Multi-Objective Deep Reinforcement Learning

Hossam Mossalam

St Cross College

University of Oxford

A dissertation submitted for the degree of

Masters of Science

Trinity 2016

Acknowledgements

I would like to equally thank both of my supervisors, Dr Diederik Roijers and Mr Yannis Assael for their excellent guidance and mentorship during the course of this work. Without their constant support and assistance, none of this work would have been done. I am very grateful to their dedication to teach me how to do proper research and teaching me how to think rather than tell me what to do. I am grateful to them for the amount of time they have spent helping me without any sign of annoyance.

I would like to thank my mentors Prof. Haythem Ismail, Prof. Amr ElMougy, Prof. Ahmad Gamal, Prof. Hazem Abbas, and Prof. Slim Abdennadher for all the mentorship they have provided me with.

Also, I am grateful to Ignacio, Tony, Nick, Antonio, Arni, Mikesh, Siqi and Francisco for making this year at Oxford an unforgettable one.

Finally, I would like to thank my family for their faith in me and their continuous support.

Abstract

Sequential decision-making problems with multiple objectives arise naturally in practice and pose unique challenges for research in reinforcement learning. While most of the problems are multi-objective in nature, reinforcement learning has largely focused on the single-objective setting. The main difference between the single-objective setting and the multi-objective setting is that the optimal solution for the single-objective setting is one policy, while for the multi-objective setting it is coverage set, i.e., a set containing at least one optimal policy (and policy value vector) for each possible utility function that a user might have. In the recent years, there has been an increasing interest in the multi-objective planning paradigm. The aim of this dissertation is to incorporate the highly successful deep learning techniques in the multi-objective learning setting.

We use DQN; a deep learning based reinforcement learning algorithm with OLS; a multi-objective planning algorithm to build a generic multi-objective learning algorithm. The resulting algorithm which we call DOL successfully finds a solution that is very close to the optimal solution for all experiments. We also extend DOL with the ability to reuse already learnt policies to speed up the learning of new ones leading to our two extension algorithms DOL-R Full and DOL-R Partial. To the extent of our knowledge, DOL and its two extensions are believed to be the first multi-objective deep reinforcement learning algorithms.

Contents

List of Symbols

θ	Q-network weights
θ'	Target Q-network weights
Π^m	Set of all Policies for MOMDP m
$\mathcal{A}$	Action Space
$\mathcal{CCS}$	Convex Coverge Set
$\mathcal{CH}$	Convex Hull
CNN	Convolutional Neural Network
$\mathbb{CS}(\Pi^m)$	Coverage Set of possibly stochastic and non-stationary policies for MOMDP m
DNN	Deep Neural Network
DOL	Deep OLS Learning
DOL-R	Deep OLS Learning with Reuse
DOL-R Full	Deep OLS Learning with Full Reuse
DOL-R Partial	Deep OLS Learning with Partial Reuse
DQN	Deep Q-Network
f	Scalarisation Function
g	Non-linearity function for Neural Networks
MDP	Markov Decision Process
MLP	Multilayer Perceptron
MOMDP	Multi-objective Markov Decision Process
n	Number of objectives for an MOMDP
OLS	Optimistic Linear Support

$\mathcal{P}$	State transition probabilities
$\mathcal{PCS}$	Pareto Coverage Set
$\mathbb{PF}(\Pi^m)$	Pareto Front set
Q	Priority queue of OLS corner weights to be explored
$\mathcal{Q}(s, a, \theta)$	Approximated action value function
$Q^\pi(s, a)$	Action Value function
$Q^*(s, a)$	Optimal Action Value function
$\mathcal{R}$	Reward function
R_t	Return
$ReLU$	Rectifier Linear Unit
RL	Reinforcement Learning
$\mathcal{S}$	Partial Convex Coverge Set
s_t^a	Agent state
s_t^e	Environment state
$\mathcal{T}$	Time
$\mathbb{U}(\Pi^m)$	Undominated set of policies for MOMDP m
V	Set of all Possible value vectors
V^π	State Independent Value vector
$V^\pi(s)$	State Value function
$V^*(s)$	Optimal State Value function
$V^*_{\mathcal{CCS}}(w)$	Maximal attainable scalarised value for scalarisation weight w using value vectors in $\mathcal{CCS}$
$V^*_{\overline{\mathcal{CCS}}}(w)$	Maximal attainable scalarised value for scalarisation w using value vectors in $\overline{\mathcal{CCS}}$
$V^*_{\mathcal{S}}(w)$	Maximal attainable scalarised value for scalarisation weight w using value vectors in $\mathcal{S}$
V^π_w	Scalarised Policy value under scalarisation weight w
$\mathcal{W}$	OLS Explored corner weights
w	Scalarisation Weight

List of Figures

List of Tables

List of definitions

Chapter 1

Introduction

1.1 Motivation

Sequential decision-making is a crucial task that humans face on a daily basis while interacting with the environment. The core feature of this type of tasks is that decisions or actions may have long term consequences. This delay in witnessing the consequences implies that in some situations it is better to compromise by giving up an immediate gain for a better gain on the long-term. Sequential decision-making problems arise naturally in everyday life and even though humans have nurtured the skills required to solve this kind of problems, there are still problems which are very complicated and time-consuming for having a human dedicated to solving it and sometimes it is even impossible for humans to solve, such as planning infrastructural maintenance with multiple contractors [27], teaching robots how to adapt to damage [7], and learning behaviour for soccer robots [32]. Therefore, it is absolutely essential to have automatic sequential decision-making algorithms to solve these problems with minimal human effort. This type of problems has been extensively studied in the fields of Artificial Intelligence [1, 17, 20, 30], Robotics [7, 16, 32] and Operations Research [12] and an enormous amount of research has been devoted to finding algorithms that can solve problems formulated as a sequential decision-making problem.

There are two main frameworks for solving this kind of problems, namely, Planning and Reinforcement Learning (RL). Planning assumes that a model of the environment in which the agent is to work within is known to the agent. The model

describes the dynamics that govern the environment. In the planning setting, the model is assumed to have full knowledge of the environment. Therefore, the agent can plan optimally without any interaction with the environment. In contrast, in RL, this substantial assumption that a model of the environment is known is dropped, and the agent is required to learn directly by trial and error from interacting with the environment. And by doing so, it improves its behaviour according to what it experiences while learning.

The agent bases its decisions on the information it has about the state in which it is in, so the state representation is crucial for developing accurate RL and Planning algorithms. And while many problems have a small state space that can be solved with the conventional methods of reinforcement learning like Monte Carlo and TD(λ) methods [37], most interesting problems have a very large state space and/or action space which makes them much more challenging to solve. This is because these conventional methods lack the generalisation power needed to solve them. Recently, there has been a great interest in using techniques from the deep learning paradigm as function approximators to address this problem and to help generalise for very large state and action spaces that could not have been solved otherwise, with the most famous algorithm being Deep Q-Network (DQN) [20, 21].

In RL and Planning , the goal of the agent is to maximise the expected future reward from any given state. A reward is a feedback signal from the environment to the agent which indicates how well the agent is performing at that time step. Although most of the sequential decision-making problems have multiple objectives, and it is both easier and more natural to express the multiple objectives with multiple rewards, the vast majority of the research addressed these problems within the single-objective setting by using a scalar reward rather than a vector of rewards. However, there has been a growing interest in the multi-objective setting to solve some of the problems which cannot be represented as single-objective problems for reasons related to the impossibility or infeasibility or undesirability of using scalar reward values [23, 25].

The multi-objective setting takes a different route to solving reinforcement learning problems in which the target is not to find an optimal policy, but to find an optimal policy for everything the user might need. But what is meant by the

statement "everything the user might need"? In the multi-objective setting, the concept of optimal policy is different from that in the single-objective setting. This is because, in the multi-objective setting, there might be a situation in which the learning algorithm might have to pick between maximising one of the objectives or the other or finding a policy that maximises neither, but constitutes a good compromise. Therefore, the solution in a multi-objective setting is called a coverage set, i.e., a set of policies that contains one optimal solution for each possible preference w.r.t. the objectives that a user might have [23, 25].

In the multi-objective domain, there are two classes of algorithms, namely, outer-loop algorithms and inner-loop algorithms. The inner-loop class is the one in which a single-objective solver is modified to solve multi-objective problems. On the other hand, the outer-loop algorithms use a single-objective solver without any changes as a subroutine to solve the multi-objective problem. One major advantage of the outer-loop approach is that any single-objective solver can be used without any re-engineering, this means that any improvement in the single-objective solvers yields an improvement in the multi-objective solvers. Another great advantage of the outer-loop methods is that they are anytime algorithms (i.e. they do not have to wait for termination to produce results).

Optimistic Linear Support (OLS) is a recent state of the art outer-loop algorithm that is being used throughout this dissertation. OLS uses a single-objective solver to incrementally find optimal policies for all possible prioritisations of the objectives until all prioritisations have been covered. The reason OLS is preferred over other outer-loop algorithms is that it intelligently explores the space of all policies leading to a very efficient algorithm for finding the coverage set. It also inherits the quality guarantees of the single-objective solver, i.e., if the single-objective solver is exact, OLS is exact. Moreover, in contrast to other outer-loop algorithms, OLS is guaranteed to converge, as long as the single-objective solver used is guaranteed to converge.

1.2 Scope and Contributions

OLS has been mainly used in the Planning framework, and in this dissertation, we investigate the potential of OLS in the RL paradigm to solve multi-objective

RL problems with very big state spaces. Furthermore, inspired by earlier work on MOPOMDP planning, we investigate the potential benefits of reusing already learnt policies to speed up the learning for new policies.

The first aim of this dissertation is to design a generic outer-loop based algorithm capable of solving multi-objective RL problems and enables the usage of highly successful deep learning techniques in a multi-objective setting. Secondly, we aim to complement the generic multi-objective learning algorithm with the ability to use already learnt policies to speed up learning of new ones. Achieving our goal would pave the way to multi-objective RL and would benefit all fields associated with solving RL problems.

1.3 Dissertation Structure

This dissertation consists of three main parts. The first part which consists of chapters 1 and 2 is of introductory nature. It covers the background theory for Reinforcement Learning and Planning in both single-objective and multi-objective settings. It also explains neural networks which are the underlying algorithm for DQN; the single-objective solver used throughout this dissertation. Furthermore, it covers the needed background for finding the optimal solutions in both single-objective and multi-objective settings. Last but not least, it presents OLS and DQN which are the algorithms integrated together to build our general multi-objective solver.

The second part which consists of chapters 3 and 4, introduces the proposed methodologies used to solve multi-objective RL problems. Chapter 4 starts by explaining how DQN was transformed to be OLS-compliant, how reuse of learnt policies was done using DQN as the single-objective solver, and how OLS was employed in the learning setting. It also covers the Deep Sea Treasure world problem which is the benchmark problem used for all the experiments in this dissertation. Additionally, it describes the recommended error metric used to evaluate multi-objective solvers. Chapter 4 provides a detailed explanation of the experiments that have been used to evaluate our proposed algorithm Deep OLS Learning (DOL) and its extensions Deep OLS Learning with Full Reuse (DOL-R

Full) and Deep OLS Learning with Partial Reuse (DOL-R Partial). It also provides a thorough discussion of the achieved results.

The last part consisting of chapter 5 concludes the dissertation by providing a summary of the achieved results and comparing them with the initial aims of the project, as well as, the contributions of the current project. Moreover, it provides a list of suggested possible extensions to the current work that can be considered for future work.

Chapter 2

Background

Reinforcement Learning (RL) and Planning are the two main frameworks for solving sequential decision-making problems. However, planning algorithms depend on the existence of a model that describes the environment's dynamics. Alternatively, RL does not make any assumption about the environment or the dynamics of the environment other than it is Markov. This chapter provides the theoretical background needed to describe the proposed algorithms to solve multi-objective RL problems. It starts by providing a brief background to the RL and Planning algorithms in both the single-objective and the multi-objective settings. It then introduces neural networks which are the underlying structure of DQN; the single-objective solver used throughout this dissertation. Last but not least, some of the methods used to solve MDPs and MOMDPs are presented, and the two main algorithms used throughout this dissertation; OLS and DQN are explained in detail.

2.1 RL and Planning

This section is a brief introduction to the components of RL and Planning algorithms. It also explains the differences between solutions to problems modelled in the single-objective setting and problems modelled in the multi-objective setting. Lastly, it introduces value functions which are an essential component of many single-objective RL and Planning algorithms. The multi-objective part in this section especially the definitions in section 2.1.5 are based on the PhD thesis of D.

M. Roijers [23].

2.1.1 Agent and Environment

In standard RL and Planning problems, the agent tries to learn by interacting with either a model of the environment or the environment itself. The agent interacts with the environment using actuators and collects information using sensors. The way in which the agent is allowed to interact with the environment and change its state is only by performing actions at each time step and the environment responds by giving back rewards and observations to the agent. The same procedure takes place in the planning setting except that the environment is replaced by its model. In typical RL and Planning algorithms, the agent is directed towards achieving some goal by being awarded rewards that indicate the effect of its actions and the agent tries to maximise these rewards in the long run.

2.1.2 Rewards

Rewards are the signals that the agent receives from the environment or its model indicating how well it is performing at any time step $t \in \text{Time}$ ($\mathcal{T}$). The agent's ultimate goal is to obtain as many rewards as possible in the long run. So far, sequential decision-making problems have mostly been addressed within the single-objective setting. In the single-objective setting, the rewards are single scalar values indicating the overall desirability of the state without explicit consideration of each objective by itself. This is because some researchers argue that it is not necessary to model problems as having multiple objectives and that the single-objective setting is sufficient. One reasoning behind this claim is that the purpose of rewards as a concept is to be able to compare actions and select the best amongst them. Thus implying the existence of a scale that can be used to compare them. One famous hypothesis which is due Prof. Richard Sutton states:

> That all of what we mean by goals and purposes can be well thought of as maximisation of the expected value of the cumulative sum of a received scalar signal (reward). [34].

This hypothesis implies two main things, the first is that there is no need to

model the multiple objectives' rewards explicitly, and the second is that for any multi-objective problem, there is always a single-objective problem that models the multi-objective problem with a single scalar reward value that incorporates the rewards for all objectives in the multi-objective variant.

The second implication further entails that a conversion between any multi-objective problem to an equivalent single-objective problem is always possible. According to [25], this conversion has to comprise of two stages; the first is a scalarisation step. In which a Scalarisation Function (f) and a Scalarisation Weight (w) are used to convert the vector of rewards at any given state to a scalar value.

Definition 2.1 (Scalarisation Weight)**.** For an MOMDP m with n objectives, a Scalarisation Weight (w) is an n dimensional vector with the constraints that $\forall_i w_i \geq 0 \wedge \sum_i w_i = 1$.

Definition 2.2 (Scalarisation Function)**.** For an MOMDP m with n objectives and Scalarisation Weight (w), a Scalarisation Function (f) is a function that maps a multi-objective value of a policy π of a decision problem; V^π, to a scalar value V_w^π

$$V_w^\pi = f(V^\pi, w)\,.$$

The second step for such conversion is constructing the equivalent single-objective decision process using the scalarised rewards. This step can be converting a multi-objective model into a single-objective one with the scalarised rewards.

However, as argued in [23, 25], there are three main motivating scenarios for studying multi-objective decision processes explicitly for which the single-objective setting is inadequate. The first one, the Unknown Weights scenario, is when w required for f is unknown during the learning or planning stage, but would be known when executing the learnt policy. The second is Decision Support scenario, which is the case when the whole scalarisation phase is infeasible due to either the difficulty of specifying w or the difficulty of specifying f or even both. The third scenario, the Known Weights scenario, is when scalarisation is feasible, but constructing a single-objective decision process equivalent to the multi-objective one is hard or is undesirable.

These three cases provide concrete reasons for studying multi-objective sequential decision-making problems. For the first scenario, the use of f is impossible during the learning or planning stage because w is unknown. One example of this scenario is when an agent is trying to go from position A to position B, and it has multiple possible routes to use each with a different degree of compromise between time and fuel cost. For example, route A is much longer than route B, so route A has a higher fuel cost, but route B is much more crowded, so it takes longer time. This situation forces any RL or Planning algorithm to do the whole task of learning or planning with the multiple rewards until the weights that are supposed to be used during the execution phase have been figured out. Only then can the agent pick the right policy which is optimal for the weights acquired. In the Decision Support case, the task of scalarisation is infeasible. One reason would be that it is difficult to specify w. This usually happens when the degree of compromise is not easily determined, for the previous example, if route C has beautiful scenery, what degree of compromise between time, fuel cost and beautiful scenery is most suitable? There are many different views, and it is very hard to agree on a weight that fits all opinions. As for f, which function would best describe the actual preferences of the user? Is a linear function always adequate? These are some of the questions that make specifying f hard. Another detail is that the choice of f has a major impact on the conversion of the multi-objective model to the single-objective model. For example, if f is nonlinear, then the single-objective model will not have additive returns making it much harder to solve. So, if f is known to be nonlinear, it might be highly undesirable to use it due to the increased difficulty of solving the generated single-objective decision problem. This is exactly the grounds for the third situation, Known Weights Scenario.

2.1.3 Agent State and Environment State

In RL and Planning problems, starting from some start state in the State Space ($\mathcal{S}$), the agent tries to reach a goal state by performing some sequence of actions which are directed towards maximising the expected future reward. However, the agent's perception of the environment can be very different from that of the actual state. Therefore, it is crucial to distinguish between two types of states,

namely, Environment state (s_t^e) and Agent state (s_t^a). As defined in [29], the environment state is the actual correct state that describes the internal structure and the dynamics of the environment, while, the agent state is the agent's internal representation of the outside environment. All RL and Planning algorithms use s_t^a because s_t^e is not accessible by the agent. Typically, s_t^a is some function of the observations, actions and rewards experienced by the agent. The relation between the s_t^e and s_t^a is one important aspect when solving RL and Planning problems. When s_t^a is the same as s_t^e, or the agent can access all the aspects it needs from s_t^e to behave optimally, this is considered a full observability problem in which the agent knows everything that is happening in the environment and would affect its decisions, so there is no uncertainty in describing the state. On the other hand, if s_t^a is different from s_t^e, meaning that the information available to the agent is restricted, usually a subset of the true s_t^e, this is considered a partial observability problem, implying that there is some degree of uncertainty about the state of the world perceived by the agent. Throughout this dissertation, the notion of environment state and agent state is dropped because the experiments are run with full observability. Therefore, there is no difference between them.

2.1.4 Markov Decision Processes

There are many different ways for representing the environment. However, MDPs and variants of MDPs are the most famous way of representing models of the environment. For example, there are Continuous state MDPs which handle problems with infinite states. Also, a superclass of MDPs is Partially Observable MDPs which is used to handle the problems in which the agent is performing in a partially observable environment. In this dissertation, we restrict ourselves to MDPs and MOMDPs. The following section describes MDPs and MOMDPs and states the similarities and differences between them.

2.1.4.1 MDPs and MOMDPs

MDP is one of the most used methods to model problems for RL and Planning. In MDPs, the problem being considered is assumed to be Markov, meaning that

the probability of going from the current state to any next state depends only on the current state and not on any other previous state.

MDPs and MOMDPs are defined by a six-tuple ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$, γ, μ)

- $\mathcal{S}$ is a finite set of states,
- $\mathcal{A}$ is a finite set of actions,
- $\mathcal{P}$ is a transition function representing the probability of transitioning from any state s_t to any other state s_{t+1} under a certain action a,

$$\mathcal{P}^{a}_{s_t\, s_{t+1}} = \Pr(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a)\,, \tag{2.1}$$

- $\mathcal{R}$ is a reward function that represents the expected immediate reward that the agent would receive after taking action a at state s_t

$$\mathcal{R}^{a}_{s_t} = \mathcal{R}(s_t, a) = \mathbf{E}[r_{t+1} | S_t = s_t, A_t = a]\,, \tag{2.2}$$

- γ is a discount factor $\gamma \in [0, 1)$ which is used to quantify the trade-off between long-term and short-term rewards, and
- μ is a function over the states representing the probability of a certain state being the initial state

$$\mu(s) = \mathbf{E}[S_0 = s]\,. \tag{2.3}$$

The difference between MDPs and MOMDPs is that for an MDP, $\mathcal{R}$ returns a single scalar value, while for an MOMDP, $\mathcal{R}$ returns a vector of rewards with one scalar value per objective. In both MDP and MOMDP settings, the aim of the agent is to maximise the expected total future reward from any state. This quantity is typically referred to as the Return (R_t) which is defined as

$$R_t = r_{t+1} + \gamma r_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+1+i} \quad \text{, where } r_k \text{ is } \mathcal{R}^{a_k}_{s_k}\,. \tag{2.4}$$

The use of discount factor (γ) has multiple benefits, the most important of them are to keep the math bounded, model the uncertainty about the future and avoid infinite cycles [29].

2.1.5 Policies and Solution Concepts

An agent whether in an MDP or an MOMDP usually uses a policy to determine how to act in any state. The Policy (π) is a function that defines the behaviour of the agent within the environment. So, for each pair of state and action (s, a), the policy defines a probability of taking that action a at state s

$$\pi(a, s) = \Pr(A = a | S_t = s)\,. \tag{2.5}$$

The policy can be learnt explicitly or implicitly using value vectors which are explained in section 2.1.6. In the single-objective setting, it is sufficient to find a single optimal deterministic and stationary policy, however, in the multi-objective setting, a set of policies is needed for an optimal solution. This section points out different classifications of policies and solution concepts for MDPs and MOMDPs and the major differences between optimal solutions for the single-objective setting and optimal solutions for the multi-objective setting.

There are multiple classifications of policies. One classification is whether the policy is deterministic or stochastic.

Definition 2.3 (Deterministic Policy)**.** A policy π is Deterministic if and only if, for each state s, it defines exactly one action a to be taken with probability of 1 and all other actions have the probability zero of being taken:

$$\text{Deterministic}(\pi) \implies \forall_{s\in\mathcal{S}}\left(\exists_{a\in\mathcal{A}}\,\pi(a, s) = 1 \wedge \forall_{a'\in A\wedge a'\neq a}\,\pi(a', s) = 0\right).$$

Definition 2.4 (Stochastic Policy)**.** A policy π is Stochastic if and only if, it has at least one state s, for which there are multiple actions with probabilities greater than 0:

$$\text{Stochastic}(\pi) \implies \exists_{s\in\mathcal{S}\wedge a,a'\in\mathcal{A}}\,\pi(a, s) > 0 \wedge \pi(a', s) > 0\,.$$

Another classification between policies is whether a policy is stationary or non-stationary.

