

Reinforcement Learning

Adam Tang
James O'Sullivan
Jordan Walsh

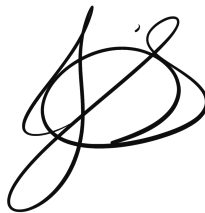
AM3064/AM6015 Written Report

School of Mathematical Sciences
University College Cork
Ireland
April 2023

This report is wholly the work of the authors, except where explicitly stated otherwise.
The source of any material which was not created by the authors has been clearly cited.

Date: 19/03/2023

Signatures:

A handwritten signature in black ink, featuring a large, stylized 'S' or 'Z' shape with a long horizontal stroke extending to the right.A handwritten signature in black ink, consisting of a large, circular loop with a horizontal stroke crossing through it.A handwritten signature in black ink, appearing to be 'T. S. W.' with a horizontal line underneath.

Note: Much of the mathematical formulations and derivations in chapters 3-5 are based on / adapted from those presented in chapters 2-6 of the text by Richard S. Sutton and Andrew G. Barto [1].

Contents

1	Introduction	4
2	Machine Learning Paradigms	5
2.1	Supervised Learning	5
2.2	Unsupervised Learning	5
2.3	Reinforcement Learning	5
3	Markov Decision Processes	6
3.1	A Motivating Example: Frozen Lake	6
3.2	Framework: Agent, Environment, Actions, Rewards	6
3.2.1	Frozen Lake Example: Revisited	7
3.3	Policies, Value Functions & Optimisation	8
3.3.1	Policies	9
3.3.2	Value Functions	9
3.3.3	Optimality	12
4	Dynamic Programming	14
4.1	Policy Iteration	14
4.2	Value Iteration	16
5	Temporal Difference Learning	17
5.1	On-Policy Control: SARSA	20
5.2	Off-Policy Control: Q-Learning	20
6	Implementing Q-Learning in Python	22
6.1	Tools and Packages	22
6.2	Solving Frozen Lake with Q-Learning	23
6.2.1	FrozenLake-v1	23
6.2.2	Tabular Q-Learning Approach	23
6.2.3	Results and Performance	24
7	An Introduction to Deep Q Networks	26
7.1	Overview: What is a Deep-Q-Network?	26
7.1.1	Experience and Replay Memory	26
7.2	Solving CartPole with a Deep Q Network	27
7.2.1	Additional Tools and Packages	27
7.2.2	CartPole-v1	27
7.2.3	Learning Algorithm	28

7.2.4	Results and Performance	29
8	Reinforcement Learning in the Real World	31
8.1	Strengths & Weaknesses	31
8.1.1	Highlights	31
8.1.2	Challenges	31
8.2	Applications	32
8.3	The Future	32
8.3.1	Multi-agent systems	32
8.3.2	Acceleration of the Training Process	32
9	Conclusions	34
A	Source Code	37
A.1	Setup and Video Wrapper	37
A.2	Q-Learning	38
A.3	Deep Q-Learning	41

Chapter 1

Introduction

In this report, we investigate the reinforcement learning model; a method of machine learning that resembles the experience of all living things. That is, learning by interaction with ones surroundings. Actions have consequences, and so our goal is to implement a method of machine learning based on this idea.

Reinforcement learning has its origins in the psychology of animal learning. Reinforcement is the act of making something stronger, in this case, a desired behaviour. In order to implement this type of trial-and-error learning in an artificial environment, the reinforcement learning method is modelled with Markov Decision Processes (MDPs). The structure of this method and all of its components will be explored in detail. We will briefly review and compare alternative methods of machine learning, highlighting the strengths and weaknesses of each.

Reinforcement learning plays a pivotal role in today's society with many applications that we will discuss. We will also follow through the implementation process of this method as we implement the Q-learning algorithm in Python, and look at more complex systems such as Deep-Q-Networks.

Chapter 2

Machine Learning Paradigms

2.1 Supervised Learning

Supervised Learning is task-driven. In the training process, fully labelled sets of data are required and a desired output to train up to is given by the supervisor. It is used to classify data and predict outcomes with applications including object detection and text recognition.

2.2 Unsupervised Learning

In contrast with supervised learning, unsupervised learning is data-driven. Unlabelled sets of data are used for the training and desired outputs are not given, but rather one tries to find hidden structures within the data. It can be used for anomaly detection and recommendation systems. For example, self-organizing maps use unsupervised learning for the training process.

2.3 Reinforcement Learning

Reinforcement learning (RL) is a completely different approach altogether; it is goal-driven. We are presented with a problem, a task, a challenge. Solving this problem involves understanding the issue itself, as well as the context, exploring different options, and mapping actions to reactions in order to maximise success and rewards and achieve long-term goals.

Just like we would struggle to find our way around a city we have never visited, artificial intelligence will struggle when placed in a new environment, until it learns as the training process progresses. The reinforcement learning algorithm utilises Markov Decision Processes, which we will discuss in detail in the next chapter.

Chapter 3

Markov Decision Processes

3.1 A Motivating Example: Frozen Lake

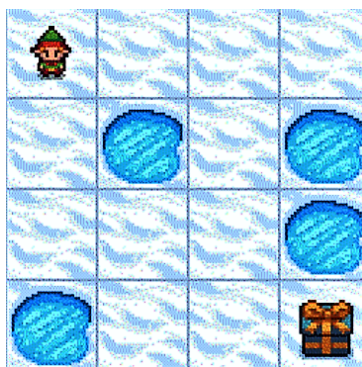


Figure 3.1: Initial state of the "Frozen Lake" motivating example [A.2]

The figure above shows what the “Frozen Lake” game initially looks like. In this specific example, the little elf is given the task or challenge of reaching the present.

How can this be achieved? Firstly, the elf must interact with its surroundings. In this set up, the squares filled with snow are considered safe spaces, while the squares with holes in the ice layer are considered dangerous, as the elf may fall into them. While exploring the different actions available to him, the elf must learn from his past experiences and find his way to the goal: the present.

3.2 Framework: Agent, Environment, Actions, Rewards

Motivated by the concepts presented in the example above, we now seek a general mathematical framework for our reinforcement learning problems. This will make clear the general objectives of any reinforcement learning algorithm. Later, we will apply various methods of control to this framework to achieve the desired results, i.e., we want the elf to get to the present!

A Markov Decision Process, or MDP, is precisely this mathematical framework underpinning the majority of reinforcement learning problems. It is a discrete-time stochas-

tic (random) control process. An MDP has four basic components:

- **Agent:** The entity over which we have full control. It is the decision-maker in the problem.
- **Environment:** As the name suggests, this is the space in which the problem takes place. The agent resides in the environment. It is a state-space \mathcal{S} , consisting of all possible states $s \in \mathcal{S}$.
- **Actions:** These are the decisions available to the agent. The *Action Space* \mathcal{A} is the state-space of all actions available to the agent in state s . It is typically state-dependent i.e., $a \in \mathcal{A}(s)$.
- **Rewards:** These are numerical values given to the agent when transitioning between states. Positive numbers typically indicate good rewards and negative numbers bad rewards, with the magnitude of the number determining the magnitude of the reward. The rewards obtainable in a certain state are typically state- and action-dependent, i.e., $r \in \mathcal{R}(s, a)$.

Note: Transitions between states are characterised by *transition probabilities* $p(s', r|s, a) \in \mathcal{P}(s, a)$ which depend on the current state s and action a taken by the agent. $p(s', r|s, a)$ denotes the probability of the agent transitioning to state s' and receiving reward r given that it is in state s and has taken action a .

With this, we are now ready to give the formal definition of an MDP, which will be the basis for tackling reinforcement learning problems:

Definition: A *Markov Decision Process* is a 4-tuple $(\mathcal{S}, \mathcal{A}(s), \mathcal{P}(s, a), \mathcal{R}(s, a))$ where \mathcal{S} is the State Space, $\mathcal{A}(s)$ the Action Spaces, $\mathcal{P}(s, a)$ the Transition Probability Distributions and $\mathcal{R}(s, a)$ the Rewards.

Note: We implicitly assume \mathcal{S} to be finite. More rigorously then, we are dealing with *finite* MDPs, the solving of which is computationally much simpler than the infinite case. Infinite, or *continuous* MDPs are dealt with briefly in chapter 7.

MDPs satisfy the *Markov Property*, the property of ‘memorylessness’, wherein the state of a stochastic process at time $t + 1$ depends only upon the present state at time t , i.e., the future is independent of the past. This can be seen in the transition probabilities as they depend only upon the current state-action pair (s, a) .

Having this property drastically simplifies the analysis of MDPs. Needless to say, it is an idealisation of reality, but many systems come *close* to being Markov, making it a useful idealisation.

3.2.1 Frozen Lake Example: Revisited

After defining the various components of an MDP, we can look back at our Frozen Lake example to better understand and visualise these concepts.

In this specific example, we have the following elements:

- **Agent:** Elf
- **Environment:** Frozen lake with holes and present

- **Actions:** Up, Down, Left, Right
- **Rewards:**
 - 0 for moving into a safe space
 - -1 for falling into a hole
 - +1 for reaching the present
- **States:** Positions where the elf can be. 4×4 grid \implies 16 states. Each time the elf takes an action and gets rewarded accordingly, he enters a new state

Take for example the initial state at the top left-hand corner (see 3.1). In this state, the actions available to the agent are to go *down* or to go *right*. If the agent chooses to go *right*, it would get rewarded 0 points for moving into a safe space, and would enter a new state.

