Robust Learning for Autonomous Agents in Stochastic Environments

Thesis presented by

Ugo LECERF

for the degree of Doctor of Philosophy

Renault Software Labs
Sorbonne Université – EURECOM
Industrial Supervisors: Sébastien Aubert, Christelle Yemdji-Tchassi
Academic Supervisor: Pietro Michiardi
Reviewers: Elena Baralis, Marco Lorenzi

Valbonne, France
October 26, 2022

Abstract

When learning to act in a stochastic, partially observable environment, an intelligent agent should be prepared to anticipate a change in its belief of the environment state, and be capable of adapting its actions on-the-fly to changing conditions. This kind of behaviour is vital for agents in order to act correctly in novel environments, and is par for the course of developing 'intelligent' autonomous agents. A high degree of adaptability is often attributed to humans as being one of the traits which separates us from the rest of the animal (and robot) kingdom. When learning to tackle tasks in which risk is present, we are able for example to form contingency plans in the face of an uncertainty to ensure we are prepared for undesirable outcomes. This allows us to deal with tasks requiring fast, on-the-fly decision-making in order to counteract high degrees of uncertainty. This is especially the case for autonomous vehicles (AVs) in a road navigation environment, where inputs to the AV are both uncertain and constantly evolving. In a context where safety is paramount and the risk of catastrophic failure is high, a strong ability to react to a changing environment is necessary.

In this work we explore data-driven deep reinforcement learning (RL) approaches for an autonomous agent to be robust to a navigation task, and act correctly in the face of risk and uncertainty. We investigate the effects that sudden changes to environment conditions have on an autonomous agent and explore methods which allow an agent to have a high degree of generalization to unforeseen, sudden modifications to its environment it was not explicitly trained to handle. Inspired by the human dopamine circuit, the performance of an RL agent is measured and optimized in terms of rewards and penalties it receives for desirable or undesirable behaviour. Our initial approach is to learn to estimate the distribution of expected rewards from the agent, and use information about modes in this distribution to gain nuanced information about how an agent can act in a high-risk situation. Later, we show that we are able to achieve the same robustness objective with respect to uncertainties in the environment by attempting to learn the most effective contingency policies in a 'divide and conquer' approach, where the computational complexity of the learning task is shared between multiple policy models. We then combine this approach with a hierarchical planning module which is used to effectively schedule the different policy models in such a way that the collection of contingency plans is able to be highly robust to unanticipated environment changes. This combination of learning and planning has shown promise in RL applications, and we are able to make the best of both worlds in terms of the adaptability of deep learning models, as well as the stricter and more explicit behavioural constraints that can be implemented and measured by means of a hierarchical planner.

Acknowledgements

Conducting this research project has been a great source of both learning and personal growth for me throughout its duration. I have had the pleasure and benefit of being accompanied by many individuals who have significantly contributed to this project, in both professional and personal qualities, without whom this work would not have been possible. For this reason I would like to extend my gratitude towards some individuals and organisations which have the most impacted my time as a PhD student.

First of all, I would like to extend my gratitude towards the Renault Software Factory path-planning team, most notably my supervisors Dr. Sébastien Aubert for helping to build up this thesis project, and Dr. Christelle Yemdji Tchassi for helping me to carry out this project till its completion. I would also like to thank EURECOM's data science department, and especially my PhD director Dr. Pietro Michiardi for providing valuable insight to the most promising directions of research, and for helping to publish this work to international venues.

On a personal note I would like to extend my appreciation to my fellow PhD students at both Renault and EURECOM who have been a great source of both support and motivation, especialy my deskmate Matthieu. I also more broadly wish to thank my other colleagues in Renault and EURECOM for the great work environment I have had the opportunity to be integrated into over the course of the last 3 and a half years.

I would further like to thank my a couple of my dearest friends Alestair and Eloi, who made the trip to attend my thesis defense, along with all those who took the time to attend online. Finally, I would like to give a warm recognition to my brothers and parents who have provided me with love and guidance and who, I hope, have been able to gain an appreciation for the qualities of life in southern France.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Approach to Uncertainty in decision-making

Making good decisions in the face of uncertainty is a difficult task. For an agent seeking the optimal behaviour in an environment, the presence of uncertainty may compromise the expected performance as well as make the optimal solution difficult to describe. In this work we are concerned with enabling an autonomous agent to be as robust as possible in the face of uncertainty, with respect to the possibility of high-consequence failure.

We can divide the uncertainties present in the environment into two categories: *aleatoric* and *epistemic*. Aleatoric uncertainty refers to the random nature of the dynamics such as the result of a dice, whereas epistemic uncertainty refers the uncertainty resulting from the lack of knowledge about the environment, for example in cases of partial observability. If we look at how human decision-making deals with both these kinds of uncertainties, we can see that we apprach them in a fundamentally different way. Aleatoric uncertainty is usually a question of expected value at the outcome, based on some threshold value we intrinsically have as a kind of 'decision boundary' for a level of uncertainty of outcome which we find acceptable. This boundary in and of itself depends the tolerance for risk which may change from person to person, and is also influenced by environmental factors. However in the case of aleatoric uncertainty it is impossible to be 100% certain of the outcome resulting from actions that are taken, and so some threshold for risk must be accepted in order for a decision to be made. Since epistemic uncertainty describes incomplete knowledge of the domain, often linked to the agent's perception capabilities (and may even vary between agents) it can be tackled by performing some exploratory actions in order to increase knowledge about the environment at hand, whether it be in terms of the state-space, or its dynamics. An example for this would be a vehicle advancing slowly at an intersection to test the intentions of other drivers

as to whether they are cooperative or not, which is impossible to know before actually going into the intersection. To this point, it is by definition possible to eliminate all sources of epistemic uncertainty through of sufficient exploration, although in practice we usually do not want to spend an excessive amount of time tackling this uncertainty and instead reduce it to some acceptable threshold before taking action.

The philosophy for autonomous agents to act in environments prone to randomness and uncertainty is the same; we wish to simultaneously reduce epistemic uncertainty through exploration and learning the environment dynamics by agent-environment interactions, as well as selecting trajectories which have a good performance while maintaining a low aleatoric uncertainty for outcome such that we are satisfied with the odds of success.

Although both aleatoric and epistemic have clear separate definitions, the category in which we should consider uncertainties present in the environment is often subjective. The determining factor is the level of detail and modeling we wish to have to describe environment dynamics. In the case of a dice-throw for example, we could theoretically say that knowing the speed, position, angular momentum of the dice along with the friction of the table surface, air density etc. along with the laws of motions we could then determine the outcome of the throw. However due to the high complexity and how sensitive the outcome is to small variations in initial conditions, we consider it as effectively random. A common example in autonomous navigation are the behaviours of other vehicles on the road: according to how we wish to model the problem, driver intentions can be considered initially random, or we can consider driver behaviour as some hidden variable which we can attempt to gather information about through exploratory actions in the environment. In reality there are few events that can be described as *truely* random - radioactive decay can be one such example. For this reason we model as aleatoric variables those which we do not wish to model otherwise.

Moreover in our work we also deal with the balance between performance and safety aspects of an agent's behaviour, specifically in an autonomous navigation environment. This means we want to strike a balance between having the safest possible action, without overly sacrificing the agent's performance metric. When considering safety as a primary objective in decision-making, a common problem that arises is known as the *freezing robot problem*. This freezing problem is common when designing autonomous control algorithms where a robot will spend so much time trying to plan ahead to reduce the risk and uncertainty of its actions that it will end up taking no action at all, which in the end is very sub-optimal. As in our case, in order for the autonomous vehicle (AV) not to freeze up in the face of an uncertain outcome, we must be able to accept some level of risk in order for the agent to learn some useful behaviours. We can clearly make a parallel with human behaviour where we take acceptable levels of risk every day, for example in driving situations, where we actually take a

relatively high amount of risk when considering the potential for negative consequences associated to the driving task. Therefore a central aspect when it comes to dealing with risky outcomes, is to prevent the agent from freezing up, and be able to correctly estimate the amount of risk it is taking when pursuing certain trajectories.

1.2 Levels of Autonomy for Self-Driving Cars

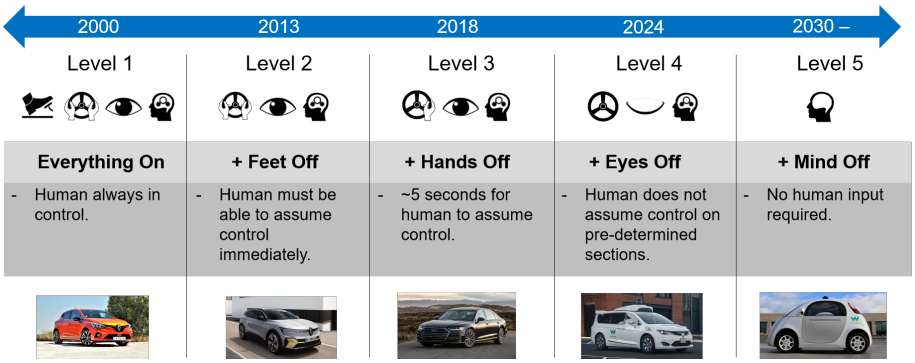


Figure 1.1: Level of autonomy for self-driving systems, represented as a progressively decreasing need for human physical inputs and degree of supervision.

Figure 1.1 shows how levels of autonomy ranging from 1-5 describes the degree of transfer of responsibility from the human to the autonomous system for a self-driving vehicle. Level 1 autonomy usually refers to driver assistance systems which help in the driving task but generally do not allow the driver to relinquish any of the usual physical inputs or attention from the driving task. Today, many car models are being released with level 2 automation which can take over longitudinal control for example, however always require a strong element of human supervision. Some systems approaching level 3 are being released into the market, however there is still much work to be done on both technical and legal frameworks for vehicles with higher levels of automation to be released onto public roads.

1.3 Problem Statement

The problem we are tackling in this work relates to the uncertainties inherent to the sensors that an AV uses to perceive its environment, and how we may use information about sensor uncertainties in order to increase the robustness of the AV decision-making system. AV systems are typically built from a pipeline

of individual components, linking sensor inputs to motor outputs. Raw sensory input is first processed by object detection and localization components, resulting in scene understanding. Scene understanding can then be used by a scene prediction component to anticipate other vehicles' motions. Finally, decision components transform scene predictions into commands that instruct AVs trajectories and short-term movements. Any errors in the input to the autonomous navigation pipeline will propagate through the system and are susceptible of causing a catastrophic failure by the agent, due to the sensitive and high-risk nature of the driving task.

In the context of this thesis project, the focus is on the development of new methodologies for dealing with the probabilistic nature of input sensor data and generating probabilistic outputs which should be robust to the environment uncertainties present, such that both safety considerations and objectives can be systematically fulfilled. For this task we focus our approach on the domain of *probabilistic reinforcement learning*, a branch of machine learning, which is able to deal with control tasks in high-dimensional environments with complex and unpredictable dynamics. With this computational approach we are able to learn to control an autonomous agent by learning from sampled agent-environment interactions. This allows us to use a simulated environment, capable of simulating a large number of interactions in a relatively short amount of time, such that an agent may learn how to deal with the stochastic nature of the autonomous navigation task in different contexts.

We have mentioned the balancing act that needs to happen between performance and safety criteria. the performance metric is classically implemented as blocs representing varying levels of temporal abstraction in the driving pipeline, which we illustrate in figure 1.2, showing the typical 2 (sometimes 3) functional blocs that make up the behavioural system for an AV.

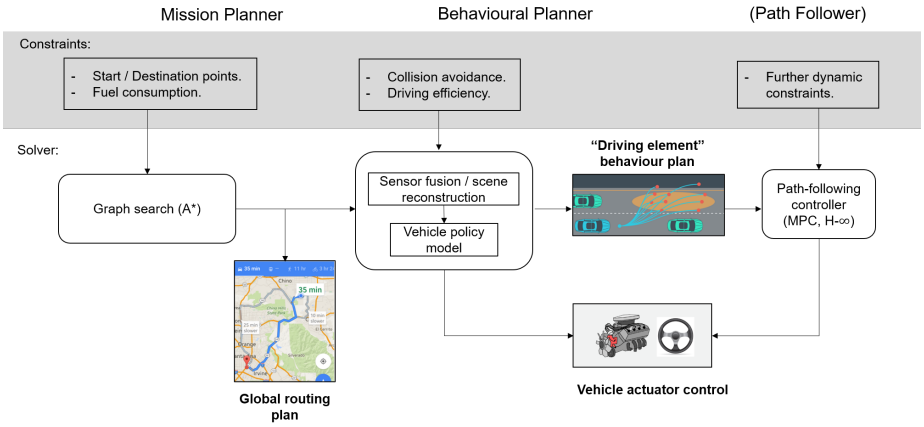


Figure 1.2: Decision-making pipeline for autonomous navigation.

The mission planner is the highest level of trajectory planning and usually corresponds to directions such as which roads to follow or which intersection to go through in order to arrive at the desired destination; In terms of human planning, this would be like reading a map. The behavioural planner is tasked with providing control over lower-level actions such as changing lanes, or accelerating to pass through an intersection, the human equivalent of which would be the decisions taken mentally to navigate through the environment. The lowest-level abstraction for control is the trajectory planner which is responsible for providing low-level actions such as steering wheel angle and desired acceleration values which interfaces directly with the vehicle's raw inputs. This lowest-level bloc is equivalent to a human's muscle control for turning the steering wheel or pressing on the acceleration pedal. In some cases where the system dynamics are too complex for the behavioural planner to output the actuator controls, it instead provides a behavioural plan in the form of a path to be followed, which is then fed into a path-following controller transforming that behavioural plan back into the lowest levels of control to be provided to the vehicle actuators.

In our problem statement we only consider the behavioural planning module, such that we assume the mission planner is able to provide us with a valid objective, and the trajectory planner is able to reproduce the desired trajectories with sufficient precision. These two blocs usually each come with their respective safety constraints which are dealt post-trajectory generation. For safety considerations related to the behavioural planner, there are two approaches: either have a form of post-planner checks (high-level safety checks) like for the other planners, or try to directly integrate the safety criteria in the behavioural algorithm during optimization, or the training process of a learning algorithm. Both approaches are possible, having a varying amount of hierarchical structure, and possibility for nuanced optimization. One of the aims of using a machine learning approach is to be able to integrate the safety considerations directly into the behavioural planner such that the robustness of the autonomous agent's decision making is increased.

Due to the industrial component of this work, an important consideration is that of degree of confidence for actions recommended by the autonomous navigation system, or equivalently being able to provide an accurate estimate for the probabilities of outcomes according to scene uncertainties. In order for AVs to be legally allowed on roads alongside human drivers, we must have strong assurances that the decisions made by the control algorithm are correct and minimize the safety risk to other road users. Unfortunately this is not one of the strong points for reinforcement learning in stochastic environments, which relies on expected, mean values of performance to guide behaviour, meaning that low-probability, and possibly high-consequence outcomes are difficult to react to. For this reason, we anticipate having to adapt the RL framework for autonomous navigation such that it may be good enough to deal with potentially high-risk situations.

Considering the above discussion, we define the following objectives which are the driving factors in our work:

- To integrate the notation of uncertainty in the decision-making pipeline, such that the agent is able to be as robust as possible to the variance of outcomes that arises when inputs are no longer known with high confidence.
- To perform a multi-objective optimization taking into account both safety and performance criteria, such that both are optimized during the learning process, and we are able to balance out satisfactory behaviours with sufficiently high confidence in the face of an uncertain environment.

1.4 List of publications & Thesis layout

The work done during this thesis has lead to a couple of publications, cited below:

Ugo Lecerf, Christelle Yemdji-Tchassi, Sébastien Aubert, and Pietro Michiardi. "Automatically Learning Fallback Strategies with Model-Free Reinforcement Learning in Safety-Critical Driving Scenarios". *International Conference on Machine Learning Technologies (ICMLT 2022)*, March 2022.

Ugo Lecerf, Christelle Yemdji-Tchassi, and Pietro Michiardi. "Safer Autonomous Driving in a Stochastic, Partially Observable Environment by Hierarchical Contingency Planning". *Generalizable Policy Learning in the Physical World Workshop (ICLR 2022)*, April 2022.

This manuscript contains an in-depth review and discussion of the work done as part of the CIFRE project in cooperation with Renault Software Factory, along with EURECOM, and is structured as follows:

Chapter 1 (this chapter) provides an introduction to the subject of decision-making under uncertainty, and its application to the context of self-driving vehicles in autonomous navigation tasks. Chapter 2 gives some background on RL algorithms and how deep learning can be used to learn policies, along with how a control task is modeled as a Makov decision process, in which the agent's behaviour is optimized. Chapter 3 presents how we use distributional reinforcement learning, where deep learning models aim to learn the distributions representing outcomes and behaviours in the stochastic environment. Chapter 4 presents a multi-agent approach to learning a robust collection of policies in an environment with uncertainties in dynamics and state-space, where modes of behaviour in the return distribution are instead instantiated as individual,

separate policies. Building on this, chapter 5 presents a hierarchical controller to schedule multiple policies available to the autonomous agent in such a way that they become robust to sudden, unforeseen changes in environment dynamics or agent observations. Finally, chapter 6 contains concluding remarks on our approach, along with perspectives and insights to future work which may follow the work presented in this manuscript.

Chapter 2

Background

2.1 Reinforcement Learning

2.1.1 Introduction

Reinforcement learning is a data-driven approach to control tasks, where the objective can be formulated as a signal to be optimized by the controller over the course of the control task, which can be of varying length (though typically finite). The RL approach is inspired by the way humans and animals appear to learn complex tasks where the notion of trade-offs due to temporal abstraction (i.e. sacrificing immediate rewards in order to obtain a greater rewards later in time) is an important consideration. Another main benefit of a data-driven approach is the ability to use a model-free approach and let functions with a high number of parameters (like neural nets) implicitly learn the complex, non-linear dynamics of a model through many interactions with the task environment.

One of the theories on what drives human behaviour, is our reaction to the dopamine signal released by the brain in response to some positive (or negative) stimulus from our environment. From observations of human and animal behaviour, it seems that this dopamine reinforcement circuit allows for complex patterns of behaviour to emerge for tasks where the optimal control is a mix of short and long-term actions. Although the drivers for human behaviour are much more complex than simply maximizing a single dopamine signal, reinforcement learning seeks to imitate this mechanism by means of an artificial reward signal given to a controlling agent, in order encourage complex emergent behaviours from a relatively simple problem formulation. This way if a positive signal is given to an autonomous agent (reward for finding its way out of a maze, for example), then the agent should seek out states which provide it with

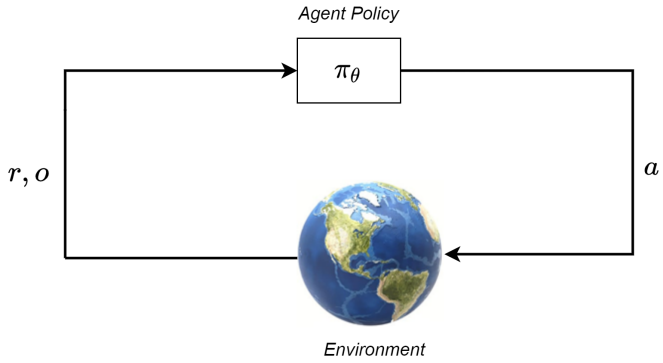


Figure 2.1: Basic reinforcement learning loop.

this reward signal it has previously encountered. The goal of RL applications is that we should be able to build a model of how good each possible action is in the current environment. Since we don't know what action the agent should take in order to have the best performance, we must use a form of unsupervised (or loosely-supervised) learning. As opposed to the supervised learning problem often formulated in machine learning applications where ground-truth labels are provided, these labels do not exist for individual actions taken in by the agent in each subsequent environment state, and so a reward signal is provided as a means of finding the best action in a state considering the ability for an agent to perceive the highest possible rewards over the course of a sequence of states (also referred to as an episode).

Therefore, the central concept of RL is to learn the 'goodness' of actions in a given state, using an estimation for the expected value of reward the agent will receive for taking actions in the environment. The focus on long or short-term rewards can be balanced according to how future rewards are weighted with respect to immediate ones. From here we can formulate the rough objective of an RL algorithm:

We denote the reward perceived by the agent at each time-step, r_t. Over the course of an episode an agent will see a sequence of rewards $\{r_0, r_1, ..., r_T\}$ until it reaches a terminal state, where the episode is ended. The sum of rewards seen by the agent during the episode is termed as the *return*: $G_t = \sum_{i=t}^{T} r_i$. Although we could aim for an agent to maximize this quantity, there is no distinction made between short-term rewards which are more certain, and long-term rewards which may be less certain for the agent. Due to the stochastic nature of tasks we tend to prefer certain, immediate rewards over long-term rewards, and this is modelled using a *discount factor* $0 < \gamma < 1$ in order to reduce the weight of long-term rewards. So when an agent estimates the sequence of rewards it

will achieve we can actually model it with: $\{r_0, \gamma r_1, \gamma^2 r_2, ..., \gamma^T r_T\}$, and the discounted return can now be written as

$$G_t = \sum_{i=t}^{T} \gamma^i r_i. \tag{2.1}$$

Actions are taken by an agent's *policy*, denoted π, such that $a = \pi(s)$ according to the state s in which the agent is currently in. Finding the optimal policy π^* which maximizes the expected discounted returns is our reinforcement learning problem:

$$\pi^* = \arg\max_{\pi} G_t, \forall t \in [0, T]. \tag{2.2}$$

Note that the policy should be optimal for any starting time-step t. The basic objective represented by (2.2) is the starting point for any reinforcement learning agent, and there are multiple possible approaches for solving this.

Figure 2.2 shows an example 5-state trajectory, consisting of a sequence of states $\{s_{0:4}\}$ with rewards associated to some of the transitions. In this case the discounted return is computed as: $G_0 = \gamma^0 \cdot 0 + \gamma^1 \cdot 0 + \gamma^2 \cdot 1 + \gamma^3 \cdot 0 + \gamma^4 \cdot 3 = \gamma^2 + \gamma^4 \cdot 3$. For example using $\gamma = 0.8$, then we can compute $G_0 \approx 1.869$ for this particular state sequence starting at s_0.

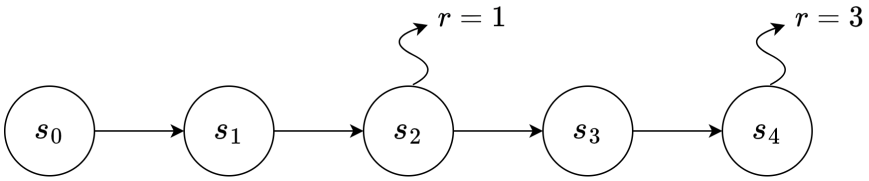


Figure 2.2: Example 5-state trajectory through an MDP.

Figure 2.3 shows the effect that the discount factor γ may have on the choice of action for an RL agent. In this situation the agent must choose between two actions a_0 or a_1 to choose between two different state trajectories. Using $\gamma = 0.8$, we may compute that the discounted return after for each action is: $G_0^{a_0} = 2.048$, $G_0^{a_1} = 1.6$, and we will choose action a_0 as the optimal. Repeating this computation with $\gamma = 0.7$ however, yields $G_0^{a_0} \approx 1.201$, $G_0^{a_1} = 1.4$, meaning that a_1 is now considered optimal, and the agent will prefer the short-terms benefits over long-term ones. Since these example trajectories have no uncertainty of outcome, it is always better to use $\gamma = 1$ and favor long-term rewards, however in practice it is important to not over-value the possibility of distant rewards as they may end up being impossible to collect. The discount factor is also useful to bound the maximum return that an agent can collect, and allows for proof of convergence even in cases of infinite length (i.e. non-terminating or continuous tasks).

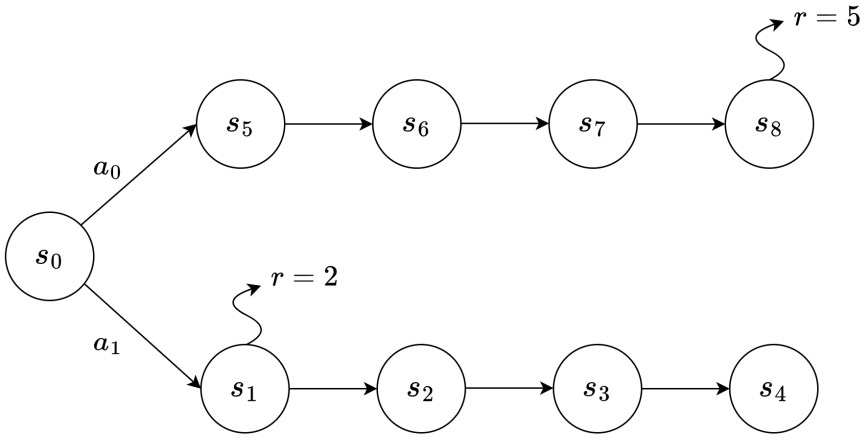


Figure 2.3: Agent may choose between two actions, to chose between two possible trajectories.

In practice, episodes are usually longer than 5 states, and so values for γ will commonly be of the form $1 - \varepsilon$ where ε is some small value close to 0. For example $\gamma = 0.99$ will reduce rewards by a factor of 10 after approximately 200 steps. In the next sections we present the mathematical framework used in deep reinforcement learning, along with the two 'families' of RL algorithms, and some standard tricks and add-ons which are standard in RL applications.

2.1.2 Standard Notation

Markov Decision Process

The mathematical framework for applying a reinforcement learning algorithm is to model the environment as a Markov Decision Process (MDP). This approach models the environment as a directed graph (Figure 2.4 illustrates a simple 3-state MDP), where nodes correspond to states in which the agent finds itself, and edges represent the actions an agent can take to transit between nodes. The model for transition between states is given by the dynamics of the environment that this modeled. An important hypothesis that is made about environment dynamics is the *Markov property*, which states that the dynamics cannot depend on previous states encountered during the agent's state-space trajectory. This means an agent should be able to make the optimal decision using only the current state it is in ($a_t = \pi(s_t)$, instead of $a = \pi((s_t, s_{t-1}, ...))$. For many control applications however this hypothesis does not seem reasonable; for example in the case of an autonomous car it is impossible to deduce the speed

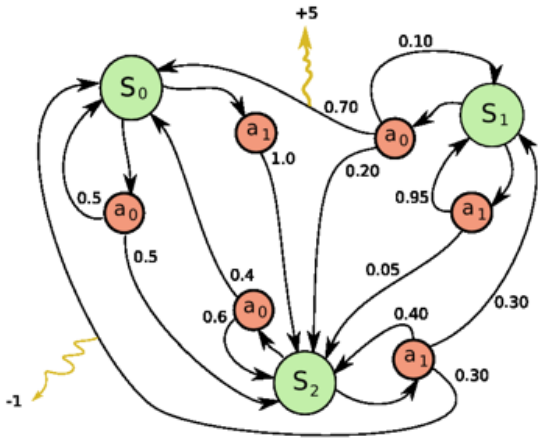


Figure 2.4: Markov decision process (MDP) as a directed graph

of a target from a still image. However in practice we can simply additional information such as speed and other time-dependant information such as length of the current trajectory, or even past agent actions, without invalidating the Markov property of the MDP. For video games where the input to an agent are the raw pixel values from which speeds of objects cannot be determined, it is common practice to aggregate the past 3 or 4 frames together as the input state to the control agent so that this information can be deduced from the input space.

A finite MDP is defined by the following elements:

- Finite set of states $s \in \mathcal{S}$. States are indexed by the time-step at which they are encountered: s_t.
- Finite set of actions $a \in \mathcal{A}$. Actions are also indexed by their respective time-steps: a_t.
- Transition model $s_{t+1} \sim \mathcal{T}(s'|s_t, a_t)$, representing the probability of passing from s_t to s_{t+1} after taking action a_t.
- Reward function $r_t \sim R(r|s_t, a_t, s_{t+1})$, representing the reward r_t perceived by the agent after transition from s_t to s_{t+1} with action a_t.
- Discount factor $\gamma \in (0, 1]$, controlling the weight in value of states further along the Markov chain.

Value-Based Agent

Actions in the MDP are taken by a policy $\pi : \mathcal{S} \to \mathcal{A}$ mapping states to actions. In order for the agent to maximize rewards in the MDP, we can model the 'goodness' of states as the estimation of the discounted returns the optimal agent is able to get from that state. This is a family of RL algorithms known as value-based learning, whereby the algorithm seeks to learn a good estimation of the discounted return G_t achieved by the agent's policy. The value of a state under a policy π, is estimated by the state-value function $V^\pi : \mathcal{S} \to \mathbb{R}$, which represents the expected future discounted sum of rewards, if policy π is followed from s:

$$V^\pi(s) := \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right], \tag{2.3}$$
$$s_0 = s, a_t = \pi(s_t), s_{t+1} \sim \mathcal{T}(s'|s_t, a_t), r_t \sim R(r|s_t, a_t, s_{t+1}).$$

An agent's policy will therefore take actions such that the value for V^π is maximized. In order for an agent to differentiate between available actions, instead of estimating the value function for the current state, we can estimate the value function for all possible future states (all states s_{t+1} attainable by the agent's actions), and choose the action a_t which leads the agent to the most desirable future next state s_{t+1}. This one step look-ahead dependant on the agent's action is known as the action-value function, or more commonly Q-function $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ which assigns values to actions according to the value of the states that are reached. Equation (2.4) is known as the *Bellman equation*, and is used to learn the Q-function from many samples of interaction between the agent and the environment.

$$Q^\pi(s_t, a_t) := \mathbb{E}_{s_{t+1}} \left[r_t + \gamma \cdot V^\pi(s_{t+1}) \right], \tag{2.4}$$
$$s_0 = s, a_0 = a, s_{t+1} \sim \mathcal{T}(s'|s_t, a_t), a_t = \pi(s_t), r_t \sim R(r|s_t, a_t, s_{t+1}).$$

We use the Q-function in order to define the optimal policy π^*, as the policy taking actions that maximize the action-value function:

$$\pi^*(s) := \arg\max_{a \in \mathcal{A}} Q^{\pi^*}(s, a).$$

In value-based learning, we use samples of interactions between the agent and the environment along with the immediate step-reward r_t, in order to learn the Q-function which in deep RL is modeled by a deep neural network with parameters θ. A single transition sample is made up of the current state s_t, action a_t, next state s_{t+1}, and the reward r_t given to the agent by the environment. In order to learn the value function over many iterations, we use the Bellman equation where $r_t + \gamma \cdot V^\pi(s_{t+1})$ is considered a better estimation than $V^\pi(s_t)$ for the value of current state s_t. Using the relationship between V^π and Q^π

$(V^\pi(s) = Q^\pi(s, \pi(s)))$ we can write:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}}\left[r_t + \gamma \cdot \max_{a'} Q^\pi\left(s_{t+1}, a'\right)\right] \tag{2.5}$$

so that only the Q-function appears on each side of the equation.

The Bellman equation is central to value-based RL, and provides some theoretical grounding in terms of convergence properties for learning the value function through an iterative scheme. This can be shown by proving that an operator, applying the Bellman equation to a value estimation (i.e. Bellman operator), is a contraction mapping in the space of value estimations and therefore must have a fixed point which can be reached by recursively applying the operator to an initial guess. The proof for the Bellman policy operator is given below:

Let U, V be two value functions, and define the Bellman policy operator $\mathcal{B}^\pi$ such that $\mathcal{B}^\pi V := \sum_{s'} \mathcal{T}(s'|s, \pi(s))\left[R^{\pi(s)}_{s,s'} + \gamma V(s')\right]$ (some notations of this operator use $\mathcal{T}^\pi$, however this may be ambiguous with our use of $\mathcal{T}$ to denote the MDP transition model, so we use $\mathcal{B}^\pi$ instead). $R^{\pi(s)}_{s,s'}$ indicates the expected reward received for the state transition i.e. $\mathbb{E}[R(r|s, \pi(s), s')]$. We show that $\mathcal{B}^\pi$ is a contraction mapping:

$$\begin{aligned}
\|\mathcal{B}^\pi V - \mathcal{B}^\pi U\|_\infty &\overset{(a)}{=} \max_s \gamma \sum_{s'} \mathcal{T}(s'|s, \pi(s))|V(s') - U(s')| \\
&\leq \left(\sum_{s'} \mathcal{T}(s'|s, \pi(s))\right) \gamma \max_{s'} |V(s') - U(s')| \\
&\text{(sum of probabilities of transitions to } s' \text{ is 1)} \\
&\leq \gamma \max_{s'} |V(s') - U(s')| \\
&\leq \gamma \|V - U\|_\infty
\end{aligned}$$

Where (a) uses the fact that that $\sum_{s'} \mathcal{T}(s'|s, \pi(s))\left[R^{\pi(s)}_{s,s'} - R^{\pi(s)}_{s,s'}\right] = 0$.

The role of $\mathcal{B}^\pi$ is to evaluate the value of a policy in the environment, such that we are able to learn one of the value functions (either V^π or Q^π) for the current policy π. This is useful in order to be able to learn the shape of the return function over the MDP state-space, and so to compare the expected performance of different policies. We use the value function in order to define the optimal policy π^* in an environment as the one where the associated value function V^{π^*} is maximal over the entire state space. Hence we are able to say that π^* is the optimal policy iff:

$$\forall s \in \mathcal{S}, \forall \pi \in \Pi, \quad V^{\pi^*}(s) \geq V^\pi(s) \tag{2.6}$$

where $\pi \in \Pi$ indicates the space of all possible policies. In the case where the policy is parametrized by a a set of parameters π_θ, then we refer to the parameters of the optimal policy as the optimal parameters θ^*. In this case, the domain of the discounted return function G_t, being approximated by the value function $V(s_t)$, is the entire parameter space. For complex policies such as neural networks which may contain a very large number of parameters, the return landscape is very high-dimensional and complex. For this reason the correct parametrization of value estimators is important to be able to be able to both correctly estimate G_t, and avoid being unnecessarily difficult to optimize, amongst other problems which arise when over-parameterizing an estimation function.

We can deduce the optimal policy π^* if we are able to learn the associated optimal value function V^{π^*}, by simply navigating to the states maximizing the associated return for the optimal policy. This way, we can transform the problem of learning the optimal policy to that of learning the optimal value function, which can be done via the Bellman optimality operator $\mathcal{B}$:

$$\mathcal{B}V := \max_a \sum_{s'} \mathcal{T}(s'|s,a) \left[R^a_{s,s'} + \gamma V(s') \right] \tag{2.7}$$

The proof for contraction of the bellman optimality operator, $\mathcal{B}$, follows a similar reasoning to that of $\mathcal{B}^\pi$ and is given in appendix B.1. Although we are able to ensure a proof of convergence for learning the optimal value function V^{π^*}, there are two additional factors to take into account when applying $\mathcal{B}$ to an agent-environment setup: knowledge of the transition dynamics $\mathcal{T}$, as well as the sum over potential next states $\sum_{s'}$. Knowing the dynamics $\mathcal{T}$ of the environment allows for the algorithm to evaluate the possible returns from many possible state transitions simultaneously, since it is able to simulate these transitions without performing them. However in some environments the transition dynamics are unknown, or only known over a limited domain. This can be the case in a multiplayer game for example, where the opponent's moves are difficult to predict. On top of this, the Bellman operators assume that we are able to perform a sum over the entire next-state-space s'; If there is a high amount of possible next states in the environment, then it is not reasonable to assume that we are able to perform computations over such a large space. For this reason, in more complex environments we use a sampled approach where interactions between the agent and the MDP are sampled over many episodes, so that over a long enough period of time, the dynamics and state-space of the environment are sufficiently well-represented such that they can be estimated by a learning algorithm.

The convergence properties for the Bellman operators gives us a theoretical foundation for constructing *Bellman targets*, in order to define a learning problem where we must learn to fit an estimator to the target values, and consider these as improvements over our current estimation of the state-action value function Q^π. In order to combine this approach with function estimation over a

large domain, these target values can then be used to construct a loss function with respect to the current estimates, which in turn can be used to optimize a model (a neural network with parameters θ, for example) by fitting a more accurate current output to target values, as is done in supervised machine learning approaches:

$$x_t = Q^\pi(s_t, a_t) \tag{2.8}$$

$$y_t = r_t + \gamma \cdot \max_{a'} Q^\pi(s_{t+1}, a') \tag{2.9}$$

We can use for example the mean squared error loss function $\text{MSE}(x_t, y_t) = \frac{1}{N}\sum_i (x_i - y_i)^2$ for multiple samples in order to train model parameters. We further discuss using learning algorithms in section 2.2 where we present how neural networks are used with agent-environment interactions samples to learn value estimators.

