Parallel Transfer Learning: Accelerating Reinforcement Learning in Multi-Agent Systems

Adam Taylor

A thesis submitted to the University of Dublin, Trinity College
in fulfilment of the requirements for the degree of
Doctor of Philosophy (Computer Science)

November 2016

Declaration

Adam Taylor

Dated: Tuesday 1^{st} November, 2016

Acknowledgements

I shall begin my acknowledgements by apologising, a great many people have had some degree of influence on the time spent doing my PhD, the work in it or this document and I will surely miss someone out, so if it's you, sorry and thanks.

To the members of DSG, notably Niall O'Hara, Dirk Hasselbalch and Franck Franck who went for regular coffees, lunches and after work drinks (less regularly than the coffees and lunches), cheers. Mithil Mohan for your sense of 'humour', thanks for trying, it was nearly funny. You all kept the stress level down on the rare occasions it was elevated.

To Andrei Marinescu, it was great to have someone at the same stage of the PhD process working in the same area. From the very dynamic and non-stationary work discussions, to listening to me complain about memory leaks, thank you. An intermittently frustrating process was made easier and more fun.

Finally and most of all, I would like to thank Siobhán Clarke and Ivana Dusparic. I received excellent supervision and doubt I would have got anywhere near here without their knowledge, patience and support. You put in a lot of work, not least in having to collaborate with me and my spelling, I appreciate it greatly.

Adam Taylor
University of Dublin, Trinity College
November 2016

Publications Related to this Ph.D.

Publications directly related to Parallel Transfer Learning:

1. Transfer Learning in Multi-Agent Systems through Parallel Transfer. **Adam Taylor**, Ivana Dusparic, Edgar Galván-López, Siobhán Clarke and Vinny Cahill. In Workshop on Theoretically Grounded Transfer Learning at the 30th International Conference on Machine Learning (ICML), vol. 28, 2013.

2. Self-Organising Algorithms for Residential Demand Response. **Adam Taylor**, Ivana Dusparic, Colin Harris, Andrei Marinescu, Edgar Galván-López, Fatemeh Golpayegani, Siobhán Clarke and Vinny Cahill. In 2nd Annual IEEE Conference on Technologies for Sustainability (SusTech), 2014.

3. Accelerating Learning in Multi-Objective Systems through Transfer Learning. **Adam Taylor**, Ivana Dusparic, Edgar Galván-López, Siobhán Clarke and Vinny Cahill. In a Special Session on Learning and Optimization in Multi-Criteria Dynamic and Uncertain Environments at the International Joint Conference on Neural Networks (IJCNN at WCCI), 2014.

Publications in the broader Reinforcement Learning/Smart Grid area (published):

1. Management and Control of Energy Usage and Price using Participatory Sensing Data. **Adam Taylor**, Edgar Galván-López, Siobhán Clarke and Vinny Cahill. In The Third International Workshop on Agent Technologies for Energy Systems at Eleventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 111-119, 2012.

2. Design of an Automatic Demand-Side Management System Based on Evolutionary Algorithms. Edgar Galván-López, **Adam Taylor**, Siobhán Clarke and Vinny Cahill. In Proceeding of the 29th Annual ACM Symposium on Applied Computing (SAC), pp. 24-28. 2014.

3. P-MARL: Prediction-Based Multi-Agent Reinforcement Learning for Non-Stationary Environments. Andrei Marinescu, Ivana Dusparic, **Adam Taylor**, Vinny Cahill, Siobhán Clarke. In International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 1897-1898, 2015

4. Maximizing Renewable Energy Use with Decentralized Residential Demand Response. Ivana Dusparic, **Adam Taylor**, Andrei Marinescu, Vinny Cahill and Siobhán Clarke. In Proceedings of the 2015 International Smart Cities Conference (ISC2). 2015.

Abstract

Learning-based approaches to autonomic systems allow systems to adapt their behaviour to best suit their operating environment. One of the more widely used learning methods is Reinforcement Learning (RL). RL agents learn by repeatedly executing actions and observing their results, and over time a representation of how to behave well is developed. A significant issue with this approach is that it takes a long time to reach its best performance. Every action has to be experienced several times in each particular circumstance for its value to be representative of its converged value. The presence of multiple agents increases the number of experiences required.