In this new state, the actions available to the agent are to go *left*, *right* or *down*. If the agent chooses to go *down*, the elf would fall into the hole and be punished by losing 1 point. This process will continue and the agent will eventually find the best decisions or, more formally, the optimal *policy* (defined in section 3.3) to maximize the rewards.

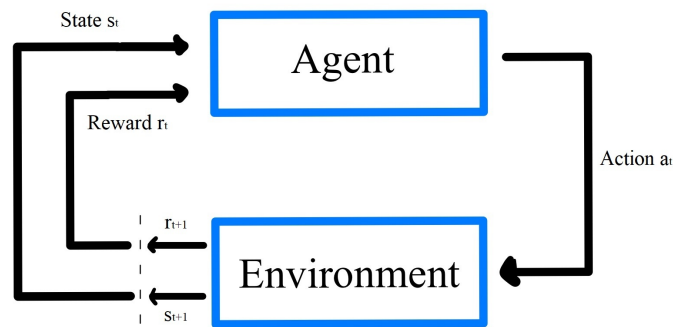


Figure 3.2: Diagram of the Reinforcement Learning Model

In conclusion, the reinforcement learning model, as illustrated above, consists of an agent placed in an environment, taking actions in said environment. Every time the agent takes an action, it is rewarded accordingly, enters a new state and the process repeats.

3.3 Policies, Value Functions & Optimisation

Having laid the foundations with MDPs, we now build upon this. MDPs alone provide information about the environment; we know what states, actions and rewards are possible. However, we know nothing about what states or actions are *favourable*, or how the agent even *decides* what actions to take. We begin with the latter issue which will be useful in tackling the former.

3.3.1 Policies

Definition: A *policy* π is a function that maps state-action pairs (s, a) to probabilities. The probability of choosing action a in state s is denoted $\pi(a|s)$. Thus, for each $s \in \mathcal{S}$, π is a normalised probability distribution over all $a \in \mathcal{A}(s)$.

$$\begin{aligned}\pi : \mathcal{S} \times \mathcal{A} &\rightarrow \mathbb{R} \\ (s, a) &\mapsto \pi(a|s)\end{aligned}$$

Thus, agents make decisions via policies. We say an agent *follows* a given policy. It is the goal of a reinforcement learning algorithm to find an *optimal policy*, i.e., one which favours actions that yield optimal *values*.

3.3.2 Value Functions

What does it mean for something to be *valuable* in the context of reinforcement learning? We answer this question with the notion of *return*:

Definition: The *discounted return* for an agent at time t in an MDP is the discounted sum of future rewards:

$$g_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad (3.1)$$

where $\gamma \in [0, 1]$ is the *discount factor*. It plays a crucial role in how the agent values short-term vs long-term rewards:

$$\begin{aligned}\gamma \rightarrow 0 &\implies \text{long-term rewards negligible} \\ \gamma \rightarrow 1 &\implies \text{long-term rewards considerably important}\end{aligned}$$

g_t gives us a notion of *value*. The higher the value of g_t , the more valuable the state at time t . Note also the recursive property which will be useful later:

$$\begin{aligned}g_t &= r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \\ &= r_{t+1} + \gamma g_{t+1}\end{aligned} \quad (3.2)$$

Now that we have a measure of value in an MDP, we return to the topic of value functions, of which there are two types:

- State-Value Functions
- Action-Value Functions

Definition: A *state-value function* $v_\pi(s)$, or ‘V-function’, yields the ‘value’ of a given state s under policy π , where value is defined as the expectation of the discounted return from state s following policy π :

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi [g_t | s_t = s] \\
&= \mathbb{E}_\pi [r_{t+1} + \gamma g_{t+1} | s_t = s] \\
&= \mathbb{E}_\pi [r_{t+1} | s_t = s] + \gamma \mathbb{E}_\pi [g_{t+1} | s_t = s] \\
&= \sum_{r \in R(s)} p(r|s)r + \gamma \sum_{s' \in S} p(s'|s) \mathbb{E}_\pi [g_{t+1} | s_{t+1} = s'] \\
&= \sum_{r \in R(s)} p(r|s)r + \gamma \sum_{s' \in S} p(s'|s) v_\pi(s') \tag{3.3}
\end{aligned}$$

Note that $p(r|s)$ is the probability of getting reward r from state s . How could this be achieved? We could take an action $a \in A(s)$, undergo a transition to state $s' \in S$, and receive reward $r \in R$. The probability of this sequence of events is given by:

$$p(s', r, a | s)$$

But there may be multiple states $s' \in S$ with reward r to which we can transition after action a is performed in s . Furthermore, there may be multiple actions $a \in A(s)$ leading to s' , each of which having a probability $\pi(a|s)$ of being selected. Thus, the transition probability $p(r|s)$ can be expressed as:

$$\begin{aligned}
p(r|s) &= \sum_{s' \in S} \sum_{a \in A(s)} p(s', r, a | s) \\
&= \sum_{s' \in S} \sum_{a \in A(s)} \pi(a|s) p(s', r | s, a)
\end{aligned}$$

The exact same analysis can be applied to $p(s'|s)$ to obtain:

$$p(s'|s) = \sum_{r \in R(s)} \sum_{a \in A(s)} \pi(a|s) p(s', r | s, a)$$

Plug these expressions into (3.3) and we arrive at the *Bellman Equation* for the state-value function:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')] \tag{3.4}$$

We have just shown something very powerful. Remember, the value of state s under policy π was defined as the *expected discounted return* from s following π thereafter. But

through the use of (3.2) in deriving (3.3) we have shown that $v_\pi(s)$ can be expressed as the expected *immediate reward*:

$$\sum_{r \in R(s)} rp(r|s)$$

Plus the discounted expected value of *next state* s' :

$$\gamma \sum_{s' \in S} p(s'|s)v_\pi(s')$$

Thus, we have decomposed a problem requiring the calculation of the expectation of a potentially infinite series into a *recursive* problem with finite terms, i.e., a sum over the potential rewards attainable from state s and a sum over potential successor states s' . It is recursive since the calculation of $v_\pi(s)$ requires knowledge of $v_\pi(s')$, $\forall s' \in S$ which, in turn, require knowledge of $v_\pi(s)$.

It may seem as though all we have done is move the problem of infinite series' to the problem of infinite iterations of v_π over all states, but it is exactly these kinds of iterative processes which computer algorithms are designed to handle, needing much less computational power and storage than large, isolated calculations.

The purpose of continuing from (3.3) to (3.4), apart from compactness, was to separate the action probabilities $\pi(a|s)$ from all other terms. These terms:

$$\sum_{s',r} p(s',r|s,a) [r + \gamma v_\pi(s')] \quad (3.5)$$

are the individual contributions of each potential action a in state s to $v_\pi(s)$. Separating the solution in this form will prove extremely useful when optimising the value function later and also motivates the definition of another type of value function. Similar to the state-value function is the *action-value function* q_π :

Definition: An *action-value function* $q_\pi(s,a)$, or 'Q-function', yields the value of a given state-action pair (s,a) under policy π :

$$q_\pi(s,a) = \mathbb{E}_\pi [g_t | s_t = s, a_t = a] \quad (3.6)$$

The only difference between v_π and q_π is that the action a is fixed in q_π . Why define this new function with such a subtle difference? Well, $v_\pi(s)$ yields the value of state s under policy π , with every action from state s onward governed by π , whereas $q_\pi(s,a)$ yields the value of taking action a in state s , with every action *thereafter* governed by π . Knowing the values of specific actions in every state is more useful than just knowing the value of a state, as we will see.

p_π and q_π are related mathematically as follows:

$$v_\pi(s) = \sum_{a \in A(s)} \pi(a|s)q_\pi(s,a) \quad (3.7)$$

Put simply, $v_\pi(s)$ is the sum of the values $q_\pi(s, a)$ of action a in state s , weighted by the probability of taking action a in s under π . Using (3.7), it is clear that the Bellman equation (3.4) for $q_\pi(s)$ is just (3.5):

$$\begin{aligned} q_\pi(s, a) &= \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')] \\ &= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \sum_{a' \in A(s)} \pi(a' | s') q_\pi(s', a') \right] \end{aligned} \quad (3.8)$$

3.3.3 Optimality

As mentioned, the goal of reinforcement learning is to find an optimal policy for the agent to follow. A policy π is optimal if:

$$v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in S \quad \forall \pi'$$

where π' is some other policy. That is, the value of every state under π is at least equal to the its value under π' . An identical definition can be given in terms of action-values. Note that while optimal value functions are unique (cannot have different optimal values for the same state), optimal policies may not be. We usually denote an optimal policy as $*$.

Associated with an optimal policy are the optimal state- and action-value functions:

$$\begin{aligned} v_*(s) &= \max_{\pi} v_\pi(s) \\ q_*(s) &= \max_{\pi} q_\pi(s, a) \end{aligned}$$

Looking at the Bellman equation (3.4), we can write the optimal state-value function as:

$$\begin{aligned} v_*(s) &= \max_{\pi} \sum_a \pi(a | s) \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')] \\ &= \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_*(s')] \end{aligned} \quad (3.9)$$

Similarly for $q_*(s, a)$ using (3.8):

$$\begin{aligned} q_*(s, a) &= \max_{\pi} \sum_{s', r} p(s', r | s, a) \left[r + \gamma \sum_{a' \in A(s)} \pi(a' | s') q_\pi(s', a') \right] \\ &= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \max_{a'} q_*(s', a') \right] \end{aligned} \quad (3.10)$$

These are known as the *Bellman Optimality Equations*. The primary objective of reinforcement learning algorithms is to obtain the optimal values for each state or state-action pair through iterative applications of (3.8) which will approach (3.10) in the infinite limit (see *policy iteration*), or simply through direct application of (3.10) (see *value iteration*). Both methods are central to reinforcement learning, as we will see.