Policy-Based Agent

The second family of algorithms is known as policy-based learning. This approach seeks to directly optimize a policy function whose outputs are the probabilities that each action is the optimal, instead of directly estimating the Q^π function. Due to this we have a slightly different notation for the policy, writing $a \sim \pi(a|s)$ instead of $a = \pi(s)$ which is deterministic in the case of a value network output. In this approach we look to perform a gradient ascent directly on the policy model parameters θ (instead of a loss). Considering our base objective given in (2.2), we can define an objective function in terms of the policy parameters θ which we then maximize (the details for this derivation is straightforward and given in Appendix B.2):

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta}\left[G_t\right] \\ \Rightarrow \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[G_t \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \end{aligned} \tag{2.10}$$

The $\nabla_\theta \log \pi_\theta(a_t|s_t)$ values in the sum are known as the log-gradients for a policy's trajectory, and $\mathbb{E}_{\pi_\theta}$ stands for the expectation under trajectories generated by policy π_θ.

One of the main differences between the two approaches is that the value-based approach results in a deterministic policy (due to the argmax operator), and the policy-based approach results in a probabilistic policy. There are further differences between the two approaches which we discuss at the end of this chapter, however recent popular algorithms seek to combine the two approaches in an actor-critic dual-network architecture [Mnih et al., 2016] (one actor network - the policy, and one critic network - the value estimation). The main

idea behind learning two networks concurrently is to replace the value of G_t in the policy gradients by a value estimation Q^π so that we're not dependant on full episodes to train the policy network. Actor-critic architectures have been combined with entropy regularization [Haarnoja et al., 2018] (soft actor-critic), along with estimates of the discounted return gradients [Schulman et al., 2017] (proximal policy optimization) to form some of the best-performing architectures in RL.

Stochasticity and Partial Observability of the Environment

A stochastic environment will have non-deterministic, aleatoric dynamics represented by a distribution $\mathcal{T}(s'|s_t, a_t)$ of possible next states conditioned on the agent's action. To model this partial obervability we can consider that the agent only has access to the MDP's state through an observation model:

$$o_t \sim O(o|s_t).$$

Optimizing in a partially observable MDP (POMDP) in notoriously difficult, due to the fact that uncertainties in current state, or transition to the next state may have a very high impact on the value of the returns seen by the agent. Due to the fact that classical RL algorithms deal with expectations on trajectories and performance, this means we would have to obtain samples from every significantly possible combinations of events in order for the agent to have a good grasp of the expectation of performance in that environment. Due to the overwhelming combinatorial complexity of such an approach, we usually make simplifications to the POMDP framework, or apply an ad-hoc algorithm which is designed to tackle that particular environment.

2.1.3 Algorithmic Patchwork

In the previous section we discussed the basic RL architectures of value and policy-based agents. Though these are the basic building blocks for reinforcement learning agents, there have been many advancements to increase the viability of RL algorithms in control tasks, from affecting the rate of random exploration done by the agent, to reducing the variance in gradient updates to give a couple of examples. Below we provide an overview of the most popular modifications to RL algorithms which are mostly all present in current RL applications.

Exploration vs. Exploitation Trade-Off

During the training phase of an RL algorithm the agent collects interaction samples according to a policy which it follows by actions that are estimated to be good (learned policy and followed policy may be different - discussed under on-policy vs. off-policy learning). Often, due to the complexity of the control task, and when using a parametrized policy π_θ, the space of returns is susceptible to having local optima which correspond to global sub-optimal solutions. There is a risk during training that if a policy is locally optimal, the agent attempting to learn the optimal policy will be stuck in this trajectory, although it is not the optimal one. Given how RL algorithms use the return as a proxy for how good the agent's behaviour is, if no higher returns are obtained by the agent, then the algorithm will have converged and stop learning. However for most environments, virtual or simulated, there are such a high number of possible trajectories that it is impossible to correctly learn the expected returns in all areas of state-space. The length of trajectories also contributes to the complexity of learning correct behaviours in the environment, and the longer the agent trajectories are, the harder it is for an agent to quantify all possible trajectories to model the obtainable return. The error in return estimation for some areas of trajectory-space may cause the agent to miss out on the global optimal policy π^* which we wish to learn.

For this reason, we implement an *exploration* mechanism into the agent's training phase, whereby we can make the agent take exploratory actions which do not follow the agent's training policy and hence can help to not get so easily stuck in a local optimum. The idea being that the agent should initially explore the state-space so as to obtain a reasonably good estimation of which trajectories seem the most promising, and only once it has formed a good initial estimate of how good each strategy is, we may refine the returns estimates for the most promising areas of state-space.

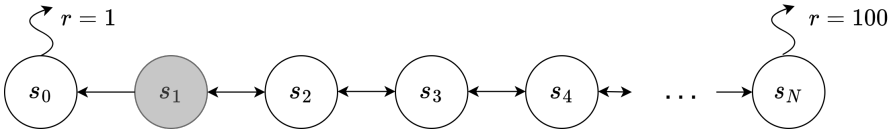


Figure 2.5: n-chain MDP, where optimal policy is difficult to find.

Figure 2.5 shows an example MDP, where the agent starts in shaded state s_1, and can either go left or right, before reaching either s_0, where it will receive a reward of 1, or s_N where it will receive a reward of 100. In this case the policy going only left will obtain a discounted return of $G_0 = 1$, and the policy going only right will obtain the discounted return of $G_0 = \gamma^N \cdot (100)$, hence which is the optimal policy will depend on the value of γ and N. To motivate our exploration example we can simply take $\gamma = 1$ and N some large positive integer. Once the

policy is initialized at the beginning of training, the actions taken by the policy will initially be random. With this n-chain MDP the probability for an agent to reach s_N is $P(s_N) = \frac{1}{N}$ (general solution of a random walk with two absorbing states at s_0, s_N is $P(s_N) = \frac{k}{N}$ when agent starts at $s_k, 0 < k < N$), hence for a value of $N = 100 \Rightarrow P(s_{100}) = 0.01$ the agent already has very little chance of observing the optimal return of $G_0 = 100$. If we further use the policy's performance in order to update policy parameters, then this optimization will likely push the policy towards preferring to go left, since it thinks that there is no reward to be had while going right, and fail to converge to the optimal policy. Because of this it is important for us to encourage exploratory action from the agent so that it is able to discover these 'hard-to-find' strategies.

However, we must strike a balance between looking for new strategies, and optimizing for exploiting the best available one. Excessive exploration prevents the agent from correctly estimating the returns and comes at the cost of performance when training is constrained by both time and computational resources. This is known as the *exploration vs. exploitation trade-off* in RL, and is the reason for some specific modifications to the agent's policy during training so that we don't fall into the local optima trap. The most common solution is to encourage exploratory behaviour in the beginning of training to obtain a sufficiently good idea of what the returns look like for various strategies in the environment, and then reduce the amount of exploration once we wish to exploit the best strategies and maximize returns obtained by the agent. By far the most common practice in RL is known as an ε-greedy policy, and uses an exponentially decaying rate of random exploration over the course of the training phase, going from $\varepsilon = 1$ down to $\varepsilon_{min} > 0$, where a minimal value is maintained to keep some chance of exploration still during the entire training phase. A typical value used is $\varepsilon_{min} = 0.01$ giving a small chance of random exploration so as not to affect the return estimation too much. For each time-step, we sample a random number $r \in [0, 1]$, and use the following policy:

$$\varepsilon\text{-greedy}(\pi) = \begin{cases} \pi(o) & \text{if } r < \varepsilon \\ \mathcal{U}(\mathcal{A}) & \text{otherwise} \end{cases}$$

where $\mathcal{U}(\mathcal{A})$ is the uniform probability distribution over the agent's actions-space, and $\pi(o)$ is the action recommended by the agent's policy. To encourage increased exploration at the beginning of the training phase, we use exponential decay based on the duration of the training phase:

$$\varepsilon = \max(C^{\frac{n_{steps}}{T}}, \varepsilon_{min})$$

where $C \in \mathbb{R}$ controls the rate of decay, n_{steps} are the number of training steps passed, $T \in \mathbb{R}$ is a time constant.

There also exists other exploration strategies, which use information on the state-space in order to improve upon the random-based ε-greedy policy. For example some directed exploration approaches use an estimation of the model's

confidence in the value of a state to encourage the agent to explore areas of state-space where the Q-function is not well learned.

Algorithm 1 gives a pseudo-code implementation for a typical RL agent training loop.

Algorithm 1 General RL learning algorithm

```
Init π
while training do
    s_0 ~ p^π(s_0)                          ▷ initialize new episode
    o_0 ~ O(s_0)
    while episode not terminated do          ▷ play episode
        a_t = ε-greedy(π)                    ▷ random action with probability ε
        s_{t+1} ~ T(s_t, a_t)
        r_t = R(s_t, a_t, s_{t+1})
        o_{t+1} ~ O(s_{t+1})
    end while                                ▷ Collect interaction samples
    TrainModel()                             ▷ Train policy from samples
end while
```

On-policy vs. Off-Policy Learning

The reinforcement learning algorithm we use in this work is based off of the popular Q-learning algorithm. Broadly, an RL algorithm may either be on-policy or off-policy. The difference being whether or not the acting and updating policies are the same during the training phase. On-policy algorithms use the same policy to both act and estimate the value of the future states for learning, whereas off-policy algorithms use the value from a policy in updates that is different from the one collecting interaction samples in the environment. This affects the TrainModel() function mentioned in algorithm 1.

The policy being learned (i.e. the value function, in the case of Q-function estimators) is the policy receiving updates from Bellman targets based on interaction samples. From the Bellman equations, learning from samples means that we must estimate the value of a future state using the policy which we wish to learn, in order to have a converging fixed point scheme. We can see the difference by comparing an on-policy algorithm, SARSA (named after the $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ sample tuple required for training, though we deal with observations o_t instead of states to be as general as possible), with the off-policy Q-learning. Both updates can be given as:

$$Q^\pi(o_t, a_t) \leftarrow r_t + \gamma Q^\pi(o_{t+1}, a_{t+1}) \text{ (SARSA)}$$

$$Q^\pi(o_t, a_t) \leftarrow r_t + \gamma \max_{a'} Q^\pi(o_{t+1}, a') \text{ (Q-learning)}$$

SARSA uses the current policy Q^π to evaluate the value of the next observation o_{t+1}, by means of the action a_{t+1} that was actually selected by the policy from that state. On the other hand, uses the *greedy action* a' (only maximizing returns, no exploration) in order to determine the value of the next observation o_{t+1}. This means that it estimates the value of states according a greedy policy, even though the samples could have been collected by an exploratory policy taking non-greedy actions. In the case where a greedy policy is collecting samples from the environment, then the distinction disappears. Off-policy learning allows us to use exploratory actions to learn better Q-estimates over the policy's domain, without biasing the value of the optimal policy with the potentially bad exploratory actions. This however in turn has the tendency of over-estimating the value of states due to the max operator being overly optimistic about the return value of states obtainable by the agent, though there are some ways of addressing this issue. Mnih et al. [2013] introduce the use of deep neural networks in conjunction with Q-learning, using what is refered to as a deep Q network, or DQN. DQNs are one of the most popular algorithms from which stronger algorithms are built, and can be found in powerful approaches such as Deepmind's AlphaGo Silver et al. [2018], able to play the complex game of Go at a super-human level.

The idea of de-coupling the acting and estimation (or learned) policies is powerful not only because it allows for the use of exploration for a single agent to better explore its domain, but also opens the door for samples collected from other agents learning in parallel to be used on a same agent. This way, experiences collected from agents attempting to learn one task can be used to train an agent learning the returns for its own task, as it will judge the value of the collected samples in terms of its own return function rather than the action from the acting policy. Furthermore, we are also able to leverage past samples collected by an agent in the environment, and use them to train the current policy even though it will be different to the past one. This is useful as it allows multiple agent to train on collected samples, or an agent to use the same training samples multiple times over the course of a training phase. Since sample efficiency is so core to data-based learning applications, and especially so to reinforcement learning algorithms, having an algorithm which is able to re-use multiple samples is very helpful. The next classical add-on to deep Q-learning is a replay buffer from which past experiences are sampled multiple times to increase the algorithm's sample efficiency.

However, there are some disadvantages to the off-policy approach, such as when there is too much of a difference between the acting and estimation policies, then the agent might have a hard time providing an accurate estimation of the Q-function. This is seen when there is too much exploration for example, where the agent is not able to converge to the optimal policy unless the exploration term is annealed down to a low value for a prolonged period of time (this is comparable to if a human would learn to do a task through only using exploratory actions: not impossible, however inefficient and counter-intuitive).

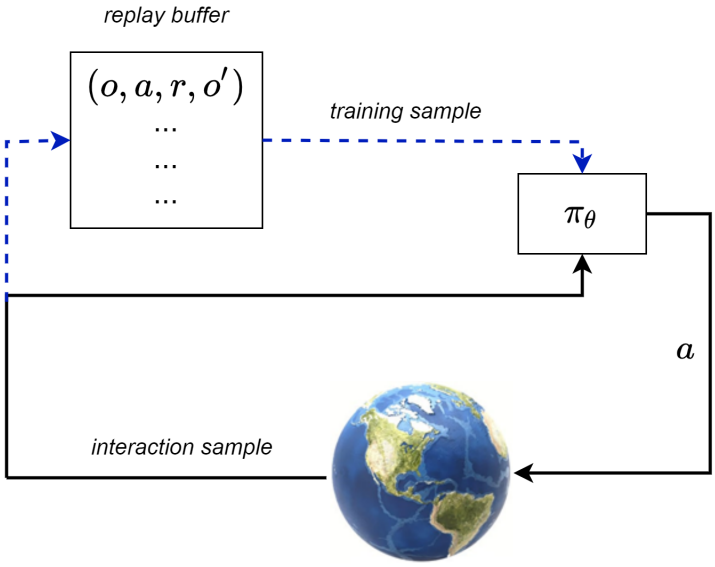


Figure 2.6: Training a policy with a replay buffer.

In practice, due to the better sample efficiency and the ability to work around some resulting inaccuracies in the return estimation, off-policy learning is much more popular than on-policy approaches.

Hindsight Experience Replay

One major limitation of reinforcement learning is the amount of training samples that are required for training the agent's policy. Each individual agent-environment interaction has little effect on the policy updates which we perform, and we require a large number of updates before the agent effectively learns the environment dynamics. Introducing off-policy learning, and being to able to train on data that was generated by a policy different to the one being learned means that we are also able to 'recycle' past interaction samples that were generated by a past version of the agents policy, and use them multiple times to increase the sample efficiency of the algorithm and avoid having to re-generate similar samples using a simulation environment. This practice is known as *hindsight experience replay* (HER) and uses a replay buffer containing past samples which are then in turn used to train the policy's parameters.

Figure 2.6 shows how the training loop is slightly modified to include a sampling operation (dashed blue line from replay buffer to policy) from the replay buffer. Now, the states used to train the policy are no longer the same as

the ones being collected at the same time in the environment. Sampling from an independent buffer also allows us to increase the iid (independent and identically distributed) property of the training data. For data-based approaches, of which RL is a part, bias in the training data distribution will in turn result to bias in the model learned from this data. Training from samples received immediately may bias the model too much towards these immediate states and lose model accuracy on other parts of its domain at test time. Additionally, the sampling operation allows us to reduce the correlation between training samples as it allows the model to train on interactions from different trajectories. Typically an MDP generates highly correlated data (the next state depends on the previous state), though between two samples of different episode, there is little correlation (just from the fact that they would have been generated by the actions of a similar policy, depending on how far back they were collected). Training from samples in this way also makes it easier to perform batch updates when the policy is modelled by a deep neural network, since we're able to sample how many interactions we want from the buffer at training time.

Following the size of the replay buffer, it may also be used to represent the latest states and trajectories that are collected by the agent during training. Later in our work, we make use of this past data available to the agent during training in order to quantify which general areas of state-space it is optimizing its policy around.

Double Q Learning with Target Networks

One issue with Q-learning is that of value overestimation when using the max operator in (2.5), using a greedy policy to evaluate the value of the next state-observation. Especially in the case of stochastic environments, where there are multiple possible rewards for a single transition, using $\max_{a'} Q(o_{t+1}, a')$ will only consider the highest expected return estimation from the next observation, and bias the agent into being over-confident about the returns, without considering the fact that the return value may in fact be much less. For this reason, Hasselt [2010] introduce *double Q-learning*, whereby the action selection and value evaluation are separated in order to reduce this value over-estimation which may lead the agent to not converge to the optimal policy. In the original double Q-learning algorithm, we use two separate networks Q^A, Q^B to separate the action of estimating the value of a state by the Bellman target.

$$Q^A(o_t, a_t) \leftarrow r_t + \gamma \; Q^A(o_{t+1}, \arg\max_{a'} Q^B(o_{t+1}, a')) \tag{2.11}$$

Instead of taking the greedy action maximizing the same evaluation estimation, we take the best estimated action from another return function, Q^B, such that we avoid a kind of 'confirmation bias' of best estimated and selected action being from the same Q-estimator. Though the original approach uses two different networks, which would half the training efficiency since both networks would

have to be trained using different samples, Van Hasselt et al. [2016] combine their double Q-learning approach with target networks introduced by Mnih et al. [2015] for stabilizing target updates in their DQN implementation.

Target networks consist in remembering an offline version of model parameters θ', and routinely updating them to match the online ones θ that change much more dynamically. In the DQN algorithm, this allows us to have access to an estimation of the return which is more constant, and allows for faster and more effective learning. This replaces the DQN update with:

$$Q_\theta(o_t, a_t) \leftarrow r_t + \gamma \max_{a'} Q_{\theta'}(o_{t+1}, a') \tag{2.12}$$

Replacing the double value functions Q^A, Q^B with online and offline versions of the Q-function parameters, we obtain the following update, known as *double DQN* to the use of double Q-learning and target networks used in the DQN updates:

$$Q_\theta(o_t, a_t) \leftarrow r_t + \gamma\ Q_\theta(o_{t+1}, \arg\max_{a'} Q_{\theta'}(o_{t+1}, a')) \tag{2.13}$$

This update is standard to use when learning Q-functions and provides both better training performance and avoids some of the value over-estimation that occurs when the selection and estimation policies are the same. This hybrid implementation has the advantage of not requiring two entirely different networks which would heavily affect learning and sample efficiency, as the target parameters θ' can be easily memorized and simply updated according to some time-constant so that every T_{target} steps, we set $\theta' = \theta$. The frequency of updating target parameters is set empirically so as to avoid value overestimation from them being too close the online parameters, and also avoiding bias in the target

2.1.4 Using RL for Control

Overall reinforcement learning is a tool that has the capability of learning complex control laws in systems where classical approaches are more difficult to implement, and benefits from the strengths (and weaknesses) of data-driven approaches to modeling. The central idea behind RL – optimizing a reward signal – is based on the necessity for a reward signal to be provided to the agent. In some tasks the values of rewards are straightforward: in a game of chess for example the only objective is to win and so giving a positive reward for winning a game and a negative reward for losing seems to encompass sufficiently well the desired behaviour of a chess-playing agent. However in some environments the reward value to be provided to the agent is not so obvious. If we wish to teach a robot how to walk for example, is it better to give a reward based on distance travelled (we risk the robot learning to crawl or just wiggle around to cover

the most distance)? Or we could combine distance along with some penalties on incorrect posture (we risk stopping the agent from discovering more efficient strides)? There are many examples in the literature of RL agents with poorly-defined reward functions finding some 'hack' in the environment which allows them to maximize the rewards they obtain but without attaining a behaviour which we would heuristically describe as 'good' (for example running around in infinite loops to collect redundant rewards instead of completing the task).

Still tied to the reward function, we have the notion of *sparse* vs. *dense* rewards. This pertains to how often the agent receives a reward signal during its training episodes. An example of a sparse signal would be only receiving a reward at the end of a chess game depending on win or loss, and a dense signal could be the inverse of the distance to a target location to which a robot must navigate, which we can give to the agent at every time-step. Since the performance of an RL agent is based on it modeling the expectation in returns, we can see that a dense reward will tend to provide more training samples and thus help the agent to learn faster. If a reward is too sparse the agent may hardly ever encounter it and thus have a hard time re-tracing the correct actions that caused the reward. On the flip-side, sparse rewards are preferred for their simplicity and lack of potential bias towards behaviours that may not have been foreseen. Taking the example of a chess game, a natural reward is for either winning or losing the game, however we could additionally reward the agent for taking enemy pieces for example, this could provide intermediary objectives and help the agent to learn good behaviours faster. However in this case we risk biasing the agent towards capturing pieces rather than winning the game, and it may even end up losing some games if it is focused on obtaining rewards for capturing pieces. This issue is not unique to RL, and arises in any multi-objective optimization problem, where the presence of more than one single objective leads to needing to balance the weight of each in order to obtain the desired behaviour. RL can be seen as providing soft constraints on the conditions modeled by the reward function, and any additional rewards that are given add another implicit optimization objective to the agent. These then need to be balanced out according to what we feel is an appropriate ratio, and often several values must be tried and tested before ending up on the correct combination for the desired behaviour the emerge from the agent.

We can use an autonomous vehicle task as an example for this dilemma, having to balance both performance and safety criteria. Let's say we give penalties for the time it takes the agent to navigate through an intersection (encouraging faster behaviour), and also a strong penalty for any collisions (encouraging safe behaviour). If the penalty per second is -0.01 and the penalty for a collision is -1, then we are implicitly telling the agent that it should cause a collision if it is able to save 100 seconds of travel time. Or, if we factor the expectation over all possible events, that it is worth it to gain 10 seconds on a trajectory if the probability for collision is only 10%. Of course then we can modify the scaling of performance to be 10,000 times less than that of safety (i.e. the agent

should gain 1s of trajectory time only if it is sure that there is less than an 0.1% chance of collisions), which seems reasonable from a qualitative point of view, however from an algorithmic perspective the time objective may become so negligible that it is ignored entirely, leading to the agent only avoiding collisions and not wanting to complete the task (i.e. simply never moving could be the safest behaviour). This 'weighting' of dense rewards when there are multiple events included in the reward function is the reason for which it is often difficult to design a good reward function for an task, and usually it will go through at least some modification before the desired behaviour is learnt by the agent. One approach to multi-criteria optimization problems is to use Pareto optimization, where the improvement over one criteria is only acceptable as long as it does not cause a less-optimal shift in the other criteria. Although this is a desirable property it is often not realistic to assume that progress in one criteria can be made with no trade-offs on any other objective.

Implementing a controller for AVs in a driving scenario is met with many challenges: both from the point of view of perception and control [Yurtsever et al., 2020]. As in most applications of real-world RL, the uncertainty linked to the perception of the agent's environment must be considered for an effective controller to be developed. Even with the best possible road maps and sensors, it is impossible to eliminate all sources of uncertainty from a driving scenario, be they epistemic from imperfections in the vehicle's sensors, or aleatoric from the unpredictable interactions with other drivers [Depeweg et al., 2018].

Autonomous navigation requires a strong notion of safety, and notably robustness with respect to unexpected changes in the agent's environment. For example, sensor perception quality can be heavily susceptible to adverse weather conditions [Zang et al., 2019]. Because of this the optimal behaviour is likely to change dynamically according to the vehicle's inputs, and a satisfactory control algorithm must be able to adapt on the fly. Safety criteria in autonomous driving applications are traditionally based on perceiving when a situation is no longer able to be handled by the acting controller, and then handing over the controls to either the driver, or a special-case controller. For example, Bouton et al. [2019] implement a deep neural network to detect the probability of a catastrophic outcome when following recommended actions, whereas other approaches such as Clements et al. [2019] or Hoel et al. [2020b] look to estimate the confidence an agent has in its predicted outcome for a sequence of actions in the environment from uncertainties on neural network parameters.

RL techniques have been shown to be able to tackle the task of control in progressively more complex environments [Badia et al., 2020a]. RL algorithms learn by optimizing their expectation of performance in an environment. In most cases, such as board games [Mnih et al., 2015] or video games [Berner et al., 2019], the environment in which we seek to obtain the optimal behaviour can be modeled as a Markov Decision Process (MDP) with no loss of generality in the solution found by the RL agent. Through advancements in target updates

[Hessel et al., 2017], as well as agent architectures [Schaul et al., 2016], RL agents have become increasingly efficient at finding the optimal solution to MDPs, even when requiring high degrees of exploration, where the optimal sequence of actions is hard to find [Sutton and Barto, 2018].

Stochastic control environments such as driving scenarios are more difficult to optimize, given the probabilistic nature of both the observation and transition dynamics. Stochastic environments may be modeled as POMDPs [Kochenderfer et al., 2015]. Solving POMDPs is possible with methods combining learning and planning, such as Hoel et al. [2020a]. However, a change in the values of the stochastic model parameters, for example a change in a vehicle's sensor accuracy, scene obstruction, or simply unplanned behaviour from another vehicle, may induce a sharp drop in the agent's performance due to its inability to generalize well to new environment parameters. Having access to a model of the environment dynamics allows us to use planning algorithms, such as MCTS [Browne et al., 2012], alongside learning to both increase sample efficiency, and have access to a better representation of the environment's state-space structure (model-based RL). In cases where planning is possible, it is much easier to find alternative strategies for an agent to solve its environment and hence be able to better adapt to eventual changes in the state-space [McAllister and Rasmussen, 2017].

2.2 Deep Neural Networks

So far we have presented the framework of reinforcement learning, using samples of interaction between an agent and an environment in order to learn optimal control. These samples are used to reinforce good behaviour through updates defined by the Bellman equation. This approach requires a policy that is able to learn, and perform the Bellman updates in order to refine it's value estimation over its domain. Early implementations of RL algorithms on environments whose domain is not high-dimensional (such as a 2-D maze, for example), use tabular policies with hard-coded updates using some learning rate attached to the Bellman targets to improve the return estimation of the agent's policy. For example we can store Q-values in a table for every possible state-action pair, and perform the Q-learning update:

$$Q^{\pi}_{o_t,a_t} \leftarrow (1-\alpha)Q^{\pi}_{o_t,a_t} + \alpha(r_t + \gamma Q^{\pi}_{o_{t+1},a_{t+1}})$$

where $0 < \alpha < 1$ is the learning rate, and $Q^{\pi}_{o,a}$ is the tabular value for observation-action pair. With enough training data a tabular policy is able to learn the optimal policy over a small domain. Once the domain starts getting more complex, or even continuous, then there is no way for the agent to correctly quantify the returns from each observation-action pair.

However with the Bellman targets, we are also able to construct a loss in order to update differentiable parametric policies so that they better fit the incoming data. One such candidate for modeling a policy is a deep neural network, which is essentially a composition of differentiable functions, allowing for a greater power of generalization while remaining able to optimize outputs in order to learn to fit data. The composition operation of deep networks (the output of one layer is the input of the next layer) allows for these models to capture complex dynamics and patterns in the data which shallow networks, even with a high number of parameters, are unable to represent. For this reason, deep neural networks are very popular for reinforcement learning applications as Q-value estimators or as policy networks. Neural nets also have the property of having highly flexible architectures, meaning that we are able to directly affect the composition operation that happens between the layers, in order to capture some prior knowledge about the task which we wish to implicitly include in the model conception. This can include embedding layers or recurrent architectures for example.

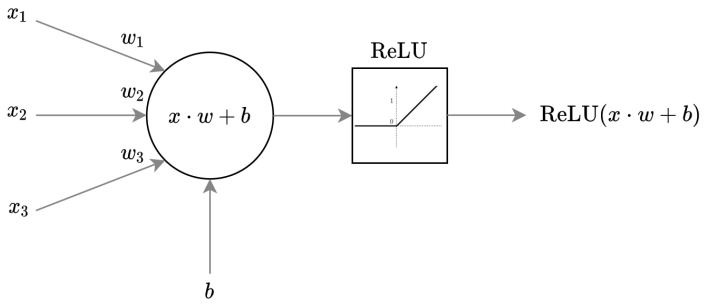


Figure 2.7: Artificial neuron

Neural networks are composed of multiple layers of artificial neurons, whose role it is to provide an activation value, as a function of multiple inputs (which are typically are the outputs of other neurons). Figure 2.7 illustrates the operations done with a vector of inputs x. The trainable parameters are the weights w which multiply the input signal, and the bias b acting as an adjustable offset. In order to help the model capture non-linear dynamics, a non-linear activation function is added to the neuron's output. Without this non-linearity a network of neurons could only capture linear boundaries for classifying data, and would not be suitable to apply to more complex problems such as those tackled by RL algorithms. The most popular activation function used in neural networks today is the Rectified Linear Unit (ReLU), which performs the following operation: $\text{ReLU}(x) = \max(0, x)$. This is an improvement over the previously-popular sigmoid function $S(x) = \frac{1}{1+e^{-x}}$, due to the reduction in the vanishing gradient problem (gradient used for learning parameter values can become small for large positive or negative activation values), and the ease of computing simply a max

operation instead of an exponential.

Combining these neurons into multiple layers, as shown in figure 2.8, with one 'hidden' or 'deep' layer, allows for the overall function to be designed resulting from the composition of the artificial neuron operation $\mathrm{ReLU}(x \cdot w + b)$. Gathering the trainable parameters $\{W_{1,2}, B_{1,2}\}$ into a single vector θ, we can write the operation performed by the entire network as $\tilde{y} = f_\theta(x)$, for any input x. The $\tilde{y}$ notation is used to indicate that f is used as an estimation for another unknown function. This network architecture is known as a fully-connected network, as all of the possible connections between the subsequent layers exist and are parameterized. There has been much progress on alternative network architectures in order to take advantage of the solution heuristics to make the function better-adapted to tackle the input and output dimension expected from the problem.

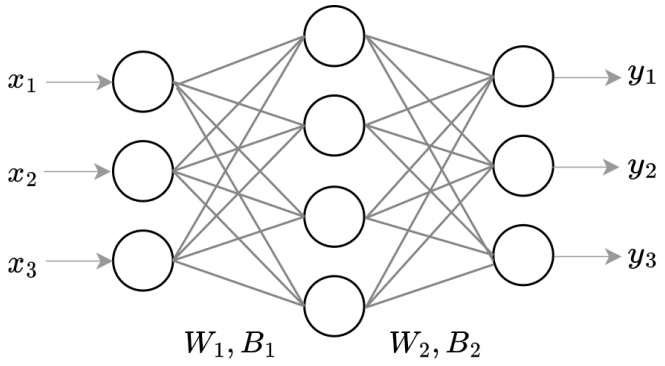


Figure 2.8: Fully-connected network of artificial neurons

2.2.1 Training a Network

Adjusting the neural network's parameters θ by means of any optimization algorithm is referred to as training, as we are improving the function's performance for estimating outputs y with its estimation $\tilde{y}$. In our work, we use a fully-connected network f_θ, which is trained by gradient descent on a loss function between targets $\tilde{y}$ and estimations y. A typical cost function we use is the mean square error (MSE) between the estimated and labelled output:

$$MSE(y, \tilde{y}) = \sqrt{\sum_{i=1}^{n} \frac{(\tilde{y} - y)^2}{n}}.$$

Another commonly used loss function is the cross-entropy: $H(y, \tilde{y})$.

In order to train the network parameters θ by gradient descent, we must compute the partial derivatives of the loss function with respect to each of the network's weights and biases $\frac{\partial L}{\partial \theta}$. Since a neural network is a composition between the operations done in each of the layers, we may use the chain rule in order to compute the gradients throughout the whole model. This is known as the *backpropagation* algorithm [LeCun et al., 1989], whereby the values of gradients for the final layers are computed initially, and then propagated backwards through the previous layers in order to compute those gradients via the chain rule. Once the gradients $\frac{\partial L}{\partial \theta}$ for each element of theta is computed, then we may use a first-order optimization algorithm such as gradient descent in order to optimize the loss function. It is important to note that a neural network loss function is not convex, and thus the optimization of the can be quite difficult, getting stuck in local optima if sufficient care is not taken during the training phase. Affecting the learning rate is one method for preventing this, for example setting the value to be initially large and then then annealing it down to a small value can be a way to explore the loss landscape sufficiently before converging to the best possible model parameter value on the testing data.

In the next chapter, we present the approach to modeling a probabilistic environment as an MDP, along with a distributional perspective on the reward signal in order for the agent to have more information about the possible outcomes it has learned from all possible outcomes during its training.

Chapter 3

Distributional Perspective on MDP Optimization

3.1 Introduction

A key limitation to applying RL algorithms to stochastic environments, is that the value function used as a proxy for 'goodness of outcome' estimation is one-dimensional. Although this allows for a nuanced optimization of the sequence of intermediate states and actions which provide the return signal, the return value functions as an expectation over all possible agent-environment interactions and thus lacks some descriptive power when faced with a set of multiple possible trajectories.

Figure 3.1 shows how the expectation can hide some more nuanced dynamics. In this case the blue curve is a mixture of Gaussians with two components centered at -3 and 3 respectively, and their mean is represented by the red line centered at 0. If each of the modes of the distribution represent a potential event related to the agent in the environment, encoded by the different values of rewards that are attributed to it, then clearly we should wish that the agent react differently to this reward distribution compared to say, a single Gaussian centered at 0, which would have the same expected value represented by the value estimation Q^{π}.

From this perspective, we can see that expected-value algorithms which only deal with the mean value in terms of the expectation of returns can fall short in some cases where the true distribution has a multi-modal shape. Additionally, using the mean value gives us no indication about the likelihood that a sampled trajectory will obtain that value of rewards, since we don't take into account

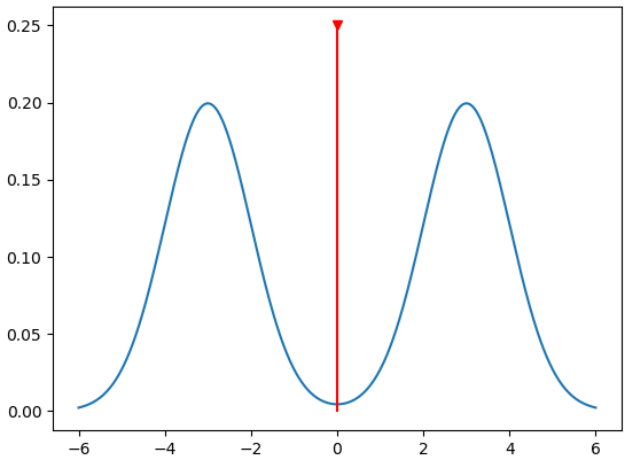


Figure 3.1: Mixture of Gaussians, and mean value.

higher probability moments (variance, tails, etc...). We can posit that there is much information about outcome probabilities linked to an agent's performance that is contained in the return distribution that is simply not available when relying on only the mean of this distribution, which may be useful for safety considerations. To this end we will investigate the usefulness of learning the probability distribution of returns, using these in order for an agent to acquire more information about its confidence in recommended actions, the certainty of outcome, and the level of safety that can be attributed independently from expectation of performance (even when safety criteria are included in the performance metric).

Sample-based methods for learning probability distributions already exist (K-means, moment-matching, or quantile regression for instance) and so we wish to leverage one of these approaches in order for an agent to learn the full distribution of returns it can expect to see in the environment. One particularity about our problem statement is the difference in sparsity of the events which we desire to model in the autonomous navigation task. We wish the agent to be able to react to certain events (notably high-consequence failures such as vehicle collisions) which may have a relatively low probability with respect to other outcomes more common for the agent. Because of this desired sensibility to low-probability events some parametrizations of the return distribution are not reasonable to use. Notably due to combination of sparse and dense events in the environment outcomes, we expect that the return distribution will have some clear demarked modes corresponding to possible agent behaviour along with possible outcomes. Because of this we should use parametrizations of the

learned distributions that take into account potentially sharp modes.

3.1.1 Related Work

Previous work has focused on using values other than the expectation of returns to improve the performance of RL agents. Much of this work has focused on applying these methods to the risk-sensitive exploration dilemma; Chow et al. [2015] for example uses a specific low quantile of the return distribution as an optimization objective.