Multi-Agent Systems (MAS) are systems in which multiple agents affect a shared environment. MAS are inherently large-scale as they are composed of many interacting agents. In MAS, the cumulative effects of agents' actions make the environment more variable. Greater variability in the outcomes of actions requires more learning as a single sample becomes less representative of the converged value. An RL system's performance is necessarily sub-optimal while it is learning. Each learning experience is expensive, taking time and affecting the system that is being controlled. Experiences should be used as efficiently as possible to reduce the time spent learning, improving overall performance. The less time needed to learn, the better the performance of a system over its lifetime.

Transfer Learning (TL) is a method of using additional knowledge to accelerate learning. It operates by taking knowledge from a source task (a process that supplies information) and reusing it in a target problem, with the aim of reducing the amount of learning to be done. Inter-Task Mappings (ITM) are used to allow more diverse tasks to share knowledge, effectively translating knowledge so that it is mutually intelligible. TL has shown promising results, but it requires a learnt source of information prior to the execution of the target system. This means it can not operate in real-time.

This thesis addresses TL's limitations by allowing the source of learnt information and the task to be accelerated to run concurrently. The main contribution, Parallel Transfer Learning (PTL), enables different agents to support each other's learning through mutual knowledge exchange. This is particularly beneficial in MAS, as agents are naturally concurrent and typically learn broadly similar things when in the same environment, so there is likely useful information to share. PTL can reuse useful knowledge in several different ways, each designed for particular types of environment. Detecting the type of environment, and if it is changing, allows PTL to self-configure for a particular system at a given time. PTL accomplishes this by modelling its performance over time and reacting to divergences in performance levels. PTL can transfer information between more diverse agents using ITM, which can be learnt in real-time by having the source and target share information.

PTL's evaluation is twofold: fundamental aspects are examined in simple environments, while overall effectiveness is evaluated in an example MAS, the Smart Grid. PTL is evaluated against standard RL, to quantify any improvement in learning time. The results show that transferred information can accelerate learning and it is of particular benefit when agents learn in different situations. When agents are in similar situations, comparable performance can be achieved in 11.11% of the time in one application and 40% in another. The learnt ITM allow improvement in homogeneous tasks and can find an effective mapping in the heterogeneous case. PTL can perform well in changing environments as long as the change stops and knowledge can by learnt.

Table of Contents

List of Algorithms

List of Figures

List of Tables

Glossary of Terms

DWL	Distributed W-Learning.
EV	Electric Vehicle.
ITM	Inter-Task Mapping.
MAD	Mean Average Deviation.
MAS	Multi-Agent System.
MDP	Markov Decision Process.
PTL	Parallel Transfer Learning.
RDR	Residential Demand Response.
RL	Reinforcement Learning.
SG	Smart Grid.
TL	Transfer Learning.
$\boldsymbol{\alpha}$	RL's learning rate parameter.
$\boldsymbol{\chi}$	An ITM in TL or PTL.

$\boldsymbol{\gamma}$ RL's discount factor.

$\boldsymbol{MM}$ The way received information is merged in PTL.

$\boldsymbol{\pi}$ A policy in RL.

$\boldsymbol{SM}$ The way pairs are selected in PTL.

$\boldsymbol{TS}$ The amount of state-action pairs per transfer in PTL.

Chapter 1

Introduction

Before I refuse to take your questions, I have an opening statement.

Ronald Reagan

This thesis presents Parallel Transfer Learning (PTL), an on-line version of Transfer Learning (TL) that allows Reinforcement Learning-based systems to learn more quickly. While learning, performance is necessarily poor, and by reducing this time overall performance can be improved.

Autonomic systems which are capable of learning, can adapt their behaviour to changing circumstance and improve their performance. Reinforcement Learning (RL) is a common method used in these systems [Kara et al., 2012; Kober and Peters, 2012; Lange et al., 2012; Peters et al., 2013]. RL takes a considerable amount of time to learn good performance due to the way it uses knowledge gained from its environment. Knowledge is learnt by repeatedly sampling the environment to develop an expectation of how it behaves. The main contribution of this thesis is a technique called PTL that shares knowledge in order to accelerate learning. It reduces the amount of time required to learn to perform well in Multi-Agent System environments. The rest of this chapter will introduce the categories of systems being addressed and learning in those systems as motivation, finally it will lay out the structure for the rest of this thesis.