Definition 2.5 (Stationary Policy)**.** A policy π is Stationary if and only if, it is constant over all time steps:

$$\text{Stationary}(\pi) \rightarrow \forall_{t,t'\in\mathcal{T}}\pi^t = \pi^{t'},$$

where π^k refers to the policy π at time step k.

Definition 2.6 (Non-stationary Policy)**.** A policy π is Non-stationary if and only if, there are at least two time steps for which the policy is not the same:

$$\text{Non-stationary}(\pi) \rightarrow \exists_{t,t' \in \mathcal{T}} \pi^t \neq \pi^{t'},$$

where π^k refers to the policy π at time step k.

In the single-objective setting, it is sufficient to find one optimal deterministic and stationary policy to solve a problem. However, the notion of optimality is different when in the context of MDP from when in the context of MOMDP [23]. This is because in the single-objective setting there is a complete ordering between policies over all states, i.e., a policy π is better than, worse than, or equal to policy π' on any state $s \in \mathcal{S}$. But, in the multi-objective setting there is no such ordering because for state s $\in \mathcal{S}$ and for i^{th} and j^{th} objectives we can have $\mathbf{V}_i^{\pi}(s) > \mathbf{V}_i^{\pi'}(s)$ and at the same time $\mathbf{V}_j^{\pi}(s) < \mathbf{V}_j^{\pi'}(s)$. Therefore, we need a way to compromise between the objectives, and we need to find a policy for every possible compromisation. The notion of value vectors will be explained in the following section, but to this point, $\mathbf{V}^{\pi}$ can be thought of as a vector representing how good it is to follow policy π for all objectives.

The lack of a complete ordering of policies over the states complicates the notion of optimality in the multi-objective setting. So, we use a taxonomy from [25] that classifies the problems according to what constitutes an optimal solution for each of them based on three factors. The first factor is whether the problem needs only a single policy to be solved optimally or a set of optimal policies. This depends on which class of problems the problem lies within, Unknown Weights, Decision Support or Known Weights. The second factor is whether f is linear or monotonically increasing and the third factor is whether the end user allows for stochastic policies or not.

The type of f is an important factor that determines what constitutes an optimal solution. There are two classes for f, Linear functions and Monotonically increasing functions. The reason that only monotonically increasing functions are studied other than linear functions is that they commit to the constraint that if the difference between two value vectors is only an increase in one of the objectives' rewards, then the scalarised value should increase as well, which is considered to

be a minimal and rational constraint. In the case of monotonically increasing f, the vectorised rewards are still additive, but the scalarised version is not. For this reason, the learning or planning algorithm needs to use either the multi-objective problem intact or convert it to a complicated single-objective problem to overcome the problem of non-additive returns and then solve it. The type of the problem in hand determines what an optimal solution is. Therefore, for the multi-objective setting, we consider the following solution concepts.

Definition 2.7 (Undominated Policies)**.** For an MOMDP m and a Scalarisation Function (f), the set of all undominated policies $\mathbb{U}(\Pi^m)$, is the subset of all possible policies Π^m for m for which there exists a Scalarisation Weight (w) for which the scalarised value is maximal:

$$\mathbb{U}(\Pi^m) = \{\pi : \pi \in \Pi^m \wedge \exists_w \forall_{\pi' \in \Pi^m} V_w^\pi \geq V_w^{\pi'}\}\,.$$

$\mathbb{U}(\Pi^m)$ is enough to solve MOMDP m but, it contains far more policies than necessary as it can have multiple policies which are only optimal for the same range of scalarisation weights. We can get rid of this redundancy by using a subset of $\mathbb{U}(\Pi^m)$, namely, Coverage Set $\mathbb{CS}(\Pi^m)$.

Definition 2.8 (Coverage Set)**.** For an MOMDP m and a Scalarisation Function (f), a set $\mathbb{CS}(\Pi^m)$ is a coverage set if it is a subset of $\mathbb{U}(\Pi^m)$ and if, for every w, it contains a policy with maximal scalarised value:

$$\mathbb{CS}(\Pi^m) \subseteq \mathbb{U}(\Pi^m) \wedge \forall_w \exists_{\pi \in \mathbb{CS}(\Pi^m)} \forall_{\pi' \in \Pi^m} V_w^\pi \geq V_w^{\pi'}\,.$$

A major difference between $\mathbb{CS}(\Pi^m)$ and $\mathbb{U}(\Pi^m)$ is that if two policies π and π' have the same scalarised values (i.e. $V_w^\pi = V_w^{\pi'}$), but different value vectors $\mathbf{V}^\pi$ and $\mathbf{V}^{\pi'}$, $\mathbb{U}(\Pi^m)$ would contain both of them, but $\mathbb{CS}(\Pi^m)$ would contain only one of them. In the case of linear f, the scalarised value of any policy can be written as the dot product between the w and the vectorised value vector

$$V_w^\pi = w \cdot V^\pi\,. \tag{2.6}$$

So, the definitions of $\mathbb{U}(\Pi^m)$ and $\mathbb{CS}(\Pi^m)$ can be redefined in terms of the dot product as follows:

Definition 2.9 (Convex Hull)**.** For an MOMDP m, the Convex Hull ($\mathcal{CH}$) is the subset of Π^m for which there exists a w for which the linearly scalarised value is maximal:

$$\mathcal{CH}(\Pi^m) = \{\pi : \pi \in \Pi^m \wedge \exists_w \forall_{\pi' \in \Pi^m} w \cdot V^\pi \geq w \cdot \mathbf{V}^{\pi'}\}\,.$$

Definition 2.10 (Convex Coverge Set)**.** For an MOMDP m, a set $\mathcal{CCS}(\Pi^m)$ is a convex coverge set if it is a subset of $\mathcal{CH}(\Pi^m)$ and if, for every w, it contains a policy whose linearly scalarised value is maximal:

$$\mathcal{CCS}(\Pi^m) \subseteq \mathcal{CH}(\Pi^m) \wedge \forall_w \exists_{\pi \in \mathcal{CCS}(\Pi^m)} \forall_{\pi' \in \Pi^m} w \cdot V^\pi \geq w \cdot \mathbf{V}^{\pi'}\,.$$

As for the monotonically increasing functions, another ordering between the policies is required because such ordering using w is not possible. One such ordering is the Pareto dominance.

Definition 2.11 (Pareto Dominance)**.** A policy π Pareto-dominates another policy π' when its value is at least as high in all objectives and strictly higher in at least one objective:

$$\mathbf{V}^\pi \succ_{\mathtt{P}} \mathbf{V}^{\pi'} \iff \forall_i V_i^\pi \geq V_i^{\pi'} \wedge \exists_j V_j^\pi > V_j^{\pi'}\,.$$

This ordering can be used to construct a set of optimal policies by only the policies which are not Pareto dominated by any other policy, namely, Pareto Front set.

Definition 2.12 (Pareto Front)**.** For an MOMDP m, the Pareto front is the set of all policies that are not Pareto dominated by any other policy in Π^m:

$$\mathtt{PF}(\Pi^m) = \{\pi : \pi \in \Pi^m \wedge \neg \exists_{\pi' \in \Pi^m} \mathbf{V}^{\pi'} \succ_{\mathtt{P}} \mathbf{V}^{\pi'}\}\,.$$

Still, the $\mathbb{PF}(\Pi^m)$ suffers from the same problem as the $\mathbb{U}(\Pi^m)$, which is the problem of having multiple policies that are optimal for the same weight range and not optimal for any other weight. So Pareto Coverage Set is used instead.

Definition 2.13 (Pareto Coverage Set)**.** For an MOMDP m, a set $\mathcal{PCS}(\Pi^m)$ is a Pareto coverge set if it is a subset of $\mathbb{PF}(\Pi^m)$ and if, for every policy $\pi' \in \Pi^m$, it contains a policy that either dominates π' or has equal value to π':

$$\mathcal{PCS}(\Pi^m) \subseteq \mathtt{PF}(\Pi^m) \wedge \forall_{\pi' \in \Pi^m} \exists_{\pi \in \mathcal{PCS}} \wedge (\mathbf{V}^\pi \succ_{\mathtt{P}} \mathbf{V}^{\pi'} \vee \mathbf{V}^\pi = \mathbf{V}^{\pi'})\,.$$

$\mathcal{PCS}(\Pi^m)$ is equivalent to $\mathbb{U}(\Pi^m)$ for monotonically increasing functions. These classifications are used to identify the class of any problem and according to that, a proper algorithm is used. According to [23], there are four cases for optimal solutions:

- w is known and either f is linear or f is monotonically increasing and stochastic policies are not allowed,
- w is known and f is monotonically increasing and stochastic policies are allowed,
- w is unknown and either f is linear or f is monotonically increasing and stochastic policies are allowed, and
- w is unknown, f is monotonically increasing and stochastic policies are not allowed.

In this dissertation, the focus is on obtaining the $\mathcal{CCS}$ of deterministic stationary policies as it is the solution to a relatively big and important portion of the problems, namely, the second scenario.

2.1.6 Value Functions

There are two essential quantities in the single-objective setting which are used to express the optimisation objective of the agent, namely, State Value function ($V^\pi(s)$) and Action Value function ($Q^\pi(s,a)$).

2.1.6.1 State Value function

The state value function represents how good it is for the agent to be in a certain state s if the agent is following a certain policy π from that time step onwards. $V^\pi(s)$ is defined as:

$$V^\pi(s) = \mathbb{E}_\pi[R_t|S_t = s]\,, \tag{2.7}$$

which can be expressed recursively as:

$$V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)\left(\mathcal{R}(\mathrm{s},\,\mathrm{a}) + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'} V^\pi(s')\right). \tag{2.8}$$

An extension to $V^\pi(s)$ that tries to measure how good a policy is in an environment is State Independent Value vector (V^π) which is defined as:

$$V^\pi = \mathbb{E}[R_t | t = 0, \pi, \mu] \,. \tag{2.9}$$

V^π can be used to compare different policies and pick the one that has the highest value of all of them. More on that in section 2.1.6.3. It is worth noting that V^π is related to $V^\pi(s)$ by the equation:

$$V^\pi = \sum_{s \in \mathcal{S}} \mu(s) V^\pi(s) \,. \tag{2.10}$$

2.1.6.2 Action Value function

The action value function represents how good for an agent at state s to take action a and then follow the policy π from the next state onwards. Action Value function is defined as:

$$Q^\pi(s, a) = \mathbb{E}_\pi[R_t | S_t = s, A_t = a] \,, \tag{2.11}$$

which can be further expressed recursively as:

$$Q^\pi(s, a) = \mathcal{R}(\text{s, a}) + \gamma \sum_{s' \in \text{states}} \mathcal{P}^a_{ss'} \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s', a') \,. \tag{2.12}$$

2.1.6.3 Optimal Value Functions

Both $V^\pi(s)$ and $Q^\pi(s, a)$ can be used to define Optimal State Value function ($V^*(s)$) and Optimal Action Value function ($Q^*(s, a)$) respectively:

$$V^*(s) = \max_\pi \quad V^\pi(s) \,, \tag{2.13}$$

which can be further expressed recursively as:

$$V^*(s) = \max_a \mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{s_t s_{t'}} V^*(s') \,, \text{ and} \tag{2.14}$$

$$Q^*(s, a) = \max_\pi \quad Q^\pi(s, a) \,, \tag{2.15}$$

which can be expressed recursively as:

$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \max_{a'} \quad Q^*(s',a')\,. \tag{2.16}$$

And for deterministic policies it becomes:

$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma \max_{a'} \quad Q^*(s',a')\,. \tag{2.17}$$

The main advantage of $Q^*(s,a)$ over $V^*(s)$ is that it allows the agent to act optimally by picking the action with the highest q-value at each state. One important thing about $V^*(s)$ and $Q^*(s,a)$ is that they depend on the assumption of additive returns, and they cannot handle any other scenario. So, they cannot handle any sophisticated model that does not use additive returns.

2.2 Neural Networks

Neural networks, which are inspired by the human brain, are computational models that aim to learn from experience presented as data. There are many different types of neural networks and different architectures that have been successfully employed to solve many different problems. In this section, we introduce the perceptrons which are the building block for neural networks then we introduce the two main architectures used throughout this dissertation, namely, Multilayer Perceptron and Convolutional Neural Networks. Furthermore, we explain the procedure employed to train neural networks to make them learn from the given data.

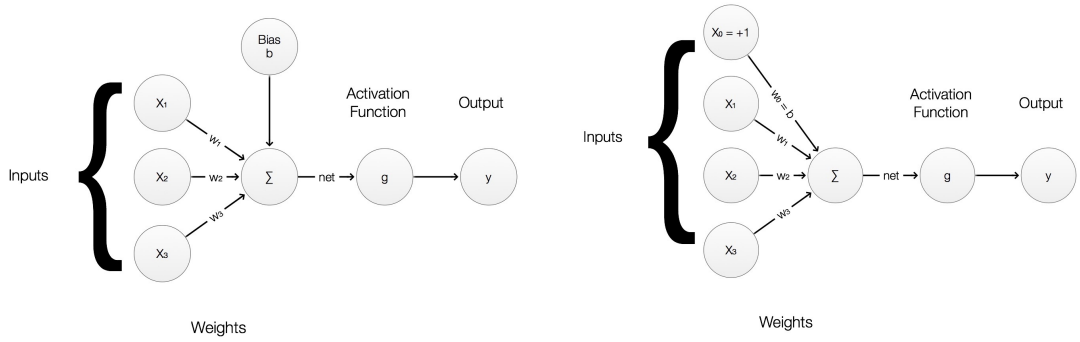
2.2.1 Perceptrons

Perceptrons are the basic building block for neural networks. A perceptron is a processing unit that takes multiple inputs and outputs a single value; this is depicted in figure 2.1a. The perceptron first applies a weighted sum of its inputs. Then, it combines this weighted sum with a bias value and then applies a non-linearity activation function to the resulting summation. The output of the non-linearity function fires if the output is above a certain threshold that is determined

by the bias. This is depicted in figure 2.1a. However, in the vast majority of literature about neural networks, the bias term is incorporated into the summation by adding one extra input with a constant value of 1 and the bias being the weight for this input. This is depicted in figure 2.1b. The equations used to compute the output of the perceptron are:

$$net = b + \sum_i w_i.x_i \,, \quad \text{and} \tag{2.18}$$

$$y = g\big(net\big) \,. \tag{2.19}$$



(a) Perceptron Unit with explicit bias.

(b) Perceptron Unit without explicit bias.

Figure 2.1: Perceptron Unit.

The weights and the bias can be adapted to force the perceptron to output specific patterns. As for the nonlinearity function g, there are multiple possible functions that are usually used. With the most famous of them being the sigmoid function; $g(net) = \frac{1}{1+e^{-net}}$ and the rectifier function; $g(net) = max(0, net)$. These functions are depicted in figure 2.2. The neurons that employ the rectifier function are called $ReLU$, and the neurons that employ the sigmoid function are called Sigmoidal. One noticeable thing about $ReLU$ is that it is not differentiable at 0 which could be problematic because learning uses gradients. But, it has subgradients which can be used instead of the gradient and a common choice for the subgradient at the nondifferentiable point is 0.

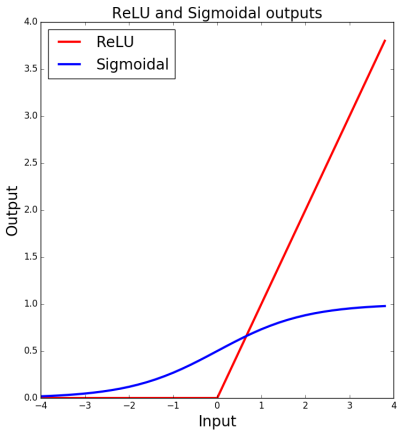


Figure 2.2: *ReLU* and Sigmoidal Activation functions.

There are two main reasons why *ReLU*s are more favoured than sigmoidal units. The first one is that it solves the vanishing gradient problem that occurs when using deep neural networks with sigmoid activation functions. "Vanishing gradients occur when lower layers of a DNN have gradients of nearly 0 because hidden layer units are nearly saturated at -1 or 1." [19]. The problem of vanishing gradients causes very slow convergence for the neural networks. Another advantage of *ReLU* is that it produces more sparse representations than when sigmoidal units are used. Sparsity has multiple advantages like Information disentangling and better Linear Separability [10].

The perceptron's capacity to learn depends on the features that are used as its input. This is problematic because the hard part for any learning algorithm is to extract good features and if a perceptron is to be used, those features will have to be hand-engineered by an expert. This limits the capacity of what a perceptron can learn, rendering it unsuitable for any task with a considerable amount of sophistication. However, this problem can be solved by stacking multiple neurons together which act as feature detectors. This is exactly what a Multilayer Perceptron is, and it is presented in section 2.2.2. Throughout this dissertation, *ReLU* and the second notation of perceptrons with the bias being treated as the zeroth weight are the ones being used.

2.2.2 Multilayer Perceptron (MLP)

A Multilayer Perceptron (MLP) is consists of multiple layers of perceptrons. A layer in the context of a neural network is a set of multiple neurons stacked together, and there are three general types of layers in a neural network. The input layer is the layer that takes the input of the neural network itself, the output layer is the layer that is used to represent the output of the neural network, and the hidden layers are the layers which are neither input nor output layers. The term Deep Neural Network (DNN) is used to refer to neural networks with many hidden layers. The output of each layer is a vector of the outputs of all the neurons in that layer. All layers take as input the output of the previous layer except for the input layer which takes the original input as its input. Also, the output layer is not restricted to be one neuron only; it has as many neurons as the number of outputs required. In figure 2.3, this is an MLP with 1 hidden layer and 3 layers of depth.

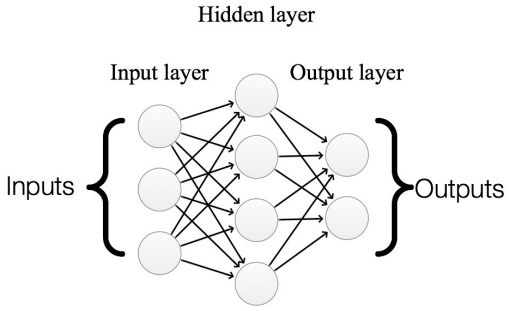


Figure 2.3: Multilayer Perceptron (MLP).

The main difference between neural networks and perceptrons is that any neural network has at least one hidden layer while perceptrons have only inputs and one output. The hidden layers act as feature detectors trying to discover the information that characterises the data it is being presented with. This makes the hidden layers play a crucial role in any task for the neural network and the ability to train them well means the difference between having a neural networks that works and one that does not. The task of calculating the output of the neural

network w.r.t. some input is denoted by forward-propagation. So, in forward-propagation on input x, the input layer is set to be the features of x plus the constant feature with value 1 to learn the bias and each hidden layer l_i takes the output o_{i-1} of layer l_{i-1} as input and outputs $o_i = g(W_{i-1} \cdot o_{i-1})$, where W_k is a matrix of the weights for all the neurons in layer $k+1$. This is true for all layers except maybe for the last layer l_L which can output either $o_L = g(W_{L-1} \cdot o_{L-1})$ or $o_L = W_{L-1} \cdot o_{L-1}$ depending on the type of the problem the neural network is used for.

2.2.3 Training

Training an MLP requires some labeled data

$$D = \{(x_i, y_i)\}, \tag{2.20}$$

where:

- x_i is a vector representing the i^{th} input features,
- y_i is either a vector or scalar representing the correct output for the i^{th} input, and
- m is the number of data points in D.

To train a neural network to do a certain task, a cost function is used to estimate how good the neural network is doing at any point in time, and the neural network tries to minimise its error w.r.t. the cost function. A typically used cost function is the mean squared error:

$$L = \frac{1}{2} \sum_{i=1}^{m} \left(\hat{y}_i - y_i\right)^2, \tag{2.21}$$

where:

- $\hat{y}_i$ is the result of applying forward-propagation on x_i.

Because $\hat{y}_i$ depends on all the weights of the neural network, this dependence is further extended to the cost function, and thus gradient based methods can be used to adjust the weights to reduce the error making the neural network learn to produce a certain function. Gradient descent is typically used when updating the weights of the neural network, and it has the form:

$$W_i^{jk} = W_i^{jk} - \alpha \frac{\partial L}{\partial W_i^{jk}}\,, \tag{2.22}$$

where:

- W_i^{jk} is the weight between layers i and i+1 connecting the neuron j in layer $i+1$ and neuron k in layer i.

2.2.3.1 Backpropagation

The central issue with neural networks is how can it learn multiple layers of features. The Backpropagation algorithm uses the chain rule to calculate the derivative of the error w.r.t. all the weights in the neural network to update the weights to reduce the overall error. It consists of two main parts. The first part computes the error derivative w.r.t. the net inputs to all hidden units and the second part extends that to compute the derivative of the net input to the hidden units w.r.t. their weights.

Starting from a neuron in the output layer that has output $\hat{y_j}$, the error derivative w.r.t. its net input when using the mean squared error loss function can be calculated using:

$$\frac{\partial L}{\partial net_j} = \left(\hat{y_j} - y_j\right) \cdot g'(net_j)\,, \tag{2.23}$$

where:

- $g'(net_j)$ is used to represent $\frac{\partial \hat{y_j}}{\partial net_j} = \frac{\partial g(net_j)}{\partial net_j}$.

To extend this to neurons of previous levels, it is essential to notice that each neuron in level i affects all neurons in level $i+1$. So, to calculate the derivative of error w.r.t. the net input to neuron k at level i given the derivative of error w.r.t. all neurons at level $i+1$:

$$\frac{\partial L}{\partial net_k} = \sum_j \frac{\partial L}{\partial net_j} \cdot \frac{\partial net_j}{\partial y_k} \cdot \frac{\partial y_k}{\partial net_k}\,, \tag{2.24}$$

which equals:

$$\frac{\partial L}{\partial net_k} = \sum_j \frac{\partial L}{\partial net_j} \cdot W_i^{jk} \cdot g'(net_k)\,, \tag{2.25}$$

where:

- y_k is any neuron at some level i, and
- j is a variable that loops over all neurons at level i + 1.

