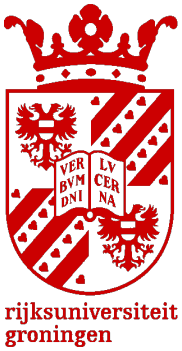
Contextual Online Imitation Learning (COIL) : A Novel Method of Utilising Guide Policies in Reinforcement Learning

Alexander Hill - Master's Thesis



1st Supervisor : Prof. Dr. Marco Grzegorczyk
2nd Supervisor : Prof. Dr. Raffaella Carloni

University of Groningen
Mathematics
July 2022

Abstract

This paper proposes a novel method of utilising guide policies in Reinforcement Learning problems; Contextual Online Imitation Learning (COIL). This paper will demonstrate that COIL has the potential to solve Reinforcement Learning tasks better than both traditional Imitation Learning, and also Deep Reinforcement Learning algorithms such as Proximal Policy Optimisation (PPO). COIL can also effectively utilise non-expert guide policies, making it more flexible than current methods that integrate guide policies. This paper demonstrates that through using COIL, guide policies that achieve good performance in sub-tasks can also be used to help Reinforcement Learning agents looking to solve more complex tasks. This is a significant improvement in flexibility over traditional Imitation Learning methods. After discussing in depth some prerequisite knowledge in Reinforcement Learning and Imitation Learning, this paper will introduce the theory and motivation behind COIL, and will also test the effectiveness of COIL in a self-driving car simulation and real-life robot. In both applications, COIL gives stronger results than traditional Imitation Learning, Deep Reinforcement Learning, and the also the guide policy itself.

Contents

1 Introduction **8**

2 Theory & Methods **10**
2.1 Markov Decision Processes 11
2.1.1 Definition 11
2.1.2 Value Functions 15
2.1.3 Bellman Equations and Optimal Policies 17
2.1.4 Policy Iteration 19
2.1.5 Value Iteration 24
2.2 Classical Reinforcement Learning 26
2.2.1 Q-Learning 26
2.2.2 Policy Gradients 29
2.2.3 Actor-Critic Methods 32
2.3 Deep Reinforcement Learning 34
2.3.1 Deep Q-Networks 35
2.3.2 PPO & TRPO 37
2.4 Imitation Learning & Guide Policies 41
2.4.1 Offline Imitation Learning 41
2.4.2 DAgger 42
2.4.3 Reward Function Coupling 44

3 Contextual Online Imitation Learning **46**
3.1 Motivation 46
3.2 Formalisation 49
3.3 Dynamic COIL 50

4 Results & Applications **54**
4.1 Simulation 54
4.1.1 Setup 54
4.1.2 Training 58
4.1.3 Generalisation 60
4.2 Robot 61
4.2.1 Setup 62
4.2.2 Results 67

5 Critical Discussion **69**

6 Conclusion **70**

A Appendix **75**
A.1 Guide Policy 75
A.2 Cubic Splines 78
A.3 Robot Implementation Details 81

Acknowledgements

Undertaking this Master's Thesis has been both an exciting and challenging experience that would not have been possible without the help of many people. Firstly, I would like to express my sincere gratitude to my supervisors Dr. M. Grzegorczyk (Marco) and Dr. R. Carloni (Raffaella), who guided me through the process and provided excellent support throughout. I would also like to thank Dr. M. Sabatelli (Matthia) for taking the time to give me his expert advice on my thesis topic, and Marc Groefsema for the great help he provided in the Robotics Lab setting up the robot for my experiments. Lastly, I would like to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster.

List of Figures

1 The standard setup for Reinforcement Learning given an agent and environment. In each time step $t \in [0, T]$, the agent is provided with information from the current state s_t, and then takes an action a_t within its environment. The reward for that time step is then calculated via the reward function $r_t = r(s_t, a_t)$. 8

2 Example of a Markov Process with two states. The transition probabilities are denoted on the arrows between states. 13

3 Example of a Markov Process with two states. The transition probabilities are denoted on the arrows between states. The rewards associated with each step can be seen adjacent to the corresponding arrows. 13

4 Example of a Markov Decision Process with two states and two actions. The transition probabilities are denoted on the arrows between states. The rewards associated with each step can be seen adjacent to the corresponding arrows. These rewards and transition probabilities are dependent on the action taken, which can be seen when comparing the left and right graphs. 14

5 Illustration of the Policy Iteration algorithm. The Policy Iteration algorithm begins with a Markov Decision Process and initial policy π_0, and by continually applying Policy evaluation and Policy improvement, returns the optimal policy π^*. 22

6 The state-value function being trained via the Value Iteration algorithm. Here we have two states from a simple toymaker example. After approximately 80 iterations the state-value function converges. 25

7 Illustration of the Q-learning algorithm. We begin with an initialised Q-table at the top indicating the state-value function $Q^\pi(a, s)$ of each action $a \in \mathcal{A}$ and state $s \in \mathcal{S}$, and through a repeated 3 step procedure we update our Q-table until we reach convergence. The output is an approximation of the optimal state-value function $Q^\pi(a, s)$. 28

8 Illustration of the Actor-Critic methodology of solving Reinforcement Learning Tasks. The policy π_θ is improved via gradient ascent, where the gradient of the evaluation metric $J(\theta)$ is calculated using an estimate of one of the value-functions. 34

9 Illustration of the Deep Q-Network methodology, from data collection in the environment to computation of the loss function and training of the prediction neural network. 37

10 Plots showing one term (a single time step) of the objective function L^{CLIP} as a function of $r_t(\theta)$, for $\hat{A}_t < 0$ (left) and $\hat{A}_t > 0$ (right). The values of $r_t(\theta)$ in which the value of L^{CLIP} is clipped is highlighted in red. This clipping removes the incentive for the agent to make an even larger update to the policy during training. 39

11 Illustration of the Actor-Critic methodology of solving Reinforcement Learning tasks where the objective function is from PPO. The policy π_θ is improved via gradient ascent, where the gradient of the evaluation metric $L^{CLIP}(\theta)$ is calculated using an estimate of the advantage value function A^π. 40

12 Illustration of the negative feedback loop that can occur when using Offline Imitation Learning/Behavioural Cloning. Poor generalisation ability leads to degradation of the supervised learning model which only leads to further deviation from the trajectories seen in the training dataset. 43

13 Illustration of the DAgger algorithm. There are three main steps to the algorithm. The first step is to simulate the current learning agent policy in the environment and collect a set of visited states s_t for $t = 1, \ldots, T$. The second step is to use the expert guide policy to label each state in the dataset with the action it would have taken given it was in that state. Lastly, the supervised learning model used to imitate the guide policy is retrained using the additional data collected in step 1. This process can continue until convergence or until a pre-determined number of iterations is met. 44

14 The standard setup for Reinforcement Learning given an agent and environment. In each time step $t \in [0, T]$, the agent is provided with information from the current state s_t, and then takes an action a_t within its environment. The reward for that time step is then calculated via the reward function $R_t = R(s_t, a_t)$. 46
15 Illustration of Contextual Online Imitation Learning (COIL). In comparison to regular Reinforcement Learning, in COIL the action of some guide policy π_{guide} is provided to the agent as an observation at each time point t. 47
16 Illustration of Contextual Online Imitation Learning (COIL) with multiple guide policies. . 47
17 Illustration of Dynamic Contextual Online Imitation Learning (Dynamic COIL). In the agent training phase, we have the regular COIL method, where the agent is being fed the actions of the guide policy at each time step t, and the agent is improving its policy through repeated interaction with the environment. In the guide policy fine-tuning phase, the agent's policy is fixed, and the learning algorithm now operates only on the parameters of the guide policy. In this way the guide policy can be fine-tuned in order to offer the most beneficial action to the agent. Dynamic COIL works by iteratively switching between these two phases. 51
18 Illustration of the Dynamic COIL architecture in the case that neural networks are used to parameterize both the agent network and the guide policies. This illustration also assumes a discrete action space $\mathcal{A}$. Each guide policy provides its suggested action as an input to the agent, and the agent learns through repeated exposure with the environment how best to use these guide policies. In Dynamic COIL, the guide policies are also fine-tuned during training. This illustration shows how the guide policies and the form one greater network which can be trained via back-propagation[1]. 53
19 Image of a self-driving car simulation programmed in Python. The simulated car can be seen driving from left to right. The left cones (yellow) and right cones (blue) can also be seen guiding the car down a track. The red dots represent the middle of the track and are used for visual purposes. Cones that are detected by the car have a dashed line drawn between the car and the cone. Furthermore, the grey dotted line on either side of the track represent a cubic spline estimate of the boundary of the track. 54
20 The COIL methodology applied to the application of a self-driving car simulation. The guide policy in this case is called a 'safety policy' because the guide policy in this case is designed entirely to make sure the car does not crash and nothing else. For more details see Appendix (A.1). 57
21 Four of the seven tracks used to train the Reinforcement Learning agent to drive autonomously. All tracks were designed in Python using the package Pygame. The yellow circles represent left cones, and the blue circles represent right cones. 58
22 The training runs of 3 agents trained by COIL and 3 agents trained by regular RL. COIL can be seen to achieve a higher mean reward at every point along the x-axis. These training runs were computed using the University of Groningen's high performance computer cluster Peregrine. 59
23 Photo of the robot car used for comparing autonomous driving algorithms. The robot was provided by the Robotics Lab in the Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence at the University of Groningen. On top of the robot there is a single Lidar detector (RPLidar A1M8) which will be the only sensor available for autonomous driving. 61
24 Photo of the robot car along with some cones representing the left and right side of the track. The left and right cones were designed to have different characteristics (size, width, etc.) so that a classifier could be trained using only Lidar data. The Lidar sensor can be seen on the top of the robot in this photo. 62

25 Scatter graph of detected cones from the Lidar. For each cone, 3 variables (width, depth, and girth) were extracted using the Lidar data and then principal component analysis was performed to find the two most informative components. In this graph, detected left cones are represented as blue, and right cones are represented as green, with the axes representing the two principal components. 63
26 Image of the 3D robot visualiser RViz during a lap of the robot on the test track. Red points represent sensor data from the Lidar, yellow circles represent detected left cones, blue circles represent detected right cones, and the white text in the middle is the RViz label given to the robot. 64
27 Illustration of the limitation of using only a Lidar for detecting cones. It can be seen here that Cone 1 is casting a shadow for the Lidar sensor, resulting in Cone 2 being undetected. 65
28 Photo of the real-life test track designed to test and compare the self-driving software. Due to limited space, this track was chosen in order to include at least one left turn, one right turn, and one straight region. 66
29 Depiction of the dynamic co-ordinate system fixed to the car (dashed lines), used to calculate the required turning angle α to steer the car towards the nearest target. d represents the distance between the car and the target and the car angle is the direction the car is facing measured from the x-axis. 75
30 The shape of the function f which takes the required turning angle α (x-axis) and the distance to the nearest cone d and returns the steering angle β (y-axis) the car must take in order to eventually arrive at the target. 76
31 The process of using visible cone locations to compute a set of targets using cubic splines. Firstly (left), the midpoints between each pair of left cones (red) and right cones (green) are calculated. Secondly (middle), a cubic smoothing spline is generated using the midpoints as input. Lastly (right), this cubic spline is discretized into a finite set of targets for the autonomous car to follow. 77
32 Illustration of the communication scheme used to receive Lidar data from the robot, and send instructions back from the computer. Due to incompatible operating systems and Python versions, a third computer (laptop) was used to process the actions of the Reinforcement Learning agent. 81

List of Notation

This section provides a concise list describing the notation used throughout this paper.

$\mathbb{P}(X)$	Probability distribution of random variable X.
$\mathbb{P}(X \mid Y)$	Probability distribution of random variable X conditioned on Y.
$\mathbb{E}[X]$	Expectation of random variable X.
$\{0, 1, \ldots, n\}$	The set of all integers between 0 and n.
$[a, b]$	The real interval between a and b inclusive.
$[a, b)$	The real interval including a but excluding b.
v_i	The i-th element of vector v.
$A_{i,j}$	The element (i, j) of matrix A.
$\lvert A \rvert$	The number of elements in set A.
$A \times B$	The Cartesian product of sets A and B.
$\nabla_x y$	The gradient of y with respect to x.
$\int f(x)\, dx$	Definite integral over the entire domain of x.
$\int_S f(x)\, dx$	Definite integral with respect to x over the set S.
$\sum_{i=1}^{n} X_i$	Summation of elements X_i for $i = 1, \ldots, n$.
$\prod_{i=1}^{n} X_i$	Product of elements X_i for $i = 1, \ldots, n$.
$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$.
$f \circ g$	Composition of the functions f and g.
$f(x; \theta)$	A function of x parameterised by θ.
$\log x$	Natural logarithm of x.
$\mathbb{1}\{A\}$	The indicator function with condition A.
$\hat{X}$	Estimator of random variable X.
$\lvert\lvert X \rvert\rvert$	The Euclidean norm (L^2 norm) of X.
$\lvert\lvert X \rvert\rvert_p$	The L^p norm of X.

1 Introduction

" If intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake. "

- Yann LeCun

Reinforcement learning is a vast field spanning many applications from computational neuroscience to robotics. Research in Reinforcement Learning began in the 1950s in the field of Optimal Control as a formal framework to define optimization methods to derive control policies in continuous time control problems [2–4]. Broadly speaking, Reinforcement Learning encapsulates the methodology of training an agent to adopt a certain behaviour through interactions with a dynamic environment. An example of this is an autonomous stock market trader which learns the optimal trading strategy through repeated buying and selling of stocks. Reinforcement Learning algorithms differ in how they use these past interactions to create future strategies. As we will see in this paper, there are many effective ways of achieving this goal.

The type of task that Reinforcement Learning is trying to solve is *sequential* in nature, there is an often element of time involved. The standard setup between a Reinforcement Learning agent and its environment can be seen below in Figure (1).

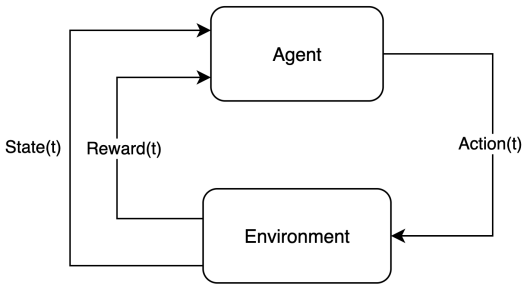


Figure 1: The standard setup for Reinforcement Learning given an agent and environment. In each time step $t \in [0, T]$, the agent is provided with information from the current state s_t, and then takes an action a_t within its environment. The reward for that time step is then calculated via the reward function $r_t = r(s_t, a_t)$.

In order to begin analysing Reinforcement Learning tasks in detail, a mathematical construction known as a Markov Decision Process (MDP) is used to describe all of the important characteristics of the environment. Markov Decision Processes are important for Reinforcement Learning because they provide a concrete mathematical framework for researchers to build environments for

agents to execute policies upon. For self-driving cars, the MDP would have to contain all of the information regarding the cars dynamics and the road that it is driving upon.

Research on Markov Decision Processes began in the 1950s by Bellman's paper *A Markovian decision process* [3]. This paper established the core concepts and structures surrounding Markov Decision Processes and allowed other mathematicians to start working on the new idea. In 1960, a large body of work on Markov Decision Processes was published by Howard in the book *Dynamic Programming and Markov Processes* [2]. This book developed the theory extensively and introduced new methods to identify optimal strategies for MDPs, such as value iteration and policy iteration. After these important initial developments, additional major works on MDPs were produced in the 1990s, such as *Dynamic programming and optimal control* by Bertsekas [5] in 1995, *Reinforcement Learning* by Sutton [6] in 1998, and *Markov decision processes : discrete stochastic dynamic programming* by Puterman [7] in 1994. The theory in this section will take inspiration from the initial methods described in Howards book, and also tie together some of the significant concepts developed in the books by Sutton, Bertsekas, and Puterman [5–7].

Following the establishment of Markov Decision Processes, many Reinforcement Learning algorithms were developed in the 1980s, most notably, Q-learning developed by Watkins in 1989 [8] and the REINFORCE algorithm developed by Williams in 1987 [9]. These algorithms will be discussed in full in Section (2.2). Following this, with the rise of neural networks and Deep Learning in the 21st century, a host of new algorithms leveraging this new technology were created in the 2010s and given the name Deep Reinforcement Learning. In particular, Deep Q-Networks developed by Mnih et al. in 2013 [10], and trust-region methods such as TRPO and PPO were developed by Schulman et al. in 2015 and 2017 respectively [11, 12]. These Deep Reinforcement Learning algorithms will be discussed at length in Section (2.3).

Following the discussion on classical and modern Reinforcement Learning algorithms, in Section (2.4) the paradigm of Imitation Learning will be explored and three different Imitation Learning methods will be compared; Offline Imitation Learning, DAgger, and Reward Function Coupling. Imitation Learning is similar to Reinforcement Learning, however, instead of learning a strategy through repeated interaction with an environment, a strategy is learned through demonstrations from an 'expert' at the task called a guide policy. Imitation Learning is therefore only applicable in the case where researchers have access to such a guide policy. Research in Imitation Learning began in software development in the paper *Programming by Example* by Halbert in 1984 [13]. It was soon picked up by Artificial Intelligence researchers and given the name 'Imitation Learning' and 'Learning from Demonstration' [14]. Since then, many powerful Imitation Learning methods have been developed, some of which also utilise the advances in Deep Learning [15]. The details of Imitation Learning methods will be treated in Section (2.4).

Lastly, a novel method of utilising guide policies will be introduced in Section (3) : Contextual Online Imitation Learning (COIL). In COIL, the actions

of the guide policy are fed into the agent as an additional *observation* during training. Within this section, the motivation behind COIL will be discussed and different adaptations of the algorithm will be explored. In particular, the biggest advantage of COIL is its flexibility regarding the types of guide policies that can be effectively utilised. In traditional Imitation Learning methods, only 'expert' guide policies can be used, in COIL this is no longer the case. COIL allows the agent to learn for itself through interaction with the environment the optimal way in which to use the guide policy. Even if the guide policy is only effective in some subset of states in the environment, COIL is capable of training the agent to 'listen' to the suggestions of the guide policy only in this set of states and 'reject' the suggestions of the guide policy in all other states where the guide policy is non-optimal, hence why it is named 'Contextual'. This is a significant methodological improvement over the current methods of Imitation Learning. This will be discussed more in detail in Section (3.1).

Additionally, in Section (3.3) an extension of COIL will be introduced, called Dynamic COIL. In Dynamic COIL, a training schedule is proposed where after the COIL agent has been trained, the guide policy itself is allowed to be fine-tuned. This allows the guide policy to be slightly adjusted in order to maximise the proficiency of the agent. It is conjectured that Dynamic COIL will be effective in multi-task Reinforcement Learning tasks. Furthermore, in Section (4), COIL will be applied to a self-driving car simulation as well as a self-driving robot, and it will be directly compared to both Imitation Learning methods and modern Reinforcement Learning methods. It will be shown here that there is strong evidence that COIL is an effective method of utilising guide policies in Reinforcement Learning tasks. It will also be shown that there is evidence that COIL is also robust to transfer learning. This will demonstrate that COIL has great potential for solving Reinforcement Learning tasks in practice as it gives researchers and engineers a method of utilising their guide policies even if they are not experts in all states or all objectives. Following this, the potential limitations of COIL will be discussed and also the possible avenues for future research on this new method.

2 Theory & Methods

This section will bring together the relevant theoretical material across a range of literature to give a coherent and clear overview of the field of Reinforcement Learning. In order to do this in one consistent format, a significant amount of notation had to be constructed and edited. Almost all of the theory in this section has re-written theorems, proofs, and algorithms in order to make the necessary clarifications and piece it all together. Furthermore, all diagrams, figures, and pseudo-code in this section were designed specifically for this paper. Before moving onto the Reinforcement Learning algorithms themselves, we must first discuss the Markov Decision Process.

2.1 Markov Decision Processes

This section will begin by defining the Markov Decision Process (MDP) in full, before moving onto some elementary algorithms designed to find optimal strategies for MDPs. In particular, Policy Iteration and Value Iteration are two highly influential algorithms which will be treated at the end of this section.

2.1.1 Definition

In this section we will incrementally build up the full definition of the Markov Decision Process from its constituent elements. First, we will mathematically define the notion of a graph, then by introducing the necessary extensions to the graph we will arrive at the Markov Decision Process, the mathematical construction widely adopted to represent agents and environments in Reinforcement Learning.

Definition 2.1. A graph is a pair $G = (\mathrm{V}, \mathrm{E})$ where V is a set of vertices and E is a symmetric subset of $\mathrm{V} \times \mathrm{V}$, containing the set of edges. E is symmetric iff $(x, y) \in \mathrm{E} \iff (\mathrm{y}, \mathrm{x}) \in \mathrm{E}$.

A *directed* graph is a graph where the edge set E no longer adheres to the property of symmetry, therefore if $(x, y) \in E$, this does not directly imply that $(y, x) \in E$. In general, the graphs that are used as the basis for Markov Decision Processes are directed graphs, as a result the symmetric graph will remain a special case for the remainder of this paper.

In Reinforcement Learning literature, the vertices in V are referred to as *states*. To remain in line with the literature we will now refer to the set of states as $\mathcal{S}$, instead of V. Now that we have a graph, we can define a Markov Process (or Markov chain) as a graph where a 'walker' is able to traverse the graph through time. In this way we can define the initial state of the walker as $s_0 \in \mathcal{S}$, and at each subsequent time step, the walker travels along an edge in E from s_t to s_{t+1}. In general, at a state s_t the walker will have multiple potential edges in E to choose from. In this case, the edge that is chosen is governed by a function $\mathcal{T}_{ss'}$ that represents the transition probabilities of the Markov process. Formally we have

$$\mathcal{T}_{ss'} = \mathbb{P}(s_{t+1} = s' \mid s_t = s) \quad . \tag{1}$$

Therefore, in the case where the walker has multiple choices, the resulting direction will be chosen randomly according to this function $\mathcal{T}_{ss'}$. It should be noted that it is common in literature to represent the transition probabilities in a matrix $\{\mathcal{T}\}_{i,j}$ where the element (i, j) represents the probability of traveling from vertex v_i to vertex v_j.

Markov Processes are named after the Russian mathematician Andrey Markov, as they exhibit the *Markov property*. The Markov property indicates that when an agent is in a state s_t, the probability distribution governing its next transition to state s_{t+1} is independent to all of the states that the agent has been in previous to state s_t. In this way we can call any system that maintains this

property as *Markovian*. Formally, this property can be expressed as

$$\mathbb{P}(s_t \mid s_{t-1},\ \ldots\ ,\ s_0) = \mathbb{P}(s_t \mid s_{t-1}) \tag{2}$$

As the name might suggest, Markov Decision Processes also exhibit the Markov property.

Markov Decision Processes are an extension of Markov Processes where at each time step t, the walker is able to take an action a_t within an action space $\mathcal{A}$ which modifies the transition probabilities $\mathcal{T}$ acting upon the graph. In this way the walker can express preferences between certain transitions in the graph. The transition probabilities are therefore now also a function of the actions at each time step a_t.

$$\mathcal{T}^a_{ss'} = \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)\ \ . \tag{3}$$

In specific cases we will benefit from also using the notation $\mathcal{T}(s, a, s')$ to denote the transition probabilities. Now that decisions are able to be made, we will from this point refer to the walker as the *agent*. This change in terminology reflects that the agent is no longer traveling in random direction, but is able to learn through exposure to its environment and make actions to influence its trajectory in strategic ways.