Chapter 4

Dynamic Programming

Dynamic programming (DP) is a technique of dividing an algorithmic problem into sub-problems, which can be solved iteratively. DP methods assume an ideal model of the environment, i.e., everything is known *a priori*; the states, actions, rewards and transition probabilities are all provided. This is highly unlikely in reality as environments are usually unknown territories that must be explored by the agent to learn about their nature. However, working in an ideal setting first allows us to learn a great deal about the general procedure before getting lost in the weeds of reality. Remarkably, the process is much the same for the real-life scenario of unknown environments.

Previously, we noted that the objective of reinforcement learning is to obtain optimal state-action values using the Bellman Equations (3.8, 3.10) which then determine the optimal policy. This general procedure is referred to as *policy iteration*.

4.1 Policy Iteration

We now turn to the problem of finding the optimal value functions v_* and q_* . We will focus on q_* as it will prove more useful. However, the theory is much the same for v_* .

In DP, there are two main techniques for obtaining q_* - policy iteration and value iteration, the latter being a particular case of the former.

Policy iteration is a repeated two-step process:

- Policy evaluation
- Policy improvement

We begin by initialising an arbitrary policy π and arbitrary Q -values $q_0(s, a)$ for each state-action pair, typically zero. We then *evaluate* the policy, i.e., we update $q_\pi(s, a)$ for all state-action pairs. This is done iteratively using the Bellman Equation (3.8), where the n^{th} approximation of the Q -value for each state-action pair is built from the Q -values of potential successor states in the $(n - 1)^{\text{th}}$ approximation:

Policy Evaluation

$$q_{\pi}^n(s, a) = \sum_{s', r} p(s', r | s, a) \left[r + \gamma \sum_{a' \in A(s)} \pi(a' | s') q_{\pi}^{n-1}(s', a') \right] \quad (4.1)$$

It can be shown that (4.1) always converges to $q_{\pi}(s, a)$ provided it exists. Policy evaluation can be repeated as many times as is necessary to reach satisfactory convergence, where convergence is determined by the difference between successive approximations of q_{π} .

Once we have approximated the current policy, the next step is to *improve* it. This is done by updating the policy to choose actions that maximise $q_{\pi}^n(s, a)$. In (4.1), the only place the policy plays a role is in the term:

$$\sum_{a \in A(s)} \pi(a | s) q_{\pi}^{n-1}(s, a)$$

Note: Dropped primes (') in sum above since irrelevant here

which is maximised by having the updated policy π' choose the action a which corresponds to the maximum $q_{\pi}^{n-1}(s, a)$:

Policy Improvement

$$\pi'(a | s) = \begin{cases} 1 & \text{if } a = \arg \max_a q_{\pi}^n(s, a) \\ 0 & \text{o.w} \end{cases} \quad (4.2)$$

The improved policy is deterministic; we know exactly which action is taken in which state. We call it a *greedy* policy with respect to the q_{π}^n , choosing actions which correspond to the highest values.

This two-step procedure is repeated until the policy converges to an optimal policy $*$. This happens when $\pi'(a | s) = \pi(a | s)$ for all state-action pairs, i.e., policy unchanged between consecutive improvements.

Note: The policy may oscillate between two optimal policies and thus never converge, but a solution has still been found.

Once the policy converges, $\pi'(a | s) = \pi(a | s)$ and thus $q_{\pi'}(s, a) = q_{\pi}(s, a)$, meaning:

$$\begin{aligned}
q_{\pi'}(s, a) &= q_{\pi}(s, a) \\
&= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \sum_{a' \in A(s)} \pi(a' | s') q_{\pi}(s', a') \right] \\
&= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \sum_{a' \in A(s)} \pi'(a' | s') q_{\pi}(s', a') \right] \\
&= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \max_{a'} q_{\pi}(s', a') \right] \\
&= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \max_{a'} q_{\pi'}(s', a') \right]
\end{aligned}$$

which is nothing but the Bellman Optimality Equation (3.10)! There was some nuance with primed variables here but the key point is that policy iteration, repeatedly evaluating (which itself is an iterative process!) and improving the policy π , leads to the Bellman Optimality Equation and, thus, an optimal Q-function and policy.

4.2 Value Iteration

As discussed, policy evaluation is an iterative sub-process within policy iteration. However, it is not essential to repeat the evaluation process up to high accuracy for every intermediate policy. After all, the tolerance is decided by us. What if we just carried out a single evaluation step before returning to policy improvement? This method still leads to overall convergence to the optimal policy and value function and is the approach taken by *value iteration*. It is simply a particular case of policy iteration. The single iteration evaluation process is as follows:

Policy Evaluation

$$q_n = \sum_{s', r} p(s', r | s, a) \left[r + \gamma \max_{a'} q_{n-1}(s', a') \right] \quad (4.3)$$

Policy improvement is then carried out as before. It is clear that (4.3) is just the Bellman Optimality Equation (3.10) converted to an update rule. Notice that (4.3) is policy independent. This will be important later when we talk about Q-Learning.

Policy and value iteration are used extensively in reinforcement learning algorithms, as we will see next, as both are extraordinarily powerful tools. However, earlier we discussed the fact that DP handles only the ideal case where everything about our environment is already known. We are yet to face the problem of unknown environments. Luckily, much of the machinery that we have developed here carries over rather seamlessly to more realistic scenarios, with some embellishments to our formulae being required.

Chapter 5

Temporal Difference Learning

It is now time to tackle realistic problems. The issues with the methods derived in DP reside almost entirely in the policy evaluation step, as it is here that we require knowledge of the transition probabilities associated with the environment, which we now assume to be unknown.

This is where we introduce *temporal difference* (TD) learning, one of the most fundamental concepts in reinforcement learning. It is the collection of methods that allows us to efficiently handle real-world problems, with unknown environment dynamics. As the name suggests, solutions are learned based on values at different time steps in the simulation process, or episodes.

What do we mean by ‘simulation process’ or ‘episode’? Since we are dealing with an unknown environment, in order to learn about its dynamics we must *explore* the environment which will later allow us to *exploit* what we have learned to our advantage. ‘Simulation’ is the process of the agent traversing the environment, transitioning between states, taking actions and receiving rewards. ‘Episodes’ are finite-time simulations, with some initial and final states. Many real-life problems can be broken into episodes. For example, in the Frozen Lake game, each run of the game is an episode.

Note: Some problems are non-episodic, or ‘continuing’, meaning there is no terminal state. However, the analysis is much the same. Thus, we shall assume episodic behaviour in our problems.

So TD algorithms require simulation of the environment to learn optimal value functions and policies. But what exactly are these algorithms and how do we deal with the ‘black box’ of environment dynamics?

Returning to (3.6), restated below, we remember that the definition of the action-value function is the average of the discounted returns from the state-action pair (s, a) .

$$q_{\pi}(s_t, a_t) = \mathbb{E}_{\pi} [g_t] \tag{5.1}$$

So, say we have just finished the k^{th} episode of our simulation, we could then approximate the Q-value of each state-action pair as follows:

$$Q_{\pi}^{k+1}(s_t, a_t) = \frac{1}{k} \sum_{i=1}^k g_t^i \quad (5.2)$$

Note: We switch $q_{\pi} \rightarrow Q_{\pi}$ to distinguish between DP and TD, i.e., known vs. unknown dynamics.

So the $(k+1)^{\text{th}}$ approximation of Q_{π} is the average of the returns g_t following (s_t, a_t) in the k previous episodes. We can then make the evaluation recursive by rewriting:

$$\begin{aligned} Q_{\pi}^{k+1}(s_t, a_t) &= \frac{1}{k} \left(g_t^k + (k-1) \left[\frac{1}{k-1} \sum_{i=1}^{k-1} g_t^i \right] \right) \\ &= \frac{1}{k} \left(g_t^k + (k-1) Q_{\pi}^k(s_t, a_t) \right) \\ &= Q_{\pi}^k(s_t, a_t) + \frac{1}{k} \left(g_t^k - Q_{\pi}^k(s_t, a_t) \right) \end{aligned}$$

and make the simplifying assumption:

$$\frac{1}{k} = \text{const.} = \alpha$$

For the first few episodes this is not even close to true, but as the number of episodes increases it becomes valid. This assumption affects only convergence *rates*, not overall convergence to q_{π}

We then have the following update rule for our Q -value estimates:

$$Q_{\pi}^{k+1}(s_t, a_t) = Q_{\pi}^k(s_t, a_t) + \alpha \left(g_t^k - Q_{\pi}^k(s_t, a_t) \right) \quad (5.3)$$

Great, we have avoided the need for knowledge of environment dynamics to approximate Q -values. However, we have the slight inconvenience of needing to wait an entire episode between updates of Q_{π} since only then do we update g_t^k . The current update rule is that of *Monte Carlo* learning. But we can do better! We know from (3.6) and (3.8) that:

$$q_{\pi}(s_t, a_t) = \mathbb{E}_{\pi} [g_t] = \mathbb{E}_{\pi} [r_{t+1} + \gamma q_{\pi}(s_{t+1}, a_{t+1})] \quad (5.4)$$

The key difference here is that g_t is calculated from the whole episode whereas $r_{t+1} + \gamma q_{\pi}(s_{t+1}, a_{t+1})$ is calculated from the following time step. We can thus make the replacement:

$$g_t^k \rightarrow r_{t+1} + \gamma Q_\pi^k(s_{t+1}, a_{t+1})$$

$$\implies Q_\pi^{k+1}(s_t, a_t) = Q_\pi^k(s_t, a_t) + \alpha \left(r_{t+1} + \gamma Q_\pi^k(s_{t+1}, a_{t+1}) - Q_\pi^k(s_t, a_t) \right)$$

Thus, estimates of Q_π at time t are governed by previous estimates of Q_π at time $t + 1$ - temporal difference learning! Note that k is now incremented at each time step rather than after each episode. The TD update rule can be written compactly:

$$Q_\pi(s_t, a_t) \leftarrow (1 - \alpha) Q_\pi(s_t, a_t) + \alpha (r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1})) \quad (5.5)$$

Here, \leftarrow indicates that new Q -values (RHS) are updated from old values (LHS) according to the given rule.

