement learning is mostly used when we cannot control the environment or it is too complex. As the name implies, we completely ignore any previously used models. It is only dependent on the agents' simulation.

The idea behind model-free reinforcement learning is to find a method for the agent to learn without learning the transitions probabilities p(s'—s,a). The probability of This transition predicts the transitions state which is also a way to model the environment

The goal is to learn value for each state for the current policies Pi we are using. This is done b estimating a 'value function' also called policy, by bypassing the transitions function and the reward function. This because the agent

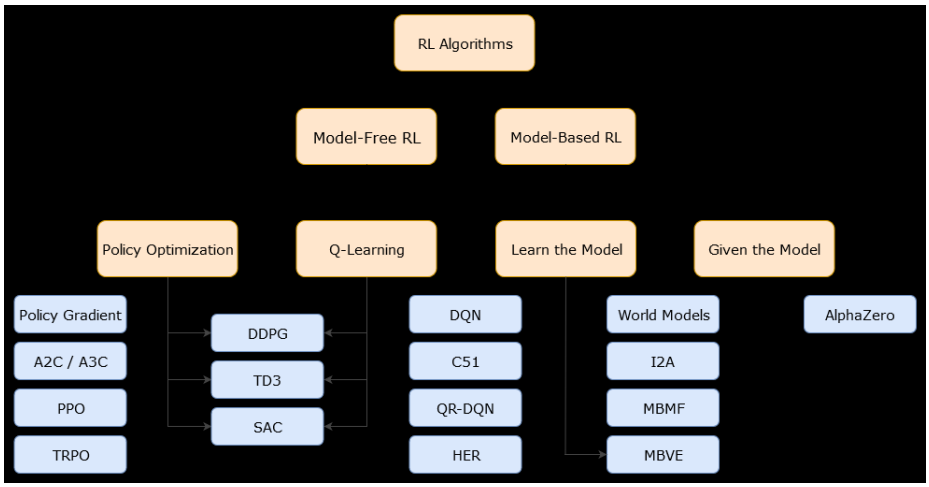


Figure 10: Caption

does not have access to the environment thus not being able to predict the state transition and rewards.

3.4 Policy Optimization

[H] Policy optimization is a technique that allows you to implement a cost function and run the gradient descent technic on it. Gradient descent is an optimization method designed to reach the minimum of a function. This is especially useful in deep learning since it allows us to maximize the cost function of a specific algorithm. Gradient descend is mostly done by using derivatives (calculus). By finding the minimum value of this function, we can find where the cost vs lost function is the most efficient. The learning rate is the size we use to reach the minimum value of the function. The larger the learning rate, the more ground we can cover in order to find the minimum. But by having a large learning rate, we increase the risk and chance of completely overshooting the targeted minima. On the other hand, having small learning steps will take much more time in order to reach the lowest point. PPO (proximal policy optimization achieves a balance by not setting a hard constraint.

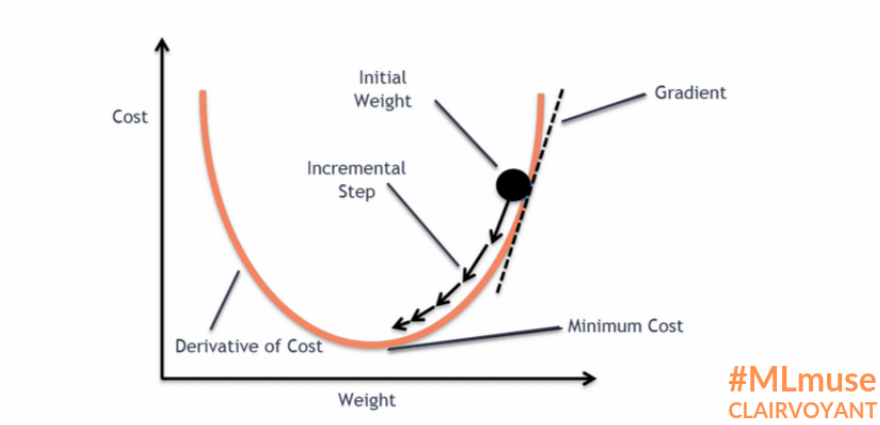


Figure 11: Caption

3.5 Q learning

Q learning is a model-free algorithm. It is based on an equation called the Bellman equation. Typically, other types of reinforcement learning are policy-based. Q learning is often referred to as an off-policy learner because it learns the value of the best possible policy without needing knowledge of the agent's actions. This type of learning seeks to learn the policy that will maximize the total reward. Q -learning also uses temporal differences to estimate the value. TD is an agent learning from and the environment through different steps without knowing the environment. Q learning works by creating a table with the model [state, action]. Then the agent interacts with the environment in 2 possible ways. Firstly, it can choose to use the previous Q value tables as past references to determine it's future actions (called exploiting) or it can act completely randomly (which is called exploring). Achieve a good balance between both methods allows the agent to discover new states while still being efficient.

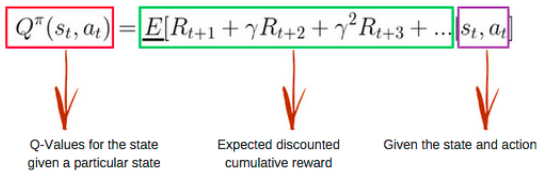


Figure 12: Caption

Then after each steps/actions, the Q table is updated. The agent will then proceed to find the best policie using this method.

3.6 Policy iteration vs Value Iteration

iteration

The value iteration algorithm manipulates the value function directly. A value-function represent how profitable it is for an agent to be in a particular state We use the Bellman equation in order for it to converge to the optimal value. A value iteration algorithm usually finds it's optimal value by using a Markov decision Process. Value iteration calculates the optimal state value by iterating through estimates of V(s). The algorithm initialises the function to random values. It then iterates and update the V(s) until it finds an optimal solution which converges to a value

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

Figure 13: Caption

Policy iteration is a dynamic programming method that calculates the expected return for every state/action. With these, we are able to calculate the reward/ lost function which can then be improved on. Instead of changing the value function directly, Policy iteration will re-define the Policy at each and every step and compute it's value until it converges to an optimal policy.

4 Some RL algorithms

4.1 Temporal-Difference Learning

The Monte Carlo and the Dynamic Programing algorithms can be combined together to produce a better and more realistic reinforcement learning program. This breakthrough method is referred to as Temporal-Difference (TD) Learning and it is a recurrent concept in the field of RL. Temporal difference allows the agent to learn from raw experience gained through countless iterations and updates of a particular task without the need for a model of the environment as the Monte Carlo idea [6]. Additionally, like Dynamic programming, TD algorithms enable a faster learning process by updating the estimations of state values based on other future estimates.

4.1.1 TD Prediction

The challenge of a machine learning process is to correctly choose an action, α, that corresponds to a particular state, s at time t. Monte Carlo and TD algorithms are capable of solving the prediction problem through experience that is gained through numerous iterations while following a policy, π. This

experience is archived into estimates v of v_π that are constantly updated upon visiting nonterminal states s_t [6]. In the Monte Carlo method, the update is made once the return is obtained at the end of the episode which will then be used as a target in the following state to have a better understanding of it. The simplest update rule for a Monte Carlo method is:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)], \tag{18}$$

where G_t is the expected return after time t and the step-size parameter α. This method called constant-α MC updates $V(S_t)$ by sampling an episode and receiving G_t and then calculating the error between the actual return and the expected one. Instead of waiting until the episode ends in order to update $V(S_t)$, Temporal-Difference method only requires the next time increment. Indeed, the updated estimate of this algorithm is built on the observed reward of the next state, R_{t+1} and the estimate of $V(S_{t+1})$ [6]. The most straightforward computation of TD method, also known as TD(0) is :

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1} - V(S_t)], \tag{19}$$

for which the target of the update is $R_{t+1} + \gamma V(S_{t+1} - V(S_t)$. Since the TD method uses value estimates of successor states in order to update its current states, it is referred to as a bootstrapping method such as Dynamic Programming.

The difference between Monte Carlo and Temporal-Difference algorithms is observable in the driving home example proposed by Richard S. Sutton and Andrew G. Barto in their introductory book on reinforcement learning. Suppose Mike is trying to predict the time of his arrival back to his home. On a Wednesday, he leaves his office at 6 o'clock, reaches his car at 6:05 and he estimates that it will take him 30 minutes to reach his home. As he starts his engine, he notices that it is starting to rain and increases the time it will take him to get home from 30 to 35 minutes since traffic is often slower in poor weather conditions. As of now, his commute back home will take 40 minutes. Mike finishes the highway portion of his journey in record time despite the rain and recalculates his time to a total of 35 minutes. Unfortunately, upon exiting the highway onto a secondary road, he gets stuck behind a slow moving truck. Unable to pass him, Mike is obliged to follow the truck until he can turn on his street at 6:40. Three minutes later, Mike is home. If the Monte Carlo method is applied to this example, the estimations for each state will be updated toward the true return of the episode, thus 43 minutes. These changes can be illustrated by plotting the predicted time of travel over the timeline of events as shown in Figure 1.

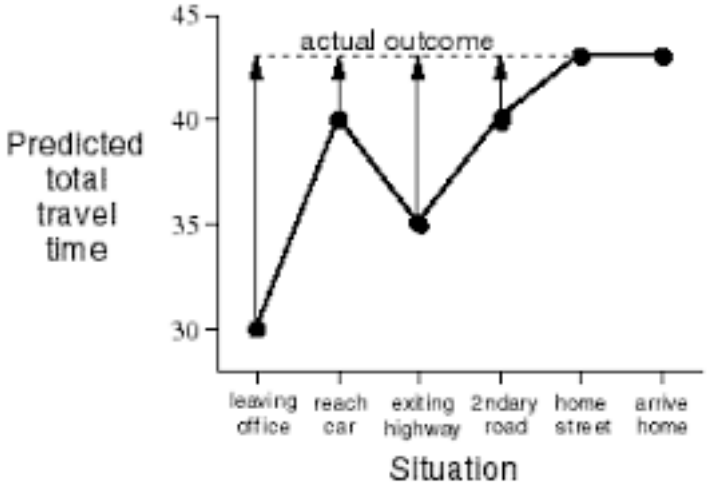


Figure 14: Updates done by the Monte Carlo method

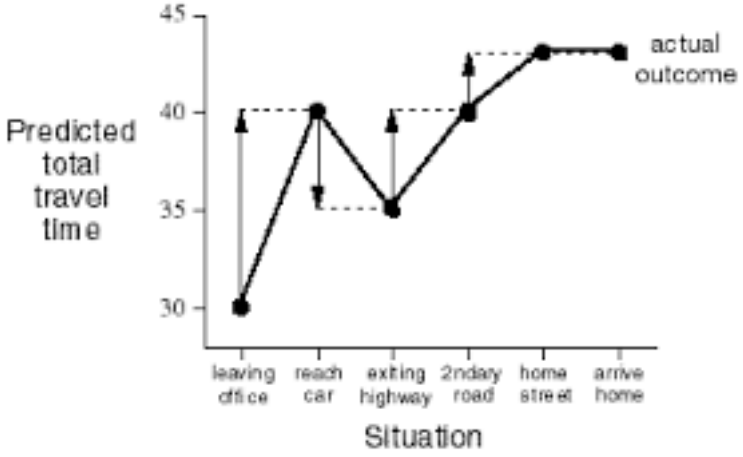


Figure 15: Updates done by the Temporal-Difference method

However, this example raises an important question regarding the way that learning occurs: is it mandatory to wait until the end of the episode for the agent to begin learning? Suppose Mike was stuck in heavy traffic on the highway. Twenty-five minutes pass and the traffic has yet to subside. As a result, Mike reevaluates his time from 30 minutes to 50 minutes. The important question here is whether he should wait to get home and then update his estimations or do it during the traffic. According to the Monte Carlo idea, he must wait until he gets home since the actual return is still unknown at that time.

On the other hand, by following the TD algorithm, each state would be updated based on an estimation of the reward of the next state, R_{t+1}. This will enable the agent to change its original estimation of 30 minutes to a new one of 50 minutes without waiting for the episode to end. By doing so, each estimate will be a reflection of the rewards earned in the successive state as shown in Figure 2.

Figure 14 and 15 clearly illustrates the difference between the Monte Carlo and Temporal-difference learning methods. Monte Carlo is unable to detect the nuances between intermediary states.

4.1.2 Advantages of TD Prediction Method

Since Temporal-Difference is a combination of Monte Carlo and DP methods, it is logical to assume that it is far better at the prediction problems compared to the others. Indeed, there are many factors that makes TD more advantageous to use and the first one is the model of the environment. Due to its Monte Carlo nature, TD algorithms are capable of learning without the need of a model like DP does. Furthermore, since the update rule of TD does not contain the discount factor, γ, it converges to the optimal policy much faster compared to the Monte Carlo method [6]. TD methods are especially useful for long episodic tasks since they do not need to wait until the true return is known to update their estimations. All they need is the next time step. This aspect of the TD algorithm begs the following question: are those estimations accurate? Is it good that they learn from mere speculations? As it turns out, TD methods are more accurate and are less prone to making errors compared to the Monte Carlo method. This surprising fact is observable in the Random walk example shown below in Figure 3 . In this context, all episode start at the center state, C, and the agent can either proceed left or right by one time step with equal probability. The terminal states are located on both extremes with the one on the right giving a return of +1. Countless iterations are made with both TD and Monte Carlo algorithms and their average root mean square error is plotted over the iterations in Figure 4.