The final important extension that we have to make is that each transition from state s to state s' is accompanied by a certain *reward*. This reward is determined by a reward function $\mathcal{R}^a_{ss'}$. We define $\mathcal{R}^a_{ss'}$ as the function

$$\mathcal{R}^a_{ss'} : \mathcal{S}^2 \times \mathcal{A} \to [0, 1] \tag{4}$$

and can be interpreted as the probability distribution over the reward collected by the agent r_t at time step t given that the agent began at s, took action a, and ended up in state s' in the following time step. In specific cases we will benefit from also using the notation $\mathcal{R}(s, a, s')$ to denote the reward function.

Now that all of the essential components are properly introduced we may finally define the Markov Decision Processes as the collection $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$. Before we continue let's look at a simple example. Imagine a toymaker involved in the novelty toy business. Given any particular week he may be in one of two states: He is in state 1 if the toy he is currently producing is successful in the local area, and he is in state 2 if the toy fails and goes unnoticed by potential buyers. We can summarise these two states in the state space $\mathcal{S} := \{\text{Success} : 1,\ \text{Failure} : 2\}$. When he is in state 1, he has a 50% chance of remaining in state 1 and a 50% chance of transitioning into state 2. When he is in state 2, he has a chance to experiment with new toys, and he returns to state 1 with a 40% chance and remains where he is with 60% chance. Therefore our transition probabilities can be expressed as

$$\mathcal{T}_{i,j} = \begin{bmatrix} 0.5 & 0.5 \\ 0.4 & 0.6 \end{bmatrix} \quad .$$

Combining both $\mathcal{S}$ and $\mathcal{T}$, all of the information can be summarised as shown below in Figure (2).

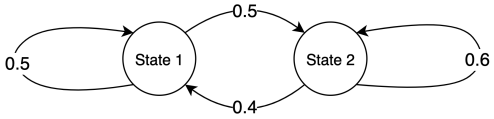


Figure 2: Example of a Markov Process with two states. The transition probabilities are denoted on the arrows between states.

To complete our MDP, we now need a reward function $\mathcal{R}$ and an action space $\mathcal{A}$. On a regular day, if the toymaker is in state 1 and remains in state 1, then he makes \$900 that week. Conversely, if he starts in state 1 and transfers to state 2, he only makes \$300. If the toymaker is in state 2 and then remains in state 2 (an unfortunate scenario indeed), he loses \$700. If he begins in state 2 and manages to get back to state 1, he makes \$300. Therefore, our reward function $\mathcal{R}$ is given by

$$\mathcal{R}_{i,j} = \begin{bmatrix} 9 & 3 \\ 3 & -7 \end{bmatrix} \quad .$$

We can now add these rewards to the graph to give a picture of everything we know about the toymaker so far. This can be seen below in Figure (3).

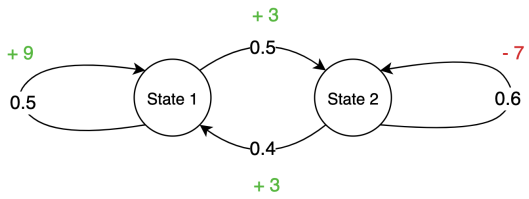


Figure 3: Example of a Markov Process with two states. The transition probabilities are denoted on the arrows between states. The rewards associated with each step can be seen adjacent to the corresponding arrows.

Note that the rewards given in Figure (3) are those when the toymaker is making toys on a 'regular' day. Now suppose that each week the toymaker has the option of advertising his toys to the local area. This gives us an action space of $\mathcal{A} = \{\text{No Advertising}, \text{ Advertising}\}$. If the toymaker decides to advertise in a given week, he will have to spend extra money on making the advertisements and distribution. However, the chance of having a successful toy next week is greatly increased. This effect can be seen below in an updated $\mathcal{T}$ and $\mathcal{R}$ in Figure (4).
It is clear from the above figure that advertising can be risky if it is unsuccessful in producing a transition from state 2 to state 1, however, if may be worth it to reduce the chance of multiple weeks in state 2. In each week, the toymaker gets to make the choice whether he wants to be situated within the Markov Process

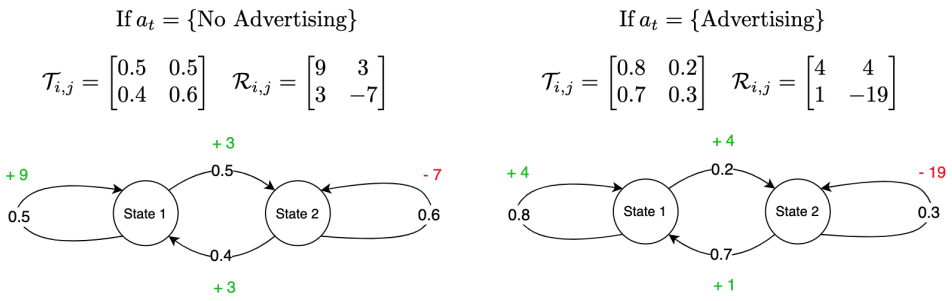


Figure 4: Example of a Markov Decision Process with two states and two actions. The transition probabilities are denoted on the arrows between states. The rewards associated with each step can be seen adjacent to the corresponding arrows. These rewards and transition probabilities are dependent on the action taken, which can be seen when comparing the left and right graphs.

on the left, or the Markov Process on the right. The reader here is encouraged to have a think about what the optimal strategy might be for the toymaker i.e. which strategy is the most profitable?

A strategy for a Markov Decision Process is called a *policy* and is denoted $\pi(a, s)$. A policy $\pi(a, s)$ is a mapping from the state-action space $\mathcal{A} \times \mathcal{S}$ to $[0, 1]$ where $\pi(a, s)$ represents the probability of taking the action a in state s. The policy can either be deterministic or stochastic. An example of a deterministic policy for the toymaker would be to advertise only when he is in state 2, formally this would give us

$$\pi(a, s) = \begin{cases} \text{No Advertising} & \text{if } s = 1 \\ \text{Advertising} & \text{if } s = 2 \end{cases}$$

An example of a stochastic policy would be if the toymaker advertised with probability 0.5 when in state 1 and advertised with probability 0.8 when in state 2. This would be

$$\pi(a, s) = \begin{cases} \text{Advertising with probability } 50\% & \text{if } s = 1 \\ \text{No Advertising with probability } 50\% & \text{if } s = 1 \\ \\ \text{Advertising with probability } 80\% & \text{if } s = 2 \\ \text{No Advertising with probability } 20\% & \text{if } s = 2 \end{cases}$$

The goal of Reinforcement Learning is to find the policy that maximises the expected total reward of the MDP. Mathematically, we want to maximise

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{5}$$

where $\gamma \in [0, 1]$ is a called the *discount factor*. The discount factor determines how much the reinforcement learning agent will prioritise rewards in the immediate future relative to those in the distant future. If $\gamma = 0$, the agent will make its decision entirely based upon the reward it will get for its next action. If $\gamma = 1$, the agent will evaluate each action based on the sum total of all of its future rewards. In practice, a discount factor of approximately 0.99 yields good results.

Given a policy π, we define a trajectory τ of length T as a sequence of state-action pairs (s_t, a_t) for $t = 1, \ldots, T$ realised during the rollout of π in the environment. Given an initial state distribution $p(s_0)$ we can calculate the probability of realising a trajectory τ with policy π as

$$\pi(\tau) = \pi\left(s_0, a_0, \ldots, s_T, a_T\right) = p\left(s_0\right) \prod_{t=0}^{T} \pi\left(a_t, s_t\right) \mathcal{T}_{s_t s_{t+1}}^{a_t} \quad . \tag{6}$$

Additionally, we define the *return* of a trajectory τ of length T as

$$r(\tau) = r_0 + \ldots + r_T = \sum_{t=0}^{T} r_t \tag{7}$$

and without loss of generality we can include the discount factor to get

$$r(\tau) = r_0 + \gamma r_1 + \ldots + \gamma^T r_T = \sum_{t=0}^{T} \gamma^t r_t \quad . \tag{8}$$

Furthermore, if an agent is currently at time t, we shall denote the future return of the agent given it follows policy π as

$$R_t = \sum_{k=0}^{\infty} r_{t+k+1} \tag{9}$$

and with the discount factor we get

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad . \tag{10}$$

Therefore, the goal of Reinforcement Learning algorithms is to find the policy π that maximises the expected future return for the agent. In the subsequent sections we will construct methods of finding optimal policies for arbitrary MDPs, and in the process we will help our toymaker in finding his optimal policy.

2.1.2 Value Functions

An incredibly useful tool in Reinforcement Learning development are *value functions*. Almost all Reinforcement Learning algorithms rely on estimating value

functions in one way or another. Value functions attempt to evaluate how valuable certain states or state-action pairs are. Firstly, given a policy $\pi(a, s)$ and a state s we have the *state-value function* :

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R_t \mid s_t = s\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right] \tag{11}$$

which represents the total expected reward provided that the agent follows the policy $\pi(a, s)$ and begins at state s. Additionally, $\mathbb{E}_{\pi}$ represents the expected value under the assumption that the policy $\pi(a, s)$ is currently being used. Intuitively, if our agent has access to an accurate state-value function then it could take actions to increase the probability that it ends up in states with a high expected total reward.

The second useful value function is the *action-value function* :

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[R_t \mid s_t = s, a_t = a\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right] \tag{12}$$

which is similar to the state-value function but it predicts the total expected reward for policy $\pi(a, s)$ given that the agent is currently in state s and taking action a. The action-value function can be seen as a specific case of the state-value function where the action is fixed in advance. The two value functions $V^{\pi}(s)$ and $Q^{\pi}(a, s)$ are intrinsically tied to one another. Given either one of them it is possible to construct the other using the following equations:

$$V_{\pi}(s) = \sum_{a} \pi(a, s) \cdot Q_{\pi}(s, a) \tag{13}$$

and

$$Q_{\pi}(s, a) = \mathbb{E}_{\pi}\left[r_t + \gamma V^{\pi}(s_{t+1}) \mid s_t = s,\ a_t = a\right] \tag{14}$$

$$= \sum_{s'} \mathcal{T}^{a}_{ss'} \left(\mathcal{R}^{a}_{ss'} + \gamma V_{\pi}\left(s'\right)\right) \tag{15}$$

where this final equality is a result of the Law of the Unconscious Statistician [16].

The action-value function Q^{π} is particularly useful because we can define a policy π' as always taking the action that provides the highest expected reward according to Q^{π}. Therefore we have

$$\pi'(a, s) = \arg\max_{a} Q^{\pi}(a, s) \quad . \tag{16}$$

Of course, the success of this policy is completely dependent on the accuracy of Q^{π}. In practice, it might be very difficult to learn an accurate action-value function.

A final value function which is useful is the $advantage function$ which is defined as

$$A^{\pi}(a, s) = Q^{\pi}(a, s) - V^{\pi}(s) \quad . \tag{17}$$

The advantage function is used to determine the advantage gained by taking action a in state s compared to the average. Therefore, we can say that if $A^\pi(a,s) > 0$ then taking the action a is better than average.

2.1.3 Bellman Equations and Optimal Policies

All of the value functions we have discussed are able to be reformulated into Bellman equations, which implies they have a special self-consistency property. The Bellman equations imply a recursive formulation of the value functions and come from the idea that the value of your starting state could be seen as the value you expect to get from starting from there plus the value you expect to get from wherever you land from there on. The Bellman equation for $V^\pi(s)$ can be derived as follows:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right] \tag{18}$$

$$= \mathbb{E}_\pi \left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s \right] \tag{19}$$

$$= \sum_a \pi(a,s) \sum_{s'} \mathcal{T}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_{t+1} = s' \right] \right] \tag{20}$$

$$= \sum_a \pi(a,s) \sum_{s'} \mathcal{T}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s') \right] \tag{21}$$

where the relationship between Equation (20) and Equation (21) holds because the expected total reward at a state s is the same at time t as it is at time $t+1$, as long as the MDP is not terminated in the near future.

We can also derive the Bellman equation for $Q^\pi(a,s)$ as follows:

$$Q^\pi(a,s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s,\ a_t = a \right] \tag{22}$$

$$= \mathbb{E}_\pi \left[r_{t+1} + \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s,\ a_t = a \right] \tag{23}$$

$$= \sum_{s\prime} \mathcal{T}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma \cdot \sum_{a'} \pi(a',s') \cdot \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s,\ a_t = a \right] \right) \tag{24}$$

$$= \sum_{s\prime} \mathcal{T}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma \cdot \sum_{a'} \pi(a',s') \cdot Q_\pi(a',s') \right) \tag{25}$$

and using Equation (13) we can also rewrite Equation (25) as

$$Q_\pi(a,s) = \sum_{s'} \mathcal{T}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma V_\pi(s') \right) \quad . \tag{26}$$

Reformulating the value functions as Bellman equations allows us to exploit the recurrence relation between subsequent time steps. The key point now is that both V^π and Q^π are fixed points of their own Bellman equations, which is an important reason why the Value Iteration and Policy Iteration algorithm work. We will cover these algorithms in the next section, but first we must discuss optimal value functions.

In order to solve a Reinforcement Learning task, it is necessary to find a policy that maximises the amount of reward attained by the agent in the environment. Using the value functions we can define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected total reward is greater than or equal to that of π' for all states $s \in \mathcal{S}$. This means that $\pi \geq \pi' \iff V^\pi(s) \geq V^{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy π^* called the optimal policy that is better than or equal to all other policies. It is possible that there may be more than one optimal policy, however, we will denote all of the optimal policies by π^*. The optimal policy is coupled by an optimal state-value function, denoted $V^*(s)$, and defined as

$$V^*(s) = \max_\pi V^\pi(s) \quad \text{for all } s \in \mathcal{S} \,. \tag{27}$$

Optimal policies also have an optimal action-value function denoted $Q^*(a, s)$, and defined as

$$Q^*(a, s) = \max_\pi Q^\pi(a, s) \quad \text{for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \,. \tag{28}$$

Additionally, $Q^*(a, s)$ and $V^*(s)$ have a straightforward relationship:

$$Q^*(a, s) = \mathbb{E}\left[r_{t+1} + \gamma V^*\left(s_{t+1}\right) \mid s_t = s,\ a_t = a\right] \quad . \tag{29}$$

If $Q^*(a, s)$ is known then we can easily take the optimal policy π^* as

$$\pi^* = \arg\max_a Q^*(a, s) \tag{30}$$

or equivalently by using Equation (29)

$$\pi^* = \arg\max_a \mathbb{E}\left[r_{t+1} + \gamma V^*\left(s_{t+1}\right) \mid s_t = s,\ a_t = a\right] \tag{31}$$

therefore, it follows that finding the optimal value functions is a sufficient condition for us to find the optimal policy π^*. A crucial step in finding these optimal value functions for a given Markov Decision Process is finding their Bellman equations.

The Bellman equations for $V^*(s)$ and $Q^*(a, s)$ are named *Bellman optimality equations*. The intuition behind these equations comes from the idea that the value of a state under the optimal policy must be equal to the expected total reward for the best action from that state. We can derive the Bellman

optimality equations as follows:

$$V^*(s) = \max_{a \in \mathcal{A}(s)} Q^{\pi^*}(a, s) \tag{32}$$

$$= \max_a \mathbb{E}_{\pi^*} \left[R_t \mid s_t = s, a_t = a \right] \tag{33}$$

$$= \max_a \mathbb{E}_{\pi^*} \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right] \tag{34}$$

$$= \max_a \mathbb{E}_{\pi^*} \left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s, a_t = a \right] \tag{35}$$

$$= \max_a \mathbb{E}_{\pi^*} \left[r_{t+1} + \gamma V^* (s_{t+1}) \mid s_t = s, a_t = a \right] \tag{36}$$

$$= \max_a \sum_{s'} \mathcal{T}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^* (s') \right] \quad , \tag{37}$$

and for $Q^*(a, s)$ we have

$$Q^*(a, s) = \mathbb{E} \left[r_{t+1} + \gamma \max_{a'} Q^* (a', s_{t+1}) \mid s_t = s, a_t = a \right] \tag{38}$$

$$= \sum_{s'} T^a_{ss'} \left[R^a_{ss'} + \gamma \max_{a'} Q^*(a', s') \right] \tag{39}$$

where the optimal value functions are fixed points in these Bellman equations. The next section on Policy Iteration and Value Iteration will demonstrate that we can build algorithms that turn these Bellman equations into update rules that successfully converge to the optimal value functions. As we have shown in equations (30) and (31), with these resulting optimal value functions we can construct the optimal policy π^* of the Markov Decision Process, which in practice is the ultimate goal for engineers aiming to build intelligent systems.

2.1.4 Policy Iteration

The Policy Iteration algorithm consists of two major parts, Policy evaluation and Policy improvement. Policy evaluation takes the current policy π and outputs the corresponding value function $V^\pi(s)$. The idea is to begin with an arbitrary guess for $V^\pi(s)$, and then we formulate an update rule using the Bellman equation derived in Section (2.1.3)

$$V_{k+1}(s) = \mathbb{E}_\pi \left[r_{t+1} + \gamma V_k (s_{t+1}) \mid s_t = s \right] \tag{40}$$

$$= \sum_a \pi(a, s) \sum_{s'} \mathcal{T}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V_k (s') \right] \tag{41}$$

as the value function $V^\pi(s)$ is a fixed point of its Bellman equation this update rule is guaranteed to converge if $\gamma < 1$. In the case that $\gamma = 1$, the value of states can become infinite as the agent will equally weigh the reward from each time step over a potentially infinite time scale. If the termination of the MDP

is guaranteed then the condition that $\gamma < 1$ is no longer necessary.

Equipped with a sensible stopping criterion of subsequent estimations of the value functions becoming very close i.e. $\mid V_{k+1}(s) - V_k(s) \mid < \epsilon$, we arrive at the Policy evaluation algorithm:

Algorithm 1 Policy Evaluation

Input : π
Initialize $V(s) = 0$, for all $s \in \mathcal{S}$
While $\Delta > \epsilon$:
    $\Delta \leftarrow 0$
    For each $s \in S$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a, s) \sum_{s'} \mathcal{T}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V\left(s'\right)\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
Output : $V \approx V^\pi$

Following Policy evaluation we have Policy improvement. Policy improvement takes our newly found value function $V^\pi(s)$ and outputs an *improved* policy $\pi' \geq \pi$. The motivation behind Policy improvement is to evaluate the expected total reward at each state $s \in \mathcal{S}$ for each action $a \in \mathcal{A}$, and in any case where there exists an action a' that provides a higher expected total reward than the action taken in policy π, we modify the policy to take this 'better' action a'. The steps to achieve this are:

- For each $s \in \mathcal{S}$ and $a \in \mathcal{A}$, compute $Q^\pi(s)$ using Equation (15)
- For each $s \in \mathcal{S}$, set $\pi'(s) = \arg\max_a Q^\pi(s)$
- Output $\pi' \geq \pi$

This works because if for each state $s \in \mathcal{S}$ we have that

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s) \tag{42}$$

then the policy π' must be at least as good as π i.e. for all states $s \in \mathcal{S}$ we have

$$V^{\pi'}(s) \geq V^\pi(s) \ . \tag{43}$$

We can summarise this within the following theorem

Theorem 2.1. Given a Markov Decision Process with policies π and π' and corresponding state-value functions $V^\pi(s)$ and $V^{\pi'}(s)$ then $\forall s \in \mathcal{S}$,

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s) \implies V^{\pi'}(s) \geq V^\pi(s) \tag{44}$$

We can prove this by starting with Equation (42) and iteratively applying Equation (14) and Equation (42) :

Proof.

$$
\begin{aligned}
V^{\pi}(s) &\leq Q^{\pi}\left(s, \pi'(s)\right) \\
&= \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma V^{\pi}\left(s_{t+1}\right) \mid s_t = s\right] \\
&\leq \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma Q^{\pi}\left(s_{t+1}, \pi'\left(s_{t+1}\right)\right) \mid s_t = s\right] \\
&= \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma \mathbb{E}_{\pi'}\left[r_{t+2} + \gamma V^{\pi}\left(s_{t+2}\right)\right) \mid s_t = s\right] \\
&= \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 V^{\pi}\left(s_{t+2}\right) \mid s_t = s\right\} \\
&\leq \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V^{\pi}\left(s_{t+3}\right) \mid s_t = s\right] \\
&\vdots \\
&\leq \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots \mid s_t = s\right] \\
&= V^{\pi'}(s)
\end{aligned}
$$

□

Therefore, each application of Policy improvement guarantees that the resulting policy π' is at least as good as π. Furthermore, we can show that after Policy improvement, if $\pi = \pi'$, then it follows that we must have achieved the optimal policy i.e. $\pi = \pi^*$.

Theorem 2.2. Given a Markov Decision Process with policy π and $\pi' := \arg\max_a Q^{\pi}(a, s)$ then

$$\pi = \pi' \implies \pi = \pi^*$$

where $\pi*$ denotes the optimal policy of the Markov Decision Process

Proof. Suppose that after Policy improvement $\pi = \pi'$. This directly implies that $V^{\pi} = V^{\pi'}$. By definition of π' and Equation (14) we have

$$
\begin{aligned}
\pi'(s) &= \arg\max_a Q^{\pi}(a, s) && (45)\\
&= \arg\max_a \mathbb{E}\left[r_{t+1} + \gamma V^{\pi}\left(s_{t+1}\right) \mid s_t = s,\ a_t = a\right] && (46)
\end{aligned}
$$

From this we get that

$$V^{\pi'}(s) = \max_a \mathbb{E}\left[r_{t+1} + \gamma V^{\pi}\left(s_{t+1}\right) \mid s_t = s, a_t = a\right] \tag{47}$$

and from our assumption that $\pi = \pi'$ we get

$$
\begin{aligned}
V^{\pi'}(s) &= \max_a \mathbb{E}\left[r_{t+1} + \gamma V^{\pi'}\left(s_{t+1}\right) \mid s_t = s, a_t = a\right] && (48)\\
&= \max_a \sum_{s'} \mathcal{T}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V^{\pi'}\left(s'\right)\right] && (49)
\end{aligned}
$$

Here we notice that we have arrived at the Bellman equation for the optimal value function $V^*(s)$ derived in Section (2.1.3). As V^{π} also satisfies this Bellman equation, it follows that $V^{\pi}(s) = V^*(s)$, and consequently $\pi = \pi^*$.

□

It has now been shown that Policy improvement is guaranteed to either improve the currently policy, or tell us that we have already reached the optimal policy.

To summarise what we have discussed so far, Policy evaluation takes the current policy π and outputs the corresponding value function $V^{\pi}(s)$, and Policy improvement takes our newly found value function $V^{\pi}(s)$ and and outputs an *improved* policy $\pi' \geq \pi$. Therefore by continually applying Policy evaluation and Policy improvement one after another, we will eventually arrive at the optimal state value function $V^{*}(s)$ and also the optimal policy π^{*}. This procedure is called Policy Iteration, and is a very significant algorithm in the field of Reinforcement Learning. The stopping criterion for this algorithm is when Policy improvement indicates that it has found the optimal policy π^{*}. Figure (5) below gives an illustration of the algorithm, and the pseudo-code for Policy Iteration can also be seen in Algorithm (2).