Using this procedure, the derivative of the error w.r.t. all inputs to all neurons can be calculated. The second step of the backpropagation algorithm is to use the error derivative w.r.t. the input of each neuron to calculate the error derivative w.r.t. all weights in all layers. This is done as follows:

$$\frac{\partial L}{\partial W_i^{jk}} = \frac{\partial L}{\partial net_j} \cdot \frac{\partial net_j}{\partial W_i^{jk}}\,, \tag{2.26}$$

which equals:

$$\frac{\partial L}{\partial W_i^{jk}} = \frac{\partial L}{\partial net_j} \cdot y_k\,, \tag{2.27}$$

where:

- y_k is the output of neuron k in layer i.

Given that the partial derivatives have been calculated, there are two main classes of gradient descent methods that are used to update the weights of the neural network based on the calculated gradients, namely, Batch Gradient Descent and Stochastic Gradient Descent.

2.2.3.2 Types of Gradient Descent

Batch Gradient Descent In batch gradient descent, W_i^{jk} is updated using the gradient calculated using all the training examples in the training dataset. This method has the advantage of calculating an accurate estimate of the gradient. It also allows for the parallelisation of the training procedure over different weights. However, this method requires all the data to be loaded and the derivative being calculated w.r.t. the error of all elements in the dataset before any update to the weights is done which takes a lot of time. For the mean squared error loss function, the update rule is:

$$W_i^{jk} = W_i^{jk} - \frac{\alpha}{2} \sum_{l=1}^{m} \left(\hat{y}_l - y_l\right)^2 \frac{\partial}{\partial W_i^{jk}} \,. \tag{2.28}$$

Stochastic Gradient Descent In stochastic gradient descent, W_i^{jk} is updated using the gradient calculated using a sample of the training examples in the dataset. Stochastic gradient descent has the advantage that it is much less likely to be trapped in local minima. It is also much faster than batch gradient descent and requires much less storage and much less computation. The extreme case of stochastic gradient descent uses only one training example and is called Online Gradient Descent. For the mean squared error loss function, the update rule is:

$$W_i^{jk} = W_i^{jk} - \frac{\alpha}{2} \sum_{l=1}^{u} \left(\hat{y}_l - y_l\right)^2 \frac{\partial}{\partial W_i^{jk}} \,, \tag{2.29}$$

where:

- $u < m$ for stochastic gradient descent and $u = 1$ for online gradient descent.

Both of these forms of gradient descent are used extensively in training neural networks. However, for most interesting problems these two variants of gradient descent are not fast enough to converge to good minima and might not even converge to good minima. One of the main reasons of that is the difficulty of setting the learning rate α to a good value. If the learning rate is too large, it leads to the divergence of gradient descent, and if it is too small, gradient descent takes too much time. Another reason why simple gradient descent methods are

not very efficient is that the error surface can have many valleys so the gradient becomes very small which slows down the convergence a lot. Therefore, the need arises for techniques that improve on gradient descent methods to make sure that learning does not get stuck at a poor minimum.

2.2.3.3 Improvements to Gradient Descent

Therefore, there has been a plethora of techniques to improve on standard gradient descent methods. Two of the most important techniques are the use of momentum with standard gradient descent and the use of RMSProp algorithm.

Momentum Momentum is a technique usually used to reduce the chances of getting stuck at local minima and valleys when optimising deep neural network architectures. When using momentum, the gradient computed does not directly change the weights, but it changes the velocity by which the weight is changing [22]. The update equations for gradient descent when using momentum become:

$$\begin{aligned} v_i^{jk} &= \beta v_i^{jk} - \alpha \frac{\partial L}{\partial W_i^{jk}} \\ W_i^{jk} &= W_i^{jk} - v_i^{jk} \end{aligned} \quad . \tag{2.30}$$

This equation sets the velocity of any weight to be the moving average of all its already computed gradients. This way, learning is much faster when reaching a valley because the velocity term retains its value for some time leading to a fast escape of those valleys. The variable β is used to adjust how much previously calculated gradients are retained affecting the current gradient update.

RMSProp RMSProp is another modification to standard gradient descent algorithms which is based on the finding that gradient descent works better when dividing the gradient used for the update at any iteration by its moving average [13]. The update equations for gradient descent when using RMSProp become:

$$\begin{aligned} m_i^{jk} &= \beta m_i^{jk} - (1-\beta) \frac{\partial L}{\partial W_i^{jk}}^2 \\ W_i^{jk} &= W_i^{jk} - \alpha \frac{\partial L / \partial W_i^{jk}}{\sqrt{m_i^{jk}}} \end{aligned} \quad . \tag{2.31}$$

2.2.4 Convolutional Neural Networks (CNN)

Convolutional Neural Network (CNN) is a type of neural networks that tries to mimic how the visual cortex processes information in the brain. CNN is used in a number of different tasks within the field of Computer Vision like Image Recognition [11, 31], Scene Labeling [8] and it has also been used as the base for many RL algorithms [20, 30, 41]. CNN has also been used in other fields like Speech Recognition [33], Natural Language Processing [5, 6] and Text Categorisation [15]. In the context of this dissertation, CNN is used within the computer vision and RL paradigm and as such, the main focus of this section would be to show how CNN is applied to images.

2.2.4.1 CNN Layers

CNN takes as input the raw pixels of a two-dimensional image possibly with multiple channels (depth), and it applies a sequence of operations that aim to abstract the image and extract information that is useful for the task required from the CNN. There are three main types of layers that are used to build a CNN, namely, Convolutional Layer, Pooling Layer and Fully-Connected Layer [18].

Convolutional Layers Convolutional layers are the fundamental building block of a CNN, and they do the main computations of a CNN. Each convolutional layer consists of multiple learnable filters (filter bank) which are applied over all subregions of the entire input on the whole depth of the input. A filter is a just a function that does a dot product between its learnable weights and a part of the image that it is applied on. One famous filter has the size $5 \times 5 \times 3$ meaning that it is applied over parts of the input of length = 5, width = 5 and depth = 3.

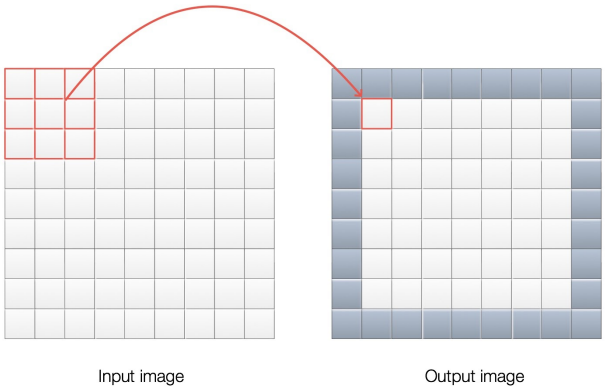


Figure 2.4: 3×3 Filter.

This is the image of a 3×3 filter that is applied to the image on the left (represented by 9×9 matrix) and produces the image to the right (represented by 8×8 light grey matrix). The result of multiplying the filter (represented by squares with red borders) by the cells beneath it is the cell with red borders in the image on the right. The filter is translated in both the horizontal and vertical directions to produce the light grey cells in the image on the right. The darker cells in the right image represent the lost cells because there are no cells in the left image that the filter can be applied to and produce those darker cells.

Each filter outputs a separate feature map which is the values that result from applying the filter across all possible parts of the image. The main intuition behind using filters is that if a filter learns to detect some feature, i.e., a vertical edge, it is required that this filter should detect the same feature across the whole image giving CNN a major advantage over MLP which is translation invariance. The number of filters in any convolutional layer and their sizes are hyper-parameters chosen by the user. Filters are modelled by neurons, and the size of the filters is usually referred to as the receptive field of the neurons. There are another two parameters that are associated with filters, namely, step size and padding size. The step size is used to determine the distance between any two consecutive pixels to which the filter is applied to. A step size of one means that the filter is applied to all pixels in the image except for the border ones which the filter cannot reach. As for the padding size, it is about extending the images with pixels with

a specified value (typically 0 so that it does not mess up the output of the filter) to make sure that the filter is applied to all pixels in the image. This is depicted in figure 2.5. A typically used formula for the size of padding is $\frac{\text{filter width}-1}{2}$ in the x-direction and $\frac{\text{filter height }-1}{2}$ in the y-direction. The output of the filters is followed by a non-linearity function, and in the recent years *ReLU* has been the most used one.

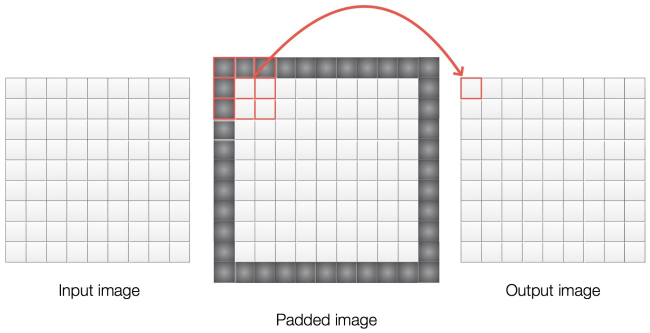


Figure 2.5: Padding.

This image represents the padding and filter operations. The input image (a 9×9 matrix on the left) is padded with 1 cell on each of the four sides allowing the 3×3 filter to produce a 9×9 image as output as it can now reach border cells.

Pooling Layers Pooling layers are used to "progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network." [18]. The pooling layers do not require any parameters, and they are used to compute some fixed function over a patch of the input outputting either one value or a patch of smaller size. There are many types of pooling layers like max-pooling which is depicted in figure 2.6 and average pooling with the most used form of pooling being the max-pooling. These pooling layers add to the translation invariance by only keeping their output across the whole patch of the image without keeping track of which pixel it was in. It is worth noting that in the max-pooling layers, the gradient propagates back only through the cells that had the maximum value so it is important to keep track of those cells during forward propagation.

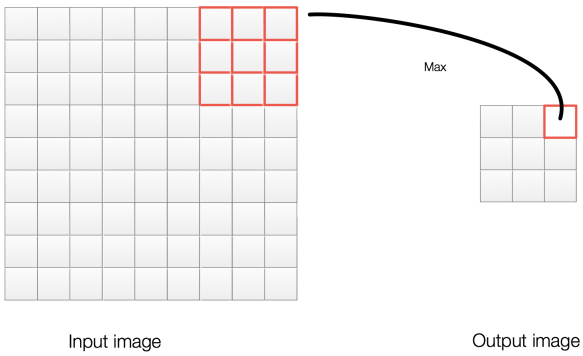


Figure 2.6: Max-pooling.

This image represents the max-pooling operation. In this image, a max-pooling operation is done on patches of 3×3 cells and produces a single cell with a value equal to the maximum value in the input patch.

Fully Connected Layers Fully connected layers are exactly like MLP having each neuron in layer i connected to all neurons in layer $i + 1$.

Any CNN consists of multiple convolutional layers followed by a zero or more pooling layers followed by at least one fully connected layer. The sequence of convolutional layers and pooling layers can be repeated multiple times. figure 2.7 represents the architecture of a simple CNN.

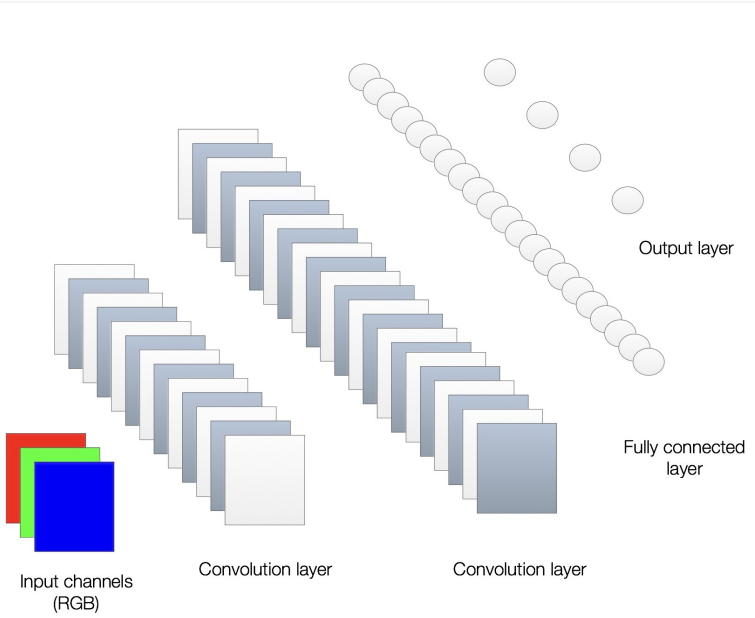


Figure 2.7: CNN.

This is an image of a CNN that takes an RGB image as input (represented by the three channels). It has two convolutional layers, one fully connected layer and an output layer without a pooling layer. The connections between layers are not depicted in this figure.

2.3 Solving MDPs and MOMDPs

The difference between the MDPs formalisation and the MOMDPs formalisation is that in the former, rewards are single scalar values while in the latter, rewards are vectors with one scalar value per objective. Although this change is minor, it leads to a drastic change in the applicability of algorithms to each of the two formalisations. In this section, some of the techniques used to solve MDPs and MOMDPs are presented, as well as, DQN and OLS the two main algorithms used throughout this dissertation. Sections 2.3.3 and 2.3.4 are mainly based on the PhD thesis of D. M. Roijers [23].

2.3.1 Function Approximation in RL

In the single-objective setting, there are many algorithms that have been proposed to solve RL and Planning problems like Dynamic Programming [3, 36] and Temporal Difference (TD) methods [35]. However, these methods lack the generalisation needed to solve interesting problems with very large state spaces and/or action spaces. The reason these methods do not generalise well is that they all depend on some representation in which each state-action pair is represented separately, then they apply some sort of value iteration to solve for the optimal value functions introduced in section 2.1.6, which then enables them to infer the optimal policy from their results. Value iteration turns eq. (2.14) and eq. (2.17) into iterative update by converting them into

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_k(s') \right), \text{and} \tag{2.32}$$

$$Q_{i+1}(s,a) = \mathcal{R}(s,a) + \gamma \max_{a'} Q_i(s',a') \tag{2.33}$$

respectively. Q-learning, which is the base for many RL algorithms, is defined as the continuous application of eq. (2.33) to approximate the true action-value function for all states and actions. Q-learning has the advantage of being guaranteed to converge to the optimum action-values.

Q-learning converges to the optimum action-values with probability 1 so long as all actions are repeatedly sampled in all states and the action-values are represented discretely. [41].

In standard RL algorithms, value iteration is applied in a table lookup way in which each state (for state value function), or state-action pair (for action value function) is represented by a cell in that table, and then they are updated according to eq. (2.32) or eq. (2.33) respectively. The term Q-table lookup is usually used to refer to the application of eq. (2.33) in a table layout format. However, in practice, such representation does not scale to problems with very big state spaces and/or action spaces due to the storage requirements needed to store such tables. In addition to that, it is very computationally demanding to calculate the separate q-values for each state-action pair individually without any form of generalisation.

These are the two main reasons why standard RL algorithms that depend on value iteration cannot solve problems with very big state spaces and/or action spaces.

The main approach to overcome this problem is to introduce function approximation techniques which help solve those two problems and scale RL algorithms. In function approximation techniques, some function with parameters is used to approximate $V^\pi(s)$ and $Q^\pi(s,a)$ using the equations:

$$\mathcal{V}(s,\theta) \approx V^\pi(s)\,, \text{ and} \tag{2.34}$$

$$\mathcal{Q}(s,a,\theta) \approx Q^\pi(s,a)\,. \tag{2.35}$$

There are two main function approximators used within the field of RL, namely, linear combinations of features and neural networks. Although function approximation can be applied to both state value function and action value function, it is generally better in the context of action value function because it allows for the direct inference of the optimal policy by following the action with highest q-value at each state. When using function approximation, a cost function needs to be defined to reduce the error w.r.t. it. A typical cost function would be of the format:

$$\text{cost} = \text{func}(\text{t} - \text{ o})\,, \tag{2.36}$$

where:

- o is the output of the function approximator,
- t is the true output that the function approximator should be trying to approximate, and
- func represents some function over the difference between the target and the exact output.

In the case of action value function, the output is the q-value of some state-action pair

$$\text{cost} = \text{func}(\text{t} - \mathcal{Q}(s,a,\theta))\,. \tag{2.37}$$

However, in RL, the actual target that the algorithm tries to approximate is assumed to be unknown. Therefore, there are multiple proposed targets to be plugged in instead of the true target like, TD(0) target which is defined as:

$$\text{cost} = \text{func}(R_{t+1} + \gamma\, \mathcal{Q}(s_{t+1}, a_{t+1}, \theta) - \mathcal{Q}(s, a, \theta))\,. \tag{2.38}$$

In this dissertation, an algorithm known as Deep Q-Network (DQN) which uses neural networks as its function approximator will be used because it has shown great results in the recent years [20, 21, 30].

2.3.2 Deep Q-Networks (DQN)

Deep Q-Network (DQN) is a function approximation technique that uses deep neural network architectures, namely, Q-networks with weights θ to approximate $Q^{\pi}(s, a)$

$$\mathcal{Q}(s, a, \theta) \approx Q^{\pi}(s, a)\,. \tag{2.39}$$

To train this neural network, an objective function has to be defined so that the error w.r.t. this objective function would be minimised. The objective function that provides the base for DQN is the mean squared error in q-values which is defined as follows:

$$\mathcal{L}(\theta) = \mathtt{E}_{s,a,r,s' \sim \mathcal{D}}\left[\left(\mathcal{R}(s, a) + \gamma \max_{a'} \mathcal{Q}(s', a', \theta) - \mathcal{Q}(s, a, \theta)\right)^2\right]. \tag{2.40}$$

The use of this loss function for DQN provides the simplest form of DQN. In this simple form, every time the agent takes an action and experiences a feedback from the environment, the loss function is applied to modify θ to fit the new data point. However,

Reinforcement learning is known to be unstable or even diverge when a nonlinear function approximator such as a neural network is used to represent the action value function. [38].

The main reasons behind this instability are that within this simple form of DQN,

the data that the agent uses to learn is highly correlated because it is sequential in nature leading to the instability of the learning algorithm. Another cause for this instability is that a slight update to θ can drastically change the policy causing the agent to be exposed to a completely different part of the state space. This continuous change in the distribution of the data that the neural network is being exposed to causes the instability problem.

In [21], two methods have been used to address the instability problem with Q-learning and approximation techniques. The first method is called Experience Replay. Experience Replay tries to remove the significant correlation between the data used by the agent for learning by sampling from the history observed by the agent and relearning from it. The history refers to all the transitions experienced by the agent, where each transition is a four-tuple consisting of (s, a, r, s'). The use of history to sample from breaks the sequential dependency of the data and tries to convert the data into the i.i.d. setting. The second method used is the use of a Target Network that the agent tries to adjust its current estimates of the state-action value towards. In DQN, the agent does that by keeping two sets of weights for the neural network; the first is θ and the second one is θ' which is a previous version of the current set of weights. The weights θ' are usually updated every C episodes. The target of the neural network with the current set of weights θ would then be to adjust its action values towards the action values when the target network's weights θ' are used. Therefore, the loss function becomes:

$$\mathcal{L}(\theta) = \mathtt{E}_{s,a,r,s' \sim \mathcal{D}}\left[\left(\mathcal{R}(s,a) + \gamma \max_{a'} \mathcal{Q}(s', a', \theta') - \mathcal{Q}(s, a, \theta)\right)^2\right] \quad . \tag{2.41}$$

Experience Replay and Target Network are used to improve the training of DQN and reduce the problem of instability of learning. However, one other problem that faces the agent while learning is the need to adequately explore the state space. This is to make sure that it is not giving up on rewards that it has not discovered. One thing that is typically done to improve the exploration of the agent while learning is that it selects and executes actions according to an ϵ-greedy policy based on $Q^{\pi}(s, a)$ rather than always selecting the action with the highest q-value. In ϵ-greedy policy, the optimal action determined by the current policy (i.e., has

maximum q-value at that time step) is taken with a probability of 1 - ϵ, while with a probability of ϵ a random action is taken. This is depicted in eq. (2.42). The use of ϵ-greedy policy is to ensure that the agent explores the state space enough to make sure that it does not miss any optimal solution or have the wrong estimation of the q-values.

$$a_t = \begin{cases} \text{random action} & \text{with probability of } \epsilon \\ \arg\max_a \mathcal{Q}(s, a, \theta) & \text{with probability of } 1 - \epsilon \end{cases}. \tag{2.42}$$

For the training of the DQN when experience replay, target network and ϵ-greedy policy are employed, each time step an action is picked according to some ϵ-greedy policy, it is then executed, a reward is received from the environment, and the whole transition is stored into the history of transitions. Then, instead of using only this data point for learning as in the simple form of DQN, a random minibatch is picked from the history of transitions and backpropagation is then applied to it.

There are two established ways for parameterising the Q-function using a neural network. The first one is depicted in figure 2.8a, and uses the state s as input to the neural network as well as the action a and the output is $\mathcal{Q}(s, a, \theta)$. However, this method has the problem that it scales linearly with the number of actions. This is because it needs one forward pass for each action to get the q-values for all actions before any comparison between those actions can be made. Therefore, the second parameterisation method depicted in figure 2.8b is usually preferred. The input to this method is only the state s and the output of the neural network is a vector of q-values for all actions. Therefore, only one forward pass is needed to compute the q-values for all actions.

(a) Action in DQN. (b) Action out DQN.

Figure 2.8: DQN Architectures.

2.3.3 Solving Multi-objective Decision Problems

In section 2.1.6, we introduced $V^{\pi}(s)$ and $Q^{\pi}(s,a)$ with their optimal versions $V^{*}(s)$ and $Q^{*}(s,a)$, which are used in the single-objective setting to express the optimisation objective of the agent. Then, in section 2.3.1, we showed how these equations have been transformed into an iterative update and how they have been used with function approximation techniques to generalise to problems with very big state and/or action spaces. However, these equations apply only to the single objective setting because for an infinite horizon MDP, there is always an optimal deterministic stationary policy [14]. While for MOMDPs, the maximisation and summation operators are not defined. This is because, in the multi-objective setting, the solutions are sets of policies not only one policy.