Figure 6.5: A small Markov process for generating random walks.

Figure 16: Random walk Markov Decision Process

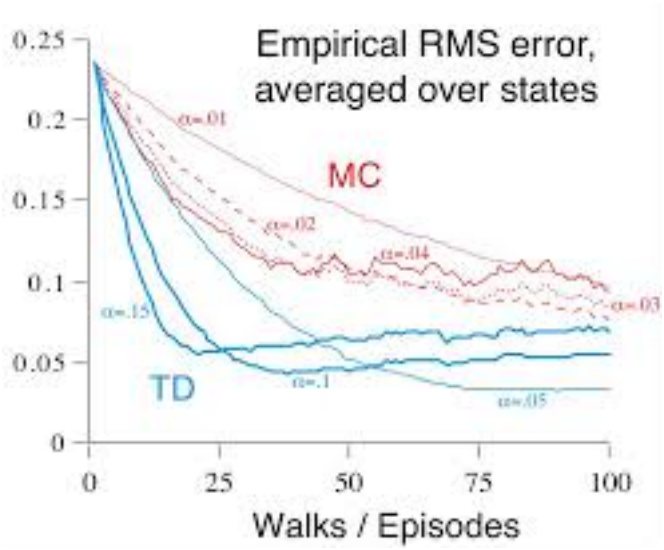


Figure 17: Root Mean Square Error of TD and MC with a small, varrying constant-α. These values represent the learning curves of both algorithms on the random walk MDP. These values are averaged over 100 episodes

As shown in Figure 4, the TD method is generally better compared to the Monte Carlo method despite updating its estimations on future estimations [6].

4.1.3 Optimality of TD(0)

Another way to prove that TD methods are superior to Monte Carlo is by using the batch updating method. This method consists of storing each observed action taken in each state and replaying that specific trajectory in a similar fashion as the policy evaluation of dynamic programming. Through this replay, the value estimates will converge to a value. Under batch updating, TD and Monte Carlo have been shown to converge deterministically to two different answers[6]. It is important to note that under normal circumstances, both methods do not reach their answers, but they tend to follow the same trajectory. Performing the random walk Markov Decision Process with batch updating with the two algorithms yielded the following graph:

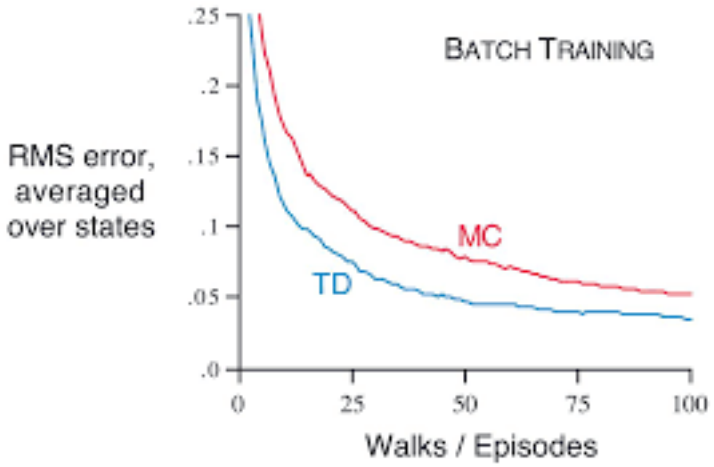


Figure 18: Root Mean Square Error of TD and MC with a small, varying constant-α. These values represent the learning curves of both algorithms on the random walk MDP under batch updating

Under batch training, Monte Carlo converges to averages of the actual return given at each state which are optimal estimates[6]. In spite of that, according to Figure 18, the TD method outclasses the Monte Carlo with a consistently lower average RMS error. This difference can be explained by the fact that TD methods are optimal in a way that is more relevant in the prediction problem.

The use of the prediction example will be needed to further develop on the nature of the optimality of the Temporal-Difference Learning algorithm. Consider the **You are the Predictor** scenario presented by Richard S. Sutton and Andrew G. Barto in their introductory book on reinforcement learning:

A, 0, B, 0	B, 1
B, 1	B, 1
B, 1	B, 1
B, 1	B, 0

There are eight episodes illustrated with the first one starting at state A and terminating at state B with a reward of 0. Six out of the eight episodes start at state B and terminates with a reward of 1. The last episode starts at state B and terminates with a reward of 0. Given this batch of data, what are the optimal predictions for V(A) and V(B)? Based on the Monte Carlo method, V(A) would be equal to 0 since it transitioned from state A to B and terminated with

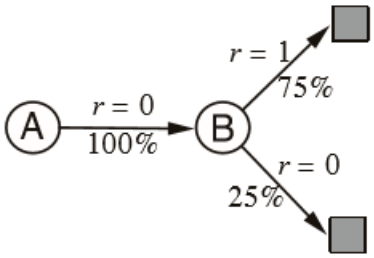


Figure 19: Representation of the Predictor MDP

a reward of 0. The other correct answer would be $\frac{3}{4}$ since the process always transitioned to state B whenever it started at state A and V(B) is equal to $\frac{3}{4}$. Plotting these results into a Markov process will give Figure 9 :

These two answers reflect a fundamental difference between the Monte Carlo and Temporal-Difference methods. The estimates derived from Monte Carlo minimizes the root mean square error on the training set while TD(0) searches for estimates that fit perfectly for the maximum-likelihood model of the MDP process[6]. The maximum-likelihood estimate is a parameter value whose probability of generating data is greatest[6]. Given this characteristic, it is possible to compute the certainty-equivalence estimate which represents the estimation of the value function that would be exact if and only if the model is exactly correct. In most cases, batch TD(0) has been proven to converge to the certainty-equivalence estimate.

These two elements help in supporting why TD(0) is often better than Monte Carlo. Indeed, by converging to the certainty-equivalence estimate, TD(0) algorithms take less time in finding the optimal value function, and thus the optimal policy.

4.2 SARSA: On-Policy TD Control

Before discussing the SARSA algorithm, it is important to define the two fundamental tasks of a reinforcement learning agent. The first task concerns the prediction problem which can be described as being able to correctly predict the consequence of an action taken in state s at time t. The predictions are usually estimations of a state value or an action value. In this section, the control is introduced. A control requires the agent to make the correct decision in a given state at a particular time. The correct decision is often the one that will yield the greatest return as determined by the TD prediction method.

The first step in computing SARSA is to learn the action-state value function

instead ot the state-value function. In other words, we must estimate q_π(s,a) for every state s and action a while under the policy π [6]. In practice, the action-state value function is more interesting since we are dealing with a model-free RL. The transition from one state-action pair to another given by the quintuplet $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ gives rise to the name SARSA and its estimation of q_π(s,a) is given by the following algorithm:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (20)$$

The update is made after each transition from a nonterminal state. If S_{t+1} is terminal, it is assigned a value of 0[6]. The algorithm updates its estimation by bootstraping a random action in the successive state and calculating the error associated while under a certain policy π. After many iterations of the SARSA method, the policy, π, will converge to greediness. According to Satinder Singh, the convergence of SARSA to an optimal policy and action-value function is assured with probability 1 as long as every state-action pairs are visited an infinite amount of times[6]. The steps involved in the SARSA method is given in the following figure:

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 20: Walkthrough of SARSA: An on-policy TD control

4.3 Q-Learning: Off-policy TD Control

The field of reinforcement learning expanded tremendously after the development of and off-policy TD control method also known as Q-Learning. This algorithm was developed by Chris Watkins in 1989 [6] and its simplest form, the one-step Q-Learning is defined as follow:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (21)$$

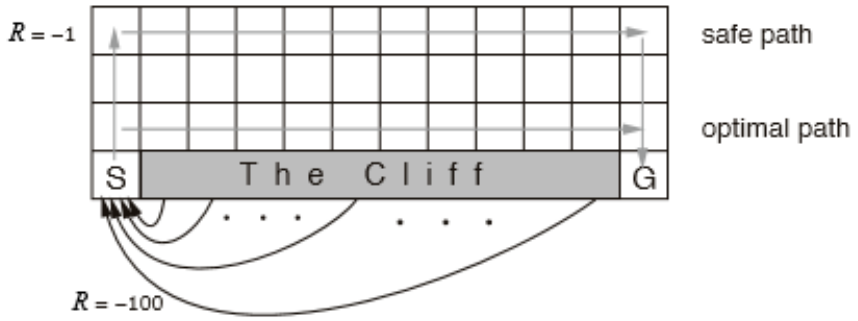
Q-Learning is described as an off-policy algorithm because it can derive the optimal policy with Q, which approximates q_*, independently of the policy being

followed as long as the agent explores enough[6]. The update of the off-policy method is similar to SARSA in which the bootstrapping process is made by sampling the successor state-action pair, but instead of choosing the one that corresponds to π, it will greedily choose the state-action with the highest Q function. The steps involved in the Q-Learning method is given by the following figure:

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Figure 21: Walkthrough of Q-Learning: An off-policy TD control

The cliff walking example is an excellent demonstration of the differences between Q-Learning, an off-policy method, and SARSA, an on-policy method. In this scenario, the agent starts in position S and must terminate in position G. There is a cliff at the bottom which result in a reward of -100 if the agent falls into that region. If that happens, the agent will be forced to start at beginning. The perfomance of the on-policy and off-policy algorithms are illustrated below:



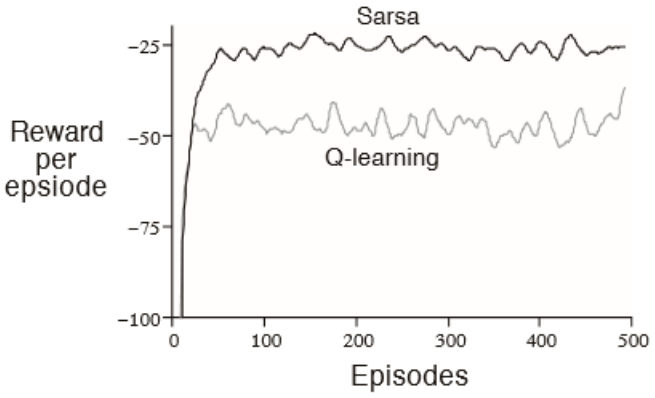


Figure 22: Performance of the on-policy and off-policy methods

The example is run with ϵ-greedy action selection of 0.1 for both SARSA and Q-Learning. The path of Q-Learning, shown above, is the optimal one since it learns from the optimal action. In contrast, the SARSA algorithm chooses the safest path because its policy knows that it will inevitably choose a down movement that will result in a reward of -100. As a result, SARSA outperforms Q-Learning since it has less chances from falling into the cliff.

4.4 Eligibility Traces

One of the most basic and powerful mechanism in reinforcement learning are eligibility traces. Indeed, their introduction into TD algorithms, such as SARSA and Q-Learning, gave birth to a more general family of learning methods that are far more efficient.

Eligibility traces can be define in two ways. In the theorethical point of view also referred to as the forward view, eligibility traces are used to form a bridge connecting TD and Monte Carlo methods together[6]. The integration of eligibility traces into TD algorithms creates a whole spectrum of reinforcement learning algorithms that are intermediates between TD and Monte Carlo. These intermediate methods are generally better than the extreme methods because they can be applied to more modern problems.

The other way of understanding eligibility traces is according to the mechanistic viewpoint or the backward view. Based on that, eligibility traces can be understood as a temporary record of the occurrence of an event. These events can be either action taken in a state s_t or the visit of a particular state. As a result, eligibility traces can be used to understand how the agent learns under π. As such, these mechanisms help connect events to training information[6].

4.4.1 η-Step Prediction

As discussed earlier, eligibility traces are used in order to create a spectrum of intermediate algorithms between TD and Monte Carlo. Instead of performing updates based on the following state like TD or updating estimates for each state after an episode, intermediate algorithms would perform these backups based on an intermediate number of rewards[6].For example, a two-step backup would be based on the first two rewards and the value of the state two steps later. Likewise, we could compute a three-step backup, four-step backup and so on until the Monte Carlo method is reached[6]. The general name given method that implement TD ideas over η-step are referred to as η-step TD methods and the one given to the TD methods discussed in the previous section is one-step TD methods.