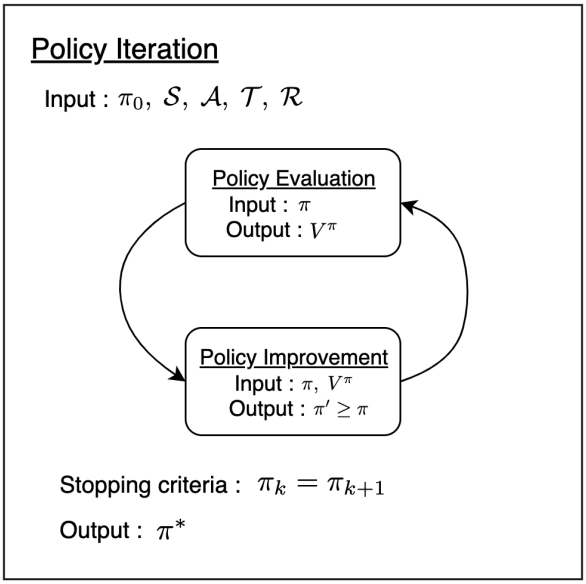


Figure 5: Illustration of the Policy Iteration algorithm. The Policy Iteration algorithm begins with a Markov Decision Process and initial policy π_0, and by continually applying Policy evaluation and Policy improvement, returns the optimal policy π^{*}.

The high-level idea behind Policy Iteration is that a better understanding of the current policy proficiency allows for a more effective improvement of the policy. For example, when studying a new topic or learning to play an instrument - initially, you are unaware of what you do not know; this can make it difficult to formulate a learning strategy. With practice, you eventually get a good idea of what you currently know, and what other useful knowledge is out there to be learned in the future. With this newfound knowledge of your ability in the chosen topic, it is much easier to plan and improve the skill effectively.

Algorithm 2 Policy Iteration

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   While $\Delta > \epsilon$:
      $\Delta \leftarrow 0$
      For each $s \in S$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{T}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V\left(s'\right)\right]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

3. Policy Improvement
   policy-stable $\leftarrow$ true
   For each $s \in \delta$:
      $b \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{T}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V\left(s'\right)\right]$
      If $b \neq \pi(s)$, then policy-stable $\leftarrow$ false
   If policy-stable, then stop; else go to 2

Policy Iteration takes advantage of this idea to accelerate learning, by balancing between learning what there is to know and improving the policy based on this knowledge.

To complete our section on Markov Decision Processes, we will discuss a special case of Policy Iteration, called Value Iteration, and then we will use this algorithm to find the optimal policy for our toymaker example.

2.1.5 Value Iteration

The motivation behind Value Iteration is that there is a trick we can use to improve the efficiency of Policy Iteration. Within each iteration of Policy Iteration we have to wait for convergence of the state-value function $V^{\pi}(s)$ within the Policy evaluation step. We can speed up Policy Iteration by making the stopping criterion of the Policy evaluation step less strict. As a result, the Policy evaluation stopping criterion triggers earlier, and although the approximation of the state-value function $V^{\pi}(s)$ is slightly less accurate, a lot of time is saved. In contrast to the Policy evaluation step the Policy improvement step is not iterative, it is a sequence of operations that only needs to happen once for each iteration of Policy Iteration. Value Iteration takes this concept of having a less strict Policy evaluation stopping criterion to the extreme, where only a single step of Policy evaluation is permitted before moving straight onto Policy improvement. In practice, Value Iteration is much more efficient at finding optimal policies than regular Policy Iteration. Further to this change in the Policy evaluation step, in Value Iteration it is possible to combine both the Policy evaluation step and the Policy improvement step into one update rule. This can be seen in pseudo-code in Algorithm (3).

Algorithm 3 Value Iteration

Initialize V arbitrarily, e.g., $V(s) = 0$, for all $s \in \mathcal{S}$
While $\Delta > \epsilon$:
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s'} \mathcal{T}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V\left(s'\right)\right]$
        $\Delta \leftarrow \max(\Delta,\ |v - V(s)|)$
Output a deterministic policy, π, such that
$\pi(s) = \arg\max_a \sum_{s'} \mathcal{T}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V\left(s'\right)\right]$

Now that we have introduced the Value Iteration algorithm, we can apply it to the toymaker example. The results of Value Iteration applied to the toymaker example can be seen in Figure (6). The Value Iteration algorithm was implemented in Python to create this figure, the code can be found on Github at `https://github.com/alex21347/Value_Iteration_Algorithm`. As one might suspect, the expected reward for the toymaker is greater if currently has a successful toy instead of an unsuccessful one. This can be see in the graph as after convergence $V^{\pi}(1) > V^{\pi}(2)$. The good news for the toymaker is that regardless of whether his current toy is successful, he is expected to be profitable in the long run. This can be seen as both $V^{\pi}(1) > 0$ and $V^{\pi}(2) > 0$. Furthermore, we can now determine from this state-action function whether the toymaker should advertise his toys to the local area.

Furthermore, using our state-value function values $V^{\pi}(s)$ and Equation (15) we can also find the action-value function $Q^{\pi}(a, s)$ for our toymaker. These

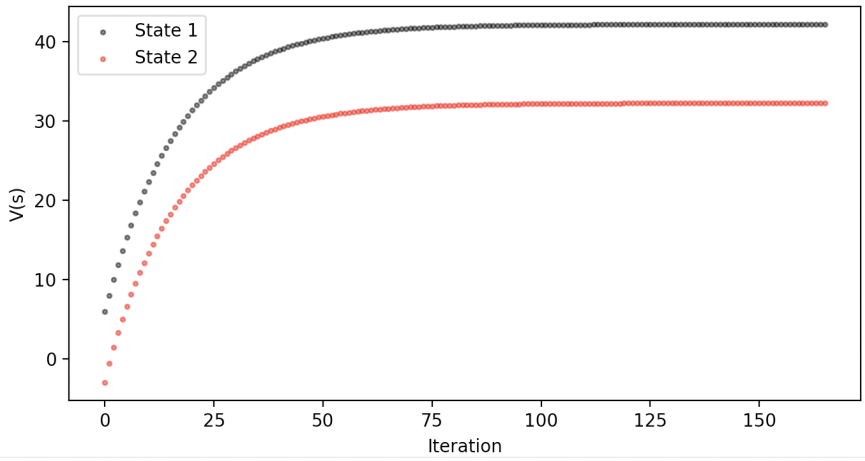


Figure 6: The state-value function being trained via the Value Iteration algorithm. Here we have two states from a simple toymaker example. After approximately 80 iterations the state-value function converges.

values can be seen below in Table (1).

State \ Action	No Advertising	Advertising
Successful Toy (1)	41.37	**42.21**
Unsuccessful Toy (2)	31.42	**32.26**

Table 1: Action-value function $Q^{\pi}(a, s)$ values for the toymaker example with 2 states and 2 actions, after convergence of the value iteration algorithm. The optimal actions for each state are in bold.

From the values in Table (1) we can estimate that on average the toymaker will make \$84 more overall if he chooses to advertise (\$4221 - \$4137), regardless of his current situation. Therefore, the optimal policy for the toymaker is to always advertise his toys.

Now that we have discussed methods of finding optimal policies for Markov Decision Processes, we will move onto more advanced methods of finding optimal policies where less information is known about the environment.

2.2 Classical Reinforcement Learning

In contrast to the previous section, from this section onward we will discuss methods of finding optimal policies where the transition probabilities $\mathcal{T}^a_{ss'}$ and reward probabilities $\mathcal{R}^a_{ss'}$ are not known beforehand, and thus information about the environment must be learned through exploration. Learning an optimal policy in this setting is a harder problem to solve as there is less information available to take advantage of. There are two major categories of Reinforcement Learning algorithms that are defined by their response to this particular problem, *model-free* algorithms and *model-based* algorithms.

Model-based Reinforcement Learning algorithms aim to build a model of the environment through repeated exploration. The transition probabilities $\mathcal{T}^a_{ss'}$ and the reward function $\mathcal{R}^a_{ss'}$ can be estimated through this exploration. Starting from from a state s_0, at each subsequent time step we allow the agent to chose random actions within the action space $\mathcal{A}$. During this process, at each time step t we can collect the samples (s_t, a_t, s_{t+1}) and the respective reward collected $\mathcal{R}_t$. After n steps, we can estimate the transition probabilities and rewards as

$$\widehat{\mathcal{T}}_n\left(s,a,s'\right) = \frac{\sum_t \mathbb{1}_{(s,a,s')}\left\{s = s_t,\ a = a_t,\ s' = s_{t+1}\right\}}{\sum_t \mathbb{1}_{(s,a,s')}\left\{s = s_t,\ a = a_t\right\}} \tag{50}$$

and

$$\widehat{\mathcal{R}}_n\left(s,a,s'\right) = \frac{\sum_t r_t \cdot \mathbb{1}_{(s,a,s')}\left\{s = s_t,\ a = a_t,\ s' = s_{t+1}\right\}}{\sum_t \mathbb{1}_{(s,a,s')}\left\{s = s_t,\ a = a_t,\ s' = s_{t+1}\right\}} \tag{51}$$

Equation (50) is a result of the frequentist paradigm of statistics where the probability of an event occurring is estimated as the number of times the event occurred out of the total number of possible times it could occur. Similarly, Equation (51) is simply estimating $\mathcal{R}\left(s,a,s'\right)$ by taking the mean reward experienced during exploration of the environment from going from state s to state s' via the action a. By the law of large numbers, as n goes to infinity $\widehat{\mathcal{T}}_n$ and $\widehat{\mathcal{R}}_n$ will converge towards $\mathcal{T}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ respectively.

In direct contract so this, model-free approaches aim to learn policies without an explicit construction of a model, i.e. estimations of $\mathcal{T}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ are no longer necessary. For the remainder of this section, we will discuss the classical *model-free* Reinforcement Learning algorithms: Q-learning, Policy Gradient methods, and Actor-Critic methods.

2.2.1 Q-Learning

The first model-free Reinforcement Learning algorithm we will discuss is Q-learning. Q-learning was first developed by Watkins in 1989 [8], and has since become one of the most widely used Reinforcement Learning algorithms. The core idea is to estimate the action-value function $Q^\pi(a,s)$ though an exploration of the environment.

The algorithm begins by setting $Q^\pi(a,s) = 0$ for all $a \in \mathcal{A}$ and $s \in \mathcal{S}$. It is common to represent each pair (a,s) in a table called a *Q-table*. After initialisation, we allow the agent to explore the environment step by step collecting

samples of (s_t, a_t, s_{t+1}, r_t). After each step we update the value of $Q^\pi(a_t, s_t)$ using the following update rule

$$Q(a_t, s_t) := Q(a_t, s_t) + \alpha \left(r_t + \gamma \max_a Q\left(a_{t+1}, s_{t+1}\right) - Q(a_t, s_t) \right) \tag{52}$$

where α is the learning rate parameter and γ is the discount parameter discussed in Section (2.1.1). This update rule is constructed using Equation (25) which is the Bellman optimality equation for $Q^\pi(a, s)$. To provide a clear overview of the Q-learning algorithm, an illustration of the entire process can be seen in Figure (7). In 1992, Watkins proved in the following theorem that given enough time, Q-learning is guaranteed to converge [17].

Theorem 2.3. Given bounded rewards $|r_n| \leq R$, and learning rates $0 \leq \alpha_n < 1$ that satisfy

$$\sum_{i=1}^{\infty} \alpha_n = \infty, \quad \sum_{i=1}^{\infty} |\alpha_n|^2 < \infty$$

then $Q_n(a, s) \to Q^*(a, s)$ as $n \to \infty$, $\forall (a, s) \in \mathcal{S} \times \mathcal{A}$, with probability 1.

Following the proof of this theorem in 1992, Q-learning became an important and easy-to-use tool for the Reinforcement Learning practitioner looking to solve Reinforcement Learning tasks. For brevity we will not recite the proof in this paper. The curious reader is directed to Watkins' 1992 paper for more details [17].

The final detail which must be discussed is how the agent should explore the environment and collect information. It has been shown that convergence is guaranteed regardless of the way the agent explores the environment [4], however, a well chosen strategy can offer faster convergence. The most common approach is called the *ϵ-greedy* strategy. In this approach, at each time step the agent takes the action with the highest expected reward according to the current $Q(a, s)$ with probability $(1-\epsilon)$, and it takes a completely random action with probability ϵ. This strategy aims to find a balance between maximising reward (exploitation), and discovering new and potentially better policies (exploration). This balancing act is a common problem in Reinforcement Learning and is known as the Exploration vs Exploitation dilemma.

Once the Q-learning algorithm is complete, the optimal policy is simply given by always taking the action in the Q-table that has the highest value for the current state. By definition of $Q^\pi(a, s)$, this will give us the highest possible expected total reward.

Q-learning has proved to be a effective algorithm as solving simple environments, however, if the state space $\mathcal{S}$ becomes very large then it becomes very difficult to adequately fill the corresponding Q-table. In Section (2.3.1) we will discuss an adaption of the Q-learning algorithm that attempts to fix this limitation. Before this, we will discuss another successful method in classical Reinforcement Learning: Policy Gradients.

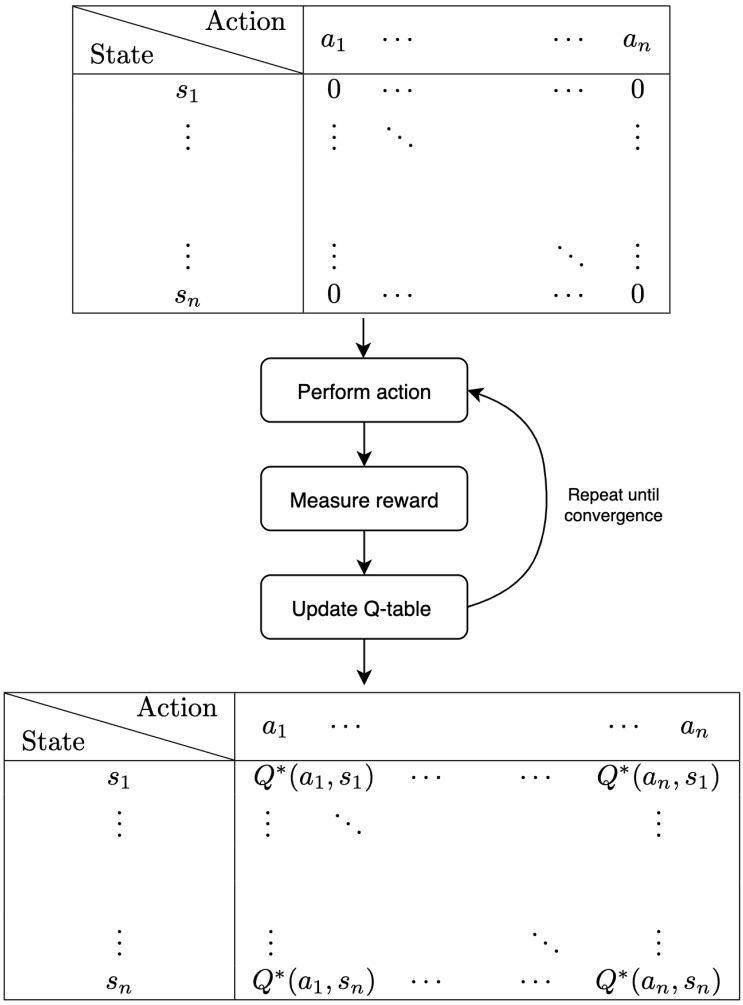


Figure 7: Illustration of the Q-learning algorithm. We begin with an initialised Q-table at the top indicating the state-value function $Q^\pi(a, s)$ of each action $a \in \mathcal{A}$ and state $s \in \mathcal{S}$, and through a repeated 3 step procedure we update our Q-table until we reach convergence. The output is an approximation of the optimal state-value function $Q^\pi(a, s)$.

2.2.2 Policy Gradients

Policy Gradient methods begin with a parameterised policy $\pi_\theta(a, s)$ with parameters $\theta := \{\theta_1, \ldots, \theta_n\}$. These parameters could be anything from the weights of a neural network, to the weights of a Logistic Regression model. Recall that a policy is any function that takes the current state s as input, and outputs a probability distribution over the actions $a \in \mathcal{A}$. Given a policy with parameters θ, we can define the optimal parameters for the policy as

$$\theta^\star = \arg\max_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} r_t\right] \quad . \tag{53}$$

Furthermore, we can define an evaluation metric $J(\theta)$ defined as

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} r_t\right] \tag{54}$$

which measures how effective π_θ is at maximising reward in the environment. Recall that given an initial state distribution $p(s_0)$ we can calculate the probability of realising a trajectory τ with policy π_θ as

$$\pi_\theta(\tau) = \pi_\theta\left(s_0, a_0, \ldots, s_T, a_T\right) = p\left(s_0\right) \prod_{t=0}^{T} \pi_\theta\left(a_t, s_t\right) \mathcal{T}_{s_t s_{t+1}}^{a_t} \quad . \tag{55}$$

The problem is that in practice this formula cannot be directly calculated as we do not know the transition probabilities $\mathcal{T}_{s_t s_{t+1}}^{a_t}$. An important methodological trick behind Policy Gradient methods is to cleverly modify our estimation of the gradient of the evaluation metric $\nabla_\theta J(\theta)$ so that it doesn't rely on unknown variables such as $\mathcal{T}_{s_t s_{t+1}}^{a_t}$ and $\mathcal{R}_{s_t s_{t+1}}^{a_t}$. This way we are able to use only our samples (s_t, a_t, s_{t+1}, r_t) collected during rollout of the policy. Therefore, Policy Gradient methods are extremely flexible as they only require the ability to rollout policies in the environment and collect information. This will soon become more clear as we continue our investigation of this class of methods. Recall that given a trajectory τ of length T, we define the return of this trajectory as

$$r(\tau) = r(s_0, a_0, \ldots, s_T, a_T) = \sum_{t=0}^{T} r_t \quad . \tag{56}$$

Using this and the definition of expectation we can rewrite Equation (54) as

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} r_t\right] = \int \pi_\theta(\tau) r(\tau) d\tau \tag{57}$$

where here we are assuming a continuous state and action space. The discrete case is analogous but instead of integrating over the possible trajectories we take

the sum. For brevity, we continue with the continuous case. By the linearity of the gradient operation we now have that

$$\nabla_\theta J(\theta) = \nabla \int \pi_\theta(\tau) r(\tau) d\tau = \int \nabla \pi_\theta(\tau) r(\tau) d\tau \tag{58}$$

Here we will now utilise a useful property of the logarithm:

$$\nabla \pi_\theta(\tau) = \pi_\theta(\tau) \frac{\nabla \pi_\theta(\tau)}{\pi_\theta(\tau)} = \pi_\theta(\tau) \nabla \log \pi_\theta(\tau) \tag{59}$$

therefore,

$$\nabla_\theta J(\theta) = \int \nabla \pi_\theta(\tau) r(\tau) d\tau = \int \pi_\theta(\tau) \nabla \log \pi_\theta(\tau) r(\tau) d\tau \tag{60}$$

and by using the definition of expectation again we have

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta(\tau)} \left[\nabla \log \pi_\theta(\tau) r(\tau) \right] \quad . \tag{61}$$

This leaves us with the final problem of evaluating this new expectation. To do this we begin with Equation (55) and take the logarithm of both sides

$$\log \pi_\theta(\tau) = \log p\left(s_0\right) + \sum_{t=0}^{T} \log \pi_\theta\left(a_t, s_t\right) + \log \mathcal{T}^{a_t}_{s_t s_{t+1}} \tag{62}$$

and by taking the gradient of both sides we get

$$\nabla_\theta \log \pi_\theta(\tau) = \nabla_\theta \left[\log p\left(s_0\right) + \sum_{t=0}^{T} \log \pi_\theta\left(a_t, s_t\right) + \log \mathcal{T}^{a_t}_{s_t s_{t+1}} \right] \tag{63}$$

$$= \nabla_\theta \log p\left(s_0\right) + \nabla_\theta \sum_{t=0}^{T} \log \pi_\theta\left(a_t, s_t\right) + \nabla_\theta \log \mathcal{T}^{a_t}_{s_t s_{t+1}} \tag{64}$$

$$= \nabla_\theta \sum_{t=0}^{T} \log \pi_\theta\left(a_t, s_t\right) \tag{65}$$

as only the second term of Equation (64) depends on θ. Subbing this result into Equation (61) and using the definition of $r(\tau)$ we arrive at

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[\left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\left(a_t, s_t\right) \right) \left(\sum_{t=0}^{T} \gamma^t r_t \right) \right] \tag{66}$$

where we have now also included the discount parameter γ of the MDP without loss of generality. With this final formulation, $\nabla_\theta J(\theta)$ can be estimated simply by running multiple runs of the policy and averaging over the samples collected. This is because each element of Equation (66) is available to us during

training; we have successfully eliminated our reliance on the unknown transition probabilities $\mathcal{T}^{a_t}_{s_t s_{t+1}}$. If we run the policy π_θ for N times, this gives

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=0}^{N} \left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_{i,t}, s_{i,t} \right) \right) \left(\sum_{t=0}^{T} \gamma^t r_{i,t} \right) \tag{67}$$

With this final equation we can estimate $\nabla_\theta J(\theta)$ using only samples (s_t, a_t, s_{t+1}, r_t). This is extremely convenient as we can now formulate a gradient ascent update rule

$$\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_\theta J(\theta_k) \tag{68}$$

which aims to improve the policy π_θ over time with respect to our evaluation metric $J(\theta)$. In this update rule, α is called the learning rate and indicates the step-size in the parameter space between subsequent iterations. This is the main idea of the original Policy Gradient algorithm invented by Williams in 1987, named REINFORCE [9]. The pseudo-code for REINFORCE can be seen below in Algorithm (4).

Algorithm 4 REINFORCE Algorithm

Initialise policy π_θ randomly
While $|\theta_k - \theta_{k-1}| > \epsilon$:
    Use π_θ for N trajectories and collect samples (s_t, a_t, s_{t+1}, r_t)
    Estimate gradient with

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=0}^{N} \left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_{i,t}, s_{i,t} \right) \right) \left(\sum_{t=0}^{T} \gamma^t r_{i,t} \right)$$

    Update policy : $\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_\theta J(\theta_k)$
Output improved policy π_θ

As with all gradient ascent/descent methods, the problem exists that the policy π_θ could converge to a local maximum instead of the global maximum. In general, the techniques used in optimisation theory to overcome this problem can also be used in this context.

REINFORCE can also be modified to improve performance in many ways. In particular, there exists better estimates for $\nabla_\theta J(\theta)$ than the one that we have derived in this section. These more sophisticated methods will be explored in the next section on Actor-Critic Methods, where we combine the Policy Gradient method with value-function based methods such as Q-learning. For more information on Policy Gradient methods the reader would benefit from Baxter and Bartlett's 1999 book on Direct Gradient-based Reinforcement Learning [18, 19].