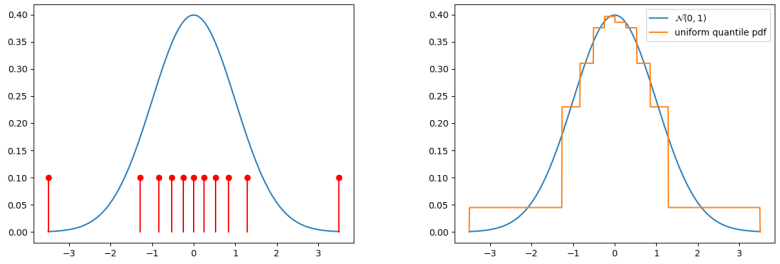
Characterizing the uncertainty of an RL environment has been a key area of research [Metelli et al., 2019, Brechtel et al., 2014, Clements et al., 2019]. for example learning the observation model [Bouton et al., 2019]. A more thorough use of the full distribution of returns, was first introduced by Morimura et al. [2010] and in Bellemare et al. [2017] where the use of explicitly learning the full distribution of returns was proposed. There has been some work done on the applications sensitice to some risk factor [Dabney et al., 2018], as well as improvement of the general framework used for learning return distributions from agent-environment interaction samples [Rowland et al., 2019].

There has also been a link made between the different kinds of uncertainty [Depeweg et al., 2018], [Clements et al., 2019] (epistemic & aleatoric) in an RL environment, however no explicit link with the shape of the return distribution observed during and after training.

Distributional RL (sometimes abbreviated as DRL – not to be confused with deep RL which is sometime also often abbreviated as DRL – and also different from distributed RL which deals with confederated learning between multiple different parallel learning processes) is a sub-domain of value-based reinforcement learning which aims to learn the full distribution of returns for a policy using quantile regression in order to learn the full distribution of returns for each action available to the agent. Though learning a collection of quantiles adds complexity to the learning process, this allows for a parametric function to be able to represent multiple modes in the return distribution around which we can design an autonomous agent a higher degree of reactivity according to the positions and sizes of the multiple modes. It has been shown through experimental work [Rowland et al., 2019, Hessel et al., 2017] that RL algorithms learning the entire distribution of returns perform better than their expected-value counterparts. There have also been a few theoretical results [Lyle et al., 2019] to motivate why this is the case for non-linear function approximation.

Quantile Regression

Distributional RL uses quantile regression in order to fit the outputs of a neural network to the quantiles of the distribution. Quantiles of a distribution are used to mark out ranges of a random variable, with equal probability. This means that quantiles for a dense probability distribution will be closer together, whereas if the distribution is more spread out, then the quantiles will also be spread out. We can denote the κ-quantile as q_κ, which for sampled random variable $x \sim X$, represents the point where $P(x < q_\kappa) = \kappa$. Figure 3.2a gives an example of the quantiles of the normal distribution for $\kappa = \left\{\frac{i}{10}, i \in [0, 10]\right\}$. q_0 and q_1 are positioned at either end of the probability distribution's support, in this case at -3.5 and 3.5 respectively, however since the normal distribution has its support over $\mathbb{R}$, the true values should be $-\infty$ and $+\infty$). Figure 3.2b shows a re-construction of the normal distribution, by assuming a uniform distribution over the range of each neighbouring quantile.



(a) 0.1-quantiles of the Normal distribution

(b) Uniform parametrization of Normal distribution from quantiles

Figure 3.2: Quantiles of the normal distribution

To learn values of q_κ of a target distribution η through sampling, we can use the following loss:

$$QR(q_\kappa, \eta, \kappa) = \mathop{\mathbb{E}}_{z \sim \eta}\left[(z - q_\kappa)(\kappa \mathbb{1}_{z > q_\kappa} + (\kappa - 1)\mathbb{1}_{z < q_\kappa})\right] \tag{3.1}$$

which is a form of lop-sided loss which, for enough samples, will balance out the positions of q_κ according to the mass of samples $z \sim \eta$ either to the left or to the right of the corresponding q_κ. Figure 3.3 illustrates what this loss looks like for different values of κ. We can see that for $\kappa = 0.8$, the loss will hardly penalize the estimated q_κ for being larger than the sampled value, since 80% of samples should be below that value, and vice-versa for $\kappa = 0.2$. $q_{0.5}$ is simply the mean of the distribution, so the loss will fit it to be in the center of the sampled values.

Learning the quantiles of a distribution gives us a great representative power

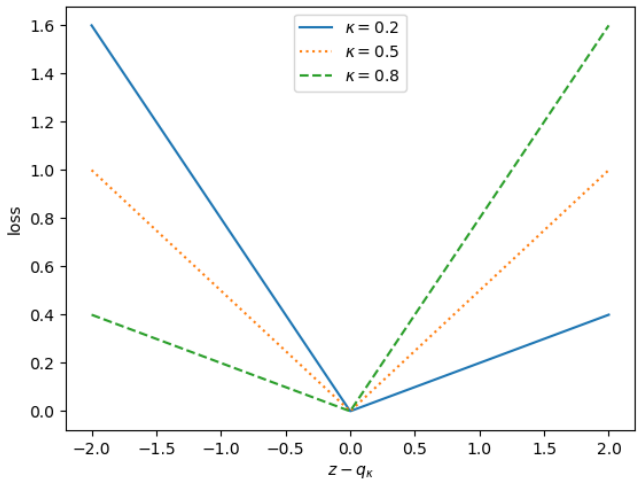


Figure 3.3: Quantile loss function for $\kappa = 0.2, 0.5, 0.8$.

to allow us to parametrize the target distribution, η. There are two main approaches to modeling η according to quantiles learned from samples: either with a sum of uniform distributions whose range is defined by neighbouring quantiles, as in figure 3.2b, or with a sum of diracs, one at each position of the quantile estimate q_κ. The implementation of either parametrization may depend on the need to have a continuous distribution (as is the case for the sum of uniform distributions), otherwise for a large number of estimated quantiles, both parametrizations become increasingly equivalent. In our implementations we tend to use the sum of diracs to parametrize the target distribution, since it is slightly faster to sample from, rather than a sum of uniform distributions.

Notation and Properties of Random Variables

We first introduce some further notation which is useful when working with probability functions:

Note: Operations on random variables, their pdfs and their cdfs are used with some abuse of notation, for example $\mathbb{E}[Z] = \mathbb{E}[\eta_Z]$, or $\mathcal{T}^\pi Z = \mathcal{T}^\pi \eta_Z$.

$Z^\pi(s, a)$: Random variable describing the distribution of returns for a state-action pair (s, a), and policy π. The expectation of $Z^\pi(s, a)$ is the value of the corresponding Q-function $Q^\pi(s, a)$: $\mathbb{E}[Z^\pi(s, a)] = Q^\pi(s, a)$. For clarity we often drop the superscript from the notation to write $Z(s, a)$,

as the relevant policy π is usually clear in the given context.

$\eta_Z : \mathbb{R} \to [0, 1]$: pdf of the random variable Z. $\eta_Z(z) = P(Z = z)$.

$\eta_{f(Z)}$: pdf of the random variable obtained by applying function f to the random variable Z. For example if $x \in \mathbb{R}$, $\eta_{Z+x}(z) = \eta_Z(z - x)$ or $\eta_{Z+X}(z) = \int \eta_Z(w)\eta_X(z - w)dw$ (sum of two random variables).

Equality in distribution $Z \overset{D}{=} X \Leftrightarrow \eta_Z(z) = \eta_X(z)$, $\forall z$. The two notations can be used interchangeably to make equations more clear, especially as the Bellman equations rely on sums of random variables.

We recall some useful properties when performing operations with random variables (since we treat the return as a sum of scaled random variables, we should take these into account when constructing the Bellman targets from 2.4 in the distributional setting).

Addition of two random variables:

Let X, Y, Z be random variables such that $Z :\overset{D}{=} X + Y$, i.e. $\eta_Z = \eta_{X+Y}$, with a domain on $\mathbb{R}$. Then,

$$P(Z = z) = \sum_x P(X = x)P(Y = z - x) = \int_{\mathbb{R}} \eta_X(x)\eta_Y(z - x)dx$$
$$\Rightarrow \eta_{X+Y}(z) = \int_{\mathbb{R}} \eta_X(x)\eta_Y(z - x)dx$$

This operation is also known as a convolution, where the distributions η_X, η_Y can be seen as the frequency domain representation of a series of sampling operations $x \sim \eta_X$ and $y \sim \eta_Y$, and so the frequency representation (Fourier transform) of the multiplication of the $x(t)$ and $y(t)$ series follows the convolutional theorem:

$$\mathcal{F}[x(t) \cdot y(t)] = \mathcal{F}[x(t)] * \mathcal{F}[y(t)]$$
$$\Leftrightarrow \eta_{X+Y} = \eta_X * \eta_Y,$$

where $x(t) \cdot y(t)$ is a point-wise multiplication.

Random variable multiplied by a scalar:

Let X, Z be random variables and $\gamma \in \mathbb{R}$ such that $X :\overset{D}{=} \gamma Z$, then:

$$\eta_X(x) = \frac{1}{\gamma}\eta_Z\left(\frac{x}{\gamma}\right)$$

Following the expression for the discounted return, we can write the distribution of its associated random variable Z as:

$$Z(s,a) :\overset{D}{=} \sum_t \gamma^t R_t \tag{3.2}$$

where R_t is the random variable associated to the reward obtained by the agent, and. This expression is the equivalent of (2.1), but with random variable notation.

We can build up Bellman targets just as in (2.4), only this time using distributions of random variables instead of expected values (we denote the distributional Bellman operator as $\mathcal{D}^\pi$):

$$(\mathcal{D}^\pi Z)(s,a) :\overset{\mathrm{D}}{=} R(s,a) + \gamma Z(X', A'), \tag{3.3}$$

where $Z(X', A')$ represents the return distribution estimation over all possible next state-action pairs when applying an action from π to the current state s. However, as with classical Q-learning, there are some extra steps needed between the theoretical Bellman update, and the one that we are able to implement in practice. Hence we define two additional elements that bridge this gap: The *stochastic Bellman approximation* $\widehat{\mathcal{D}}^\pi$, and a parametrization operator Π_θ. equivalently to the expectation-based paradigm, we may use the Wasserstein metric on probability distributions, to prove that the *distributional Bellman operator* is a contraction, providing theoretical grounding for our use of Bellman targets to learn the value function by means of a fixed-point iteration scheme. The proof for this is provided in appendix B.3.

Though we would like to treat the environment rewards as random variables, in practice when the agent takes actions in the environment it observes samples $r \sim R(s,a)$. Hence in practice we are unable to compute the full distributional form of Bellman targets as expressed in (3.3). Instead, we only have access to a stochastic approximation of the true Bellman update, through the sampled agent-environment interactions (s, a, r, s', a'). This leads to a stochastic approximation for the distributional operator, which we define below:

$$\begin{aligned}(\widehat{\mathcal{D}}^\pi Z)(s,a) &:\overset{D}{=} r + \gamma Z(s', a') \\ r \sim R(s,a), s' &\sim \mathcal{T}(s'|s,a), a' \sim A'\end{aligned} \tag{3.4}$$

Since the two forms of distribution targets are quite different, we must ensure that $\widehat{\mathcal{D}}$ and $\mathcal{D}$ have the same fixed point solution, such that it makes sense to apply $\widehat{\mathcal{D}}$ in order to find the solution to (3.3):

$$\lim_{N_{\text{samples}} \to \infty} \widehat{\mathcal{D}}^\pi Z(s,a) \overset{D}{=} \mathcal{D}^\pi Z(s,a).$$

The proof for this is quite straightforward, and so it is provided below:

$\widehat{\mathcal{D}}^\pi$ is an unbiased estimate of $\mathcal{D}^\pi$:

let Z, R be two random variables with respective pdfs $\eta_Z : \mathbb{R} \to [0, 1]$ and $\eta_R : \mathbb{R} \to [0, 1]$.

It is straightforward to show, with the definition for the expectation:

$$\begin{aligned}\mathop{\mathbb{E}}_{r\sim R}[\eta_{r+Z}(z)] &= \mathop{\mathbb{E}}_{r\sim R}[\eta_Z(z-r)] \\ &= \int_{\mathbb{R}} \eta_R(r)\eta_Z(z-r)dr \\ &\overset{(a)}{=} \eta_{R+Z}(z)\end{aligned}$$

Where (a) follows from the property of addition of two random variables.

$\eta_{Z(s',a')}$, the next-state return estimation, is a distribution. However it is also a sample from the true next-state distribution $\eta_{Z(S',A')}$ where (S', A') are themselves random variables distributed following the true (unknown) dynamics of the environment, $\mathcal{T}(s'|x, a)$, and the resulting action taken from the acting policy $a' = \pi(s')$, forming a joint distribution $p(s', a'|s, a)$. Using the composition of probability laws:

$$\begin{aligned}\eta_{Z(S',A')}(z) &= \int_{s',a'} \eta_{Z(s',a')}(z)\eta_{S',A'}(s', a')ds'da' \\ &= \int_{s',a'} \eta_{Z(s',a')}(z)p(s', a'|s, a)ds'da'\end{aligned}$$

We can evaluate the expectation of next-state return distribution $Z(s', a')$ where the next state-action pair (s', a') has been sampled from $p(s', a'|s, a)$:

$$\begin{aligned}\mathop{\mathbb{E}}_{(s',a')\sim p(s',a'|s,a)}[\eta_{Z(s',a')}(z)] &= \int_{s',a'} \eta_{Z(s',a')}(z)P(s', a'|s, a)ds'da' \\ &= \eta_{Z(S',A')}(z)\end{aligned}$$

We have shown unbiased estimation of the distributional Bellman target for sampled rewards as well as for sampled environment transitions. Hence the stochastic Bellman target $\widehat{\mathcal{D}}^\pi Z(s, a) \overset{D}{=} r + \gamma Z(s', a')$ is an unbiased estimate of the true distributional target $\mathcal{D}^\pi Z(s, a) \overset{D}{=} R(s, a) + \gamma Z(S', A')$.

The variance of the updates (i.e variance of the incoming samples from the true value function distribution), affects the rate of convergence of the estimated pdf to the true one. Running some simple experiments (Figure 3.4) the

convergence measured with the Wasserstein metric between the true and estimated distributions, follows an inverse square law with respect to the number of samples.

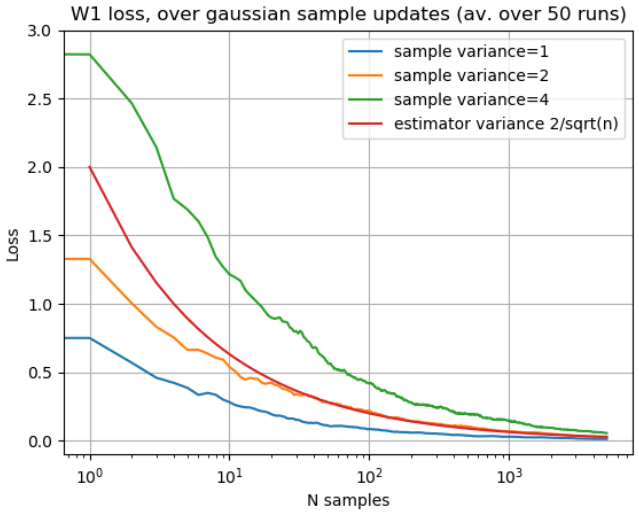


Figure 3.4: Value of Wasserstein metric following sample updates with $\widehat{\mathcal{D}}^{\pi}$.

Control

Just as in the classical expectation-based RL, we must consider the extension of distributional RL operators in both *evaluation* (previous section) and *control*. Evaluation deals only with learning the distribution of returns for an agent following a fixed policy π. During evaluation, the neural network weights will converge towards the solution that gives the true value-function for that policy. In the case of control, the policy that the agent follows is continuously improved until the optimal policy is reached, and during which time the neural network will converge to the policy that the agent is following at the time; eventually learning the value function of the optimal policy.

The distributional Bellman optimality operator is defined as the following:

$$(\mathcal{D}Z)(s,a) :\overset{\mathrm{D}}{=} R(s,a) + \gamma Z(s',a'), \tag{3.5}$$
$$s' \sim \mathcal{T}(s'|s,a), a' = \mathrm{argmax}_{a' \in \mathcal{A}} \mathcal{M}(Z_{target}(s',a'))$$

Where $\mathcal{M}(Z(s,a))$ is some measure of the distribution of $Z(s,a)$, which a greedy policy will maximize for the best performance. Most commonly this is

the mean (classical Q-learning), however having access to the whole distribution $\mathcal{M}$ can be the variance, tail statistics or quantiles, for example, or another form of action selection based on mode detection, in order for different actions to be ranked from best to worst. In our application of distributional RL, we expect to use some custom measure making use of the presence of modes, in order to select the best possible action in the current state.

The stochastic approximation of the control operator $\widehat{\mathcal{D}}$ corresponds to a similar sample-based update as for the evaluation setting:

$$(\widehat{\mathcal{D}}Z)(s,a) :\overset{D}{=} r + \gamma Z(s',a') \tag{3.6}$$
$$r \sim R(s,a), s' \sim S', a' = \text{argmax}_{a' \in \mathcal{A}} \mathcal{M}(Z_{target}(s',a'))$$

$\widehat{\mathcal{T}}$ is a biased estimate of $\mathcal{T}$:

However, it has been shown that $\widehat{\mathcal{D}}$ is actually a biased estimate of $\mathcal{D}$ ([Amortila et al., 2020], Theorem 5.3), where if Z^*, $\widehat{Z}^*$ are the fixed points of $\mathcal{D}, \widehat{\mathcal{D}}$ repecfully, then $\mathbb{E}[\widehat{Z}^*] \geq Z^*$, with the equality holding if and only if the expectation and maximum operators commute (i.e. $\mathbb{E}[\widehat{\mathcal{D}}Z] = \widehat{\mathcal{D}}\mathbb{E}[Z]$). This unfortunately means that, generally:

$$\lim_{N_{\text{samples}} \to \infty} \widehat{\mathcal{D}}Z(s,a) \overset{D}{\neq} \mathcal{D}Z(s,a),$$

And so there will be a slight over-estimation of the return distribution in the case of the stochastic operator $\widehat{\mathcal{D}}$. This is for similar reasons as for the value over-estimation problem mentioned in section 2.1.3, though it can be compensated for during training by double Q-learning, for example.

Parametric Probability Densities

A final consideration we must make before applying this framework to a working algorithm, is the parametrization of the return distribution. In practice when we are using the estimation of returns from next states, we have access to the output parametrization, Π_ψ, of the return distribution:

$$Z_\psi \overset{D}{=} \Pi_\psi Z \Leftrightarrow \eta_\psi = \Pi_\psi \eta$$

We then define the parametric form of the distributional Bellman operator as: $\Pi_\psi \mathcal{D}$.

It should be the case that the fixed point for $\Pi_\psi \mathcal{D}$ is the parametrized form of the fixed point solution to $\mathcal{D}$, such that if Z^*, Z^*_ψ are the fixed points of $\mathcal{D}^\pi$

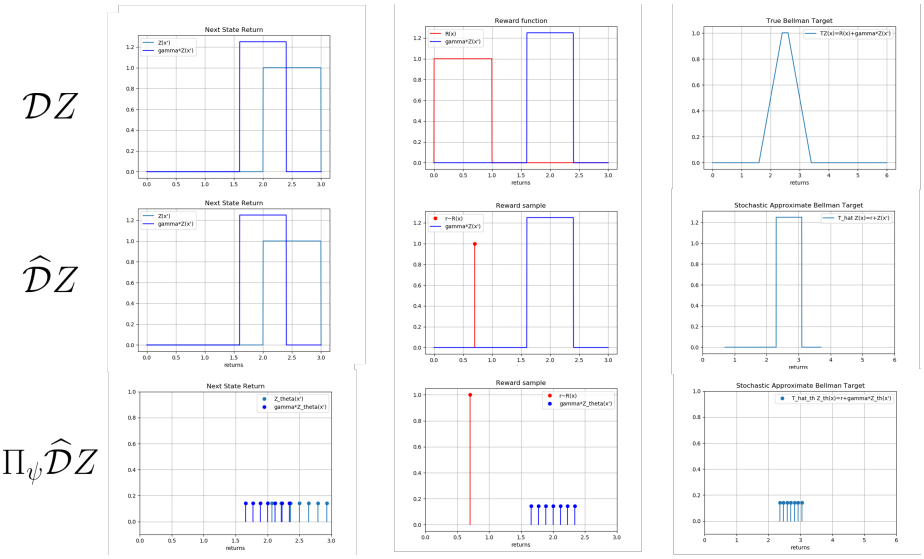


Figure 3.5: Bellman operators from including sample-based and parametric targets

and $\Pi_\psi \mathcal{D}^\pi$ respectively, (assume that the fixed point solutions exist) then we wish the following to be true:

$$Z^*_\psi \overset{D}{=} \Pi_\psi Z^*. \tag{3.7}$$

This property is referred to as *Bellman closedness* by Rowland et al. [2019], and refers to the ability of a parametric update to converge to the same point as the true Bellman update. A more in-depth discussion of distribution parametrization is present in appendix B.4.

A commonly used parametrization is the dirac-quantile parametrization, where ψ_i is an estimation of the q_κ quantile ($\kappa = \frac{i}{N}$ for N total quantiles, not considering quantile q_0) of Z:

$$Z_\psi(s,a) \overset{D}{=} \frac{1}{N}\sum_{i=1}^{N} \delta_{\psi_i}$$

One advantage of quantiles is that they have the nice property of following the same shift and translation as is applied to the corresponding random variable: given two random variables, X, Z, $r \in \mathbb{R}$, and $\gamma \in (0,1]$ such that $Z \overset{D}{=} r + \gamma X$, then:

$$\Pi_\psi Z = \gamma \Pi_\psi X + r$$

This means we can directly apply the Bellman equation to each quantile in order

to compute its target for learning updates:

$$\mathcal{D}\psi_i(s,a) = r + \gamma\psi_i(x', a^*) \tag{3.8}$$

where a^* can be chosen either for evaluation or control.

Figure 3.5 shows the effect of applying the various distributional RL operators to return estimations, which are then used as targets to train the value network. We can see that the true update should take into account the probabilistic nature of the reward signal that is perceived by the agent at every timestep. However due to the sampled nature of an agent's interaction with its environment, we learn with an objective which is not representative of the true return distribution we would like the agent to estimate.

Loss Function

The dirac parametrization Π_ψ of the return estimation allows us to slightly re-write the quantile regression loss, taking into account the positions of each dirac δ_{ψ_i}. Recalling the expression for quantile regression loss from (3.1), we re-write it using the parameter ψ_i:

$$QR(\psi_i, \eta, \kappa) = \mathop{\mathbb{E}}_{z\sim\eta}[(z-\psi_i)\kappa\mathbb{1}_{z>\psi_i} + (\kappa-1)\mathbb{1}_{z<\psi_i}] \tag{3.9}$$

In our case, the target distribution η is itself a mixture of diracs, obtained from our estimation of the value of the next state, and taking into account the perceived reward r along with scaling by γ, according to (3.8):

$$\eta_\psi = \frac{1}{N}\sum_{j=1}^{N}\delta_{r+\gamma\psi'_j},$$

where the ψ' contains the quantiles from the next-state estimated distribution: $\Pi_\psi Z(s', a')$ (output of our value network). We can now explicitly compute the expectation over the new target distribution η_ψ so that:

$$QR(\psi_i, \eta_{\psi'}, \kappa) = \frac{1}{N}\sum_{j=1}^{N}\Big[(r+\gamma\psi'_j - \psi_i)(\kappa\mathbb{1}_{r+\gamma\psi'_j>\psi_i} + (\kappa-1)\mathbb{1}_{r+\gamma\psi'_j<\psi_i})\Big] \tag{3.10}$$

We can sum the quantile loss over all the current quantile estimate ψ_i, to find the loss that should be optimized per training step of the RL algorithm:

$$\begin{aligned} L(\psi, \eta_{\psi'}) &= \sum_{i=1}^{N} QR(\psi_i, \eta_{\psi'}, \kappa) \\ &= \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{N}\left[\xi_{ij}\cdot(\kappa\mathbb{1}_{\xi ij>0} + (\kappa-1)\mathbb{1}_{\xi_{ij}<0})\right] \end{aligned} \tag{3.11}$$

where $\xi_{ij} = r + \gamma\psi'_j - \psi_i$.

3.2 Experiments

In this section, we present our main simulation environment which was used to model an intersection task for an autonomous vehicle, along with the initial results that we obtained when applying distributional RL algorithms to a task requiring both levels of performance and safety, in a partially observable environment. Part of the work done during this thesis was to develop our own intersection environment in order to have a greater flexibility for applying our algorithms to various different testing scenarios. There exist many free open-source traffic simulators (CARLA, Sumo, Openroad, ...) that are used in research projects, however they lack the modularity and ability to easily develop and implement different task setups and vehicles behaviours, for example. For this main reason, with the path-planning team in Renault Software Labs we have developed a simple driving task simulator, allowing a controllable vehicle to make decisions in order to complete the navigation task while avoiding collisions with oncoming target vehicles.

The simplified nature of this environment also allowed us to perform more simulation runs, which is often a crucial point to take into consideration when training an agent to learn complex behaviours, with limited computational resources. As a point of comparison, for the neural network architecture we used, it would take no more than 500k training steps (approximately 2-3 hours) to max out performance in most tasks, whereas popular benchmarks such as Hessel et al. [2017] present training regimes of up to 200 million training frames (the equivalent of over a month of training time with our resources). Hence, especially with reinforcement learning which requires much hyper-parameter adjustment, having as efficient a simulator as possible is vital. In the next section we briefly present the environment used for simulation an autonomous navigation task, however a more in-depth discussion about the design and implications of designing a simulation environment is given in appendix A.

3.2.1 Training Environment

Our simulation environment consists of a road intersection, where the controlled vehicle (ego vehicle) has the objective of passing through the intersection without crashing into any of the other vehicles (target vehicles) present in the driving scenario. Figure 3.6 shows an illustration of the environment. Blue ellipses around the targets represent uncertainty on their position in the scene, due to sensor uncertainty. This kind of intersection scenario is commonly used in state-of-the-art applications of RL to autonomous navigation, as it allows us to

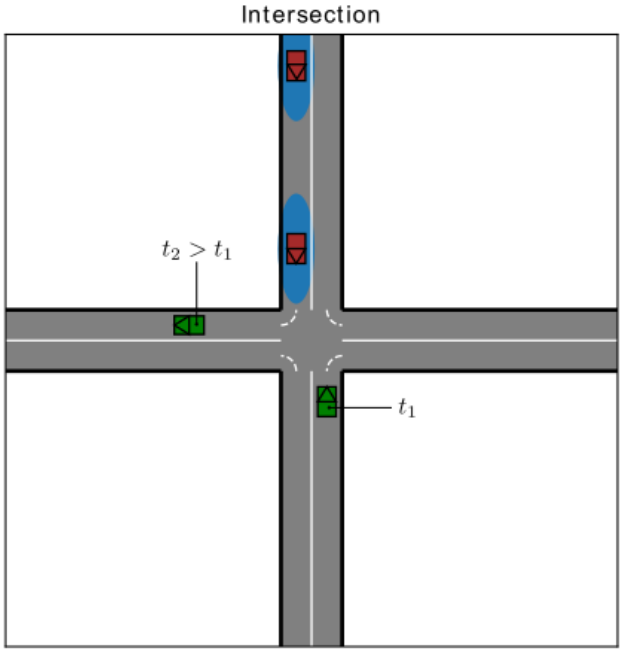


Figure 3.6: Left-hand turn intersection task

model the need for nuanced decision-making in the face of uncertainty, from the point of view of the ego vehicle [Brechtel et al., 2014, Hubmann et al., 2018, Bouton et al., 2019, Bernhard et al., 2019, Rhinehart et al., 2021].

Ego and Target Behaviour

The ego vehicle is initialized with an objective direction, out of the following possibilities: {straight, left, right}, and so the path is pre-determined. The target vehicles' path is also pre-determined, and in this use-case they keep going straight ahead. Since the ego's path is set at the beginning of training, the navigation task boils down to planning a correct speed profile in order to avoid any collisions while still making it through the intersection. The set of actions available to the agent (ego vehicle) are:

$$\mathcal{A} = \{-4, 2, -1, 0, 1, 2\}\ m/s^2$$

which correspond to longitudinal acceleration values (i.e. ego is able to accelerate, maintain speed, break, of perform a hard emergency break of -4 m/s^2.

Target vehicles start off at some initial speed around 20 m/s, and in our initial use-case, have no reaction to the presence of the ego vehicle (i.e. we assume

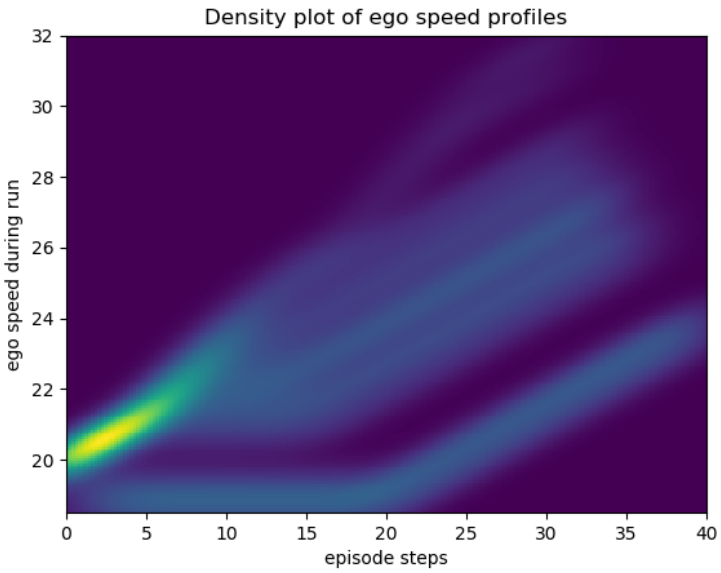


Figure 3.7: Density of speed profiles, over multiple episode runs

right of way for the target vehicles). They are initially spaced in such a way that the ego vehile has a couple of valid strategies for solving the navigation task: either have an aggressive approach, and speed up in order to pass the intersection before a dense series of targets pass through (higher performance but higher risk strategy, denoted π_1), or maintain a conservative approach and wait for the dense train of vehicles to pass through the intersection before crossing (lower risk but lower performance, denoted π_2). This experimental set-up is made to represent a core dilemma we wish to tackle in our probabilistic modeling approach: how to balance out safety concerns with the goal of having a well-performing agent in terms of time to complete the navigation task. Although safety is a primary concern and takes precedence over the notion of time performance, we do not wish for the autonomous agent to be overly cautious – as this can lead to freezing robot problem, a situation where a robot freezes up and does not act since it prefers not to act rather than risk taking an unsafe action. A simple scenario such as this should allow the agent to represent multiple modes of behaviour, represented by π_1 and π_2, and according to the observability of the scene, be able to determine which course of actions is best for that use-case.

In order to analyze the behaviour of the ego vehicle, we can analyze the ego's speed profile. Figure 3.7 shows the density of speed profiles, of a fully trained expected-value policy, when reacting to oncoming vehicles with *varying speeds* (over many episodes), under *full observability*. For every episode, the agent

makes the choice, through a sequence of actions, to either speed up (higher ego speed values: π_1), or slow down (lower ego speed values: π_2). We can see that the resulting behaviour of the ego is distributed along the two possible modes of behaviour, one being low speed, and one being high-speed, with the variance caused by the slight difference in initial positions from the oncoming target vehicles. We can see due to the strong bright patch of initial increasing speed, meaning that under full observability conditions the ego will prefer to take higher-speed trajectories due to the low level of risk.

The aim for a distributional algorithm, is to be able to characterize the reward function in a similar way, if the uncertainty in the *observation model* of it's environment allows for more than one of the 'family' of policies π_1 or π_2 to be the best course of action.

State Space

The state space $\mathcal{S}$ is the input space to our agent policy π. In this scenario, we use an input space based on a list of the 3 closest detected objects to the ego vehicle, assuming the use-case where some sensor fusion pre-processing has already been done by the car's various sensors. In general, one of the challenges of developing an autonomous agent is that the amount of information in the input space can change dramatically: we expect an agent to be able to handle an intersection whether there be no oncoming targets, or 10 of them. Some approaches deal with this issue by using an occupancy grid on an area around the ego, or convolutional filters on a version of this occupancy grid, so that the dimension of the input space remains the same in any situation. In our case, we use a list of the three closest target vehicles to the ego including their positions (with some amount of noise represented as blue circles on the sample frame of the environment), speeds (also subject to noise), and estimated *time to collision* between the ego and relevant target under a constant speed hypothesis:

$$s = \{s_{\text{ego}}, \dot{s}_{\text{ego}}, s_1, \dot{s}_1, \text{ttc}_1, s_2, \dot{s}_2, \text{ttc}_2, s_3, \dot{s}_3, \text{ttc}_3\} \tag{3.12}$$

Each of these values is normalized with respect to some maximum value in order to normalize the input to our network.

We notice that there is some redundant information contained in the state, between the positions and speeds of target vehicles, and the resulting estimated time to collision between target and ego. Although this is the case, we quickly found that adding the extra information of time to collision allows for faster learning from the agent's policy. This is probably due to the fact that time to collision is an important part to deciding which action to take, and hence this value was implicitly learned anyway. Since this value is quite straightforward to compute and provide to the ego, we decided to include it in the available information for the agent to best react to the current environment state.

In the case that there are less than 3 target vehicles in proximity to the ego, the useless positions and speeds are set to 1 and 0 respectively, representing a target vehicle that is already passed the intersection, and hence the ego should have learned not to change its behaviour with respect to it.

Reward Function

As discussed at the end of chapter 2, designing a good reward function is a challenging part of implementing an RL algorithm. In order to encourage the agent to navigate through the intersection as fast as possible, we give it a small negative penalty for every time step. Representing passenger comfort, we would like to avoid any hard breaking if possible, also attributing some small negative reward for any hard-breaking action. Finally we attribute the maximum penalty of -1 if ever there is a collision between ego and target vehicles. The reward function is as follows:

$$\begin{cases} \text{each time step} & r = -0.001 \\ \text{ego hard brake} & r = -0.002 \\ \text{collision} & r = -1 \end{cases}$$

We have previously discussed the limitations of having to include many different objectives (here speed, comfort, and safety) in a single optimization algorithm. The design in this case is supposed to represent a hierarchy of objectives: safety being the most important, and comfort and speed being optimized with approximately the same level of importance. Our simulation environment uses a time-resolution of 0.1s per agent action, so in this case we are telling this agent that a 1% chance of collision is equivalent to saving 10 seconds of time from the navigation task. This may seem small, though as we will see later on, the desired hierarchy of objectives poses some issues when wanting to represent them as part of the return distribution.

In terms of the reward distribution, we expect to see multiple modes emerge from the various possible trajectories an agent can take during the navigation task. For example any trajectory whose outcome results in a collision should push the return distribution to represent a mode around the -1 value. After this, any agent behaviour which completes the task without collisions should appear as modes of return closer to 0 in the return distribution. The probability mass associated to the modes should represent the probability of that outcome occurring for the relevant agent following its expected trajectory. For example since the returns around -1 group all collision trajectories, these should be treated as the probability for a collision to occur, and can be estimated from the positions of the quantiles in the return distribution.

Agent Policy Model

We model the agent's policy with a deep neural network, with two hidden layers of 200 neurons each. We use Rectified Linear activation Units (ReLU) as neuron activation functions. We implement a distributional algorithm using quantile regression (QR-DQN) in order to capture modes of behaviour in the environment. This uses the same basic network shape, only the output layer outputs a set of N quantile estimates instead of a single expected value, so the shape is $11 \times 200 + 200 \times 200 + 200 \times (6 \times N)$. Theoretically, the number N of quantile estimates depends on what kind of 'resolution' we wish to have for representing low probability events, in terms of their return value. For example if the probability of occurrence for a certain outcome of agent trajectory τ_A is $P(\tau_A) = 0.05$, then we should have a number of quantiles $N > \frac{1}{P(\tau_A)} = 20$ in order to assign probability mass to this event without also falsely assigning it to other return values. For example in our experiments with a high quantile resolution we use $N = 31$, resulting in approximately 80,000 weights and biases. We use an odd number of quantiles so that the middle quantile is able to represent the mean of the distribution.