1.1 Motivation

Large-scale, autonomic systems have many interacting components that lead to them being variable and complex. As a result learning-based control is often used in such systems [Liu et al., 2014; Sichman and Coelho, 2014; **?**]. The behaviour of these systems can change over time, making learning-based control attractive. Learning takes time to perform well in large-scale, autonomic systems, because their behaviour can fluctuate during the learning process. While learning is occurring performance is poor, so the time spent learning should be reduced.

1.1.1 Large-Scale Systems

The classification of a system's scale is very much dependent on its subject area. Very few definitions of a particular system scale attempt to apply number ranges to the amount of actors in a system, instead they attempt to categorise systems by some other aspect. For example, ultra-large-scale systems are described as being larger than a system of systems [Northrop et al., 2006]. Their categorisation also mentions the amount of code, data produced, number of uses etc. Categorisation is not limited to one aspect and is usually not exact. The classification of a particular system's scale is often subjective and affected by whichever aspect is the particular focus of a work.

To make the classification of systems more concrete, the following definition for large-scale systems will be used: a system can be considered large-scale, if it can be subdivided into more than one interconnected system [Jamshidi, 1996]. This definition captures the complexity of large-scale systems and introduces a requirement that they have multiple actors. While it is rather vague and encompasses many types of system, it includes the set of systems that can be addressed by autonomic control (i.e., systems that have multiple interacting components). For example, using this definition most distributed systems and all Multi-Agent System are considered large-scale (see Section 2.1 for further detail). It also has implications for control. If a system can be divided into sub-systems, then the control scheme can be as well.

When controlling large-scale systems, managing the interactions between entities can be complex. Relationships can change over time and in ways that can not be anticipated by designers. This causes a system's behaviour to fluctuate, in these situations it becomes important that systems are adaptive enough to adjust without designer intervention.

1.1.2 Autonomic Systems

Autonomic Systems are those which can manage themselves when provided with high level goals [Kephart and Chess, 2003]. Autonomous systems are those capable of operating on their own without user input. This necessitates self-management making them also autonomic, both terms will be used interchangeably to mean systems that do not require user input and can self-manage. Originally proposed to address the challenges caused by the software complexity crisis[1], they can equally apply to control of large-scale systems. Large-scale systems' control systems exhibit many of the same problems, so similar solutions should apply. There is no precise definition of what an Autonomic System is, but systems implementing more than one of the self-* properties are generally classed as autonomic [Huebscher and McCann, 2008].

Autonomous systems require that a system fulfil functions that would have been done for the system by designers and technicians. This is commonly called self-management or the self-* properties, which are as follows [Di Marzo Serugendo et al., 2005; Ganek and Corbi, 2003]:

- **Self-Configuration** is the system property that given a set of high level goals, the system sets and adjusts parameters to achieve the required functionality. This can occur at any level from allocating device addresses to integrating with other systems.
- **Self-Healing** is the property by which a system automatically detects, isolates and repairs faults. This can be at a hardware or software level.

[1]For many problems, software solutions are very complex and require experienced technicians to instal, maintain and customise. This incurs considerable costs and causes inertia for systems [Evans, 2004].

- **Self-Protection** is a system trait that anticipates or defends against significant failures or deliberate attacks.
- **Self-Optimisation** is the property of continual improvement. The system seeks to improve its performance over time, which can be done through reconfiguration or by better actuation.
- **Self-Organisation** is the process of adjusting the configuration of, or interaction between, internal components to meet goals. It can be used to achieve the other properties, or for other goals of the system.

This thesis is particularly focused on self-optimisation and self-organisation (although not exclusively as the properties are not independent). Self-optimisation necessitates learning-based approaches, as it improves performance over time (which is the very definition of learning) [Van den Berg et al., 2008]. Self-organisation requires the internal structure of a system be changeable, which is commonly achieved with Multi-Agent Systems [Liu et al., 2014; Sichman and Coelho, 2014; **?**]. Multi-agent learning can provide properties other than self-optimisation and self-organisation, but they are the most affected by the rate of learning.

1.1.3 Multi-Agent Systems

In large-scale autonomous systems, there are multiple entities in the system, so a single RL process can not provide all the control. This necessitates the use of a Multi-Agent System (MAS). In a MAS, there are multiple agents which interact, each solves a local problem and from these local solutions, a global solution emerges. These local solutions are often similar; there is repetition in what is learnt by the agents in a MAS [Olfati-Saber et al., 2007; Van der Hoek and Wooldridge, 2008].