2.2.3 Actor-Critic Methods

Actor-Critic methods combine the Policy Gradient methodology with value-function estimation. In literature, Reinforcement Learning algorithms are often categorised as either *value-based* (e.g. Q-learning) which is centered around estimating value-functions, or *policy-based* (e.g. REINFORCE) which directly optimises the policy using gradient ascent. Actor-Critic methods are both policy-based and value-based simultaneously. This is because Actor-Critic methods approximate the gradient of the evaluation metric $\nabla_\theta J(\theta)$ *using* the value-functions we defined in Section (2.1.2).

Recall that for REINFORCE, we approximate $\nabla_\theta J(\theta)$ using

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta(\tau)} \left[\nabla_\theta \log \pi_\theta(\tau) r(\tau) \right] \tag{69}$$

$$\approx \frac{1}{N} \sum_{i=0}^{N} \left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_{i,t}, s_{i,t} \right) \right) \left(\sum_{t=0}^{T} \gamma^t r_{i,t} \right) \quad . \tag{70}$$

The first insight we can use to improve this estimate is that within the environment there is causality. So far we have not utilised the fact that we are working with a sequential decision process that evolves through time linearly. Specifically, Equation (70) does not take into account that future actions cannot effect past rewards. Using this fact that only future rewards can be effected by the current state and action, we arrive at the better estimate:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=0}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_{i,t}, s_{i,t} \right) \left(\sum_{t'=t}^{T} \gamma^{t'} r_{i,t'} \right) \tag{71}$$

which can be shown to be an unbiased estimator for $\nabla_\theta J(\theta)$ with a lower variance than Equation (70). Intuitively, once can imagine an estimator that depends on strictly less random variables (as $r_{t'}$ for $t' < t$ are no longer used) would have less random noise in practice. For brevity, we will not cover the proof of this in this paper.

By definition, we have

$$Q^\pi(a_t, s_t) = \mathbb{E} \left[\sum_{t'=t}^{T} \gamma^{t'} r_{i,t'} \right] \tag{72}$$

which means we can rewrite Equation (71) as

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=0}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_{i,t}, s_{i,t} \right) \hat{Q}^\pi(a_t, s_t) \tag{73}$$

where

$$\hat{Q}^\pi(a_t, s_t) = \sum_{t'=t}^{T} \gamma^{t'} r_{i,t'} \quad . \tag{74}$$

This gives us our first estimate for $\nabla_\theta J(\theta)$ calculated using the action-value function $Q^\pi(a,s)$. Recall from Section (2.1.2) that the advantage value-function $A^\pi(a,s)$ is defined as

$$A^\pi(a,s) = Q^\pi(a,s) - V^\pi(s) \quad . \tag{75}$$

It turns out that you can also use the advantage value-function $A^\pi(a,s)$ to estimate $\nabla_\theta J(\theta)$:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=0}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_{i,t}, s_{i,t}\right) \hat{A}^\pi(a_t, s_t) \tag{76}$$

and this estimate turns out to be even better than using $Q^\pi(a,s)$.

A convenient trick we can use at this stage is to rewrite both A^π and Q^π in terms of V^π:

$$Q^\pi\left(a_t, s_t\right) = r_t + \sum_{t'=t+1}^{T} \mathbb{E}_{\pi_\theta}\left[\gamma^{t'-t} r_{t'} \mid s_{t'}, a_{t'}\right] \tag{77}$$

$$\approx r_t + V^\pi\left(s_{t+1}\right) \tag{78}$$

and subbing this into Equation (75) gives us a corresponding approximation for $A^\pi(a,s)$

$$A^\pi\left(a_t, s_t\right) \approx r\left(s_t, a_t\right) + V^\pi\left(s_{t+1}\right) - V^\pi\left(s_t\right) \quad . \tag{79}$$

It follows from this that in order to use any of the value-functions to approximate $\nabla_\theta J(\theta)$ all we need is just an estimation of V^π and we can use Equation (78) and Equation (79) to calculate the other value-functions.

There are multiple ways of finding good estimates of V^π, however, most modern algorithms use a neural network. This neural network takes the state s and outputs the corresponding $V^\pi(s)$. To train this neural network we can collect estimates during rollout of the current policy as

$$V^\pi(s_t) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{80}$$

then use these estimates as labels for the neural network. Thus we have the training data: $\{s_{i,t},\ \sum_{t'=t}^{T} \gamma^{t'-t} r_{i,t'}\}$. We can then use the loss function

$$\mathcal{L}(\psi) = \frac{1}{2} \sum_i \left\| \hat{V}_\psi^\pi\left(s_i\right) - \sum_{t'=t}^{T} \gamma^{t'-t} r_{i,t'} \right\|^2 \tag{81}$$

where ψ represent the weights of the network. Through many cycles of the data set, this neural network can learn to very accurately approximate $V^\pi(s)$. Furthermore, this method offers a stronger robustness to predicting the value of states unseen in the training data, in this way the neural network has a better generalisation ability. For more details on neural networks, please see *Deep*

Learning by Goodfellow and LeCun [20].

Once V^π has been estimated with the neural network, either A^π or Q^π can be found soon after as we have discussed. Figure (8) below shows the four main steps of an Actor-Critic algorithm.

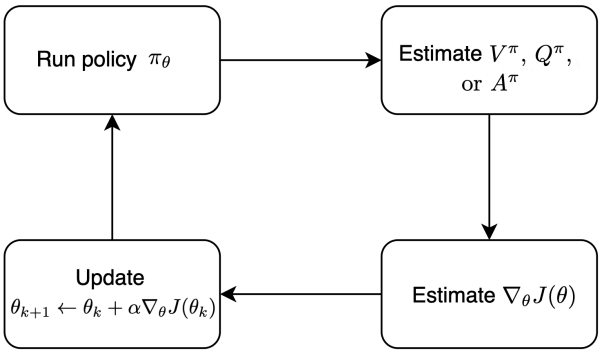


Figure 8: Illustration of the Actor-Critic methodology of solving Reinforcement Learning Tasks. The policy π_θ is improved via gradient ascent, where the gradient of the evaluation metric $J(\theta)$ is calculated using an estimate of one of the value-functions.

We can summarise the Policy Gradient and Actor-Critic methods we have discussed so far by their estimation of $\nabla_\theta J(\theta)$:

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(\tau) \cdot r(\tau)\right] && \text{REINFORCE} \\ &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(\tau) \cdot Q^\pi(a,s)\right] && \text{Q Actor-Critic} \\ &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(\tau) \cdot A^\pi(a,s)\right] && \text{Advantage Actor-Critic (A2C)} \end{aligned}$$

Now that we have discussed the classical methods of Reinforcement Learning, we will turn our attention to Deep Reinforcement Learning algorithms, which have been shown to be highly effective in many advanced applications and contexts.

2.3 Deep Reinforcement Learning

Classical Reinforcement Learning algorithms have achieved impressive performance in many applications since its conception in the 1950s, however, they have been mostly limited to applications with relatively small state spaces [21]. These classical methods have been considerably improved since the development of other advances in Artificial Intelligence, such as Deep Learning. In this section we will discuss two state of the art Reinforcement Learning methods that have considerably impacted the field: Deep Q-Networks (DQN) and Proximal Policy Optimisation (PPO).

2.3.1 Deep Q-Networks

Deep Q-Networks (DQNs) were introduced by Mnih et al. in 2013 in their paper *Playing atari with deep reinforcement learning* [10]. DQNs are an adaptation of Q-learning where instead of storing a Q-table with every possible combination of $Q^\pi(a, s)$, we use a neural-network which directly estimates $Q^\pi(a, s)$ using a and s as inputs. Storing a table of Q-values can even become impossible for large state spaces. For example, imagine we wanted to train an agent where the state space is the screen of a video game with (160×192) grey-scale pixels with values in $\{0, \ldots, 255\}$. Regardless of the number of actions in the action space, the total number of possible states alone that we would need to store in a Q-table is

$$256^{160\times 192} = 2^{245760} \approx 10^{81920} \quad . \tag{82}$$

This means we would need 10^{81920} rows in our Q-table, which is more than the number of atoms in the observable universe. Therefore, Q-learning is simply not a feasible method in this case. On the other hand, neural networks with as many as (160×192) input variables are commonly applied in our modern era [22].

First, we design a neural network called the *prediction network* to take the current state and action (a, s) as input, and outputs the action-value function $Q(a, s; \theta)$ where we now include the θ parameter to represent the parameters of the neural network. The neural network architecture can chosen based on the nature of the state-space, as long as it can effectively transform the input (a, s) into a lower-dimensional representation. For images, the CNN architecture has proved very effective at this task[21]. The first step in training the prediction network is collecting information from the environment in the form of samples (s, a, s', r). Similarly to Q-learning, we can use the ϵ-greedy strategy to collect these samples. Then using these samples we can compute target values for our prediction network as

$$y_i = r_i + \gamma \max_{a'_i} Q\left(s'_i, a'_i; \theta\right) \quad . \tag{83}$$

We can then use gradient descent equipped with the following loss function:

$$L_i\left(\theta_i\right) = \mathbb{E}\left[\left(y_i - Q\left(a, s; \theta_i\right)\right)^2\right] \tag{84}$$

with derivative

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}\left[\left(y_i - Q\left(s, a; \theta_i\right)\right) \nabla_{\theta_i} Q\left(s, a; \theta_i\right)\right] \tag{85}$$

to update our network.

Unfortunately, this simple setup has been shown to be unstable when a nonlinear model such as a neural network is used to represent the action-value function $Q(a, s; \theta)$. There are two main causes for this instability:

1) *Statistical independence of data points*

An underlying assumption of many statistical models is independence of data points within the data set. Neural networks are no exception. When we collect the samples (s_t, a_t, s_{t+1}, r_t) during exploration of the environment, any two subsequent samples are highly dependent on one another. In particular, the ending state at time t is the same as the starting state at time $t+1$. Therefore the statistical independence assumption does not hold in the current setup.

2) *Moving target during training*

During training the prediction network, for each (a, s) collected in the environment we compute the target values as $y_i = r_i + \gamma \max_{a'_i} Q\left(s'_i, a'_i; \theta\right)$. However, as the prediction network trains, the parameters θ change, and thus the target values also change as a result. Therefore after each update of the prediction network, the *targets* of the network is also updated. This causes instability because regression is not perfectly well-defined on target values that consistently change in distribution each iteration.

The researchers in Google's DeepMind team who developed the Deep Q-Network identified these two aforementioned issues and designed two additional features to combat these issues [10].

The first feature is called *experience replay* and aims to mitigate the lack of statistical independence between subsequent samples in the environment. The idea is to collect a large number of samples from the environment into a data set called a *replay buffer*. Then at each iteration of training the neural network, instead of taking the last collected data points from the environment, you randomly sample them the replay buffer. If the replay buffer is large enough then the probability of collecting two highly dependent data points can be made arbitrarily small.

The second feature is the use of a *target network*. Every N updates of the prediction network, the target network is constructed by taking the exact architecture as the prediction network and freezing its parameters. Therefore, at the moment the target network is updated it is identical to the prediction network, but then for the next $N-1$ iterations of the prediction network the target network remains unchanged. With the addition of the target network, we now compute our target values as

$$y_i = r_i + \gamma \max_{a'_i} Q\left(s'_i, a'_i; \theta^-\right) \quad . \tag{86}$$

where θ^- represents the parameters of the target network. This ensures that the target values do not change for N steps of the prediction network, therefore mitigating the moving target issue.

Using the same loss function but with our new target values, we can begin to train our Deep Q-network using the update rule

$$\theta_{k+1} = \theta_k - \alpha \nabla_{\theta_i} L_i\left(\theta_i\right) \quad . \tag{87}$$

With the target network and replay buffer implemented, have all the key ingredients of a successful DQN. An illustration with all of the important features we have discussed in this section can be seen in Figure (9), including the data collection in the environment, and the training of the neural network.

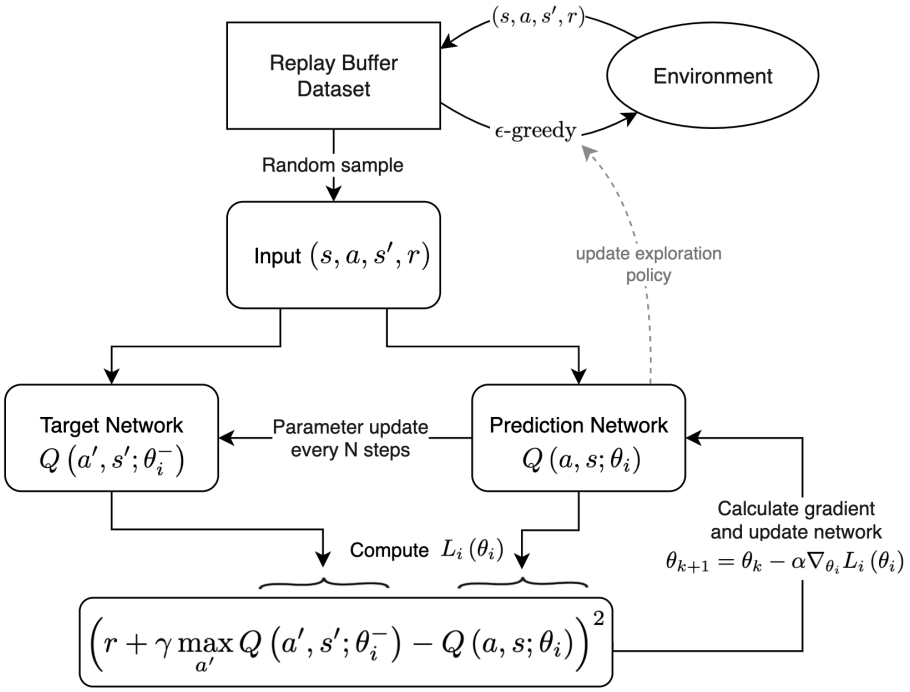


Figure 9: Illustration of the Deep Q-Network methodology, from data collection in the environment to computation of the loss function and training of the prediction neural network.

2.3.2 PPO & TRPO

Proximal Policy Optimisation (PPO) was designed in 2017 by OpenAI and aimed to solve Reinforcement Learning problems with greater stability and reliability than previous state of the art algorithms [12]. PPO falls under the same category as Policy Gradient and Actor-Critic methods discussed in Section (2.2.2), however it has multiple important modifications which massively improves performance in practice.

As discussed previously, some commonly used Policy Gradient loss functions are

$$
\begin{aligned}
J(\theta) \quad &= \mathbb{E}_{\pi_\theta}\left[\log \pi_\theta(\tau) \cdot r(\tau)\right] && \text{REINFORCE} \\
&= \mathbb{E}_{\pi_\theta}\left[\log \pi_\theta(\tau) \cdot Q^\pi(a, s)\right] && \text{Q Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\log \pi_\theta(\tau) \cdot A^\pi(a, s)\right] && \text{Advantage Actor-Critic (A2C)}
\end{aligned}
$$

where the A2C loss function has seen particular success. However, it has been shown that these loss functions lead to destructively large policy updates. In these destructive policy updates a huge amount of training progress can be lost, this is also referred to as *catastrophic forgetting*, and plagued many early Reinforcement Learning practitioners. PPO begins to solve the problem of policy updates that are too large by using the idea of *trust regions*. Trust regions were first introduced by Schulman et al. in 2015 when the Trust Region Policy Optimization (TRPO) algorithm was proposed [11]. The idea behind trust regions is to dynamically constrain the size of policy updates during training so that the probability of large policy updates is significantly reduced. In particular, Kullback–Leibler divergence (KL divergence) is used to measure the distance between two probability distributions, and then if the KL divergence between the old policy $\pi_{\theta_{\text{old}}}$ and the current policy π_θ is greater than a certain threshold δ, then we reject the policy update.

Instead of the loss functions we have already seen, TRPO aims to maximise

$$\mathbb{E}_t \left[\frac{\pi_\theta \left(a_t, s_t\right)}{\pi_{\theta_{\text{old}}} \left(a_t, s_t\right)} \hat{A}_t \right] \quad . \tag{88}$$

Recall that the advantage value function $\hat{A}(a_t, s_t)$ indicates the advantage gained by taking action a_t in state s_t compared to the average. Therefore, if $A^\pi(a_t, s_t) > 0$ then taking the action a_t is better than average and if $A^\pi(a_t, s_t) < 0$ then taking the action a_t is worse than average. Let us denote $r_t(\theta) := \frac{\pi_\theta(a_t, s_t)}{\pi_{\theta_{\text{old}}}(a_t, s_t)}$, so $r(\theta_{\text{old}}) = 1$. Given a sequence of sampled actions and states, $r_t(\theta)$ will be greater than 1 if the action a_t is more probable for the current policy π_θ than it is for the old policy $\pi_{\theta_{\text{old}}}$. Additionally, $r_t(\theta)$ will be less than 1 when the action a_t is less probable for our current policy than the old policy. Therefore, Equation (88) is greater in value when favourable actions (indicated by $A^\pi(a_t, s_t)$) become more likely in the subsequent policy update (indicated by $r_t(\theta)$), and equally Equation (88) is lesser in value when favourable actions (indicated by $A^\pi(a_t, s_t)$) become less likely in the subsequent policy update (indicated by $r_t(\theta)$). Now, by also including the trust region constraint we have the TRPO objective function:

$$\underset{\theta}{\text{maximize}}\ \mathbb{E}_t \left[\frac{\pi_\theta \left(a_t, s_t\right)}{\pi_{\theta_{\text{old}}} \left(a_t, s_t\right)} \hat{A}_t \right] \tag{89}$$

$$\text{subject to } \mathbb{E}_t \left[\text{KL} \left[\pi_{\theta_{\text{old}}} \left(\cdot \mid s_t\right), \pi_\theta \left(\cdot \mid s_t\right) \right] \right] \leq \delta \ . \tag{90}$$

Proximal Policy Optimization attempts to simplify the optimization process whilst retaining the advantages of TRPO [12]. The objective function of TRPO without the trust region constraint can also be written as

$$L^{CPI}(\theta) = \mathbb{E}_t \left[\frac{\pi_\theta \left(a_t, s_t\right)}{\pi_{\theta_{\text{old}}} \left(a_t, s_t\right)} \hat{A}_t \right] = \mathbb{E}_t \left[r_t(\theta) \hat{A}_t \right] \tag{91}$$

where the CPI refers to 'conservative policy iteration', originally proposed by Kakade in 2002 [23]. The PPO objective function is very similar to $L^{CPI}(\theta)$,

however, there is a clipping function that removes the incentive for the agent to make larger updates to the policy during training. This massively improves the stability of training. The objective function used is:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{92}$$

where ϵ is a hyper-parameter to adjust the size of the trust regions. Taking the minimum between the clipped and unclipped objective makes it so the final objective is a lower bound (i.e. a pessimistic bound) on the unclipped objective. It has been shown that $\epsilon = 0.2$ is a sensible first choice in practice [12]. The clipping function is defined as

$$\text{clip}(x, a, b) = \begin{cases} a & \text{if } x > a \\ x & \text{if } x \in [a, b] \\ b & \text{if } x < b \end{cases} .$$

This objective function retains the same advantages as TRPO but it also avoids the high computational cost and implementation complexity [24]. An illustration of the effect of the clipping on this objective function can be seen in Figure (10).

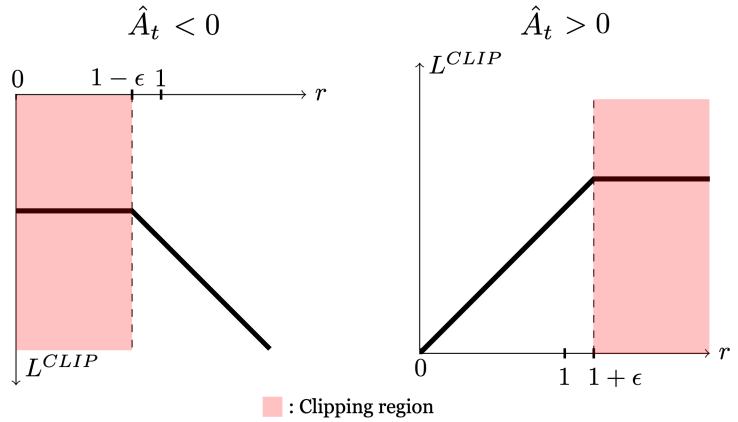


Figure 10: Plots showing one term (a single time step) of the objective function L^{CLIP} as a function of $r_t(\theta)$, for $\hat{A}_t < 0$ (left) and $\hat{A}_t > 0$ (right). The values of $r_t(\theta)$ in which the value of L^{CLIP} is clipped is highlighted in red. This clipping removes the incentive for the agent to make an even larger update to the policy during training.

It can be seen in Figure (10) that once the policy update reaches a certain size (indicated by how far $r_t(\theta)$ is from 1), there is no longer an increase in the objective function $L^{CLIP}(\theta)$. Therefore, it is clear that once the policy update gets clipped there is no incentive for the algorithm to make greater changes to the policy in the current update step. With this new objective function, PPO

can train using the same methodology as other Actor-Critic algorithms. This can be seen in Figure (11).

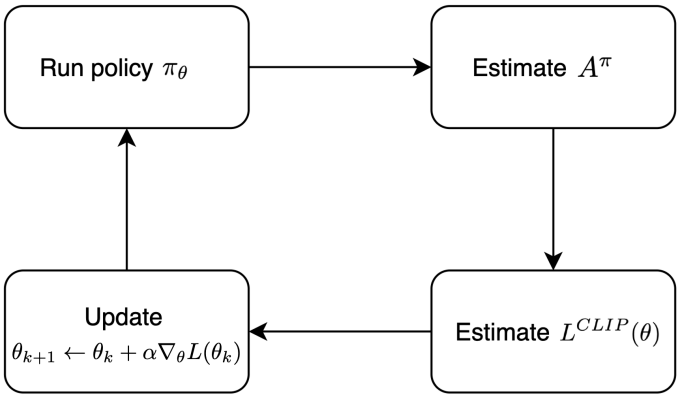


Figure 11: Illustration of the Actor-Critic methodology of solving Reinforcement Learning tasks where the objective function is from PPO. The policy π_θ is improved via gradient ascent, where the gradient of the evaluation metric $L^{CLIP}(\theta)$ is calculated using an estimate of the advantage value function A^π.

Furthermore, in order to concretely display the algorithm in full, example pseudo-code of PPO can also been seen below in Algorithm (5). It should be noted here that the pseudo-code provided is just one version of PPO; for example, other gradient ascent algorithms (e.g. RMSProp, Adam, AdaGrad) can be used to update the policy network and value function neural network. Despite this, the pseudo-code in Algorithm (5) gives a clear overview of how PPO can be implemented.

Proximal Policy Optimisation is one of the most influential Reinforcement Learning algorithms that has been designed in recent years, and has shown countless potential in solving Reinforcement Learning tasks. As a result of the reliability and ease of use of PPO, it shall be the main Reinforcement Learning algorithm used in Section (4), where we shall test different Reinforcement Learning algorithms on a self-driving car application. Before this, we shall first discuss the paradigm of Imitation Learning, and the use of guide policies.