As shown in Figure **??**, the backup in a Monte Carlo method are always updated towards the actual reward gained at the end of the episode. This update can be defined as follow:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + +... + \gamma^{T-1-t} R_T, \quad (22)$$

where T is the target. In the case of the one-step TD method, the updates are capable of detecting the nuances from the transition of one state to the following, and thus the target will be an estimation that reflects that of the following state. As such, their target is defined as:

$$R_{t+1} + \gamma V_t(S_{t+1}) \quad (23)$$

The same logic can be applied to all intermediate algorithms. Indeed, a two-step backup will have its target defined as:

$$R_{t+1} + \gamma R_{t+2} + \gamma^2 V_t(S_{t+2}) \quad (24)$$

Similarly the target for an η-step TD method is:

$$G_t^{t+n}(c) = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_h + \gamma^n c, \quad (25)$$

for n $\geq$ 1 and for all scalar correction c $\in \mathbb{R}$. The time $\boldsymbol{h = t + 1}$ is called the horizon. The horizon is a parameter that tells the agent to maximize it reward in x number of steps. For example, a two-step horizon tells the agent to choose actions that will maximize its reward in the following two steps. If the episode ends before the horizon has been reached, then the truncation of the η-return return occurs at the end of the episode, resulting in a complete return[6]. Simply put, if h $\geq$ T, then $G_t^h = G_t$. According to this definition, the Monte Carlo method can be treated as an exception to the infinite-step targets.

By definition, an η-step backup is a backup toward the η-step return. As shown below, the backup takes the form of an incremental $\triangle_t(S_t)$ to allow different ways of making the update[6]:

$$\triangle_t(S_t) = \alpha[G_t^{t+n}(V_t(S_{t+n}) - V_t(S_t)] \quad (26)$$

In an on-line update, the updates are made during the episode, as soon as the increment is computed and can be written:

$$V_{t+1}(s) = V_t(s) + \triangle_t(s), \quad \forall s \in S \quad (27)$$

In the case of off-line updating, the computed increments are accumulated on the side and are integrated once the episode ends in a way that the new value of the following episode is the sum of all increments computed during the episode[6].

$$V_{t+1}(s) = V_t(s), \quad \forall t \in T \quad (28)$$

$$V_T(s) = V_{T-1}(s) + \Sigma_{t=0}^{T-1} \triangle_T(s) \quad (29)$$

The η-step return has an important property that assures the convergence of the η-step backups to the correct predictions under appropriate technical conditions. That property is called the ***Error Reduction Property of η-step returns***. According to it, the expected value of the η-step return using υ is guaranteed to be a better estimate of υ_π than υ is, in the worst case scenario. The error reduction property can be demonstrated as follow[6]:

$$\max_s |E[G_t^{t+n}(\upsilon(S_{t+1}))|S_t = s] - \upsilon_\pi(s)| \leq \gamma^n \max_s |\upsilon(s) - \upsilon_\pi(s)|, \quad \forall n \geq 1 \quad (30)$$

Nonetheless, η-step TD methods are not used a lot since they are hard to implement. Using η-step returns requires waiting η-step before obtaining the return. This process can become very long for large η and will hinder the control applications. η-step returns are only used in a purely theoretical manner to demonstrate the convenience of intermediary algorithms.

4.4.2 The Forward View of TD(λ

A complex backup is defined as a backup that averages simpler component backups[6]. With the introduction of eligibility traces, backups can be done not just after η-step, but also toward any average of η-step returns[6]. For example, $\frac{1}{2}G_t^{t+2}(V_t(S_{t+2})) + \frac{1}{2}G_t^{t+4}(V_t(t+4))$ is a backup made toward a target that is half of a two-step return and half of a four-step return. Similarly to the η-step TD methods, a complex backup posseses the error reduction property which will assure its convergence to a correct answer. By averaging any η-step returns, we substantially expand the spectrum bridging one-step TD and Monte Carlo methods which can help improve the learning process of agents in certain scenarios.

The complex backup can be computed as follow:

$$L_t = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{t+n}(V_t(S_{t+n})), \quad (31)$$

where L_t represents the λ-return. Each of the η-step backup are proportionally weighted to λ^{n-1} for $\lambda \in [0,1]$ and normalized by a factor of (1-λ) to ensure a

sum of 1 from all of the weights. The following figure illustrates the weights attributed to each η-step backup:

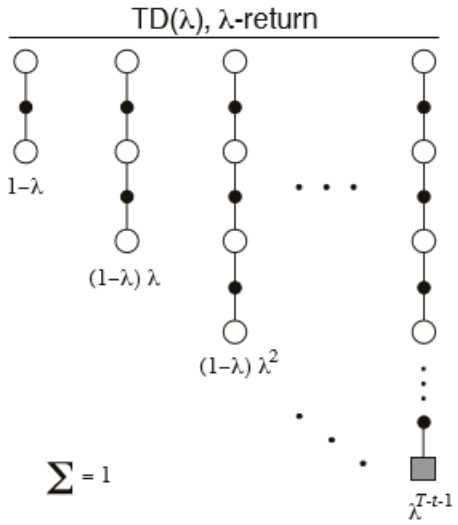


Figure 23: Weights given to each TD(λ) backup. Each subsequent step from the first one fades by a factor of λ

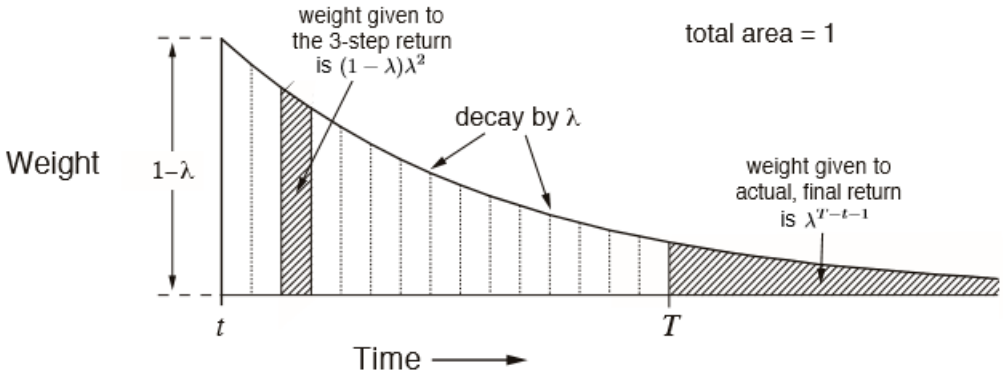


Figure 24: Weighting attributed in the λ-return for each η-step returns. As t approaches T, the weights associated to each η-step become less important.

By separating the post-termination terms from the sum of the complex

backup equation (31), we get:

$$L_t = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} G_t^{t+n}(V_t(S_{t+n})) + \lambda^{T-t-1} G_t \quad (32)$$

Equation 31 clarifies what happen when λ=1. If such is the case, the main sum goes to 0 and the λ-return is the same as the Monte Carlo return, G_t. As opposed to that, when λ=0, the λ-return simply becomes $G_t^{t+n}(V_t(S_{t+n}))$, the one-step TD method discussed earlier.

The λ-return algorithm estimates backups that are targeted toward the λ-return by computing an increment, $\triangle_t(S_t)$ associated to the value of the state on that step. This process is the same as the one discussed in the η-step prediction section. Given this, the following equation arises:

$$\triangle_t(S_t) = \alpha[L_t - V_t(S_t)] \quad (33)$$

The forward view of TD(λ) algorithms searches for the best combination of future states that will yield the greater reward, and thus lead to optimality. Figure 25 summarizes the concept of the theoretical view of eligibility traces.

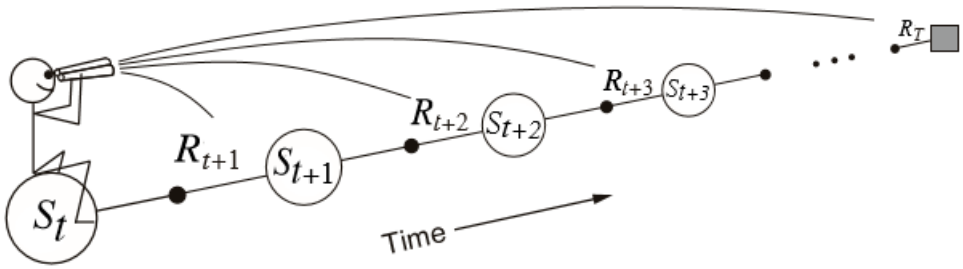


Figure 25: The Forward View of Eligibility Traces. The agent looks into subsequent states and chooses the best combination.

4.4.3 The Backward View of TD(λ

In the previous section, the theoretical or forward view of eligibility traces was presented as a way of relating one-step TD to Monte Carlo methods by creating intermediary algorithms. In this section, we present the mechanistic or backward view of eligibility traces. Contrary to the forward view, the mechanistic view is causal and simple both conceptually and computationally. As a result, it is easily implementable. The backward view provides an incremental mechanism for approximating the forward view[6].

The backward view introduces a new parameter that serves as a memory of actions taken or states visited known as an ***eligibility trace***. The eligibility trace for a certain state s at a time t is denoted $E_t(S_t)$ and is expressed as follows:

$$E_t(s) = \gamma\lambda E_{t-1}(s), \quad \forall s \in S, s \neq S_t \quad (34)$$

Equation 34 represents the trace decay for all states. What about the trace associated with the state S_t? The classical expression for the trace decay of a particular state at time t is given by the following equation:

$$E_t(S_t) = \gamma\lambda E_{t-1}(S_t) + 1 \quad (35)$$

= As shown in equation 35, the eligibility trace of non-visited states decay by a factor of $\gamma\lambda$, where γ is the discount factor and λ, the trace-decay parameter. Eligibility traces are memory archives that record the visiting of a state or the taking of an action that recently occurred, where the notion of "recently" is defined by the variable λ. For large λ but less than 1, the agent will keep remember states or actions that are further in the past and will use them to undergo learning changes. There exist three types of eligibility traces. Accumulating traces add up each time a state is visited, replacing traces are reset to one at every states and dutch traces are an intermediary between accumulating and replacing traces[6]. Indeed, dutch traces add up every time a state is visited by a factor of α. The three types of eligibility traces are illustrate in Figure 26

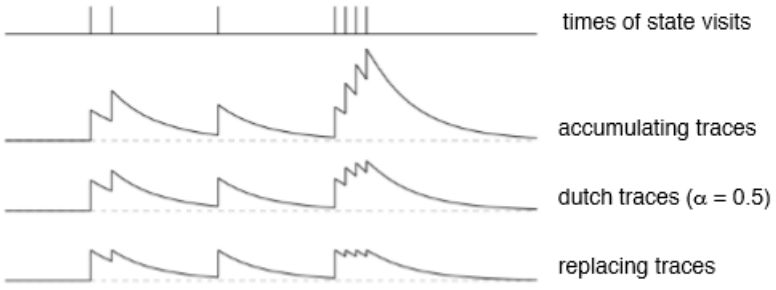


Figure 26: The three types of eligibility traces

The process of learning in an individual consists of identifying a mistake that led him to where he is and avoid repeating said mistake again. This concept is applied to reinforcement learning as well. Indeed, by using the one-step TD error, δ_t and applying it to the incremental change, $\triangle V_t(s)$, we get the following:

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t) \quad (36)$$

$$\triangle V_t(s) = \alpha\delta E_t(s), \quad \forall s \in S \quad (37).$$

Given this, if $\lambda = 1$, the decay at each step would only be by a factor of γ which is the same as the Monte Carlo behavior. On the other hand, if $\lambda = 0$, then all traces are equal to 0 except for the one associated to S_t, which results in the simple TD(0) algorithm that was discussed in Section 4.1.3.

In short,the mechanistic view of eligibility traces is oriented toward the past. At each state, the agent will look back to it learning process and assign the error to the corresponding states with respect to its eligibility trace at that time. This operation gives rise to Figure 27.

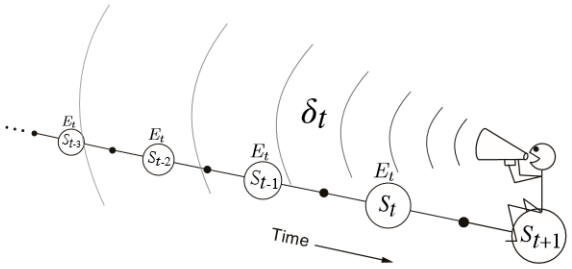


Figure 27: The mechanistic view on eligibility traces. Each update is influenced by past actions or states with respect to their eligibility traces.

4.4.4 SARSA(λ)

The application of eligibility traces into the SARSA algorithm requires giving eligibility traces not to state, but to state-action pairs. The traces attributed to state-action pairs is denoted by $E_t(s, a)$ and it can be either of the three traces presented earlier. Of course, the updates are triggered by the visiting of a state-action pair given by the identity-indicator notation, $I_{sS_t} I_{aA_t}$. The eligibility traces for the state-action pairs are given below.

$$\begin{aligned}
E_t(s,a) &= \gamma\lambda E_{t-1}(s,a) + I_{sS_t} I_{aA_t} && \text{(accumulating)} \\
E_t(s,a) &= (1-\alpha)\gamma\lambda E_{t-1}(s,a) + I_{sS_t} I_{aA_t} && \text{(dutch)} \\
E_t(s,a) &= (1 - I_{sS_t} I_{aA_t})\gamma\lambda E_{t-1}(s,a) + I_{sS_t} I_{aA_t} && \text{(replacing)}
\end{aligned}$$

for all s ∈ S and a ∈ A. By implementing these eligibility traces into the Q function of SARSA, the following equation is obtaind:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t E_t(s,a), \quad (38)$$

for all s, a where

$$\delta_t = R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(St, A_t) \quad (39)$$

Being an on-policy method, SARSA(λ) will approximate $q_\pi(s,a)$, the action values for policy π, then improve the policy gradually based on these approximations for the current policy[6]. The complete SARSA(λ) method is shown in Figure 28.