The presence of multiple agents in a system increases the amount of learning that each agent needs to do over a single agent version of the same problem, as other agents affect the environment, which from any particular agent's point of view, makes the environment more variable [Stone and Veloso, 2000]. The greater the variability in an environment, the more samples of actions needed to learn a good expected value.

1.2 Learning

The basic question when considering learning is as follows: if a learning process can improve its own performance while it is executing, then is there any reason why designers can not achieve at least the same gains and provide them to a non-learning-based approach without the added complexity that learning brings and have the performance benefits throughout? The answer is, the abilities of designers are limited by when information is available to them, not everything can be known at design-time, this necessitates learning in at least the following cases [Russel and Norvig, 2010]:

- A designer can not possibly know all the situations a process may find itself in, let alone account for them. For example, consider an arbitrary maze solver. The process must learn to solve each new maze, as the solutions to all possible mazes can not be supplied by a designer [Hahn and Zoubir, 2015].
- Many environments change over time in ways that can not be anticipated. For example, a process designed to drive a car amongst self-driving vehicles would fail when sharing a road with human drivers [Coelingh and Solyom, 2012].
- There are problems which designers may be unable solve; either due to the problem's scale or complexity. For example, a traffic management system requires considerable configuration and adjustment for each new road network; the characteristics of a particular network greatly impacts on its configuration [Bazzan, 2009].

More complex systems—particularly large-scale systems—exhibit a combination of these situations, and therefore can greatly benefit from learning. In these systems, the interactions between individual entities and the environment can create a feedback loop in which the system and environment mutually impact on one another. This causes fluctuations in behaviour that can impact on both performance and learning. In these cases, a designer can not produce a good control solution due to the number of entities involved. Even if they could, static solutions can not continue to perform well as their setting is constantly changing, so learning is needed to accommodate this change. In these large-scale systems where learning is required, the amount of learning to do takes a significant time. While learning is happening performance is poor, so to perform better over the system's lifetime, learning needs to be accelerated. These complex and large-scale environments are what PTL is designed for.

1.2.1 Reinforcement Learning

To achieve the self-optimisation necessary for autonomous systems, learning is needed. There are several categories of approach to machine learning (further discussed in Section 2.3), each with its own strengths and weaknesses. This thesis will focus on Reinforcement Learning-based systems.

In behavioural psychology, the concept of Reinforcement—on which Reinforcement Learning is based—has the following definition:

> "Reinforcement is a consequence that will strengthen an organism's future behaviour whenever that behaviour is preceded by a specific antecedent stimulus" [Skinner, 1953].

This concept is familiar in human and animal learning; good behaviour is rewarded, bad behaviour is punished and over time correct behaviour is learnt. When applied in Computer Science, this is called Reinforcement Learning (RL) [Barto, 1998]. RL—regardless of the particular algorithm used—follows a basic pattern: observe the environment, select and execute an action, receive feedback. After multiple iterations following this pattern, an understanding of what should be done in each particular circumstance is developed, in short, the process has learnt.

RL usually works by learning a value function, which is a representation of what to do in all states. A state is a single combination of the parameters that represent the environment. After sampling the reward several times in a state by taking actions, the value function will represent the expected value for each action in that state. Using this information the agent—an RL process—can choose the best action for a particular situation and thereby behave optimally. During its learning, an agent affects the system it is operating in, which in turn affects the system's behaviour.

While the RL process is exploring (trying different actions to learn what is good), it is affecting the environment. To know that a particular action is bad, it must execute it several times. This causes the system to perform poorly. The amount of exploration needed is determined by how complex the system is (discussed further in Section 2.3.2). More complex environments will have more variable outcomes from actions. The more variable experiences in a state can be, the longer it takes to learn in that state, as a single experience becomes less representative of the expected value. The amount of time spent learning is linked to the performance of a system, reducing the time spent learning is called accelerating learning [Taylor et al., 2014]. This is where accelerating learning's main benefit is, the less time spent performing poorly, the better overall performance is.

In RL, the notion of best performance is based on the process's objective. It is when performance can not be improved on. This is made more complex by multiple objectives (see Section 2.2), but basically it is when performance in one objective can not be improved without disimproving another. Knowing when best performance has been achieved is non-trivial and is based on what RL has learnt (see Section 2.4). Performance can only be measured relative to the reward function. It is possible that a poorly chosen reward function would not correctly encode the designer's actual goals. In this case, fully converged learning could appear poor according to a metric based on the designer's goals, as RL will learn what the reward function encodes rather than the actual goal.