In this dissertation, we aim to find the $\mathcal{CCS}$ for MOMDPs with very large state spaces. There are two main approaches to finding the $\mathcal{CCS}$ of an MOMDP, namely, outer-loop approach and inner-loop approach. In the inner-loop approach, a single-objective solver is altered to solve multi-objective problems. As for the outer-loop approach, the $\mathcal{CCS}$ is built incrementally starting from a Partial Convex Coverge Set ($\mathcal{S}$) using a single-objective solver that is applied to scalarised instances of the multi-objective problem.

Definition 2.14 (Partial Convex Coverge Set)**.** A Partial Convex Coverge Set ($\mathcal{S}$), is a subset of a $\mathcal{CCS}$, which is in turn a subset of all possible state independent value vectors (V)

$$\mathcal{S} \subseteq \mathcal{CCS} \subseteq V\,.$$

Any single-objective solver tries to find the optimal policy that maximises the expected future reward. To do that, it has to be able to compute the sum of rewards for each action and compare their corresponding attainable sum to be able to pick the optimal action. Therefore, to convert any single-objective solver into a multi-objective one, these two operations (summation and maximisation) need to be converted into their analogous multi-objective operations (cross-sums and pruning). The reason behind this change in operators is that the solution for multi-objective problems is a set of optimal policies rather than a single optimal policy for single-objective problems. Therefore, the new operators needed to work on sets of policies rather than work on only one policy trying to improve it. These are the two changes needed to convert any single-objective solver into a multi-objective one. Although these steps are easily defined, it can be very hard to re-engineer the single-objective solver with those two definitions. On the other hand, the outer-loop approach has been designed to address the multi-objective problems using a different approach that avoids those two problems with the inner-loop approach. Outer-loop algorithms build the $\mathcal{CCS}$ by picking weights and solving a linearly scalarised version of the multi-objective problem using the picked weights then, adding the discovered optimal policy's value vector to $\mathcal{S}$ only if it improves the scalarised value for some linear w. Also, although the $\mathcal{CCS}$ can be used to solve problems with monotonically increasing f, it is sufficient to use linear scalarisation to discover the true $\mathcal{CCS}$ for problems with linear or monotonically increasing f [23].

One major aspect that affects the performance of outer-loop algorithms is the method used to select the scalarised instances to be solved by the single-objective solver. For example, an outer-loop algorithm can keep picking scalarised instances of the multi-objective problem which it already has the optimal policy and state independent value vector for. This causes the algorithm to make many useless calls to the single-objective solver. From this point onwards, we refer to state independent value vectors as value vectors because they are the ones used to compare policies. In the next section, we discuss Optimistic Linear Support (OLS) which is the outer-loop approach used throughout this dissertation.

2.3.4 Optimistic Linear Support (OLS)

Optimistic Linear Support (OLS) is a novel outer-loop approach that uses a series of scalarised instances of the multi-objective problem and a single-objective solver as a subroutine to build the $\mathcal{CCS}$ [23]. OLS tries to build the $\mathcal{CCS}$ one policy at a time starting from an empty $\mathcal{S}$ by intelligently exploring the weight space until $\mathcal{S}$ has been completed.

OLS, like all outer-loop algorithms, builds $\mathcal{S}$ one value vector at a time by solving a sequence of scalarised instances of the multi-objective problem, but what distinguishes OLS from all other outer-loop algorithms is the way it picks those scalarised instances and the way it prioritises them. OLS uses the concepts of corner weights to pick the weights to use for creating scalarised instances and the concept of estimated weight improvement to prioritise those corner weights.

Corner weights, depicted in figure 2.9, are defined as "the weights at the corners of the convex upper surface" [23]. In the case when there are only two objectives, corner weights associated with the new value vector to be added to $\mathcal{S}$ can be calculated by finding the intersection points between the new value vector and all value vectors in $\mathcal{S}$ and add the intersection points to Q, if and only if, they lie within the range in which the old value vector was optimal. This is illustrated in figure 2.10. Also, it is worth noting that when there are only two objectives each new value vector can at most add two corner weights, but this restriction does not apply to cases with a higher number of objectives. As for the case when the problem has more than two objectives, a linear equation can be solved to determine all the vertices of the polyhedral subspace above all optimal value vectors, i.e., the grey shaded area in figure 2.10.

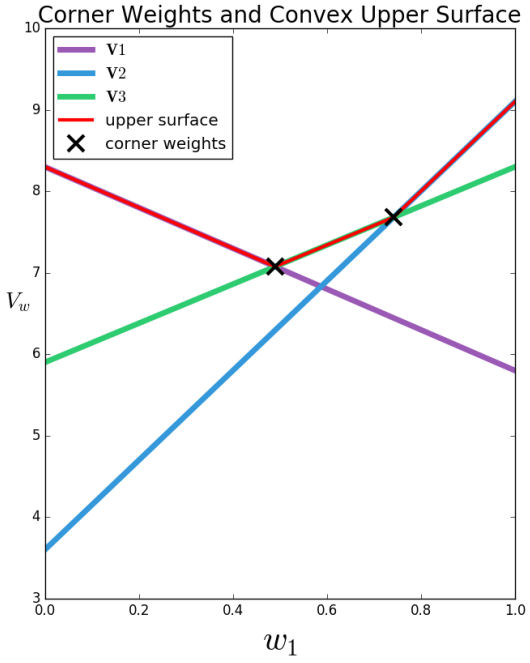
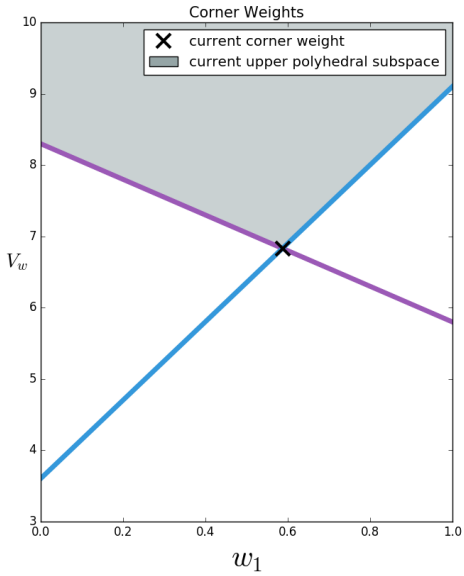
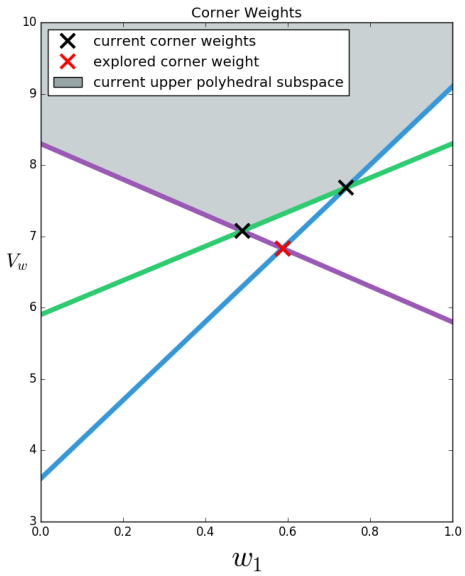


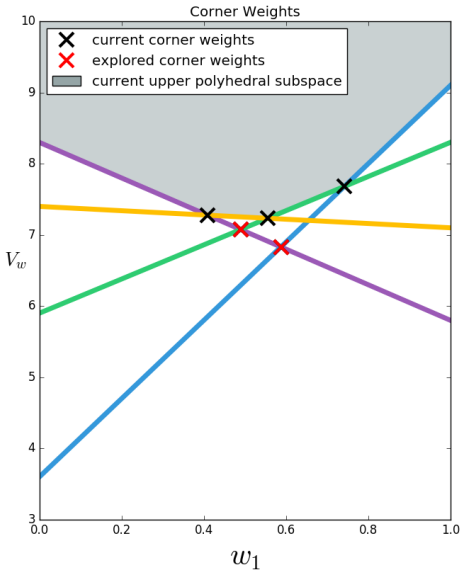
Figure 2.9: Corner weights and convex upper surface demonstration.



(a) First corner weight.



(b) Second and third corner weights.



(c) Forth and fifth corner weights.

Figure 2.10: Corner Weights.

In figure 2.10a, the corner weight represented by the black cross is the only intersection between value vectors in the weight simplex. In figure 2.10b, that corner weight has already been used to scalarise an instance of the multi-objective problem (and was coloured in red), and the solver returned the green value vector as the optimal value vector for this weight. The new value vector produced two new corner weights represented by black crosses in figure 2.10b. In figure 2.10c, one of the two corner weights has been used and produced the orange value vector. This value vector added two new corner weights which are represented by the two new black crosses.

As for the concept of estimated weight improvement, it is defined as "an upper bound on the difference between $V^*_{\mathcal{S}}(w)$ and the optimal scalarised value function $V^*_{\mathcal{CCS}}(w)$" [23], where

$$V^*_{\mathcal{S}}(w) = \max_{V^\pi \in \mathcal{S}} w \cdot V^\pi \,, \tag{2.43}$$

$$V^*_{\mathcal{CCS}}(w) = \max_{V^\pi \in \mathcal{CCS}} w \cdot V^\pi \,, \tag{2.44}$$

$$\text{Improvement}(w) = V^*_{\mathcal{CCS}}(w) - V^*_{\mathcal{S}}(w) \,. \tag{2.45}$$

However, one problem with eq. (2.45) is that it uses the optimal value for the weight using the true $\mathcal{CCS}$ which is not available during planning or learning. Therefore, OLS uses $\overline{\mathcal{CCS}}$ instead of $\mathcal{CCS}$.

Definition 2.15 (Optimistic Hypothetical Convex Coverge Set)**.** An Optimistic Hypothetical Convex Coverge Set ($\overline{\mathcal{CCS}}$) is a set of payoff vectors that yields the highest possible scalarised value for all possible w consistent with finding the vectors $\mathcal{S}$ at the weights in $\mathcal{W}$.

Therefore weight improvement is redefined to be

$$\text{Improvement}(w) = V^*_{\overline{\mathcal{CCS}}}(w) - V^*_{\mathcal{S}}(w) \,, \tag{2.46}$$

where:

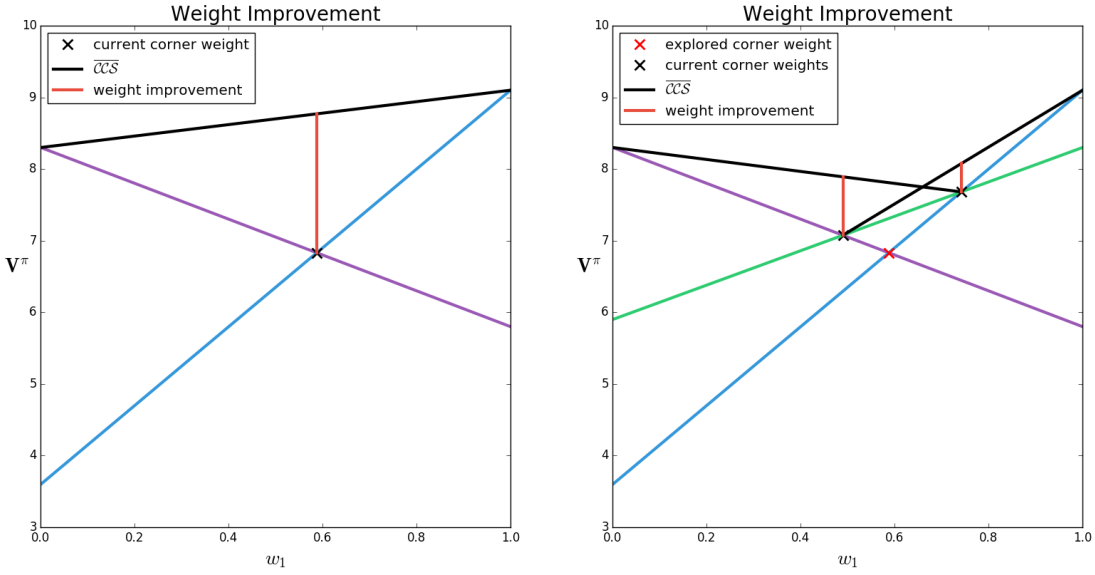
$$V^*_{\overline{\mathcal{CCS}}}(w) = \max_{V^\pi \in \overline{\mathcal{CCS}}} w \cdot V^\pi \,. \tag{2.47}$$

$V^*_{\overline{\mathcal{CCS}}}(w)$ can be calculated by solving the linear program

$$\begin{aligned} \max \quad & V^*_{\overline{\mathcal{CCS}}}(w) = \max_v w \cdot v \\ \text{subject to} \quad & \forall_{w \in \mathcal{W}, s \in \mathcal{S}}\, w \cdot v \leq w \cdot s \end{aligned} \,. \tag{2.48}$$

However, it is simpler to calculate the weight improvement when there are only two objectives. The linear program tries to find the highest possible value vector for the current weight that would not be higher than any of the already existing vectors in $\mathcal{S}$ for any weight. This is to make sure that the new value vector will not remove any of the existing ones from the $\mathcal{CCS}$. Therefore, this upper bound when plotted becomes the edge connecting the two other ends of the value vectors

which intersect at that corner weight. The estimated weight improvement becomes the difference in height between current V^{π} on the upper surface salarised by the corner weight and $\mathbf{V}^{\pi'}$ on the upper bound scalarised by the corner weight. This is illustrated in figure 2.11.



(a) (b)

Figure 2.11: Weight Improvement.

The weight improvement is the distance represented by the red lines.

After weight improvement has been calculated, it is then used to prioritise weights to use for creating the scalarised instances to be solved by the single-objective solver. The major advantage of using this way of prioritising the weights is that it is independent of the single-objective solver which is the most computationally expensive part of OLS.

OLS for Exact Solvers

OLS keeps track of three collections of items, the Partial Convex Coverge Set ($\mathcal{S}$) and OLS Explored corner weights ($\mathcal{W}$) lists and the Priority queue of OLS corner weights to be explored (Q). Q keeps track of the weights that are to be used to

create scalarised instances of the multi-objective problem and apply the single-objective solver to it. These weights are ordered according to their estimated improvement value. Q starts initially with the n extremum weights (each has the whole weight being put on one objective only) and with priorities of infinity, i.e., $<w_0 = 1, w_1 = 0$, estimated improvement $= \infty>$, $< w_0 = 0, w_1 = 1$, estimated improvement $= \infty>$ for 2 objectives (lines 9 - 11). The reason these weights are put in Q with a priority of infinity is that their value vectors are guaranteed to be in any $\mathcal{CCS}$ because they cannot be dominated by any value vector. Therefore, they need to be explored first making sure that their corresponding value vectors are added to $\mathcal{S}$ so that they can direct the exploration of the weight simplex. Then, at each iteration of OLS, the weight with the highest improvement is popped (line 13) and passed to the single-objective solver which uses it to scalarise the multi-objective problem before solving it. The multi-objective problem is scalarised by computing the scalarised rewards and in the case of planning, converting the MOMDP into an MDP. Then, the single-objective solver returns both the optimal policy and it's corresponding value vector (line 14). In this form of OLS, the single-objective solver is assumed to be exact, so the returned value vector is only stored if it does not exist in $\mathcal{S}$ (line 20). If the value vector is to be added, then its corner weights and their estimated improvement are calculated (line 19 and line 22). Then they the corner weights are added to Q, if and only if, their improvement value is greater than τ (lines 21 - 25). Also, corner weights in Q which are made obsolete (no more in the convex upper surface) by the new value vector are removed (line 17). This is done until there are no more corner weights in $\mathcal{W}$ and in that case OLS has found a $\mathcal{CCS}$ and it terminates. This is depicted in algorithm 2.1.

Algorithm 2.1: OLS with exact single-objective solver [23]

```
1  input: problem m  // Multi-objective problem.
2  input: SolveSODP      // Exact single-objective solver>
3  input: τ          // Minimum improvement.
4  output: S // Found CCS .
5  begin
6    S = φ    // Empty partial-CCS  set.
7    W = φ    // Empty list of explored corner weights
8    Q = φ    // Empty priority queue
```

```
9     foreach extremum of the weight simplex w_e do // This loop fills Q
          with the extremum weights
10       Q.add(w_e, ∞)
11    end
12    while !Q.isEmpty() and !timeOut do
13       w = Q.pop()
14       π, V^π = SolveSODP(m, w)
15       W = W ∪ w
16       if not S.contains(V^π) then
17             W_del = W_del ∪ corner weights made obsolete by V^π from Q
18             W_del = W_del ∪ {w}
19             W_V^π = newCornerWeights(.)
20             S = S ∪ {V^π}
21             foreach w in W_V^π do
22                if estimateImprovement(w) > τ then
23                      Q.add(w)
24                end
25             end
26       end
27    end
28 end
```

One main problem with OLS and all other multi-objective solvers is the repetitive use of the single-objective solver which is very computationally demanding and time-consuming. However, [23] suggested reusing an already discovered policy as an initial policy when solving new scalarised instances of the multi-objective problem and the reasoning behind the reuse is

When two corner weights w and w', are similar, the value vectors for these weights found by the single-objective solver are also likely to be similar. [23].

Hence, OLS needs to keep track of already found policies so as to use them for new corner weights.

Chapter 3

Methodology

In this chapter, we present our main contribution Deep OLS Learning (DOL), a multi-objective reinforcement learning algorithm building on top of OLS, an outer-loop algorithm for planning, and the single-objective learning algorithm DQN. We also analyse the consequences of using an outer-loop method in a learning setting rather than a planning setting. We start by presenting how we made DQN OLS-compliant before integrating it with OLS as its single-objective solver. Combining the multi-objective variant of DQN and OLS leads to the first multi-objective RL algorithm that employs deep learning methods, DOL. Additionally, we propose and present two different methods for reusing already learnt policies to speed up the learning of new ones. Moreover, we also present Deep Sea Treasure world which is the problem we used to benchmark DOL. Lastly, we present Max $\mathcal{CCS}$ Difference, an error metric that is usually used in the context of multi-objective planning to evaluate the results of different algorithms.

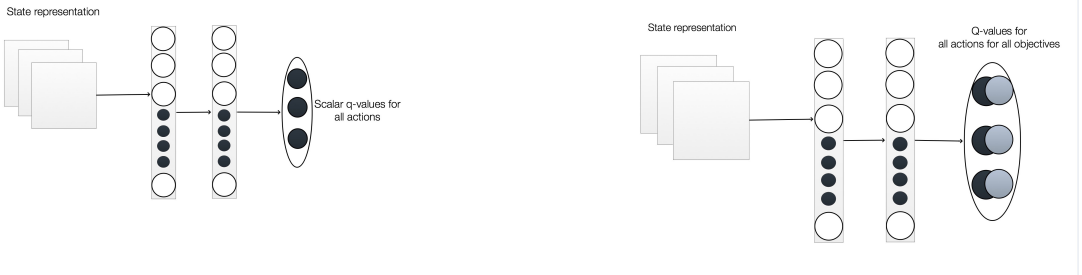
3.1 Multi-objective DQN

In section 2.3.2, we illustrated how DQN uses neural networks to estimate the q-values for all actions. Also, in section 2.3.2, we showed the two most used architectures for DQN in the single-objective setting which are presented in figure 2.8. Additionally, we explained why the architecture in figure 2.8b that outputs the q-values for all the actions is usually preferred over the one in figure 2.8a which outputs the q-value for one action only. However, in these two standard architec-

tures, each action is modelled with one q-value only. This is because it is assumed that the rewards are single scalar values. In this section, we present a variant of the architecture depicted in figure 2.8b that can be used by outer-loop algorithms to solve multi-objective problems.

To make DQN compliant with outer-loop methods, we needed to create a DQN-based scalarised learning algorithm that, given a linear scalarisation weight and a multi-objective problem, learns the optimal policy, the scalarised value corresponding to it and its corresponding multi-objective value vector. This is challenging as many DQN-based methods focus only on learning the optimal policy, and often disregard the need to accurately estimate the q-values for all actions. This is because, in the single-objective setting, the optimal policy does not need the exact q-values, but only a good approximation of them. While, in the multi-objective setting, it is crucial to get very accurate approximations of the true q-values.

In this dissertation, we use a variant of DQN introduced in [9]. In this variant, each action has Number of objectives for an MOMDP (n) values and the neural network tries to learn $n * |\mathcal{A}|$ values instead of only $|\mathcal{A}|$ values. This change in the architecture of DQN is depicted in figure 3.1.



(a) Single-objective DQN architecture.

(b) Multi-objective DQN architecture.

Figure 3.1: DQN Single-objective and multi-objective architectures.
In figure 3.1a, this is the exact same architecture as the one depicted in figure 2.8b with one q-value per one action. On the other hand, figure 3.1b shows the architecture used for problems with two objectives as each action is represented by two q-values to be learnt.

This change in architecture is the main change to DQN that was done in this dissertation. This change is needed because OLS and other outer-loop algorithms

need the single-objective solver to estimate the q-values for all actions for all the objectives at all states to be able to pick amongst them. Another change that was done is associated with the ϵ-greedy policy. While in the single-objective DQN, an action is picked by following an ϵ-greedy policy, the max operator is not defined in the multi-objective setting. Therefore, the max operator is done on a weighted average of the q-values and the given w. Thus, ϵ-greedy policy's definition becomes:

$$a_t = \begin{cases} \text{random action} & \text{with probability of } \epsilon \\ argmax_a \, w \cdot \mathcal{Q}(s, a, \theta) & \text{with probability of } 1 - \epsilon \end{cases}. \tag{3.1}$$

The rest of the training part in DQN goes the same except that in this version, the agent gets a vector of the error in the q-values which is then backpropagated through the DQN instead of a single scalar value.