Algorithm 5 PPO Algorithm

Initialise policy π_θ randomly
Initialise V^π neural network weights ψ randomly
For $k = 0, 1, 2, \ldots$:
Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ using π_{θ_k}
Compute $\hat{A}_t \approx r\left(s_t, a_t\right) + \hat{V}\left(s_{t+1}\right) - \hat{V}\left(s_t\right)$
Estimate PPO objective function using collected trajectories:

$$L^{CLIP}(\theta_k) \approx \frac{1}{|\mathcal{D}_k|\, T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(r_t(\theta_k)\hat{A}_t, \operatorname{clip}\left(r_t(\theta_k), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)$$

Update policy : $\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_\theta L^{CLIP}(\theta_k)$
Estimate mean square error of V^π using collected trajectories:

$$\mathcal{L}(\psi_k) \approx \frac{1}{|\mathcal{D}_k|} \sum_{i=1}^{|\mathcal{D}_k|} \left\| \hat{V}_{\psi_k}^{\pi}\left(s_i\right) - \sum_{t'=t}^{T} \gamma^{t'-t} r_{i,t'} \right\|^2$$

Update V^π neural network weights: $\psi_{k+1} \leftarrow \psi_k + \beta \nabla_\psi \mathcal{L}(\psi_k)$

2.4 Imitation Learning & Guide Policies

Imitation Learning is a technique of solving Reinforcement Learning problems where instead of training an agent using the rewards collected through exploration of an environment, an expert guide policy provides the learning agent with a set of 'demonstrations' of the form $(s\ ,\ \pi_{\text{guide}}(s))$ where $\pi_{\text{guide}}(s)$ is the action the guide policy takes in state s. The agent then tries to learn the optimal policy by imitating the expert's decisions. In general, Imitation Learning is a powerful technique when one has access to an expert guide policy to show the desired behaviour in the environment.

2.4.1 Offline Imitation Learning

Offline Imitation Learning (OIL) aims to mimic the expert guide policies behavior from only its demonstration without further any interaction with the environment. An example of OIL is Behavioural Cloning (BC), which focuses on learning the expert's policy using supervised learning. The first major usage of BC is ALVINN, a car equipped with sensors which used a neural network to map the sensor inputs into steering angles and drive autonomously [25]. The steps of BC are quite straightforward; the pseudo-code for BC can be seen below in Algorithm (6).

In stark contrast to Reinforcement learning methods, Offline Imitation Learning does not require any reward function information whatsoever in order to work. There are two significant limitations with OIL. Firstly, as explained in Section (2.3.1), an underlying assumption of many statistical models is indepen-

Algorithm 6 Behavioural Cloning Algorithm

1. Collect set of trajectories $\mathcal{D} = \{\tau_i\}$ using π_{guide}
2. Build dataset of state-action pairs using trajectories:

$$\left\{ \Big(s_{i,t} \ , \ \ \pi_{\text{guide}}(s_{i,t}) \Big) \ \text{ for } i = 1, \ldots, |\mathcal{D}| \ \text{ and } \ t = 1, \ldots, T_i \right\}$$

where $\pi_{\text{guide}}(s_{i,t})$ is the action the guide policy takes in state $s_{i,t}$ and T_i is the length of trajectory τ_i.

3. Use supervised learning to train a model on this dataset, which can then be used to predict the guide policy actions in future states.

dence of data points within the data set. When we collect the state-action pairs from the expert guide policy, any two subsequent samples are highly dependent on one another. In particular, the ending state at time t is the same as the starting state at time $t + 1$. Therefore the statistical independence assumption does not hold. The second limitation is that if the dataset is sparse with regards to the state space $\mathcal{S}$ then the supervised learning model will struggle to generalise to unseen states.

It is clear from these two limitations that the quality of the dataset used to train the supervised learning model is very important for successful Offline Imitation Imitation Learning. Fortunately, there is a clever method of curating better datasets: DAgger.

2.4.2 DAgger

Dataset Aggregation (DAgger) is a method of Imitation learning which aims to generate more sophisticated datasets for the supervised learning methods to more accurately model the expert guide policy. Before we move onto the details of how this goal is achieved we shall first explain why it is necessary in the first place. In OIL, only the trajectories from the guide policy are used to train the model and this can cause significant sparsity in the training dataset. A common consequence of this dataset sparsity is that a negative feedback loop can be created where the more the learning agent deviates from the guide policy in its trajectory in the environment, the more unreliable the supervised learning model becomes due to poor generalisation ability. Following this, the slightly less reliable model causes the agent to further deviate in its trajectory, and thus a vicious cycle is instantiated. This is illustrated in Figure (12).

This effect is a direct result of the training dataset containing examples from only the guide policy trajectories. DAgger was designed specifically to remedy this unfortunate occurrence. Instead of only running the guide policy in order to collect policies, in DAgger, the learning agent is allowed to create its own trajectories in the environment, collecting a list of visited states s_t in the process.

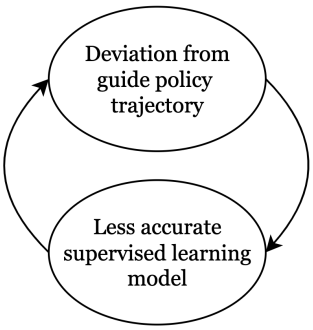


Figure 12: Illustration of the negative feedback loop that can occur when using Offline Imitation Learning/Behavioural Cloning. Poor generalisation ability leads to degradation of the supervised learning model which only leads to further deviation from the trajectories seen in the training dataset.

Then, the expert guide policy labels each of these states s_t with what action it would have taken in that same state, which we shall denote $\pi_{\text{guide}}(s_t)$. Finally, with this labelled dataset we can retrain the supervised learning model with the addition of these new data points, hence the name Dataset Aggregation. With this retrained model, the process starts again and the learning agent explores the environment collecting more data points to use in further training. This results in a training dataset that contains a much wider variety of states within the state space than in Offline Imitation Learning, leading to a more robust final policy. The DAgger algorithm is illustrated in Figure (13).

Additionally, the pseudo-code for the DAgger algorithm can be seen in Algorithm (7).

Algorithm 7 DAgger Algorithm

1. Initialise policy π_θ
2. Collect dataset of state-action pairs $\mathcal{D}$ using guide policy π_{guide}
3. For k = 1, 2, ... :
   - 3a. Use supervised learning to train π_θ using $\mathcal{D}$
   - 3b. Rollout π_θ and collect states $\{s_1, \ldots, s_T\}$
   - 3c. Create $\mathcal{D}_{\pi_\theta}$ by labeling the states with the guide policy:

$$\mathcal{D}_{\pi_\theta} = \left\{ \Big(s_t \ , \ \ \pi_{\text{guide}}(s_t) \Big) \ \text{ for } t = 1, \ldots, T \right\}$$

   where $\pi_{\text{guide}}(s_t)$ is the action the guide policy takes in state s_t.

   - 3d. Aggregate data : $\mathcal{D} = \mathcal{D} \cup \mathcal{D}_{\pi_\theta}$

DAgger is a very useful tool in the case that the expert guide policy is avail-

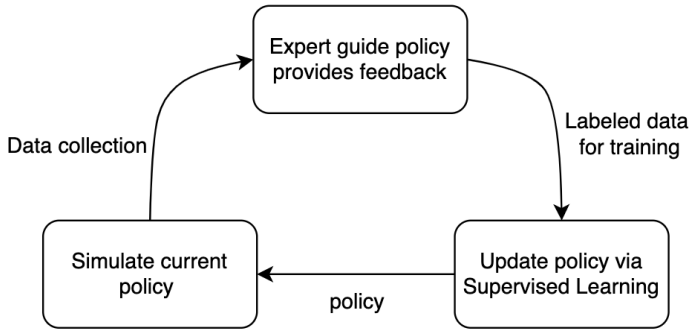


Figure 13: Illustration of the DAgger algorithm. There are three main steps to the algorithm. The first step is to simulate the current learning agent policy in the environment and collect a set of visited states s_t for $t = 1, \ldots, T$. The second step is to use the expert guide policy to label each state in the dataset with the action it would have taken given it was in that state. Lastly, the supervised learning model used to imitate the guide policy is retrained using the additional data collected in step 1. This process can continue until convergence or until a pre-determined number of iterations is met.

able for labelling data points during training. However, in many applications humans are the only available experts to label the training data. This means that each iteration of the algorithm requires a human to label data manually, which can massively slow down the speed and practicality of the learning process. Fortunately, there are other Imitation Learning methods we can use in this case. To finish off our discussion on Imitation Learning, we shall now discuss an entirely different method where we use the reward function as a tool for integrating guide policy decisions.

2.4.3 Reward Function Coupling

In contrast to Offline Imitation Learning and DAgger, Reward Function Coupling (RFC) uses the reward function in order to incentivise behaviour similar to the expert guide policy. Let the reward function of the environment be denoted by $\mathcal{R}$. The core idea is to modify the reward function to include an additional term which penalises deviation from the actions of the guide policy. At each timestep t the action of the learning agent a_t and the action of the guide policy $\pi_{\text{guide}}(s_t)$ are compared and a negative reward is given to the agent proportional to the difference of these two actions. The modified reward can then be written as

$$\tilde{\mathcal{R}} = \mathcal{R} - \beta \cdot \|a_t - \pi_{\text{guide}}(s_t)\| \tag{93}$$

where $\beta > 0$ is a hyper-parameter which determines how much priority the learning agent should place on imitating the actions of the guide policy. The

norm $\|\cdot\|$ can be chosen on a case by case basis. With this modified reward function we can now train the agent using any of the Reinforcement Learning techniques that we have discussed in Section (2.2) and Section (2.3). An advantage of RFC is that if there are certain states in the environment where the guide policy makes non-optimal decisions, then the agent can 'reject' the actions of the guide policy in order to seek more lucrative trajectories with higher expected future reward. However, as long as $\beta > 0$, the agent might learn to choose non-optimal actions with respect to the original reward $\mathcal{R}$ in order to maximise the modified reward $\tilde{\mathcal{R}}$. Therefore, in RFC the guide policy and learning policy are always coupled to some extent. This can be an issue in the case that the guide policy is not in fact an expert at the task. This would incentivise the agent to favor non-optimal actions with respect to the unmodified reward $\mathcal{R}$. Therefore, for a successful use of this method, the guide policy is assumed to be an expert.

A guide policy is said to be *present* if it is available whilst the learning agent is exploring he environment. RFC requires that the guide policy is present as $\pi_{\text{guide}}(s_t)$ must be computed at each time step in order to calculate the modified reward function. This is a significant limitation as in many cases researchers don't have access to present guide policies, and in extreme cases they may only have a single dataset of trajectories of a no longer existent guide policy. In this situation OIL or DAgger must be used as they do not require a present guide policy.

To summarise some important characteristics of the three Imitation Learning methods we have described in this section, Table (2) is provided below.

Method	**Interaction with environment**	**Present guide policy**	**Guide policy assumed an expert**
OIL	No	No	Yes
DAgger	Yes	No	Yes
RFC	Yes	Yes	Yes

Table 2: Table of three Imitation Learning methods and three important binary characteristics. In all cases the guide policy is assumed to be an expert at the Reinforcement Learning task. Only in Reward Function Coupling (RFC) is the guide policy required to be present. Only in Offline Imitation Learning (OIL) is it not necessary to have interaction with the environment.

In the next section we will present a novel technique of utilising guide policies that no longer requires the guide policy to be an expert.

3 Contextual Online Imitation Learning

Now that we have discussed some classical and modern techniques of utilising guide policies for Reinforcement Learning and Imitation Learning, we will now introduce a novel method: Contextual Online Imitation Learning (COIL). First, we will discuss the motivation behind the method and the high-level ideas that accompany it. Second, we will construct a mathematical basis for the novel method and discuss a few properties of the method. Lastly, we will extend the approach to a potentially more powerful technique: Dynamic COIL.

3.1 Motivation

Recall that the standard Reinforcement Learning method works by a continuous exchange of actions, states, and rewards between the agent and its environment. This can be seen below in Figure (14).

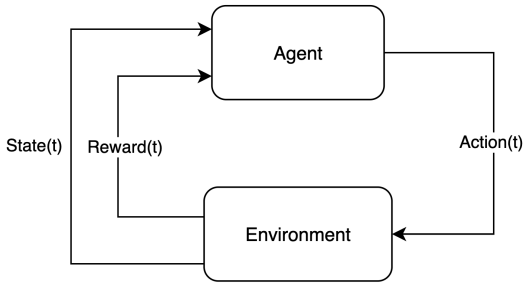


Figure 14: The standard setup for Reinforcement Learning given an agent and environment. In each time step $t \in [0, T]$, the agent is provided with information from the current state s_t, and then takes an action a_t within its environment. The reward for that time step is then calculated via the reward function $R_t = R(s_t, a_t)$.

Now suppose that you have access to a guide policy π_{guide} that is able to achieve good performance at a task you would like to train an agent to perform. We shall denote the action that the guide policy would perform in state s as $\pi_{\text{guide}}(s)$.

In contrast to the methods of utilising guide policies that we have seen in the previous section, we will give the agent complete access to the actions of the guide policy during training. In particular, at each time step t during training of the agent, $\pi_{\text{guide}}(s)$ will be provided as an additional observation to the agent (along with the regular observations that are given by the environment). The term *observation* is related to the fact that not in all cases will the agent have access to all of the necessary information to fully represent the underlying state of the environment. In real life, in order to fully represent an environment in all of its detail, it would require the information of every single particle involved in the system. As this is clearly not practical, it is more appropriate to call

the inputs to the agent *observations* and not the *state*, as the state is not fully knowable in practice.

By providing the action of the guide policy as an observation to the agent, we arrive at Figure (15).

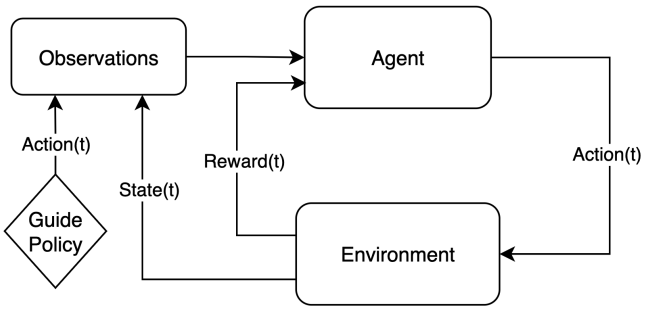


Figure 15: Illustration of Contextual Online Imitation Learning (COIL). In comparison to regular Reinforcement Learning, in COIL the action of some guide policy π_{guide} is provided to the agent as an observation at each time point t.

Additionally, if we have a set of guide policies $\pi_{\text{guide}_1}, \ldots, \pi_{\text{guide}_n}$, then we can generalise this framework to include the actions of all guide policies at time step t as observations to the learning agent. This can be seen in Figure (16).

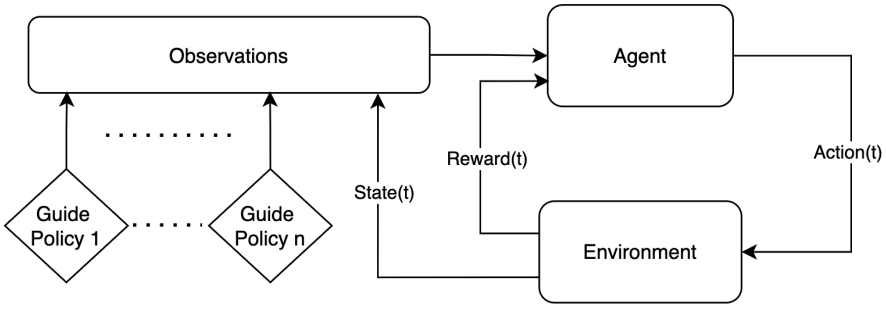


Figure 16: Illustration of Contextual Online Imitation Learning (COIL) with multiple guide policies.

Now that the agent is able to 'see' what the guide policy would do given the current context of the environment, it is able to learn through exploration how best to use this information in order to maximise its expected future reward. In other words, the agent is able to learn for itself in which *contexts* within the environment it should 'listen' to the guide policy's suggestion, and in which

contexts within the environment it should 'ignore' the guide policy's suggestion. This methodology is therefore a form of contextual imitation learning, as the agent is not encouraged to follow the actions of the guide policy regardless of the current state. Furthermore, this method is *online* as it is trained using continuous interaction with the environment. Therefore, this method is aptly named Contextual Online Imitation Learning.

It is important to note that COIL makes no explicit assumption about the type of Reinforcement Learning algorithm equipped with this new observation to the agent. COIL can be used equally well with Q-learning as with Policy-gradient methods. Future research on COIL could include a comparison between the different types of Reinforcement Learning algorithms when using COIL.

A major advantage of COIL in comparison to other methods of utilising guide policies is that there is no requirement for the guide policy to be an expert at the desired task. Even if the guide policy is only able to exhibit success in some subset of the total states within the environment, the agent in theory would be able to learn this fact through enough exploration of the environment, and then appropriately use this information to its advantage. Additionally, if the guide policy exhibits only moderate success across all states in the environment, the agent could also learn to *fine-tune* the actions of the guide policy in order to maximise reward. For example, if we had a guide policy able to walk a bi-pedal robot safely but also in an unnatural manner, with COIL, the Reinforcement Learning agent could theoretically adjust the suggested actions of the guide policy in order to make the bi-pedal robot a more natural and smoother walker. This would of course require a reward function that incentivises the Reinforcement Learning agent to walk in such a natural way.

This method can therefore be seen from two perspectives. The first is that COIL allows us to take a non-optimal hand-crafted guide policy, and use Reinforcement Learning methods to encode additional desired behaviours into the policy via the reward function (i.e. helping a non-optimal bi-pedal robot walk more naturally). The other perspective is that this method allows us to utilise existing guide policies to aid the agent during training, and in the process the agent can learn to use the suggested actions of the guide policy in whatever way maximises the total expected reward in the environment. If the agent decides that guide policy is useful in only one particular state in the environment, then it can learn to ignore the guide policy in all other states. This is in strong contrast to classical Imitation Learning methods, which focus on minimising the difference between the guide policy and the agents policy. In COIL, there is no explicit penalty in deviating from the guide policy, this gives the Reinforcement Learning agent complete flexibility to use the guide policy in complex and situational ways.

It is an open question whether this novel method can be combined with other forms of Imitation Learning. Implementing Reward Function Coupling in conjunction to COIL in particular remains an interesting avenue for future research. We can now add COIL to Table (3) below in order to see the important differences between the different Imitation Learning methods.

Method	Interaction with environment	Present guide policy	Guide policy assumed an expert
OIL	No	No	Yes
DAgger	Yes	No	Yes
RFC	Yes	Yes	Yes
COIL	Yes	Yes	No

Table 3: Table of four Imitation Learning methods and three important binary characteristics. COIL is the only Imitation Learning method where the guide policy is not assumed to be an expert at the Reinforcement Learning task. Both in COIL and Reward Function Coupling (RFC) is the guide policy required to be present. Only in Offline Imitation Learning (OIL) is it not necessary to have interaction with the environment.

3.2 Formalisation

In many tasks in Reinforcement Learning there are multiple competing objectives that determine the overall success of a given policy. Let $\Omega_1, \ldots, \Omega_n$ be the set of *objectives* that that we want the agent to learn during training, where n is the total number of objectives.

Examples of Reinforcement Learning tasks with multiple competing objectives include:

Air traffic control system :

- Ω_1 : No aircraft crashes
- Ω_2 : Minimise aircraft waiting time
- Ω_3 : Maximise fuel efficiency of aircraft

Nuclear assembly design :

- Ω_1 : No nuclear meltdown
- Ω_2 : Minimise cost (€)
- Ω_3 : Minimise assembly time

Then let Ω_{I} be the set of objectives that the guide policy is capable of achieving near-optimal performance. This guide policy could be hand-crafted by engineers or extracted from data via imitation learning. We have that $\mathrm{I} \subset \{1, \ldots, n\}$.

Now, we shall denote the part of the reward function associated with the objective Ω_i at time step t as $r_{\Omega_i}(s_t, a_t)$. Then, the overall reward function taking into account each objective is defined as:

$$r(s_t, a_t) = \sum_i C_i \cdot r_{\Omega_i}(s_t, a_t) \tag{94}$$

and additionally the reward function that the guide policy achieves near-optimal performance is given by

$$r_{\Omega_I}(s_t, a_t) = \sum_{i \in I} C_i \cdot r_{\Omega_i}(s_t, a_t) \tag{95}$$

where C_i is the coefficient determining how much the agent should prioritise objective Ω_i. It is clear from the above examples that some objectives are more important than others, therefore these coefficients are very important for encoding the desired behavior for the agent to learn.

So far, the policy of the agent π has been a function of the current state s and action a. In practice, the state is represented by a finite dimensional vector of data points $s = \{s_1, \ldots, s_m\}$ which gives the agent as much information about the context of the environment as possible. Therefore, written out in full we have

$$\pi(s, a) = \pi(s_1, \ldots, s_m, a) \tag{96}$$

and in COIL we also provide the action taken by the guide policy π_{guide} in that same state:

$$\pi(s, a) = \pi(s_1, \ldots, s_m, \pi_{\text{guide}}(s), a) \ . \tag{97}$$

An important and relatively straightforward insight we can make is that policy of the agent trained via COIL π_{COIL} is capable of achieving at least the same performance as the guide policy π_{guide}. This is because the agent is capable of completely mimicking the actions provided by the guide policy. Many modern policy-based Reinforcement Learning algorithms use neural networks to model the policy of the agent. A result of the universal approximation theorem is that neural networks are capable of approximating any continuous function given enough units within the hidden layers [26]. It is consequently possible that a neural network can learn a simple function that mimics only a single input:

$$\pi(s, a) = \pi(s_1, \ldots, s_m, \pi_{\text{guide}}(s), a) \approx \pi_{\text{guide}}(s) \tag{98}$$

where this approximation can become arbitrarily accurate [26]. However, if there are more advantageous policies to be discovered then it is very likely that the agent will surpass the performance of the guide policy given enough exploration of the environment. In fact, if we are using Q-learning then it is guaranteed that the COIL agent will surpass the guide policy in expected reward if the guide policy is not the optimal policy. This is a direct consequence of Theorem (2.3).

Following this, it is interesting to investigate whether the policy learned via COIL can be shown to achieve better results than the policy learned via regular Reinforcement Learning methods. In general, the success of COIL is likely strongly dependent on the environment, the task, and the quality of the guide policy utilised. In Section (4) we will apply COIL to various applications and investigate the success of the resulting policies.

3.3 Dynamic COIL

So far we have introduced Contextual Online Imitation Learning where we have a fixed, pre-existing guide policy π_{guide}. Dynamic Contextual Online Imitation

Learning (Dynamic COIL) is different to this in that it no longer keeps the guide policy fixed, but it treats it as another trainable component in the system. The first phase of Dynamic COIL is the agent training phase, where the agent is trained as in static COIL, where we feed in the actions of the guide policy $\pi_{\text{guide}}(s)$ to the agent at each time step. After a sufficient amount of learning (ideally once convergence of the policy has been achieved), the current policy of the agent is fixed and we begin to train the guide policy instead. This is called the guide policy fine-tuning phase. It should be noted here that we do not want to radically change the guide policy once the agent's policy is fixed, because this could lead to rapid instability of the agents policy, instead of this, we want to fine-tune the guide policy in order improve its ability to maximise the expected reward of the agent in the environment. In order to reduce the amount the guide policy changes during this phase, a smaller learning rate can be selected. After the guide policy has been fine-tuned, it once again becomes fixed, and we return to the agent training phase, with the agents policy activated again for further training, however this time with a smaller learning rate than before to ensure stable convergence. Through repeated exchange between training the agent and fine-tuning the guide-policy, it is possible that the agent's resulting policy is very good at effectively using the information provided by the guide policy. In Figure (17), an illustration of Dynamic COIL can be seen.