1.2.2 Accelerating Learning

Regardless of the reasons to learn, accelerating learning can provide better performance in less time than learning otherwise would have. Final performance can be improved, less computing power or input data are required for equivalent performance, learning solutions become practical for a wider range of problems. In short, getting to an answer sooner is better than later.

Learning requires some form of training to perform well. This training can be in the form of prior experience or input data. Whether this training data is provided by the learning process itself or by a designer, performance is contingent on it. Generally, a process given less training will perform worse than the same process with more training data (until it reaches the maximum performance) [Batista et al., 2004; Weiss and Provost, 2003]. Accelerating learning allows the training data to be used more efficiently in some sense. More can be learnt from less data.

1.2.3 Transfer Learning

There have been several attempts to accelerate RL (see Section 2.5). One such method is TL [Taylor and Stone, 2009]. TL is based on an idea borrowed from psychology. When learning how to accomplish a task, knowledge from a related task is often used as a starting point for the new task. In terms of RL, this involves gathering the knowledge learnt in one problem and supplying it to the problem to be accelerated. TL operates between a source task (the process that provides information) and a target task (the knowledge recipient). The idea is that in the target task it is easier to learn the correct value function, if learning starts from some mid-point, than at the very beginning. This mid-point is provided by the knowledge learnt in the source task which has been transferred. *Generally, the closer related two tasks are, the more likely it is that they can accelerate learning by transferring knowledge* [Taylor, 2008]. TL is an off-line process. A source task completes its learning, the information is gathered and provided to the target task which then learns. Two processes that wish to transfer information must have a mutually intelligible representation of the information transferred. This is achieved by mapping the information from one representation to the other. These mappings are non-trivial and often produced by designers or require considerable off-line calculation [Taylor and Stone, 2009].

1.3 Problem Statement and Hypothesis

For an RL process to learn in simple environments, it must sample the environment by executing actions and observing their results. After several samples, an expected value is learnt for each of its actions. In more complicated problems, the amount of samples required to learn increases. This is particularly true in MASs, as the actions of agents affect one another. Agents mutually affecting one another causes the system's behaviour to fluctuate more. This makes the outcomes of actions appear more variable to an agent in the systems.

While agents are exploring and trying actions, a system's performance is necessarily sub-optimal, as to know that an action is bad it must be experienced several times, which worsens performance. Reducing the amount of time spent exploring improves overall performance, as less time is spent performing poorly.

TL can accelerate learning and improve performance, but it can only operate off-line. It supplies information to target tasks before the start of their execution begins. *This prevents the closely related tasks in a MAS from being exploited.* In addition, as TL is off-line it can not address an *unanticipated fluctuations* in behaviour introduced by either the learning or inherent in the system. These fluctuations can invalidate what has been learnt or any knowledge supplied off-line. It is impractical to run parts of a MAS before others just to accelerate learning. Equally, running a MAS just to accelerate learning in that MAS is paradoxical. This leaves on-line knowledge transfer as the only way to exploit relatedness of tasks in a MAS and thereby, accelerate learning. This raises the following question: can the relatedness of tasks in a MAS be leveraged to accelerate learning by moving TL on-line?

Exploiting the relatedness of tasks in a MAS can allow TL to accelerate learning and thereby improve the performance of RL in such systems. The relatedness of agents' knowledge in a MAS can only be used after the agents have learnt something, this requires an on-line scheme to accelerate learning. On-line acceleration of RL will also allow a system to adapt to change in the environment at run-time more quickly as learning can be accelerated at other times rather than just initially as with TL. In short, by allowing TL to operate on-line, the amount of samples of actions an agent needs to learn in a state can be reduced by transferring knowledge that has been learnt by other agents.

The hypothesis of this thesis is that by allowing TL to operate on-line, knowledge can be shared between the related processes in a MAS and their learning can be accelerated. Transferring knowledge on-line allows the dynamic nature of MASs to be captured in the knowledge shared, which is not possible in off-line TL.