We have conquered unknown environment dynamics and now have an algorithm to evaluate policies, as in the last chapter, except now our algorithm is independent of transition probabilities. Comparing (5.5) with (4.1) we see that both algorithms are similar.

The parameter α can be thought of as a *learning rate*, $0 \leq \alpha \leq 1$, with $\alpha = 0$ indicating that new Q -values are unchanged (no learning) and $\alpha = 1$ indicating that new Q -values are independent of previous estimates. Clearly, a value in between is optimal. The particular value is problem-specific and must be determined heuristically.

We have essentially solved the problem, there is just one issue remaining, which lies in policy improvement. In DP, the improved policy was simply the greedy policy for the given Q -values. Since we knew all state-action pairs, these Q -values were all continually updated regardless of the policy followed, i.e., we didn't have to be *in* a state to evaluate it. In fact, we never simulated anything, so we never *were* in any states.

However, if we take the same approach in TD learning, then there are particular state-action pairs whose Q -values will never be updated, since the policy will not allow us to visit them. But we are now *required* to visit a state-action pair to update its value.

Hence, we don't simply want to update the policy to be greedy, but rather to *tend* towards greediness. We call such a policy an ϵ -greedy policy. It is defined as follows:

$$\pi(a|s) = \begin{cases} \frac{1}{|\mathcal{A}(s)|} & \text{with probability } \epsilon \\ \delta(a, a_{max}) & \text{with probability } 1 - \epsilon \end{cases} \quad (5.6)$$

where $\delta(a, a_{max})$ is compact notation for (4.2). The parameter ϵ is the *greedy parameter*, with $0 \leq \epsilon \leq 1$ where $\epsilon = 0$ indicates a greedy policy and $\epsilon = 1$ a random policy.

Such a policy allows for a trade-off between *exploration* and *exploitation* as discussed previously. The general policy improvement procedure for TD learning is to begin

with $\epsilon = 1$ and to slowly decrement it between successive episodes, allowing the agent to begin by exploring the environment and thus learning while slowly tending towards the exploitation of its knowledge of optimal Q -values, in the limit converging to the optimal policy.

5.1 On-Policy Control: SARSA

Finally, we briefly discuss the types of TD control, the first being SARSA, an acronym which will become apparent shortly. SARSA is nothing more than policy iteration with the new policy evaluation and improvement steps derived above used in place of those in the last chapter. It is therefore a policy-dependent or *on-policy* method. The pseudo code for the SARSA algorithm is described below:

SARSA Algorithm

- Initialise $Q_\pi(s, a) = 0$ for all state-action pairs
- Set $\epsilon = 1$
- Initialise $\alpha \in [0, 1]$
- For each episode:
 - Choose initial state $s_0 \in \mathcal{S}$
 - For each time step t in episode:
 - * Take action a_t according to ϵ -greedy policy π
 - * Receive reward r_{t+1} , transition to state s_{t+1} .
 - * Take action a_{t+1} according to π
 - * $\underbrace{\text{Update } Q_\pi(s_t, a_t) \text{ according to (5.5)}}_{\text{Policy Evaluation}}$
 - $\underbrace{\text{Decrement } \epsilon \text{ by some small amount}}_{\text{Policy Improvement}}$

This process of starting in a state s_t , taking action a_t , receiving reward r_{t+1} , transitioning to state s_{t+1} and taking action a_{t+1} before updating $Q_\pi(s_t, a_t)$ is where the acronym SARSA finds its origin. SARSA is guaranteed to converge to the optimal Q -function q_* and optimal policy $*$ in the ideal régime of each state-action pair being visited an infinite number of times and provided ϵ tends to zero in the limit of infinite episodes.

5.2 Off-Policy Control: Q-Learning

As one may guess, following the pattern of the previous chapter, Q-Learning is simply *value* iteration with the new policy evaluation and improvement methods replacing

those of the last chapter. Since value iteration is policy independent, Q-Learning is an *off-policy* method. The pseudo code is shown below:

Q-Learning Algorithm

- Initialise $Q_\pi(s, a) = 0$ for all state-action pairs
- Set $\epsilon = 1$
- Initialise $\alpha \in [0, 1]$
- For each episode:
 - Choose initial state $s_0 \in \mathcal{S}$
 - For each time step t in episode:
 - * Take action a_t according to ϵ -greedy policy π
 - * Receive reward r_{t+1} , transition to state s_{t+1} .
 - * Update $Q_\pi(s_t, a_t)$ according to:

$$\underbrace{Q_\pi(s_t, a_t) \leftarrow (1 - \alpha) Q_\pi(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_{a'} Q_\pi(s_{t+1}, a') \right)}_{\text{Policy Evaluation}} \quad (5.7)$$

- $\underbrace{\text{Decrement } \epsilon \text{ by some small amount}}_{\text{Policy Improvement}}$

Comparing (5.7) with (5.5) and (4.3), we see that it is just a combination of both, as we would expect. Note that Q-Learning does not require a ‘next action’ a_{t+1} to be taken in order to calculate $Q_\pi(s_t, a_t)$ (it is policy-independent). This is why a' is used in the algorithm, to represent *possible* next actions.

Since Q-Learning is policy independent, it is not actually essential for the policy to be ϵ -greedy. A random policy would still lead to convergence of Q to q_* in the limit of infinite state-action visits, from which the optimal policy could easily be extracted. However, having an ϵ -greedy policy speeds up convergence so is usually preferable.

We have looked at the main theory governing reinforcement learning problems and the approaches used to solve them - dynamic programming and temporal difference learning. In the following chapter, we will see the Q-Learning algorithm implemented directly in the Frozen Lake environment where (5.7) will play a central role.

Chapter 6

Implementing Q-Learning in Python

6.1 Tools and Packages

Python is, without doubt, the most popular programming language for machine learning at the time of writing. Being a high-level language (code and syntax are easily understood and read by humans), Python naturally is very popular and as such has many excellent community-developed packages for high-level computing. The fundamental packages used for developing the implementations showcased in this report are as follows:

‘Gym’ by OpenAI: Gym is a package for developing and testing learning agents. Gym acts as the communication bridge between the programmer and the environment. A number of environments, designed specifically to test, develop and showcase learning algorithms, come pre installed with the package.

‘NumPy’: NumPy provides the programmer with an extended library of functions for creating and manipulating arrays. Packages such as NumPy are considered fundamental in scientific computing as NumPy arrays are defined to be fixed in size, unlike Python’s ‘lists’ which can increase/decrease in size dynamically, often unintentionally.

‘PyVirtualDisplay’ and ‘IPython’: Due to a lack of support, Gym’s environment rendering does not run native on non Linux based systems. This required the use of a cloud based IDE (Google Colab), however similar issues were then posed in terms of visually observing the environment. **This was overcome by developing a video wrapper** using pre-defined Gym functions, PyVirtualDisplay, and IPython. It can record, store and display mp4 files of instances within the environment. Corresponding source code is in Appendix A.1.

‘Colaboratory’ by Google: The IDE used for implementation was Colaboratory, a Linux based hosted Jupyter Notebook service by Google. Colab is very well suited to machine learning, particularly for group-based projects as package versions remain consistent among systems and contributors.

6.2 Solving Frozen Lake with Q-Learning

Exemplified in section 3.1 as a motivating example of Markov Decision Processes, Frozen Lake is one of many environments to come pre-installed with Gym.