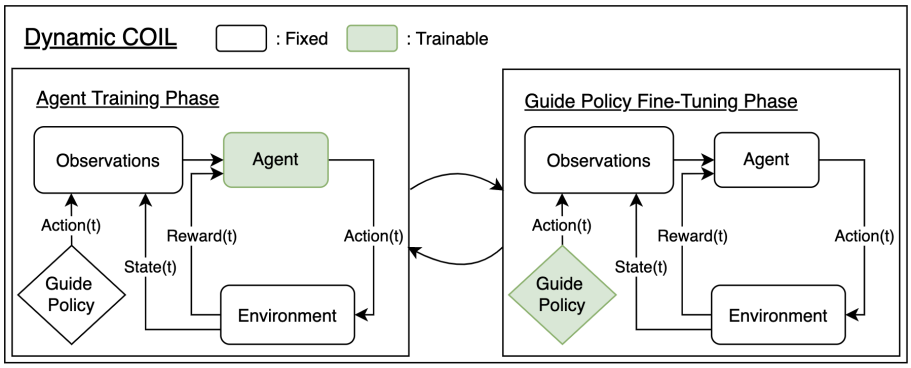


Figure 17: Illustration of Dynamic Contextual Online Imitation Learning (Dynamic COIL). In the agent training phase, we have the regular COIL method, where the agent is being fed the actions of the guide policy at each time step t, and the agent is improving its policy through repeated interaction with the environment. In the guide policy fine-tuning phase, the agent's policy is fixed, and the learning algorithm now operates only on the parameters of the guide policy. In this way the guide policy can be fine-tuned in order to offer the most beneficial action to the agent. Dynamic COIL works by iteratively switching between these two phases.

Dynamic COIL can also be used with multiple guide policies. This poses the question of how and when to train the individual guide policies. We shall now introduce two Dynamic COIL training schedules: stochastic and sequential.

In the stochastic training schedule, during the guide policy fine-tuning phase a random guide policy is selected to be trained for the entire phase. Fine-tuning one policy at a time ensures that the inputs to the agents policy don't change too quickly. In the sequential training schedule, during the i-th guide policy fine-tuning phase, the (i mod n)-th guide policy is chosen to be trained, where n is the number of guide policies. This ensures that each guide policy undergoes the same amount of fine-tuning in the long run. A comparison of these two training schedules would be a good avenue for future research on this topic.

A particularly interesting application of Dynamic COIL is the setting where we want an agent to be able to solve multiple tasks. In order to leverage Dynamic COIL, we could provide a set of guide policies where each guide policy is capable of solving a desired task in the environment. For example, if researchers had access to a bi-pedal robot that can be trained to perform many different tasks such as walking, running, jumping, opening doors, etc. The researches could train the robot on each individual task first to build the set of guide policies that achieve good performance in each task. Following this, they could train the entire system as a whole by feeding the actions of the guide policies as observations to the learning agent. This robot would then be able to discover through exploration of the environment which guide policy to 'listen' to in which contexts. In this way the agent can be seen almost like a master puppeteer, able to make use of the 'running policy' when running offers the most expected reward, and use the 'jumping policy' when jumping is most optimal. Then, by using Dynamic COIL, the transitions between certain tasks could be fine-tuned, such as the transition between running and jumping for our bi-pedal robot. Dynamic COIL is therefore a potentially very powerful framework for multi-task Reinforcement Learning tasks. Additionally, by allowing training of the guide policies, it offers researchers the ability to begin with guide policies with only moderate success in its given task, and then through training it can be improved to a more powerful guide policy. As the guide policies are no longer fixed as in static COIL, Dynamic COIL is a more flexible methodology. An illustration of a potential Dynamic COIL network architecture can be seen in Figure (18).

As a final note to this section, we shall discuss a particular 'trick' in the case that the researcher has access to guide policies but they are not parameterised, and therefore cant be directly trained.

In this case, the researchers could use Offline Imitation Learning where they train a neural network with inputs $s_1, \ldots, s_m$ and output $\pi_{\text{guide}}(s)$. The training dataset could be extracted through use of the guide policy in the environment. If training is successful, the researchers would now have a parameterised version of the guide policy. They could then use this parameterised version as a proxy for the original guide policy. This would allow the researchers to still use Dynamic COIL even in the case where they don't have a parameterised guide policy.

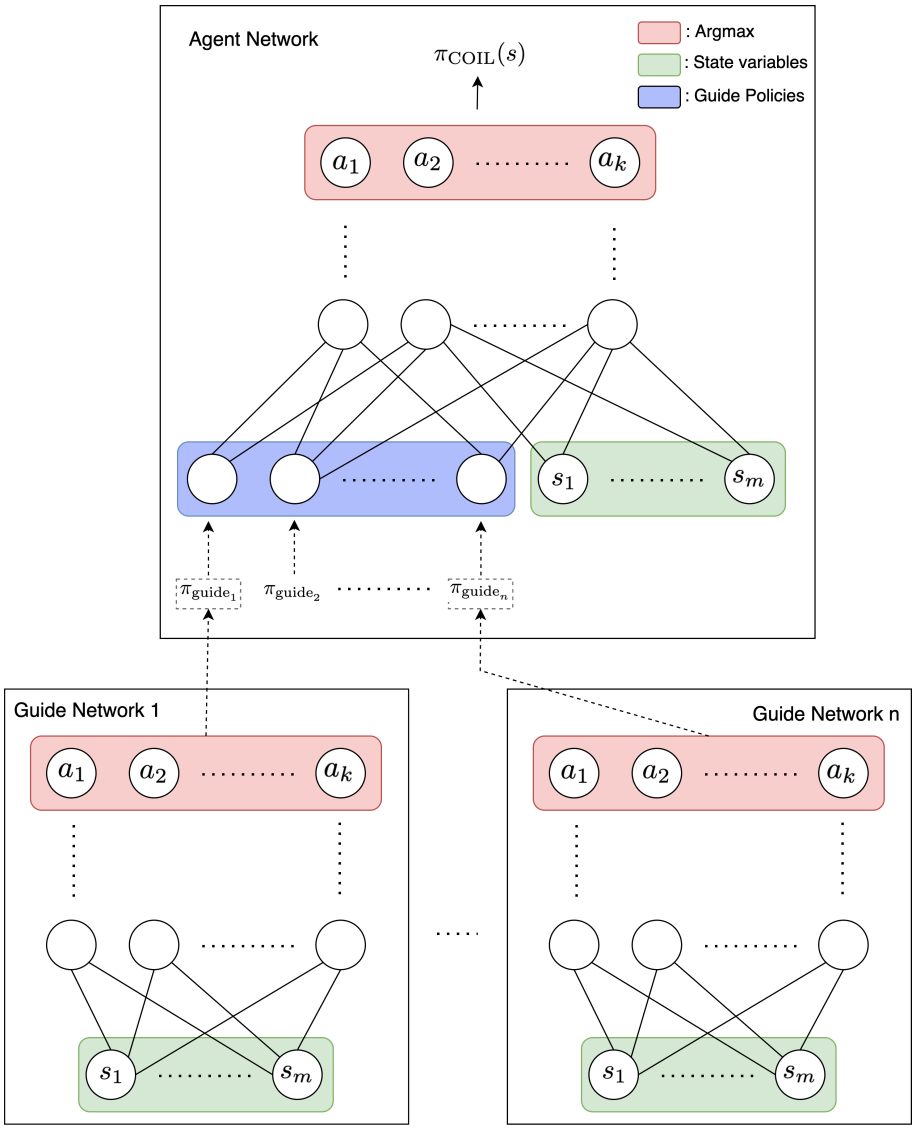


Figure 18: Illustration of the Dynamic COIL architecture in the case that neural networks are used to parameterize both the agent network and the guide policies. This illustration also assumes a discrete action space $\mathcal{A}$. Each guide policy provides its suggested action as an input to the agent, and the agent learns through repeated exposure with the environment how best to use these guide policies. In Dynamic COIL, the guide policies are also fine-tuned during training. This illustration shows how the guide policies and the form one greater network which can be trained via back-propagation[1].

4 Results & Applications

In this section, we will first test COIL on a simulation of a self-driving car. Following this we will test the robustness of COIL on a real-life self-driving robot. We will begin by discussing the agents simulated environment and task we have designed for our agent, as well as how we reward the agent. Following this, we will compare four self-driving policies: COIL, regular Reinforcement Learning (PPO), a guide policy, and Offline Imitation Learning of the guide policy.

4.1 Simulation

4.1.1 Setup

The self driving car simulation was designed in Python3 using the package Pygame. The Python code can be found on Github at : `https://github.com/alex21347/Self_Driving_Car`. In the simulation, the car is able to drive around on a 2D plane where its movement is governed by a dynamic model usually called a bicycle model, as we are taking into account the length of the car and not assuming the car is simply point in space [27]. Within the simulation there are also left cones and right cones which specify the sides of a track for the car to drive through. An example of such a track can be seen in Figure (19).

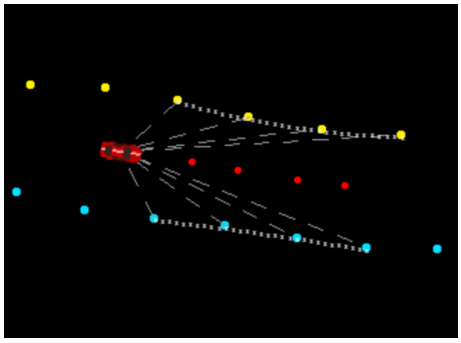


Figure 19: Image of a self-driving car simulation programmed in Python. The simulated car can be seen driving from left to right. The left cones (yellow) and right cones (blue) can also be seen guiding the car down a track. The red dots represent the middle of the track and are used for visual purposes. Cones that are detected by the car have a dashed line drawn between the car and the cone. Furthermore, the grey dotted line on either side of the track represent a cubic spline estimate of the boundary of the track.

In order to turn this simulation into a Reinforcement Learning problem, we need to decide what our action space $\mathcal{A}$ will be, what our observation space $\mathcal{O}$ will be, and what our reward function $\mathcal{R}$ will be. As a side note, we now consider the observation space instead of the state space because the state space is assumed

to contain all the necessary information in the underlying state, which is not possible in practice. For the remainder of this section we shall first discuss the observation space, the action space, and the reward function of our self-driving car agent, before moving onto a discussion on how COIL and Offline Imitation Learning can also be applied.

Firstly, the car has a fixed field of view of 60 degrees in either direction from the direction it is facing. Additionally, the simulated sensors on the car have a fixed range for of 7m. If either a left cone or right cone falls within the correct angle and range of the car, it is considered as 'visible' to the car and can thus be used to form an observation for the Reinforcement Learning agent. The design of an informative observation to provide to the agent is a non-trivial task. The observation provided to the agent must have the same number of dimensions throughout the entire exploration of the environment. However, at each time step t the car will have detected some unknown number of cones in front of it. Therefore, an observation must be designed so that it is not effected by the variation in the number of visible cones from one time step to another.

One solution to this problem is the use of cubic splines. For both the left side of the track and right side of the track, we fit a cubic spline using the coordinates of the cones in the simulation. This cubic spline acts as a *boundary estimation* for each side of the track. This cubic spline boundary estimation can be seen in Figure (19) as grey dotted lines between detected cones. For more information on cubic splines the reader is directed to Appendix (A.2).

Finally, to turn the boundary estimates into a fixed-size observation for the agent, we can take 5 equidistant sample points along each side of the track. Therefore, we end up with 10 sample points in total, each of which is comprised of two pieces of information; the distance to the car r, and the relative angle of the sample point with respect to the car axis, θ. These 20 values are what we shall use as our observation space, thus we have

$$\mathcal{O} = \{r_1, \theta_1, \ldots, r_{10}, \theta_{10}\}$$

where the first five (r, θ) pairs represent the boundary sample points on the left side of the track, and the last five (r, θ) pairs represent the boundary sample points on the right side of the track.

For the action space $\mathcal{A}$ we simply allow the car to decide which steering angle it would like to take in the interval $[-80, 80]$. Thus

$$\mathcal{A} = [-80, 80] \ .$$

Now we shall discuss the reward function for our Reinforcement Learning problem. The self-driving car in our simulation has three main objectives:

- Ω_1 : Survival (how safe the passengers are)
- Ω_2 : Speed (how fast the car can drive)
- Ω_3 : Smoothness (how smooth the journey is)

Therefore, a possible way of encoding these objectives within our reward function is:

i) A positive reward for successfully progressing along the track (Ω_1)

ii) A negative reward for crashing (Ω_1)

iii) A positive reward for driving fast (Ω_2)

iv) A negative reward for driving in an unstable manner (Ω_3)

Using these four motivations, our reward function to impose these objectives can be:

- $r_{\Omega_1} = 70 \cdot \mathbb{1}\{\text{Car finishes track}\} - 100 \cdot \mathbb{1}\{\text{Car crashes}\} + 2.5 \cdot \mathbb{1}\{\text{Cone detected}\}$
- $r_{\Omega_2} = -0.8T \cdot \mathbb{1}\{\text{Car finishes track}\}$
- $r_{\Omega_3} = - \parallel \theta - \tilde{\theta} \parallel$

where $\mathbb{1}$ is the indicator function, T is the time taken for the car to finish the track, θ is the direction the car is facing, and $\tilde{\theta}$ is a exponentially weighted moving average of previous car directions. Our final reward function for the entire task is then defined as

$$r = \sum_{i=1}^{3} r_{\Omega_i} \quad .$$

Following this, in order to use COIL with this simulation, we need a guide policy π_{guide}. Consequently, it is necessary to design a suitable guide policy. A suitable choice is to create a guide policy that aims to keep the car as 'safe' as possible at all times. The guide policy first calculates an estimate for the centre path that goes through the middle of the track, and then takes the steering angle best suited to follow this path. For the details on how exactly the guide policy executes these two steps, please see Appendix (A.1). We can call this guide policy the 'safety policy' as it encodes our safety objective Ω_1. Figure (20) shows the COIL methodology applied to our self-driving car simulation.
Now we must decide exactly which Reinforcement Learning algorithm to use for our agent. Through experimentation, it became clear that Proximal Policy Optimisation (PPO) offered the strongest and most stable training runs, and therefore we will use PPO as our Reinforcement Learning algorithm for our self-driving car agent, and also when applying the COIL methodology. Future research could include a full comparison of COIL applied to different Reinforcement Learning algorithms.

As a final method to use for comparison, Offline Imitation Learning (OIL) will be applied to mimic the behavior of the guide policy discussed previously. For this paper, OIL will be the only Imitation Learning method we will use in our comparison. As the guide policy is not an expert in our case, it is not necessary to implement multiple Imitation Learning methods as they are unlikely to

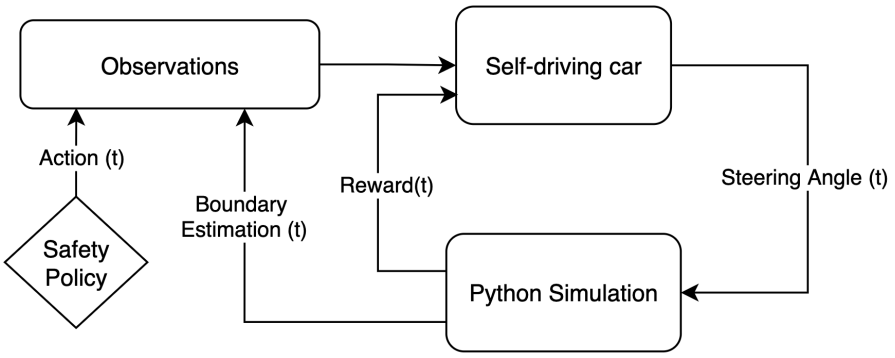


Figure 20: The COIL methodology applied to the application of a self-driving car simulation. The guide policy in this case is called a 'safety policy' because the guide policy in this case is designed entirely to make sure the car does not crash and nothing else. For more details see Appendix (A.1).

be highly competitive with Reinforcement Learning and COIL. This is because mimicking a non-expert policy is likely to produce a another non-expert policy. Further research could compare COIL to other state of the art Imitation Learning algorithms, such as Jump-start Reinforcement Learning [28] and DAgger [29].

As discussed in Section (2.4), Offline Imitation Learning is a form of Supervised Learning. At each time step t, the boundary estimate samples $(r_1, \theta_1, \ldots, r_{10}, \theta_{10})$ will be fed into a Machine Learning model, and asked to predict the action of the guide policy $\pi_{\text{guide}}(s)$. In order to train the Machine Learning model, a dataset

$$\left\{ \left(r_1(t), \theta_1(t), \ldots, r_{10}(t), \theta_{10}(t) \right) , \;\; \pi_{\text{guide}}(s_t) \;\; \text{for } i = 1, \ldots, T \right\}$$

is required, where T is large enough for the Machine Learning model to learn to accurately predict the actions of the guide policy given the current state of the environment.

To collect this dataset, the guide policy was simulated to drive around each of the training tracks, and at each time step t the sample

$$\left\{ \left(r_1(t), \theta_1(t), \ldots, r_{10}(t), \theta_{10}(t) \right) , \;\; \pi_{\text{guide}}(s_t) \right\}$$

was stored. In total, this generated a dataset with 16510 data points to train the Machine Learning model with. The Machine Learning model that was used to predict $\pi_{\text{guide}}(s_t)$ was a Random Forest Regression model. This was chosen as it offered the best Mean Square Error (0.01489 on the 'test' data) for this dataset, Linear Regression was also applied but this was not as accurate. For

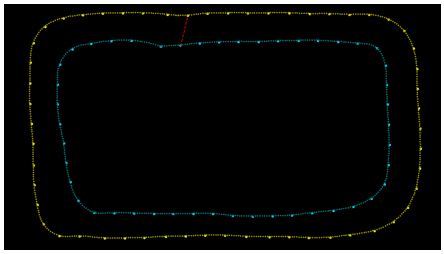
more details on Random Forests the reader should read the original 2001 paper by Breiman [30].

Using n-fold cross validation, the hyper-parameter optimised Random Forest Regression model achieved a Mean Square Error of 0.01489. The optimal hyper-parameters included a maximum tree depth of 20, and a total of 200 decision trees.

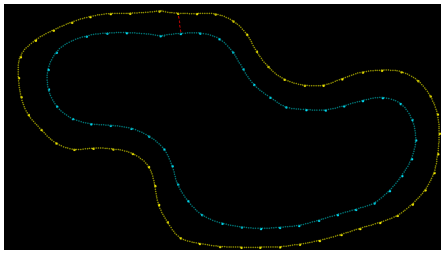
Now that we have set up COIL, RL, a guide policy, and OIL, we can train our COIL and RL agents and then compare all of the policies together afterwards.

4.1.2 Training

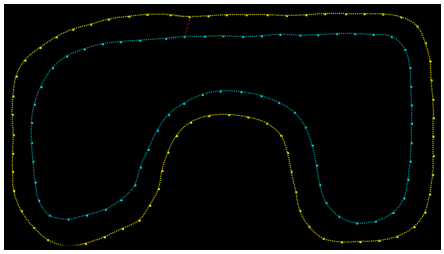
To train the RL and COIL agents, 10 tracks were created in the simulation. For training, 7 of the tracks were provided to the agent randomly during each episode of the simulation, this was to ensure the agent would not just learn memorise a single track. The remaining 3 tracks were reserved for testing generalisation. In Figure (21) below, 4 of the training tracks can be seen.



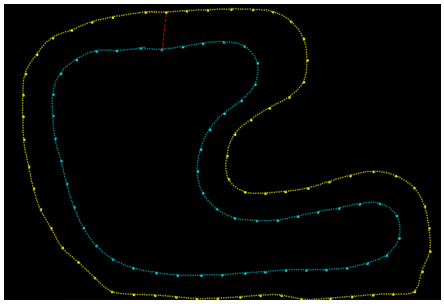
(a) Track 1



(b) Track 2



(c) Track 3



(d) Track 4

Figure 21: Four of the seven tracks used to train the Reinforcement Learning agent to drive autonomously. All tracks were designed in Python using the package Pygame. The yellow circles represent left cones, and the blue circles represent right cones.

The COIL and RL agents were trained using Proximal Policy Optimisation over

5 runs, and the maximum reward achieved during training was recorded for each run. The average maximum reward for COIL and RL can be seen in Table (4) below.

Method	**Average Max Reward**
COIL	477.6
RL	414.2

Table 4: Table of training results for standard Reinforcement Learning (RL) and Contextual Online Imitation Learning (COIL). Over 5 training runs the highest achieved average reward was recorded and the average result can be seen in this table. It can be seen that COIL is the better training algorithm in this case, giving a 15.3% increase in the average max reward.

It can be seen in Table (4) that COIL achieves much stronger results during training, giving a 15.3% increase in the average max reward.

An additional trait that was noticed during training was that COIL trained 'quicker' in the sense that after a given number of time steps t, COIL generally had a larger increase in mean reward than the regular Reinforcement learning agent. This can be seen clearly in Figure (22) below.

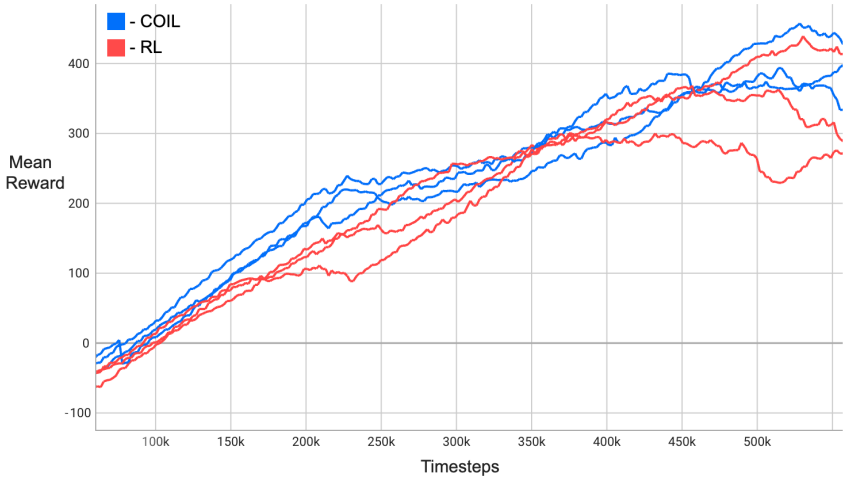


Figure 22: The training runs of 3 agents trained by COIL and 3 agents trained by regular RL. COIL can be seen to achieve a higher mean reward at every point along the x-axis. These training runs were computed using the University of Groningen's high performance computer cluster Peregrine.

Therefore, we have found evidence that by providing the actions of the guide policy to the agent during training in COIL, we achieve both quicker training and with a higher maximum reward. Before we can make conclusions, we shall also test our different algorithms on 3 unseen test tracks and compare results.

4.1.3 Generalisation

Firstly, it should be noted that comparing the average reward of the training tracks and the test tracks is not appropriate because all of the tracks are unique, and the length of the track strongly affects the total reward that can be attained on the track. Thus it is only meaningful to make comparisons between the policies on these test tracks.