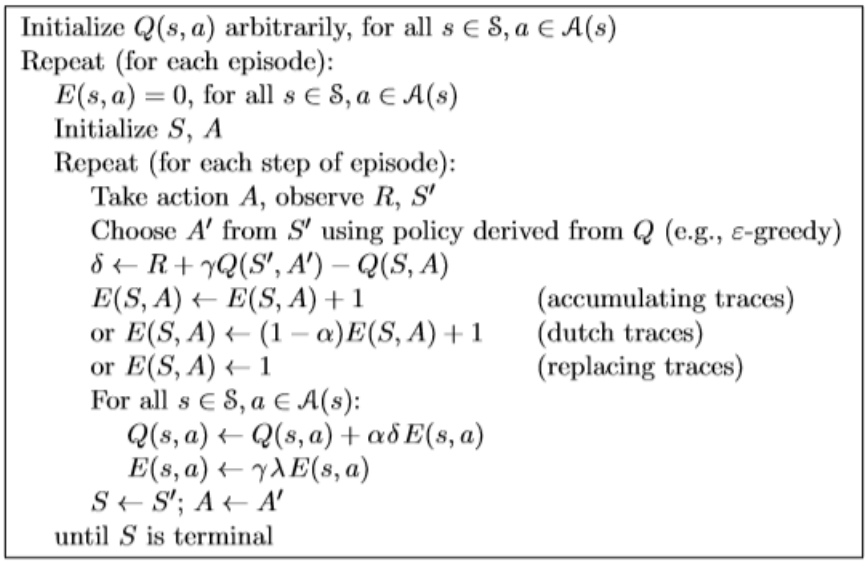


Figure 28: SARSA(λ)

4.4.5 Watkin's Q(λ)

Like SARSA, eligibility traces can be applied in Q-Learning methods. However, instead of looking ahead until the end, Watkin's Q(*lambda*) only looks as far as the next exploratory move. Being an off-policy method, Watkin's Q(*lambda*) learns the optimal policy from the Q function that approximates $q_\pi(s,a)$ independently of the current policy being followed as long as there are enough exploratory actions[6]. As a result, eligibility traces are assigned to the optimal policy being discovered and will only be applied to state-action pairs that occurred because of it. For example, in a given task, the agent chooses a greedy action on the first five steps and the one after that is an exploratory one. Because the sixth one does not correspond to the optimal policy, we can use the one-step, two-step, all the way to the five-step return, but not the six-step return.

The integration of eligibility traces to Q-learning to form Watkin's Q(λ) algorithm is simple. Eligibility traces are used in the same way as in the SARSA algorithm except when an exploration action is taken. If that is the case, the eligibility trace of that state-action pair is set to 0. The trace update process can be divided into two separate steps: the traces of all state-action pairs are

either decaying by $\gamma\lambda$ or set to 0. Then, the corresponding traces to the current state and action is incremented by 1 [6]. These steps are computed as follow:

$$E_t(s,a) = \begin{cases} \gamma\lambda E_{t-1}(s,a) + I_{sS_t} \cdot I_{aA_t} & \text{if } Q_{t-1}(S_t,A_t) = \max_a Q_{t-1}(S_t,a); \\ I_{sS_t} \cdot I_{aA_t} & \text{otherwise.} \end{cases}$$

Of course, the eligibility trace can either be accumulating, replacing or dutch. By combining $E_t(s,a)$ with the Q-Learning algorithm, we obtain this equation:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t E_t(s,a), \quad \forall s \in S, a \in A(s) \quad (40)$$

where

$$\delta_t = R_{t+1} + \gamma max_a Q_t(S_{t+1}, a') - Q_t(S_t, A_t) \quad (41)$$

The complete outline of Watkin's Q(λ) is given in Figure 29.

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $A^* \leftarrow \text{argmax}_a\, Q(S',a)$ (if A' ties for the max, then $A^* \leftarrow A'$)
    $\delta \leftarrow R + \gamma Q(S', A^*) - Q(S,A)$
    $E(S,A) \leftarrow E(S,A) + 1$ (accumulating traces)
    or $E(S,A) \leftarrow (1-\alpha)E(S,A) + 1$ (dutch traces)
    or $E(S,A) \leftarrow 1$ (replacing traces)
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
      If $A' = A^*$, then $E(s,a) \leftarrow \gamma\lambda E(s,a)$
                  else $E(s,a) \leftarrow 0$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

Figure 29: Watkin's Q(λ)

The efficiency of Watkins's Q(λ) algorithm is greatly dependent on the number of exploratory actions the agent will take. If the machine explores a lot, the eligibility traces will be less effective and most backups will be done on a two-step return.

5 Neural networks

Reinforcement learning algorithms such as TD Learning, Q-Learning and SARSA, discussed in the previous section, are different algorithms that train a learning agent to achieve a specific goal through environmental interactions [6]. Reinforcement learning highly contrasts with supervised learning algorithms. The latter enables computers to learn how to associate some inputs to some outputs, given a observational set of data or examples of inputs x and outputs y [7]. The set and examples are often provided by a human "supervisor". This type of learning is studied in neural network [6].

As shown in the figure below, neural networks are composed of multiple layers of neural connectivity often known as multilayer perceptrons or MPLs. This network has been inspired by the human brain's neural network. Indeed, it is observed that the neurons are depicted as circles or nodes and the synapses as weighted lines. Figure 30 depicts a feedforward network with an input layer, an output layer and one hidden layer.

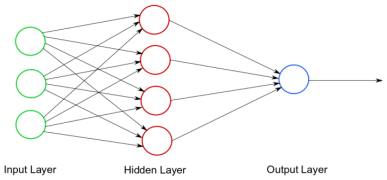


Figure 30: Simple Neural Network.

However, neural networks are not limited to one hidden layer. In fact, increasing the amount of hidden layers greatly benefits the accuracy in the neural networks' approximation of a function. This particular characteristic will be explained later on. The figure below demonstrate appropriately how neural networks act as supervised learning algorithms. The network associates some inputs x with some outputs y through the synapses in the hidden layers. While this is a general description of how neural networks work, further explanation will be provided later to lay out an adequate framework to explain how neural networks learn.

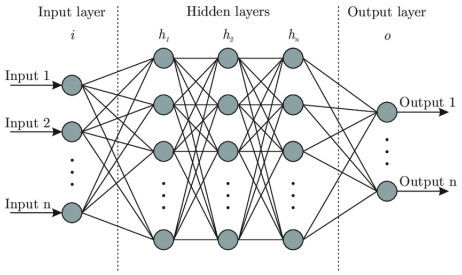


Figure 31: An example of neural network.

5.1 Structure of a Neural Network

Neural networks offer solutions in many recognition domains such as facial recognition, speech recognition and pattern recognition. A particular application of neural network is that they can approximate any function with incredible accuracy, provided that the function is continuous."The universal approximation theorem states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one infinite-dimensional space to another with any desired non-zero amount of error, provided the network is given enough hidden units" [7]. This mean that regardless of the function we're trying to learn, linear or non-linear, there will always be a MLP (multilayer perceptrons) to represent an approximate the function.

To understand this particular application, let's understand the structure of a neural network by proposing the following example: physical contact with a hot surface. As mentioned previously, artificial neural networks were inspired from the brain's biological neural network. In an example where one's hand is placed on the stove's hot surface, the hand's sensory neurons are activated through the two stimulus, the surface's temperature and the pain inflicted, and a signal is transmitted through the network to the brain where it sends back a stimuli to the motor neurons in the hand, ordering to remove the hand. This situation can be represented by a function that an artificial neural network will approximate. Its' structure is shown in the figure below.

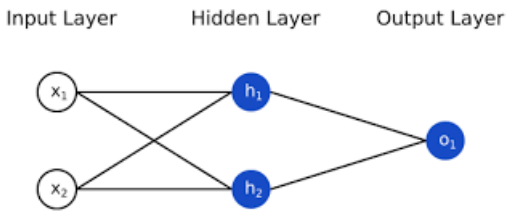


Figure 32: Simple three layered Neural Network with two input neurons and one output neuron.

The input layer takes two sets of data or information, in this case, the top neuron receives a value for the surface's temperature and the bottom neuron receives a value for the pain inflicted. The values are called activation numbers. The activations of the first layer determine the activations of the hidden layer, which in turn will determine the activation of the neuron in the output layer. This process is analogous to the neurons firing others in the brain's network. In this example, the activation number of the top neuron takes two inputs, x_1 and x_2, and produces the output for the h_1 neuron. It should be noted that the h_1 neuron is attached to x_1 and x_2 through "weighted" connection. In other words, these connections have "real numbers expressing the importance of the respective inputs to the output" [3]. Comparatively to the brain, whenever an individual learns to accomplish a specific task, for example, learning a programming language, the connectivity between neurons during the learning phase is firstly developed. Then, through repetition, the connectivity associated with the task at hand will strengthen, favouring certain connectivities while unfavouring others. The weights attributed to each connections in the neural network are parameters. Thus, the output of the h_1 neuron is determined by taking the activation number of each neurons in the previous layer (input layer) and computing the weighted sum such that

$$output = w_1x_1 + w_2x_2[3]. \tag{20}$$

In general terms, for an arbitrary neuron,

$$output = w_1x_1 + w_2x_2 + w_3x_3 + ... + w_nx_n[3]. \tag{21}$$

Furthermore, the outputs, depending on the weights and activation number of the previous neurons, can compute to any $\mathbb{R}$ number. However, the outputs are desired to be in the range [0, 1]. To force such outputs in this range, as quoted earlier, it is need to apply an activation function such as the logistic sigmoid activation function.

5.1.1 Logistic Sigmoid Activation Funtion

The sigmoid function, denoted by the following equation, $\sigma(x) = \frac{1}{1+e^{-x}}$, is shown in the figure below.

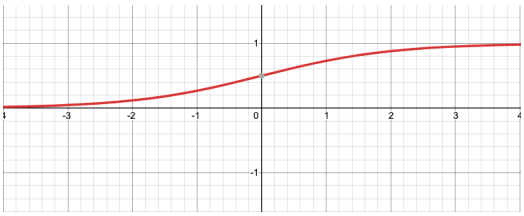


Figure 33: Sigmoid Activation Function.

From the graph, it can be analyzed that across the domain, negative inputs compute to values near 0 while positive inputs compute to values near 1 [7]. As expected, this activation function squashes the weighted sums between 0 and 1. This reveals how positive the weighted sums are [3]. By implementing this sigmoidal activation function in Equation (21), we get the following:

$$output = \sigma(w_1x_1 + w_2x_2 + w_3x_3 + ... + w_nx_n)[3]. \tag{22}$$

5.1.2 The Bias

We will introduce the concept of a thresh hold value, called the bias. This parameter, when applied on the weighed sum, signify how high the weighted sum needs to be before the neuron can be active [3]. In other words, the neurons' bias brings a degree of importance in whether or not the neuron should be activated. The following equation includes the concept of bias, denoted as b, in Equation (23) which shows the computation for a neuron's activation number. Below Equation (23) shows the computation under matricial form for a neuron in the hidden layer.

$$activation = \sigma((w_1x_1 + w_2x_2 + w_3x_3 + ... + w_nx_n) + b)[3]. \tag{23}$$

$$a_1^2 = \sigma\left(\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{bmatrix} \begin{bmatrix} x_1^1 \\ x_2^1 \\ \vdots \\ x_m^1 \end{bmatrix} + \begin{bmatrix} b_1^1 \\ b_2^1 \\ \vdots \\ b_m^1 \end{bmatrix}\right)$$

Moreover, instead of having neurons fire when their weighted sums are greater than 0, we can purposely make them fire at some other value, either positive or negative. If the bias is a large positive value, the equation indicates how easy it is for the neurons to be activated, and thus activates the neurons in the following layer. If the bias is a large negative value, it is harder for the neurons to be activated as their weighted sums must be greater than that value in order to output a significant positive value.

5.1.3 Rectified Linear Unit

While some neural networks implement the logistic sigmoid activation function, other neural networks use rectified linear units function (ReLu). This function outputs any negative input of a neuron to 0. In contrast, as the name indicates, positive inputs will be linearly outputed. Mathematically, the function is defined as $g(z) = max[0, z]$ [7]. ReLu are widely used in Reinforcement Learning whenever a neural network contains multiple hidden layers. Since ReLu outputs 0 across half of its domain, it is more reliable and efficient to use ReLu instead of the sigmoid activation function for feedforward neural networks due to the latter's widespread saturation. ReLu appeals better to the gradient-based learning than the sigmoid function [7]. The figure below shows the characteristic of the rectified linear units.