For training, we use the same approach as a deep Q-netowork, the only difference being the backpropagation of gradients using the quantile loss for each action output. Algorithm 2 gives an overview of the training regime for a distributional algorithm. The BackpropQR function shows how the loss is computed from estimates of the return distribution using interaction samples from the replay buffer. In order to use the Bellman optimality operator (3.5), we must decide on a strategy $\mathcal{M}$ for choosing the optimal action according to the estimated distribution. We have presented multiple possible approaches for this, such as Conditional Value at Risk, reacting to specific quantiles of the distribution which represent certain probabilistic likelihoods of worst-case scenarios for the agent. Though we anticipate using some form of mode detection in order to inform agent behaviour, our initial implementation simply uses the empiric mean of the estimated distribution: $\mathcal{M}(Q_\theta(o, a)) = \frac{1}{N} \sum_{i=1}^{N} \psi_i$. This allows us to simply rank the actions making it easy to choose the best next extimated action. Using the mean of the distribution should yield similar results to expected-value algorithms, though what we are looking for is to be able to correctly represent the return modes first, and afterwards we may burden ourselves with the task of having the agent qualitatively rank the return distributions to choose the best action for the current time step.

In these experiments we use a memory buffer of size 10,000 along with an ε-greedy exploration strategy with an exponential decay of 0.998 every 100 steps, decaying to a minimum value of $\varepsilon = 0.01$ (1% chance of taking a random action) after approximately 200k steps. We present experiments using $N = 11$ quantiles, making for a network with shape: $11 \times 200 \times 200 \times 66$ for a total of 110,800 trainable weights and biases. We use the learning rate $lr = 2.5 \times 10^{-4}$.

Algorithm 2 Training Quantile Q-network

Init Q_θ
while not converged **do**
  $s_0 \sim p^\pi(s_0)$ ▷ init. episode state
  $o_0 \sim O(s_0)$
  $\text{Exp} = \{\}$ ▷ init. episode experience
  while episode not terminated **do** ▷ play episode
    $a_t = \varepsilon\text{-greedy}(\arg\max_{a \in \mathcal{A}} Q_\theta(o_t, a))$
    $s_{t+1} \sim \mathcal{T}(s_t, a_t)$ ▷ environment step
    $r_t = R(s_t, a_t, s_{t+1})$ ▷ step reward
    $o_{t+1} \sim O(s_{t+1})$
    $\text{Exp} = \text{Exp} \cup \{o_t, a_t, r_t, o_{t+1}\}$
  end while
  $\text{Memory}(\pi) \leftarrow \text{Exp}$ ▷ store samples in replay buffer
  $\text{Exp}' \sim \text{Memory}(\pi)$ ▷ sample from replay buffer
  BackpropQR(Q_θ, Exp') ▷ train network on samples
end while

function BackpropQR(Q_θ, Exp) ▷ compute loss according to (3.11)
  for $(o_t, a_t, r_t, o_{t+1}) \in \text{Exp}$ **do**
    $\boldsymbol{\psi} = Q_\theta(o_t, a_t)$ ▷ current quantile estimate
    $a^* = \arg\max_{a'} \mathcal{M}(Q_\theta(o_{t+1}, a'))$ ▷ next optimal action
    $\boldsymbol{\psi}' = Q_{\theta'}(o_{t+1}, a^*)$ ▷ next-step quantile estimate
    $\xi_{ij} = r_t + \gamma \psi'_j - \psi_i$
    $QR = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} \left[\xi_{ij} \cdot (\kappa \mathbb{1}_{\xi ij > 0} + (\kappa - 1) \mathbb{1}_{\xi_{ij} < 0}) \right]$
    Perform gradient descent on QR loss
  end for
end function

3.2.2 Results

In this section we present an analysis of applying the distributional algorithm QR-DQN to our simulated intersection scenario, under various conditions of observability.

Return Estimation During Training Phase

We can look at the shape the return distribution has during the agent's training phase, before it converges to a single value when there is no uncertainty of outcome. From value networks that had not yet converged, we can see some evidence of the ego trying out both π_1 (higher returns) and π_2 (lower returns) to avoid collisions with the target vehicle:

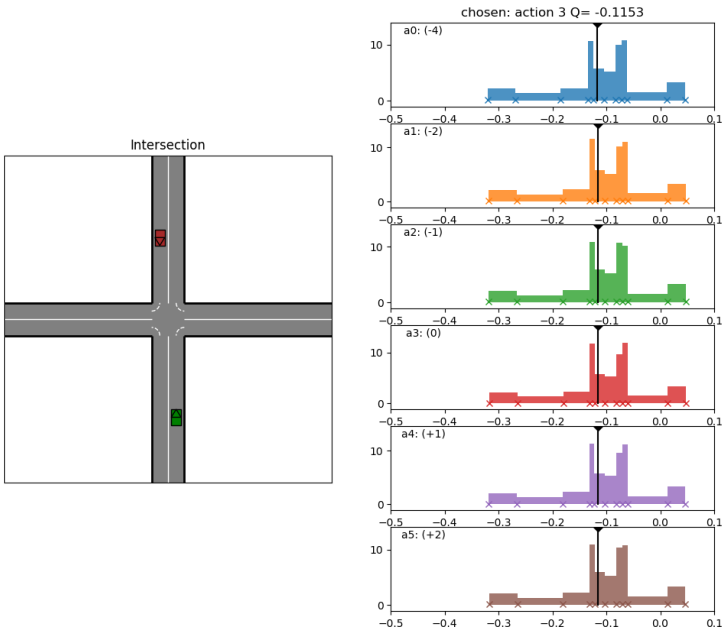


Figure 3.8: Intersection scenario (right), along with return distributions corresponding to each possible action (left).

Figure 3.8 shows the estimated return distribution for each action, in a state

where the ego vehicle is approaching the intersection, with an oncoming target vehicle. Superposed plots on the right show the estimated return for each of the different actions, the black bar indicates the empirical mean of each distribution. What immediately stands out is the similarity between the estimated return of each possible action. This is due to two criteria: firstly, the time-step length for the agent is 0.1 seconds for each action. This means that any major difference in the return distributions of each action will be caused by if there is a clear demarcation in strategy in the next 0.1 seconds. In other words, the shorter the time scale, the less of an immediate impact a single action will have, until the 'point of no return' for one outcome or another.

Another consideration we must have is linked to value overestimation problem we have when learning the optimal policy. Assuming that the policy will take the optimal action in the next state, this means that the algorithm does not take into consideration the possibility of lower-return outcomes if it consider that the agent is able to avoid them, and hence this acts as a sort of 'optimism screen' whereby the agent judges the value of the state according to the best possible outcome of the next states. If the timescale is short, then the next states $\{s' \sim \mathcal{T}(s'|s,a), a \in \mathcal{A}\}$ according to all available actions will be very similar, and so the optimal strategy in each case will be equally similar. This leads to a trade-off where analyzing the distribution of returns per action makes it more difficult to predict the different effect of each of the possible strategies π_1, π_2 which themselves are actually made up of a series of actions.

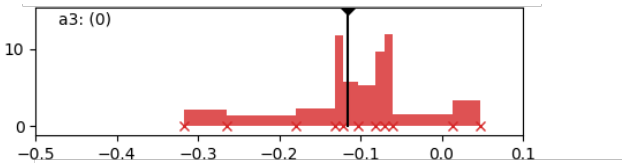


Figure 3.9: Return estimation for a single action (0 acceleration action), during training.

Figure 3.9 shows the return distribution for a single action, the black bar indicating the empirical mean of the parametrized distribution (i.e. the middle quantile). We can see evidence of two return modes between 0 and -0.2, corresponding to strategies from the agent passing both in front and behind the initial target crossing the intersection. This corresponds to the initial exploration done by the agent such that random actions cause it to encounter both possible outcomes. We also see a lack of any quantiles close to returns of -1 (collision event), such that we can deduce that at this point in the training the agent has learned to avoid collisions and so does not consider any collision outcomes in its return estimation.

However, the presence multiple modes in a fully observable environment is in part due to the initial exploration of the agent, meaning that multiple

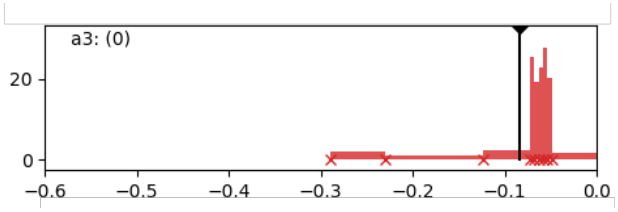


Figure 3.10: Return estimation for a single action (0 acceleration action), at the end of training.

strategies are investigated at first, and what we see eventually is a collapse of the modes of return that are not exploited anymore by the agent once it has converged to a behaviour it finds suitable. This phenomenon is known in RL and is usually referred to as *catastrophic forgetting*, where the agent discovers potentially interesting behaviours during its initial environment exploration, however since these alternate strategies are sub-optimal in terms of the returns they obtain from the environment, they are gradually replaced by a single mode of behaviour deemed optimal by the agent's learned value function.

Figure 3.10 shows the return estimation from quantiles at the same state as the previous Figure 3.9, at the end of training once the agent's policy has converged. We can see how the previous mode to the left of -0.1 is no longer present in the distribution estimation, making the mean value shift to the right. This is because of the reduced amount of exploration performed by the agent during the later stages of training, meaning it has learned to exploit a single mode of behaviour in order to solve the environment. This 'mode collapse' also illustrates how a certain behaviour can be observed by the agent during training and subsequently be forgotten if it is deemed sub-optimal. The leaning problem is formulated such that we encourage the agent to explore the space of policies by random state exploration. As mentioned in chapter 2, part of this reasoning is so that the agent avoids getting stuck in local optima of the return function, such that it is able to find the global optimum to the control task. However the design means that local optima, or global sub-optimal solutions are discarded during the training process even though they may have been learned at one point during the agent's training. We will see later on that this forgetfulness may actually be detrimental to the ability for an agent to generalize to unseen scenarios, where sub-optimal solutions have the capacity of still performing well under certain conditions that may not have been represented in the training data distribution.

In this example where the agent has full observability of the environment (target speed, target position, target behaviour, etc.) it makes sense that it is able to deterministically estimate the outcome of its policy, having observed that outcome many times over the last training episodes.

Multi-modality in the return distribution

In the following part of our experiments, we wish to observe how adding uncertainties to the environment will affect the shape of the reward distribution. We make the hypothesis that higher values of uncertainty should induce clearer demarcation between different modes corresponding to outcome probabilities in the return distribution. To induce this multi-modality we add an observation model $O(o|s)$ to the MDP, affecting the agent's perception of environment state. In these experiments we add a Gaussian noise around the target vehicles' true position as expressed in (3.12), such that:

$$O(o|s) \stackrel{D}{=} \{(s_{\text{ego}}, \dot{s}_{\text{ego}}, \mathcal{N}(s_1, \sigma), \dot{s}_1, \text{ttc}_1, \mathcal{N}(s_2, \sigma), \dot{s}_2, \text{ttc}_2, \mathcal{N}(s_3, \sigma), \dot{s}_3, \text{ttc}_3\} \tag{3.13}$$

with a variance of $\sigma^2 = 5m$ (in meters. for reference the length of the vehicles is 2m). Since the uncertainty related to the position of target vehicles, the uncertainty affects only their longitudinal position inside the same lane, and we assume a perfect detection of the lane the target vehicle is in. This is because of the nature of the intersection environment which we are investigating here, in which we do not consider any lane-changes from the part of the ego or any of the targets (single lanes).

The uncertainty on position is represented by a circle with a radius of 2 standard deviations around the target vehicles, as illustrated on the left-hand side of Figure 3.11. We can see from the estimated distribution that the current observation model has had little to no effect in terms of new modes of outcome being represented by the output quantiles of agent's policy. We can clearly see that the return estimation has converged to a single mode, and considers that there is a 0 probability for there to be a collision with the target vehicle under the agent's current policy. When looking at sample episodes, we can indeed see that the ego is still able to navigate this intersection environment without entering into the 2σ zone around the target, i.e. avoiding any collisions with at least 0.95 probability. The convergence of the policy to a single mode means that this policy has a high confidence in its ability to solve the environment, placing no probability mass around the returns for any other outcome.

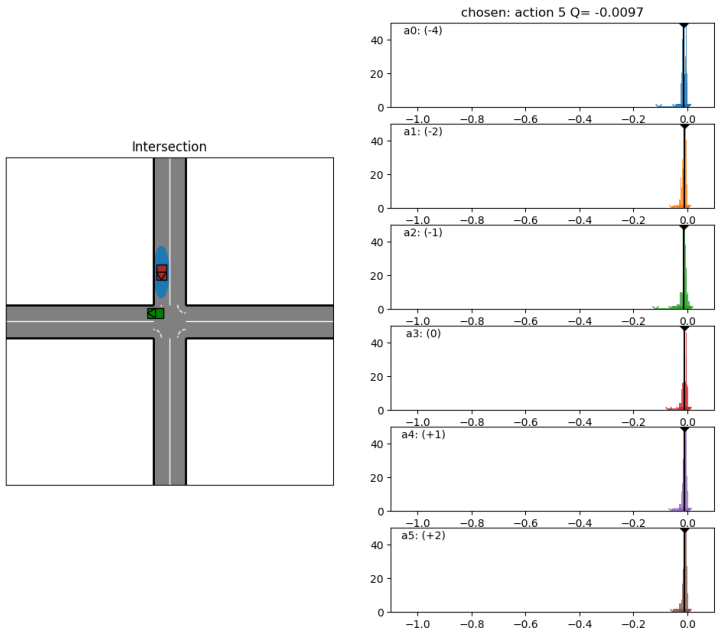


Figure 3.11: Intersection scenario (right), along with return estimations (left), targets having Gaussian noise on their observed positions ($\sigma^2 = 5$).

We can further increase the noise on the target vehicles' observed position in order to observe the multiple possible outcomes as modes in the return distribution. We run the same training setup as in the previous experiment with a single target vehicle crossing the intersection, though this time with a higher uncertainty on its position ($\sigma^2 = 10m$) in the agent's observation model (3.13). In this use-case we expect the learned quantiles to represent modes of outcomes according to the probability of occurrence (indirectly the frequency at which the network is trained to observe particular return values). Figure 3.12 shows the result of training the same QR-DQN set-up as for the previous case, simply increasing the variance linked to the agent's observation model.

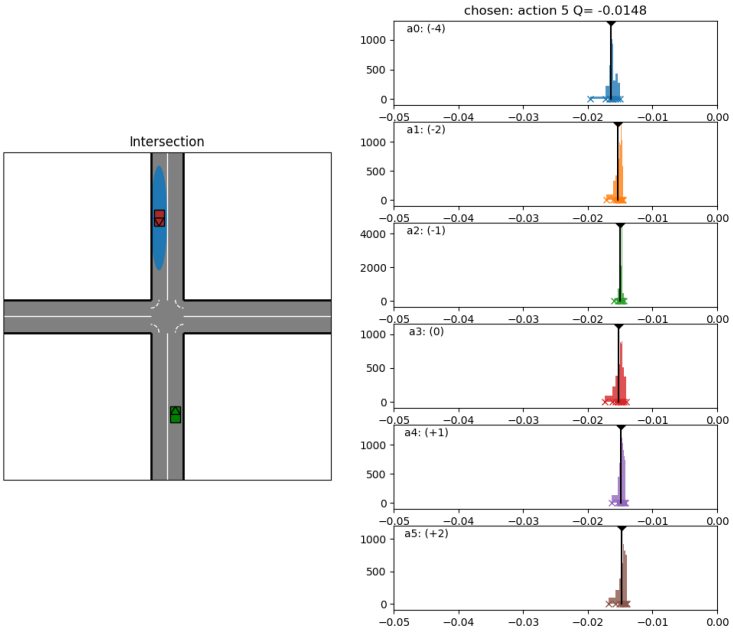


Figure 3.12: Increased uncertainty on target position ($\sigma^2 = 10m$).

In this case, we can conclude that the increased uncertainty is not enough to force the agent into having to make any significant trade-offs in terms of expectation. This is seen from the low variance in the return estimation which can be interpreted as a high confidence in terms of expected return estimated by the policy. On the other hand, we do see that the agent has learned to steer clear of the zone of uncertainty so that it can maintain as deterministic an outcome as possible, though this is equivalent to increasing the radius for collision detection between the ego and target vehicles. One point to note for these experiments is that the model for sensor uncertainty (Gaussian noise) does not take into account the variation in levels of uncertainty according to the distance of the target vehicles to the ego's sensors. This means we model the variance on position as aleatoric meaning it cannot be reduced through exploratory actions, although in real-world applications, some of the uncertainties linked to sensors are actually epistemic and reduced, for example, as the detected targets gets closer to the ego's sensors. This should allow an agent to take actions in order to reduce the uncertainty of outcome according to some exploratory actions, for example slowly approaching an intersection if it is uncertain about exactly where the target vehicles are located while they are far away. We could seek

to implement a more accurate model for sensor uncertainty, however this is not yet vital to demonstrate the usefulness of the proposed approaches for tackling these uncertainties when they are present in the environment.

Figure 3.13 shows the quantile regression loss computed during backpropagation of loss through the Q-network. The convergence of this loss to a minimum value close to 0 indicates that the current and next-state quantiles ψ_i and $r + \gamma\psi'_j$, are close and hence that the Q^π-network has been able to learn the correct return distribution for that policy. We are able to see how fast the estimation error of the return model Q_θ falls to 0, even before the end of the exploratory phase from the ε-greedy strategy at around 200k steps, meaning that the agent is able to very quickly detect an optimal point in policy space and converge to it. We see some continual 'flickering' in the loss ever after it appears to have reached its minimum: this is due to the minimum value $\varepsilon = 0.01$ meaning that in 1% of cases the agent will take a random unexpected action which has the potential of receiving an unexpected reward, causing some error between the expected and observed returns. We can observe in this case that the policy seems stable, since the loss is relatively small the updates on policy parameters θ are rather small.

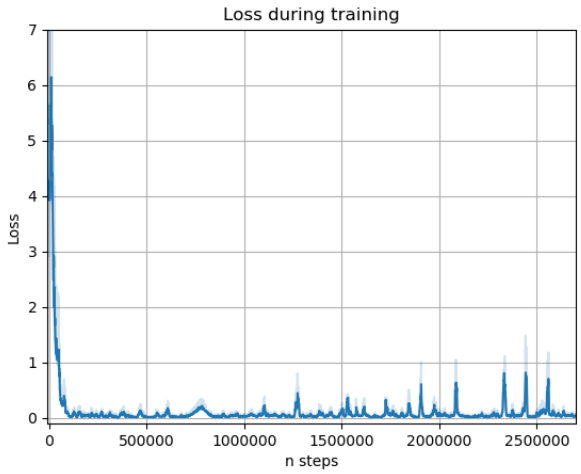


Figure 3.13: QR loss over 2.5M training steps.

Although the previous result is a testament to the adaptability of a neural network policy model, we wish to place the agent in a situation where uncertainty of outcome is inevitable, such that the policy is *not confident* with respect to its performance because of the uncertainty present in the environment. It is in these cases where policy confidence is low, and the agent is able to attribute some probability mass to possible collisions, that we expect the qualitative analysis of the return distribution to prove useful. In order to force uncertainty of

outcome, instead of further increasing the uncertainty on the target vehicle's position, we can place the ego in a scenario such that it cannot steer completely clear of a possible collision. Additionally we prevent the ego from simply waiting for the target to pass through the intersection by placing a time constraint on the agent for crossing the intersection.

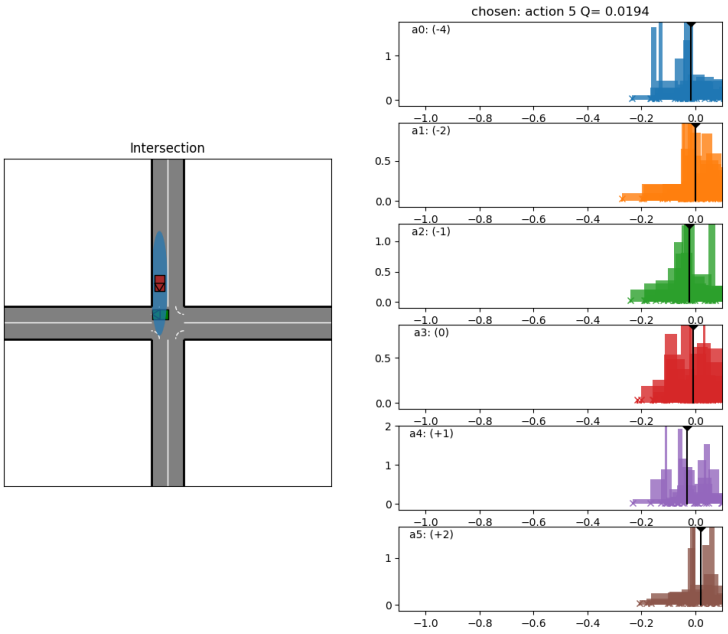


Figure 3.14: Intersection scenario (left) with quantile estimates (right), with high uncertainty on target position.

Figure 3.14 shows the effect of training the QR-DQN algorithm over a long period of time (enough for an algorithm to converge – though we will see that this is unfortunately not the case). We can see that the ego vehicle is clearly within the 2σ radius of the target vehicle, meaning that there is a nonzero probability of collision in this scene.

Figure 3.15 shows a close-up for the quantile estimates of the constant acceleration action. The main difference we see with the previous low-uncertainty experiment (Figure 3.11) is that the variance of the estimated distribution goes up dramatically, though the shape of the return estimation remains difficult to interpret from a qualitative standpoint: there are no clear modes of outcome which we would expect to see for this situation, for example where the agent

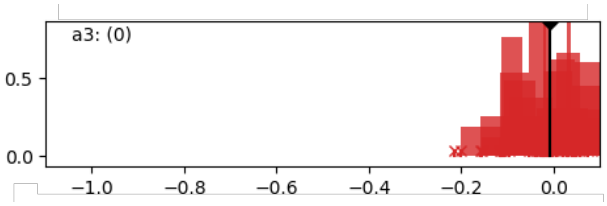


Figure 3.15: High variance on return estimation for 0 acceleration action.

has some probability of crashing. We can see that there is no probability mass attributed to the return values around -1, which would correspond to a collision event. This should be the case however, since there is a nonzero probability that the environment will result in a crash. Simply looking at the distribution mean (expected Q value) might indicate that the expected performance of the agent is quite good, though we can assume from the shape of the distribution that this is due to some lack of convergence to a good estimation of the return quantiles. We also see that some of the probability mass is in the positive domain of the return space. This should not be possible since all reward terms are negative hence episodic returns cannot be positive. This is further indication that the estimates are off and probably invalid. Specifically relating to quantiles in the positive domain, we can be certain that this comes from the intitialization of the value network, where initial guesses for the return quantiles will be equally positive or negative.

Seeing the lack of satisfactory convergence in the quantile estimates of the distribution, we can look to change some of the hyper-parameters used to during the distributional agent's training. The lack of convergence from an RL agent may be attributed to multiple causes: insufficient exploration may lead to the agent missing optimal policies in the return landscape of the RL problem, network learning rates which are not well adapted to the learning problem may hinder trainable parameters from learning the correct parametrization for reproducing the distribution as their model output, or somehow the training data distribution is not representative of the current episodes being played out (linked to the sampling process from the agent's replay buffer – changing the size of the replay buffer has an effect of the iid property of the data which may hinder convergence). Another factor to consider is the architecture of the neural network model. Usually larger and deeper networks with more parameters have more representational power, and are able to deal with more complex environments due to their need to implicitly encode the environment and policy interactions through their weights and biases, although this comes at the cost of increased training time and complexity (possible need for learning rate annealing, for example) due to the high number of parameters to train.

Unstable Learning

Seeing the resulting distribution estimation in the last experimental set-up, we can investigate the performance of the quantile estimates during training by looking the the quantile regression loss function defined in (3.1). We should expect this value to decrease over time while the neural network learns increasingly better estimates of the return distribution. This is due to the quantiles in the return estimation becoming increasingly accurate, and hence on average the difference between ψ_i and $r + \gamma\psi'_j$ should go down to a minimum.

Figure 3.16 shows the QR loss computed on the policy network model over the course of 1M training steps. We see a stark difference when compared to previous Figure 3.13 where the loss converged very rapidly to a minimum value close to 0. In this set-up the network is clearly unable to correctly predict the outcome return distribution, meaning that the agent is constantly being surprised with respect to the outcome of its training episode. This of course is undesirable as the whole idea behind using a deep model to model the return is so that the agent is able to implicitly learn the environment dynamics and avoid surprising outcomes. Seeing that the model must now learn an entire distribution (and not a simple dirac), we expect that training should take longer due to the multiple possible returns possible from a single starting observation. However it seems that the addition of multiple simultaneous outcomes has proven too complex for the agent to learn through quantile regression.

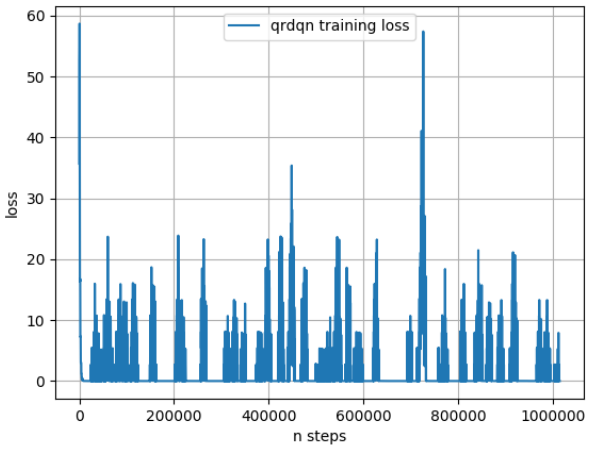


Figure 3.16: QR loss over 1M training steps.

We are able to tune various hyper-parameters of the agent's training phase, following some of the examples that were given earlier. To re-adjust the agent's training to this increased uncertainty we can now run training with a lower

learning rate, and over a longer period of time. This should stop the agent from being too sensitive to outlier updates (at the risk of discarding low-probability events). We can also reduce the size of the replay buffer to attempt to reduce the variance of gradient updates. Modifying the exploration term is not relevant in this use-case: the issue is not with the agent converging to a sub-optimal solution, so increasing or decreasing the exploration rate in the ε-greedy function will not affect convergence. We can make a similar argument with respect to the depth or number of parameters in our deep neural network model. Although it may happen that the agent-environment interaction dynamics are too complex for a small and non-deep (less function composition gives reduced model flexibility) neural net, in our case we have seen in Figure 3.8 for example the distribution is able to capture multiple modes during training before collapsing down to a single mode. Additionally the loss we observe is not typical of an under-parametrized function, where we would still expect some convergence, though to a higher minimum. We may also think of changing the number of quantiles N, although using $N = 11$ we expect to capture modes of events with $P(\tau_A) > 0.1$ which seems sufficient for representing the probability of outcome for the agent's episode at least during the training phase (where much more collisions will happen before the agent learns to avoid then), therefore a finer parametrization doesn't seems useful. It also doesn't seem reasonable to think that the model is over-parametrized since it has perfectly converged to the optimal policy in the previous observation scenarios. For these reasons, we keep the same model architecture and parametrization, and simply reduce the learning rate, replay buffer size, and increase training time. Figure 3.17 shows the same loss estimation during the agent's training phase, over 14M training samples. Initially there is no improvement in the training dynamic and the loss profile looks the same as that of Figure 3.16, however letting the algorithm run until almost 14M samples we see some instability grow which causes the loss to explode. This is representative of the fact that the quantile estimate are way off and we are certain that the network has not learned anything useful. At the same time, Figure 3.18 shows the performance of the RL agent throughout training, in terms of cumulated rewards over a single episode. Even though there is some initial high performance, the more we train the deep Q-network, the more random the policy performance becomes. This is in line with rapidly changing policy parameters since the new points in policy space will be further away and thus more likely to result in different, random behaviours. In this case we can see that the training set-up is such that the agent eventually goes off to some non-coherant, non-stable policy.

From looking at the profile of the loss, and in comparison to our other initial experiments, we can assume that it is the increased variance in quantile regression updates that does not allow for our return esimation to converge in this training paradigm. We recall the light theoretical results of the previous section, where the Bellman optimality operator (3.5) is *not* a contraction mapping and hence there are no strong theoretical guarantees for the Bellman updates to converge to the distributions fixed point. It would appear that in the more

simple cases without multiple outcome modes this approach to learning is sufficient, although we lose much of the potential representative power of modeling entire distributions. We can imagine that there exists a set of training hyperparameters which would allow for a distributional algorithm to correctly model the dynamics of a partially-observable set-up. Though recalling our objective of using a qualitative analysis of the distribution modes in order to inform safe ego behaviour, and seeing how far we are from learning interpretable and thus exploitable return estimates, we can see how it could be difficult to push this approach much further in terms of complexity and nuance in autonomous decision-making. For this reason, in the next section we take a step back and provide some analysis of our distributional implementation in order to attack the problem using a potentially different approach than learning and explicitly modelling all possible probabilities of outcomes.

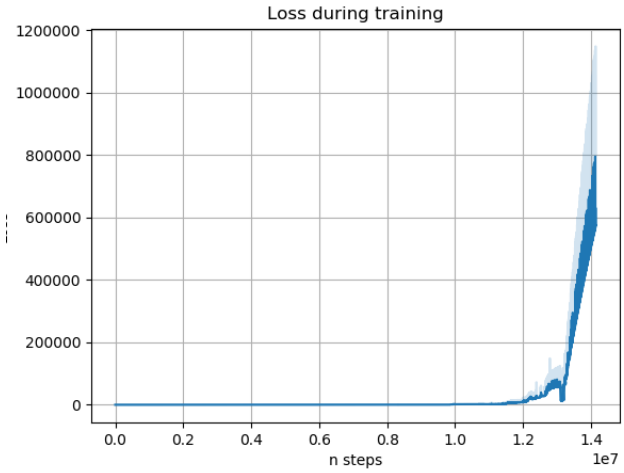


Figure 3.17: QR loss over 14M training steps.

3.2.3 Conclusion on Distributional RL

In this section, we have presented the use of a distributional framework in order to quantify uncertainties linked to the performance of an agent in a stochastic environment. Although this framework is built from a theoretical basis there are no mathematical guarantees for it to function as expected (as is the case with many deep learning approaches), and we have seen this in applying it to a relatively complex aleatoric environment. Distributional RL, and QR-DQN specifically has seen some implementations in works using complex environments [Hessel et al., 2017, Bellemare et al., 2019, Rowland et al., 2019], however none of these are applied to environments with either aleatoric nor epistemic uncer-

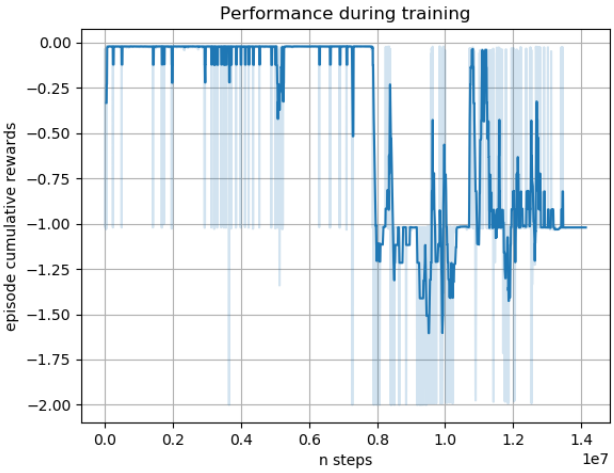


Figure 3.18: QR-DQN performance, over 14 Million samples.

tainty, and use model sizes and computational resources which are substantial.

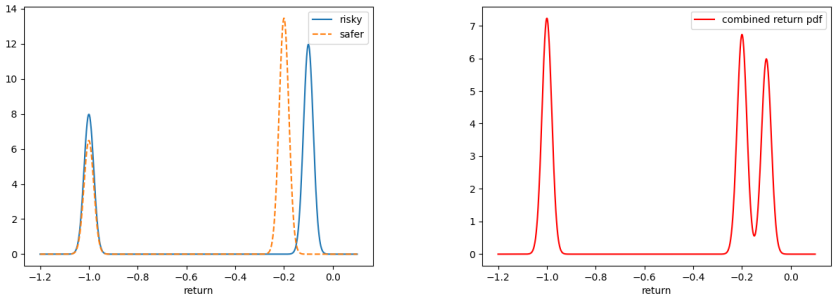
However, although we were not able to reach our objective of learning qualitatively representative distributions of policy returns, through the use and application of this distributional approach to training an RL agent we have gained some insights to our problem of training polices for robust behaviours. We present the main takeaways from our use of a distributional algorithm below:

Modalities in Outcomes vs. Behaviours

We recall that the purpose of representing the entire return distribution was so that an agent having access to return modalities could differentiate between outcomes of different behaviours for a single policy facing an environment containing some level uncertainty in either its transition or observation models. Aside from the ability to correctly estimate this distribution, we make a few remarks on using the return as a proxy for qualifying different agent behaviours. Firstly, we have to be clear by what we wish to model in terms of difference in *outcome* versus difference in agent *behaviour*. The outcome of a policy relates to the final performance or value of a trajectory realized by that agent, and for the same sequence of actions from an agent, we may estimate the probability of different outcomes resulting from that policy (for example an agent approaching the intersection at high speed may either result in a collision or not, depending on the uncertainty around the position of target vehicles around the intersection). One the other hand, an agent's behaviour is a looser term and relates to the heuristic 'strategy' used by the agent to solve its environment (for example

an agent may navigate through an intersection by either being aggressive with high-speed trajectories, or with conservative low-speed trajectories). The policy learned in Figure 3.11 learns to avoid collisions by giving way to the target vehicle before crossing the intersection, obtaining the same (or very similar in terms of return) outcome as the optimal policy in the fully-observable case, although the heuristic behaviour is itself very different. We can easily think of a simple example of a maze containing two exits, whose paths for finding each exit is exactly the same length. In this case the behaviours of navigating to either side of the maze are very different, although the outcome of exiting the maze is exactly the same in both cases. In other words, policies that are close in return space may actually be very far in parameter space. This also makes sense since in some cases a single wrong action is liable to cause catastrophic failure such as a collision.

Due to the single dimensional nature of the return estimation, there is the possibility for confusion between which return modes correspond to which behaviours or outcomes. Figure 3.19 shows an illustrative example of this for the return estimation in our intersection set-up, where we can imagine a policy learning two different behaviours in the environment: one considered more 'risky' with a higher chance of collision but with a chance of navigating the intersection in less time, and another considered as being 'safer' with a lower chance of collision, but taking a longer amount of time to complete the navigation task. The returns are roughly scaled to correspond to the reward function defined in the intersection MDP so that a collision will provide a return around -1, faster episode times will provide returns around the -0.1 value, and slower episode times will provide return around the -0.2 value.



(a) (b)

Figure 3.19: (a) Return distributions from different agent behaviours. (b) Combined return distribution with both behaviours being equally likely.

We place ourselves in a hypothetical situation where the agent has to chose between the two strategies to crossing the intersection. In terms of expected value, the return distributions for both behaviours actually have the same mean

value of -0.46, so we wish to use the shape of the return estimation for a better decision strategy, for example the one which will carry the least amount of risk. The true return distribution for each strategy is shown in Figure 3.19a. If each behaviour has been explored with the same frequency during the training phase, then the estimated return distribution available to the agent from its initial state will be that represented in Figure 3.19b, combining the possible outcomes from all previously seen behaviours. We can see that there is some overlapping in the modes of the returns for each possible behaviour, notably at the value of -1 where the probability for collision is reflected in terms of the total number of times the agent has observed collisions for *both* 'safe' and 'risky' behaviours, without distinction. In essence, we cannot guarantee that the mapping from the space of policies to the space of returns be injective, and therefore we cannot use modes in the return distribution to correctly identify the different behaviours of a policy.

This leads us to look for an approach where we are able to make the distinction between uncertainty of outcome, and policy behaviour. Though the terms behaviour or strategy are relatively heuristic, we can image some ways to identify them in terms of sequences of states, sequences of actions, policy parameter space, or some other metric that is not dependant on the reward values obtained by the agent. This perspective is discussed further in the following chapter where we propose a more representative metric for measuring how close two policies may be in terms of behaviour or strategy.

Hierarchy of Performance Objectives

There are multiple approaches to multi-objective optimization, according to the desired output behaviour that is required from a controller. In our problem statement for autonomous vehicles for example, we must balance out safety and performance in order to obtain the best possible behaviour in each scenario. However optimizing one criteria will most likely detrimentally affect another, for example increasing performance (i.e. speed at which an ego navigates through the intersection) will come at the expense of a safer driving behaviour. As described for the intersection scenario we consider above, in RL these criteria are implicitly weighted in importance according to the amplitude of rewards that are attributed to reinforce each style of behaviour. Changing the reward function is changing one of the defining elements of the MDP, and so the resulting optimal policy will also be affected, meaning that logically a change in the ratio of amplitude in the reward terms will most likely lead to a change in behaviour of the resulting optimal policy that is learned in the MDP.