6.2.1 FrozenLake-v1

Action Space: Discrete (4)

- 0 : LEFT
- 1 : DOWN
- 2 : RIGHT
- 3 : UP

Rewards:

- Reached Goal: +1
- Reached a Frozen Surface: 0
- Entered Hole in Ice: 0

State Space: Discrete

Default 4x4 map: 16

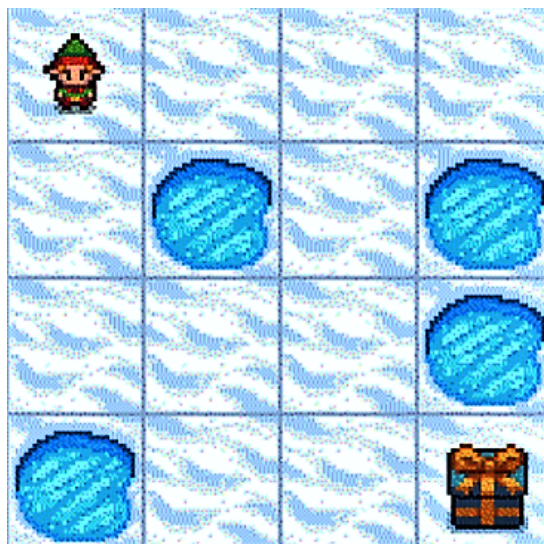


Figure 6.1: FrozenLake-v1 under default environment parameters [A.2]

The map size (and consequently state space) can be altered with the argument `map_name`. Custom built and randomly generated maps can be created and imported. The default argument (used throughout this section) is `map_name = "4x4"` which is shown in Figure 6.1 and has a state space of 16.

Slipperiness: The `is_slippery` argument (Boolean) adds inherent randomness to the actions of the agent. Specifically, if `is_slippery = True` the agent only has a 1/3 chance of moving in the direction specified and a 1/3 chance of moving in either of the two directions perpendicular (For example, if the agent moves RIGHT, there is a 1/3 chance the agent moves UP and 1/3 chance it moves DOWN).

The game ends when the agent enters an endgame state. Frozen Lake's endgame states are states that correspond to the goal or holes in the frozen surface of the lake.

6.2.2 Tabular Q-Learning Approach

In section 5.2 a recursive Bellman Equation (Eq. 5.7) for calculating the Q-Value of a state-action pair was determined. An equivalent form in simplified notation is shown in Eq. 6.1.

$$Q(s, a) = Q(s, a)(1 - \alpha) + \alpha(r + \gamma \max_{a'} Q(s', a')) \quad (6.1)$$

where s is the state of the system, a is the action taken, s' is the state as a result of a and r is the reward Eq. 6.1 is premise of the implemented algorithm. Due to the discrete nature of Frozen Lake, we can define a 'Q-Table' in which to store and calculate the Q-Values for every possible state-action pair as calculated by Eq. 6.1. On paper, this would look as follows in Table 6.1.

State	Q_{LEFT}	Q_{DOWN}	Q_{UP}	Q_{RIGHT}
1				
2				
3				
\vdots				
16				

Table 6.1: FrozenLake Q-Table

The algorithm follows an ‘epsilon greedy’ strategy. As discussed previously, this means that the algorithm initially prioritises exploration of the environment and gradually shifts its behaviour toward exploiting the information the agent already knows (in this case, exploiting information gathered and stored in the Q-table). The algorithm implemented is shown below (Algorithm 6.1).

Algorithm 6.1: Q-learning algorithm with epsilon greedy strategy

γ = discount factor

α = learning rate

N = number of episodes

T = max steps per episode

Initialise Q [action space size, state space size] to zeros

For $episode = 1, N$ **do**

 Determine state s_0

 Determine $\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \exp(-\epsilon_{decay} * episode)$

For $step = 1, T$ **do**

 Determine a random $x \in [0, 1]$

If $x > \epsilon$ **do**

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

Else select random action a

 Execute action a ; determine new state s' , reward r

 set $Q(s, a) = Q(s, a)(1 - \alpha) + \alpha(r + \gamma \max_{a'} Q(s', a'))$ (6.1)

$s = s'$

If episode is done **break**

End For

End For

6.2.3 Results and Performance

The results as visualised in Figure 6.2 indicate a functioning learning algorithm.

Observations: The algorithm has performed well, most notably adapting to the inherent randomness present in case (b) (where slipperiness was present during training). If visually observed (by running the code in A.2 and observing the video output) the agent in case (a) can be seen to rapidly determine an optimal path to the goal. Case (b) is incomparable in this regard, the agent can be seen to gradually adapt and prioritise the safest actions to maximise it’s chances of success. These two cases show in essence the strengths of the algorithm, however should the slipperiness condition, map size

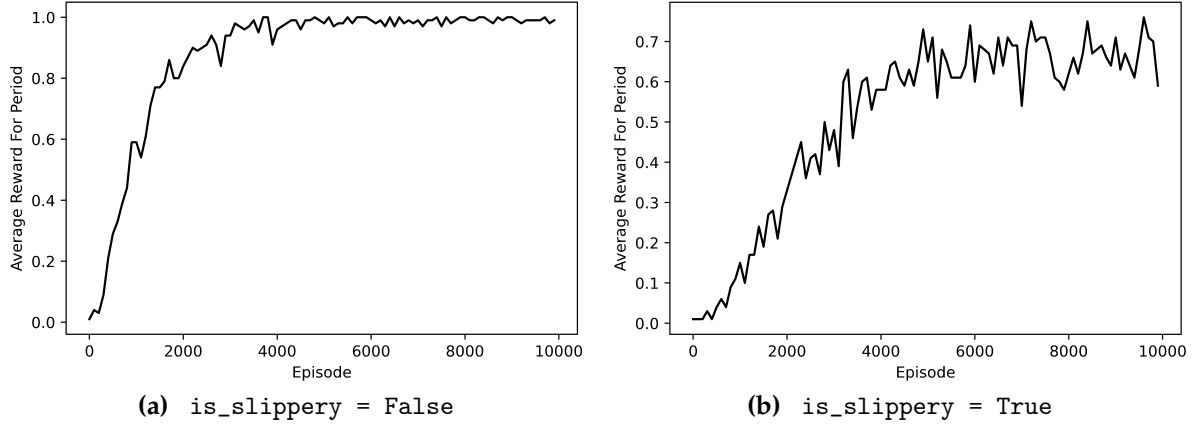


Figure 6.2: Results as produced from code in A.2. Sub-figures (a) and (b) represent independent runtimes of the algorithm learning to play ‘FrozenLake-v1’ under differing slipperiness parameters and identical hyper parameters, showcasing the mean reward over discretised periods of 100 episodes

or map layout be altered during training; the algorithm completely breaks down as the tabular method applies only to static discrete cases. Implemented source code, used to obtain these results (complete with video output and visualisation of Q-Table evolution), is provided in [A.2](#).

Chapter 7

An Introduction to Deep Q Networks

Traditional Q-Learning generally becomes obsolete in dynamic or continuous systems. For example, if the agent in Frozen Lake (the environment discussed and solved previously with a tabular approach) was not contained to just its 16 discrete states/tiles, say the elf could be placed at any point/pixel within or between the tiles; there is now a much larger state space, limited only by the resolution of observation. The computation required to tabulate a solution for a situation such as this is unlikely to be quantifiable. Environments with an infinite number of states are labelled continuous. Dynamic systems are another example, these are generally games/environments that incorporate any kind of physics engine to determine the current state of the system.

In this chapter we will generally discuss Deep Q-Learning and implement a deep learning model capable of solving the famous 'Cart-Pole' problem by OpenAI.

7.1 Overview: What is a Deep-Q-Network?

A deep Q-network (or DQN) is an artificial neural network that employs the conceptual ideas of traditional Q-learning. However, they are capable of adapting to dynamic and continuous environments. DQNs can interpret the state of the environment and approximate an action-value function (thus capable of approximating the Q-value for a state-action pair).

The objective of a neural network is to minimise loss by updating weights of the network. Loss, in a DQN, can be obtained by setting realistic targets for the network to reach or by predicting successive Q-values for a particular state and then comparing the networks actual performance against its targets. 'Experience Replay' is a common technique used to assess these.

7.1.1 Experience and Replay Memory

Experience Replay is a technique used in deep-learning primarily for training reinforcement algorithms. Replay Memory (or the "Replay Buffer") is a collection of 'experiences' the agent has experienced. This data set may be represented by a tuple, and would generally contain the state of the environment s_t , action a_t taken in state s_t , the reward r_{t+1} received from performing a_t and the resulting state s_{t+1} .

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1}) \quad (7.1)$$

Replay memory is referenced in random batches by the network when looking to approximate Q-values of the possible actions for a particular state (We reference in random batches so that we do not have consecutive correlations between experiences - preventing the network from "chasing it's own tail" if you will). This is demonstrated in detail in section 7.2.3.

7.2 Solving CartPole with a Deep Q Network

'CartPole' is a pre-installed Gym environment; a physics simulation of an inverted pendulum with it's fixed point of rotation attached to a 'cart' which the player/agent can control. The aim of CartPole is to keep the pendulum upright by moving the cart back and forth. This is a classic problem in robotics and control engineering (where servo-motor-systems monitor the pole angle and move the cart accordingly) translated into a deep-learning exercise.

7.2.1 Additional Tools and Packages

'PyTorch' by Meta AI: PyTorch was created and open-sourced by Meta AI (formerly Facebook) in 2016. It is a popular Machine Learning framework that provides a streamlined approach to creating artificial neural networks for a wide range of tasks. PyTorch uses Tensors as the core data-structure for communication.

Definition: A *Tensor (Machine Learning)* is a representation of high-dimensional data in a multidimensional array. Tensors are commonly used to represent the data of videos, images, and audio among a list of many possibilities.

PyTorch, in the context of this document, is used to create an artificial neural network capable of comprehending the state of a dynamic environment in order to compute an action-value function.

7.2.2 CartPole-v1

Action Space: Discrete (2)

0 : Move cart left

1 : Move cart right

State Space: Infinite*

CartPole-v1 has, effectively, an infinite number of states. In actuality this isn't necessarily true, however the environment is complex enough that the total number of possible states is not computationally plausible for tabular methods.