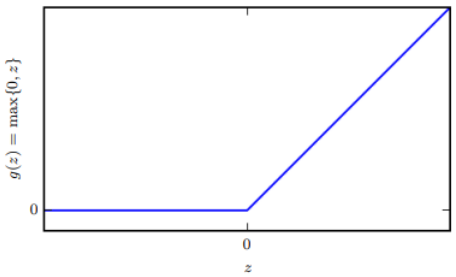


Figure 34: Rectified Linear Unit [7]

5.2 Neural Networks as function approximators

Now that an appropriate framework has been layed out concerning neural networks' structure and mechanism, let's explore the initial application of neural networks as function approximators. Given a particular function, we know from the universal approximation theorem, there will be an MLP capable of representing an accurate approximation of the function, provided many hidden layers. To prove this, let's propose the following function: f(x) = $cos^2 x + x$.

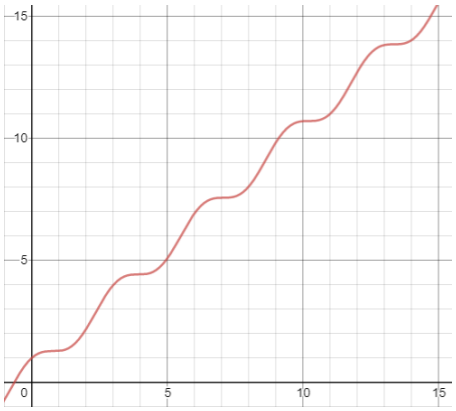


Figure 35: f(x) = $\cos^2 x + x$

5.2.1 Visual understanding of weights and bias

First, we construct a simple neural network with only one input neuron, two hidden neurons and one output neuron. It is possible to build complicated systems and functions by assembling many simple and smaller components [7]. To understand how a three layered neural network can approximate this given function, let's visually understand how the weights and biases affect the output of a neuron through a graph. In this specific example, the activation function is the logistic sigmoid activation function. Note that the following example, figures, explanations, and equations are based on Michael Nielsen's book on deep learning, called *Neural Networks and Deep Learning*.

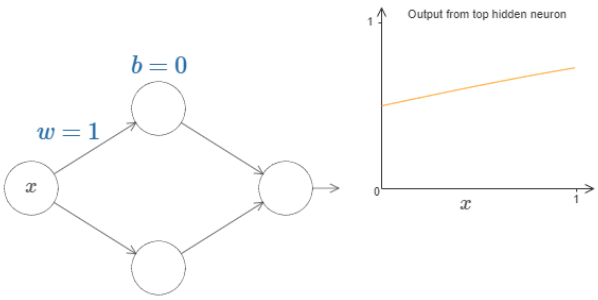


Figure 36: Output from the top hidden neuron.

For this example, the activation number of the input neuron is assumed to be equivalent to 1. Then, by carefully picking the weight and bias to be respectively 1 and 0, the graph depicted resembles a piece of the sigmoidal activation function (refer to Figure 35). This should be expected considering the chosen parameters. The following figures show substantial changes to the graph if we were to change the initial set of parameters.

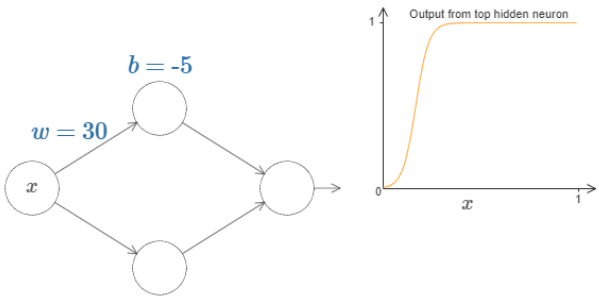


Figure 37: The output changes as parameter change.

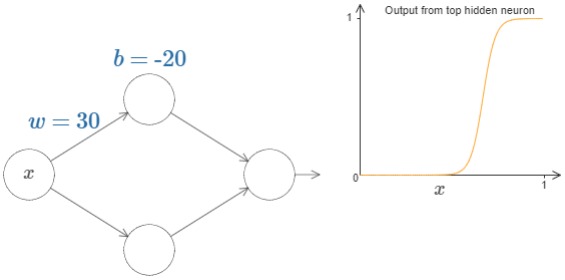


Figure 38: Increasing the bias of the previous figure.

Observe how the bias parameter affects the graph in Figures 37 and 38. We observe that by keeping the weight parameter constant, as the bias parameter decreases from $b = -5$ to $b = -20$, the graph's shape and curve remain the same, but its position has been displaced towards the right. Inversely, as the bias' value increases, the graph's position is displaced to the left.

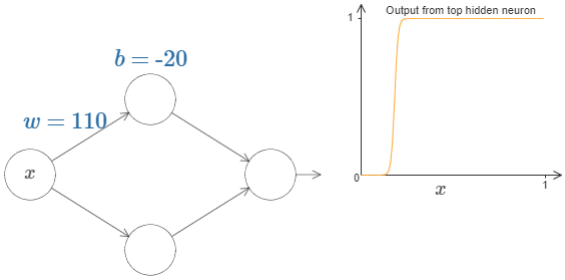


Figure 39: Keeping the bias constant, but changing the weight.

Meanwhile, comparing Figures 38 and 39, where the bias parameter value has been kept constant, the graph's curve has steepen as the weight's value

increased drastically from $w = 30$ to $w = 110$. Indeed, the weight parameter influences the steepness of the curve. As the weight's value approaches large numbers, the curve get steeper. Eventually, the curve becomes so steep that the graph approximately outputs a step function shown below. It should be noted that the bias parameters has been decreased in order to keep the curve in a centered frame.

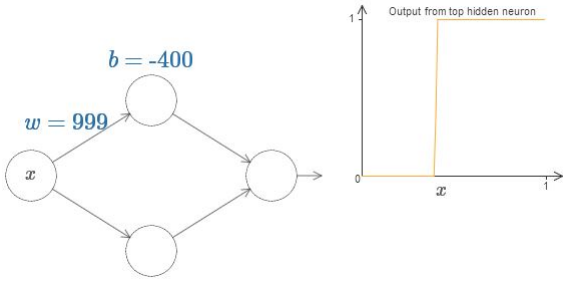


Figure 40: An approximate step function is outputed as the weight's value reaches high numbers.

It is preferred to deal with step functions rather than sigmoidal functions since it is required in a later step to add up all their contribution into the output neuron. Thus, it is easier to sum up step functions than to sum up sigmoidal functions by manually fix every weight and bias of each neuron in the hidden layer.

By varying the weight and the bias at the same time, it can be observed that the position of the curve changes. Therefore, the position of the step function is proportional to the weight and the bias. The following equation and figure demonstrate the proportionality where s is the step function's position.

$$s = \frac{-b}{w} \tag{24}$$

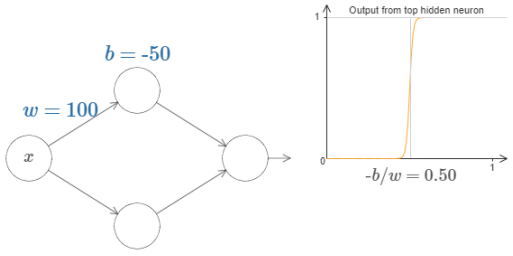


Figure 41: Relationship between the weight, the bias, and the step's position.

5.2.2 Multiple step function

Now that a visual understanding of weights and biases has been reach and that a new parameter has been designated to control directly the step's position, let's look at the behaviour of the network considering also the bottom hidden neuron.

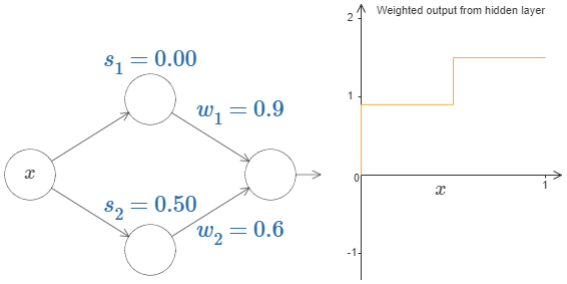


Figure 42: Behaviour of the neural network.

The top and bottom neurons are respectively parameterized as s_1 and s_2. Both have weighted connections to the output neuron. From the values of s_1 and s_2, it is understood that the top neuron's step function starts at $x = 0$ and the bottom neuron's step function starts at $x = 0.5$. This means that the first neuron to fire or to activate is the top neuron followed by the bottom neuron. Such order can be changed simply by making $s_2 < s_1$. Again, the graph indicates the weighted output from the hidden layer, which is the weighted sum of the hidden layer's neurons (refer to Equation 23). The weight parameter of each neuron will have an effect on the overall behaviour of the graph. Indeed, changing a neuron's weight will change the neuron's contribution to its respective step function. For example, reducing w_1 to zero would lead to the weighted output of the top neuron in the hidden layer to zero, as shown in the figure below.

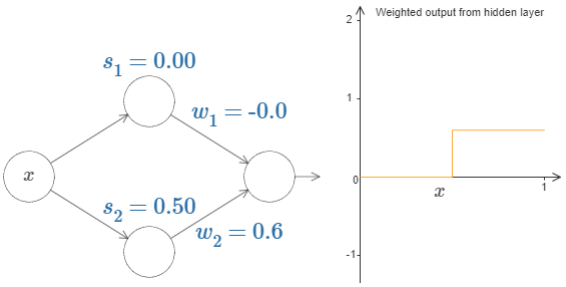


Figure 43: Contribution of w_1 to the graph.

Furthermore, from the graph, the height of the bottom neuron's step function lies directly at $y = 0.6$, which is the value of w_2. Note that this height is not

relative to $y = 0$, but rather the height of the first step function. It should be noted that coincidentally, it is also zero. Keeping this in mind, what would happen if $w_1 = -w_2$ while keeping $s_1 = 0.00$ and $s_2 = 0.50$? Suppose, $w_1 = 0.6$ and thus, $w_2 = -0.6$. It should be expected that the weighted output of the top neuron compute to 0.6, while the weighted output of the bottom neuron compute to -0.6 relative to the former. The graph should represent a "bump" or a vertical rectangle. This vertical rectangle will prove to be useful when approximating the function $f(x) = cos^2 x + x$ as we add more neuron pairs in the hidden layer.

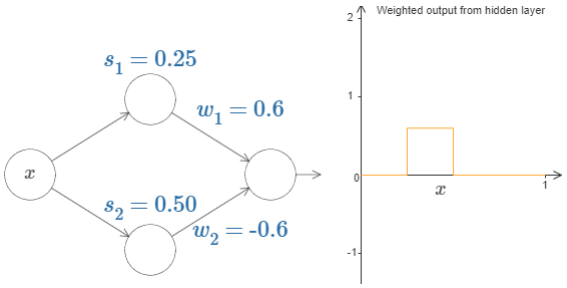


Figure 44: A vertical rectangle is graphed due to $w_1 = -w_2$.

To simplify, to directly control the height of this vertical rectangle, the proportion $w_1 = -w_2$ will be attributed as the parameter h. In this approximation example, the neural network only has one pair of hidden neurons, thus only one vertical rectangle. To approximate the given function $f(x)$, more pairs of hidden neurons should be added in order to have more vertical rectangles to control. Indeed, the neural network should be structure as follow if four pairs of hidden neurons were added.

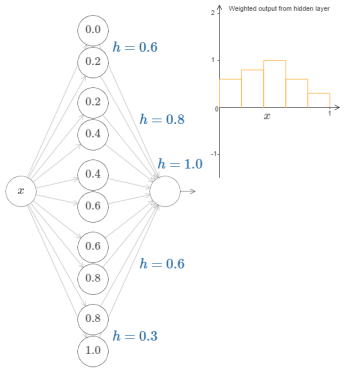


Figure 45: Neural Network with five pairs of hidden neurons.

Each pair has a parameter h that can be controlled in order to approximate

$f(x)$. As the parameters change, the neural network is actually approximating the function. The weighted output graph from the hidden layer should choose specific values for the h parameters in order to approximate $f(x)$. The figure below shows what the graph should look like. $F(x)$ is approximated using only 5 step functions. Thus, there will be a significant lack in accuracy. However, the approximation's accuracy can be improved by increasing the amount of step functions. In other words, increasing the amount of hidden neuron pairs in the a same layer, or even implementing multiple hidden layers. Recall that any function can be approximated with great accuracy provided the neural network has enough hidden units.

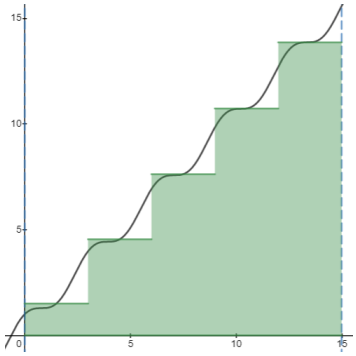


Figure 46: Neural Network approximating $f(x)$.

5.3 How do Neural Network learn?

The process of approximating a function is rather long since the parameters, w, b, s, h, have to be manually fixed. In reality, this can be quite cumbersome as functions are not as simple as the one proposed. Complicated functions require rather large MLPs, and manually setting each parameter would be wasted efforts. Instead, wouldn't it be possible to train the neural network to learn on its own the specific values at which each parameter should be set in order to approximate a function? Indeed, this is possible due to machine learning algorithms. "Most deep learning algorithms are based on an optimization algorithm called stochastic gradient descent" [7]. However, stochastic gradient descent method will be explored in later sections. This section emphasizes, under the framework of reinforcement learning, the gradient descent-based learning implemented by the back propagation algorithm.