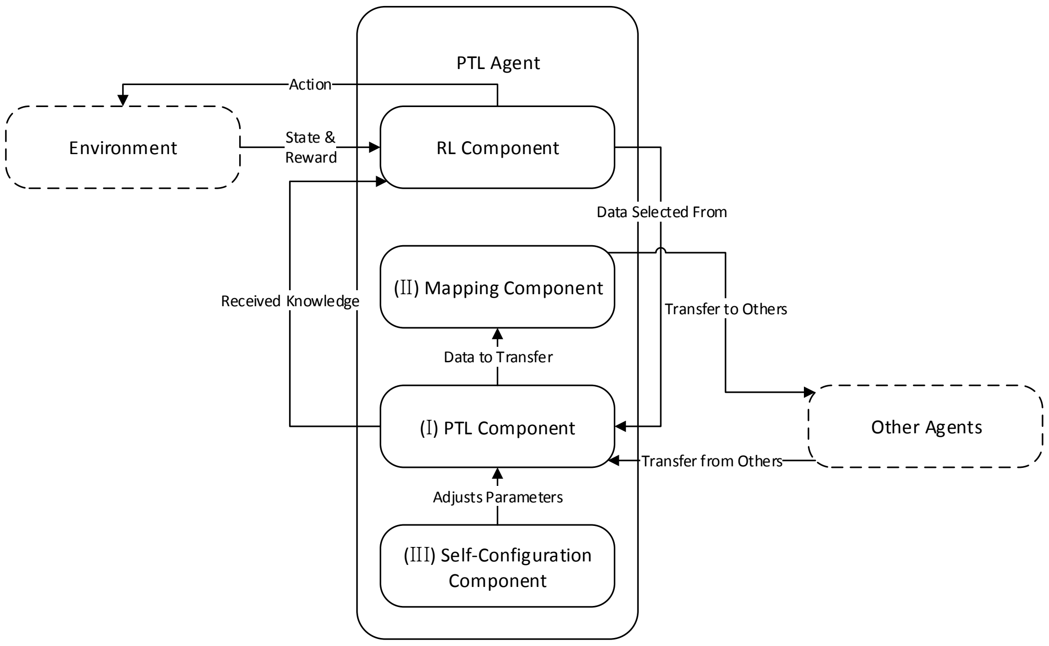


Figure 1.1 Agent/Environment Interaction Loop in Parallel Transfer Learning.

1.4 Contribution

The contributions of this thesis address the rate of learning in large-scale MASs that use RL. By accelerating learning the benefits of these systems can be arrived at earlier and even improved on. Figure 1.1 shows the functional architecture of a PTL agent. The contributions are detailed below with indexes corresponding to their major architectural component.

(I) **Parallel Transfer Learning** As TL is necessarily an off-line process it can not leverage the relatedness of tasks in a MAS, nor can it adapt to any fluctuations in behaviour due to learning or inherent in a system while that system is running. This is addressed by PTL, which allows knowledge to be shared between processes on-line, thereby allowing agents to support each other's learning and react to changes in the system. It is addressed by the PTL Component in the system architecture in Figure 1.1.

(II) **On-line Learnt Mappings** To transfer information from one process to another in TL or PTL, information must be mutually intelligible. In mapping work to date, Inter-Task Mappings (ITMs)—which allow translation—have been calculated off-line. Producing an ITM off-line is impractical for on-line learning as it prevents relatedness of tasks in a MAS being exploited. PTL is capable of learning ITMs on-line, which allows it to adapt to fluctuation in the environment as well as exploit relatedness of tasks. These are produced by the Mapping Component in Figure 1.1.

(III) **Self-Configuration** PTL's various methods of transfer are effective in different types of environment. If the environment changes on-line, then this change can be detected and reacted to by PTL. It also reduces the amount of design effort required as PTL is capable of choosing the best methods for a particular environment. To do this, the environment must be modelled. PTL categorisation uses a well-known approach CUSUM [Brook and Evans, 1972] to model performance. If the performance deviates from the expectation, then PTL can react. The Self-Configuration Component in Figure 1.1 is responsible for this.

Aspects of PTL are evaluated in small-scale applications to demonstrate effectiveness, but the main evaluation is done in Smart Grid applications. The Smart Grid is appropriate as it can provide dynamic, non-stationary environments for large-scale systems. In it, performance can be improved and made more reliable.