In our application though, we have a clear hierarchy between safety and performance criteria, the former being of course more important. We wish to model the fact that safety is always more important than performance, and so we give a large amplitude ratio between the two corresponding reward terms.

Although it might seem reasonable that the ratio expresses that in expectation an increase in trajectory time of 10 seconds is as good as a reduction of risk of collision of 1%, this remains arbitrary, and we could easily use the same heuristic justification for different values and relative weights of the reward terms. Moreover, the difference in terms of many order of magnitude poses a practical issue for a neural network learning to estimate the returns for a policy: the large difference in output amplitudes leads to large differences in amplitudes of gradients through the network, and hence each updates affects the network parameters differently. The learning rate is a parameter used to control the influence each update has on the network parameters, and is usually set according to the amplitude of gradients being back-propagated through the model along with the number of model parameters. Setting a single learning rate for training may then cause the network to not learn to correctly represent some of the returns with higher or lower orders of magnitudes, and this could explain, for example, the lack of representation of the collision returns in Figure 3.15. If we were to adjust the learning rate for rewards scaled to that of the collision reward, the effect of the time-step reward on the model may be negligible and this dynamic may not be captured by the model parameters.

These are issues that arise when aiming to combine multiple criteria into a single one. However the hierarchical nature of our task objectives may lead us to look for another approach with respect to how each are optimized, differentiating the criteria with an ad-hoc controller architecture rather than wieghting them within the same RL problem. Hierarchical reinforcement learning is an area of RL which deals with tasks of a hierarchical nature, and may provide some insight to approaching the problem of safety-performance balance in agent behaviour. We go into more detail on this subject in chapter 5, and investigate the use of a hierarchical architecture to separate the criteria of performance and safety to better tackle this issue.

Policy Forgetfulness

During the optimization of the RL agent's policy, we have seen (Figure 3.8) that it may encounter sub-optimal solutions to the MDP which are then eventually discarded in favor of better global optima. Through this approach we unfortunately run the risk of discarding strategies that may still be useful, simply because they are sub-optimal. We have seen this in our experiments where in cases where the agent was very confident during training (low uncertainty), a more conservative strategy would initially be explored and then forgotten in favor of the more aggressive driving style. Though these policies encountered during training are sub-optimal with respect to the current training paradigm, they may represent alternate behaviours for navigation through the environment which could be considered as useful to the agent. Given the issue of different behaviours having potentially overlapping return values, we can imagine a situ-

ation where two totally different strategies obtain very close values in terms of rewards in an MDP task (global and local optima of very similar values, but at different locations in policy parameter space). In this case it is might be possible for an RL algorithm to initially converge to either of the two solutions, but ultimately 'forget' about the slightly sub-optimal one due to random exploration designed to find the global optimal policy. Given that additional uncertainties and different environment scenarios affect the shape of the return landscape, it is reasonable to assume that local optima points may shift to being the global optimum, and vice-versa. If this is the case, then we might have used a large amount of training time simply to un-learn useful behaviours which would have increased the robustness of the agent with respect to uncertainties changing the shape of the return landscape.

3.2.4 Using Sub-Optimal Policies for Robustness

Given these lessons learned from our use of distributional RL, we seek to use a different approach to our problem statement. One area in which we noticed the agent compensate for environment uncertainty, was in the case illustrated in Figure 3.11, where the agent learned a different strategy from the fully observable environment (Figure 3.8) in order to avoid the zone of uncertainty altogether. In this use-case we saw the agent learn to adopt a more conservative driving style in order to avoid altogether an area of uncertainty in which there is a possibility for a collision with a target vehicle. Following this line of thought, we can imagine that some form of 'super agent' having access to both policies learned in a low-uncertainty and high-uncertainty environment might be robust to both cases, given a method of knowing what the level on uncertainty present in the environment is. This comes back to the power of generalization given to deep learning models, and although we can imagine cases where agent are robust to uncertainties present in their training data distribution, directly training an agent on multiple values of environment uncertainty gives an extra dimension to the navigation task which will greatly increase both the required size of the model along with the training time in order for the neural network parameters to implicitly model the adaptation in behaviour required from an agent in order to react to different levels of uncertainty as direct inputs to its controller model. Moreover, as we have seen in the application of a distributional framework, adding additional dimensions to the model in this way adds much difficulty and hyper-parameter tuning to the RL algorithm.

This leads us to imagining a multi-agent training set-up, where the complexity of training for multiple sources of environment uncertainty would be distributed across multiple agents instead of a single one, increasing the computation required for training but not affecting the model complexity needed to learn adequate behaviours for solving the MDP control task. Given this approach, each agent's policy can learn a separate 'behaviour' for solving the task,

each adapted to the relevant training conditions. In the next chapter, we build a multi-agent training framework with the objective of learning a set of policies which are collectively robust to expected (but unencountered) epistemic and aleatoric uncertainties.

Chapter 4

Learning Contingency Policies

4.1 Introduction

The need for having different strategies for a single input state, leads us to want to train multiple policies concurrently in an available training environment, with some constraints which reflect the need for robustness with respect to the uncertainty linked to environment uncertainties.

When learning to act in an uncertain environment, as humans, we intuitively attempt to build a 'plan B' for when our usual solution to the task at hand might not be valid. We build these alternative strategies by imagining a variation of environment dynamics where our current approach becomes invalid, and then attempting to find the best solution for that new case. The formation of multiple strategies for solving a task has the potential to greatly increase the robustness of an agent with respect to unanticipated changes to its environment. As a simple example, we can imagine a person planning what to wear when going out for a walk, according to the weather condition. If the weather has been sunny all week, then a naive approach would be to dress accordingly to this precedent information. However this leaves us vulnerable to the case where the weather may suddenly change and start to rain. Naturally, we would imagine a situation where we are able to plan for a dressing strategy to implement specifically in cases such as rain, where the attire deduced from the previous weather is unable to handle. This approach is called contingency planning [Pryor and Collins, 1996], where our aim is to develop a way to learn contingency solutions for cases where the expected optimal solution is unable to perform well in the current environment setup. This approach is equivalent to wanting

to increase the generalization capabilities for an agent, with respect to scenarios it may not have encountered during training or, if encountered, not retained the appropriate solution for that use-case. Further, when uncertainty is involved, we may wish to have multiple concurrent solution available according to the most probable outcomes of random variables such as decisions made by other road users, or presence of other vehicles obstructed from detection for some reason.

Designing controllers to have a high capacity for generalization is key to applying control algorithms when there is a high degree of uncertainty and combinatorial complexity in the environments in which we want autonomous agents to perform well. In this chapter, we design an algorithm for concurrently learning polices which are collectively able to act well enough over the whole space, having trained only on a tractable subset of environment configurations.

4.2 Related Work

The main element susceptible to engineering in MDPs is the reward function, which indirectly defines the agent's goal; learning about multiple goals can be translated into learning to solve MDPs with multiple reward functions. One of the most common uses for augmenting the reward function is to artificially boost the RL agent's degree of exploration [Badia et al., 2020b], [Eriksson and Dimitrakakis, 2020]. These methods dynamically change the value of the immediate rewards an agent gains, in order to encourage actions towards areas of state-space which are deemed more important. Our approach is similar to these in that we base an extra reward term on an external factor in order to affect the behaviour of exploring agents.

Although we base our approach on engineering the MDP's reward function, the problem we are tackling with our approach is distinct from the exploration problem in RL, and our objective is not to find an optimal exploration scheme. Exploration boosting methods are used within the context of a single environment, in order to avoid having the agent fall victim to being trapped within a local optima. Our goal however is to have the RL algorithm be able to retain sub-optimal solutions to the environment, once the training regime is ended. In a similar spirit to how TRPO Schulman et al. [2015] seeks to mitigate the concern that small changes in the parameter space may lead to sharp drops in performance, we seek to mitigate sharp drops in performance resulting from changes to stochastic environment parameters.

Moreover, the approach explored in this paper aims to learn policies with different behaviours, given the same initial environment conditions. Compared to methods which learn only the environment's optimal policy, this allows us to have sub-optimal policies to use as fallback strategies which we can switch through using a hierarchical structure. This has an advantage over attempting

to have a single policy learn to generalize over the space of MDPs, since this task (meta-learning) requires many more samples in order for an agent to achieve a good performance [Kirsch et al., 2019], our approach is more attractive for certain safety-critical applications.

Many previous works aim for agents to generalize to different goals within the same MDP [Schaul et al., 2015], or even self-discover sub-goals that make learning more efficient [Machado et al., 2017]. In these cases, a difference in goals is translated through a change in the reward function for a goal-state g: $R(s, a, g) = R_g > 0$. These methods take advantage of the underlying structure in the goal-space in order to increase sample efficiency [Andrychowicz et al., 2017], as well as the generalization capabilities of agents to tasks with goals that were not present during training [Eysenbach et al., 2019a]. They provide a good way of finding different behaviours within a same environment, where the modification of the reward function is intended for a control algorithm to be robust in terms of changing objectives. Our problem statement focuses on dealing with scenarios when the objective (i.e. goal) of an environment remains the same, but the environment is expected to change in some unknown way.

4.2.1 Variable Elements of MDPs

A vital component of a reinforcement learning algorithm is modelling the environment as a Markov decision process, which means defining some elements in order to both best represent the task being solved, and some further parameters affecting the reward attributed to the acting agent, and the discounting factor. The design of each MDP-defining element can be more or less arbitrary: most likely the transition dynamics $\mathcal{T}(s'|s, a)$ are given by the optimization problem and not susceptible to being changed if the environment itself does not change. The state and action spaces $\mathcal{S}$, $\mathcal{A}$ are subject to a little more design work because of the ability to define which information is the most relevant for the agent to react, and which actions are possible in response to those states [Dalal et al., 2018, Alshiekh et al., 2018]. However sometimes the state space must be augmented for example in the case of Atari games, it is common practice to include multiple frames of the game in the input space in order to give some information of direction of movement for objects on the screen instead of the static image. Action may additionally be constrained according to a set of pre-defined rules to constrain the agent's to not break a set of hard constraints during training, or in order to reduce the learning complexity of the problem if there are some states for which we already know the optimal actions to take. Furthermore, the type of action is subject to an engineering decision: we may want to give the agent the ability to take the lowest-level actions, or to simplify training give it access to a complex combination of low-level actions which we know give the agent sufficient 'action resolution' for solving the task at hand.

Changing only the value of the γ parameter for example, will affect the ratio of weights between long and short-term rewards on the expected sum of discounted rewards G_t, leading the agent to switch between preferring either short-term or long-term future rewards [Xu et al., 2018]. Though the value for this parameter is usually set to some value very close to 1 (usually 0.99 in most RL applications), modifying it will also change the return landscape and by extension the optimal policy resulted from training the agent.

In the related work section, we mentioned the use of modifying the reward function to obtain a better agent performance in terms of exploration, sample efficiency, and generalization capabilities. Modifying the reward function is by far the most common approach for modifying the behaviour of a reinforcement learning agent, since it is through this function that goals and incentives are described in the optimization problem. Nevertheless modifying any of the elements of the MDP will change the optimization problem that is being modeled and hence likely change the optimal policy π^* associated to that MDP.

4.3 Learning Contingency Policies

In this section, we motivate the use of a new RL paradigm aimed at learning policies which are robust to changes in the environment's stochastic parameters. We make the hypothesis that changes in environment dynamics affect its state-space locally, meaning that the best way for maximizing the agent's robustness is to find alternative paths to the objective, with a trajectory through state space that is sufficiently different from the other strategies such that a local change in environment stochastic parameters will not affect at least one of the strategies available to the autonomous agent.

We initially consider a model-free setting for learning, to make our approach as general as possible, not being affected by possible model biases of real-world applications. In a model-free approach however, efficient learning is notoriously hard, one of the factors being the increased variance of target updates from of the lack of a planner able to average out return values from multiple simulated runs. A lack of an environment model also reduces the ability for an agent to adapt to changes to the environment after training, since we are unable to use a planner to explore the new dynamics before acting. Works such as Hausknecht and Stone [2015] and Zhu et al. [2017] use deep recurrent Q-networks in order to build up knowledge of the MDP's state over time, and use a latent representation of the history of states and actions in order to inform an agent about its current state. In the case of stochastic parameter change, we are unable to predict how the returns of a policy will be affected. Kumar et al. [2020] seeks to learn a set of policies which are collectively robust to changes in environment dynamics, through the use of latent-conditioned policies [Eysenbach et al., 2019b]. Our approach is similar, though we use an explicit metric on trajectories to ensure

diversity in contingency behaviour rather than a learnt descriminator function. Furthurmore we train our contingency plan to perform well when environment parameters change during the execution phase, which couples well with an on-line high-level controller.

4.3.1 Agents' Behaviour

First of all, we wish to have some measure on the similarity in terms of behaviour from different agents in the same environment. There are multiple possible different interpretations for what defines similarities or differences in behaviour [Cool to have some reference on this point]. In the context of an agent acting in an MDP, we decide to use a metric on trajectories through state-space $\mathcal{S}$ to measure similarities in behaviour. It could be possible for example to use the action-space $\mathcal{A}$ or state-action space $\mathcal{S} \times \mathcal{A}$. Though there is a slight nuance between actions taken by an agent, and the states in which the agent finds itself as a consequence of its actions. Since our application also considers uncertainties in environment dynamics (i.e. results of actions), we stick to considering only the state-space trajectory. Later on in this section, we formally define a degree of behaviour similarity between agents in order to develop an algorithm for learning behaviours which are different in terms of expected trajectory through state-space. In practice, because of the partial observability of the environment, we only have access to observations of the states encountered rather than the states themselves. We denote τ_π the trajectory through observation-space for an agent:

$$\tau_\pi = \{o_t\}|_{t\in[0,T]}$$
$$s_0 \sim p(s_0), a_t = \pi(o_t), s_{t+1} \sim \mathcal{T}(s_{t+1}|s_t, a_t), o_t \sim O(s_t)$$

However a difference in behaviour is not a sufficient condition for learning a policy with that behaviour: we require learned behaviours to be useful in the environment, which is a condition which also may vary according to the task at hand. To this end, we define a property for when an agent's behaviour may be considered a valid strategy or not, depending on whether it sufficiently solves the task modelled as an MDP.

Learning Valid Strategies

Definition. (Valid Strategy) An agent's policy π is considered as having a valid strategy in its respective MDP, if either of the following conditions are met, depending on the nature of the control task:

(A_1) Agent is able to reach a specific goal-state $g \in \mathcal{S}$.

(A_2) Agent is able to expect accumulated discounted rewards above a threshold score $V^{\pi}(s_0) > G_{min} \in \mathbb{R}$, for s_0 sampled from initial state distribution (Note that V^{π} is the true value function as in (2.3), not an approximation).

The use of either condition for what is considered a valid strategy depends on the task, and what it heuristically means to solve it. For example, in the case of an Atari game (e.g. Breakout), any behaviour from a policy which reaches above a threshold score, is typically considered to be a valid strategy. Another example would be an AV passing through an intersection, where any behaviour passing the intersection (reaching goal-state g) without collisions is a valid strategy.

The nature of completing a task will more often than not affect the shape the reward function will take: most tasks where the end objective is for the agent to find itself in a goal-state g, will have the MDP set-up so as to perceive a large reward at the end of the task once the desired goal-state is reached. This kind of MDP is described as having a sparse reward. Although the task objective in this case is best described by the sparse reward, this sparsity in terms of training signal can be a big hindrance for an agent's learning speed. For example in a game of Go where the average number of moves is 211, with an intial action space of 361 moves, the link between which moves were the correct ones and which ones are bad can be difficult to determine. This is why having a dense reward function helps algorithm convergence speed, at the cost of the bias introduced by the human design of additional reward terms. Dense rewards lend themselves more naturally towards ongoing tasks, where the definition for a *valid strategy* is closer to the second point in the aforementioned definition. For example, collecting points in a video game, or a robot learning to balance items in a certain way, are both examples of ongoing tasks where the reward function can naturally be modelled as being dense.

In our application to an autonomous navigation task, we primarily use the the former definition of aiming to reach a desired goal-state g, since it better represents the objective of an autonomous navigation task which can be modelled as correctly navigating through various sequential road 'modules', and the ego vehicle's objective is to make it to the end of the module while avoiding any possible collisions. Hence this interpretation is more useful as it pertains to our use-case. However, when modelling the navigation task we also use dense rewards to represent the performance objective for the agent in the form of a small negative term at every time-step. This is one of the differences in objective types between safety and performance which we can exploit to model differently in the MDP. This way, we give the agent the opportunity to more freely make sacrifices in performance in the interest of safer behaviour. This is the idea behind implementing a contingency plan for the agent, where we expect to be sacrificing some performance in order to obtain a better behaviour from the point of view of the safety criterion. Following this, we introduce the definition for a *sub-optimal policy* representing a policy which performs a little

worse in terms of gathered reward than the optimal, but is still a valid strategy according to either part of the previous definition.

Definition. (Sub-optimal policy) A sub-optimal policy, denoted π_{sub}, is a policy whose expectation of return is within a margin ϵ, to that of the optimal policy π^*, at some given initial state s_0 sampled from the initial state distribution:

$$V^{\pi^*}(s_0) - V^{\pi_{\text{sub}}}(s_0) < \epsilon. \tag{4.1}$$

Condition (4.1) is equivalent to saying that policy π_{sub} is a sub-optimal policy, whose behaviour is a valid strategy (in the threshold-score sense considered in constraint (A_2), $\epsilon = V^{\pi^*}(s_0) - G_{min}$). For example in the case of an Atari game, any agent that achieves a score higher than the threshold, but less than the one obtained by the optimal policy π^*, verifies condition (4.1). This definition is a little looser when considering constraint (A_1), where the value for ϵ may even be very large, as long as the goal-state g is reached. For example, in the case of an autonomous navigation task, if an agent manages to reach the goal-state g the task in a higher amount of steps than the optimal policy, we consider it a sub-optimal policy. It is possible to specify a certain available performance budget in terms of ϵ , to determine how much of the performance we are willing to sacrifice in search of alternate solutions before we consider those solutions non-valid as strategies.

Measuring Different Behaviour in Agents

In order for the two policies to be considered as having different behaviours, they must be sufficiently different in the state-distributions that are encountered during execution. This implies the need for a metric $\mathcal{M}$ measuring the difference between agents' trajectories in state-space. There is no standard metric for measuring the similarities for agent trajectories. The difficulty lies in the potential high dimensionality of state-space, along with the variable trajectory lengths. high dimensionality may be an issue due to the potential difference in importance of various state features when determining the similarity of trajectories. The different impact of various state features is something that can be determined in a case-by-case basis for each implementation. The main difference however is the variable size of the trajectory vector. To be able to compare trajectories of different lengths, the most straightforward approach is to compare the density of states encountered. This works well, however we lose the sequential nature of agent trajectories and similar states encountered at entirely different times during the episode will have the same representation using state densities. Another approach uses a descriminator function [Eysenbach et al., 2019b, Kumar et al., 2020] in order to force states to correspond to identifiably different policies, though this has the same issues ignoring the sequential nature

of state trajectories.

Definition. (Sufficiently different behaviours) We can say that two policies, π_1, π_2 have $\mathcal{M}_d$-different behaviours, iff:

$$\mathcal{M}\left(\mathbb{E}\left[\tau_{\pi_1}\right], \mathbb{E}\left[\tau_{\pi_2}\right]\right) \geq d. \tag{4.2}$$

Condition (4.2) is equivalent to saying that the behaviours of π_1 and π_2 can be described as being heuristically different. In our approach, we base this heuristic on the similarity in terms of their respective trajectories through the MDP state-space. Setting a value $d \in \mathbb{R}$ is subjective: for instance, human experts may have arbitrary boundaries for when an agent's path through state-space is sufficiently different from a reference path, to consider both as having different behaviours. Condition (4.2) can be thought of as a non-parametric clustering with boundary d, where $\mathcal{M}(\cdot, \mathbb{E}[\tau_{\pi_{ref}}])$ is the feature map in a policy's state-trajectory space, with respect to a reference policy π_{ref}. Once more, the correct segmentation of this space is subjective and may vary between experts based on experience.

To recap, we propose that in order to increase the robustness of an RL algorithm to changes in environment parameters, it should be able to learn sub-optimal contingency strategies which have sufficiently different behaviours from the optimal policy in a training environment, satisfying both conditions (4.1) and (4.2). Changes in $\mathcal{T}$ or $\mathcal{O}$ may affect local areas of the environment's state-space differently, affecting some policies' expected returns more than others, depending on whether the introduced uncertainty affects their respective state-space paths. Condition (4.1) ensures that we only learn policies with satisfactory performance in the environment, whereas (4.2) aims to maximize the likelihood that at least one of the learned policies will have an expected return that is minimally affected by changes to either $\mathcal{T}$ or $\mathcal{O}$.

Figure 4.1 gives some insight on how an agent having access to policies with different policy behaviours has the potential of increasing the robustness of the overall algorithm when the uncertainty is local, and able to be avoided through intelligent trajectory planning. We can imagine the collection of all possible agent trajectories from starting to final state, and that in the presence of local uncertainty there should be at least one of these possible alternate valid trajectories which will the be the least affected by modifications to environment models.

4.3.2 Reward Augmentation

We wish to learn, on one hand, π^*, the optimal policy in our given environment, and on the other, π_1, a contingency policy able to navigate more safely through the environment, at the cost of performance, if ever there is high uncertainty

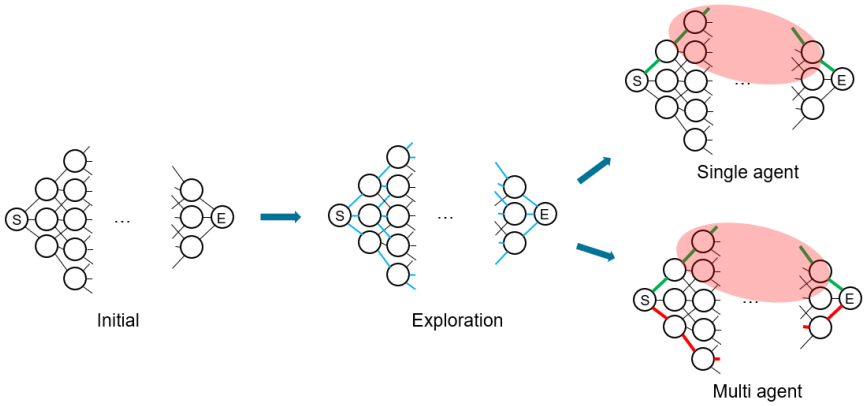


Figure 4.1: Illustration of usefulness of a contingency plan to avoid unforeseen uncertainty. 'S' indicates the starting state, 'E' indicated the end or goal state

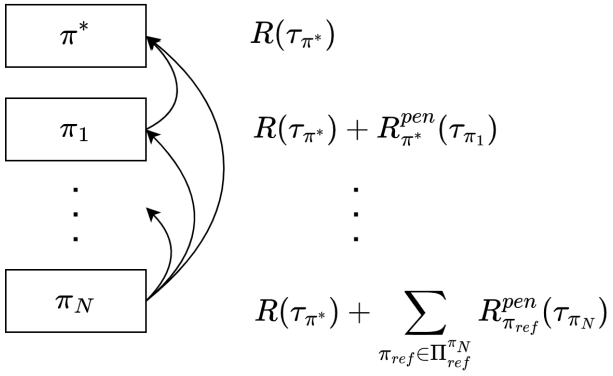


Figure 4.2: Reward dependence for multiple contingency agents.

linked to following the optimal policy. We wish this contingency plan (or multiple instances thereof) to be both a valid strategy, and sufficiently different in behaviour to the optimal, following both (4.1) and (4.2) respectfully.

Our approach is to introduce additional agents in modified versions of the MDP, in order for those agents to converge to different policies, having different behaviours. In section 4.2.1 we discussed the possible modifications to an environment modeled as an MDP, and the effects these modifications can have on the convergence point of the agents optimizing their respective behaviours. Due to the desired property of wanting policies with explicitly different trajectories, we chose to introduce an additional dense reward term to encourage policies to take trajectories that are different from that of a reference agent (the optimal

agent in our case). We do this by introducing a reward penalty, $R^{pen}_{\pi_{ref}}(\tau_\pi) \leq 0$, as a function of the trajectory of the current policy π, as compared to that of a reference policy π_{ref}. The trajectory penalty is based on the metric $\mathcal{M}$ between agents' state-space trajectories in order to satisfy (4.2). We use the following expression for the trajectory penalty to discourage agents observing this term to be too close to others' expected path through state-space $\mathbb{E}\left[\tau_{\pi_{ref}}\right]$:

$$R^{pen}_{\pi_{ref}}(\tau_\pi) = -\frac{\alpha}{\mathcal{M}\left(\tau_\pi, \mathbb{E}\left[\tau_{\pi_{ref}}\right]\right) + \delta}, \tag{4.3}$$

where $0 < \delta < 1$ avoids infinite penalties for exactly following $\mathbb{E}\left[\tau_{\pi_{ref}}\right]$, giving the penalty an upper bound of $-\frac{\alpha}{\delta}$. π_{ref} is the reference policy, which may or may not be the optimal, according to the number of agents with unique behaviours we wish to train. α is a scaling factor to adjust the amplitude of the penalty term, compared to the regular rewards. Contingency policies will be training concurrently to the optimal one, aiming to converge to distinct valid strategies within the same environment.

General Approach for Multiple Contingency Agents

Theoretically this approach could be used for an arbitrary number of additional contingency agents, as illustrated in Figure 4.2, where the soft constraint given by trajectory penalties for trajectory similarities between agents can be implemented for any pairs of agents meaning each subsequent agent can be given an incentive to be different from the previous existing agents, following (4.2). Although this adds complexity to the RL problem, the number N of total agents should remain reasonably limited: we should increase N according to the anticipated uncertainty on the environment parameters. If we denot the set of all agents in the training environment as Π, then $\pi_{ref} \in \Pi^{\pi}_{ref}$ represents all previous agents (shown by the arrows in Fig. 4.2) whose trajectories are used as references to compute the trajectory penalty term:

$$\Pi^{\pi_n}_{ref} = \Pi \setminus \{\pi_m | m \geq n, m \in [0, N]\}$$

As an example, $\Pi^{\pi^*}_{ref}$ is empty in the case of the optimal agent, $\Pi^{\pi_1}_{ref} = \{\pi^*\}$ for the 1^{st} contingency agent, $\Pi^{\pi_2}_{ref} = \{\pi^*, \pi_1\}$ for the 2^{nd} contingency agent, and so on. For N contingency agents, this approach adds a computational cost of $o(N^2)$ in terms of trajectory penalty computation. Increasing state-space size and dimensionality is susceptible to increase the number N of concurrent agents we wish to maintain as there are more opportunities for alternate valid strategies. However, N is limited by the number of expected changes in the MDP's state-space we wish the RL agent be robust to, hence the computational cost is expected to remain within the same order of magnitude as without the pseudo-reward implementation.

Implementing the additional trajectory penalties will impact the new value-function estimate of agents learning sub-optimal policies. So (4.1) should become:

$$V^{\pi^*}(s_0) - V^{\pi_{sub}}(s_0) < \epsilon + \sum_{\pi_{ref} \in \Pi^{\pi_{sub}}{}_{ref}} R^{pen}_{\pi_{ref}}\left(\tau_{\pi_{sub}}\right), \tag{4.4}$$

such that π_{sub} would still be considered a valid sub-optimal policy.

Trajectory Metric

We have put forward the need for an explicit metric between agent trajectories, with the following properties:

- We must be able to compute differences between trajectories of different lengths.
- This metric must represent the heuristic differences in behaviour between agents generating those trajectories. This includes aspects of a trajectory such as sequentiality of states (being consistent with describing loops for example) or not being too sensitive to difference between states, since some trajectories with all different states may have the same heuristic behaviour.

To the first point, we can simply use a density function to represent the distribution of states included in the trajectory vector. This approach is useful since computing difference in densities is relatively straightforward, though this will eliminate the sequentiality of states which would be ideal to model. Using densities has the disadvantage that the positions of encountered states in the trajectory vector are lost. For example the same trajectory but travelled in the opposite direction will have the same state densities. Additionally, as mentioned above, any loops in the state trajectory will not be represented: a trajectory will have the same density as another which loops 100 times over it. This is in part due to the fact that we wish to compare trajectories of different lengths, so we eliminate the information of length from those trajectories, although this clearly could be some important information to have when describing differences in heuristic behaviours between agent trajectories.

To increase the relation to heuristic differences in behaviour for the metric, we can use a feature map on states to not include the entire raw state, and seek to use the most informative features to describe similarities in agent behaviour. This is heavily correlated to the nature of the environment state-space, and subject to change according to the environment with which the agent interacts. Therefore we allow for additional design – and hence also additional bias – in

the information from states that is used to determine trajectory similarities. In our experiments we mostly use a mask function to only select certain dimensions of the state which we find best describe the behaviour of the autonomous agent. For example in the case of an intersection where the path is pre-determined we can use the speed of the ego vehicle to differentiate between age behaviours, disregarding all other elements of the state such as those relating to target vehicles which do not provide much information about the ego behaviour. Therefore we implement the following expression for what we refer to as the *trajectory metric* $\mathcal{M}$:

$$\mathcal{M}(\tau_\pi, \tau_{\pi_{ref}}) = \int \left| \nu(\phi(\tau_\pi)) - \nu(\phi(\tau_{\pi_{ref}})) \right| d\phi(o) \tag{4.5}$$

Where ϕ is an feature map over state observations, and ν is the density function. We integrate the difference in densities over the observation feature space over which the densities are computed. It is possible to make this approach as general as possible with $\phi(o) = o$, however with some domain knowledge we can modify ϕ to retain only the most informative features.

4.3.3 Algorithm Description

Algorithm 3 gives a pseudo-code description of our implementation for learning contingency policies alongside the optimal policy. $\pi_{ref} \in \Pi^{\pi}_{ref}$ represents the same set of reference agents as in Fig. 4.2.

Both agents train concurrently, alternating training episodes. However the contingency agent π_1 starts its training only when the replay buffer of π^* is full. This is reflected in the training curves, though it is negligible with respect to the order of magnitude of total training samples. Computing the value for $R^{pen}_{\pi^*}(\tau_{\pi_1})$ before π^*'s expected trajectory memory buffer has converged to a stable value means that the MDP being solved by π_1 is initially greatly changing, though we did not do a detailed investigation of the effects of different training scheduling for the contingency agent.

We must also decide to which state transitions samples (training samples for the RL agent) the trajectory penalty is attributed to. In this case there are two possible approaches: either the R^{pen} term can be computed at the end of the agent's trajectory hence considering the different in state distributions for both entire trajectories, or R^{pen} can be computed on a state-by-state basis where we input single states into $\mathcal{M}$ to compute how similar a single state is to the expected distribution of states of the reference agent. The latter approach could be seen as more desirable since it is a denser reward function, however we can notive that there will be a strong negative bias towards longer trajectories which will have the tendency to accumulate a lot of negative reward terms if

Algorithm 3 Training Contingency Policies

Init $\Pi = \{\pi^*, \pi_1, ..., \pi_N\}$
while not converged **do**
  for $\pi \in \Pi$ **do**
    $s_0 \sim p^\pi(s_0)$ ▷ init. episode state
    $o_0 \sim O(s_0)$
    $\tau_\pi = \{o_0\}$
    while episode not terminated **do** ▷ play episode
      $a_t = \pi(o_t)$
      $s_{t+1} \sim \mathcal{T}(s_t, a_t)$ ▷ environment step
      $r_t = R(s_t, a_t, s_{t+1})$ ▷ step reward
      $o_{t+1} \sim O(s_{t+1})$
      $\tau_\pi = \tau_\pi \cup \{o_{t+1}\}$
    end while
    $r^{pen} = \sum_{\pi_{ref} \in \Pi^\pi_{ref}} R^{pen}_{\pi_{ref}}(\tau_\pi)$ ▷ compute trajectory penalty
    $r_T = r_T + r^{pen}$ ▷ attribute r^{pen} (only to contingency agent)
    for $t \in [0, T]$ **do**
      Memory$(\pi) \leftarrow (o_t, a_t, r_t, o_{t+1})$ ▷ store samples in replay buffer
    end for
  end for
end while

they contain many more states than shorter trajectories. Because of this we prefer to consider the vector of all states in the trajectory for computing the similarity metric $\mathcal{M}$, and we compute this value at the very end of the episode. There are additional approaches known as eligibility traces [Sutton and Barto, 2018] which allow us to reduce the sparsity of a reward signal using a form of reward attribution with exponential decay according to how far the states are to the final state of the trajectory. However in our approach, we find that this does not make a noticeable difference on the learning rate of agents perceiving the additional trajectory penalty. Due to this, we only attribute the R^{pen} term to the final episode transition that is then stored in the agent's replay buffer. Since we do not attribute any reward augmentation to the optimal agent, we simply define that $R^{pen}_{\pi^*}(\tau_{\pi^*}) = 0$ for a simpler notation, although inputting these values into the expression for the trajectory penalty gives some non-zero value, as is investigated in the results section of this chapter.

Training an Off-Policy Deep Q Network

The off-policy Q-learning algorithm [Mnih et al., 2015] is well-suited to learning a policy's Q-function in the discrete action space which we consider. Training off-policy allows us to train on past transitions stored in a replay buffer [Schaul et al., 2016] which both stabilizes training, and increases sample efficiency since

a single transition sample may be used during multiple training steps. We perform a stochastic gradient descent on the MSE between the Q-estimate for the current step, and the discounted 1-step 'lookahead' Q-estimate summed with the transition reward:

$$\mathcal{L}(o_t, a_t, r_t, o_{t+1}) = \left(Q_\theta(o_t, a_t) - \left[r_t + \gamma Q_{\theta'}(o_{t+1}, \arg\max_{a'} Q_\theta(o_{t+1}, a')\right]\right)^2,$$

where θ are the current parameters for the Q-network, and θ' are parameters that are 'frozen' for a certain amount of steps. This is known as double Q-learning using target networks [Van Hasselt et al., 2016] and reduces the variance of gradient updates.

Sampling Trajectory Expectation from Replay Buffer

Since we're using the similarity in trajectories as a training signal for the contingency agent, we wish this signal to be as stable as possible, so that the solution to the MDP being solved by the contingency policy doesn't move around to much. Due to the fact that the trajectory penalty is based on the behaviour of the optimal policy, a change in this reference behaviour will lead to a change in the training signal given to the contingency agent and hence change the return landscape with which it tries to optimize its behaviour. If the reference trajectory changes often and with high amplitude, then the training signal provided by R^{pen} will not enable the contingency agent to converge to a policy with different behaviour. In order to increase the stability of the signal, instead of using the instantaneous trajectory from the reference agent $\tau_{\pi_{ref}}$, we can average out its trajectories from the last K episodes $\mathbb{E}[\tau_{\pi_{ref}}]$. If we make the hypothesis that the reference agent will end up converging to a single policy, then the average trajectory should also converge to a single state distribution (as would the instantaneous trajectory $\tau_{\pi_{ref}}$), but with a smoother trajectory penalty signal. Hence we replace the equation given by (4.5) with the following:

$$\mathcal{M}(\tau_{\pi_1}, \mathbb{E}\left[\tau_{\pi^*}\right]) = \int \left|\nu\left(\phi\left(\tau_{\pi_1}\right)\right) - \nu\left(\phi\left(\mathbb{E}\left[\tau_{\pi^*}\right]\right)\right)\right| d\phi(o), \tag{4.6}$$

One nice way of getting the expectation over the density of states in an agent's trajectory, is to consider the states in its memory replay buffer. Due to the fact that our implementation for $\mathcal{M}$ has no temporal information on the states in the trajectory, we can simply sample states from the reference agent's replay buffer to represent the expectation over states encountered over past episodes. This approach is also consistent with having a smooth training signal, as the replay buffer becomes increasingly self-similar when an agent starts to converge to the optimal policy in an environment, and hence the state density distribution should become almost static at this point, giving a fixed objective for the contingency agent. So when computing $\mathcal{M}(\tau_{\pi_1}, \mathbb{E}[\tau_{\pi^*}])$, in practice we

replace the expectation operator by the mean value over the last K samples of the agent's memory as illustrated in Figure 4.3.