5.3.1 Gradient Descent-Based

A neural network learns with the gradient descent algorithm. This algorithm involves the optimization of some function $f(x)$ (different from a function a neural network wants to approximate) either by maximizing or minimizing

it by varying its input, x. Some deep learning algorithms would tend to optimize by maximizing some function. An example which will be discussed in later sections is the REINFORCE algorithm. Most algorithms would minimize this function $f(x)$. Formally, this function is called the objective function or criterion. In the gradient descent algorithm, this function is usually referred to as the cost function, lost function or error function. The latter will be used [7]. As mentioned previously, neural networks have many application in image segmentation, image recognition and pattern recognition. Suppose we build a neural network that learns to recognize handwritten digits from 0 to 9. The network process a 28 by 28 pixel image of a digit and outputs the correct recognized digit. Here's the constructed neural network.

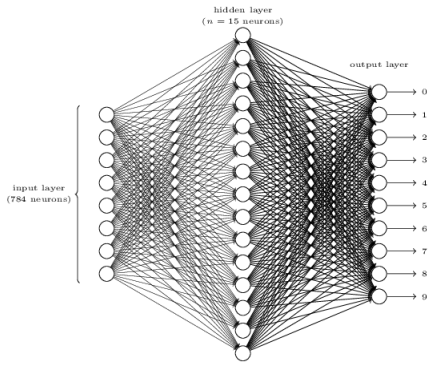


Figure 47: A Neural Network built to recognize handwritten digits.

In order to quantify the significance of the network's image processing training, a cost function has been established. The cost function is to measure the sum of the squares of the differences between each of the training's output (the activation number of each output neuron when the network processes an image) and the desired output (the desired activation number when the network processes an image). This is also known as the mean squared error. Suppose the network inputs a 28 by 28 image depicting the digit 3. The desired outputs is denoted as a 10-dimensional vector, y(x) = (0, 0, 1, 0, 0, 0, 0, 0, 0, 0)^T while the training's outputs might be denoted as a(x) = (0.20, 0.10, 0.55, 0.15, 0.28, 0.30, 0.25, 0.10, 0.21, 0.28)^T where x represents the training inputs [3]. For each training input, the neural network will calculate using the cost function using the mean squared error. Mathematically, the cost function is defined by the following equation

$$C(w,b) = \frac{1}{2n}\sum_x |y(x) - a(x)|^2 [3]. \tag{25}$$

where w, b represent respectively every weight and bias, n is the number of training input, a is the network's output. Remember that the neural network

function takes in 784 pixels and through some decision making based on parameters such as weights and biases, outputs 10 numbers. Here, the cost function takes the weights and biases as inputs, and outputs a non-negative value that indicates how well the weights and biases help the network achieve its goal (i.e to recognize handwritten digits) based on many training sets. The smaller the value, the closer $a(x)$ is to $y(x)$, meaning that the weights and biases are optimal. Notice that the cost function is summed over all the training sets. This ensures that when the gradient descent algorithm is employed, the cost function for all training sets is optimized. In essence, the gradient descent algorithm's objective is to minimize the average cost function, $C(w, b)$, by finding appropriate sets of weights and biases for each training example [3].

In this example of handwritten recognition, the cost function has over 13 thousand parameters, let's reduce the amount of parameters to a single parameter, v. Thus, The cost function, $C(v)$ has inputs v such that its dimension space has been reduced to a $\mathbb{R}^2$ space. This reduction in complexity allows us to use analytical methods such as calculus to solve for an input v such that $C(v)$ is either a local minima or a global minima. Indeed, by taking the first derivative and second derivative of $C(v)$, such input v would be easily determined. The figure below proposes a simple example with the concept of gradient descent.

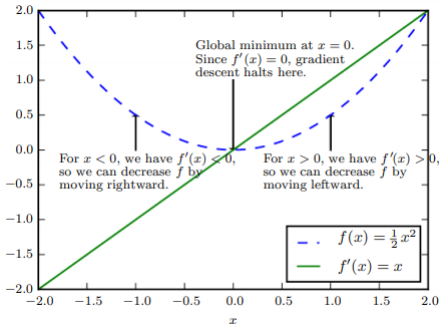


Figure 48: Using calculus to determine the extremum [7]

Here, the idea is to start at some $C(v)$ for some input v, and take a small step in the direction that would converge to a local minima, or preferably, a global minima. The function's derivative, denoted as $C'(v)$, at some point v indicates in which should the step, denoted as h, in order to make $C(v + h)$ lower. $C(v)$ can be reduced by small steps in the opposite direction of the input's slope. Therefore, if the slope at input v is positive, then $C(v)$ is decreased by stepping leftward. Meanwhile, if the slope is negative, then the step should be rightward. The process of repeatedly taking the slope and taking a step in a direction would eventually converge to a local minima [7].

Now, returning to the original handwritten digit recognition example, the cost function has complex dimensions that cannot be visualized, but can be conceptualized. Indeed, for cost functions with multiple parameters, the concept

of partial derivatives, rather than ordinary derivatives, is used such that the gradient vector of $C(w, b)$ is defined as

$$\nabla C = (\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial b_1}, ..., \frac{\partial C}{\partial w_n}, \frac{\partial C}{\partial b_n})^T \tag{26}$$

where the partial derivatives indicates how much the function changes with respect to each parameter. In multiple dimensions, critical points are defined as points where every partial derivative of the function's gradient is zero. The gradient of the function indicates the direction of steepest ascent such to increase the function most rapidly. Thus, taking the negative gradient will indicate the direction in which the step would decrease the function most rapidly which is known as the method of steepest descent or gradient descent. Conceptually, the gradient points directly uphill and the negative gradient points downhill. By taking a step, proportional to the negative gradient, in the latter's direction, a new point in space is defined as the following

$$w'_n = w_n - \epsilon \nabla C(w_n) \tag{27}$$

[3].

$$b'_m = b_m - \epsilon \nabla C(b_m) \tag{28}$$

[3].

where ϵ represents the learning rate which is a small positive scalar that determines the step size.

Recall that the gradient descent method is to minimize the cost function over all the training inputs. This method achieves this by adjusting every parameters, w_k and b_l, for each training set, by a small amount in the negative gradient direction until it converges to a local minima. The parameters at the local minima are the values for every w_k and b_l that minimizes the average cost function.

5.3.2 Back Propagation

Neural Network are known for their forward propagation of information. Indeed, they are structured to accept an input x, and produce an output y, such that the flow of information is forward. From the input layer, to the hidden layers, and finally to the output layer. The forward propagation of information during their training allows the neural network to output a value, called the cost function, often denoted as $C(\theta)$ or $J(\theta)$ where θ is a set that contains all parameters w_k, b_l of the network. Much like how, in the previous section, the input v of $C(v)$, characterized the parameters. The information of the cost function, $J(\theta)$, flows backward through the network, using the back propagation algorithm. This lets the neural network compute efficiently $\nabla_\theta J(\theta)$, the gradient of $J(\theta)$. This algorithm gives specific details to the neural network in how changing the weights and biases affect its behaviour. This is how neural networks learn [7].

To compute $\nabla_\theta J(\theta)$, as mentioned previously, back propagation must compute the partial derivatives with respect to every element in the set θ.

First, let's define a new notation to rewrite Equation (23) which was a general equation to determine the activation number of a neuron. The weight for the connection from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^th layer is denoted as w^l_{jk}. The bias of the j^{th} neuron in the l^{th} layer is defined as b^l_j and the activation of the j^{th} neuron in the l^{th} layer, as a^l_j. Therefore, rewriting Equation 23 using this notation, we get the following equation

$$a^l_j = \sigma(\sum_k w^l_{jk} a^{l-1}_k + b^l_j) \tag{29}$$

Furthermore, this notation can be rewritten under matricial form if we define a weight column vector w^l with entries w^l_{jk}, an activation column vector a^{l-1} with entries a^{l-1}_k, a bias column vector b^l with elements b^l_j. This results in the the column vector a^l with entries a^l_j described by this equation

$$a^l = \sigma(w^l a^{l-1} + b^l) \tag{30}$$

We would also like to compute an intermediate quantity defined as $z^l = w^l a^{l-1} + b^l$ such that

$$a^l = \sigma(z^l). \tag{31}$$

Then, another intermediate quantity, δ^l_j, is introduced, which is the *error* in the j^{th} neuron in the l^{th} layer. This intermediate quantity is defined as

$$\delta^l_j = \frac{\partial J}{\partial z^l_j}. \tag{32}$$

The backpropagation is based on four fundamental equation that will lead to the computation of $\nabla_\theta J(\theta)$. The first is an equation for the error in the output layer defined as

$$\delta^L_j = \frac{\partial J}{\partial a^L_j} \sigma'(z^L). \tag{33}$$

which can also be rewritten as

$$\delta^L = \nabla_a J \odot \sigma'(z^L). \tag{34}$$

The second is an equation for the error in terms of the error in the next layer defined as

$$\delta^l_j = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l). \tag{35}$$

The third equation is for the rate of change of the cost function with respect to an arbitrary bias of the network.

$$\delta_j^l = \frac{\partial J}{\partial b_j^l}. \tag{36}$$

The final equation demonstrates the rate of change of the cost function with respect to an arbitrary weight of the network.

$$\frac{\partial J}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l. \tag{37}$$

.

These four fundamental equations, taken from Nielsen's book, *Neural Networks and Deep Learning*, provide a method to compute the gradient of the cost function of a neural neural. Indeed, the algorithm starts by setting activation numbers for each neuron in the input layer. The neural network propagates the information forwardly to the following hidden layers and output layer, computing $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$. Then, the network outputs the error, δ^L, using the first fundamental equation. The network backpropagates the error, δ^l, obtained by the second fundamental equation, from the output layer to each previous layer for each training input. Finally, the gradient of the cost function is computed as the third and fourth fundamental equations determine the partial derivatives of the gradient. Thus, since the gradient for a single training input has been computed, the network can update its parameter in the direction of the negative gradient in order to optimize the cost function. This process is repeated for all the training inputs, until the parameters converge to values such that the cost function is at a local minima [3].

6 REINFORCE

The REINFORCE algorithm includes a study of parameter policies that create the likelihood of action in a given state. Agents use this strategy directly to act in the environment. The key idea invented in 1992 by Ronald J. Williams in "A Simple Algorithm for Tracking Statistical Gradients to Improve Learning Abilities" is that actions that produce good learning results should be more likely. (These measures are supported positively). Conversely, actions with poor results should be less likely. If training is successful, the potential for action due to policy changes in the direction of deployment in many iterations results in superior performance in the environment. The likelihood of action depends on the policy gradient where REINFORCE is known as the policy gradient algorithm [1].

6.1 Monte-Carlo Policy Gradient

The REINFORCE algorithm is a Monte-Carlo variant of policy gradients. The Monte-Carlo are computational algorithms that use repeated sampling for approximating numerical values. The Monte-Carlo algorithms are episodic

meaning they must complete a full run of the numerical analysis before the experiment can be repeated again. Using the Monte-Carlo method with the Policy Gradient methods allow us to optimize the parameterized policies used in the neural networks which control the optimal actions to take in the current states we are in using probability distribution functions. To do this, we first need a parameterized policy, an objective function to be maximized and a method for updating the policy parameters. We need a policy function which is being approximated by the neural network. For this function to be maximized we use an agent which generates a trajectory T [1]. The return of the trajectory is the sum of the discontinued rewards from time t where:

$$R_t(T) = \sum_{t'=t}^{T} \gamma^{t'-t}\ r'_t \tag{38}$$

We are looking for the expected return over all possible trajectories generated by the agent which is

$$J(\pi_0) = E_{(T\sim\pi_0)}[R(T)] = E_{(T\sim\pi_0)}[\sum_{t'=t}^{T} \gamma^{t'-t}\ r'_t] \tag{39}$$

These expected values can be approximated by sampling trajectories under the policy π_0. Our objective is to maximize the objective J(π_0). To reach this maximal value, we use gradient ascent on the policy parameter θ in the direction of the steepest ascent following the trajectory which leads us to the Policy Iteration rule [1].

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)},$$

Figure 49: Policy Iteration Rule

One of the advantages of using parameterized policies is that the approximate policy can be deterministic. This is seen as an advantage because it removes the random probability of selecting a non-optimal action because of the ϵ-greedy algorithm which causes randomness for exploring new trajectories. One could select according to a soft-max distribution based on action values, but this alone would not allow the policy to approach a deterministic policy[6]. Another advantage of using parameterized policies would be that it enables the selection of actions with arbitrary probabilities. For example in the case where the optimal probability approximation would be stochastic like bluffing in poker where there is an uncertainty involved. The parameterized policies are also very easy to implement which helps with simpler functions as it allows for much faster and a decently precise approximation [6]. Since the policy parameterization is continuous, it allows for smooth changes over the policy probabilities as a function of the learned parameter whereas in ϵ-greedy selection there can be a change of outcome based on even the smallest changes in the

estimated action-values if it changes the action which has the maximal value [6].

Through optimizing the policy, the weights of the neural network also become optimal. The Policy Iterations are ran numerous times to achieve the best possible policy for a state by using the results from the previous iteration and through this method, the weights of the neural network will converge to a given value.