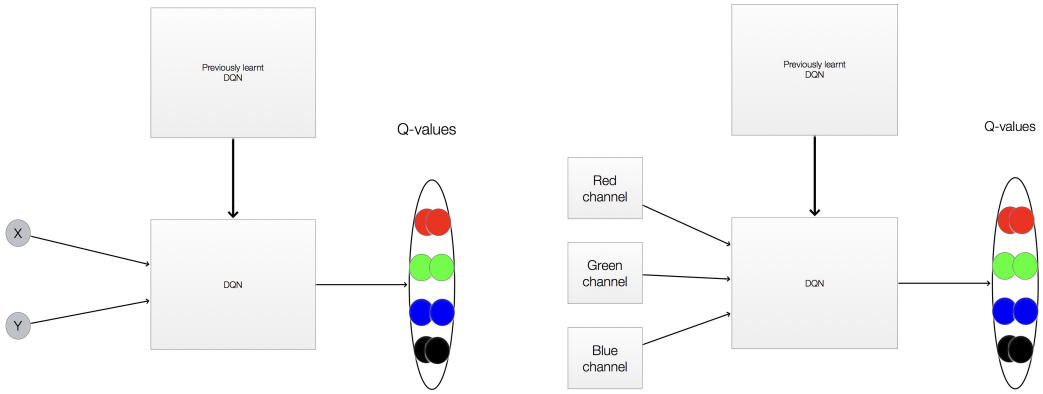
3.2 Reuse of learnt Policies

The concept of reusing already learnt policies is of great importance to all outer-loop algorithms. The reason is that the major time complexity within outer-loop methods is the application of the single-objective solver to a scalarised instance of the problem in hand. Therefore, starting from a policy that is close to the optimal policy would greatly reduce the time needed to learn the new optimal policy, thus speeding up the outer-loop algorithm a lot.

The problem of reuse is twofold. The first part of it is associated with the selection of which learnt policy should be used as an initialisation for the new policy to learn. In this dissertation, the way this was done is by using the policy discovered for the closest corner weight. This is based on the assumption that policies that have been found for a similar corner weight are close to the optimal policy, so we could use the policy found at the closest w from previous iterations [26].

The second part of the problem is related to how can an already learnt policy be used to initialise a new one. This part of the problem depends on the single-objective solver used by the outer-loop algorithm. Because in this dissertation, the single-objective solver that was used is DQN which is based on neural networks.

Thus, the way to reuse the learnt policies is by reusing the weights of their respective neural networks. In this dissertation, two methods of reusing already learnt policies are suggested. The first one is the full reuse of an already learnt policy. So, in the case of DQN, all the weights of the DQN learnt for a previous policy are used to initialise the DQN used for learning the new policy. This is depicted in figure 3.2. For this task, before OLS executes DQN on any new corner weight, an already learnt DQN is picked and set to be the initialisation for learning at the newly picked corner weight. The main intuition behind full-reuse is to try to make DQN backward search from a previously found optimal policy rather than start searching from a completely random policy. This would ensure that more episodes are spent by DQN searching for the new optimal policy around a good estimate of it. Therefore, increasing the chances of finding the true optimal policy.



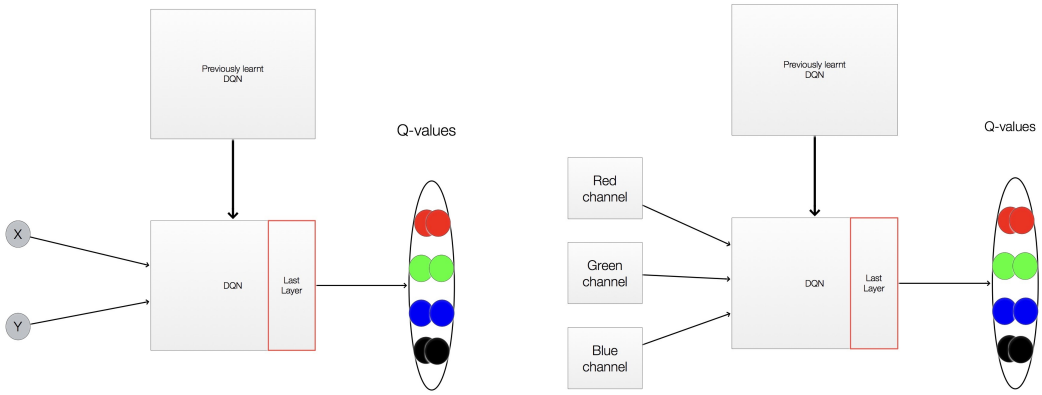
(a) Map version with full-reuse (b) Image version with full reuse

Figure 3.2: Models used for Map and Image versions of deep sea treasure world with full reuse.

In these experiments the DQN reached at the end of learning for one corner weight is used as the initial DQN for learning at a new corner weight.

The second suggested method is partial reuse of learnt policies. In this dissertation, this is implemented by reusing all the weights of an already learnt DQN except for the weights of the last layer which are randomly reinitialised. This is depicted in figure 3.3. The intuition behind this idea is to avoid getting DQN stuck

following the policy used for initialization for a long time during the beginning of learning for the new policy.



(a) Map version with partial reuse (b) Image version with partial reuse

Figure 3.3: Models used for Map and Image versions of deep sea treasure world with partial reuse.

In these experiments the DQN reached at the end of learning for one corner weight is used as the initial DQN for learning at a new corner weight. However, the last weight matrix (i.e., the one connecting last hidden layer with the output layer) is randomly initialised.

3.3 Deep OLS Learning (DOL)

One of the major challenges with this project is to make OLS work in the learning setting. This challenge is composed of two different parts. The first part which has already been addressed in section 3.1, is using a single-objective solver to learn the optimal policy and its corresponding value vector for a scalarised instance of a problem. The second part is making sure that OLS is compliant with the learning setting. This part requires modifications to the core of OLS because OLS was mainly designed for the planning setting in which the discovered policies and their value vectors are exact. However, in the learning setting, there is no such guarantee.

Depending on the type of the single-objective solver, it can be exact, meaning that it finds the true optimal value vector, or it can be non-exact, where it finds

only approximate value vectors. In the case of exact solvers, if the found value vector is not already included in $\mathcal{S}$, this means that it should be added to it because there is at least one weight vector for which it is optimal (the weight used to scalarise the multi-objective problem) which guarantees its existence in the $\mathcal{CCS}$ and at the same time no other value vector can do better, they can at most do as good because the solver is exact. As for the case of non-exact solvers, a new value vector can be worse than an already discovered value vector and the converse is still valid where this value vector can be better than some of the value vectors in $\mathcal{S}$. This situation requires special handling as to whether to add the value vector or drop it depending on the content of $\mathcal{S}$.

In this dissertation, the single-objective solver used is not exact. Therefore, OLS needs to be modified to handle this case. OLS has already been used with approximate planning single-objective solvers in [23], and in this dissertation, we employ the same adaptations but for the RL setting to build DOL. When the single-objective solver is not exact, the found policy and its corresponding value vector are not guaranteed to be optimal. As illustrated in figure 3.4, two value vectors can be found at the same weight. If $\mathbf{V}1$ (represented by the green line) is found after $\mathbf{V}2$ (represented by the orange line), $\mathbf{v}1$ should not be added to $\mathcal{S}$ because there already exists a value vector in it which is better. On the other hand, if it were found first, then when $\mathbf{V}2$ is found, it should remove $\mathbf{V}1$ from $\mathcal{S}$ because it is more optimal than it. It is worth noting that OLS and DOL do not run the single-objective solver for the same weight multiple times as in figure 3.4 so, this situation will never happen in OLS. But, the situation that might happen is when the single-objective solver at another w discovers a value vector that yields the situation of having one value vector better than the already found one.

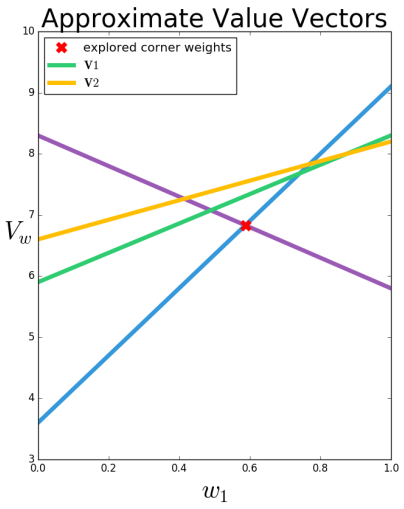


Figure 3.4: Value vectors using approximate single-objective solver.

In this plot, the lines in blue and magenta represent already discovered value vectors and the corner weight produced by them is represented by the red cross. An approximate single-objective solver may return varying value vectors if run twice at the same corner weight. This may result in the two value vectors represented by the orange and green lines.

So, the way in which approximate-OLS presented in [23] handles approximate solver is as follows, it goes normally like exact-OLS, but the new value vector is checked whether it is better than all value vectors in $\mathcal{S}$ at the weight used for creating the scalarised instance of the multi-objective problem or not. If it is not, it is discarded and OLS continues normally, but if it is, then it is added to $\mathcal{S}$ and if by adding it to $\mathcal{S}$ it takes another value vector from the convex upper surface, this value vector is then removed from $\mathcal{S}$. Also, the corner weights are regenerated after adding any value vector to $\mathcal{S}$. This algorithm is presented in algorithm 3.1.

Algorithm 3.1: Approximate OLS

```
1  input: problem m  // Multi-objective problem.
2  input: SolveSODP      // Exact single-objective solver.
3  input: τ          // Minimum improvement.
4  output: S // Found CCS .
5  begin
6    S = φ    // Empty partial-CCS  set.
7    W = φ    // Empty list of explored corner weights.
8    Q = φ    // Empty priority queue.
```

```
    foreach extremum of the weight simplex w_e do // This loop fills Q with the extrema weights..
        Q.add(w_e, ∞)
    end
    while !Q.isEmpty() and !timeOut do
        w = Q.pop()
        π, V^π = SolveSODP(m, w)
        W = W ∪ w
        if not S.contains(V^π) then
            W_del = W_del ∪ corner weights made obsolete by V^π from Q
            W_del = W_del ∪ {w}
            W_V^π = newCornerWeights(.)
            remove vectors from S that are no longer optimal for any w after adding V^π
            S = S ∪ {V^π}
            foreach w in W_V^π do
                if estimateImprovement(w) > τ then
                    Q.add(w)
                end
            end
        end
    end
end
```

In the case of DOL, there is no change needed to the approximate OLS algorithm. All the changes needed are the ones used to convert the multi-objective problem into its scalarised instance. However, in the case of DOL-R, the approximate-OLS algorithm needes to be adapted to store and reuse already learnt models. This is depicted in algorithm 3.2.

Algorithm 3.2: Deep OLS Learning with Reuse (DOL-R)

```
input: problem m  // Multi-objective problem.
input: SolveSODP      // Exact single-objective solver.
input: τ          // Minimum improvement.
input: full_reuse    // A flag that indicates whether the full reuse should be used or partial reuse
output: S // Found CCS .
begin
    S = φ    // Empty partial-CCS set.
    W = φ    // Empty list of explored corner weights.
    Q = φ    // Empty priority queue.
```

```
10    DQN_Models = ϕ // Empty table of DQNs learnt for policies.
11    foreach extremum of the weight simplex w_e do // This loop fills Q
          with the extrema weights..
12        Q.add(w_e, ∞)
13    end
14   while !Q.isEmpty() and !timeOut do
15        w = Q.pop()
16        model = findNearestModel(w, DQN_Models, full_reuse)
17        π, V^π, new_model = approxSolveSODP(m, w, model) // The
              single-objective solver takes the model and uses it for
              initialization if it is not equal to nil
18        W = W ∪ w
19        if not S.contains(V^π) then
20              W_del = W_del ∪ corner weights made obsolete by V^π from Q
21              W_del = W_del ∪ {w}
22              W_V^π = newCornerWeights(.)
23              remove vectors from S that are no longer optimal for any w
                    after adding V^π
24              S = S ∪ {V^π}
25              DQN_Models[w] = new_model
26              foreach w in W_V^π do
27                 if estimateImprovement(w) > τ then
28                    Q.add(w)
29                 end
30              end
31        end
32    end
33
34    function findNearestModel(w, DQN_Models, full_reuse)
35    begin
36        if DQN_Models == ϕ then
37            return nil
38        end
39        w' = nearest explored corner weight to w in the keys of
              DQN_Models
40        Nearest_DQN = DQN_Models[w'] // DQN
41        if not full_reuse then
42            randomly initialize the final layer of Nearest_DQN
43        end
44        return Nearest_DQN
45    end
46 end
```

To implement DOL-R, approximate-OLS had to be modified in a number of different ways. Firstly, DOL-R takes an extra input parameter which is a boolean that indicates whether full reuse should be employed or partial reuse (line 4). Also, DOL-R keeps track of a table of learnt DQN models, namely, DQN_Models (line 10). Before DOL-R invokes the single-objective solver at a new corner weight w, it invokes the method findNearestModel (line 16) which tries to find the model associated with the nearest used weight. If DQN_Models is empty, findNearest-Model returns nil indicating that there is no existing model to be used (lines 36 - 38). But, if DQN_Models is not empty, the model associated with the nearest weight in the keys used for DQN_Models is found (lines 39 - 40), and if the flag full_reuse is false, the last layer of the model is randomly initialized (lines 41 - 44). Then the found model is returned (line 44). Then when DOL-R invokes approxSolveSODP, it adds an extra parameter which is the found model. If the single-objective solver finds that the model is nil, it uses the default initialization method it normally uses. But if the model is not nil, it uses the model provided as its initialization. Then learning continues normally for the single-objective solver except that it returns the final model it finds at the end of learning, as well as, π and $\mathbf{V}^{\pi}$. Then DOL-R continues exactly like approximate-OLS to determine if the new value vector will be added to $\mathcal{S}$. If this is the case, DOL-R continues exactly like OLS by adding the value vector to $\mathcal{S}$ and its corner weights to Q, except that it stores the model obtained from the single-objective solver in the table of models with its key being that corner weight (line 25). Then DOL-R keeps doing the same steps until there are no more weights in Q and it returns $\mathcal{S}$.

3.4 Deep Sea Treasure world

Most of the research in multi-objective RL uses the benchmark problems and error metric presented in [39] to compare results across different multi-objective algorithms. However, these benchmarks and the error metric are concerned with finding the $\mathcal{PCS}$ not the $\mathcal{CCS}$. Therefore, the approach followed in this dissertation cannot be applied directly to those benchmarks and another error metric needs to be used. Therefore, for this dissertation, the benchmark used is a modified version of the Deep Sea Treasure world problem presented in [39]. In this next section,

the deep sea treasure world problem is presented, and the modifications that were made to make it suitable for the approach in this dissertation are highlighted. The problem description is based on the original description of the problem in [39].

3.4.1 Problem Description

As described in [39], in seep sea treasure world, the agent controls a submarine searching for treasures, and there are ten different locations which contain treasures with different values as indicated in figure 3.5. The agent has two objectives that it tries to maximise. The first one is the actual value of the treasure reached and the second one is the time cost to reach the treasure. Each episode starts with the agent at the start state s_0 (top left corner) and ends when the agent reaches a goal or has taken 1000 actions. The agent has four actions at each state which represent the four possible directions, and whenever the agent takes an action that would take it outside the grid, its state does not change. The agent receives a vector of rewards each time step. The first element is the treasure value received which is either 0 when the agent does not arrive at a cell with a treasure in it or the value of the treasure in that cell. The second element is the time penalty which is -1 for all time steps. Both values were normalised to improve the accuracy of DQN. This variant differs from the original problem in two different ways. The first one is the size of the grid where the original one is a 10×10 grid, but this one is 11×11. The reason for this change is to optimise the work for the CNN when applied on the image version of the problem. The other difference is the actual values of the rewards. The original problem has a concave coverage set, and the approach in this dissertation is only applicable to convex coverage sets. The coverage set for this variant is illustrated in figure 3.6.

The previously described version is called the map version in which the agent's state is represented by the coordinates of the submarine within the map. There is also the image version of the treasure world problem that was used in some of the experiments. In that version, the submarine was represented using the green colour, the walls were in black, all goals were in red, and all other cells are in white.

s_0										
0.5										
	28									
		52								
			73	82	90					
						115	120			
								134		
									143	

Figure 3.5: Deep Sea Treasure world.

This is the map of the problem showing the unnormalised values of the treasures.

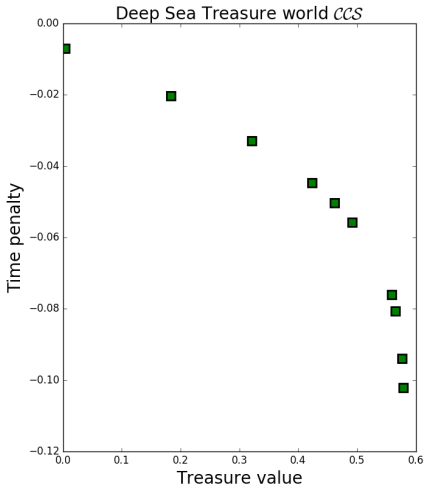


Figure 3.6: Convex Coverage Set of Deep Sea Treasure problem.

This plot shows the $\mathcal{CCS}$ using the normalized values of both the treasures and time penalties.

3.5 Error Metric

To be able to correctly evaluate how good any solution is an error measure from [26] has been employed which calculates the maximum difference between the $\mathcal{S}$ discovered by OLS and the true $\mathcal{CCS}$ after each iteration done by the OLS . The error measure is called Max $\mathcal{CCS}$ Difference, and it is defined as:

$$\text{max-err}_i = \max_{w \in \mathcal{W}} \left| (\max_{c \in \mathcal{CCS}} w.c) - (\max_{s \in \mathcal{S}_i} w.s) \right|, \tag{3.2}$$

where:

- i is a variable that represents the i^{th} iteration in OLS, and
- $\mathcal{S}_i$ is the partial convex coverage set that OLS has acquired by the i^{th} iteration.

This error measure shows how $\mathcal{S}$ is being shaped every iteration to match the true $\mathcal{CCS}$.

Chapter 4

Experiments and Results

In this chapter, we present the scope of this work and what we aim for. We also present the set of experiments used to evaluate DOL and present the results from those experiments. Last but not least, we discuss the results obtained from the set experiments we proposed.

4.1 Scope

The primary objective of this work is to investigate the potential of DOL in the RL paradigm. Throughout this dissertation, the main focus is on RL techniques that can be used to solve problems with very big state spaces which occur naturally in real life. In sequence, we provide a thorough evaluation on the following points:

- the ability to use OLS in the RL setting,
- the potential of DOL in solving multi-objective RL problems, and
- the potential benefit of reusing learnt policies in speeding up learning new ones.

To do so, we test three algorithms, the first one integrates OLS with standard Q-learning. This enables the usage of outer-loop methods in the multi-objective RL setting, but cannot yet tackle large problems. The second algorithm is DOL, which is used to check the potential of the integration between OLS and DQN. Lastly, we check DOL-R, which improves upon DOL by trying the two proposed methods of reusing learnt policies; DOL-R Full and DOL-R Partial.

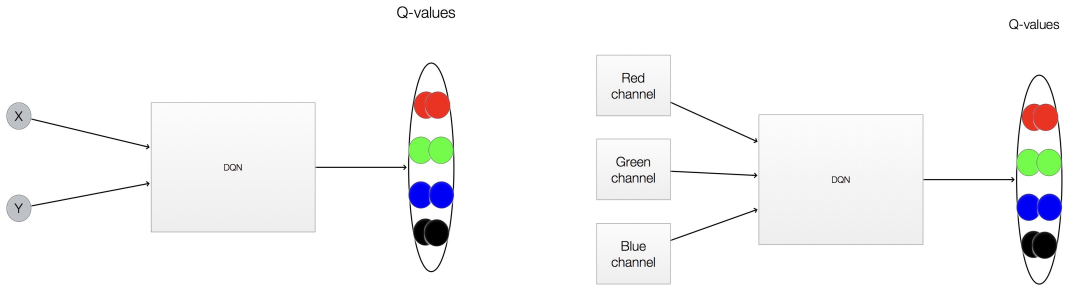
We perform seven different experiments to compare the above mentioned algorithms:

1. OLS with Q-table lookup,
2. DOL on the map version of the deep sea treasure world problem,
3. DOL-R Full on the map version of the deep sea treasure world problem,
4. DOL-R Partial on the map version of the deep sea treasure world problem,
5. DOL on the image version of the deep sea treasure world problem,
6. DOL-R Full on the image version of the deep sea treasure world problem, and
7. DOL-R Partial on the image version of the deep sea treasure world problem.

The first experiment does not really fit with the general focus of the experiments which is about testing OLS for RL algorithms that generalise well for complex problems. However, because this is the first application of OLS in the RL setting, it is important to see if OLS would work with vanilla RL algorithms and check if any unforeseen complications would arise or not. This experiment helped in establishing a baseline to be used to compare the results of other experiments. Additionally, it was used to estimate roughly the number of episodes needed for training other algorithms. The reason behind the use of Q-table look-up is that it is the closest possible RL algorithm to an exact solver, so it is expected that OLS would work well with it and achieve perfect results.

4.2 Experimental Setup

The work for this project was done using Torch framework [4]. The code used was implemented on top of the code provided in [2]. Each experiment was run 24 times. All the neural networks used had 8 outputs because the deep sea treasure world problem has two objectives, i.e., two q-values per action, and four actions. This is depicted in figure 4.1. We also used an Intel(R) Xeon(R) CPU X5690 @ 3.47GHz server to run all the experiments.



(a) Map version. (b) Image version.

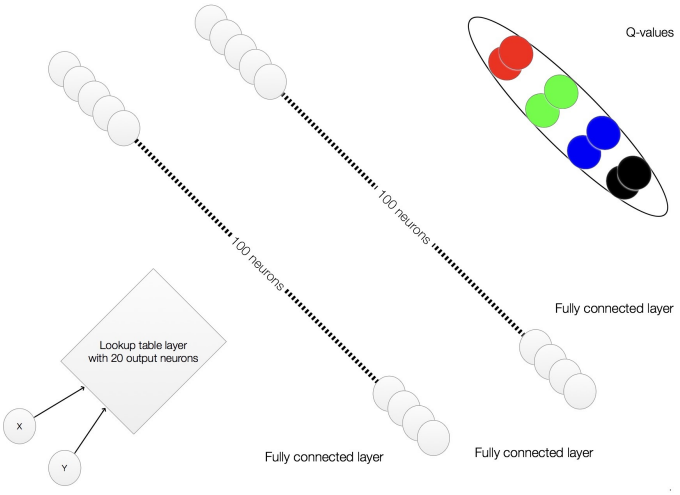
Figure 4.1: Models used for Map and Image versions of deep sea treasure world.