To evaluate the success of the policies, we will analyse the lap times of the agents, the smoothness of their driving, the total reward they achieve, and lastly how often they crash. The smoothness is calculated by taking the average of the magnitude of all of the r_{Ω_3} values generated during the tests. To recap, we have that

$$r_{\Omega_3} = - \parallel \theta - \tilde{\theta} \parallel$$

where θ is the direction the car is facing, and $\tilde{\theta}$ is a exponentially weighted moving average of previous car directions. Therefore, the larger the average magnitude of r_{Ω_3}, the less smooth the journey is.

Table (5) below contains all the results for the different policies. It should be noted that for COIL and RL, the policies that was chosen for testing were the policies that achieved the highest average reward during training.

Method	Mean Lap Time	Mean Smoothness	Mean Reward
Guide Policy	48.81s	0.121	609.4
OIL	48.82s	0.061	620.8
RL	48.58s	0.557	614.5
COIL	48.48s	0.606	**632.9**

Table 5: Table of results for four self-driving car algorithms tested in a Python simulation; a guide policy, Offline Imitation Learning (OIL), Reinforcement Learning (RL), and Contextual Online Imitation Learning (COIL). The algorithms are judged on speed, smoothness, and safety, where these three objectives are encapsulated by the reward. It can be seen that COIL offers the highest average reward. Each policy did not crash once during testing so this was left out of the table.

Table (5) shows that COIL achieved the highest mean reward of all four methods when tested on three test tracks. This provides evidence that COIL has strong generalisation ability, as its performance is highly competitive both on the training tracks and the test tracks. All four methods did not crash once during the tests, and was therefore left out of the table for brevity. COIL is also the fastest of the policies, achieving the lowest average lap time of the four methods. However, COIL is unexpectedly also the least smooth of the policies. This might be because the agent has learned to prioritise speed over smoothness in order to achieve the higher rewards. As we have seen, Reinforcement Learning algorithms are designed to always maximise the expected reward, therefore, the fact that COIL and RL have a higher average smoothness than the guide

policy and OIL is indicative that the reward function places less emphasis on the smoothness objective, and more emphasis on the safety and speed objectives.

To complete our analysis on COIL, we will now apply our four policies to a real-life robot car, and analyse how the four methods hold up in a more complex situation and a significant change in incoming sensor data. The following section will aim to answer the following questions:

i) *Does COIL still work effectively with a robot car with real sensors?*

ii) *How robust are the algorithms with regards to transfer learning?*

4.2 Robot

In this section, four self-driving car algorithms will be compared on a real-life robot car; Contextual Online Imitation Learning (COIL), standard Reinforcement Learning (RL), a guide policy, and Offline Imitation Learning of this guide policy. The robot was provided by the Robotics Lab in the Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence at the University of Groningen. This robot can be seen in Figure (23) below. On top of the robot there is a single Lidar detector (RPLidar A1M8) which will be the only sensor available for autonomous driving.

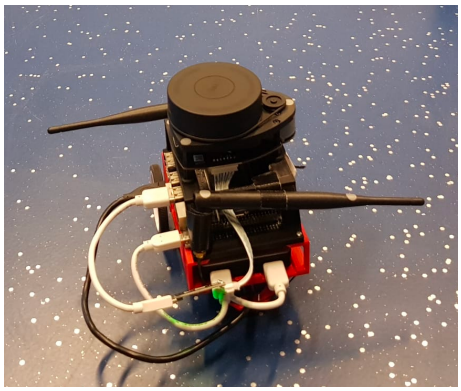


Figure 23: Photo of the robot car used for comparing autonomous driving algorithms. The robot was provided by the Robotics Lab in the Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence at the University of Groningen. On top of the robot there is a single Lidar detector (RPLidar A1M8) which will be the only sensor available for autonomous driving.

First we will discuss the details for setting up a robot capable of running autonomous driving software, and some of the practical difficulties that arise when transitioning from simulation to real life. Following this we will test our four models on a designated track, and analyse the results.

4.2.1 Setup

Recall that in the simulation, whether a cone is a left cone (indicating the left side of the track) or a right cone (indicating the right side of the track) is automatically given. In real life, it is necessary to construct a method of determining the orientation (left/right) of the cone using only the Lidar sensor data. To do this we can use two differently shaped cones, one thick and one thin. Figure (24) below shows an image of the car alongside some left (thin) cones and right (thick) cones.

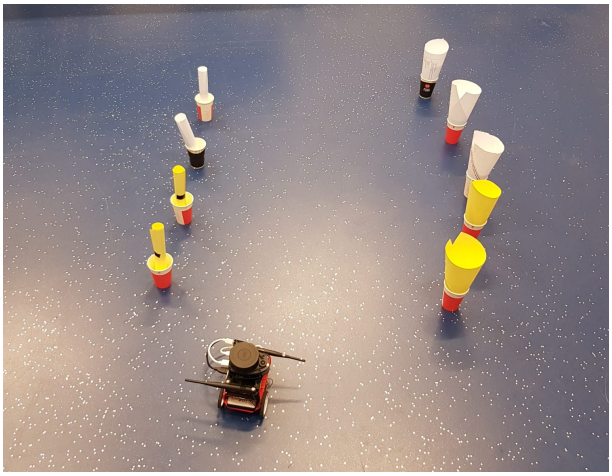


Figure 24: Photo of the robot car along with some cones representing the left and right side of the track. The left and right cones were designed to have different characteristics (size, width, etc.) so that a classifier could be trained using only Lidar data. The Lidar sensor can be seen on the top of the robot in this photo.

For each 360 degree rotation of the Lidar sensor, a simple clustering algorithm was used to detect potential cones. Two points were determined to be within the same cluster if and only if they were less than 5cm away from one another in 2D space. Following this, for each cluster three features were extracted using the Lidar data; the width of the cluster, the depth of the cluster, and the girth of the cluster. Then using these three features a Support Vector Machine (SVM) can be trained to classify the cones between left and right cones.

To collect a training dataset, the robot was first driven manually around a set of left cones exclusively, and each cluster identified in this time period was automatically labeled as a left cone (as all right cones had been removed). Following this, the robot was driven in the same way exclusively driving around right cones, and each cluster was labeled as a right cone. Combining these two data sets together gave us our training dataset for the SVM.

The SVM achieved 98.8% classification accuracy. Fortunately, the data distributions across the three features were sufficiently different between classes.

This can be seen clearly in Figure (25) where Principal Component Analysis was applied in order to visually analyse the two classes.

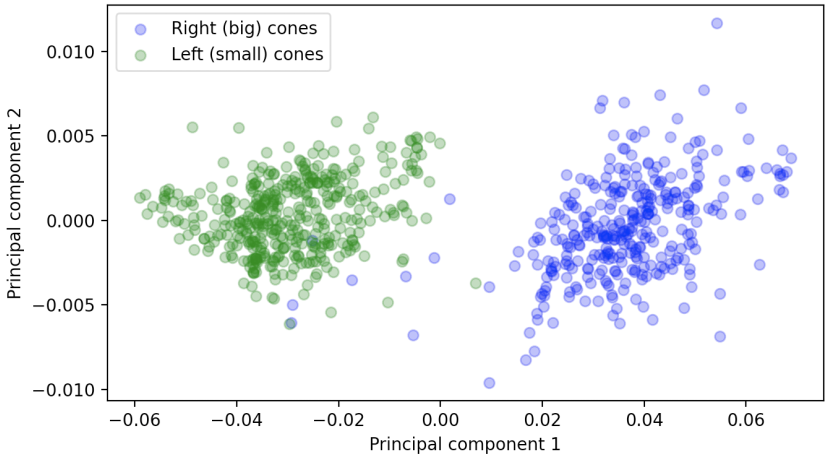


Figure 25: Scatter graph of detected cones from the Lidar. For each cone, 3 variables (width, depth, and girth) were extracted using the Lidar data and then principal component analysis was performed to find the two most informative components. In this graph, detected left cones are represented as blue, and right cones are represented as green, with the axes representing the two principal components.

Unfortunately, the cone classifier was unable to achieve 100 % accuracy, therefore, it was almost certain that during the self-driving, the track would be misinterpreted at some point (approximately once every 100 frames). Through a careful programming workaround, this 1 in 100 event can be made insignificant to the overall success of the self-driving robot. For brevity, we will not go into the details of this workaround here. The full explanation can be found in Appendix (A.3).

In order to select the optimal hyperparameters for the SVM, n-fold cross-validation was also applied, this ensured that we had the best possible Machine Learning model to accurately classify cones. Other Machine Learning models were also tested, in particular a Random Forest model was also trained for this task, however, the SVM offered the strongest results and was therefore chosen to be the cone classifier for our robot.

Once an accurate cone classifier is trained, the car is able to extract the relevant information from its environment; a list of left cone locations and a list of right cone locations, all relative to the location of the car. This information can then be sent over a wireless network from the robot to a remote computer in order to create a reconstruction of the environment. An example of such a reconstruction can be seen in Figure (26).

Now that it is possible to effectively communicate the Lidar sensor data in

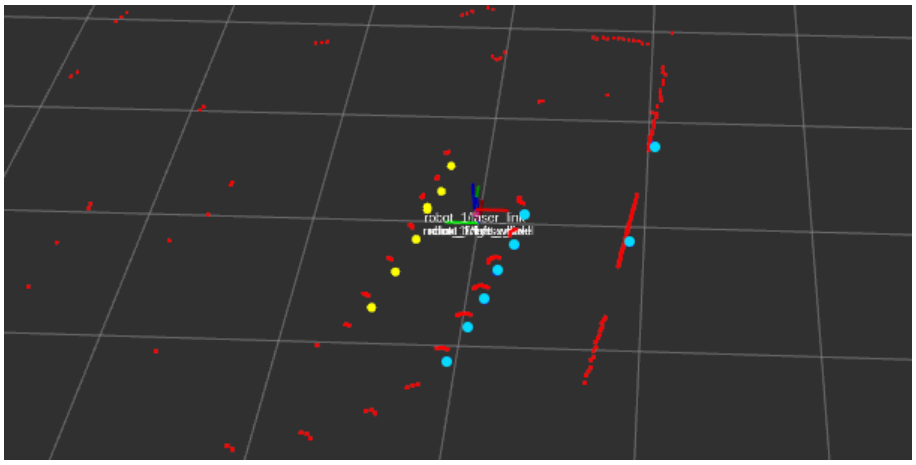


Figure 26: Image of the 3D robot visualiser RViz during a lap of the robot on the test track. Red points represent sensor data from the Lidar, yellow circles represent detected left cones, blue circles represent detected right cones, and the white text in the middle is the RViz label given to the robot.

a format that resembles the simulation, we are able to start sending driving instructions back to the robot. The first step is to initialise the Python simulation and place left and right cones in accordance to what the robot is detecting. Then, in order to use COIL or RL, we must construct our observation in the same format as before:

$$\mathcal{O} = \{r_1, \theta_1, \ldots, r_{10}, \theta_{10}\} \quad .$$

Thus, similarly to before, we must compute the cubic spline boundary estimates for both sides of the track and then sample them at 5 equidistant points. Once our observation is formed we can simply feed it into the agent as an input and then it will predict the next action. This action is then sent back to the robot for completion, and the simulation awaits it next batch of Lidar sensor data from the robot.

There are two final important caveats worth discussing in order to set up the robot for self-driving:

i) Difference in observation distribution.

In the simulation, cones were 'detected' by the car if it was no further than 7m, and it was also within 60 degrees of either side of the direction the car is facing. In real-life, the robot was able to accurately detect cones up to a distance of 1m. This is because of two reasons. Firstly, if two cones are perfectly aligned with the Lidar sensor, then the closer of the two cones will cast a shadow, and the further cone will be completely undetected. This situation is illustrated in Figure (27). This effect also introduces the bias that right cones are harder to

detect as they are thicker and thus cast larger shadows.

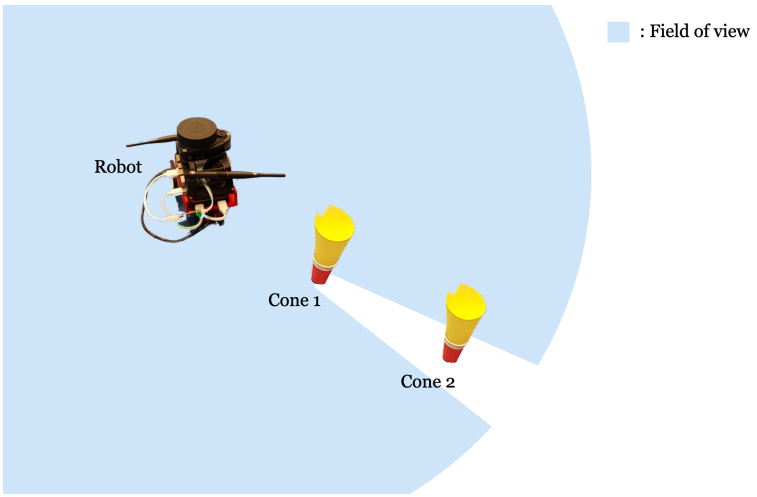


Figure 27: Illustration of the limitation of using only a Lidar for detecting cones. It can be seen here that Cone 1 is casting a shadow for the Lidar sensor, resulting in Cone 2 being undetected.

Secondly, the Lidar sensor has fixed angular resolution, therefore, the closer a cone is to the sensor the more data points are within its cluster. Quite simply, the more data points per cluster, the more reliable the cone classifier. In the extreme case, a cluster with a single data point is impossible to accurately classify as there is no way to extract the width, depth, or girth of the cluster.

Therefore, with these practical considerations in mind, it was required to limit the range of the Lidar to 1m. Additionally, the Lidar is capable of 360 degree detection of cones, whereas the simulated car could only detect cones in front of it ± 60 degrees. Therefore, during training of the Reinforcement Learning agent in the simulation, the agent was only exposed to cones within this more restricted field of view. It general, the closer the distribution of the observation is to the observations seen during training, the better the agent will be able to drive. Therefore, all cones detected by the Lidar that were more than 60 degrees on either side were removed in a post-processing step.

A final step to adjust the Lidar sensor data to closer match the observations in the simulation is a scale transformation of the cone distances. Recall that in the simulation the car's field of view had a radius of 7m, however, as we have discussed the Lidar is only capable of detecting cones accurately up to 1m. To correct for this vast difference of scale, the distance of each cone was multiplied by 7. All of these steps combined offered considerable improvement in practice.

ii) Robot driving instructions

So far, the action that the agent takes at each time step is a steering angle

θ. Unfortunately, the instructions that the robot accepts is instead an angular velocity ω. Thus, for each steering angle θ, the corresponding angular velocity ω must first be calculated before sending to the robot. First we calculate the turning radius of the car:

$$r = \frac{\text{Car length}}{\sin(\theta)} \tag{99}$$

and then we have

$$\omega = \frac{v}{r} \tag{100}$$

where v is the velocity of the car.

Finally, with the robot set up, a test track was made in order to compare our different algorithms. A photo of the real-life test track can be seen below in Figure (28). Due to limited space, this track was chosen in order to include at least one left turn, one right turn, and one straight region.

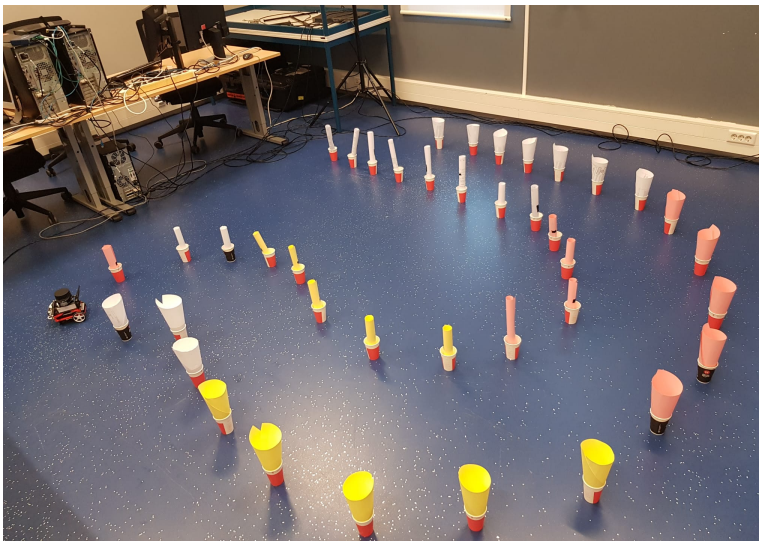


Figure 28: Photo of the real-life test track designed to test and compare the self-driving software. Due to limited space, this track was chosen in order to include at least one left turn, one right turn, and one straight region.

4.2.2 Results

To make a comparison between the success of the four self-driving car algorithms, some evaluation metrics must be chosen. The reward function designed for the simulation cannot be directly applied in real life because we lack the necessary information to calculate the reward at each time step. Instead we can record the lap time that the car achieved on the test track, whether the car crashed into a cone or not, and by recording the steering angle of the car over time we can determine the smoothness of the cars journey. The smoothness of the cars journey was calculated by the same method as in the simulation, by taking the difference between the current steering angle and the exponentially weighted moving average of previous steering angles.

However, once the robot had been adequately set up with the necessary modifications discussed in the previous section, all four algorithms succeeded in driving around the track without crashing into the cones. Thus, it is not a useful evaluation metric for comparison in this case. Therefore, the speed and smoothness of the cars journey are the principal methods of comparing algorithms for the robot.

For each self-driving car algorithm, three attempts at driving around the track were made and the lap times and angular velocities were recorded. The results can be seen in Table (6).

Method	**Mean Smoothness**	**Mean Lap Time**
Guide Policy	0.265	43.67s
OIL	0.104	43.32s
RL	0.102	40.14s
COIL	**0.082**	**39.53s**

Table 6: Table of results for four self-driving car algorithms tested by a robot; a guide policy, Offline Imitation Learning (OIL), Reinforcement Learning/PPO (RL), and Contextual Online Imitation Learning (COIL). The algorithms are judged on speed, smoothness, and safety. It can be seen that COIL offers the best overall result of the four methods.

Table (6) shows that COIL offers the fastest and smoothest journey of all four self-driving car algorithms. This provides evidence that COIL is capable of effectively solving more complex and difficult challenges. Despite the change of environment, agent, and observations, COIL still achieves strong results as a self-driving car algorithm. Furthermore, it is clear that all four algorithms were capable of solving the task of driving autonomously, which indicates a significant level of robustness. Before we continue with our analysis, there some additional qualitative observations made during the tests. These observations can be seen in Table (7).

Method	Qualitative Observations
Guide Policy	All three laps were completed successfully, with the robot staying relatively close to the middle as expected. Trajectory through the track was overly cautious. The robot struggled to create a smooth path, swaying side to side. No evidence of corner-cutting or competitive behavior.
OIL	All three laps were completed successfully. The robot followed smoother curves in its path than the guide policy, but it was less able to remain central in the track. No evidence of corner-cutting or competitive behavior.
RL	All three laps were completed successfully, however, the proximity to the boundary of the track was greatly reduced as the agent attempted to achieve a faster track time. The robot followed very smooth curves in movement, almost to its detriment.
COIL	All three laps were completed successfully. Very similar to regular Reinforcement Learning, however the agent seemed to have a better awareness of cutting corners to reduce lap time. Although the robot did not crash, this method seemed the closest to causing a crash due to the competitive edge it achieved in lap time.

Table 7: Table of qualitative observations for the testing of four self-driving algorithms on a robot: Guide Policy, Offline Imitation Learning (OIL), Reinforcement Learning (RL), and Contextual Online Imitation Learning (COIL).

It must be noted that the success of the self-driving car software on the robot is highly dependent on how realistic the simulation is during training. Aligning the dynamics of a simulation to reality is very important when testing algorithms. The accuracy and complexity of the simulation was limited by the resources at hand and scope of the project, and thus the results for the real-life robot have additional components to consider. By switching from simulation to reality, we are introducing a significant shift between the training data (from the simulation) and the test data (from the Lidar sensor), therefore these results allow us to examine the affect of transfer learning on our algorithms. Given this additional difficultly in the task, the results in this section are indicative that each algorithm exhibits robustness to transfer learning. Given more resources and time, the alignment between reality and simulation can be further improved, and the results will be more suitable for direct comparison.

These results show that COIL offers a method of effectively utilising guide policies even when the guide policy is both non-expert, and when faced with transfer learning. In this case, not only is the task much more difficult because of the transfer learning, but COIL is faced with the additional difficulty

that the actions from the guide-policy that it is using is *also* effected by the transfer learning. Despite the degradation of the guide policy, COIL remains the strongest algorithm at the task, which demonstrates that COIL is not only capable of utilising regular non-expert guide policies, but it can also withstand the effect of transfer learning too. There is consequently strong evidence that COIL is a robust and highly flexible method of effectively incorporating guide policies into Reinforcement Learning problems.

5 Critical Discussion

In this section we will discuss the potential shortcomings of Contextual Online Imitation Learning. Firstly, the computational cost of running the COIL agent in the environment is greater than regular Reinforcement Learning because it is necessary to compute the action of the guide policy at each time step (on top of the other computations in the Reinforcement Learning algorithm). This is the only additional computational cost of COIL as every other part of Reinforcement Learning algorithms remains the same when using COIL; only the interaction of the environment has any computational difference. In practice, the computational cost of computing the guide policies action is minimal and thus using COIL is only marginally more computationally expensive than regular Reinforcement Learning methods and consequently this is not a cause of concern for most researchers. However, in cases where calculation of the guide policies action is computationally heavy, the available computing power should be taken into account when using COIL. In this case, it might be more feasible for researchers to apply Imitation Learning techniques in order to achieve a more lightweight approximation of the guide policy.

Additionally, a potential shortcoming of COIL is the requirement of a *present* guide policy. If the guide policy cannot be queried during the exploration of the environment then it is not possible to use it in COIL. However, there is a suitable alternative given a guide policy that is not present. What the researchers can do is apply Imitation Learning techniques in order to generate a present guide policy as an approximation of the original non-present guide policy. In theory, any Offline Imitation Learning algorithm could be used for this purpose. Therefore, the requirement of a present guide policy is certainly a limitation of the proposed COIL method, however, there are certain workarounds to this limitation given the successful implementation of Imitation Learning. Furthermore, as COIL does not require an expert guide policy, it is still able to be used even if the Offline Imitation Learning is limited in capacity. It can now be seen that theoretically, even a non-present and non-expert guide policy can still be utilised with COIL, which further reinforces the notion that COIL is a highly-flexible technique.

Finally, in some tasks it may be that using any Reinforcement Learning algorithm or Online Imitation Learning algorithm is not possible due to the task being too complex or the environment too unpredictable and dynamic. This could mean that only Offline Imitation Learning methods such as Behavioural

Cloning applied to an expert guide policy are suitable for the task. As COIL is an online method, it requires interaction with the environment and therefore it might also be unsuitable in this scenario. Future research could analyse the capability of COIL to operate in such complex and unpredictable environments, even more so than the robot environment discussed in Section (4.2).

6 Conclusion

Reinforcement Learning is a vast and important sub-field of Artificial Intelligence capable of huge potential for future applications in our modern society, especially with self-driving car technology on our doorsteps. The methods of combining guide policies to Reinforcement Learning algorithms remains an active area of research with no one method shining above the rest.