6.2 Reinforce Algorithm

Through the policy gradient theorem which shows:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta})$$

Figure 50

We can derive the REINFORCE algorithm because the sample gradients need only be proportional to the gradient because any constant of proportionality can be absorbed into the step size α and we arrive at the REINFORCE algorithm:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}.$$

Figure 51: REINFORCE Algorithm

Each increment is proportional to the product of a return G_t and a vector which is the gradient of the probability of taking the measurements actually taken, divided by the probability of taking this measurement. The vector is the direction in a parameter space that further increases the probability of repeating the action,A_t, during future visits to indicate S_t. The update increases the parameter vector in this proportional direction upon return, and is inversely proportional to the probability of action. This makes sense because it moves the parameter more in the directions that favor the actions that give the highest performance and because otherwise the actions that are frequently selected have an advantage and they could win even if they don't get the best performance. REINFORCE uses the complete return from time "t", which includes future rewards as well. It is because of this reason that REINFORCE is seen as a Monte-Carlo method and is only defined in episodic cases [6].

Algorithm 2 REINFORCE algorithm

1: **Initialize** learning rate α
2: **Initialize** weights θ of a policy network π_θ
3: **for** *episode* = 0, . . . , *MAX_EPISODE* **do**
4: Sample a trajectory $\tau = s_0, a_0, r_0, \ldots, s_T, a_T, r_T$
5: Set $\nabla_\theta J(\pi_\theta) = 0$
6: **for** $t = 0, \ldots, T$ **do**
7: $R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r'_t$
8: $\nabla_\theta J(\pi_\theta) = \nabla_\theta J(\pi_\theta) + R_t(\tau)\, \nabla_\theta \log \pi_\theta(a_t|s_t)$
9: **end for**
10: $\theta = \theta + \alpha \nabla_\theta J(\pi_\theta)$
11: **end for**

Figure 52: Steps for Initializing REINFORCE,[1]

It is important to disregard the trajectory after every iteration as the REINFORCE algorithm is an on-policy algorithm meaning it only takes into account and depends on the current policy. This can be seen as the policy gradient directly depends on action probabilities, $\pi_\theta(a_t \mid s_t)$ generated by the current policy π , but not some past policy . Correspondingly, the return R(T) where $T \sim \pi_\theta$ must also be generated from π, otherwise the action probabilities will be adjusted based on returns it wouldn't have generated [1].

α is a scalar known as the learning rate; it controls the size of the parameter update [6]. We want this learning rate to be small but if it is too small, the time taken to converge to a value will be extremely long and will take many iterations. In the case that it is large, the time taken for it to converge to a value will be fast but the variance will be immense and the approximation will not be very accurate.

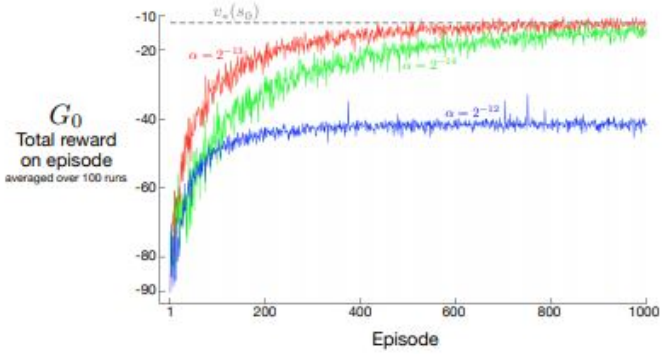


Figure 53: REINFORCE on the short-corridor gridworld. With a good step size, the total reward per episode approaches the optimal value of the start state [6]

.

When using Monte Carlo sampling, the policy gradient estimate may have high variance because the returns can vary significantly from trajectory to trajectory. This is due to three factors.First, actions have some randomness because they are sampled from a probability distribution. Second, the starting state may vary per episode. Third, the environment transition function may be stochastic [1].Therefore The REINFORCE Algorithm can be improved by subtracting a baseline from the return G_t. The baseline can be any function as long as it is action-independent; the equation remains valid because the subtracted quantity is zero [6]:

$$\sum_a b(s)\nabla\pi(a|s,\boldsymbol{\theta}) \;=\; b(s)\nabla\sum_a \pi(a|s,\boldsymbol{\theta}) \;=\; b(s)\nabla 1 \;=\; 0.$$

And through the derivation of this rule like of the general REINFORCE algorithm we arrive at:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big(G_t - b(S_t)\Big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}.$$

Figure 54: REINFORCE with baseline

The goal of the baseline is so that the variance of the return can be reduced thus allowing for the optimal policy to be approximated much faster. The baseline makes it so that the An option for the baseline function can be the state value, $V(S_t, w)$, where w is a weight vector; These are known as actor-critic algorithms [6].

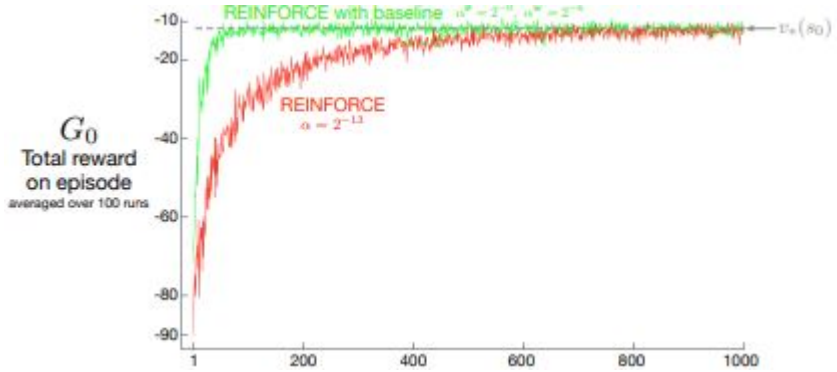


Figure 55: Adding a baseline to REINFORCE can make it learn much faster, as illustrated here on the short-corridor gridworld. The step size used here for plain REINFORCE is that at which it performs best [6]

.

6.3 Cart Pole Example

To demonstrate the REINFORCE algorithm we can use the cart-pole example. The objective of the cart-pole is to keep the pole upright for a time step which is chosen as seen in section 1.1. The objective is completed when that time step is reached. A neural network is created to approximate the parameterized policy function π. In the case of the cart-pole, π would represent the probabilities of taking an action between moving left or right, in an observed state.The neural network begins by taking the state of the cart-pole by looking at the cart's velocity and position, and the pole's angular velocity and angle. The network then uses its current policy to to output an action and a cost function which measure that indicates the fitness of the weights. The network then is rewarded from the the gradient of the probability of taking the action actually taken divided by the probability of taking that action. The information is then recalled to the network to update the weights and biases called back propagation. Meaning the the probabilities of committing to an action would be more in tuned to what will help the pole stay upright. This process is repeated episodically until the optimal policy would be found which would be until the cart has exceeded the time steps you have given it to complete.

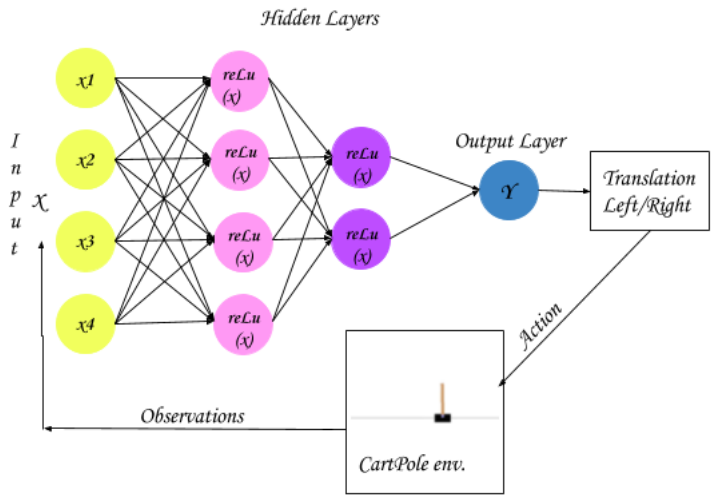


Figure 56: Neural network of Cart-pole Game

7 Proximal Policy Optimization algorithm

In recent years, a new class of reinforcement learning algorithms has taken the industry by storm as the proximal policy optimization (also known as PPO) has provided an algorithm that performs as well or even better than other approaches while being simpler and an easier algorithm to implement. This new family of reinforcement learning algorithms developed by Schulman alternates between sampling through previous data with the environment and using an objective function that stems from the ever optimizing stochastic policy which allows the agent to explore the state space through different paths and actions. This is done thanks to a stochastic gradient ascent which is a mathematical procedure that optimizes an objective function with smoothness.[4] The proximal policy optimization takes its principles from policy gradient methods and trust region methods and improves on them by simplifying the implementation and by using better sample complexity. For these reasons, PPO is becoming the default reinforcement learning algorithm for many artificial intelligence projects such as OPEN AI. This algorithm has been proven successful on a multitude of tasks going from robotics to complicated tasks such as playing Dota 2.[1] [5]

7.1 Supervised Learning

To ease in the concept of policy gradients methods, we will first touch upon supervised learning algorithms. This machine-learning algorithm functions by associating an input to the optimal output thanks to a gradient vector that has been computed from implemented and previous data. Simply put, its goal is

to approximate the mapping function or the gradient vector with accuracy to provide the optimal output variable from the new input data every time. To do so, the supervised learning algorithm learns by computing a gradient vector which will make the network more likely to predict the right action by telling it which parameters to change to increase the likelihood of making the right prediction. The gradient vector computed by the following equation

$$\nabla W logp(y = UP|x) \tag{40}$$

Often in reinforcement learning, log probability will be used for the simple fact that it makes the math cleaner. If we were to implement a change by entering a gradient of 1.0 on the log probability of the correct decision, the resulting gradient will change any one of the million parameters to make it more likely next time to predict said correct decision on its own. This is where the name supervised learning stems from as the process of the algorithm learning from the training data set consists of an algorithm making predictions on the training data (which consists of a set of training examples) and a person supervises its learning process and corrects its mistake. This process ends when the teacher has updated the parameter until the algorithm has achieved an acceptable level of performance. To facilitate comprehension, this is similar to the human psychological idea of concept learning. Also known as category learning, this concept consists of showing a person a set of examples with their respective labels. Then, the learner then has to observe and condense the observed examples into information. This information will then be used as the learner is faced with future examples where he will have to use his collected information to classify the new examples into labels. Finally, with supervised learning, the process essentially consists of implementing a cost function, running a gradient descent on it which will determine what parameters to change, and we will get positive results without much hyperparameter tuning to do.[2]

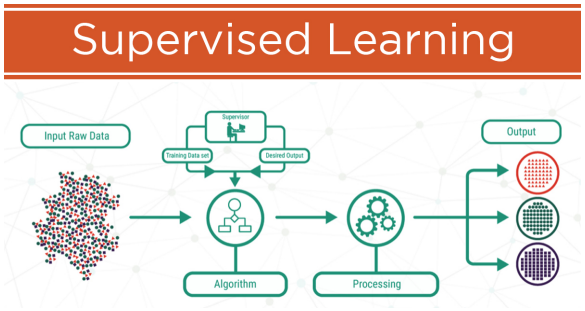


Figure 57: Supervised learning diagram where correct labels are provided [2]

7.2 Policy Gradients Methods

Supervised learning may be an effective algorithm for certain problems such as the recommendation list on Netflix, however there is still much to improve

for it to be useful in reinforcement learning as it is not as obvious to get good results. This brings us to Policy Gradients methods. In contrast to supervised learning, this method does not rely on providing the correct labels or a training data set before operating. Without any prior data, the policy network will calculate the prob of each possible action and one of the actions will be sampled from the distribution. At this point, we wait for the outcome of the action and if the outcome was positive, the gradient for said action gets rewarded by adding +1. Now if the outcome is negative, it will immediately fill in the gradient for said action with -1. Therefore, when the action leads to a good outcome, the gradient is filled in with +1 on said action and we do the back-propagation, it will result in a gradient that encourages the network to take the action that resulted in a good outcome. The same idea goes for any negative outcome, the gradient is filled with -1 for the action that lead to a negative outcome, and this will result in a gradient that discourages the network to take any similar actions. [2] Therefore, policy gradient methods work by computing an estimator of the policy gradient which has the form of

$$\hat{g} = \hat{E}_t[\nabla_\theta log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{41}$$

Here, the π_θ is the stochastic policy which allows the network to explore multiple options through multiple paths because it outputs a probability distribution over multiple actions according to the current state. In brief, this term is our policy, its a neural network that takes the observed states from the environment as an input and suggests actions to take as an output. The $\hat{A}_t$ is the advantage function which tries to estimate what the relative value of the selected action is and can be given by

$$A(s,a) = Q(s,a) - V(s) \tag{42}$$

Essentially it is the difference between the Q value (which is a value calculated to show which future path will be optimal) for a given state and the V value (which is the average of actions that it would have taken at that state). Thus, the advantage function tells the agent the extra reward which could be gained by taken an action. Therefore, it approximates with much noise if the action that our agent took was better than expected or worse. Furthermore, the $\hat{E}_t$ represents the expectation. To do so, it takes the mean of a finite batch of samples thanks to an algorithm that switches between sampling and optimization. Then by multiplying the log probabilities of your policy actions with the advantage function we can get the final optimization objective used in policy gradient methods

$$L^{PG}(\theta) = \hat{E}_t[log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{43}$$