The experiments can be split into three types; the first type is the one consisting of only the experiment that used Q-table. The second type is when the experiments were run on the map version of the problem, while the third type is when the experiments were on the image version.

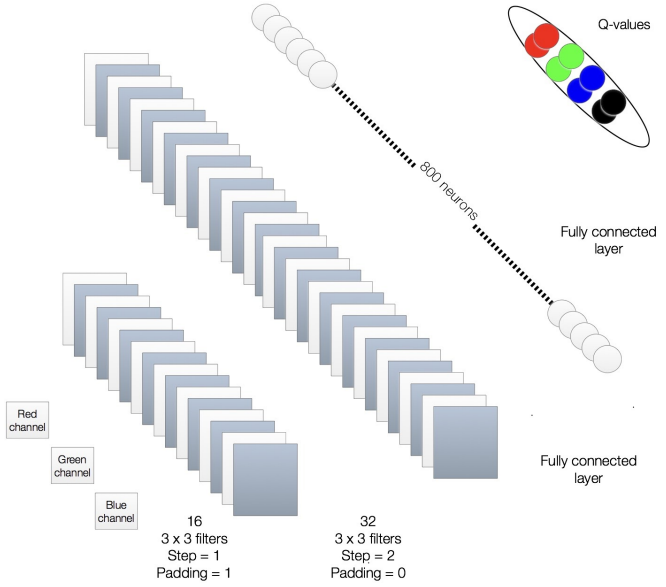
4.2.1 DQN architectures

For experiments applied to the map version of the problem, we used an MLP as the base neural network for the DQN. The MLP had two hidden layers each with 100 neurons. The input to the MLP is the output of the lookup table applied to the x and y locations of the submarine. The main reason for the use of the lookup table function is to decorrelate the input. This architecture is depicted in figure 4.2a. As for the case when the input is the image of the state, we used a CNN as the neural network underlying DQN. The images representing the state are of size 11 $\times$ 11 with three channels depth (RGB channels). As stated in section 3.4, the submarine was represented in green, the goals were in red, the walls were in black, and free cells were in white. The CNN used for the image version had an input layer, two convolutional hidden layers, one fully connected layer and an output layer. The input layer was the three channels of 11 $\times$ 11 image representing the state. The first hidden layer contained 16 (3×3) filters which moved by a step of 1 and had a 1 cell padding on each side. The second convolutional layer had 32

(3×3) filters which moved by a step of 2 and had no padding. This architecture is represented in figure 4.2b.



(a) Map version using MLP.



(b) Image version using CNN.

Figure 4.2: Architectures of DQN used for Map and Image versions of deep sea treasure world.

4.2.2 Hyper-parameters

	Map Version		Image Version
	Q-table	DQN	DQN
Episodes	6000	10000	15000
Steps per episode	1000	1000	1000
Target network update	-	100	100
Replay memory size	-	1e+5	1e+5
γ	0.97	0.97	0.97
Start ϵ	1	1	1
End ϵ	0.05	0.05	0.05
ϵ annealing end episode	3000	5000	7500
Batch size	1	32	32
Initialisation	Zero Vectors	Random Weights	Random Weights

Table 4.1: Single-objective solver hyper-parameters

	Map Version	Image Version
τ	0	0

Table 4.2: OLS Hyper-parameters

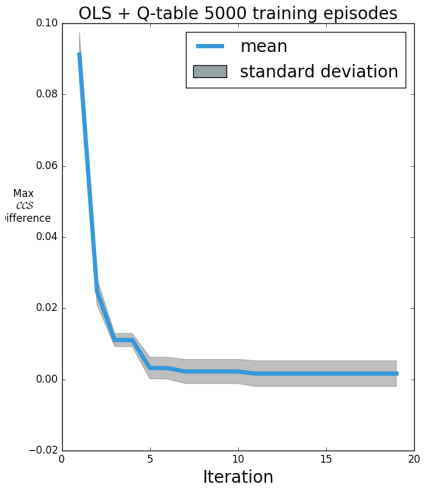
4.3 Results

In this section, we present the results obtained from running the above-mentioned experiments. We also start by providing a brief comparison between each experiment on both the map and image versions. Last but not least, we provide a thorough discussion and comparison between the obtained results from all the experiments.

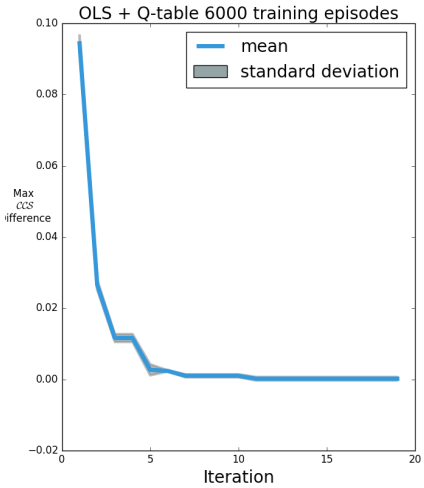
4.3.1 OLS with Q-table

In order to test the applicability of OLS in the RL setting, we experimented with the integration between OLS and Q-table lookup. The experiments have shown very accurate results for the integration between OLS and Q-table. This is depicted in figure 4.3. In the experiments, the combination between OLS and Q-table

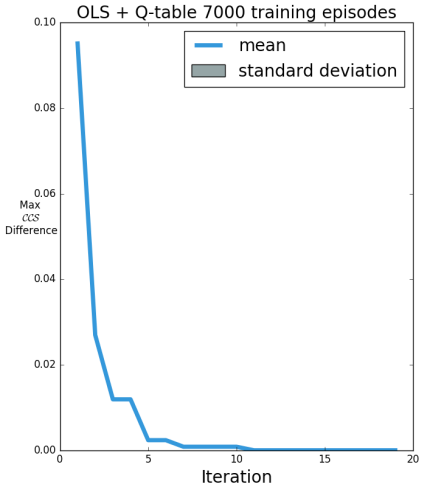
lookup managed to find the whole $\mathcal{CCS}$ 100% of the time if the number of episodes used is 7000 or above. When the number of episodes is 5000 or 6000 the true $\mathcal{CCS}$ is found 79% and 91.7% of the time respectively.



(a) 5000 Episodes.



(b) 6000 Episodes.

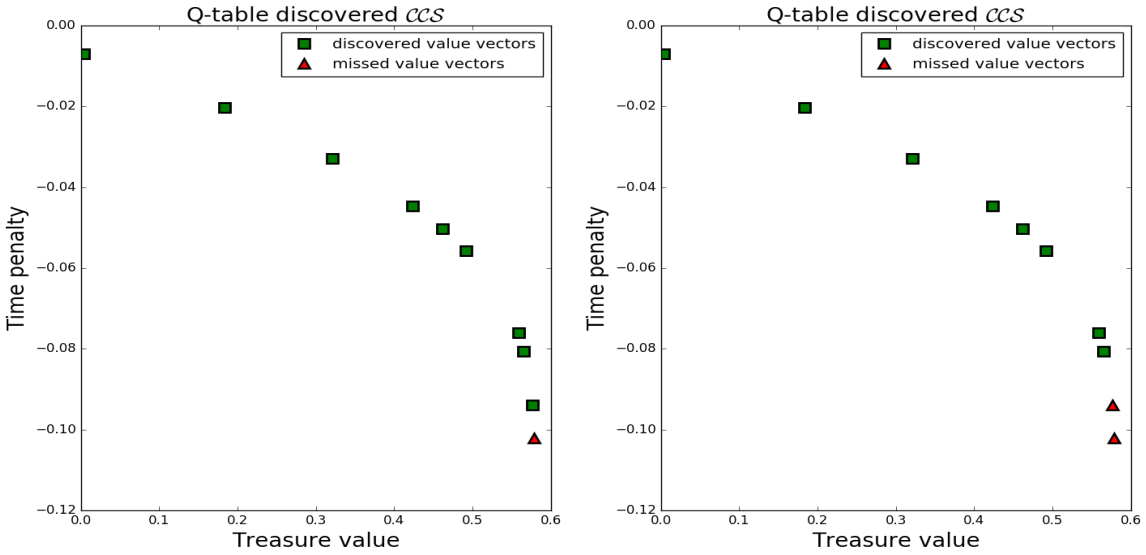


(c) 7000 Episodes.

Figure 4.3: OLS + Q-table Max $\mathcal{CCS}$ Difference.

This experiment showed that OLS can be used successfully within the RL paradigm. In figure 4.3c we see that the true $\mathcal{CCS}$ is found with 0% error on the

value estimates 100% of the time. The algorithm then terminates in 19 iterations which corresponds to the number of corner weights that the OLS had to invoke Q-table at their scalarised instances. It also showed that it takes a considerable number of episodes to find the true $\mathcal{CCS}$. After investigation, it appeared that this is because Q-table would usually get stuck finding sub-optimal goals while learning. The cases in which the true $\mathcal{CCS}$ was not found were split into two cases which are presented in figure 4.4.



(a) Case 1 $\mathcal{CCS}$. (b) Case 2 $\mathcal{CCS}$.

Figure 4.4: OLS + Q-table two classes of errors.

The errors are split into two cases, the first one is when only the last value vector is not discovered. This case is represented in figure 4.4a. The other case is when the last two value vectors are not discovered, and this case is represented in figure 4.4b.

These two classes of error have shown that Q-table lookup needed more than 6000 episodes to be able to accurately approximate the last two value vectors. In the experiments in which Q-table lookup did not accurately estimate the $\mathcal{CCS}$, it always got stuck reaching suboptimal goals, therefore, it could not find and accurately approximate the true optimal goal. These two cases represent the two

scenarios that were faced by Q-table lookup that caused it to miss-approximate the true $\mathcal{CCS}$ when 5000 or 6000 training episodes were used.

4.3.2 DOL

A problem that exists within the previous experiment is that it used the Q-table lookup algorithm which is not a practical algorithm because it lacks the generalisation power. Therefore, we used a DQN with the architectures depicted in figure 4.2 instead. The main advantage of DQN over Q-table is the generalisation power it has. One major difference between DQN and Q-table lookup algorithms is that DQN is less accurate than Q-table.

4.3.2.1 Map Version

An MLP was employed for this task. The architecture used for this task which is depicted in figure 4.2a had a lookup table to decorrelate the inputs. We tried four different architectures

- MLP without table lookup and with two hidden layers each with 10 neurons,
- MLP without table lookup and with two hidden layers each with 20 neurons,
- MLP with table lookup and with two hidden layers each with 50 neurons, and
- MLP with table lookup and with two hidden layers each with 100 neurons.

However, through experimental evaluation, the fourth architecture exhibited optimal performance, while other architectures that did not make use of a lookup table or had fewer neurons in the other hidden layers exhibited much worse performance. The max $\mathcal{CCS}$ difference across all 24 experiments with the fourth architecture is depicted in figure 4.5 and the worst and best results found are depicted in figure 4.6.

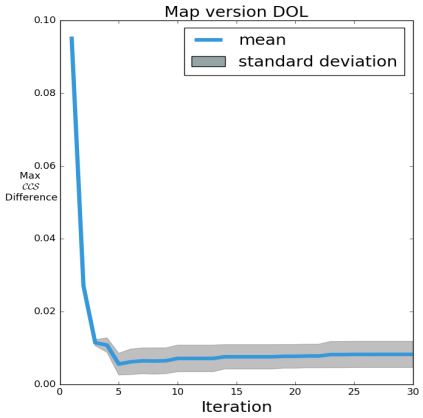
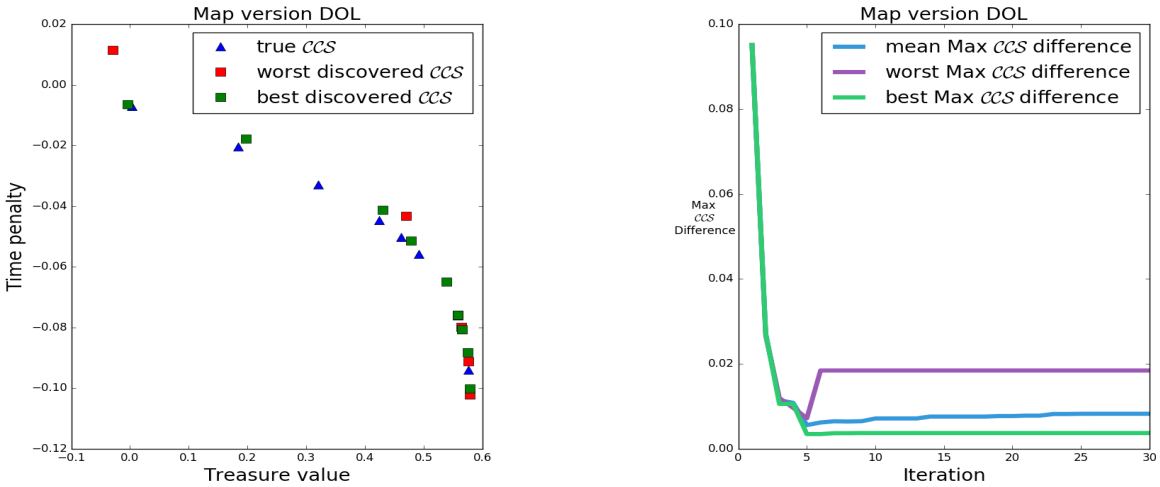


Figure 4.5: Mean and standard deviation of max $\mathcal{CCS}$ difference for map version DOL.



(a) $\mathcal{CCS}$. (b) Max $\mathcal{CCS}$ difference.

Figure 4.6: Extreme results for map version DOL.

A problem that was revealed by these experiments is that the final max $\mathcal{CCS}$ difference usually had a value greater than some of the previous iterations through out learning. This is clear in figure 4.5. After further analysis of the results, we discovered that DQN had lower accuracy at the later stages of learning. This is especially clear for the worst obtained max $\mathcal{CCS}$ difference depicted in figure 4.6b,

where the worst result terminated with more than the double of the max $\mathcal{CCS}$ error it had at the fifth iteration of DOL. More on this point in section 4.3.5. Another interesting thing is that for the worst result, the furthest goal was accurately approximated while the closest one was not. This can be seen in figure 4.6a. When this experiment was further examined, it appeared that an accurate estimation of the closest goal was removed by the wrong final estimation of that goal. This takes us back to the problem of the increasing inaccuracy of DQN as learning goes on.

4.3.2.2 Image Version

We employed the CNN architecture depicted in figure 4.2b to handle the image version of the problem. Multiple other architectures have been tested. However, one thing that was noticed is that the bigger the size of the filters used, the higher the max $\mathcal{CCS}$ difference is. Therefore, the CNN made use of 3×3 filters only. The increase in error was especially clear when the first layer used 4×4 or 5×5 filters. For this case, DOL was never able to detect the existence of the last two goals and it always missed them. On the other hand, an increase in the size of the filters used in the second layer had much less effect on the error. Nevertheless, the error still increased when 4×4 or 5×5 filters were used in the second layer. Similar results were achieved when the filters' step sizes were increased to two cells instead of one in the first layer. Also, image padding with zero valued cells had minor effect on the final result, but it was always slightly better to use padding. We attribute this behaviour to the problem specification as the map used had a very small size, so smaller filters would work better. A plot of the mean max $\mathcal{CCS}$ difference using the chosen CNN architecture is depicted in figure 4.7 and plots of the worst and best results are in figure 4.8.

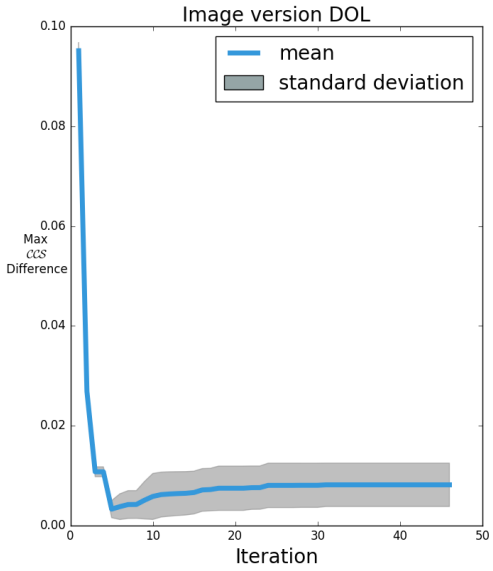
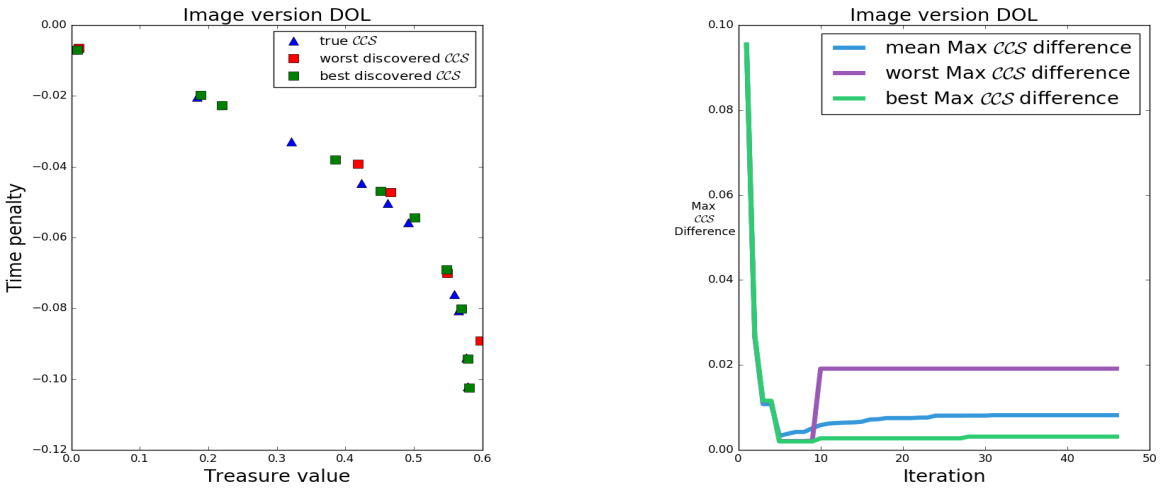


Figure 4.7: Mean and standard deviation of max $\mathcal{CCS}$ difference for image version DOL.



(a) $\mathcal{CCS}$. (b) Max $\mathcal{CCS}$ difference.

Figure 4.8: Extreme results for image version DOL.

Similar to the map version experiments, the image version experiments also suffered from a similar increase in max $\mathcal{CCS}$ difference towards the end of learning.

This can be seen in figure 4.7. However, the effects of this problem were more drastic on the image DOL. This can be seen in figure 4.8b, where the worst result had approximately a 10x increase in the max $\mathcal{CCS}$ difference just after one value vector was miss-approximated. These results show how one single miss-approximation while learning drastically changes the final $\mathcal{CCS}$ and consequently the max $\mathcal{CCS}$ difference.

4.3.2.3 Map versus Image Versions

One very interesting finding is that the mean max $\mathcal{CCS}$ error for the image version was a bit less than the one for the map version. This is depicted in figure 4.9. This could be attributed to the architectures chosen for the DQN, but it is still very interesting because the image version of the problem is inherently more difficult than the map version. Moreover, in all the experiments, the max $\mathcal{CCS}$ error exhibited an increase in value towards the end of training. This is because towards the end of learning the discovered value vectors were much less accurate than the ones discovered during the beginning of learning. This forced DOL to discard many of the accurate value vectors discovered during the start of learning resulting in an increase of error. This problem is carefully investigated in section 4.3.5. Another distinguishable and expected finding is that the number of iterations done by DOL for the image version was almost always more than it used for the map version.

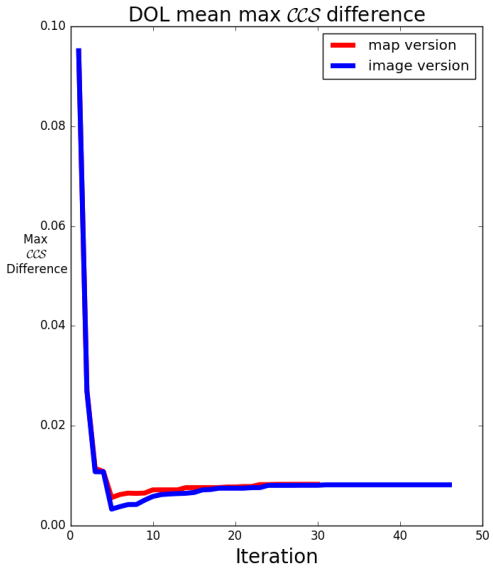


Figure 4.9: Mean max $\mathcal{CCS}$ error for map and image versions with no reuse of learnt DQN.

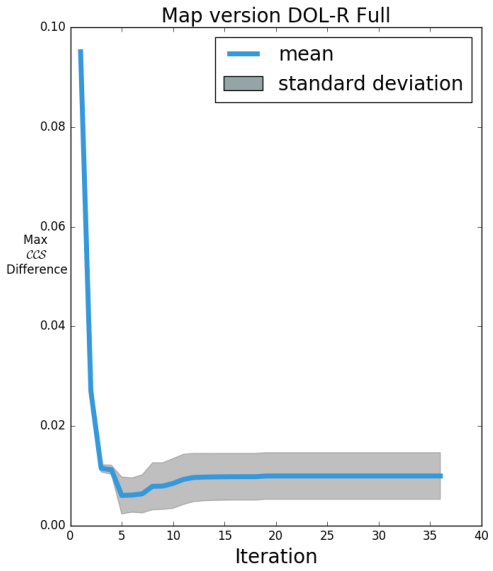
4.3.3 DOL-R Full

One major bottleneck with DOL is that it takes a very long time to terminate. The map version took an average running time of 3 hours to find the whole $\mathcal{CCS}$. As for the image version it took an average of 11 hours. The ability to reuse already learnt DQNs for scalarised instances of the problem in hand to hot-start the learning of new instances would increase the potential of DOL a lot. As previously mentioned in section 3.2, this could help direct the search for the optimal policy starting from an already good approximate of it, speeding up the search for the optimal policy a lot. To test the effectivity of DOL-R Full against DOL, the previous experiments were done again, but with the full reuse of DQNs associated with already learnt policies.