When the environment is observed, Gym returns an array of size 4 with the following information (Table 7.1):

Array Position	Observation	Minimum	Maximum
0	Cart Position	-4.8	4.8
1	Cart Velocity	$-\infty$	∞
2	Pole Angle	-0.418 rad	0.418 rad
3	Cart Angular Velocity	$-\infty$	∞

Table 7.1: Returned information from observation

Rewards:

Action Taken: +1

Episode Termination:

- Pole Angle = $\pm 0.2095 \text{ rad}$
- Cart Position = ± 2.4 (if cart is not visible on screen)

At every time-step an action must be taken under the default environment configurations. The agent is rewarded +1 for every action taken. **CartPole-v1 has a maximum reward of 500** per episode, which if reached abruptly ends the episode.

7.2.3 Learning Algorithm

Algorithm 7.1 is an adaptation of an algorithm showcased by DeepMind in their 2015 paper "*Human-level control through reinforcement learning*" [4]. The algorithm showcased was designed to adapt and learn multiple classic Atari games by approximating target Q-values using a convolutional neural network. The network interpretes frames from environment's visual output, determining the environment itself (Atari game being played) and current state of said environment. Targets are determined (as tensors) by a piece-wise function (Eq. 7.2)^[4] which are then used to calculate the networks loss. Eq. 7.2 is an equivalent form of the Bellman Optimality equation, a brief justification of which can be sourced from the paper itself.

$$Q_j^{\text{targets}} = \begin{cases} r_j & \text{if episode done at } j + 1 \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a', \phi) & \text{otherwise} \end{cases} \quad (7.2)$$

where j represents time step in experience, r_j is the reward at j , s_{j+1} is the state at $j+1$ and ϕ is network weights

The implementation shown in Algorithm 7.1 does not utilise a convolutional neural network to interpret the environments state or an independent target network as done in [4]. Rather, the returned observational data from Gym as listed in Table 7.1 is fed into a single network. This has draw backs, which will be discussed in section 7.2.4 when reviewing the models performance.

Algorithm 7.1: Deep Q-learning via epsilon greedy strategy with experience replay

γ = discount factor

N = number of episodes

T = max steps per episode

Initialise replay memory R to capacity N

Initialise action-value function Q with weights ϕ

For $\text{episode} = 1, N$ **do**

```

Determine state  $s_0$ 
For  $t = 1, T$  do
    Determine a random  $x \in [0, 1]$ 
    Determine  $\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \exp(-\epsilon_{decay} * step)$ 
    If  $x > \epsilon$  do
         $a_t = \max_{a'} Q(s_t, a', \phi)$ 
    Else select random action  $a$ 
    Execute action  $a_t$ ; determine new state  $s'$ , reward  $r$ 
    Store experience  $e_t = (s_t, a_t, r_t, s_{t+1})$  in  $R$ 
    Sample random batch of experience  $(s_j, a_j, r_j, s_{j+1})$  from  $R$ 
    Determine  $Q_j^{targets}$ :


$$Q_j^{targets} = \begin{cases} r_j & \text{if episode done at } j+1 \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a', \phi) & \text{otherwise} \end{cases} \quad (7.2)$$


    Compute loss  $y = (Q_j^{targets} - Q(s_j, a_j, \phi))^2$ 
    Perform gradient descent step on  $y$ 
    reset  $s_t = s_{t+1}$ 
     $step++ = 1$ 
    If episode is done break
End For
End For

```

7.2.4 Results and Performance

The results as visualised in Figure 7.1 indicate a functioning learning algorithm.

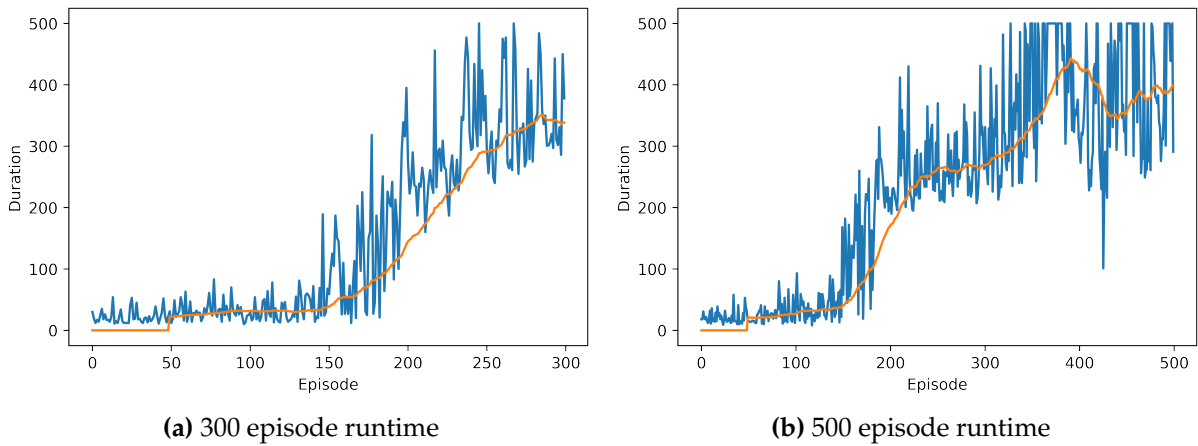
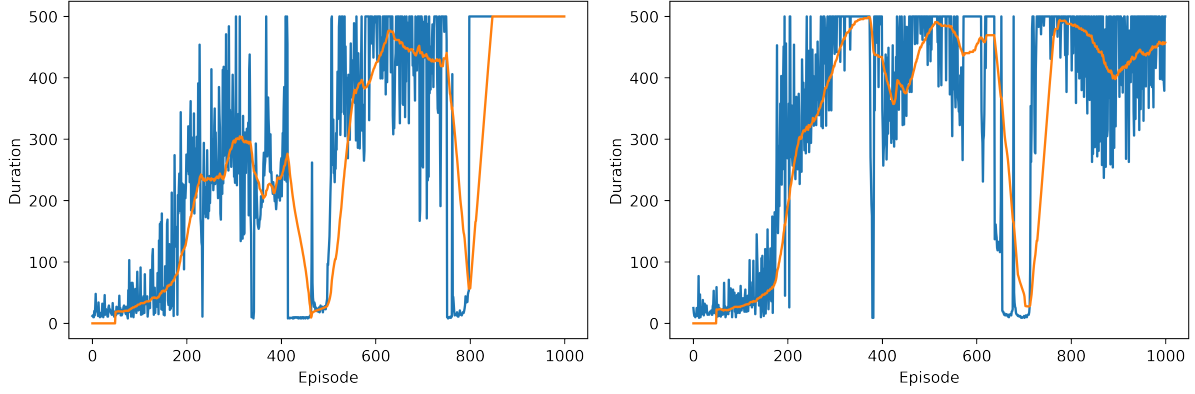


Figure 7.1: Results as produced from code in A.3. Sub-figures (a) and (b) represent independent runtimes of the algorithm, showcasing the duration/reward for episodes and a moving average (orange) over 50 episodes to visualise learning trends.

Observations: The algorithm as implemented is capable of learning to play CartPole at a level comparable to human performance, as demonstrated by Figure 7.1, however there are a few important points of discussion for run times longer than those shown in Figure 7.1. Firstly, the algorithm and network are capable of reaching an optimal solution of CartPole-v1 (500 steps) and we can use a successfully trained model to



(a) 1000 episode runtime, 2 counts of interference (b) 1000 episode runtime, 2 counts of interference

Figure 7.2: Additional results as produced from code in A.3. Sub-figures (a) and (b) are posed in identical mannerism to those of Figure 7.1.

beat the game optimally at a frequency very-unlikely for a human player. This is negatively counter weighted by what appears to be a, brief but consistent, breakdown of the implemented model during training. In the demonstrated cases of Figure 7.2 the network can be observed to "forget" how to optimise its reward for periods of up to 90 episodes.

"Catastrophic Interference" was first investigated by M. McCloskey and N. J. Cohen in their 1989 paper *"Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem"* [5]. One of the papers conclusions is that *"at least some interference will occur whenever new learning may alter weights involved in representing old learning"* [5] and that in some networks this is *"catastrophic"*. This may precisely explain the situation discussed, although, Figure 7.2 does not display a totally catastrophic behaviour as the network recovers its progression eventually and suddenly in both cases. The proposition is that interference is happening due to how the network is provided observations, which in this case is an array of floating point numbers as per Table 7.1 (rather than a visual feed of frames from which it can determine the pole's angle). As per [5]; should the network receive information from this array late in training which is unlike what may have been seen previous we expect the trained weights to update negatively. A possibility is that an extreme starting state (such as all initial observable parameters maximised or minimised - or configurations such that the cart pole accelerates initially towards termination) may be the root of this type of interference, since the network calculates loss based on experiences as buffered in replay memory, and initial conditions such as this are statistically less-likely to occur than any other combination. A key observation supporting this is that interference of this kind occurred a maximum of twice in testing, and was not a guaranteed event, which would align with the proposed hypothesis (this can be seen by running many instances of A.3).

In Figure 7.2 (b) interference occurs which the network very quickly recovers from (just before episode 400), likely due to an unfamiliar state not as extreme as those discussed. For cases where the network completed a training cycle before recovering from an instance of interference, the model failed to engage with the environment for more than 10 steps (less than 1 second) in post-training situations.