1.5 Evaluation

The evaluation of PTL is twofold: lower level concerns (e.g., parameter selection, reaction to change etc.) are addressed in more simple applications, before a larger scale test is done. This is done using three applications and a range of environments. The well-known Cart Pole [Berenji et al., 2013; Bonfè et al., 2011; Selfridge et al., 1985; Sutton and Barto, 1998], Mountain Car [Ammar et al., 2014; Knox et al., 2011; Moore, 1990; Sutton et al., 2012] and an electrical grid simulator called GridLAB-D [Chassin et al., 2008][2]. The former two will be used to evaluate PTL without the complexity introduced by multiple objectives. In these more simple applications, performance measurement is cleaner and as agents are not affecting the same environment it is less variable which clarifies the effects of PTL. The other application, GridLAB-D, has scenarios with multiple objectives and the smart grid environment is particularly variable, making it a much fuller test. It will be used for larger scale experiment. This will allow the performance of the algorithm to be evaluated in a real-world problem, together with scalability, flexibility and reactiveness. These are important characteristics for an algorithm that will operate in real-world systems, as the underlying environment can change on-line and the algorithm must adapt.

1.6 Assumptions

This work makes a number of assumptions which—where necessary—will be further justified in following chapters. Generally, they are commonly taken assumptions in MASs and are used to define the scope of the problem-space, rather than reduce its complexity. Assumptions about particular experimental scenarios will be discussed in Chapter 4, the following are those that impact on the design and implementation of PTL:

- All agents in the system use Reinforcement Learning (discussed in Chapter 2).

[2]GridLAB-D is a registered trademark (see `gridlabd.org`)

- Agents are capable of communicating with each other.
- Communication is not necessarily reliable, but it does not corrupt messages perversely. This means that messages can not have their content changed and still be intelligible.
- Agents have a set of neighbours with which they can communicate. They do not need to discover their neighbours. While in the experiments, sets are fixed they are not necessarily so. Introducing new neighbours during learning could improve PTL by providing more diverse knowledge.
- In the case of mobile agents, the discovery of agents is provided.
- There are no malicious agents. Any information shared is well-intentioned whether, it is correct or not. If agents have contradictory goals, they progress towards them without intentionally interfering with others. The natural collision of objectives is allowed.
- The environment is not truly random or unbounded. There is something that can be learnt.
- The domains focussed on are discrete.This means that both time and the state-spaces are treated as quantised ranges.

1.7 Roadmap

The remainder of this thesis is organised as follows:

- Chapter 2 provides background information and the state-of-the-art in accelerating RL with a specific focus on TL.
- Chapter 3 Section 3.3 introduces the main contribution of this thesis, Parallel Transfer Learning. It describes an algorithm that can accelerate reinforcement learning by sharing knowledge between agents on-line.

- Chapter 3 Section 3.4 discusses how knowledge can be made mutually intelligible between agents who do not share a common representation. This is done by learning a mapping on-line.
- Chapter 3 Section 3.5 discusses how PTL can detect the environment and react to it. The way PTL is configured for a particular environment affects its performance. In order to self-configure and achieve its best performance, PTL needs to be able to categorise the environment and adjust accordingly.
- Chapter 4 details the lower level details of PTL's implementation and design.
- Chapter 5 describes the applications, environments, experimental set-up and results obtained.
- Chapter 6 finishes the thesis with conclusions and possible direction for future work.

Chapter 2

Related Work

People who think they know everything are a great annoyance to those of us who do.

Isaac Asimov

This chapter provides an introduction to learning in Multi-Agent Systems (MASs) in Section 2.1 and Multi-Objective Systems in Section 2.2. Section 2.3 introduces learning in general with a particular focus on Reinforcement Learning (RL). It describes the state-of-the-art in accelerating RL in Section 2.5. The chapter closes with some discussion of the state-of-the-art before outlining requirements for this thesis's contribution, Parallel Transfer Learning.

2.1 Multi-Agent Systems

In large-scale autonomous systems, entities in the system will need to reconfigure themselves to achieve the self-* properties. The reconfiguration of one entity should not affect other entities unduly. This requires that entities are at least somewhat independent. However, achieving overall control of a system will necessitate interactions between these entities, which induces some interdependence [Pipattanasomporn et al., 2009]. These requirements for agent-dependence mean that the entities must be self-contained, capable of acting independently and able to communicate. The entities must solve their individual control problems as well as the system's overall control problem. This naturally maps to a MAS.

Historically there are broadly two approaches to Distributed Artificial Intelligence: Distributed Problem Solving and MASs [Weiss, 1999]. Distributed Problem Solving focused on task decomposition and solution aggregation and MASs were about process interaction and knowledge sharing. Now the term MAS has come to subsume both.