Now if we look at what this objective function does we realize that if the advantage function is positive, which means that the action that the agent took in the sample trajectory resulted in a outcome better than the average action return at that point. Since the advantage function is positive, the gradient is

positive and this will increase the probability of the action the agent took in the future in similar states. This method brought many recent breakthrough in reinforcement learning thanks to the use of deep neural networks however it comes with a few downsides. [2] [5]

In gradient descent methods, the objective function is computed thanks to trajectories produced by the policy in the policy space. Since the optimal policy is found in the parameter space by finding the exact parameters that allows for it, finding the ideal step size is difficult because the distances in the policy space will not be equivalent to the distances in the parameter space. Since the policy gradient computes the steepest ascent direction for the rewards and update the policy towards that direction, too large of a step can lead to failure. A large step could overdo the update and cause a performance collapse as it has left the area with good policies, therefore, it is hard to recover from this as the next policy update will generate bad data for future updates. On the other hand, if the step size is too small, the progress will be very slow. Even though it can provide good results, policy gradient methods alone are not as efficient as it takes a lot of samples and millions or billions of time steps to learn simple tasks. Therefore, to avoid making too harsh of a change and reduce the sample size, the solution is to constraint the policy changes so the algorithm does not make any aggressive moves towards failure by moving too far away from the old policy. This is where trust region policy optimization comes into play. [1] [5]

7.3 Trust Region Policy Optimization

Now that we have taken a look at Gradient descent, we will expand on the topic of trust regions. In fact, there are two popular optimization methods: line search such as gradient ascent and trust regions. Line search works by determining the direction of descent and taking a step towards that direction. However, trust regions consist of finding the optimal point within a trust region after determining what is the maximum step size we want to take. Thus, the objective function

$$\hat{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \hat{A}_t \right] \tag{44}$$

is maximized and subject to the constraint size of

$$\hat{E}_t \left[KL[\pi_{\theta old}(.|s_t), \pi_\theta(.|s_t)] \right] \leq \delta \tag{45}$$

Now, the process consists of finding the optimal point of the first objective function within the constraints of δ. This is further repeated until the function is optimized. Also to prevent the updated policy to move to far away from the current policy, this method adds the Adaptive Kullback-Leibler Divergence as a hard constraint to the objective. This KL constraint has the sole purpose of preventing the new updated policy from straying too far away from the current policy so that the function ca stay close to the area where it knows everything works effectively. However, using the KL divergence to model sets also has its limitations as there is no guidelines in how it determines the size of the

ambiguity set. In fact, trust region methods can also work by using a penalty instead of a hard constraint to optimize

$$\hat{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \hat{A}_t - \beta KL[\pi_{\theta old}(.|s_t), \pi_\theta(.|s_t)] \right] \tag{46}$$

Here instead of using the KL as a hard constraint we use a penalty instead which forms a lower bound on the performance of the policy π. The principle of this method comes from the Minorize-Maximization MM algorithm which guarantees that the policy updates always improve the expected rewards by maximizing a lower bound function which approximates the expected reward locally as seen in figure 58.Therefore by using a max KL we get a lower bound on the policy performance. [5]

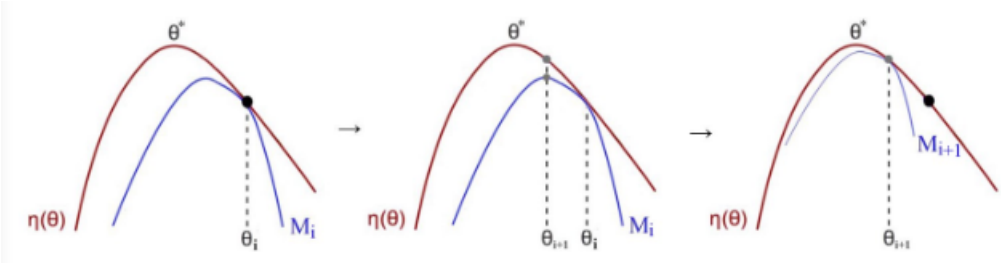


Figure 58: A Minorize-Maximization algorithm that maximizes a lower bound function (the blue line) by approximating the expected reward at the current guess. In the second diagram, the optimal point M found previously is used to find the next approximation. Eventually the blue line converges to the optimal policy η.[4]

However, experiments show that the previous method of using a penalty is less effective than using a hard constraint. The reason is that it is not sufficient to choose a fixed value for the β coefficient and optimize the penalized objective equation from there. The β as in essence the same purpose than equation (45). To function optimally, the value of β would need to be larger when the difference between π_θ and $\pi_{\theta old}$ is larger, while the opposite applies. As a result, the value β would need to change as the policy updates itself as it is impossible for a single value of β to perform well across different problems and harsh changes.[1] [5]

7.4 Clipped Surrogate Objective

This brings us to clipped surrogate objectives which is the main objective function used in proximal policy optimization. Let's begin by denoting the probability ration as

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \tag{47}$$

In TRPO (aka Trust Region Policy Optimization), the idea is that the value of $r_t(\theta)$ is the probability ratio between the updated policy and the old version of the policy network. This ensures that for a given sample of actions, the $r_t(\theta)$ value will be larger than 1 if the action at that state has a higher probability with the updated policy than with the older policy. On the flip side, the value will be between 0 and 1 if the opposite is true where the older policy is more likely. Now, by multiplying the $r_t(\theta)$ value with the advantage function we finally get the TRPO objective in the form

$$L^{CPI}(\theta) = \hat{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \hat{A}_t \right] = \hat{E}_t \left[r_t(\theta) \hat{A}_t \right] \tag{48}$$

Nevertheless, without any constraints, this objective function will have large policy updates and risk destructive policy updates. To solve this problem, the idea of clipped surrogate objective is introduced. Using clipped surrogate objective the following objective was obtained

$$L^{CLIP}(\theta) = \hat{E}_t \left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right] \tag{49}$$

This equation is the main objective function in proximal policy reinforcement. Once again in the objective function there is the expectation operator $\hat{E}_t$ which means that the objective function is computed over a batches of trajectories. In this objective function, the first term in the min operator is the previous equation L^{CPI}. Thus, this term is the default objective for normal policy gradients and will bring the policy to towards taking actions that will bring the highest positive advantage over the baseline. The second term in the min will clip the surrogate objective probability ratio. This term is similar to the first one however, it will be a shortened version of the first one as a clipping operation is applied at 1 - ϵ and at 1 + ϵ. ϵ here is a hyperparameter which is proposed to be 0.2. Therefore, we take the minimum of these two terms and the final objective will be a lower bound for the unclipped objective. [5]

Furthermore, the advantage function in the equation can either be a positive value or a negative one (as previously mentioned in part 7.2) and this changes the behaviour of the main operator. For instance, figure 59 shows the behaviour of the L^{CLIP} function when the advantage function is positive. Since the value of A is positive here, the actions chosen were better than the expected return.[5]

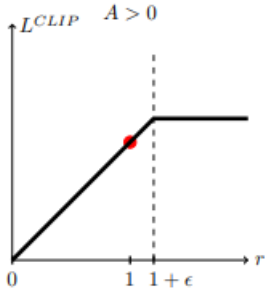


Figure 59: Plot showing the surrogate function L^{CLIP} as a function of the probability ratio r and the value of the advantage function is positive.[5]

Here when r gets high, the function flattens out. This happens because the action is much more likely under the current policy than on the older policy. Therefore, to prevent the update from overdoing, the objective function is clipped at $1 + \epsilon$ so that the gradient update is not too harsh. [5]

Now when we look at the function when A is lesser than zero, the opposite happens as seen in figure 60. Here the objective is flat when r is near zero and diverges towards negative infinity after 1 - ϵ. Since the value of A means that the actions here are less likely in the updated policy than in the older one. Consequently, when the probability ratio r is near 0, this will have the same effect than in the previous case where the update is prevented from doing any harsh updates which might lead these probabilities to zero. The reasoning for this is that because of all the noise, policies must be protected from being destroyed rapidly based on a single wrong estimate. In addition, the right side of 1 - ϵ, where the value of r is bigger, happens when the previous gradient step made a bad action more probable. Therefore since the value of r is bigger and the advantage is negative, this step made the policy worst than its previous iterations. In the even of this happening, the objective function of proximal policy optimization will work to undo the last bad gradient step. Since the function is negative here as seen in the next figure, the gradient will be negative and will make the bad action less probable by an amount proportional to how bad the update was. It is important to note that this portion of the second term is the only area where the unclipped term has a lower value than the clipped term. This means that the first term will be returned by the minimization operator.[1]

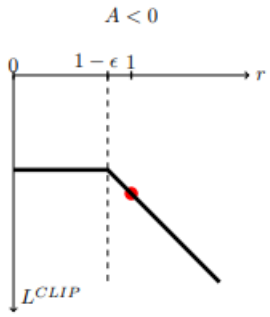


Figure 60: Plot showing the surrogate function L^{CLIP} as a function of the probability ratio r and the value of the advantage function is negative.[5]

Finally, with this main objective, the change in probability ratio is ignored when the objective is improving to prevent overstepping, and the probability ratio is taken in account when it makes the objective worse. Thus, this objective function will be the one used in proximal policy optimization as it keeps the policy update stable when it does drastic changes to the policy. [1] [5]

7.5 Proximal Policy Optimization Algorithm

After talking about pros and cons of supervised learning, policy gradient methods, trust region policy optimization, and clipped surrogate objective algorithms, we can finally discuss the combined method that has taken the reinforcement learning fields by storm: Proximal Policy Optimization. This method has been heavily favored over others since it is easy to implement and the step size does not need to be chosen beforehand. The proximal policy optimization algorithms exist in two different variants as seen previously. One uses the adaptive KL penalty like in 7.3 and the second uses a clipped objective like in 7.4. However, it was shown through multiple trials the clipped surrogate objective variant outperforms the the adaptive KL penalty as it is simpler and better as seen in figure 61. [5]

algorithm	avg. normalized score
No clipping or penalty	-0.39
Clipping, $\epsilon = 0.1$	0.76
Clipping, $\epsilon = 0.2$	**0.82**
Clipping, $\epsilon = 0.3$	0.70
Adaptive KL $d_{\text{targ}} = 0.003$	0.68
Adaptive KL $d_{\text{targ}} = 0.01$	0.74
Adaptive KL $d_{\text{targ}} = 0.03$	0.71
Fixed KL, $\beta = 0.3$	0.62
Fixed KL, $\beta = 1.$	0.71
Fixed KL, $\beta = 3.$	0.72
Fixed KL, $\beta = 10.$	0.69

Figure 61: Table of results from the experiments from the PPO paper [5]. Average normalized scores (over 21 runs of the algorithm, on 7 environments) for each algorithm / hyperparameter setting . β was initialized at 1.

Finally, by combining all the terms, we get the following objective:

$$L_t^{CPI+VF+S}(\theta) = \hat{E}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{50}$$

In the PPO objective function, we see the appearance of two new terms with constants c_1 and c_2. These constants are just hyperparameters that help weigh how much each term will contribute. The first additional term here is in charge of updating the baseline network. The reason why this is also part of the loss function is because the value and the policy are both part of the same computation graph. This is due to the fact that the parameters space is shared for both the policy network and the value estimation network, therefore, you are going to need similar feature extraction pipelines for both estimating the value of the current state and taking the best current action.[5]

The second term here is the entropy term. This entropy term ensures that the network does enough exploration by having more possible options which results in high unpredictability. This term is mostly important in the beginning phases as it will make the policy act more randomly to explore all possibilities before the other terms of the objective function start dominating in the later phases.[5]

Finally, we can see how proximal policy optimization is favored in machine learning as it is unaffected by tuning problems and is easy to code, and compute. Not only that, PPO is capable of reaching very good results on a multitude of tasks. [1]

8 Conclusion

While Reinforcement Learning may still be an area of machine learning that still hasn't been fully explored, it has definitely contributed in today's field

of artificial intelligence, providing machines a proper framework to learn the process of decision making. Its contribution has been observed in fields such as ATARI Games, Go, robotics, and image segmentation, where the combination of Reinforcement Learning and Deep learning has lead to massive breakthrough in solving challenging real world problems and situations. In terms of assessing technological opportunities, "Reinforcement Learning leads the way" [6] and [1].

References

[1] L. Graesser and W. Keng. *Foundations of Deep Reinforcement Learning.* Addison Wesley, 2020.

[2] Andrej Karphaty, May 2016.

[3] M. Nielsen. *Neural Networks and Deep Learning.* 2018.

[4] John Schulman. Proximal policy optimization, Mar 2019.

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[6] R. Sutton and Barto A. *Reinforcement Learning, An Introduction.* MIT Press, 2018.

[7] Bengio, Y., A. Courville, and Goodfellow, I. *Deep Learning.* MIT Press, 2016.