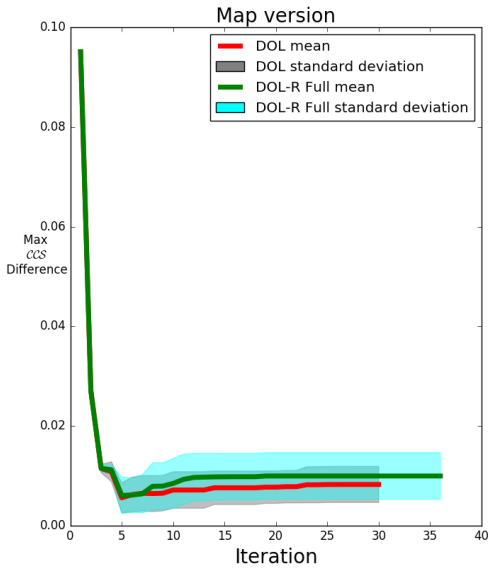
4.3.3.1 Map Version

The experiments done on the map version exhibited faster training time. They took an average of 1 hour 40 minutes to discover the whole $\mathcal{CCS}$. However, they

exhibited an increase in the mean max $\mathcal{CCS}$ difference across the 24 experiments. This is depicted in figure 4.10a. This is because, during many of the runs of DQN, the total number of episodes would be reached before DQN has approximated the optimal value vector. This forced DOL-R Full to disregard the discovered value vector as it is not optimal, therefore, it terminates with a suboptimal $\mathcal{CCS}$. This can be seen from the increase in the standard deviation around the mean max $\mathcal{CCS}$ error depicted in figure 4.10b. The reduction in time for these experiments was mainly because each DOL-R Full iteration was faster. This is because it was much faster to train DQN. The max $\mathcal{CCS}$ difference across all 24 experiments is depicted in figure 4.10a and the worst and best experiments are depicted in figure 4.11.



(a) Mean and standard deviation of max $\mathcal{CCS}$ difference for map version DOL-R Full.

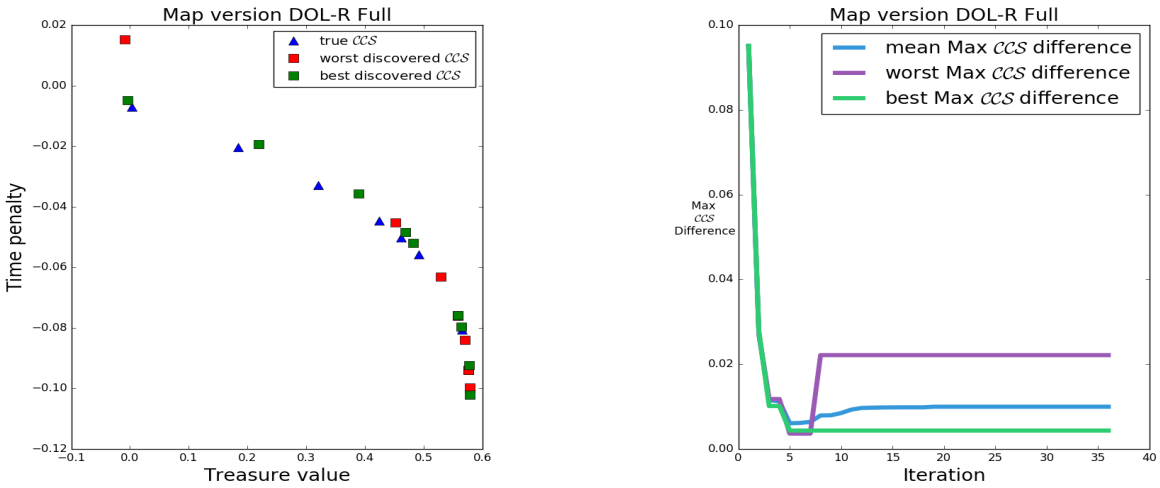


(b) Mean and standard deviation of max $\mathcal{CCS}$ difference for map version DOL and DOL-R Full.

Figure 4.10: Mean and standard deviation of max $\mathcal{CCS}$ difference for map version DOL and DOL-R Full.

Figure 4.10a shows that the problem of increasing max $\mathcal{CCS}$ difference at the end of learning still persists for DOL-R Full. As for figure 4.10b, it compares the mean and standard deviation achieved for both DOL and DOL-R Full on the map

version of deep sea treasure world. It shows that both the mean and standard deviation have slightly increased. Also, it shows that the max number of calls to DQN has increased from 30 calls to 36.



(a) $\mathcal{CCS}$. (b) Max $\mathcal{CCS}$ difference.

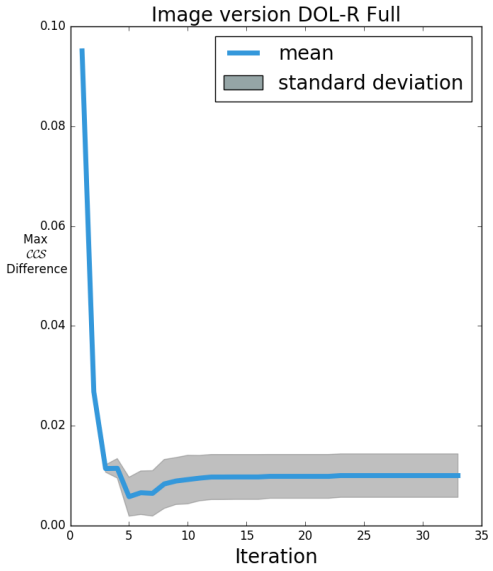
Figure 4.11: Extreme results for map version DOL-R Full.

Figure 4.11 shows a comparison between the best and worst results for DOL-R Full. Figure 4.11a shows that the optimal experiment managed to discover better policies for the three closest treasures, while the experiment with the worst result underestimated the time penalty taken for the first treasure, therefore, it could not discover the true optimal policies for the next two treasure values. This is the same problem that happened for the experiment with the worst result in the map version DOL.

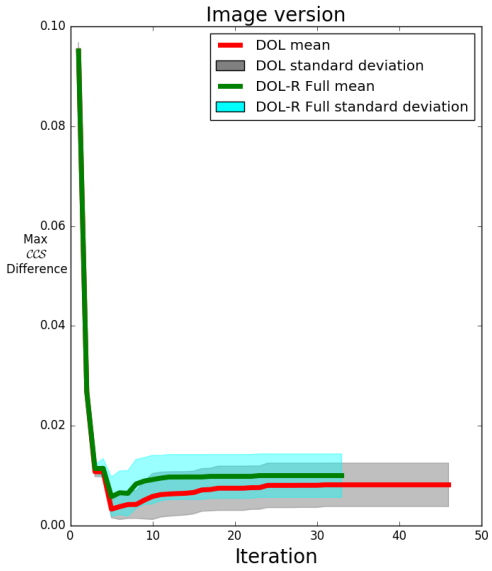
4.3.3.2 Image Version

As for the image version, the training time was reduced from an average of 11 hours to an average of 7 hours except for two experiments which took around 10 hours. However, unexpectedly this is not mainly attributed to a reduction of training time per one iteration of OLS. It is mainly attributed to the fact that there were fewer calls to DQN, thus, less overall time. The max $\mathcal{CCS}$ difference

across all 24 experiments is depicted in figure 4.12a and the worst experiment is depicted in figure 4.13.



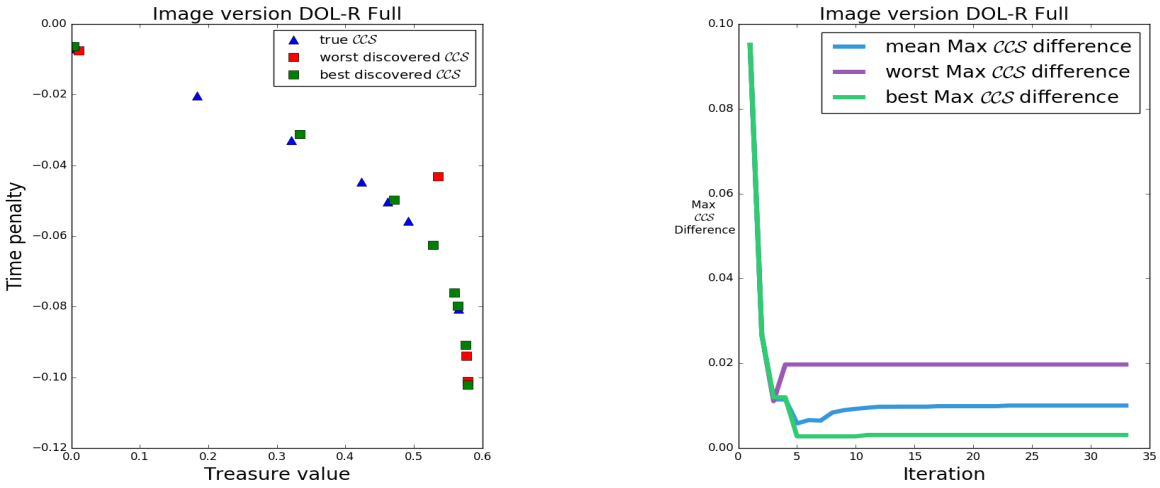
(a) Mean and standard deviation of max $\mathcal{CCS}$ difference for image version DOL-R Full.



(b) Mean and standard deviation of max $\mathcal{CCS}$ difference for image version DOL and DOL-R Full.

Figure 4.12: Mean and standard deviation of max $\mathcal{CCS}$ difference for image version DOL and DOL-R Full.

Similar to the map version, DOL-R Full on the image version still exhibited an increase in the mean max $\mathcal{CCS}$ difference towards the end of learning. This can be seen in figure 4.12b. On the other hand, the image version DOL-R Full exhibited a slight reduction in the standard deviation compared to the image version DOL.



(a) $\mathcal{CCS}$. (b) Max $\mathcal{CCS}$ difference.

Figure 4.13: Extreme results for image version DOL-R Full.

Figure 4.13 shows the best and worst results for DOL-R Full on the image version. Comparing figure 4.13a to figure 4.8a, we can see that the image version with DOL-R Full achieved a worse final $\mathcal{CCS}$ than with DOL. When the worst experiment for DOL-R Full was further examined, it appeared that it exhibited inaccurate results from the first iterations of DOL-R Full. The first two iterations had very good approximations of the furthest and closest goals, but after that, the accuracy dropped significantly, and the weight simplex was not adequately explored.

4.3.3.3 Map versus Image Versions

Both experiments, map version and image version, exhibited very similar results with full reuse of already learnt policies. This is depicted in figure 4.14. Also, both of them retained the problem of terminating with a higher max $\mathcal{CCS}$ difference than what they obtain during the training. Another unexpected finding was that in many situations when the DQN used for initialization already corresponded to the optimal policy, the miss-approximation error increased significantly. We attribute this problem to the fact that in this situation, DOL-R Full had many episodes to use for learning that it did not need, therefore, the inaccuracy of

DQN accumulated and the optimal policy was miss approximated. However, the main and unexpected gain from these experiments is that in the case of the image version, the maximum number of iterations needed by DOL-R Full to terminate was significantly less than the one needed for DOL and without much loss in accuracy. This means that it is perfectly suitable to fully reuse already learnt DQNs given that a small loss of accuracy is acceptable. Moreover, figure 4.14 shows that DOL-R Full exhibited very similar results on both the image version and map version of the problem. This shows that DOL-R Full is not affected by the significant increase in complexity when switching from the map version of deep sea treasure world to the image version.

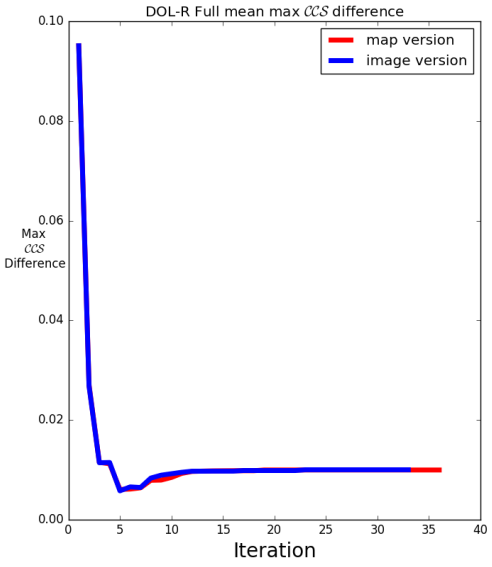


Figure 4.14: Mean max $\mathcal{CCS}$ error for map and image versions with full reuse of learnt DQN.

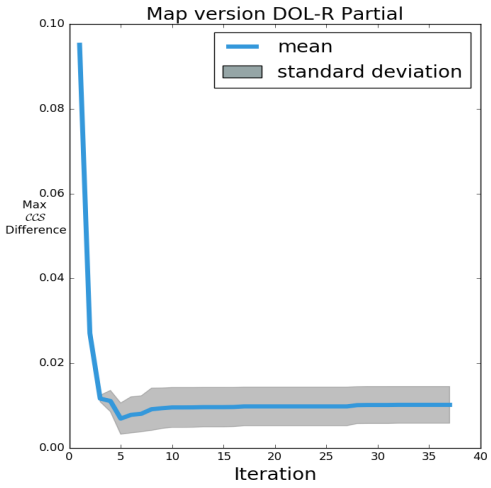
4.3.4 DOL-R Partial

Although DOL-R Full exhibited much faster training times, the results were not as good as expected. After further analysis of the experiments of DOL-R Full, we discovered that for many experiments, at the beginning of training the DQN for any new corner weight for DOL-R Full experiments, DQN was usually misinformed,

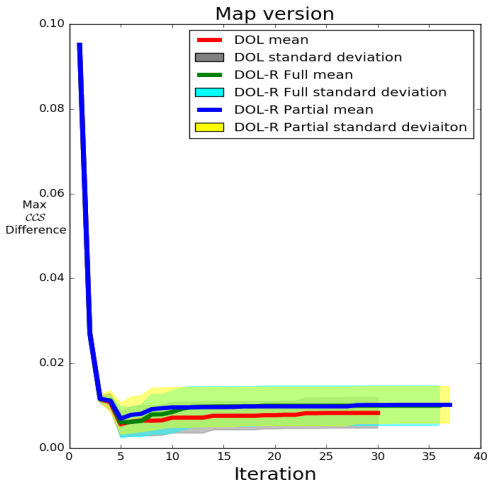
i.e., it wasted many episodes following the policy by which it was initialised without learning the accurate value vectors. This is because, unlike DOL, the policies used for initialisation do lead to a goal which when reached terminates the learning procedure. This forced DQN to reach a suboptimal goal many episodes because of its initialisation causing DQN to waste many episodes without learning. The use of ϵ-greedy policy starting from 1 did reduce the effects of this problem a lot. But, nevertheless, DQN still suffered from it.

4.3.4.1 Map Version

The changes in the results for the map version between DOL-R Full and DOL-R Partial were minor. The average training time increased a little bit than the one needed for DOL-R Full. Also, the max $\mathcal{CCS}$ difference increased slightly. This can be seen in figure 4.15b, however, there is a considerable overlap between the lines representing the mean for DOL-R Full and DOL-R Partial. Also, the standard deviation at the end of training was slightly reduced, this can be seen in figure 4.15b. The standard deviation also suffered from a considerable overlap with both standard deviations of DOL and DOL-R Full. The results from figure 4.15 show that there was no benefit from using partial reuse over full reuse in the map version of deep sea treasure world. The max $\mathcal{CCS}$ difference across all 24 experiments is depicted in figure 4.15a and the worst and best experiments are depicted in figure 4.16.

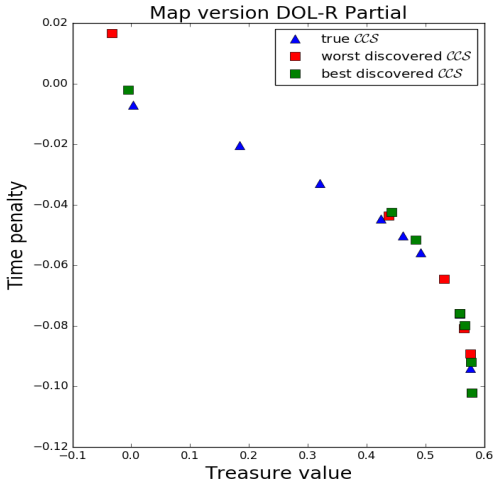


(a) Mean and standard deviation of max $\mathcal{CCS}$ difference for map version DOL-R Partial.

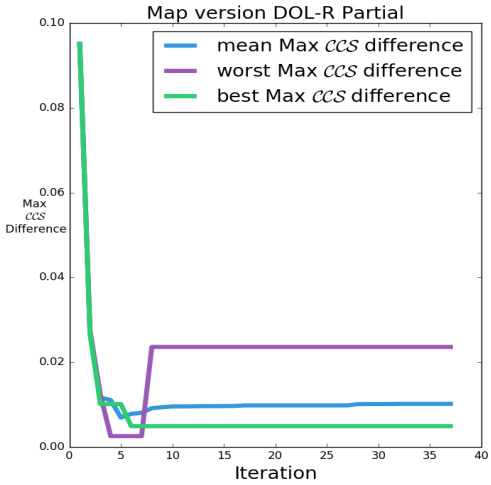


(b) Mean and standard deviation of max $\mathcal{CCS}$ difference for map version DOL, DOL-R Full and DOL-R Partial.

Figure 4.15: Mean and standard deviation of max $\mathcal{CCS}$ difference for map version DOL, DOL-R Full and DOL-R Partial.



(a) $\mathcal{CCS}$.



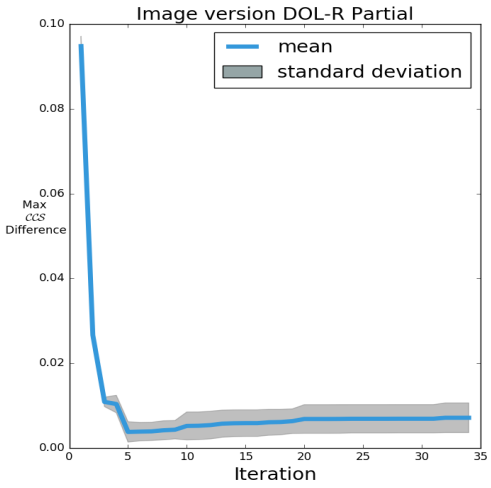
(b) Max $\mathcal{CCS}$ difference.

Figure 4.16: Extreme results for map version DOL-R Partial.

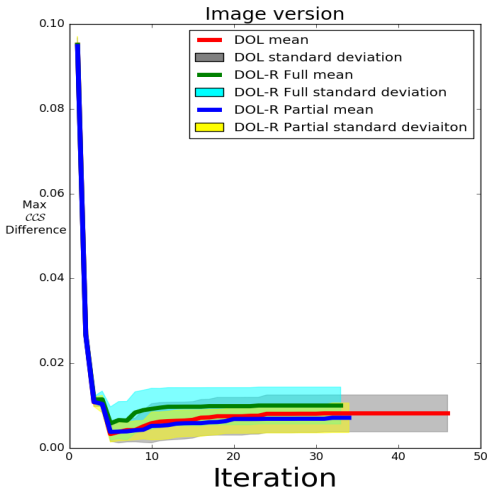
DOL-R Partial exhibited a similar increase in max $\mathcal{CCS}$ error to the one in DOL-R Full. However, one interesting finding is that the best discovered $\mathcal{CCS}$ for DOL-R Partial depicted in figure 4.16a is worse than the one discovered for DOL-R Full depicted in figure 4.11a. This is interesting because theoretically speaking, in DOL-R Partial, DQN was less prone to getting stuck reaching suboptimal goals because it is no more following a policy that leads to a terminating goal. Thus, it had more episodes to learn to go to the true optimal goal. We attribute this to the architecture used for all map version experiments. The reason we are doing so is that all the map version experiments had a considerable similarity across all types of DOL indicating the existance of some limiting factor outsie DOL.

4.3.4.2 Image Version

On the contrary, the image version exhibited an improvement with partial reuse. The max $\mathcal{CCS}$ error was reduced without a great increase in the time needed for training from the one needed for the full reuse. Also, the standard deviation was reduced as well. This is depicted in figure 4.17b and it also exhibited a better worse case policy than the map version's worst case one. This result is very interesting because it is significantly harder to use the image version than the map version. The max $\mathcal{CCS}$ difference across all 24 experiments is depicted in figure 4.17a and the worst and best experiments are depicted in figure 4.18.

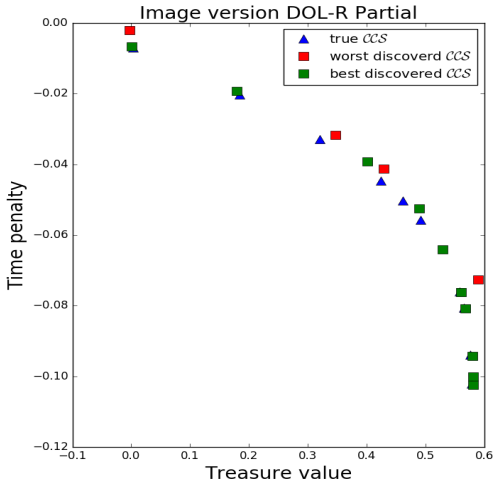


(a) Mean and standard deviation of max $\mathcal{CCS}$ difference for image version DOL-R Partial.

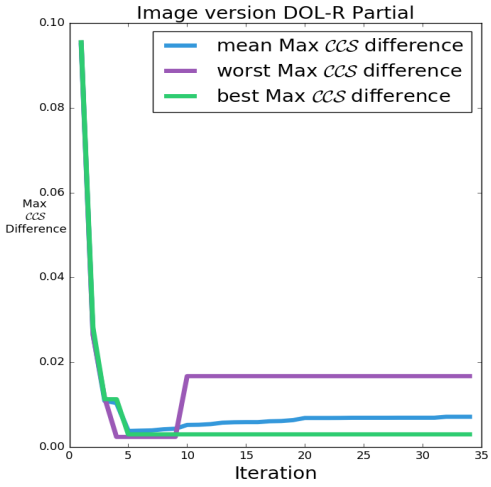


(b) Mean and standard deviation of max $\mathcal{CCS}$ difference for image version DOL, DOL-R Full and DOL-R Partial.