Implemented source code utilised to obtain results (complete with video output) is available in A.3.

Chapter 8

Reinforcement Learning in the Real World

8.1 Strengths & Weaknesses

8.1.1 Highlights

Reinforcement learning algorithms are inherently very adaptive to changes in environment dynamics. For example, if we made a small change in the slippery parameter probabilities of our Frozen Lake example, the agent will adjust and relearn, and will eventually find the new optimal policy for this new environment.

Another point that has been emphasized in this report is that the reinforcement learning method focuses on maximising the **long-term rewards** while exploring multiple possibilities autonomously, rather than simply completing an assigned task. It is, after all, an optimising algorithm.

8.1.2 Challenges

Since reinforcement learning relies on the exploration of the environment and the storage of action values, it requires a lot of time and computing resources. Another issue that arises from this is the **Exploration vs Exploitation trade-off**. The agent must find a balance between settling for what it knows works well (a possibly sub-optimal policy), based on its past experiences (*exploitation*), and exploring all of the unknown states of the environment in order to find the optimal policy (*exploration*).

Realistically, not everything is as simple as in our Frozen Lake example. In more complex environments, such as a game of chess, the appropriate reward for a particular action may be unknown until future actions are taken. For example, taking a pawn with a knight may seem like a positive reward, but this may leave the knight open to attack, a negative reward. This issue can propagate through multiple moves, making the assignment of rewards to specific actions at the time-of-play challenging and can introduce a large variance during training.

8.2 Applications

Without most of us even realising, reinforcement learning is used all around us. From social media algorithms to building controls and stock market predictions, reinforcement learning plays a very important role in our day to day life.

Given the nature of reinforcement learning of maximizing long-term rewards, it is present in the world of finance and trading. It can be used for optimal trade execution and portfolio optimization. Similarly, it is used in social media and recommendation systems in order to show users relevant content and keep them engaged with the app or service, to maximise usage time.

Reinforcement learning plays a big role in gaming nowadays. We see artificial intelligence giving incredible game performances that are out of reach of human capabilities. One example of this is *AlphaGo*, a computer program designed by *Deepmind* that was taught to play Go, an ancient board game originating in China and considered one of the most complex games in the world. In 2016, *AlphaGo* defeated the legendary Go player Lee Sedol.

Automated processes, such as autonomous vehicles and buildings with automated controls are being gradually introduced in our lives. In ‘smart homes’, reinforcement learning is used as a control technique for building controls. When implemented in real buildings (domestic hot water controllers, HVAC controllers, lighting controllers and window controllers), Wang’s article [3] found that 3 studies out of the 9 reviewed “reported energy/cost savings or comfort improvements compared with other controllers”.

8.3 The Future

8.3.1 Multi-agent systems

Multi-agent reinforcement learning (MARL) consists of multiple agents interacting in a common environment. They can be classified in different categories:

- **Centralized:** One agent who has access to information about the whole environment makes the decisions.
- **Decentralized:** Multiple agents who only have access to their own environments make their own decisions.
- **Cooperative:** Agents are allowed to share information about the environment with the goal of maximizing the reward of all agents.
- **Non-cooperative:** Agents do not share information and have the goal of maximizing their own rewards only.

8.3.2 Acceleration of the Training Process

As we have seen, the training process is time and resource consuming. Therefore, we want to accelerate this process. One way of doing so is by reducing the dimension of state-action space by decreasing the number of state variables. This can be done by replacing coupled state variables with a single state variable (which is defined by the

variables it replaces). For example in [2], Guan replaces two state variables, *electricity load* and *PV generation*, with one state variable, the *net power*, which is defined by the difference between the load and the generation.

The development of Deep-Q-Networks, where artificial neural networks are used to help the traditional Q-learning model adapt to more complex dynamic environments, will also help us complete more complicated tasks in a shorter period of time.

Chapter 9

Conclusions

In this report, we have explored the topic of machine learning through the lens of reinforcement learning. We have seen how dynamic programming and temporal difference control methods can be applied to Markov Decision Processes to optimise behaviour in basic and even rather complex models, in both idealistic and realistic settings, allowing machines to learn from their environments and make intelligent decisions. These models are being used in everything from robotics to finance to the automotive industry.

We then investigated the applications of Q-Learning in real problems with the Frozen Lake game and further, saw the implementation of neural networks with Deep Q Networks and the CartPole problem. Deep Q learning has the potential to revolutionise the way we approach problem-solving of complex dynamic processes.

Though lengthy, this report has only scratched the surface of reinforcement learning, covering the core concepts in detail. However, the full scope of the methods presented lies far beyond what can be contained in an undergraduate report.

Author contributions

Adam Tang contributed Introduction, Chapters 1, 2, 3.1, 3.2.1, 8. and Conclusion
James O'Sullivan contributed Chapters 3 (excl. 3.1, 3.2.1), 4, 5 and Conclusion
Jordan Walsh contributed Chapters 6, 7, Conclusion and A.1, A.2, A.3.

Bibliography

- [1] Sutton, R.S. and Barto, A.G. *Reinforcement learning: An Introduction*. 2nd edn. Cambridge, Massachusetts ; London, England: The MIT Press (2018).
- [2] Chenxiao Guan, Y. Wang, Xue Lin, S. Nazarian and M. Pedram, *Reinforcement learning-based control of residential energy storage systems for electric bill minimization*, 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas, NV, USA, 2015, pp. 637-642.
- [3] Zhe Wang, Tianzhen Hong, *Reinforcement learning for building controls: The opportunities and challenges*, Applied Energy, 269 (2020).
- [4] V. Mnih, K. Kavukcuoglu, D. Silver, et al. *Human-level control through deep reinforcement learning*. Nature **518**, 529–533 (2015). <https://doi.org/10.1038/nature14236>
- [5] M. McCloskey, N. J. Cohen, *Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem*, Academic Press, Volume 24, 109-165 (1989). [https://doi.org/10.1016/S0079-7421\(08\)60536-8](https://doi.org/10.1016/S0079-7421(08)60536-8)

Appendix A

Source Code

The code shown throughout appendix A is intended to run in **Google Colab** analogous to discussion in Chapter 6. As such, code shown in A.1 through A.3 may not be compatible with other hosting services or run in other local IDEs.

A.1 Setup and Video Wrapper

Package/Dependency Installs (*Requirement for Colab - run as shown*)

```
[ ]: !pip install gym pyvirtualdisplay > /dev/null 2>&1
      !apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
      !apt-get update > /dev/null 2>&1
      !apt-get install cmake > /dev/null 2>&1
      !pip install --upgrade setuptools 2>&1
      !pip install ez_setup > /dev/null 2>&1
      !pip install gym[classic_control]
      !pip3 install torch
```

Imports and Packages (*all*)

```
[ ]: #PyTorch
import torch
from torch import nn
from torch.nn import functional as F
from torch import optim

#OpenAI Gym
import gym
from gym import logger as gymlogger
from gym.wrappers import RecordVideo
gym.logger.set_level(40) #log errors encountered only

#Misc
import numpy as np
```

```

import random
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
import math
import time
from collections import deque
from itertools import count

#Display Window
import glob
import io
import base64
import IPython
from IPython.display import HTML
from IPython import display as ipythondisplay
is_ipython = 'inline' in matplotlib.get_backend()
from IPython.display import clear_output

```

Display Setup (Environment Monitor)

```

[ ]: from pyvirtualdisplay import Display
display = Display(visible=0, size=(1400, 900))
display.start()

```

```

[ ]: def show_video():
    mp4list = glob.glob('video/*.mp4')
    if len(mp4list) > 0:
        mp4 = mp4list[0]
        video = io.open(mp4, 'r+b').read()
        encoded = base64.b64encode(video)
        ipythondisplay.display(HTML(data='''<video alt="test" autoplay
            loop controls style="height: 400px;">
            <source src="data:video/mp4;base64,{0}" type="video/
            mp4" />.'''.format(encoded.decode('ascii'))))
    else:
        print("error finding video file.")

def wrap_env(env):
    env = RecordVideo(env, './video')
    return env

```

A.2 Q-Learning

Training Algorithm | Implementation of Algorithm 6.1 with video outputs, learns to beat a particular instance of 'Frozen Lake' by OpenAI

```

[ ]: #CHANGE RUNTIME PARAMETERS HERE
document_video = True
video_frequency = 1000 #How often video updates show if enabled
clear = True           #Clear output for each update
env_set = ["FrozenLake-v1", True, "4x4"]
           #environment, is_slippery, map size

env = gym.make(env_set[0], is_slippery=env_set[1], \
                map_name=env_set[2])
action_space_size = env.action_space.n    #returns as integer
state_space_size = env.observation_space.n #returns as integer

q_table = np.zeros((state_space_size, action_space_size))

#HYPERPARAMETERS
num_episodes = 10000
max_steps_per_episode = 100
learning_rate = 0.1
discount_rate = 0.99
exploration_rate = 1
max_exploration_rate = 1
min_exploration_rate = 0.01
exploration_decay_rate = 0.001

rewards_all_episodes = []

#Q-Learning / Training
for episode in range (num_episodes):

    if episode%video_frequency == 0 and document_video == True:
        env = wrap_env(gym.make(env_set[0], is_slippery=env_set[1],\
                                map_name=env_set[2]))

    state = env.reset()    #starts/resets simulation
    done = False          #episode state (boolean)
    rewards_current_episode = 0 #reset rewards

    for step in range (max_steps_per_episode):
        exploration_rate_thresh = random.uniform(0, 1)
        if exploration_rate_thresh > exploration_rate:
            action = np.argmax(q_table[state,:]) #EXPLOITATION
        else:
            action = env.action_space.sample()   #EXPLORATION

        #take step, return env info
        new_state, reward, done, info = env.step(action)