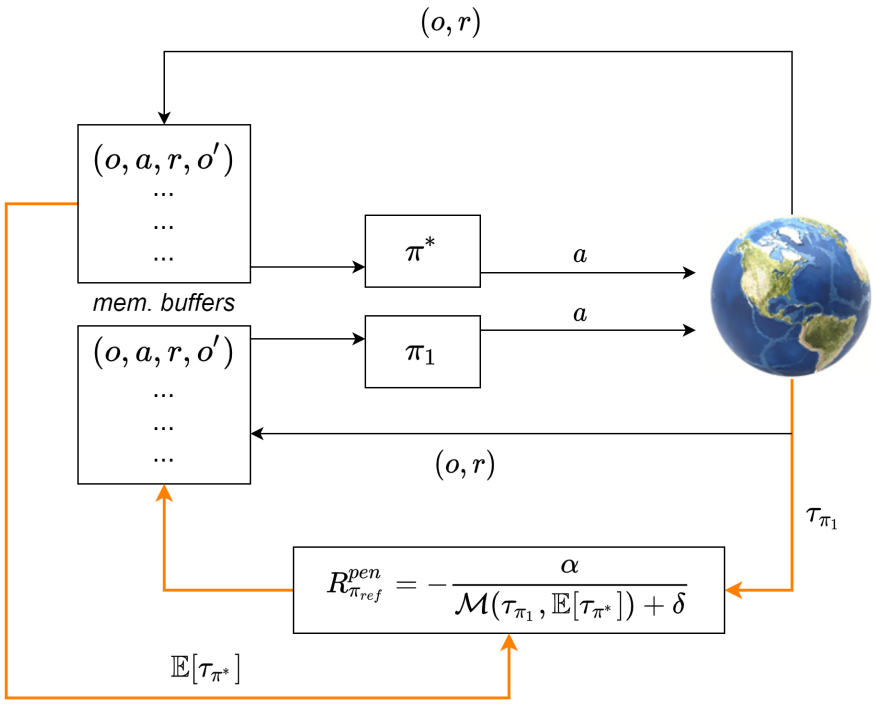


Figure 4.3: Illustration of RL training loop with a single contingency agent, π_1. Arrows shown in orange are part of the computation for the trajectory penalty term. Inputs for the trajectory metric $\mathcal{M}$ are taken from both the instantaneous trajectory for π_1 and the expected trajectory for the optimal agent, $\mathbb{E}[\tau_{\pi^*}]$

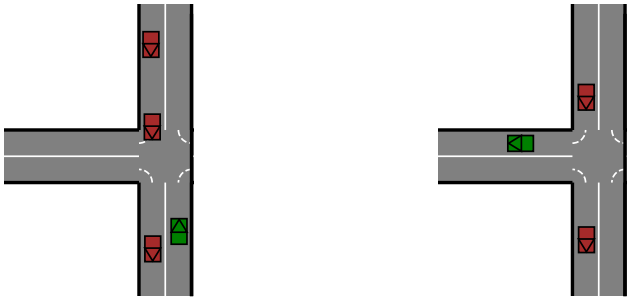
4.4 Experiments

4.4.1 Intersection Environment

Environment Description

In our experiments we use the same simulation environment as for the previous results. Figure 4.4 shows two frames of the environment with oncoming vehicles in the intersection. The ego must still perform a left-hand turn in the face of oncoming traffic, and must adjust its speed in order to pass through in the shortest possible time while avoiding collisions with target vehicles. We expect a

contingency policy to be useful in this scenario, when there is a sudden change in estimated target behaviour by the planner and the ego will have to either slow down to let an aggressive target through, or speed up if it seems the target is slowing down too much which may also cause a collision. More details on this environment are provided in appendix A. We can anticipate that a change in scene detection may affect the variance in detected positions of the target vehicles, and cause less conservative behaviours to be too risky. In this case, learning a fallback strategy that may be less affected by a drop in target position confidence, may be considered safer and more useful.



(a) $t = 3$s (b) $t = 5$s

Figure 4.4: Navigation task: ego makes a left turn across the intersection with oncoming traffic. Target vehicles may either be aggressive (i.e. disregarding presence of ego in intersection) or cooperative (i.e. slowing down if ego is close to intersection point).

Training Details

The state and action-space of the MDP are the same as described in the previous chapter:

$$s = \{s_{\text{ego}}, \dot{s}_{\text{ego}}, s_1, \dot{s}_1, \text{ttc}_1, s_2, \dot{s}_2, \text{ttc}_2, s_3, \dot{s}_3, \text{ttc}_3\}$$
$$\mathcal{A} = \{-4, 2, -1, 0, 1, 2\}\ m/s^2$$

To penalize collisions and encourage faster episode termination, the reward is set-up as follows per time step t:

$$r_t = \begin{cases} -5 & \text{if collision} \\ -0.1 & \text{otherwise} \end{cases}. \tag{4.7}$$

In this version of the reward function we have eliminated the penalty for hard-barking (i.e. $-4m/s^2$ action). This is in part due to the small influence we found it to have on the agent's behaviour on one hand, but also since it represents an objective of comfortable driving, which is not actually an objective we wish to

optimize for in the current state of our problem statement. Introducing a third objective on top of efficiency of task completion and safety, might unnecessarily complexify the behaviours which we wish to learn, and detrimentally affect the quality of behaviours robust to incoming sources of uncertainty.

The scaling factors for the trajectory penalty are:

$$\alpha = 1, \quad \delta = 0.1 \ .$$

These determine the relative weight of the pseudo-reward, with respect to the regular reward function R. A lower value for α will hardly penalize the pseudo-agent for having a similar state distribution to the reference agent, whereas higher weighting will make the pseudo-agent seek to have a highly different state-space trajectory, disregarding the original objective of the task given by the regular reward function. They are fixed by a rough initial sweep. Figure 4.7 shows the effect of sweeping values of α on the final performance of the contingency plan in terms of trajectory similarity to the optimal policy.

For the feature map ϕ used for computing $\mathcal{M}$, we use a function selecting only the ego speed out of the observation vector $\phi(s) = \dot{s}_{ego}$. In this application where the ego's control only applies to the longitudinal acceleration values due to the path already being planned out as a left turn across the intersection, we found that the speed feature alone is enough to differentiate between different ego trajectories.

4.5 Results

Fig. 4.5 shows the training scores for both agents. We clearly see the second agent's convergence to its optimal performance 'lags' behind that of the optimal agent. This is most likely due to the fact that the pseudo-reward term $R^{pen}_{\pi^*}$ depends on the states present in the memory buffer for π^*.

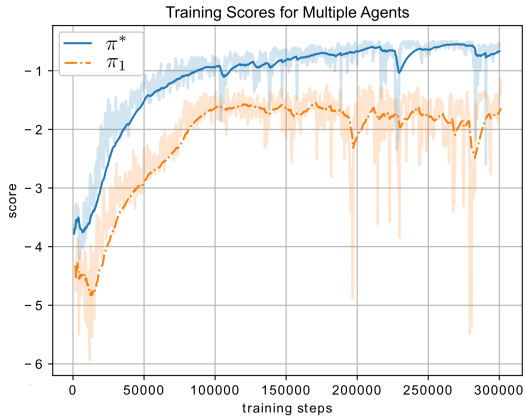


Figure 4.5: Training scores for both agents. Each is trained for 300k steps. π^* is the optimal agent, π_1 is the pseudo-agent.

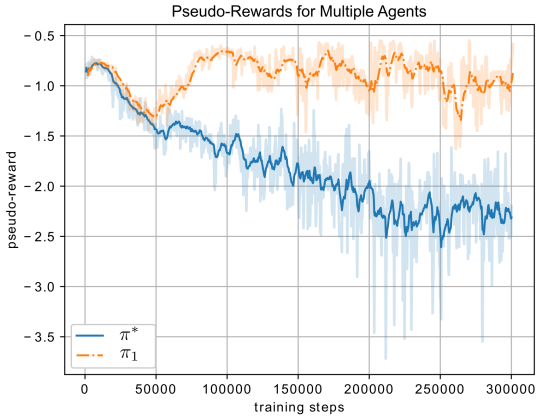


Figure 4.6: Pseudo-reward, $R^{pen}_{\pi^*}$ calculated for both agents. The values for π^* are computed only for comparison to the values used by π_1.

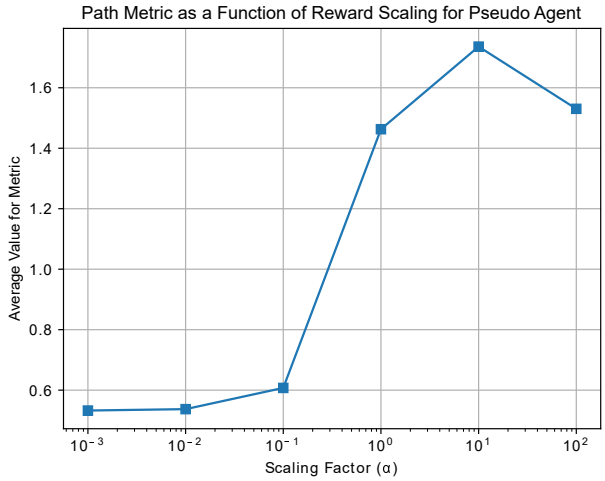
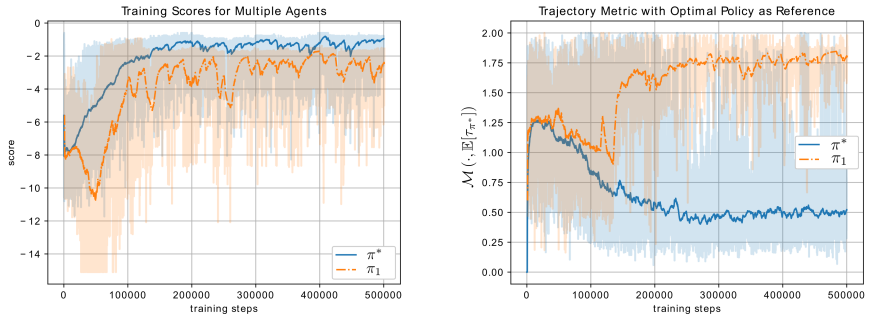


Figure 4.7: Average values for $\mathcal{M}$ on the final 100 episodes of each pseudo-agent, for different pseudo-reward scaling α.

Hence $R^{pen}_{\pi^*}$ cannot be stable until π^* has converged, and there is little change in its memory's state distribution. This prevents the corresponding pseudo-agent, π_1, from converging earlier. Interestingly, π_1 reaches its best score before π^*: in our implementation, the optimal path (accelerating before the 1st target vehicle) is harder to find through exploration than the sub-optimal one (passing in-between the target vehicles). Once the pseudo-reward is stable enough to dissuade the pseudo-agent from copying the optimal agent's path, it is faster to converge to its new optimal policy (being the original sub-optimal policy).

Looking at Fig. 4.6, we see that both agents' path metrics are similar for approximately the first 50k steps, after which their policies start diverging. This means both had similar state distributions (mainly due to a high degree of random exploration) until that point. The value of $R^{pen}_{\pi^*}(\tau_{\pi^*})$ keeps decreasing while the optimal agent is converging to its best policy, and levels out once it converges to its peak performance at approximately 200k steps. $R^{pen}_{\pi^*}(\tau_{\pi_1})$ however levels out quite soon, closely corresponding to π_1 reaching its peak performance. Though it is still changing due to the changing state distribution in π^*'s memory buffer, this is hardly seen on the plotted values compared to the random oscillations.

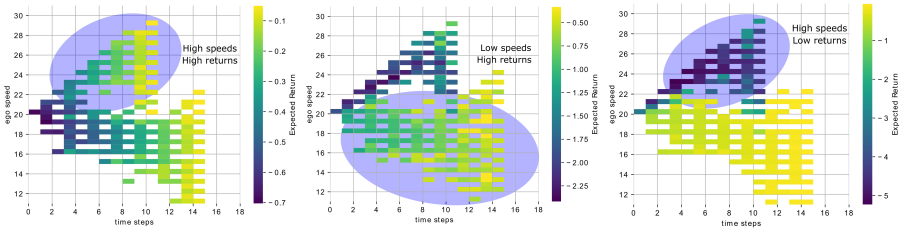
Fig. 4.7 shows the effect that modifying the parameter α has on the resulting policy learned by the pseudo-agent π_1. As mentioned in section 4.3, the pseudo-reward must be scaled in such a way to fulfill both conditions (4.2) and (4.4). Learning with pseudo-agents can fail if it is not scaled properly. We see that there is a critical value for α, after which the pseudo-agent switches to a sufficiently different behaviour, according to condition (4.2). In this case, we



(a) (b)

Figure 4.8: (a) Training scores for both agents during training phase. This figure does not take into account the trajectory penalties R^{pen} for π_1, only the regular rewards R. (b) Evolution of computed trajectory metric term $\mathcal{M}(\cdot, \mathbb{E}[\tau_{\pi^*}])$ for optimal and contingency policies. Although computed for both policies, the resulting trajectory penalty is only attributed to π_1.

can deduce that any value for d in the approximate interval $[0.8, 1.4]$ is suitable. Values $d < 0.8$ will not steer the pseudo-agent towards a trajectory different to the optimal agent, whereas $d > 1.4$ would falsely rule out policies which we can consider as being heuristically different.



(a) (b) (c)

Figure 4.9: Q-functions evaluated at different areas of feature space: (a) Q^{π^*} unaffected by the pseudo-reward, favors higher-speed trajectories. (b) Q^{π_1} using pseudo-rewards ($\alpha = 1$) favors lower-speed trajectories. (c) Q^{π^*} with increased uncertainty on target position, higher-speed trajectories result in a collision with first target vehicle.

Figs. 4.9a and 4.9b show the ground truth for Q^{π^*} and Q^{π_1} respectively, in the ego trajectory feature space, represented as a 2D-tuple of ego vehicle speed, along with the corresponding time step of the episode $(t, \dot{s}_{ego})$. In our use-case, this representation is sufficient to see the difference between varying ego behaviours. We can see in Fig. 4.9a that with the unmodified reward, the optimal agent π^* prefers trajectories having higher speeds, as they correspond

to a shorter episode duration which is optimal in the sense of the original reward structure. In Fig. 4.9b, adding an extra pseudo-reward changes the optimization landscape, and tends to steer the pseudo-agent towards areas of lower ego speeds. In all figures, we sampled 10 trajectories from different instances of both π^* and π_1, and plotted the mean Q-value for each $(t, \dot{s}_{ego})$ pair.

Fig. 4.9c shows the change in the expected returns in the case where there is an increase in uncertainty around the first target vehicle's position. In our experiments, we modelled a local increase in sensor uncertainty by increasing the effective collision radius of the first target vehicle by 50%. This modification leads to a sharp drop in the performance of π^*, whereas the state-subspace exploited by π_1 remains safe and unaffected. We can see that the new optimal policy in the case of Fig. 4.9c, is also reflected in Fig. 4.9b after adding the pseudo-reward term. This will allow us to use π_1 as a valid fallback strategy during execution, if ever there is a change in the environment that would not have been accounted for during the initial training phase.

4.6 Conclusion

In this chapter, we have introduced a new objective in an RL learning pipeline: keeping track of, and learning, sub-optimal policies encountered during the initial training phase. We have shown that through an intuitive modification of the reward model, that we are able to consistently learn these sub-optimal policies in the case of a driving scenario.

The context of this approach is intended for methods to be applied to model-free problem statements. In the case where the model, even a partial model, or estimation thereof is available to the agent, we gain access to more powerful and data-efficient methods for dealing with introduction of local uncertainties to the MDP. In the next chapter, we combine this training paradigm with a hierarchical controller, so as to quickly switch from optimal to available contingency policies in the case of unexpected environment change during the execution phase. This will allow an autonomous vehicle agent to make use of its fallback strategies learned during training, according to its perception of the environment, much like a human would.

Chapter 5

Low-Level Policy Scheduling with Model-Based Planning

5.1 Introduction

Planning, in the context of control tasks, is an incredibly useful tool when dealing with environment in which there is potential high degree of uncertainty. Although the act of planning for real-world applications requires a model which invariably introduces some form of bias in to the predicted behaviours, in the case of tasks where there is a strong constraint on the state-space an agent can encounter, such as any task with a high concern for safety, planning allows us to predict future actions and trajectories for the agent, and seek to mitigate any future failures before the agent is faced with them. Training an agent with a reinforcement learning algorithm is a form of planning: we use a model of the environment to train the a value function to predict the return values according to the reward function, over the course of many simulated agent-environment interactions. This is akin to an outcome estimation, though the value function also considers values of intermediate states and goals within the same return estimation.

More often than not, we tend to use models when training RL agents. This is due to the high number of agent-environment interactions required to learn an effective policy, which disqualifies many real-world training applications. Due to this fact we usually seek to model the task environment so that we're able to simulate many agent-environment interactions. When this is the case we

can use the model to perform some planning operations to increase the quality of our estimation of the agent's performance past the learned value function, during either training or execution. Our problem statement seeks to deal with the safety concerns of an autonomous navigation task during execution, which is closely linked to the stochastic nature of the environment. Implementing a planning module during the execution phase of the algorithm allows us to better predict the possible outcomes of agent-environment interactions over long periods of time, as well as compensate for the bias of the agent's return estimator inherited from a potential bias in the training data distribution. Bias inherited from the training data is common in machine learning applications, and in RL applications this can come from limitations in the environment model when compared to the true world dynamics, or even environment parameters that are different from those seen during training, once the agent is in the execution phase.

Combining planning and learning has lead to some of the strongest approaches in reinforcement learning applications, such as the AlphaGo model [Silver et al., 2018] developed to have better-than-human performances on the game of Go which has traditionally been difficult for non-human players due to the size of the state-space and the combinatorial complexity of various strategies that can extend over a long chain of possible actions. As mentioned previously, the strength of this approach is to help the value estimation function by exploring the most probable outcomes according to the environment dynamics.

Hierarchical controllers can be used in tasks with multiple heuristic resolutions of actions possible in the environment. In a sense it corresponds to the 'divide and conquer' reasoning for increasing the learning and performance efficiency of controller. In most applications of hierarchical reinforcement learning, we differentiate between micro and micro-level actions. While a low level policy can learn the finer dynamics of the environment according to the raw actions available to the agent, a higher level controller can be made to pick between different low-level policies which each correspond to semantic actions with a longer time-scale. This is presented in Sutton et al. [1999] as the *options framework*, and is used in many RL applications such as videos games [Romac and Béraud, 2019] which have a clear distinction between macro and micro strategies which are fundamentally different in nature. As well as separating policies with different temporal resolutions, we can use a hierarchical controller in order to use separate criteria for shaping the behaviour of an agent, for example safety and performance. In our case, the use of an additional layer of control allows us to separate the computation of uncertainty from the estimation of performance given an environment state. One of our objectives is to be able to not confuse aleatoric and epistemic uncertainties in the same return estimator. When mixing belief about environment dynamics (epistemic) with belief on policy outcome given a set of environment dynamics (aleatoric) then the we lose some of the understanding on the environment and are less able to react accordingly when choosing the best policy for that situation. Additionally, dur-

ing the training phase of an RL agent, when using a simulated environment we are able to train a policy without epistemic uncertainties, and hence it would make sense to include a separate level of control for dealing with the epistemic uncertainties of the real-world environment during the policy execution phase.

In our approach, we wish to use a hierarchical controller structure in order to increase the robustness of the overall controller with respect to possible changes to the environment dynamics from what was learned by the agent during training. We wish for this high-level controller to make use of the various contingency policies learned during the training phase, and be able to switch between the various policies according to it's belief about the current environment state, to be as robust as possible with respect to policy failure (in our autonomous navigation use-case: vehicle collisions). In this chapter we present our work on combining contingency plans with the optimal, along with a hierarchical planning layer, with the following contributions:

- We improve our method for concurrently learning policies, with the aim of the contingency policy being increasingly robust to switches between low-level policies from the high-level planning module.
- We combine learned policies with a high-level model based controller, and experimentally show through an autonomous navigation task that our approach is able to achieve a much safer agent performance in the case of a stochastic environment, while sacrificing a minimal amount of performance.

5.2 Related Work

5.2.1 Hierarchical Reinforcement Learning

Hierarchical reinforcement learning [Dayan and Hinton, 1993, Sutton et al., 1999] is a promising approach for helping RL algorithms generalize better in increasingly complex environments. Several works apply the hierarchical structure of control to navigation tasks, which lend themselves well to modular controllers [Fisac et al., 2018]. Approaches such as Andreas et al. [2017], Nachum et al. [2018] use sub-task or goal labelling in order to explicitly learn policies that are able to generalize through goal-space. Our approach differs where we don't wish to learn different goals, rather striving to attain the same goal under modified environment conditions, moreover, we aim to learn alternative strategies without having to subtask or goal labels. Works such as Machado et al. [2017], Zhang et al. [2021] also have the same aim of self-discovering strategies to be used in a hierarchical controller, though our method differs by targeting the value function of the agent, through additional reward terms. Similar to our

approach, Cunningham et al. [2015] uses a form of voting through policy simulation, though our work also integrates training the contingency policy with robustness to environment modifications in mind.

5.2.2 Agent Performance Estimation

Some approaches use an estimation of the confidence of the actions proposed by an agent's policy [Bouton et al., 2019, Clements et al., 2019, Hoel et al., 2020b] to determine whether or not an agent's policy is sufficiently good in the current environment state. When this is not the case, control is typically given to a separate, often open-loop controller looking to mitigate any possible negative behaviour if failure cannot be avoided otherwise [Dalal et al., 2018, Filos et al., 2020]. One issue with open-loop contingency plans – or any open-loop policy in general – is that they do not take into account the closed-loop nature of most real-world environments, whose dynamics are dependent on the actions of the agent and may themselves fail if not implemented carefully. Rhinehart et al. [2021], Killing et al. [2021], tackle the problem of high environment uncertainty, by prioritizing information gathering if the agent is too uncertain about its policy's outcome. These approaches choose to approach by default with caution, if ever there is missing or uncertain information in the agent's input space.

5.3 Improvements to Training Contingency Policies

5.3.1 Policy's Domain

In chapter 4 we presented a framework for learning one or more contingency policies concurrently to the optimal policy by augmenting the reward function with a penalty term encouraging contingency plans to exploit trajectories through sufficiently different areas of state-space, such that the collection of learned policies will be more robust with respect to local changes in environment dynamics. When seeking to combine this approach with a higher-level controller to determine the scheduling of those policies for acting in the environment, we choose to retain the our initial approach of learning the contingency policies since we were able to have multiple strategies with different behaviours consistent with our objective of making the overall controller more robust to changing environment dynamics.

However, there are further considerations to have once we plan on combining the various low-level policies and make them all available to the agent. Notably, the way we have set up the hierarchical controller is that the planner should

be able to dynamically allocate control of the agent to either of the available low-level policies according to its belief of the current environment state. The ability for an RL agent to act well is dependant on the distribution of data that it has seen during its training phase, and in the case of reinforcement learning, an agent will slowly converge to its expected trajectory to better learn the value of states that it expects to find itself in. In environments with high-dimensional state spaces, time and computing constraints prevent us from training the Q-function over the policy's entire domain. This means that the return estimator (Q-function) will be less well-learned in states that are poorly represented in its training distribution. This is due to the decreasing amount of exploration in the exploration-exploitation trade-off during training where the agent seeks to better learn areas of state-space that it considers as high in potential returns, in order to converge more quickly to the optimal solution without having to explore a lot of bad states before finding the desirable optimal trajectory. For example in the case of an ε-greedy exploration strategy the rate of exploration is slowly diminished so that 1 out of every 100 actions is exploratory (random action). Following this, the probability that the agent takes 5 exploratory actions in a row will be 1 out of 100^5, which is extremely unlikely. Even if $\varepsilon = 0.5$, the probability of taking 10 consecutive exploratory actions is approximately 0.1%. Although this is designed to help the policy converge faster, states that are further along unlikely trajectories are quickly filtered out of the training distribution, and the return estimation will have a high error at these lesser-seen states. The contingency policies which we learn explicitly seek to converge to trajectories which are far away in terms of state-space from the optimal's trajectory, meaning that the contingency agent's policy will likely not be well-learnt for states along the optimal's expected trajectory, and vice-versa. This becomes an issue when we wish to switch from one policy to another during an episode run, since we're giving control of the agent to a policy which will find itself at some point along the trajectory of another, in a state where its Q-function is potentially not well learnt.

We can call the state where control is switched from one low-level policy to another as the 'hand-off' state. In order for a hand-off from one policy to another to be done without issues, the return estimators for both policies must be reasonable well learned on that point of their domain. For most applications of this kind of hierarchical scheduling this is not an issue because the Q-functions are sufficiently well-defined at all points of interest of the policy's domain. Furthermore, usually navigating agents do not take into account the difficulty of going from one trajectory to another: for example, in the case of an autonomous vehicle, in order to go from a high-speed trajectory to a low-speed one, the inertia of the vehicle will mean that a series of breaking actions must happen before the vehicle finds itself along the low-speed trajectory. If an ego vehicle starts out at a high-speed trajectory, and at one point in time the planning module estimates that a collision is about to happen and the ideal policy to follow is a low-speed trajectory, then according to the time the ego has to react, we may need some hard breaking actions which might not have been needed for either

training distributions of high and low-speed policies alone. Due to this fact the contingency policy must be trained to not only solve the task using a different strategy to that of the optimal agent, but also know how to *correct* for the initial erroneous behaviour made from the intially acting policy. We can clearly see that the effectiveness of this hierarchical set-up is dependent on the quality of the planner and the environment parameter estimation, since the better we are able to predict the outcomes of a scene, the less likely it is that the controller will have to perform some last-minute policy switches at a hand-off point that is difficult for the new acting policy to handle. Of course this could be seen as insufficient exploration during training where this use-case could have been covered, however due to the inherent limitations to the amount of exploration an agent can do during training, it isn't reasonable to assume we can predict and cover every use-case in the training data.

In the following section we motivate the need for a contingency plan to be able to correct for initial bad actions from the agent, with a simple maze environment in which we have limited the amount of training each agent has done.

5.3.2 Illustrative Example of Contingency hand-off Point

In order to illustrate the effect that lack of learning across the appropriate domain might have when switching policies mid-run, we use a simple maze environment having more than one possible valid strategy, in which we've purposefully limited the amount of training two different policies have done. Figure 5.1 illustrates an example toy maze game, where an agent starts on the green tile, and must navigate to exit the maze through either of two exits (red / blue). Since we anticipate that some tiles may become blocked during the agent's execution phase, we wish to learn a contingency agent alongside the optimal that is capable of exiting the maze through either one of the exit tiles. In our case the optimal agent's policy is represented in blue, and successfully navigates from the starting tile along the shortest path to the exit. A well-designed contingency policy, represented by the red arrows, will be able to successfully navigate from the starting tile to the maze exit by means of a different trajectory.

In our example, we have trained the policies such that their respective value functions Q^{π} are well-learnt on a domain that we expect them to perform on. However due to limitations in computational budget we are unable to correctly train the Q-functions everywhere on the domain and so there are some states for which the error on the Q-function is such that the correct behaviour is not learnt yet. During the agent evaluation step, both agents always start from the green tile and, crucially, this is also reflected in the training regime where the initial state distribution is the same as for the evaluation part of the algorithm. This is important to note, since we don't consider the possibility of switching

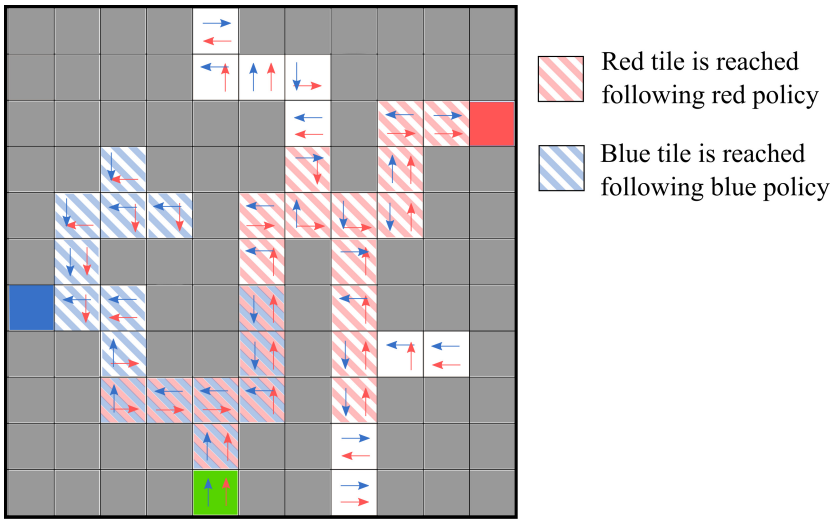


Figure 5.1: Maze environment, with multiple exits (blue & red)

from one controller (policy) to the other during the execution step.

The resulting behaviour is that there are some states in which an agent following the blue policy might end up, from which the red policy is then unable to correctly navigate to the red exit tile. In Figure 5.1, states where the agent can successfully navigate to either blue or red tiles are shaded in the objectives' respective colors. States which are shaded both colors are ones in which the agent is able to switch between objectives without losing the ability to navigate to either of them. If one of the policies, say the red policy, is to be used as a contingency plan, we must make sure that it's functional domain is large enough such that it covers the most probable so-called 'hand-off' points from the optimal policy's trajectory.

We observe that the need for this property becomes stronger in environments where the agent has a hard time navigating between specific areas of state-space. In our experiments on an autonomous navigation task, the inertia imparted the ego vehicle by an overly aggressive policy may be difficult to handle for a contingency plan which then requires strong breaking if, during training, the contingency agent has not had to use strong breaking to reach it's low-speed state-space. This could potentially be fixed by changing the policy's output to be a desired speed, for example, and let a lower level controller handle the desired acceleration values. However this is application-specific and moves complexity from our machine learning model to a separate controller which may not be desirable either.

5.3.3 Random Contingency Initialization

A standard approach for increasing the domain where the policy is valid is to increase the variance of states present in the training data. This allows the value-function to be better learned over a greater domain, at the cost of convergence speed. To this end, our approach is to modify the initial state distribution for an agent to force a greater initial state robustness for the contingency agent. Taking inspiration from the maze example in Figure 5.1, a first naive implementation is to randomize the starting state over all possible environment states. Given the initial state distribution $p(s_0)$, we define the new initial state distribution for the contingency agent as:

$$p^{\pi_1}(s_0) = (1 - \beta)p(s_0) + \beta p(s), \tag{5.1}$$

where $p(s) = \mathcal{U}(\mathcal{S})$ is the uniform distribution over environment states, and $\beta \in [0, 1]$ is a parameter controlling the ratio of initial states sampled from the regular initial state distribution and ones randomly sampled from the available state-space. The optimal agent π^* uses the standard environment initial state distribution: $p^{\pi^*}(s_0) = p(s_0)$.

However, with the aim of using the contingency policy to correct bad decisions made by initial acting policy, we wish in fact for the contingency policy to be specifically robust to areas of state-space heavily explored by the optimal policy, being the most likely 'hand-off' point to the contingency policy. Having both optimal and contingency policies train concurrently allows us to sample random states in the replay buffer of the optimal, as initial states for the contingency policy. This is equivalent to letting the optimal agent take the first random number of steps in the environment before handing over control to the contingency policy, mirroring the intended use-case for this training paradigm. We use $p(\tilde{s}_{\pi^*})$ to represent states sampled from the π^*'s replay buffer.

Moreover, our aim with replay buffer intialization is to help the robustness of the contingency agent over the most probable 'hand-off' points with the optimal agent when the latter is unable to handle the current environment parameters. To this end, we may further use domain knowledge to constrain the states sampled from $p(\tilde{s}_{\pi^*})$ such that they best represent these possible 'hand-off' points, as explained in the next section.

5.3.4 Replay Buffer Contingency Initialization

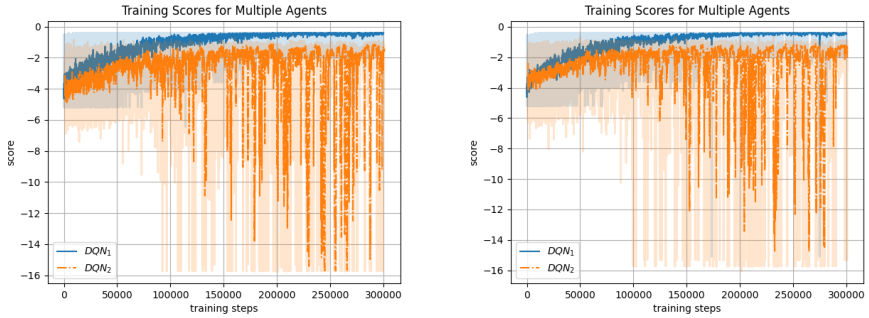
To increase the likelihood that $p(\tilde{s}_{\pi^*})$ well represents these states, using domain knowledge, we constrain it to have a uniform density only over states where the ego vehicle has not yet passed the intersection. This avoids the contingency policy learning to act once the ego has passed the intersection which is not useful

in our use-case:

$$p(\tilde{s}_{\pi^*}) = \mathcal{U}(\tilde{\mathcal{S}}), \quad \tilde{\mathcal{S}} = \{s \in \text{Memory}(\pi^*) \mid s|_{x_{ego}} < x_{int}\},$$

where $s|_{x_{ego}} < x_{int}$ represents all states in which the ego has not yet crossed the intersection. In practice we only have access to observations of states in the memory buffer hence we map the sampled observations, which contain the ego's position, back onto environment states which would result in the sampled observation. Even though we're not assured to map back onto the exact same environment state as was encountered in the optimal agent's state-space trajectory, this nevertheless increases the contingency agent's ability to be robust to initial states sampled from the optimal agent's trajectory.

Following Equation (5.1) we explore multiple values for β to see the trade-off between increased generalization capability and convergence speed for the contingency agent. Figure 5.2 shows how the training phase returns of the contingency agent are affected by the modified initialization distribution. We can see that the higher value of 0.8 leads to a higher variability of the returns, which can be expected from the greater need for generalization to deal with the difference in starting states.



(a) $\beta = 0.8$ (b) $\beta = 0.2$

Figure 5.2: Training run with different values for β using (5.1) for initial state distribution.

From our experiments we found that a ratio of 0.5 between regular and optimal-replay-buffer episode initializations gave the best results. Lower values tended to decrease the contingency agent's ability to function well at the 'hand-off' points, whereas higher values tended to overly impact the convergence of the contingency agent's parameters; $p(\tilde{s}_{\pi^*})$ adds a lot of variance in the state-space encountered by the contingency agent, and hence increases the complexity of correctly learning Q^{π_1} over this larger domain. In our experiments used for training an agent in the intersection scenario we use the value $\beta = 0.5$ which we observed to be the best compromise between generalization and convergence, with an equal amount of initialization between the initial distribution $p(s_0)$ and

from the optimal agent $p(\tilde{s}_{\pi^*})$:

$$p^{\pi_1}(s_0) = \frac{1}{2}p(s_0) + \frac{1}{2}p(\tilde{s}_{\pi^*})$$

5.4 Hierarchical Controller and Planner

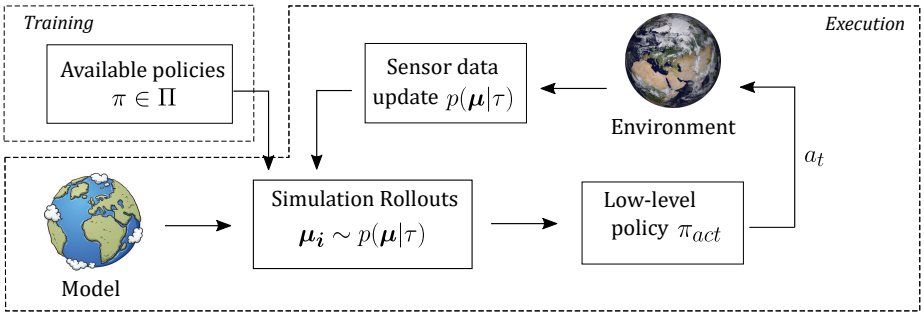


Figure 5.3: Structure of hierarchical controller composed of available policies and model-based planner (High-level policy selection).

As mentioned in the beginning of this chapter, contingency plans alone are not sufficient: a higher-level controller is required, capable of selecting the best policy with respect to the current environment observation. In our approach, we combine the learned optimal and contingency policies with a model-based planner, in order to increase the robustness and safety of the acting agent with respect to environment uncertainty. Figure 5.3 shows the structure of the hierarchical controller, whose high-level policy selection is in fact the planner module responsible for estimating the safety of each of the available policies $\pi \in \Pi$, and selecting the one with the lowest estimated chance of failure. A pseudo-code description for the hierarchical controller is provided in appendix 5.4.1. In this approach, we indirectly prefer trajectories with a low level of uncertainty, which does not take into account a potential data collection step for areas of state space which may have high epistemic uncertainty – able to be reduced through further exploring the state-space.