This paper began by presenting the key theories and concepts in the field of Reinforcement Learning. Starting with the Markov Decision Process, this paper described the important mathematical constructs such as value functions, optimal policies, and Bellman equations, which paved the way for algorithms such as value iteration and policy iteration to be later developed. Following this, the most influential Reinforcement Learning algorithms were analysed in depth, such as Q-learning, Policy Gradient methods such as REINFORCE, and Actor-Critic methods such as A2C. After this, the most important Deep Reinforcement Learning and Imitation Learning methods were discussed and compared at length.

The goal of the Theory & Methods section was to provide a clear foundation for the mathematical material necessary to explore the new results introduced later in the paper. In order to give a coherent and relevant overview of this material, a necessary process of collecting and refining a selection of literature was conducted, and a consistent system of notation was made to bring the different concepts together into one clear format. Almost all of the theory in this section has re-written theorems, proofs, and algorithms in order to make the necessary clarifications and piece it all together. Furthermore, all diagrams, figures, and pseudo-code in this section were designed specifically for this paper. Therefore, a significant contribution to the Theory & Methods section was the rewriting of literature from different authors and piecing it all together with the same notation, style, and format.

Following the Theory & Methods section, a new technique of utilising guide policies in Reinforcement Learning tasks was introduced. In particular, Contextual Online Imitation Learning (COIL) offers researchers a powerful technique of incorporating guide policies into the learning process of Reinforcement Learning algorithms with a simple modification of the observation space. Furthermore, Dynamic COIL has significant potential to solve complex multi-task Reinforcement Learning problems. In this paper, COIL has been demonstrated to provide significant improvement over traditional Imitation Learning, Deep Reinforcement Learning, and the also the guide policy itself. In both the self-driving car simulation and the real-life robot this was the case. Additionally, the

results from the robot experiment demonstrated that COIL has strong ability to withstand the effect of transfer learning.

Future research on COIL could focus on three main topics not treated in this paper. Firstly, more research could be conducted to analyse the effect of using multiple guide policies with COIL. In particular, it would be interesting to analyse the deviation between the agents actions and the guide policies actions during exploration of the environment to see which of the guide policies are being 'listened to' in which states. Secondly, future research on COIL could incorporate Reward Function Coupling into the algorithm, where on top of providing the actions of the guide policy as an observation, the agent is rewarded proportionally to how similar its actions are to the guide policies. It remains an open question into how effective this might be. Finally, future research could focus on implementing and testing dynamic COIL in a multi-task setting.

A potential limitation of COIL is that access to a guide policy is required, however, this is also true for all Imitation Learning methods. In fact, COIL is more flexible than traditional Imitation Learning methods because a greater range of guide policies can be effectively utilised. As discussed in Section (3), even if the guide policy acts optimally only in a subset of states in the state space, the agent can learn to 'listen' to the suggestions of the guide policy only in this subset of states. This is a significant advantage over classical Imitation Learning methods. Consequently, COIL has great potential for solving Reinforcement Learning tasks in practice as it gives researchers and engineers a method of utilising their guide policies even if they are not experts in all states or all objectives. Furthermore, the results in Section (4) demonstrated that COIL offers very strong results in practice and also performs well under transfer learning.

References

[1] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[2] Ronald A Howard. Dynamic programming and markov processes. 1960.

[3] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[4] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4: 237–285, 1996.

[5] Dimitri P Bertsekas et al. Dynamic programming and optimal control 3rd edition, volume ii. *Belmont, MA: Athena Scientific*, 2011.

[6] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[7] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[8] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[9] R Williams. A class of gradient-estimation algorithms for reinforcement learning in neural networks. In *Proceedings of the International Conference on Neural Networks*, pages II–601, 1987.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[11] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[12] J Schulman, F Wolski, P Dhariwal, A Radford, and O Klimov. Proximal policy optimization algorithms. corr. 2017; abs/1707.06347, 2017.

[13] Daniel Conrad Halbert. *Programming by example*. PhD thesis, University of California, Berkeley, 1984.

[14] Nishanth Kumar. The past and present of imitation learning: A citation chain study. *arXiv preprint arXiv:2001.02328*, 2020.

[15] Kai Arulkumaran and Dan Ogawa Lillrank. A pragmatic look at deep imitation learning. *arXiv preprint arXiv:2108.01867*, 2021.

[16] Mark J Schervish and Morris H DeGroot. *Probability and statistics*. Pearson Education London, UK:, 2014.

[17] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[18] Jonathan Baxter and Peter L Bartlett. Direct gradient-based reinforcement learning: I. gradient estimation algorithms. Technical report, Citeseer, 1999.

[19] Jonathan Baxter, Lex Weaver, and Peter L Bartlett. Direct gradient-based reinforcement learning: Ii. gradient ascent algorithms and experiments. *preparation, July*, 1999.

[20] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL `https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

[23] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *In Proc. 19th International Conference on Machine Learning*. Citeseer, 2002.

[24] Ju-Seung Byun, Byungmoon Kim, and Huamin Wang. Proximal policy gradient: Ppo with policy gradient. *arXiv preprint arXiv:2010.09933*, 2020.

[25] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems*, 1, 1988.

[26] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[27] Romain Pepy, Alain Lambert, and Hugues Mounier. Path planning using a dynamic vehicle model. In *2006 2nd International Conference on Information & Communication Technologies*, volume 1, pages 781–786. IEEE, 2006.

[28] Ikechukwu Uchendu, Ted Xiao, Yao Lu, Banghua Zhu, Mengyuan Yan, Joséphine Simon, Matthew Bennice, Chuyuan Fu, Cong Ma, Jiantao Jiao, et al. Jump-start reinforcement learning. *arXiv preprint arXiv:2204.02372*, 2022.

[29] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[30] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

A Appendix

A.1 Guide Policy

The first step for the guide policy (autonomous path planning system) is generating a path for the agent to follow, where the path consists of a set of 'targets' denoted by $\tau := \{\text{Target}_1, \dots \text{Target}_k\}$ for the car to pass through one by one. The second step is simply to calculate which target is closest and drive towards it in some manner.

To autonomously drive towards this nearest target the car must steer so that it is perfectly 'lined up' to hit the target. Mathematically, this implies that the car needs to iteratively alter the direction it's facing so that it matches with the straight line drawn between the car and the target. We can denote the required change in angle by α. Additionally, we denote the distance between the car and the cone by d. Figure (29) below depicts this situation. The car angle is measured starting from the positive x-axis with the anti-clockwise direction representing a positive angle.

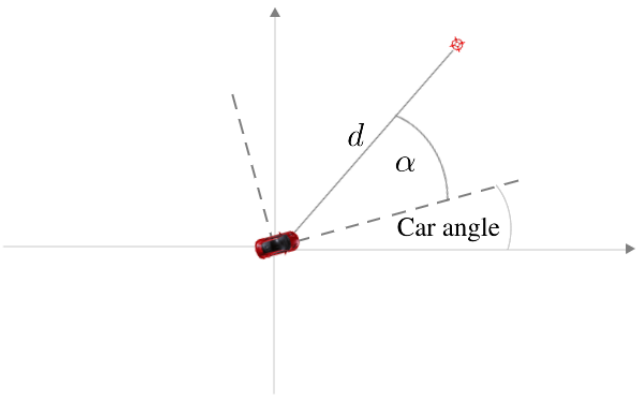


Figure 29: Depiction of the dynamic co-ordinate system fixed to the car (dashed lines), used to calculate the required turning angle α to steer the car towards the nearest target. d represents the distance between the car and the target and the car angle is the direction the car is facing measured from the x-axis.

To calculate α we first make a co-ordinate change so that the origin becomes the position of the car and the new x-axis is the direction that the car is facing. This can easily be done by subtracting the cars position from any given co-ordinates and then multiplying the result by a general rotation matrix with angle equal to the car angle. This new co-ordinate system can be seen in Figure (29) as the dashed lines. Let the original co-ordinates of the target be denoted by (x, y) and the co-ordinates of the target in the new co-ordinate system by

$(\tilde{x}, \tilde{y})$. We can now calculate α as:

$$\alpha = \arctan\left(\frac{\tilde{y}}{\tilde{x}}\right) \quad . \tag{101}$$

This new co-ordinate system also helps us calculate d:

$$d = \sqrt{\tilde{x}^2 + \tilde{y}^2} \quad . \tag{102}$$

Now that the desired turning angle α has been calculated we need to calculate a steering angle for the car. The steering angle, denoted by β, determines the change in car angle from one iteration of the simulation to the next. It is not enough to set the steering angle equal to α as this would cause a very sudden direction change for the car, and there is also the possibility that α is very large and cannot be achieved by the car immediately (e.g. if the target is behind the car!). Using α and the distance between the car and the nearest target d, a function $f(\alpha, d)$ to generate the steering angle β can be derived.

There are 3 properties we desire for this function f:

i) $\beta \in [-Max_{\text{Steering}}, Max_{\text{Steering}}] \ \forall \alpha, d$ where Max_{Steering} is the maximum possible steering angle.

ii) As α increases, the steering angle β also increases.

iii) As d increases, the more gradual the car's steering can be, and thus β decreases.

With these properties in mind we arrive at the function

$$\beta = f(\alpha, d) = \frac{2 \cdot Max_{\text{Steering}}}{\pi} \cdot \arctan\left(\frac{\alpha}{d}\right) \quad . \tag{103}$$

Figure (30) below displays the shape of f for $Max_{\text{Steering}} = 90$ and $d = 10$.

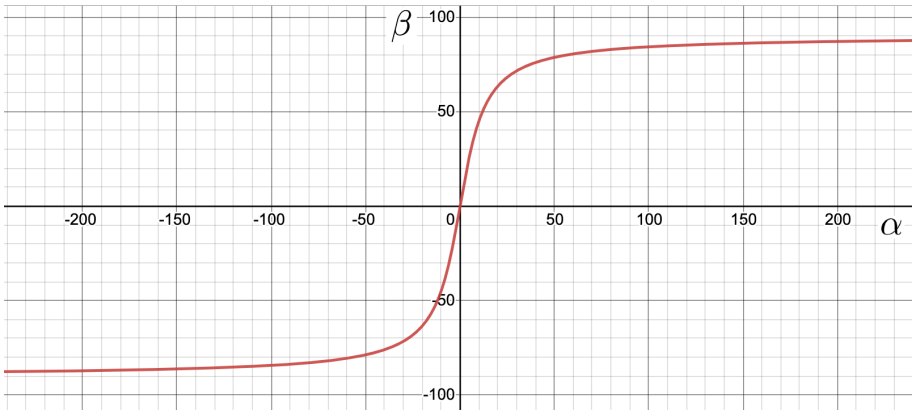


Figure 30: The shape of the function f which takes the required turning angle α (x-axis) and the distance to the nearest cone d and returns the steering angle β (y-axis) the car must take in order to eventually arrive at the target.

With the full calculation of β now set in place, the agent can now drive through a sequence of targets one at a time. The next step in the path-planning system is to find out which sequence of targets to follow, which is when cubic splines become essential. An explanation of cubic splines can be seen in Appendix (A.2).

The path planning system generates the path of targets by first collecting the list of visible cones, and then constructing a list of midpoints between left cones and right cones (corresponding to the left and right sides of the track). In general, this set of midpoints will form a point cloud in the middle of the track. The agent can now compute a cubic smoothing spline through this point cloud of midpoints. This spline can now be evaluated at equidistant points along the curve to generate a set of 'targets' for the car to follow, a process referred to as *discretization*. This three step method of using cone information to generate a set of targets can be seen in Figure (31) below.

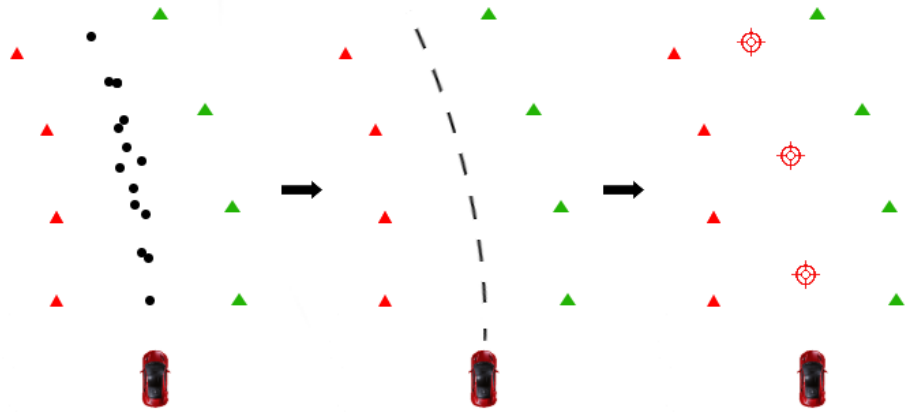


Figure 31: The process of using visible cone locations to compute a set of targets using cubic splines. Firstly (left), the midpoints between each pair of left cones (red) and right cones (green) are calculated. Secondly (middle), a cubic smoothing spline is generated using the midpoints as input. Lastly (right), this cubic spline is discretized into a finite set of targets for the autonomous car to follow.

Finally, to turn this into an online system we need to recompute this cubic spline every time a new cone is detected, giving the agent more information about the upcoming track. An overview of this online system can also be seen below in pseudo-code.

Algorithm 8 Path planning

$\tau \leftarrow \emptyset$
$Autonomous\ driving \leftarrow True$

while $Autonomous\ Driving = True$ **do**
  if $\tau \neq \emptyset$ **then**
    $Drive\ through\ targets\ in\ \tau$

  if $New\ cone\ detected$ **then**
    $midpoints\ list \leftarrow \{midpoints\ of\ all\ left/right\ cones\}$
    $New\ spline \leftarrow Compute\ Spline\ (midpoints\ list\ ,\ \lambda)$
    $\tau \leftarrow Discretization\ (New\ spline)$

There are a few subtleties in the path planning system not included in the above pseudo-code for the sake of clarity and brevity (for example the mechanism that governs which targets have been reached by the agent and thus need to be removed from the target list), however, all of the key elements are addressed. Additionally, to estimate the boundaries of the track, cubic splines can also be used to interpolate the cones themselves, as can be seen in the final simulations. To see a live simulation of the path planning system in action, the reader is strongly encouraged to visit the github repository : `https://github.com/alex21347/Cubic_Spline_Path_Planning`.

A.2 Cubic Splines

Splines first and foremost offer a powerful method of modelling relations of the form

$$y_i = f(x_i) + \epsilon_i \ , \tag{104}$$

where the ϵ_i are independent random variables with mean 0 and variance σ^2 representing the noise in the data.

Splines are particularly effective at modelling relationships with vastly different behavior across different regions of the domain of the explanatory variable. In essence, splines are piecewise polynomials joined together smoothly at *knots*. What we consider to be smooth in this context depends upon the order of the spline.

Definition A.1. Let $a < \xi_1 < \cdots < \xi_k < b$ be fixed points called knots. Let $\xi_0 = a$ and $\xi_{k+1} = b$. Then a spline of order d is a real-valued function on $[a, b]$, $f(x)$, such that

(i) f is a piecewise polynomial of order d on $[\xi_i, \xi_{i+1})$ $for\ \ i = 0, 1, \ldots, k$

(ii) f has $d - 2$ continuous derivatives

From this definition one can see that splines are simply piece-wise polynomials with some additional constraints to preserve smoothness of the resulting function. When $d = 3$ we arrive at the *cubic spline*.

A spline of odd degree $d = 2r-1$ is called a *natural spline* if it is a polynomial of degree $r - 1$ outside the range of the knots. Therefore, a *natural cubic spline* is a cubic spline which is linear outside the range of the data. Natural cubic spines are usually a robust choice of model if one wants to make predictions outside the range of the training data.

The method we now use to fit the piece-wise polynomials to the data will now determine what type of spline we create. A *regression spline* is the piece-wise polynomial $f(x)$ which minimises the *residual sum of squares* (RSS)

$$RSS = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f\left(x_i\right)\right)^2 \tag{105}$$

subject to the constraints in Def. (A.1).

The regression spline can now be shown to be just a particular case of the more general *smoothing spline*. The relationship between regression splines and smoothing splines is analogous to the relationship between linear regression and ridge regression. Just as ridge regression introduces a penalty term on the norms of the model parameters with the goals of regularisation and improved generalisation, smoothing splines add a penalty term on the norms of the derivatives of the piece-wise polynomial, with the exact same goals in mind.

Thus, the smoothing spline of order d is the piece-wise polynomial $f(x)$ which minimises the *penalised residual sum of squares* (PRSS)

$$PRSS = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f\left(x_i\right)\right)^2 + \lambda\int_a^b\left(f^{(d-1)}\right)^2 dx \tag{106}$$

where λ is the *smoothing parameter*.

The smoothing parameter has a significant qualitative effect on the resulting smoothing spline. As $\lambda \to 0$ we impose no smoothness penalty and end up with a very close fit to the data, however the resulting curve could be very noisy as it follows every detail in the data. As $\lambda \to \infty$ the smoothness penalty dominates and the solution converges to the ordinary least squares line, which is as smooth as you can get (with second derivative always 0), but may be severely underfitting. The cubic smoothing spline is for when d = 3

Surprisingly, it can be shown that minimizing the PRSS for a fixed λ over the space of all continuous differentiable functions leads to a unique solution. We will now turn our attention to how we can compute this unique solution for a cubic smoothing spline of fixed λ.

The penalty may be written as a quadratic form

$$\int_a^b\left(f''(x)\right)^2 dx = \mu' K\mu$$

where $\mu = f\left(x_i\right)$ is the fit, K is an $n \times n$ matrix given by $K = \Delta' W^{-1}\Delta, \Delta$ is

an $(n-2) \times n$ matrix of second differences, with elements

$$\Delta_{ii} = \frac{1}{h_i}, \Delta_{i,i+1} = -\frac{1}{h_i} - \frac{1}{h_{i+1}}, \Delta_{i,i+2} = \frac{1}{h_{i+1}}$$

W is a symmetric tridiagonal matrix of order $n-2$ with elements

$$W_{i-1,i} = W_{i,i-1} = \frac{h_i}{6}, w_{ii} = \frac{h_i + h_{i+1}}{3}$$

and $h_i = \xi_{i+1} - \xi_i$, the distance between successive knots. This implies that we can compute a smoothing spline as

$$\hat{f}(x) = (I + \lambda K)^{-1} y$$

Proof. : Write the penalized residual sum of squares as

$$\text{PRSS} = (y-\mu)'(y-\mu) + \lambda \mu' K \mu$$

Taking derivatives

$$\frac{\partial \text{PRSS}}{\partial \mu} = -2(y-\mu) + 2\lambda K \mu$$

Setting this to zero gives

$$y = \hat{\mu} + \lambda K \hat{\mu} = (I + \lambda K)\hat{\mu}$$

and pre-multiplying both sides by $(I + \lambda K)^{-1}$ completes the proof. □

This computation using matrices K, Δ and W allows for efficient implementations of computing the smoothing splines. For implementation of smoothing splines in Python the library SciPy was used, which has many well designed functions for creating smoothing splines. Similarly, in R, one can use the inbuilt function smooth.spline() to fit smoothing splines to data.

Finally, with the cubic smoothing spline properly introduced we may now define the *self-evolving cubic spline*.

Definition A.2. Let S be an online system where new data points (x_i, y_i) are 'discovered' at times t_i with $0 \leq t_0 \leq \ldots \leq t_n \leq t$. Then the self-evolving cubic spline is a cubic smoothing spline, which upon the discovery of a new data point (x_i, y_i) at time t_i, re-computes the entirety of the cubic smoothing spline in light of the new information.

The self-evolving cubic spline can be used in any online system that continuously gathers information.

A.3 Robot Implementation Details

In this section we will discuss some additional robot implementation details encountered in this paper.

Firstly, we shall discuss the communication scheme between the robot and the computer. Robot Operating System (ROS) was used to communicate between the computer and the robot. Here, the robot would send Lidar data to the computer, and the computer would send back the instructions for the next moment in time. The computer used here was using the Linux (Ubuntu 18.04) operating system. Ideally, the Python scripts used to compute the agents actions would run on this computer, however, the Python scripts required Python version 3.8.5, and this was not possible to get on the current version of Linux on the computer. The workaround that was used was to run the Python scripts on a nearby laptop with the correct Python version, and send the desired actions back to the computer using User Datagram Protocol (UDP), which refers to a protocol used for network communication over the internet. This communication system was all designed by hand in Python and can be found on Github at : `https://github.com/alex21347/Self_Driving_Car`. This communication system is illustrated below in Figure (32).

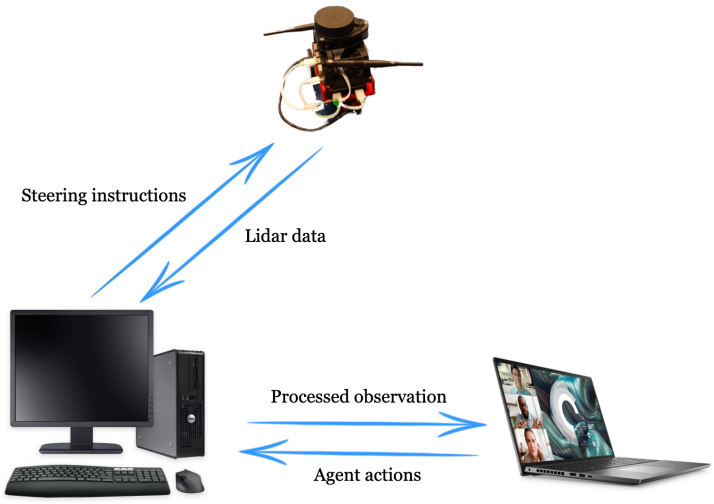


Figure 32: Illustration of the communication scheme used to receive Lidar data from the robot, and send instructions back from the computer. Due to incompatible operating systems and Python versions, a third computer (laptop) was used to process the actions of the Reinforcement Learning agent.

In the end, this system required 13 Python programs running in parallel over 3 different computers and 3 Python versions. Given the constraints of the operating systems and Python versions, this became the best solution at hand.

The second important detail we shall discuss is regarding the construction of the observation for the Reinforcement Learning agent. Recall from Section (4.1.1) that the observation to the agent is a cubic-spline interpolation of the boundaries of the track. In the simulation, the cubic splines were always made with no issues, however, when the same script was used on the robot, some errors became apparent. This is because of two reasons. Firstly, the cone classifier was not 100% accurate, which meant that sometimes the boundary of the track was misinterpreted and this caused errors within the cubic-spline Python code. Secondly, the cubic-spline interpolation required that *at least* two left cones and two right cones could be seen by the agent at any one point in the environment. This held true in the simulation but not in real life. This caused the Python script to through an error and crash.

To fix this, a buffer was created where the simulation would store the last known valid observation to the agent. An observation was deemed valid if there were at least two left cones and two right cones, and if the values of the observation were within a suitable and realistic range (If the boundary of the track is calculated to be over 1km away, it is safe to say that the cubic-spline script failed due to one of the aforementioned difficulties). Following this, if the current observation is deemed invalid, then the observation stored in the buffer would be fed to the agent, and the agent would send back the same action as before. Therefore, to get the robot working it was necessary to ignore some of the agents actions when the track was massively misinterpreted. This could have been avoided if more sensors were used than just a Lidar.