```



```

#update Q-Table using Bellman Equation
q_table[state, action] = q_table[state, action] * (1 - \
    learning_rate) + learning_rate * (reward + discount_rate \
    * np.max(q_table[new_state, :]))

state = new_state
rewards_current_episode += reward

if done == True:
    break #next episode

#check if episode is at an update_ep and render video if so
if episode%video_frequency == 0 and document_video == True:
    env.close()
    if clear == True:
        clear_output()
    print("Episode ", episode, " Playback:\n")
    show_video()
    print("Q-Table\n", q_table)
    env = gym.make(env_set[0], is_slippery=env_set[1], \
        map_name=env_set[2])

#update exploration rate
exploration_rate = min_exploration_rate + (max_exploration_rate \
    - min_exploration_rate) * np.exp(-exploration_decay_rate*episode)
rewards_all_episodes.append(rewards_current_episode)

#SUMMARY OUTPUT
rewards_per_thousand_episodes = np.split(np.array(\
    rewards_all_episodes), num_episodes/1000)

print("\n ~ Runtime statistics ~ \nAverage reward per thousand_\
    episodes\n")
count = 1000
for r in rewards_per_thousand_episodes:
    print(count-1000, " - ", count, " : ", str(sum(r/1000)))
    count += 1000

```

Aside: Plotting High-Res Figures (Chapter 6)

```

[ ]: episodes_to_move_avg = 100
data_split = np.split(np.array(rewards_all_episodes), \
    num_episodes/episodes_to_move_avg)

avgs_y = []
avgs_x = []
j = 0
for i in data_split:
    avgs_y.append(sum(i/episodes_to_move_avg))

```

```

    avgs_x.append(j*episodes_to_move_avg)
    j += 1

plt.figure(2, dpi=900)
plt.clf()
plt.xlabel('Episode')
plt.ylabel('Average Reward For Period')
plt.plot(avgs_x, avgs_y)
plt.plot()

```

Play Using Determined Q-Table

```

[ ]: env = wrap_env(gym.make(env_set[0], is_slippery=env_set[1], \
                             map_name=env_set[2]))
state = env.reset() #start simulation
while True:
    action = np.argmax(q_table[state,:])
    state, reward, done, info = env.step(action)
    if done:
        break;

env.close()
show_video()

```

A.3 Deep Q-Learning

Solving the ‘Cart-Pole’ problem (by OpenAI) | Implementation of Algorithm 7.1 with video outputs.

note: runtime needs to be cleared and restarted if FrozenLake cells have been run previously - video wrapper tends to malfunction and display incorrect videos if environment type changes.

```

[ ]: #obtains a rolling average of rewards to plot live during training
def get_moving_average(period, values):
    values = torch.tensor(values, dtype=torch.float)
    if len(values) >= period:
        moving_avg = values.unfold(dimension=0, size=period, step=1) \
            .mean(dim=1).flatten(start_dim=0)
        moving_avg = torch.cat((torch.zeros(period-1), moving_avg))
        return moving_avg.numpy()
    else:
        moving_avg = torch.zeros(len(values))
        return moving_avg.numpy()

#matplotlib setup for live rolling average stats
def plot(values, moving_avg_period, clear, dpi_in):
    plt.figure(2, dpi=dpi_in)

```

```
plt.clf()
plt.xlabel('Episode')
plt.ylabel('Duration')
plt.plot(values)
plt.plot(get_moving_average(moving_avg_period, values))
plt.pause(0.001)
if clear == True:
    if is_ipython: ipythondisplay.clear_output(wait=True)
```

```
[ ]: #converts input into tensor
def to_tensor(arr, dtype=torch.float, reshape=False):
    output = torch.from_numpy(np.array(arr)).type(dtype)
    if reshape:
        output = output.reshape(-1, 1)

    return output
```

```
[ ]: class DQN(nn.Module):
    def __init__(self, env, layer_size):
        super(DQN, self).__init__()
        in_features = int(np.prod(env.observation_space.shape))

        self.net = nn.Sequential(
            nn.Linear(in_features, layer_size),
            nn.ReLU(),
            nn.Linear(layer_size, layer_size),
            nn.ReLU(),
            nn.Linear(layer_size, env.action_space.n),
        )

    def forward(self, observation):
        return self.net(observation)
```

```
[ ]: class Memory(object):
    def __init__(self, maxlen):
        self.memory = deque(maxlen=maxlen)
    def store(self, experience):
        self.memory.append(experience)
    def sample(self, n_samples):
        return zip(*random.sample(self.memory, n_samples))
    def __len__(self):
        return len(self.memory)
```

```
[ ]: class EpsilonGreedyStrategy():
    def __init__(self, start, end, decay):
        self.start = start
        self.end = end
        self.decay = decay
```

```

def get_exploration_rate(self, episode):
    return self.end + (self.start - self.end)*np.exp(-episode*self.
↳decay)

```

Training Algorithm | Solves 'CartPole-v1'

potential epilepsy warning for Google Colab

```

[ ]: #Hyper Parameters
memory_capacity = 10000
learning_rate = .0001
discount_factor = 0.99
EPSILON_START = 1.0
EPSILON_END = 0.001
EPSILON_DECAY = 0.0001
NUM_EPISODES = 300
replay_memory_minimum = 500
MOVING_AVERAGE_INT = 25 #Episodes to consider in rolling average

env = gym.make("CartPole-v1")

model = DQN(env, 128)
strategy = EpsilonGreedyStrategy(EPSILON_START, EPSILON_END,
↳EPSILON_DECAY)

# Optimizer and loss function initialisation
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
loss_fn = nn.MSELoss()
memory = Memory(memory_capacity)
episode_durations = []

observation = env.reset()
episode_reward = 0
episode = 0

#INITIALISING REPLAY MEMORY
for step in range(replay_memory_minimum):
    action = env.action_space.sample()
    new_state, reward, done, info = env.step(action)
    episode_reward += reward
    if done:
        new_state = np.zeros(env.observation_space.shape)
        memory.store([observation, action, reward, int(done), new_state])
        done = done
    if done:
        observation = env.reset()
    else:
        observation = new_state

```

```

#MAIN TRAINING LOOP
observation = env.reset()
episode_reward = 0
episode = 0
step = 0
for step in count():
    if np.random.random() <= strategy.get_exploration_rate(step):
        action = env.action_space.sample()
    else:
        state = to_tensor(observation).unsqueeze(0)
        action = model(state).argmax().item()

    # Take action and store in replay memory
    new_state, reward, done, info = env.step(action)
    episode_reward += reward
    if done:
        new_state = np.zeros(env.observation_space.shape)
        episode_durations.append(episode_reward)
    memory.store([observation, action, reward, int(done), new_state])
    done = done
    observation = new_state

    if done:
        plot(episode_durations, MOVING_AVERAGE_INT, True, 100)
        obs = env.reset()
        episode_reward = 0
        episode += 1

    # Main Training Loop once replay memory is minimum capacity
    model.train() #training mode - affects how weights are calculated
                 #greatly improves the algorithms overall performance
                 #during training.
    states, actions, rewards, dones, new_states = memory.sample(128)

    # Wrap all values
    states = to_tensor(states)
    actions = to_tensor(actions, torch.int64, reshape=True)
    new_states = to_tensor(new_states)
    rewards = to_tensor(rewards, reshape=True)
    dones = to_tensor(dones, reshape=True)

    # Get current q-values
    qs = model(states)
    qs = torch.gather(qs, dim=1, index=actions)

    # Compute target q-values

```

```

with torch.no_grad():
    predicted_qs, _ = model(new_states).max(dim=1)
    predicted_qs = predicted_qs.reshape(-1, 1)

#Form of the piecewise function shown in the DeepMind 2015 paper
target_qs = rewards + discount_factor * (1 - dones) * \
    predicted_qs.reshape(-1, 1)

# Compute loss
loss = loss_fn(qs, target_qs)
optimizer.zero_grad()
loss.backward()
optimizer.step() #Backpropagation

#break loop and run model
if episode == NUM_EPISODES:
    env.close()
    wrap_env(env)
    done = False
    while done == False:
        state = to_tensor(obs).unsqueeze(0)
        action = model(state).argmax().item()
        obs, reward, done, info = env.step(action)
    env.close()
    plot(episode_durations, MOVING_AVERAGE_INT, False, 100)
    print("Final Run:")
    show_video()
    print("See Cell below to run more.")
    break

```

Play Using Trained Model

```

[ ]: env = wrap_env(gym.make('CartPole-v1'))
obs = env.reset()
done = False
total_reward = 0
while done == False:
    state = to_tensor(obs).unsqueeze(0)
    action = model(state).argmax().item()
    obs, reward, done, info = env.step(action)
    total_reward += reward
env.close()
show_video()
print("Total Reward/Runtime = ", total_reward)
if total_reward >= 500:
    print("Optimal Solution for CartPole-v1!")

```

Aside: Plotting High-Res Figures (Chapter 7)

```
[ ]: plot(episode_durations, 50, False, 900)
```