5.4.1 Hierarchical Controller Algorithm

Algorithm 4 shows how the hierarchical controller chooses between available policies. The idea is for it to choose the policy with the highest estimated safety in the environment, using estimates of the environment dynamics.

Algorithm 4 Executing Hierarchical Controller

$\Pi = \{\pi^*, \pi_1\}$ ▷ available policies
Init o_0 ▷ initial env observation
$\tau = \{o_0\}$
while episode not terminated **do**
  for $\pi \in \Pi$ **do**
    for i in simulation budget M **do**
      $\boldsymbol{\mu}_i \sim p(\boldsymbol{\mu}|\tau)$ ▷ sample dynamics given observation history
      $C^{\pi}_{\boldsymbol{\mu}_i} = \texttt{Simulate}(\pi, \boldsymbol{\mu}_i)$ ▷ simulate env using $\boldsymbol{\mu}_i$, and env model
    end for
  end for
  $\pi_{chosen} = \arg\min_{\pi \in \Pi} \frac{1}{M} \sum_{i=1}^{M} C^{\pi}_{\boldsymbol{\mu}_i}$ ▷ choose estimated safest policy
  $a_t = \pi_{chosen}(o_t)$
  $o_{t+1} \sim \texttt{Environment}(a_t)$ ▷ policy acts, env returns new observation
  $\tau = \tau \cup \{o_{t+1}\}$
  Update $p(\boldsymbol{\mu}|\tau)$ ▷ Update conditional probability on $\boldsymbol{\mu}$
end while

5.4.2 Planner

The planner's role is to estimate the chance of failure for each $\pi \in \Pi$ available to the controller. The approach we retain for our purposes is straightforward: we perform roll-outs over the set of possible environment parameters $\boldsymbol{\mu}$ for each π, and return an estimated failure rate based on those roll-outs which will define which policy is selected by the planner. This process estimates the probability of policy failure given the current belief about environment stochastic parameters $\boldsymbol{\mu}$. Let $P(s_{fail}|\pi, \boldsymbol{\mu})$ denote the probability that policy π will fail for a given $\boldsymbol{\mu}$. The planner's goal is to select a policy according to:

$$\pi = \arg\min_{\pi \in \Pi} P(s_{fail}|\pi, \boldsymbol{\mu}^*),$$

where $\boldsymbol{\mu}^*$ are the true (unknown) parameter values. We may approximate it by sampling from a probability density function conditioned on the history of agent observations over the course of the episode $\boldsymbol{\mu}_i \sim p(\boldsymbol{\mu}|\tau)$. Let $C(\pi, \boldsymbol{\mu}_i) \in [0, 1]$ represent whether or not there is a failure (collision) from policy π, after roll-out. Then:

$$P(s_{fail}|\pi, \boldsymbol{\mu}^*) \approx \frac{1}{M} \sum_{i=1}^{M} C(\pi, \boldsymbol{\mu}_i),$$

for M samples of $\boldsymbol{\mu}_i$. We note that the quality of the approximation relies on how well the $\boldsymbol{\mu}_i$'s are sampled. The closer they are to $\boldsymbol{\mu}^*$, the better the planner will be able to estimate the true probability of failure for each policy. We use a simple approach consisting in eliminating $\boldsymbol{\mu}_i$'s from the sampling pool, if targets are observed behaving in contradiction to the considered environment parameters (based on vehicle speed, in our navigation task).

5.4.3 Sampling Environment Dynamics

The quality of our estimation of collision probability depends on the quality of estimation of environment parameters. The estimation of the possible parameters is updated using the history of observations from the agent: $p(\boldsymbol{\mu}|\tau)$. We start with a uniform density on $\boldsymbol{\mu}$ at the start of each episode, and with every new observation o_{t+1}, we compare the actions taken by target vehicles, with the actions according to either possible target behaviour model ($b_i \in \{0, 1\}$). If the observed target speed v^{obs} is within a certain threshold ε_v of the simulated behaviour speeds $v_{b_i}^{sim}$, then that target behaviour is retained in $p(\boldsymbol{\mu}|\tau)$. Using a stricter notation we can write $\boldsymbol{\mu}|_i = b_i$, where:

$$p(b_i) = \mathcal{U}(B), \quad B = \{b \mid \left|v_b^{sim} - v^{obs}\right| < \varepsilon_v, b \in \{0, 1\}\},$$

where v_0^{sim}, v_1^{sim} are the simulated speeds for a cooperative and aggressive target, respectfully. All b_i's are independent, meaning there is no correlation between target behaviours.

To obtain our success rate and average score results, we sample without replacement $M = 200$ values for $\boldsymbol{\mu}$, and average out both the success rate (i.e. how many times the ego successfully navigated the intersection without crashing), and the score (i.e. regular reward function in the environment).

5.5 Experiments

5.5.1 Simulation Environment

In our experimental setup we use a similar navigation task to the presented in chapter 4, although the position initialization for target vehicles has been modified to make them more random, and prevent biasing the agents towards a pre-defined, well-known solution. We also augment the targets with different behaviours, to which the ego must now learn to be robust to. For these experiments, we have moved the uncertainty of observation from the positions and speeds of target vehicles, to their behaviours. This is a better representation of the kinds of uncertainties that are present in the autonomous navigation environment, as the intentions of target vehicles in the scene are one of the major factors in the difficulty of planning trajectories in this environment. This also allows us to implement a variety of different behaviours for target vehicles that might resemble those encountered in real-world scenarios. However the consideration for uncertainty of outcome is similar whether we were to consider lack of knowledge about the positions of targets, or their behaviours.

Stochastic environment parameters

The environment dynamics depend on the targets' behaviours. The behaviour for each target vehicle is a random variable B_i representing degree of aggressiveness, which are collected in $\boldsymbol{\mu}$ where $\boldsymbol{\mu}|_i = B_i$ for $i \in [1, N]$ for N targets. In practice we use either $b_i = 0$ for a cooperative target, or $b_i = 1$ for an aggressive target.

During training we set $b_i = 1, \ \forall i \in [1, N]$. Both optimal and contingency policies are trained on these environment parameters.

Trajectory Penalty

We use the same implementation of the trajectory penalty as for learning contingency policies in the previous chapter: we replace the expectation operator $\mathbb{E}[\tau_\pi]$ in (4.6) by the mean value over the last 100 samples of π^*'s replay buffer. In these experiments we also use the speed of the ego vehicle as an observation feature $\phi(o) = \dot{s}_{ego}$. Trajectory penalty scaling factors used in (4.3) are: $\alpha = 3, \delta = 0.1$, which are fixed by a rough initial sweep.

5.6 Results

To evaluate the overall performance of our hierarchical controller, we compute the number of successes vs. failures on the navigation task, over a range of environment parameter values $\boldsymbol{\mu}$ for different agents. Table 5.1 compiles the results for each tested controller, over an environment parameter sample size of $M = 200$.

Table 5.1: Performances of various agent setups in the intersection environment.

Controllers	**Success rate**	**Average Score**
π^*	**0.496**	**−1.200**
$\pi_{\mathbf{1}}$ without replay buffer init.	0.885	-1.500
$\pi_{\mathbf{1}}$ with replay buffer init.	0.954	-1.466
H-control without replay buffer init.	0.890	-1.380
H-control with replay buffer init.	**1.000**	**−1.238**

Table 5.1 presents the performances of a range of agents in the intersection scenario. Success rate gives ratio of number of successes vs. failures over

all sampled environment configurations. Average score gives us the mean of scores obtained in cases where the agent does not fail. Results are obtained by averaging 4 runs over the same random seed.

From our results we clearly deduce that a single agent has a hard time generalizing to new target behaviours, even though it may have achieved optimal performance within its training environment. Although π^* has the highest average score in cases when it does not fail, it does not generalize well and performs very poorly in unseen environment instances. In our navigation task, the contingency policy that is learnt corresponds to the ego having a more conservative driving attitude; this explains why the success rate is higher for π_1 with respect to π^*, although the average score decreases due to sacrificing performance for safer behaviour. When adding the replay buffer initialization to π_1, the success rate further increases due to the increased ability of the contingency agent to generalize to greater areas of observation-space.

Compared to the individual policies, we expect the hierarchical controller to perform better, due to its access to a model-based planner combined with both policies. Interestingly, we find that the contingency policy with replay buffer initialization has a higher success rate than the hierarchical controller using π_1 without the initialization. This highlights the importance of generalization when designing safe, robust algorithms. Finally, we see that the highest success rate is achieved by our proposed approach of combining optimal, and robust contingency policies. More importantly, though success rates are similar with the single π_1 with replay buffer initialization, we are able to obtain a good performance score close to the optimal policy, due to π^* being available to the hierarchical controller. This demonstrates how our approach is able to ensure much safer behaviour in unseen environment configurations than a single policy, without sacrificing performance by being overly cautious.

5.7 Conclusion

In conclusion, we have presented an approach for learning multiple policies in an autonomous navigation task and adapting the approach to specifically learn a robust contingency policy, which when combined with a model-based planner, is able to increase the robustness of the agent with respect to stochastic environment parameters. In our intersection use-case, we are able to reach a rate of no collisions for any (sampled) environment configuration, even when we only had access to a single one of these configurations during training. We acknowledge that the planner module has a simple behavioural prediction for target vehicles, and although it is sufficient in our simulation environment to obtain good performance, better detection of the environment's stochastic parameters $\boldsymbol{\mu}$ will increase the robustness of the overall agent, and ultimately the effectiveness of having an available contingency policy.

One main advantage of this hybrid approach is the ability to separate performance and safety in an RL framework. Whereas using a single reward function and relying on reward engineering to obtain the correct behaviour can be arbitrary, in this case we are able to optimize for performance in a complex environment, and ensure a high level safety in unseen instances of environment dynamics without having to tweak performance and safety terms in a single reward function.

Chapter 6

Conclusion and Perspectives

6.1 Conclusion

During our work we have investigated an approach to ensure the robustness of an autonomous agent's behaviour in the face of environment uncertainty, additionally taking into consideration the balance between safety and performance in the navigation task. We use the data-driven deep reinforcement learning approach which is well-suited for tackling environments with complex state-spaces. Given the nature of the objectives for implementing autonomous navigation in an industrial context, we must keep in mind the need for explicitly measuring the level of risk that is taken by the autonomous agent, such that it may also provide a way for judging whether or not the level of risk associated with a proposed trajectory is acceptable. This has been implemented in the form of a hierarchical planner which has been presented in the final chapter, and allows for a stricter control over the performance of an agent in the stochastic navigation environment.

Our initial work dealt with learning the distribution for episodic returns as a proxy for potential outcomes. The idea behind this approach is that classical RL algorithms use the expectation of outcome (i.e. the mean value) in order to inform decisions in the agent's current state. However the presence of uncertainty may lead to the presence of multiple outcome modes, which may change the value of this proxy to something which will not actually correspond to any meaningful outcomes that might occur, since only one of the present modes will be realised during the episode run. This approach seems sensible, due to the ability of using quantile regression in order to learn quantiles of the

return distribution. The formulation in terms of learning quantiles gives us the added benefit of being able to specify a minimum resolution for the probability of events, setting a threshold for probabilistic mass (total probability) for outcomes which we wish to be sensitive to.

Though when we apply the distributional RL framework to a stochastic navigation task with high episode lengths, we notice that there are some issues with both the practical application, and the way of using the return values as a proxy for estimating the probability of certain outcomes. Asides from the ability for the model to converge to a meaningful, qualitatively exploitable distribution, the ambiguity between the use of modes as separate strategy indicators, and the fact that return values to outcomes is a non-injective mapping, means that this approach is not well suited for our application of wanting an agent choose a suitable strategy according to the possible episodic outcomes.

To counter this issue, we instead develop a framework based on a hierarchical structure, dealing with aleatoric and epistemic uncertainties at separate levels. This approach has a greater focus on the robustness of a policy being able to act correctly following a changing perception of the environment. The presence of multiple concurrent policies also allows us to increase the learning efficiency of the algorithm by the logic of 'divide and conquer', since we are no longer attempting to learn a single policy model for all possible use-cases of environment dynamics and agent perception quality. We have shown that this approach has the ability of increasing the overall safety of an agent in an autonomous navigation task, without overly sacrificing the performance as can be the case for the 'freezing robot problem'. This property is crucial for an autonomous agent to be able to interact with an environment with some measure of success, especially in the case of autonomous vehicles which are expected to interact with humans in a navigation task where common sense is often referred to as a means for balancing the amount of acceptable risk associated to our actions on the road, as well as in terms of cooperation with other vehicles.

Overall, results from our proposed hierarchical model are promising, and moreover remain general enough in its formulation to have the potential for application to a greater domain of tasks, not just automotive navigation. There is a point to be made about the use of a simulation environment for validation, which is still one step removed from applying the algorithm to an industrial context, along with the integration of the proposed controller to a real-world vehicle architecture. Making the bridge between performance in simulation vs the real world is complex and RL applications are no exception. For this reason the experiments and results of this work are focused on validating the autonomous agent's behaviour inside of a simulation environment, such that they can be developed more quickly and efficiently; adapting the controller to a real-world prototype is another task in and of itself. In conclusion, our work has investigated a couple of approaches to adapting a machine learning algorithm to a stochastic decision-making problem, with some good measure of

success. We have been able to highlight some of the most important aspects to take into consideration for this problem statement, mainly concerning the need for generalization from the agent, to both be able to switch between different behaviours during an on-line run, as well as of course generalizing to other areas of its task domain. This is shown in our need for including a specific modification to the training regime (section 5.3.2) such that multiple policies are able to be combined in a hierarchical structured controller.

6.2 Perspectives

6.2.1 Continuation of our Work

Given the current state of our work, there are some clear directions for further development of our hierarchical controller, notably the way multiple policies with different strategies are learned in the environment, the planning module, and the method for stochastic parameter update.

We recall that the aim for learning multiple policies with different behaviours in the environment is so that we may expect them to be robust to local modifications of the environment dynamics, or local changes to the agent's perception (such as obstruction of one spatially correlated part of the input space), as is shown in figure 4.1. However in our logic of learning 'plan B's', we make the assumption that exploiting different areas of state space will result in an overall robust environment. A better way of looking at the problem, and the apparent way that contingency strategies are formulated by humans, is that the contingency strategy is actually generated as the optimal solution of an 'environment B' which is imagined by the agent as having the most expected modification to the original environment, and whose optimal policy is different from the current one. With this in mind we can formulate this objective in the same terms which we have used in this work: instead of aiming to learn π_1 with a condition of distance between policy behaviours

$$\mathcal{M}\left(\mathbb{E}\left[\tau_{\pi_1}\right], \mathbb{E}\left[\tau_{\pi^*}\right]\right) \geq d,$$

we should instead attempt to learn $\boldsymbol{\mu'}$ (which defines the dynamics and observation model of the environment), such that:

$$\mathcal{M}\left(\mathbb{E}\left[\tau_{\pi^*_{\boldsymbol{\mu}}}\right], \mathbb{E}\left[\tau_{\pi^*_{\boldsymbol{\mu'}}}\right]\right) \geq d,$$

and then retain the relevant contingency policy $\pi^*_{\boldsymbol{\mu'}}$ where $\pi^*_{\boldsymbol{\mu}}$ denotes the optimal policy in the environment defined by stochastic parameters $\boldsymbol{\mu}$. This way, we are actually learning to anticipate changes in the environment rather than blindly learning policies with different strategies from one another in the hope that they will be useful in a modified version of the training environment. We

can think of an example where we are learning what clothes to wear when going on a walk outside when it is currently sunny, though it may rain later. With our current approach, we may learn to use an umbrella, or even go out without a shirt at all, since both of these can be different enough from our optimal policy of using just a shirt for sunny weather. However clearly only one of these two 'different strategies' (i.e. the umbrella) is the correct solution for if there is rainy weather, and the no-shirt strategy – although being sufficiently different from the shirt and umbrella strategies – will be useless in the new environment. We can expect that by using the 'environment B' perspective, the learning of useless policies will have been eliminated.

Moreover, the field of reinforcement learning is seeing some big advancements in terms of robustness with respect to both domain generalization, and task generalization, known as meta-RL (learning to learn). Models which use architectures that are able to decompose the defining elements of tasks into some value that can be treated as an input to the autonomous agent, such as Kirsch et al. [2019], are seeing some good results for knowing how to adapt to new unseen tasks. However, as discussed in Badia et al. [2020a] the ability for agents to learn over a great number of environments is a challenging one, showing how far we still are from the equivalent of a 'general AI' in terms of agent control, even for domains with similar structures and goals such as the Atari suite. We have considered a more ad-hoc approach for increasing the robustness of an agent faced with different data distributions due to the element of safety being so crucial, since this method gives us a more explicit estimate for outcome probabilities and allows us to have greater control for the agent's performance. However a more general solution with perhaps a greater ability for adaptation across all autonomous navigation tasks (i.e. not just intersection scenarios), might require such an approach as meta-RL, and this direction of work is definitely an interesting one to pursue in order for a comprehensive autonomous solution to be developed for AVs.

A natural continuation for this work, as mentioned in the previous section, would be to validate the decision-making in a more advanced simulation environment, taking into account real models for sensor uncertainties, and interactions with target vehicles, as well as realistic vehicle dynamics. Some assumptions that we have made during this work have been for example: incoming objects are always detected, the agent has full knowledge about the direction of travel for the target vehicle, or that the actions recommended by the autonomous navigation algorithm are able to be exactly executed by the vehicle's actuators (no feasibility issues, no communication errors within the vehicle's systems). Although this work strays away from the machine learning element of the project, it is nonetheless vital for integrating such a system into a modern vehicle's software architecture.

6.2.2 Machine Learning for Autonomous Navigation

Achieving autonomous agent control is possible with a wide range of algorithms using different approaches than data-driven machine learning. However one clear advantage that data-driven algorithms have is the ability to improve over time, and improve with additional data collected by different agents performing the same – or similar – tasks. This is part of the strength of off-policy learning for example (section 2.1.3), where an agent can improve its performance by learning from experiences of other agents' interactions. The ability for collaborative learning might mean an exponential improvement in an autonomous vehicle's ability to react to the various scenarios it encounters whilst driving, since they will have been learned by at least one of the other agents collecting data about the driving environment. We could benefit from an implementation using data sets between different physical zones where the driving style is different (say, more or less cooperative), in order to rapidly adapt an agent behaviour on the road from one driving style to another. The value of having extra data to use for both training and validation is the reason why modern car companies are collecting driving data from both test vehicles as well as commercially sold ones, in some cases. As we have seen primarily in the world of social media, we might expect that the value of personal driving data will increase as an increasing number of autonomous driving systems are rolled out.

The initial push for developing AVs has been motivated by the aim of reducing the amount of deaths and accidents that happen on the road each year. Our work largely follows the same motivations, and the general problem statement that has been considered for this task has been that of an autonomous agent having the capabilities of co-existing in an environment with human actors. One of the reasons why the development of a truly autonomous vehicle is such a complex task is due to the unpredictability of human actors which may not always be considered as rational actors especially during driving scenarios. For example, many autonomous transport systems have already been implemented such as trains or trams which usually do not have to take into consideration the high combinatorial complexity associated with such human-robot interactions.

For this reason there is still some need to balance out safety and performance criteria, however this entails the necessity for placing some kind of threshold between the two, which can at times seem quite arbitrary. If we consider the natural human approach to risk in such a situation, we also have some type of natural threshold past which we will change our behaviours due to the situation being more or less risky. However this of course is not some pre-set constant value: firstly, the level for risk tolerance may vary greatly between individuals, and secondly your personal tolerance for risk is susceptible to be influenced by external factors which may lead to a variation is decision-making according to when the task is performed. We may also consider the fact that the 'decision-boundaries', which represent the limits in our input space which may make us

consider multiple possible courses of action, are not hard-drawn lines, but rather blurry boundaries for which we can attribute a probability for taking either possible course of actions that is considered 'good enough'. Following this, the way we code safety requirements into an autonomous algorithm attempts to evaluate the realistic probability for collision according to expectations of outcomes, and then react to the best available strategy according to these expectations. However in human decision-making, realistic levels of expectation are always distorted such that even an action that is good in expectation may appear as not desirable due to the uncertainty tied to it. Veritasium [2015] shows a version of this, where in spite of good odds for winning money, some people may decide not to take a bet simply due to the risk associated with the decision, independently from the fact that the expectation of outcome is to win money. The level of risk-aversion should be taken into consideration when designing an autonomous agent to not only interact with human agents, but to imitate human behaviour in the driving task. We may ask the question pertaining to the level of risk-aversion a self-driving car should have in order to be considered reasonable in order to implement it into the driving environment. Even when learning to drive and obtaining a driving license, we are taught that an excessive level of risk-aversion may be dangerous during the driving task, due to the behaviour being not expected by other actors in the environment. Considering this, we could make the argument that the safest form of driving is the one which is the most expected from other agents in the environment. From the point of view of our autonomous algorithm there is certainly some truth to this, since having a high degree of knowledge about the future actions of other agents in the environment diminishes the total uncertainty of the environment and makes it much easier to plan optimal trajectories. This reasoning is further supported by some of the most effective safety features on modern vehicles, namely indicators and break lights, which are used in order to indicate intention to other road users so that they may formulate the best plan to adapt to your driving trajectory.

Implementing deep machine learning algorithms for AV control shows some promise, most notably for their ability to generalize over a large domain of scenarios with a high-dimensionality, as well as their ability to use incoming data to continuously improve their performance over time in a collaborative manner, potentially using data from a large fleet of AVs. The most constraining elements of implementing an autonomous control algorithm is the high dimensionality of the input space, and a high degree of uncertainty related to both scene detection, and behaviour of other road users. Nonetheless AVs are already seeing some use on use-cases with the lowest probability for unplanned interactions such as highways, or intersections with clear traffic rules where nuanced decisions do no have to be made. The added constraints that come with limiting the use to simple use-cases allows us to make simplify the input space and reduce the need to make safety-performance trade-offs since we are able to almost always ensure a high degree of safety due to the high observability of the environment. Validation of such algorithms in terms of safety can be given by some threshold value in terms of the amount of 'wrong' decisions taken by the agent, normalized

to the distance travelled on roads or road-types, and given as some very small value such that the error rate is not only on par with, but much lower than for human actors. This can be done when there exist clear labels for correct and incorrect behaviour, however if we should want to use AVs in navigation tasks with inherently higher degrees of uncertainty where it is challenging to define an objectively correct behaviour, then the combination of both planning and learning algorithms seem to provide the best compromise between verifiable safety and strength of adaptation. This factor of explainability, which is so crucial for justifying the actions taken by the autonomous agent, is a weakness of deep reinforcement learning which can be complimented by a non-machine learning algorithm, such as that which we have used in our hierarchical planner which learns to schedule contingency policies learned by deep models.

Appendix A

Simulation Environment

This appendix presents a more in-depth view of the simulation environment developed for our work for both training and validation. This simulation tool was continuously developed over the course of this thesis by Renault's path-planning team with the purpose of being able to provide a proof-of-concept for autonomous algorithms by validating autonomous agents in more or less complex driving scenarios.

In the context of this work as an industrial CIFRE PhD, we had to decide on a training and validation pipeline in order for our work to later be integrated in Renault's models as a part of their path planning project. The algorithm training and validation is done in a simulated driving environment, and as a part of this PhD project we must choose a simulation environment to fulfill this role. Renault already has simulation environments such as SCANeR, which it uses for validating ADAS functions. SCANeR is a powerful tool, and includes models which account for chassis dynamics, as well as tyre-road contact mechanics, amongst others. Although these are an important consideration to have before implementing algorithms into real-world prototypes, model complexity during simulation is something that we strive to reduce as much as possible in reinforcement learning applications, due to the limitation on computational resources for generating as many agent-environment interactions as possible during the algorithm's training phase. For this reason, we have decided to develop our own lightweight framework for training and validation. This approach not only allows us to optimize the computations done with an application to machine learning, but also gives us the flexibility of adapting the framework for being able to contain additional testing scenarios, along with modifying vehicle behaviours more quickly than if it were a larger, more widely used tool.

A.1 Driving Scenarios

The main driving scenario that was considered in our work is the intersection scenario, and most of our work has been focused on various implementations of the intersection scenario, with the overall structure of the task remaining the same.

As can be seen in figure A.1 the intersection environment consists of a 4-way intersection where each road allows for both directions of travel for vehicles. We consider that a single one of the vehicles is controllable and referred to as the ego, the rest of the vehicles in the driving scenario being referred to as targets.

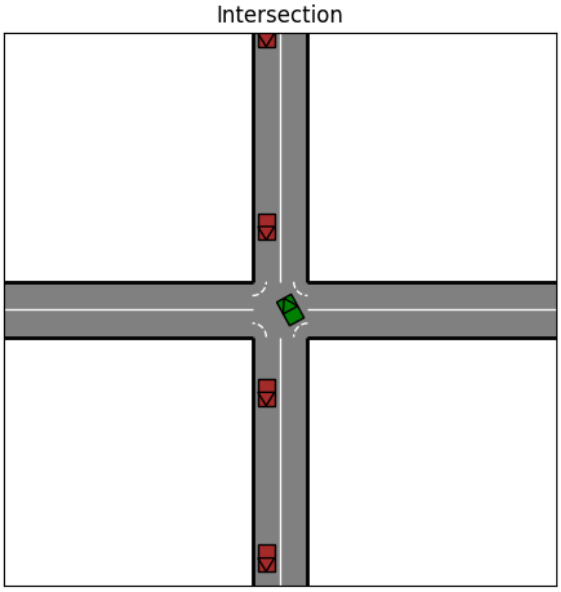


Figure A.1: Capture of the simulated navigation task.

In this situation, a vehicle has three possible path choices once reaching the intersection: turn left, turn right, or follow straight ahead. The path for either one of these options is pre-determined such that once the intention of the vehicle is known, then it will follow this path and not have to act on any lateral movement in order to reach its destination. This means that an agent will only have control over its longitudinal position along this path. For this reason we use a Frenet frame of reference for each vehicle to describe their position relative to their path, with the tangent part always being 0 (always on the path), and the longitudinal variable changing according to the speed control of the vehicle. The path while crossing the intersection is made up of a straight line until the vehicle is onto the intersection, and then a quarter circle until the center of the

objective lane, if the vehicle turns either way, or continues straight if the vehicle crosses straight through the intersection.

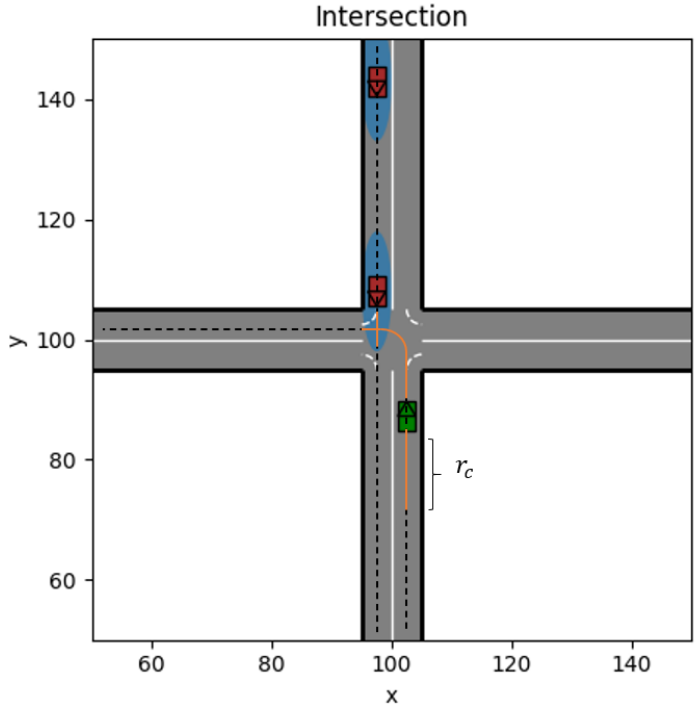


Figure A.2: Intersection scenario with oncoming vehicles. Dotted lines represent the path for ego (green) and targets (red). Shaded ellipse around the targets represent the uncertainty on their positions, whereas the orange zone around the ego, of dimension r_c represents the collision radius around the ego vehicle.

In practice, each vehicle has access to a relative frame of reference (Frenet), along with a transformation available to the environment, which provides a way to compute the absolute position of the vehicle with respect to the intersection, according to the different starting points of that vehicle which may vary according to starting lane and initial offset (how close it is to the intersection). As a standard value, we initialize most vehicles at approximately $100m$ from the intersection which corresponds to $5s$ of driving at the initialized speed of $20m$. In almost all of our experiments, the ego vehicle is initialized at $100m$ from the intersection.

With this intersection set-up, we have a large number of possibilities in terms of navigation tasks for the ego to learn to solve, according to the possibilities for target vehicles to arrive from either sides of the intersection. Due to the symmetry of the scenario, we may fix the ego as always coming from a single lane (the bottom lane, for example). As our nominal scenario for performing

initial experimentation, we pick one where the target vehicle must cross an incoming flow of vehicles which may or may not give way when crossing at the intersection. In our implementation we consider a case where there are no predetermined traffic rules pertaining to giving or having the right of way in the intersection. This gives no prior information about the expected trajectory for the target vehicles and allows for the ego to learn in a slightly more complex use-case. Moreover, since we are designing and algorithm to have a high degree of safety it is reasonable to consider an adversarial target vehicle which would not respect the right of way indication. We use the left-hand turn scenario where targets are oncoming from the opposite top lane, and cross the ego's path. Given the symmetry of the problem, including target vehicles in other lanes is almost equivalent to placing additional targets in the oncoming lane due to the fact that multiple vehicles cannot be in the intersection at the same time due to the collision radius attributed to the vehicles. In order for the ego vehicle to avoid collisions, it must wait outside the intersection and only go through once all other vehicles have exited the intersection.

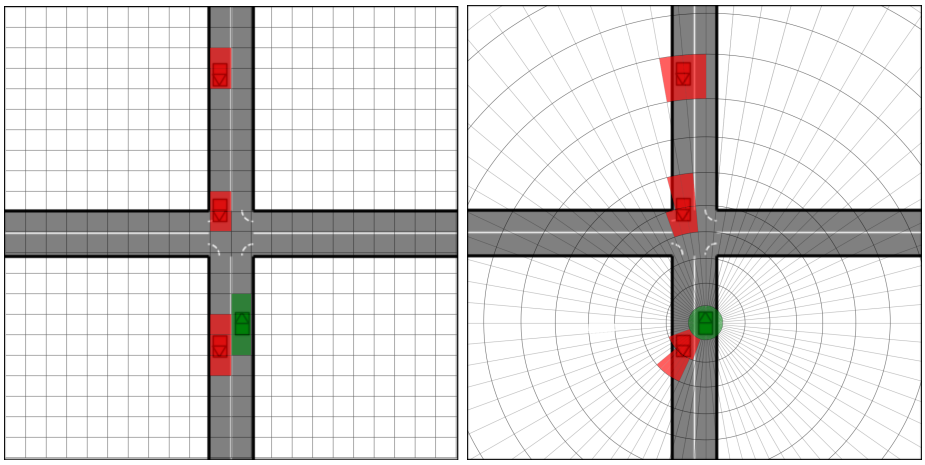
During our work, we have maintained the same task topology for training and validation, though the behaviours as well as action and observation spaces are modified in order to add complexity and variation to the autonomous navigation task. During development, we have kept the simulator as general as possible so as to implement different topologies in the future such as merges, or roundabouts for example.

A.2 Input and Output Spaces for Autonomous Agent

A.2.1 General Description

There are multiple approaches for modeling the observation space for an autonomous vehicle. By far the two most popular are the occupation grid, or a list of objects. Figure A.3 shows an example of what occupation grid inputs look like to the agent. In real-world applications sensors will return information such as depth and angular position, making the radial occupation grid the most realistic, however in some control environments such as mazes or video games, a square occupation grid makes more sense.

The same information that is contained by the occupation grids can also be more compactly represented as a single vector containing relevant information such as the relative positions of targets in the environment, along with speed and other information about the targets and/or the ego which are not able to



(a) Square occupation grid (b) Radial occupation grid

Figure A.3: Different occupation grids from a single input state to the agent.

be conveyed by the occupation grid representation:

$$s_{input} = \begin{bmatrix} X_{ego} \\ X_1 \\ X_2 \\ X_3 \end{bmatrix}$$

Each approach has its own advantages and disadvantages. When using an occupation grid, the size of the input remains fixed, while the size of a list of objects may be of varying size according to the amount of vehicles in the scene. Additionally, an occupation grid by itself does not transmit any information about the speed or acceleration values of target vehicles in the environment. This can be overcome though, just as in Atari game implementations which use raw pixel data as inputs, by stacking multiple subsequent frames together as the input to the agent's policy, so that the notion of speed and acceleration may be transmitted to the agent, although these values have to be learned. Moreover, the occupation grid is useful for transmitting information about spatial correlation of objects in the scene. This may help to capture the spatial interactions between vehicles in a more succinct way than if the raw position values were to be fed in the agent's policy. This property of the observation grid treats the scene just like an image containing pixel values, and for this reason an occupancy grid input space is usually paired with a convolutional network model which is well-adapted to extracting information about spatial correlation. It is important to note though, that additional information that we may wish to be given to the agent for its decision-making, such as intentions of other cars, driving style, or trajectory history is more difficult to be given in observation grid form, and will require a more complex design of the policy's architecture

in order to accommodate the different nature and dimensionality of the input variables.

On the other hand, we can choose to supply the agent's policy with a list of target vehicles present in the navigation scenario, that is commonly what is returned by the vehicle's sensor fusion module. This approach contains less spatial correlation information, however we're able to more easily pass information such as speed, acceleration, or driver intention for example, to the agent's policy. This allows us to combine the input information into a single vector to feed to a deep neural network model, and doesn't require a special consideration for extracting relevant information such as is the case for an occupation grid. However there are two difficulties relating to dealing with a list of objects: firstly the agent should be agnostic to the ordering of the objects in the list, and secondly the variation in the number of vehicles in the scene means that a list containing these objects will be of variable size.

To the first point, we wish for the agent to react in the same way to vehicles in the scene, regardless of their ordering in the input vector. If we say that all information about a single target vehicle can be written as X_i, then the agent's recommended action for either $\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$ or $\begin{bmatrix} X_2 \\ X_1 \end{bmatrix}$ should be the same. In practice this is difficult to achieve, especially in the case of learning algorithms, if the policy function is not designed with this property in mind. One way of compensating for this is how the targets are ordered in the input vector to the agent's policy. If for example we can make sure that the targets are always ordered according to the most critical target first, then we make sure that the vehicles will always be given as inputs in the same order for a similar scene. The second point is more difficult to treat, though it is important to take into consideration when feeding this input into a policy model. The input dimension for deep neural networks is not designed to change, and so it is set in advance according to the kind of data that will be seen by the model. However if we dimension the network for a list of 3 target vehicles for example, then we have to 'crop out' any extra vehicles from the policy's input (i.e. tell the policy to disregard any non-important vehicles when making a decision). If on the other hand there are less than 3 target vehicles, then we cannot simply set the respective input values to 0 since this will still affect the weights and biases of the policy model. Instead, one solution is to create imaginary target vehicles which are sufficiently out of the way of the ego such that they would not affect the ego's decision making.

In our implementation, along with the positions and speeds of both ego and target vehicles, we augment the input state with an estimation of the *time to collision* (ttc) between the ego and each of the targets. In the case where ego and target vehicles are on a collision course, while maintaining current speeds, this value represents the estimated time to collision between the two. However if the hypothesis of constant speed does not result in a predicted collision,

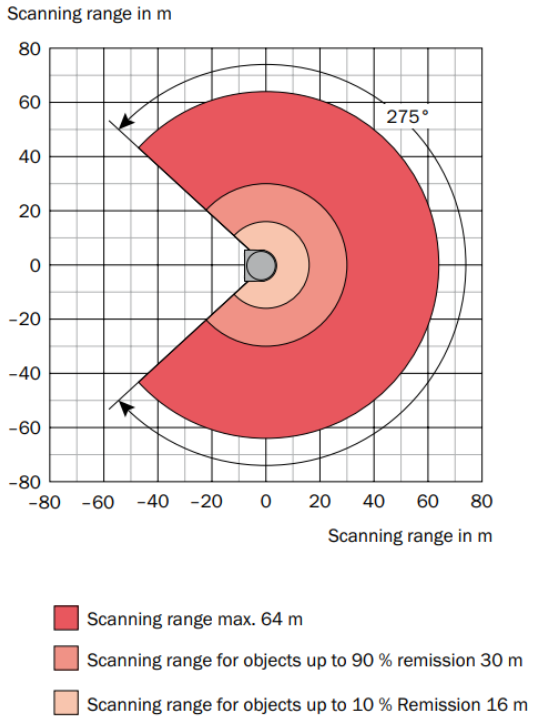


Figure A.4: Example characteristics for LiDAR ranging

this value is set to a maximum value. Although this information is redundant with the positions and speeds of the targets, providing this information as an explicitly computed value may help the policy in making appropriate decisions in the environment. In our experiments, we observed that the inclusion of this information on average helped the policy to converge, meaning that doing this allowed to alleviate some of the complexity for learning this computation from the neural network.