Figure 4.17: Mean and standard deviation of max $\mathcal{CCS}$ difference for image version DOL, DOL-R Full and DOL-R Partial.



(a) $\mathcal{CCS}$.



(b) Max $\mathcal{CCS}$ difference.

Figure 4.18: Extreme results for image version DOL-R Partial.

Contrary to the preliminary expectations, DOL-R Partial did better on the image version of deep sea treasure world than on the map version. We expected a far better result for the map version than on the image version. However, it appears that the idea of partial reuse enabled DQN to dedicate more episodes to learning the optimal policy. It turned out that partial reuse is similar to the idea of translation invariance for CNN, but across different runs of DOL-R Partial. As stated in section 2.2.4, the main advantage of CNN is that each filter learns to detect the same feature across the whole image. DOL-R Partial did not alter what the filters learnt, thus making proper use of previously learnt feature detectors. This enabled DOL-R Partial to dedicate more episode to finding the optimal policy rather than learning the features needed to behave optimally.

4.3.4.3 Map versus Image Versions

Map version DOL-R Partial and image version DOL-R Partial exhibited very different results. While the map version achieved its worst result when it used partial reuse, the image version achieved its best result with partial reuse. The main and constant thing amongst all the previous experiments is that none of them managed to fix the problem of the increase of max $\mathcal{CCS}$ difference in the late stages of training. Also, we infer from the fact that in figure 4.19, the mean max $\mathcal{CCS}$ error for the image version is lower than that of the map version, that similar to DOL-R Full, the benefit of DOL-R Partial is not affected by the increase of the state space.

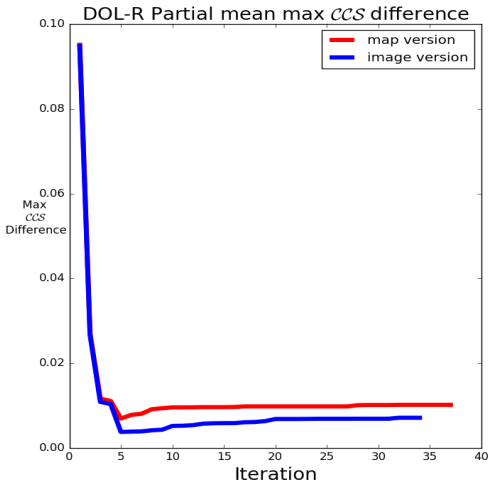
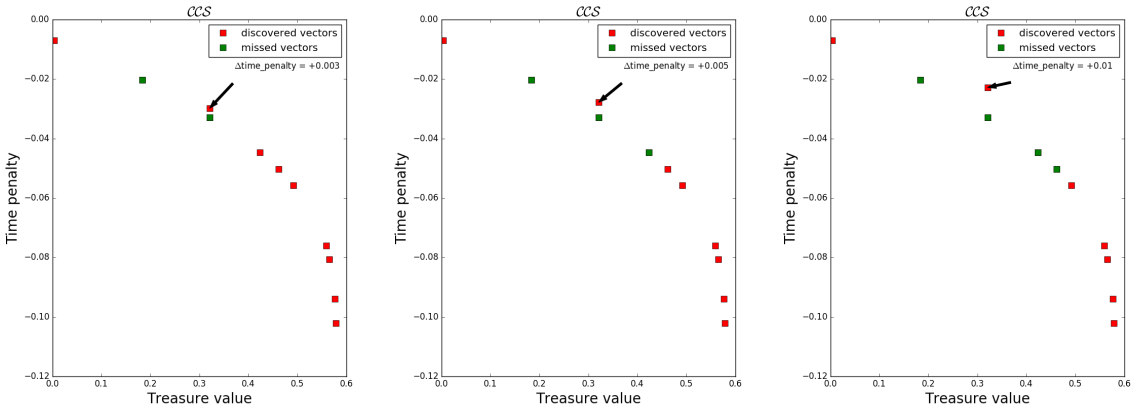


Figure 4.19: Mean max $\mathcal{CCS}$ error for map and image versions DOL-R Partial.

4.3.5 Discussion

The deep sea treasure world problem possess one very interesting challenge when considered in the multi-objective setting of RL with an approximate solver. The challenge is that the true value vectors represented in figure 3.6 are very close to each other, and the accuracy of the single-objective solver has a great effect on the final discovered $\mathcal{CCS}$. As explained in section 4.3.1, it is important to run the single-objective solver for an adequate number of episodes to make sure it finds the true optimal value vectors and approximates them correctly. Because, if the single-objective solver approximates only one of the value vectors poorly this may greatly affect the final discovered $\mathcal{CCS}$. There are two cases for the solver to badly approximate the true value vectors, it could either overestimate the actual value vector or underestimate it in either of the objectives.

In the case of overestimation depicted in figure 4.20, a minor overestimation of any of the value vectors drastically changes the $\mathcal{CCS}$, this results in excluding many correct value vectors.



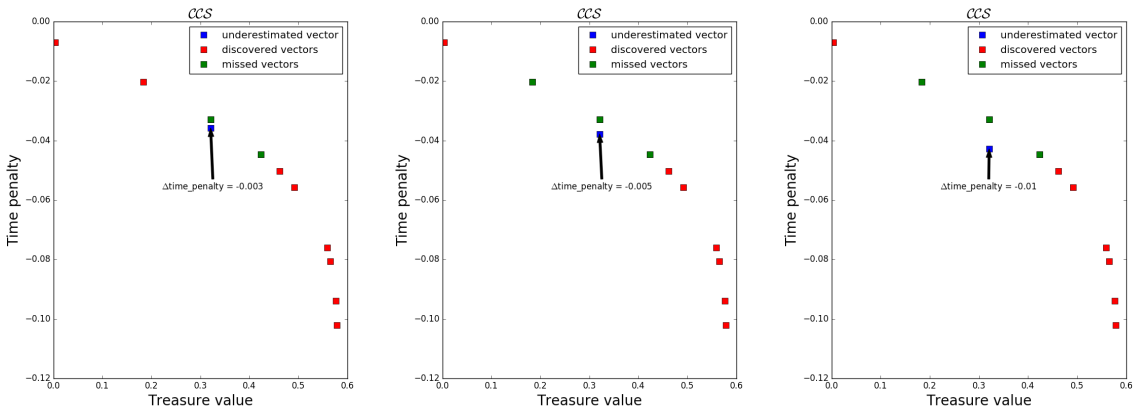
(a) Inaccuracy within 0.003 (b) Inaccuracy within 0.005 (c) Inaccuracy within 0.01

Figure 4.20: Overestimation in the overall value vectors (by underestimating the negative value of the time penalty).

In figure 4.20, the solver estimates the time penalty at one of the value vectors to be better than the original one. This makes the overestimated value vector overshadow other value vectors which even if discovered would no longer be on the $\mathcal{CCS}$. This means that for the weights for which the other overshadowed vectors are optimal if the agent would follow the overestimated value vector's policy, it would find a worse goal than it could have found, had it estimated the overestimated value vector correctly. figure 4.20a has a max $\mathcal{CCS}$ difference value of 0.0027, while figure 4.20b has max $\mathcal{CCS}$ difference of 0.0045 and figure 4.20c of 0.0091. Also, if the overestimation happens in multiple vectors at the same time, this might yield a completely different $\mathcal{CCS}$ that looks far from the original one. This problem is also applicable to the case when the overestimation is in any of the objectives.

On the other hand, underestimation of value vectors is equally problematic when it comes to discovering the true $\mathcal{CCS}$. The underestimation of one value vector makes OLS miss the corner weights which would have been added had this value vector been estimated correctly. This might lead to the scenario in which OLS completely ignores some part of the weight simplex which otherwise would have been checked and further explored to find possible value vectors within this

section. These scenarios are depicted in figure 4.21.



(a) Inaccuracy within 0.003

(b) Inaccuracy within 0.005

(c) Inaccuracy within 0.01

Figure 4.21: Underestimation in value vectors (by overestimating the negative value of the time penalty).

In figure 4.21, the solver estimates the time penalty at one of the value vectors to be worse than the original one. This makes OLS ignore the underestimated value vector and not add it to the $\mathcal{S}$ when discovered and thus it's corner weights will not be added to the list of corner weights to explore. This means that there will be a part of the weight simplex which will not be further explored. Thus a whole range of optimal value vectors and their policies will not be found. Figure 4.21a has a max $\mathcal{CCS}$ difference of 0.0016, while figure 4.21b and figure 4.21c have a max $\mathcal{CCS}$ difference of 0.0032.

The effects of overestimation and underestimation differ according to which value vector in the $\mathcal{CCS}$ they affect. For instance, for a problem with two objectives, if the value vector affected by them is the third one (i.e., the one found at the first discovered corner weight between the two extrema), the discovered $\mathcal{CCS}$ could either become the one consisting of only the first two value vectors (in case the third one is heavily underestimated) or could consist of the third value vector only (in case the third value vector is heavily overestimated). In both cases, the weight simplex will not be further explored causing a significant difference between the discovered $\mathcal{CCS}$ and the real one. But if the single-objective solver has a constant

behaviour, either always overestimating or always underestimating, this does not cause any problems. The $\mathcal{CCS}$ that it will discover will be a translated variant of the original one, so optimal policies will be the same. On the contrary, if the solver does not have a constant behaviour, the effects of overestimation and underestimation accumulate leading to a significant diversity in the discovered $\mathcal{CCS}$s. This is exactly the case for DQN, and hence, DOL and DOL-R. Another thing that is very important to notice is that the more objectives a problem has, the more drastic the effects of overestimation and underestimation are. This is because in higher dimensions each value vector will have a wider range of effect over the weight simplex resulting in major changes in the $\mathcal{CCS}$. One major difference between the effects of overestimation and the effects of underestimation is that the effects of underestimation can be fixed in later episodes if a more accurate value vector is found. However, for overestimation, there is no way for DOL to fix it.

The effects of overestimation and underestimation are not specific to deep sea treasure world, but they are deliberately further empowered by this problem's setting. Therefore the ability to perfectly solve this problem enforces the need to have accurate single-objective solvers. In practice, however, DQN managed to do a relatively good job approximating the true value vectors, but one thing was very noticeable about the approximation problem which is that it tends to take effect towards the end of building the $\mathcal{CCS}$. The main reason for this delay is that corner weights with high estimated improvement are explored first. This method of selecting value vectors leads to a steep reduction in the max $\mathcal{CCS}$ difference error during the first stages of learning. This can be seen in figure 4.5, figure 4.7, figure 4.10a, figure 4.12a, figure 4.15a and figure 4.17a, where the major decrease in error happens in the first five discovered value vectors. This is because this method of picking corner weights ensures that value vectors that have biggest added difference to the convex upper surface are checked first, and those value vectors are typically much easier to approximate. Another reason for the delay in miss-approximations is that at the beginning of learning, the weights are in a sense very optimal for learning, i.e., they are very close to the corner weights that provide the largest difference between the value vector to learn and the current optimal value vector in $\mathcal{S}$. On the other hand, at later iterations of DOL, DQN

has already been run many times and the miss-approximations have already accumulated leading the corner weights which do not provide a big enough difference for the value vector to learn.

The miss-approximation problem leads to another problem across all the experiments which is that they all ended up with larger max $\mathcal{CCS}$ error than what they had at many earlier steps during the training. This is obvious from the same plots as the mean error always starts to increase at some point during learning. This is due to the miss-approximations done by DQN in the later stages of learning as the overestimated vectors remove the other better value vectors from $\mathcal{S}$ leading to an increase in error.

These two problems can be solved by changing τ in DOL. This method, however, has a major drawback, picking τ is not very straight forward. If τ is too large, the algorithm would be risking ignoring corner weights which can actually provide a better $\mathcal{CCS}$. On the other hand, if τ is too small, it becomes useless and does not fix the problem itself. In the experiments we had, we tested τ for value 10^{-3} and 10^{-4}, but they did not have a constant behaviour, so τ was set to 0 across all experiments.

Another problem that was unveiled during the experiments is related to the use of extremum weights. In OLS and DOL, the extremum weights are added with priority ∞. The definition of extremum weights as described in section 2.3.4, is putting the whole weight on one objective with a complete disregard for the others. This is acceptable in planning, but it is bit problematic in learning. For instance, in the treasure world, if the whole weight is put on the treasure reward value, then the agent will try to get the maximum treasure reward no matter how many time steps it takes. However, because the maximum treasure reward is 19 steps away from the initial goal, this means that it is decayed by the factor γ^{18}. So, if the agent would take at least one wrong action on its way to reaching the goal, the decaying factor would be at least γ^{19}. This difference in the decaying factor can easily be detected by planning algorithms, but it is really small and really hard to detect using an approximate solver like DQN. However, this is not problematic in the single-objective setting. The reason it is not is because in the single-objective setting the goal is to learn the optimal policy not having a very good estimate of the state independent value vector of the start state given that optimal policy. So

in the case of deep sea treasure world, if the target is to maximise the treasure value, then the concept of time penalty is dropped, and any policy that takes the agent to the maximum treasure would be optimal. However, in the multi-objective setting, it is required that the agent would also approximate the other objective correctly even when the weight is put on one objective only. This is because, in the multi-objective setting, the accuracy is of great importance because if the value vectors are not approximated accurately, this would lead to the same problems resulting from underestimation and overestimation of value vectors.

One method was proposed to solve the problem with the extremum weights is to use a value very close to 1 rather than using 1 (e.g. 0.99 or 0.999). This method has the problem that it deliberately cuts off a part from the weight simplex. This leads to DOL completely ignoring that range and it will not be able to detect any better policies within that range. This proposed solution was tested on both the image version and map version of the problem with no reuse of learnt DQNs. This solution did make improvements to the final $\mathcal{CCS}$ produced by OLS, but the results did not have a stable behaviour. For that reason and for time constraints, this problem was addressed as if it was associated with the neural network architecture not being able to approximate the true value vector correctly. Therefore, the number of neurons in the hidden layers were increased to help DQN overcome this problem.

Chapter 5

Conclusions

In this chapter, we summarise the contributions of the work done throughout this project comparing the contributions with the initial aims of the project. Moreover, we briefly mention the achieved results for DOL, DOL-R Full and DOL-R Partial, and we present a summary of the work done. Last but not least, different directions for possible future work will be suggested.

5.1 Contributions

In this research, the main purpose was to investigate the applicability of OLS in the RL paradigm and to build an OLS-based multi-objective deep RL algorithm. We successfully managed to integrate deep RL with OLS, by employing DQN with a neural network that has as many outputs as the number of actions multiplied by the number of objectives. This is important because OLS requires accurate estimates of the value vectors of the optimal policy for all actions for all objectives at each scalarisation weight. Furthermore, we successfully reused the parameters of DQN learnt for previous policies (inspired by earlier work in this direction on MOPOMDP planning), to speed up the learning for subsequent policies. The main intuition behind that is that the neural network underlying DQN encodes the important features for the values of the policies and that these features are likely to be very similar when a slightly different scalarisation weight is used. However, because of the chance of getting stuck while learning and not being able to accurately approximate the true value vectors, it is not best to reuse the entire

neural network. Instead, we use the entire network except for the last layer that controls the final output of the neural network. We have shown experimentally that this works much better in practice, and leads to both faster learning, and a better estimation of the $\mathcal{CCS}$ than DOL. Last but not least, we applied our three proposed algorithms to both the map and image versions of the deep sea treasure world problem, and we conclude that our DOL successfully employs an outer-loop approach to the multi-objective RL setting, and enables the usage of highly successful deep learning techniques in a multi-objective setting. To our knowledge, this is the first algorithm to achieve this. We have shown empirically that the $\mathcal{CCS}$ can be estimated with very high accuracy even in MOMDPs with an extremely large state-space, i.e., MOMDPs with an image state space.

5.2 Summary

We have successfully integrated OLS in the RL paradigm and we established three algorithms; DOL, DOL-R Full and DOL-R Partial. The experiments have shown a considerable success at estimating the true $\mathcal{CCS}$. Across all six classes of experiments that used DQN, they all had a mean error of less than 11%. Furthermore, the experiments have shown a considerable reduction in time needed for training when the DQN was initialised using an already learnt DQN for a different policy. However, unlike the preliminary expectations, reusing already learnt policies on the map version of deep sea treasure world did not show improvement in accuracy. Lastly, during the experimental procedure, it was discovered that several issues act like a bottleneck that is limiting the full capacity of DOL in general. These obstacles have been slightly addressed in chapter 4 and several methods that can solve those obstacles are stated in the future works section.

5.3 Future Work

Given the novelty of this work and the fact that it is the first use of a multi-objective planning algorithm in the RL paradigm, there are multiple directions for future extensions

1. Investigate the possibility of coming up with better ways of picking an already learnt policy from all learnt ones to improve the speed and accuracy of learning. As we have seen in section 4.3.4.2, it is not always good to start from the nearest found optimal policy especially when we are using DQN because it might lead to bigger miss approximation problems.

2. Investigate different possible ways of reusing already learnt DQNs instead of being limited to full and partial reuse. One possible direction is the use of a mixture of learnt policies. This way the chances of getting stuck with a suboptimal policy is reduced.

3. Further investigate the reasons behind the increase in error in the later stages of learning and methods that could prevent that. This is essential because as explained in section 4.3.5, most runs of DOL and its variants do reach a very good estimate of $\mathcal{S}$ while learning, however, this increase of miss-approximation error leads to much worse final $\mathcal{CCS}$. One possible way for that is to try to integrate multiple parallel runs at the same corner weight to increase the probability of reaching the true optimal policy and its value vector.

4. Investigate different possible ways of picking τ that can work independently from the problem. This is very important to reduce the learning time for all variants of DOL by identify the corner weights which are not worth running the single-objective solver at, reducing the overall learning time. One possible way for that is to define τ based on the estimated improvement value for the first generated corner weight. This is because this value would provide a proper starting point to the ranges of expected improvement values.

5. Investigate the possibility of inputting the scalarisation weights into DQN instead of using them to scalarise the multi-objective problem. It would be very interesting to see whether a multi-objective solver can arise and discover the $\mathcal{CCS}$ without any change being made to the multi-objective problem.

References

[1] C. Amato and G. Shani. High-level reinforcement learning in strategy games. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 75–82. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[2] Y. M. Assael. torch-dqn. `https://github.com/iassael/torch-dqn`, 2016. commit: ce67992d59c25de45db2367ea6e39db101abbcbc.

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

[4] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[5] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[6] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun. Very deep convolutional networks for natural language processing. *arXiv preprint arXiv:1606.01781*, 2016.

[7] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.

[8] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Learning hierarchical features for scene labeling. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1915–1929, 2013.

[9] J. N. Foerster, Y. M. Assael, N. de Freitas, and S. Whiteson. Learning to communicate to solve riddles with deep distributed recurrent q-networks. *arXiv preprint arXiv:1602.02672*, 2016.

[10] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

[11] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[12] D. P. Heyman and M. J. Sobel. *Stochastic models in operations research: stochastic optimization*, volume 2. Courier Corporation, 2003.

[13] G. Hinton, N. Srivastava, and K. Swersky. Neural networks for machine learning lecture 6a. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. Accessed: 25-08-2016.

[14] R. A. Howard. Dynamic programming and markov processes.. 1960.

[15] R. Johnson and T. Zhang. Effective use of word order for text categorization with convolutional neural networks. *arXiv preprint arXiv:1412.1058*, 2014.

[16] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[17] M. Lai. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*, 2015.

[18] F. F. Li, A. Karpathy, and J. Johnson. Stanford university cs231n: Convolutional neural networks for visual recognition. `http://cs231n.stanford.edu/`.

[19] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, 2013.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[22] A. Ng, J. Ngiam, C. Foo, Y. Mai, C. Suen, A. Coates, A. Maas, A. Hannun, B. Huval, T. Wang, and S. Tandon. Unsupervised feature learning and deep learning. `http://ufldl.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/`. Accessed: 25-08-2016.

[23] D. M. Roijers. *Multi-Objective Decision-Theoretic Planning*. PhD thesis, University of Amsterdam, 2016.

[24] D. M. Roijers, J. Scharpff, M. T. Spaan, F. A. Oliehoek, M. de Weerdt, and S. Whiteson. Bounded approximations for linear multi-objective planning under uncertainty. In *ICAPS 2014: Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, pages 262–270, June 2014.

[25] D. M. Roijers, P. Vamplew, S. Whiteson, and R. Dazeley. A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research*, 48:67–113, 2013.

[26] D. M. Roijers, S. Whiteson, and F. A. Oliehoek. Point-based planning for multi-objective POMDPs. In *IJCAI 2015: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, July 2015.

[27] J. Scharpff, T. Spaan, L. Volker, and M. De Weerdt. Planning under uncertainty for coordinating infrastructural maintenance. In *Proceedings of the 8th annual workshop on Multiagent Sequencial Decision Making Under Certainty, MSDM-2013, St. Paul, Minnesota, USA, May 7, 2013*, 2013.

[28] D. Silver. Gradient temporal difference networks. In *EWRL*, pages 117–130, 2012.

[29] D. Silver. University college london reinforcement learning course. 2015. `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`.

[30] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[31] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[32] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[33] S. Sukittanon, A. C. Surendran, J. C. Platt, and C. J. Burges. Convolutional networks for speech detection. In *Interspeech*. Citeseer, 2004.

[34] R. Sutton. The reward hypothesis. `http://incompleteideas.net/rlai.cs.ualberta.ca/RLAI/rewardhypothesis.html`. Accessed: 25-08-2016.

[35] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[36] R. S. Sutton. Planning by incremental dynamic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 353–357, 1991.

[37] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[38] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE transactions on automatic control*, 42(5):674–690, 1997.

[39] P. Vamplew, R. Dazeley, A. Berry, R. Issabekov, and E. Dekker. Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine learning*, 84(1-2):51–80, 2011.

[40] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR, abs/1509.06461*, 2015.

[41] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.