A.2.2 Observations by Vehicle Sensors

As mentioned in our overview of how an MDP is modeled, there may be an associated observation model $O : \mathcal{S} \to \mathcal{O}$ which dictates how information about the current state of the environment is perceived by the agent. This is the case in most real-world applications where the agent will not have full observability on all environment variables. In the case of our navigation scenario, the full environment state contains information such as the true positions, speeds,

accelerations, and paths of target vehicles in order to carry out the simulation process. However the agent is not able to perceive all of this information, given the limitations and uncertainties linked to the performance of its sensors. real-world sensors are calibrated and have their uncertainties quantified before being integrated into a vehicle design so that these values may be taken into consideration when used to decide the vehicle's next action.

Different sensors have varying degrees of certainty, for example LiDAR (light detection and ranging) is very useful for depth perception, whereas a camera is more useful for detecting the relative angular position of objects. Figure A.4 shows the scanning range of a 3D LiDAR sensor, typically used for target detection on roads, with the remission rate according to the distance to target. The values given in this figure are just for reference, as there exist more expensive and powerful sensors able to detect objects at up to 100m, for example. Although LiDARs, RADARs and cameras are the most commonly used, other sensors such as ultrasounds are a cheaper alternative for low-precision close-range detection. The information from these sensors is combined in such a way to give the vehicle a more complete picture of the environment around it. In our experiments we consider various degrees of uncertainty ranging from positions and speeds of target vehicles, as well as the driving style of the targets. Figure A.5 shows how the uncertainty on the target's position is seen by the agent.

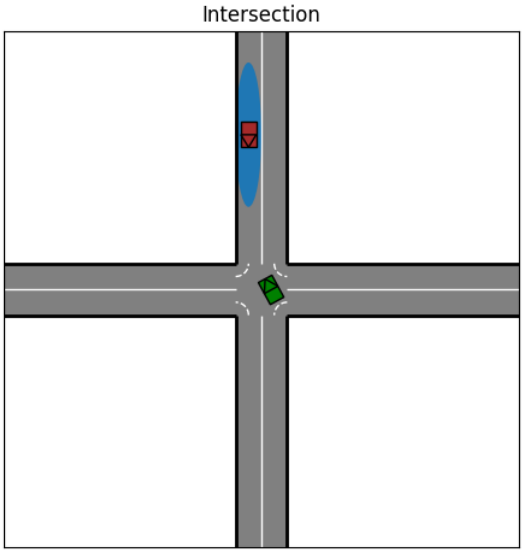


Figure A.5: uncertainty on the position of incoming target vehicle

The shaded area corresponds to the 2-σ zone of a Gaussian distribution on either side of the mean, along the 1-dimensional path that is defined for the vehicle. In our application we assume there is no uncertainty on the direction

of travel of the vehicles which translates to no uncertainty on the lateral position of the vehicles in their lanes (the 2-D shape of the Gaussian is simply for visual effect). In practice, the true position of the vehicle is known by the simulation environment, but the position detected by the agent is sampled from a 1-dimensional normal distribution according to the relevant value of σ. This of course increases the complexity of matching input states to return estimations, as similar perceived states from the agent may have been generated by targets at different true positions. In our experiments, we keep a constant value for σ such that the uncertainty on target vehicle position doesn't change with the distance between the ego and target.

A.3 Different Behaviours for Target Vehicles

In chapter 5 we use an uncertainty on target behaviours in order to model some level of uncertainty in the navigation environment. We define a degree of aggressiveness that will affect the reactions of target vehicle with respect to the ego. Along with modeling realistic epistemic uncertainties of the driving environment, targets reacting to the ego creates a closed-loop system which is more complex to plan behavior in than an open-loop environment which does not react to the actions of the agent. With this in mind, we describe below the different possible target behaviours that we used in experiments:

`act_aggro()`:
An aggressive target will act as if it has the right of way, and not react to the presence of the ego in the intersection. In this situation it is up to the ego to plan its trajectory around that of the aggressive target.

`act_coop()`:
A fully cooperative target will always give the right of way to the ego when it is approaching the intersection at some minimum distance. In the case where the target gives the right of way, it will come to a stop before the intersection as long as it is able to within the limits of its possible deceleration values.

`act_coop_with_slowing()`:
The same behaviour as for `act_coop`, except that a cooperative target will preemptively enter the intersection at a lower speed than an aggressive target. This is done so that there is a difference in behaviour early on in the episode trajectories between both types of target behaviours. Early behaviour detection is an important factor in the ability for the ego to react to the current environment state. If we are able to correctly plan our the future behaviour of targets then the task of risk reduction becomes much easier.

`act_coop_without_stopping()`:
In this iteration of cooperative behaviour, targets will slow down to a very low speed if they detect the ego's presence in the intersection, however will not come to a complete stop. This use-case allows the ego to be more confident in maintaining a high-speed trajectories to cross the intersection when the targets are more cooperative, however will not be able to totally disregard the flow of incoming targets simply because the first one is cooperative and will stop at the intersection.

Additionally targets are programmed in such a way that they aim to maintain a certain target speed (20 m/s), and will break in order to maintain a minimum distance with respect to the vehicle directly in front of them. This avoids same-lane collisions between the targets and makes collisions with the ego the only possible source of collisions in the navigation environment.

In the experiments described in section 5.5, the random variable $B_i = [0, 1]$ corresponds to either `act_coop_without_stopping()` or `act_aggro()` as either cooperative or aggressive behaviours respectfully.

A.4 Adapting the Environment for Training Deep Learning Models

A.4.1 Gym Framework

In the interest of compatibility between various environments and reinforcement learning agents, *gym* is a wrapper framework that help to model environments as MDPs, and gives a common interface so that agents can be applied to multiple different control tasks. The Markov property of the MDP means that the transition from one state to another is done using only information about the current state and the action taken by the agent. The episodic nature of MDP tasks also means that the episode must be repeatable. For this reason, the gym wrapper requires the definition of 2 main functions:

`step(action)` $\rightarrow$ `next_observation, reward, done, info`:
From the current state of the MDP, return an observation of the next state, the reward corresponding to the transition, a flag value in order to indicate whether or nor the episode is over (if the goal is reached or if the task is failed), and an extra variable to pass any extra information about the episode.

`reset()`:
Reset the state of the environment to an initial state (can be drawn from

initial state distribution), after the `done` flag is set to 1.

Defining a gym environment also requires defining the `observation_space` and `action_space` class attributes which respectfully define the input and output dimensions of the policy function (this makes it easier to automatically initialize network layer sizes, for example). Additional functions are able to be overridden, such as `render()` which draws the current state for the purpose of visualization. Using the gym wrapper is standard is RL applications, and allows for use of pre-existing libraries of agents such as *stable-baselines* [Hill et al., 2018] on custom-defined environments.

A.4.2 Computational Resources

Training time is a big factor is how efficient we can be in testing the efficiency of deep learning algorithms. For this reason the use of GPUs and TPUs (tensor processing units) for performing the computations needed for training neural networks at a higher speed, is important and makes a big difference when running a large series of experiments. Today modern GPUs and even CPUs have dedicated architecture in order to optimize computation times in deep neural networks. In the context of this work, we had access to computational resources within both Renault and EURECOM in the form of GPUs. Deep learning which deals with image processing benefits the most from a GPU setup as both computations on images and neural networks are optimized. However in the case of reinforcement learning, a large share of computing power is needed for simulating the environment dynamics rather than image processing. These operations are better performed by CPUs, and hence RL applications require us to balance the share of GPU and CPU usage so as to obtain the best overall performance. In planning algorithms, such as the famous AlphaGo [Silver et al., 2018], a higher number of simulations equates to a better expected performance of the agent as it allows it to explore a large area of trajectory-space and anticipate as many outcomes as possible. Our implementation of a planning module in the hierarchical controller uses a similar concept in that the higher amount of environment parameters we are able to simulate out, the higher probability we have of reacting to the correct one. In this sense, a greater simulation budget is as important as the computational budget for training deep learning models. In running experiments for our work, we often found that the limiting factor in the total training time was often the simulation budget rather than that for training neural network models.

Additionally, the use of a replay buffer to improve learning requires the use of a potential large part of computer memory to store these. Since GPU memory is rather scare and usually dedicated to storing the neural network models on which the GPU-optimized computations are performed, the replay buffer is stored on the computer's memory. However this means that when

training, the samples have to be loaded into GPU memory in batches, which further slows down training.

These dimensioning factors make up some of the reasons for choosing to develop our own simulation environment so that we are able to use as light a framework as possible in the interest of being able to run more extensive experiments.

Appendix B

Derivations and Proofs

B.1 Contraction of Bellman Optimality Operator

We define the Bellman optimality operator, applied to a value function V: $\mathcal{B}V := \max_a \sum_{s'} \mathcal{T}(s'|s,a)\left[R^a_{s,s'} + \gamma V(s')\right]$. Let V, U be two different value functions, we now prove the contraction of the mapping $\mathcal{B}$:

$$
\begin{aligned}
&\|BV - BU\|_\infty = \\
&\|\max_a \sum_{s'} \mathcal{T}(s'|s,a)\left[R^a_{s,s'} + \gamma V(s')\right] - \max_{a'} \sum_{s'} \mathcal{T}(s'|s,a')\left[R^{a'}_{s,s'} + \gamma U(s')\right]\|_\infty \\
&\leq \|\max_a \sum_{s'} \mathcal{T}(s'|s,a)\left[R^a_{s,s'} + \gamma V(s')\right] - \sum_{s'} \mathcal{T}(s'|s,a)\left[R^a_{s,s'} + \gamma U(s')\right]\|_\infty \\
&\leq \gamma \|\max_a \sum_{s'} \mathcal{T}(s'|s,a)\left[V(s') - U(s')\right]\|_\infty \\
&\leq \gamma \max_{a,s'} \sum_{s'} \mathcal{T}(s'|s,a)\left[V(s') - U(s')\right] \\
&\leq \gamma \|V - U\|_\infty
\end{aligned}
$$

Along with the fact that $(R, \|\cdot\|_\infty)$ is a complete metric space, we can use the Banach fixed point theorem to conclude that there exists a unique optimal value function V^{π^*} for every MDP, associated to an optimal policy π^*, which can be found by a fixed point iteration scheme using the Bellman optimality operator $\mathcal{B}$.

B.2 Deriving Log-Gradients for Policy Updates

In order to optimize the performance of a policy in terms of collecting discounted rewards in an environment, instead of defining an explicit Q-function to be maximized through actions, we can instead formulate a differentiable objective function directly as a function of policy parameters θ. This is referred to as policy gradients, and uses an objective function $J(\theta)$ defined as the expectation over trajectories τ of the policy π_θ, of the sum of discounted rewards over that episode, denoted $r(\tau)$:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi}\left[r(\tau)\right] = \int \pi(\tau) r(\tau) d\tau$$

Now we wish to take the derivative w.r.t. to the policy parameters. Here we use the log-derivative trick: $f \cdot \nabla \log(f) = \nabla f$:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \int \nabla \pi(\tau) r(\tau) d\tau \\
&= \int \pi(\tau) \cdot \nabla \log \pi(\tau) \cdot r(\tau) d\tau \\
&= \mathbb{E}_{\tau\sim\pi}\left[r(\tau) \nabla \log \pi(\tau)\right]
\end{aligned}$$

Now looking at the $\log \pi(\tau)$ term, we can use the log to simplify the product of probabilities for the trajectory τ:

$$\begin{aligned}
\pi(\tau) &= P(s_0) \cdot \pi(a_0|s_0) \mathcal{T}(s_1|a_0, s_0) \cdot \pi(a_1|s_1) \mathcal{T}(s_2|a_1, s_1) \ldots \\
&= P(s_0) \cdot \prod_{t=0}^{T} \pi(a_t|s_t) \mathcal{T}(s_{t+1}|a_t, s_t) \\
\Rightarrow \log \pi(\tau) &= (s_0) + \sum_{t=0}^{T} \left(\log \pi(a_t|s_t) + \log \mathcal{T}(s_{t+1}|a_t, s_t)\right) \\
\Rightarrow \nabla \log \pi(\tau) &= \sum_{t=0}^{T} \nabla \log \pi(a_t|s_t)
\end{aligned}$$

Plugging this result back into the initial expression we can write:

$$\nabla J(\theta) = \mathbb{E}_{\tau\sim\pi}\left[r(\tau) \cdot \sum_{t=0}^{T} \nabla \log \pi(a_t|s_t)\right]$$

This result is cool because we don't actually need to know the exact environment dynamics to find the best policy. However in practice to find the expectation under all possible trajectories, we need to sample a lot from these dynamics. Sampling these trajectories is known as *Markov Chain Monte Carlo*

(MCMC), and requires averaging over many sampled trajectories to get unbiased parameter updates. However due to the $r(\tau)$ term, the variance of the updates can be quite large. Following this, there have been many improvements upon the use of policy networks, notably in actor-critic methods such as proximal policy optimization (PPO).

B.3 Properties of the Distributional Bellman Operators

Wasserstein Metric

The Wasserstein metric originates from optimal cost theory, where the optimal transport cost between two measures μ, ν is:

$$C(\mu, \nu) := \inf_{\pi \in \Pi(\mu,\nu)} \int c(x, y) d\pi(x, y)$$

This can be generalized into a metric [Villani, 2009]:

Let (χ, d) be a Polish metric space, and let $p \in [1, \infty)$. For any two probability measures μ, ν on χ, the Wasserstein distance of order p between μ and ν is defined by

$$d_p(\mu, \nu) := \left(\inf_{\pi \in \Pi(\mu,\nu)} \int_\chi c(x, y)^p d\pi(x, y) \right)^{\frac{1}{p}} = \left(\inf_{\pi \in \Pi(\mu,\nu) X, Y \sim \pi} \mathbb{E}[d(X, Y)^p] \right)^{\frac{1}{p}}$$

Where d is a distance, and $\Pi(\mu, \nu)$ is the set of all probability measures on $\chi \times \chi$ (couplings) with marginal μ and ν. This is useful as it can also be written in terms of the inverse distribution functions of the random variables [Major, 1978]:

$$\begin{aligned} d_p(X, Y) &= \left(\int_{\mathbb{R}} |F_X^{-1}(x) - F_Y^{-1}(x)|^p dx \right)^{1/p} \\ &= \int_0^1 |F_X(u) - F_Y(u)| \, du \text{ if } p = 1 \end{aligned}$$

The Wasserstein distance has the advantage of being a metric, we recall below the proofs of some useful properties from Bickel and Freedman [1981]. Let $a \in \mathbb{R}$, and X, Y, Z be independent random variables.

1. $d_p(aX, aY) \leq |a| d_p X, Y)$:

$$
\begin{aligned}
d_p(aX, aY) &= \inf_{X,Y} \mathbb{E}\big[\|aX - aY\|^p\big]^{1/p} \\
&\leq \inf_{X,Y} \mathbb{E}\big[|a|^p \|(X - Y)\|^p\big]^{1/p} \\
&= |a| \inf_{X,Y} \mathbb{E}\big[\|(X - Y)\|^p\big]^{1/p} \\
&= |a| d_p(X, Y)
\end{aligned}
$$

2. $d_p(Z + X, Z + Y) \leq d_p(X, Y)$ We prove the general case for the sum of independent r.v's :

$$
\begin{aligned}
d_p\bigg(\sum_i X_i, \sum_i Y_i\bigg) &\leq \inf_{X,Y} \mathbb{E}\big[\|\sum_i (X_i - Y_i)\|^p\big]^{1/p} \\
&\overset{(a)}{\leq} \sum_i \inf_{X,Y} \mathbb{E}\big[\|X_i - Y_i\|^p\big]^{1/p} = \sum_i d_p(X_i, Y_i)
\end{aligned}
$$

   Where (a) is Minkowski's inequality

3. $d_p(ZX, ZY) \leq \|Z\|_p d_p(X, Y)$:

$$
\begin{aligned}
d_p(ZX, ZY) &= \inf_{X,Y} \mathbb{E}\big[\|ZX - ZY\|^p\big]^{1/p} \\
&\leq \big(\|Z\|^p\big)^{\frac{1}{p}} \inf_{X,Y} \mathbb{E}\big[\|X - Y\|^p\big]^{1/p} \\
&= \|Z\|_p d_p(X, Y)
\end{aligned}
$$

Contraction of the Distributional Bellman Equation

Bellemare et al. [2017] has shown that under the maximal form of the Wasserstein metric, $\bar{d}_p(Z_1, Z_2) := \sup_{x,a} d_p(Z_1(x, a), Z_2(x, a))$, the distributional Bellman operator $\mathcal{D}^\pi$ is a contraction. They also show however, that $\mathcal{D}$ (optimal Bellman operator) is generally **not** a contraction.

$\mathcal{D}^\pi : \mathcal{Z} \to \mathcal{Z}$ is a γ -contraction in $\bar{d}_p$:

First of all we note some properties of the d_p metric whose proofs are in the previous section:

- $d_p(aU, aV) \leq |a| d_p(U, V)$
- $d_p(A + U, A + V) \leq d_p(U, V)$
- $d_p(AU, AV) \leq \|A\|_p d_p(U, V)$

We recall the definition of $\bar{d}_p$, which is itself a metric:

$$\bar{d}_p(\mathcal{T}^\pi Z_1, \mathcal{T}^\pi Z_2) = \sup_{s,a} d_p(\mathcal{T}^\pi Z_1(s,a), \mathcal{T}^\pi Z_2(s,a))$$

From the properties of d_p we have that:

$$\begin{aligned} &d_p(\mathcal{T}^\pi Z_1(s,a), \mathcal{T}^\pi Z_2(s,a)) \\ &\qquad = d_p(R(x,a) + \gamma P^\pi Z_1(s,a), R(s,a) + \gamma P^\pi Z_2(x,a)) \\ &\qquad \leq \gamma d_p(P^\pi Z_1(s,a), P^\pi Z_2(s,a)) \\ &\qquad \leq \gamma \sup_{s',a'} d_p(Z_1(s',a'), Z_2(s',a')) \end{aligned}$$

Combining with the definition of $\bar{d}_p$ we get:

$$\begin{aligned} \bar{d}_p(\mathcal{T}^\pi Z_1, \mathcal{T}^\pi Z_2) &= \sup_{x,a} d_p(\mathcal{T}^\pi Z_1(s,a), \mathcal{T}^\pi Z_2(s,a)) \\ &\leq \gamma \sup_{x',a'} d_p(Z_1(s,' a'), Z_2(s',a')) \\ &= \gamma \bar{d}_p(Z_1, Z_2) \end{aligned}$$

Hence $\mathcal{D}^\pi$ is a contraction in $\bar{d}_p$, and by Banach's fixed point theorem (given that the space of distributions is a complete space (it is)), we are able to iteratively find a unique fixed point solution to the equation $Z(s,a) \overset{D}{=} \mathcal{D}^\pi Z(s,a)$

Unfortunately there is no similar proof for the contraction of the control-setting operator $\mathcal{D}$. This does not mean however, that $\mathcal{D}$ is not a contraction on its domain, however the theoretical basis for using bellman targets is not present for this training paradigm.

B.4 Alternate Parametrization of the Return Distribution

In order to store the return distribution for updates, it has to be parametrized by a set of *statistics* (*e.g.* mean and variance for a Gaussian). Though for modelling more complex distributions, a greater amount of statistics are required.

Quantiles and Expectiles

One of the more popular statistics that have been used in recent implementations are quantiles and expectiles [Dabney et al., 2017], [Rowland et al., 2019]. Expectiles generalize the mean, in the way that quantiles generalize the median. Another way to look at them, is that these statistics are the solutions to L1 and

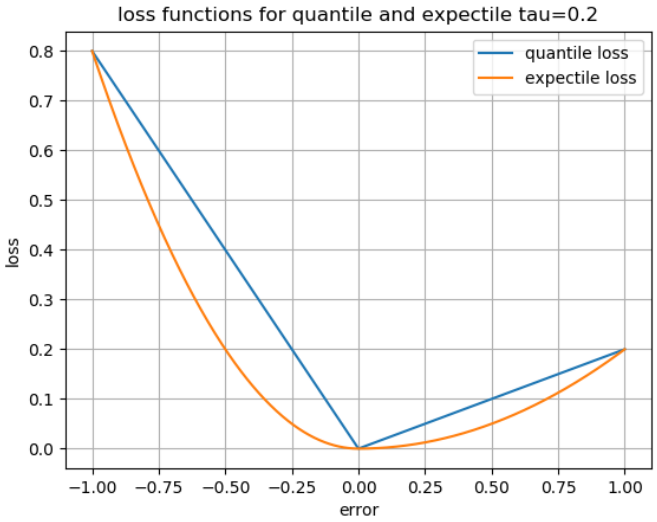


Figure B.1: Difference between quantile and expectile losses

L2 lop-sided losses for quantiles and expectiles respectively on the expected values of samples from a distribution η. In the case of quantiles, this is also known as the *pinball loss*. $\forall \kappa_i \in [0,1]$, Z is the sample from our pdf, ψ is the parameter we are solving for:

$$q_{\kappa_i}(\nu) = \arg\min_{\psi \in \mathbb{R}} \mathop{\mathbb{E}}_{Z \sim \eta} \left[|Z - \psi| \left[\mathbb{1}_{Z > \psi}(\kappa_i) + \mathbb{1}_{Z \leq \psi}(1 - \kappa_i) \right] \right]$$

$$e_{\kappa_i}(\nu) = \arg\min_{\psi \in \mathbb{R}} \mathop{\mathbb{E}}_{Z \sim \eta} \left[(Z - \psi)^2 \left[\mathbb{1}_{Z > \psi}(\kappa_i) + \mathbb{1}_{Z \leq \psi}(1 - \kappa_i) \right] \right]$$

Most commonly, the parameter $\psi = \{\psi_i, i \in [1, N]\}$ is the positions of a comb of diracs $\Pi_\psi \eta = \sum_{i=1}^{N} \delta_{\psi_i}$. Looking at the resulting cumulative distribution function, this actually corresponds to the quantiles of the parametric distribution $\Pi_\psi \eta$.

Bellman Closedness

In order to keep track, and update a fixed set of statistics which characterize the return distribution means that there must a a coherent Bellman-type equation which we use to update them [Rowland et al., 2019].

A set of statistics is defined as being *Bellman-closed* if there exists a closed-form equation of the sort:

$$\psi = \mathcal{T}\psi$$

If this is the case, then the true statistics are the solution to the fixed point equation define by that Bellman equation. However to find them through an iterating scheme we have to prove the contraction of this new $\mathcal{T}$ under some metric.

Which Statistics are Bellman-closed ?

If we take the example of a quantile-dirac parametrization for the return distribution, we actually see that the quantiles are *not* Bellman-closed. This means that we cannot learn the true parametrized distribution by passing the quantile statistics through a Bellman-like equation. This is best illustrated by the following counter-example:

Let a pdf η be parametrized by its $\frac{2i+1}{2N}$ quantiles, $i \in 1, ..., N$. i.e. $\Pi_\psi \eta = \sum_{i=1}^{N} \delta_{q(\kappa_i)}$. Since the quantile q_0 is not taken into consideration, the support of the parametrized pdf will be shrinked compared to the true one. Without any furthur prior knowledge on the distribution, any quantile parametrization will fail to capture the tail behavior.

We can see in an N-chain environment, that there will be an increase in the W1-loss when propagating the reward back. This means that the accurate reward pdf will eventually be more and more distorted the more it "travels" through the mdp.

Rowland et al. [2019] proposes that the expectation of the (sub-)gradient of the loss function at the true parameter value ψ^* is always 0, if:

(i) The true statistic ψ^* of a distribution ν satisfies $\psi^* = \text{argmin}_{\psi \in \mathbb{R}} L_k(\psi; \nu)$

(ii) The loss L_k is *affine* w.r.t. ν

Huber [2009] show that this is the case with *M-estimators* and their associated statistics. Quantile DRL is an example of this.

Stochastic Bellman Updates of Statistics

Analogously to the way a stochastic Bellman update is used when dealing with the entire distribution, we must consider the same limitations when computing statistics through the bellman equation. The proof of non-biased sample gradients on the loss function, should assure us that the pdf will converge to the correct value.

B.4.1 Sufficient statistics

Following previous work done on distributional RL, we mention the effect that increasing the number of estimated statistics has on the mean performance of agents during training.

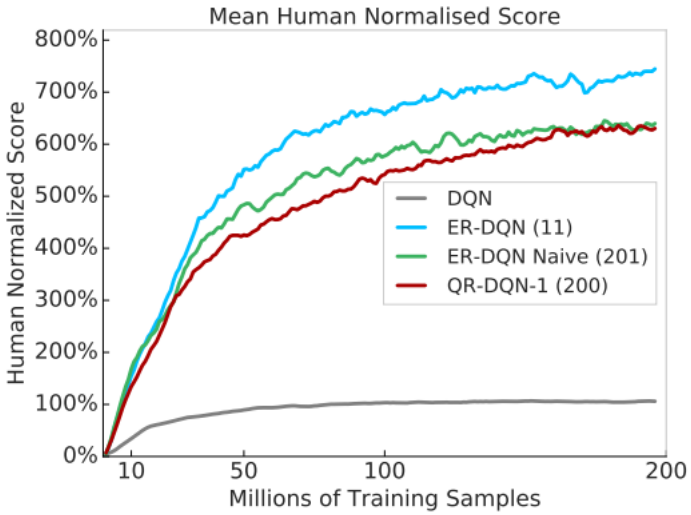


Figure B.2: Training score for various distributional agents [Rowland et al., 2019]

Figure B.2 shows that different statistics along with the amount thereof may be more applicable to learning the return distribution. In the case of expectile regression for example (ER-DQN), 11 statistics are sufficient to give a good enough representation of the value distribution function, so that the agent may take good decision with respect to it, as opposed to a higher number of statistics used in less well-adapted algorithms. This work was conducted by Rowland et al. [2019] over a suite of Atari games. As discussed in chapter 3, although these authors' work (along with others') has shown that distributional RL has the potential for improving the performance of agents in some environments, this does not take into account the modeling of stochastic dynamics which we have aimed to do in our work.

Bibliography

Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Philip Amortila, Doina Precup, Prakash Panangaden, and Marc G Bellemare. A distributional analysis of sampling-based reinforcement learning algorithms. *arXiv preprint arXiv:2003.12239*, 2020.

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pages 166–175, 06–11 Aug 2017.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the Atari human benchmark. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119, pages 507–517, Jul 2020a.

Adrià Puigdomènech Badia, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, Bilal Piot, Steven Kapturowski, Olivier Tieleman, Martin Arjovsky, Alexander Pritzel, Andrew Bolt, and Charles Blundell. Never give up: Learning directed exploration strategies. In *International Conference on Learning Representations*, 2020b.

Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *CoRR*, abs/1707.06887, 2017. URL `http://arxiv.org/abs/1707.06887`.

Marc G. Bellemare, Nicolas Le Roux, Pablo Samuel Castro, and Subhodeep Moitra. Distributional reinforcement learning with linear function approx-

imation. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, 2019.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. 2019. URL `http://arxiv.org/abs/1912.06680`.

Julian Bernhard, Stefan Pollok, and Alois Knoll. Addressing inherent uncertainty: Risk-sensitive behavior generation for automated driving using distributional reinforcement learning. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 2148–2155, 2019.

Peter J Bickel and David A Freedman. Some asymptotic theory for the bootstrap. *The annals of statistics*, pages 1196–1217, 1981.

M. Bouton, A. Nakhaei, K. Fujimura, and M. J. Kochenderfer. Safe reinforcement learning with scene decomposition for navigating complex urban environments. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 1469–1476, 2019.

Sebastian Brechtel, Tobias Gindele, and Rüdiger Dillmann. Probabilistic decision-making under uncertainty for autonomous driving using continuous pomdps. In *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 392–399, 2014.

Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, pages 1–43, 2012.

Yinlam Chow, Aviv Tamar, Shie Mannor, and Marco Pavone. Risk-sensitive and robust decision-making: a cvar optimization approach. *CoRR*, abs/1506.02188, 2015. URL `http://arxiv.org/abs/1506.02188`.

William R. Clements, Benoît-Marie Robaglia, Bastien Van Delft, Reda Bahi Slaoui, and Sébastien Toth. Estimating risk and uncertainty in deep reinforcement learning. *arXiv Preprint*, 2019.

Alexander G. Cunningham, Enric Galceran, Ryan M. Eustice, and Edwin Olson. Mpdm: Multipolicy decision-making in dynamic, uncertain environments for autonomous driving. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1670–1677, 2015.

Will Dabney, Mark Rowland, Marc G. Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. *CoRR*, abs/1710.10044, 2017. URL `http://arxiv.org/abs/1710.10044`.

Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. *CoRR*, abs/1806.06923, 2018. URL `http://arxiv.org/abs/1806.06923`.

Gal Dalal, Krishnamurthy Dvijotham, Matej Vecerik, Todd Hester, Cosmin Paduraru, and Yuval Tassa. Safe exploration in continuous action spaces. *arXiv preprint*, 2018.

Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 5, pages 271–278. Morgan-Kaufmann, 1993.

Stefan Depeweg, Jose-Miguel Hernandez-Lobato, Finale Doshi-Velez, and Steffen Udluft. Decomposition of uncertainty in Bayesian deep learning for efficient and risk-sensitive learning. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80, pages 1184–1193, Jul 2018.

Hannes Eriksson and Christos Dimitrakakis. Epistemic risk-sensitive reinforcement learning. In *Proceedings of European Sympiosium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2020.

Ben Eysenbach, Russ R Salakhutdinov, and Sergey Levine. Search on the replay buffer: Bridging planning and reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 32, 2019a.

Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. In *Proceedings of the International Conference on Learning Representations*, 2019b.

Angelos Filos, Panagiotis Tigkas, Rowan Mcallister, Nicholas Rhinehart, Sergey Levine, and Yarin Gal. Can autonomous vehicles identify, recover from, and adapt to distribution shifts? In *Proceedings of the 37th International Conference on Machine Learning*, volume 119, pages 3145–3153, 2020.

Jaime F. Fisac, Eli Bronstein, Elis Stefansson, Dorsa Sadigh, S. Shankar Sastry, and Anca D. Dragan. Hierarchical game-theoretic planning for autonomous vehicles. In *2019 IEEE Internation Conference on Robotics and Automation (ICRA)*, pages 9590–9596, 2018.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

Hado Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. In *AAAI Fall Symposia*, 2015.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. URL `http://arxiv.org/abs/1710.02298`.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

Carl-Johan Hoel, Katherine Driggs-Campbell, Krister Wolff, Leo Laine, and Mykel J. Kochenderfer. Combining planning and deep reinforcement learning in tactical decision making for autonomous driving. *IEEE Transactions on Intelligent Vehicles*, 5:294–305, 2020a.

Carl-Johan Hoel, Krister Wolff, and Leo Laine. Tactical decision-making in autonomous driving by reinforcement learning with uncertainty estimation. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 1563–1569, 2020b.

Peter J. Huber. *Robust Statistics*, pages 1248–1251. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

Constantin Hubmann, Jens Schulz, Marvin Becker, Daniel Althoff, and Christoph Stiller. Automated driving in uncertain environments: Planning with interaction and uncertain maneuver prediction. *IEEE Transactions on Intelligent Vehicles*, 3:5–17, 2018.

Christoph Killing, Adam Villaflor, and John M. Dolan. Learning to robustly negotiate bi-directional lane usage in high-conflict driving scenarios. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 8090–8096, 2021.

Louis Kirsch, Sjoerd van Steenkiste, and Jürgen Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. *arXiv preprint arXiv:1910.04098*, 2019.

Mykel J. Kochenderfer, Christopher Amato, Girish Chowdhary, Jonathan P. How, Hayley J. Davison Reynolds, Jason R. Thornton, Pedro A. Torres-Carrasquillo, N. Kemal Üre, and John Vian. *Decision Making Under Uncertainty: Theory and Application*. The MIT Press, 1st edition, 2015.

Saurabh Kumar, Aviral Kumar, Sergey Levine, and Chelsea Finn. One solution is not all you need: Few-shot extrapolation via structured maxent rl. In *Advances in Neural Information Processing Systems*, volume 33, pages 8198–8210, 2020.

Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, R. Howard, Wayne Hubbard, and Lawrence Jackel. Handwritten digit recognition with a back-propagation network. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989.

Clare Lyle, Pablo Samuel Castro, and Marc G. Bellemare. A comparative analysis of expected and distributional reinforcement learning. *CoRR*, abs/1901.11084, 2019. URL `http://arxiv.org/abs/1901.11084`.

Marlos C. Machado, Marc G. Bellemare, and Michael Bowling. A Laplacian framework for option discovery in reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pages 2295–2304, 2017.

Péter Major. On the invariance principle for sums of independent identically distributed random variables. *Journal of Multivariate analysis*, 8(4):487–517, 1978.

Rowan McAllister and Carl Edward Rasmussen. Data-efficient reinforcement learning in continuous state-action gaussian-pomdps. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Alberto Maria Metelli, Amarildo Likmeta, and Marcello Restelli. Propagating uncertainty in reinforcement learning via wasserstein barycenters. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 4333–4345. Curran Associates, Inc., 2019.

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.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

Tetsuro Morimura, Masashi Sugiyama, Hisashi Kashima, Hirotaka Hachiya, and Toshiyuki Tanaka. Nonparametric return distribution approximation for reinforcement learning. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 799–806, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

Ofir Nachum, Shixiang (Shane) Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 31, 2018.

Louise Pryor and Gregg C. Collins. Planning for contingencies: A decision-based approach. *Journal for Artificial Intelligence Research*, 4:287–339, 1996.

Nicholas Rhinehart, Jeff He, Charles Packer, Matthew A. Wright, Rowan McAllister, Joseph E. Gonzalez, and Sergey Levine. Contingencies from observations: Tractable contingency planning with learned behavior models. In *2021 IEEE International Conference on Robotics and Automation*, pages 13663–13669, 2021.

Clément Romac and Vincent Béraud. Deep recurrent q-learning vs deep q-learning on a simple partially observable markov decision process with minecraft. *CoRR*, abs/1903.04311, 2019. URL `http://arxiv.org/abs/1903.04311`.

Mark Rowland, Robert Dadashi, Saurabh Kumar, Rémi Munos, Marc G. Bellemare, and Will Dabney. Statistics and Samples in Distributional Reinforcement Learning. *arXiv e-prints*, art. arXiv:1902.08102, Feb 2019.

Tom Schaul, Dan Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37, pages 1312—-1320, 2015.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *Proceedings of the International Conference on Learning Representations*, 2016.

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.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

Richard Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstaction in reinforcement learning. *Artificial Intelligence*, 1999.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Veritasium. Would you take this bet?, 2015. URL `https://www.youtube.com/watch?v=vBX-KulgJ1ot=247s`.

Cédric Villani. *The Wasserstein distances*, pages 93–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

Zhongwen Xu, Hado P van Hasselt, and David Silver. Meta-gradient reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 31, 2018.

Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE Access*, 8:58443–58469, 2020.

Shizhe Zang, Ming Ding, David Smith, Paul Tyler, Thierry Rakotoarivelo, and Mohamed Ali Kaafar. The impact of adverse weather conditions on autonomous vehicles: Examining how rain, snow, fog, and hail affect the performance of a self-driving car. *IEEE Vehicular Technology Magazine*, PP:1–1, 03 2019.

Jesse Zhang, Haonan Yu, and Wei Xu. Hierarchical reinforcement learning by discovering intrinsic options. In *International Conference on Learning Representations*, 2021.

Pengfei Zhu, Xin Li, and Pascal Poupart. On improving deep reinforcement learning for pomdps. 2017. URL `http://arxiv.org/abs/1704.07978`.