Using this formulation for Distributed Artificial Intelligence, the entities in a system are called agents. There are many definitions of what an agent is, but most are similar to the following:

> "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators" [Russel and Norvig, 2010].

This definition fits well in autonomous systems, as an agent can be a self-contained entity as is required. It logically follows from this definition that a MAS is a system with more than one agent in it, which the name implies. In practice, a more concrete definition is used which is as follows:

> "[a MAS is a] loosely coupled network of agents that interact to solve problems that are beyond the individual capabilities or knowledge of each agent" [Sycara, 1998].

This definition is the most common used to describe MASs. It introduces a constraint on what a MAS is that was not present in the name-implied definition, that being that the problem is beyond the capabilities of a single agent. This constraint is naturally satisfied in large-scale systems, so this definition will be used throughout this thesis.

The underlying principle of MASs is that a globally good solution will arise out of several sub-solutions. MASs have the following characteristics which are inherent due to their distributed nature:

- There is no global view for any single agent.

- There is no overarching global control system.
- Data is distributed.
- Agents operate asynchronously.

These characteristics mean that the interactions between agents must bring about some form of coordination, so that the agents' local sub-solutions effectively mesh to a good whole solution. The coordination can be achieved through explicit or implicit communication [Stone and Veloso, 2000]. Explicit communication is message passing, implicit communication involves the agents achieving some sort of consensus through observing either the behaviour of other agents or the results of that behaviour. Regardless of how communication happens, the agents are either competitive or collaborative [Panait and Luke, 2005]. When agents are competitive, the hope is that through competition the agents will strive to perform better on their own sub-problem and in aggregation overall performance will improve. When agents are collaborating, the assumption is that overall performance will improve, if sometimes agents allow their performance on a sub-problem to drop to benefit another agent.

The agents composing a MAS can be heterogeneous or homogeneous. Both can occur at the algorithm level or within an algorithm. For example, consider a MAS with two agents, one of which implements Algorithm A, the other Algorithm B. These agents are heterogeneous as Algorithms A & B are different. In this thesis, this type of heterogeneity we will call algorithmic heterogeneity (this phrase will not be shortened). If both agents instead implemented Algorithm A but had different internal representations of their environment, this would be representational heterogeneity (this may be shortened to heterogeneity). Heterogeneity can also occur if the agents themselves differ in capabilities e.g., different actions or sensors. This is effectively representational heterogeneity, as different capabilities will affect how like data is represented internally. The terminology is similar for homogeneity.

MASs have been shown to be effective in a variety of large-scale control problems. In urban traffic control, agents have been used to represent junctions and actuate traffic lights [Dusparic and Cahill, 2012]. It has also been applied to air traffic control [Tumer and Agogino, 2007] and the Smart Grid [Pipattanasomporn et al., 2009]. In the Smart Grid, it has been applied in a range of application from very small-scale residential applications [Dusparic et al., 2013] to whole grid systems [Hernandez et al., 2013].

2.2 Multi-Objective Systems

Large-scale systems and particularly MASs, tend to include multiple different stakeholders, who may have different goals for what a system is to achieve. This can lead to contradictory goals for individual agents in the systems. When faced with two or more objectives, some form of arbitration must occur. In single-objective systems, algorithms search for an optimal solution, a point at which it can do no better. In multi-objective systems, the concept of an optimal solution must be changed, as only in rare cases can a solution be optimal to all objectives simultaneously. There will usually be the option to improve more according to one objective, this however, may reduce performance according to other objectives. The concept of Pareto-Optimality reflects this [Pareto, 1906]. It can be stated formally as the following:

$$x \in X^* \iff performance(x + \delta) \leq performance(x) \; \forall \; o \in O \tag{2.1}$$

Where x is the test solution, X^* is the set of Pareto optimal solutions, O is the set of objectives and δ is a small change in the test solution [Marler and Arora, 2004]. This means a solution is Pareto optimal if it can not be improved for one of its objectives without reducing performance on at least one other [Deb, 2014]. In other words, it has reached a point where a trade-off between objectives is necessary. From this, the concept of a Pareto front follows. A Pareto front is the set of solutions which are Pareto optimal. Some points in a Pareto front will be preferable to others due to the relative priorities of objectives or designers' preferences. Effectively the Pareto front is a set of possible trade-offs that could be used that has been calculated based on supplied information, more information is needed to select a particular solution.

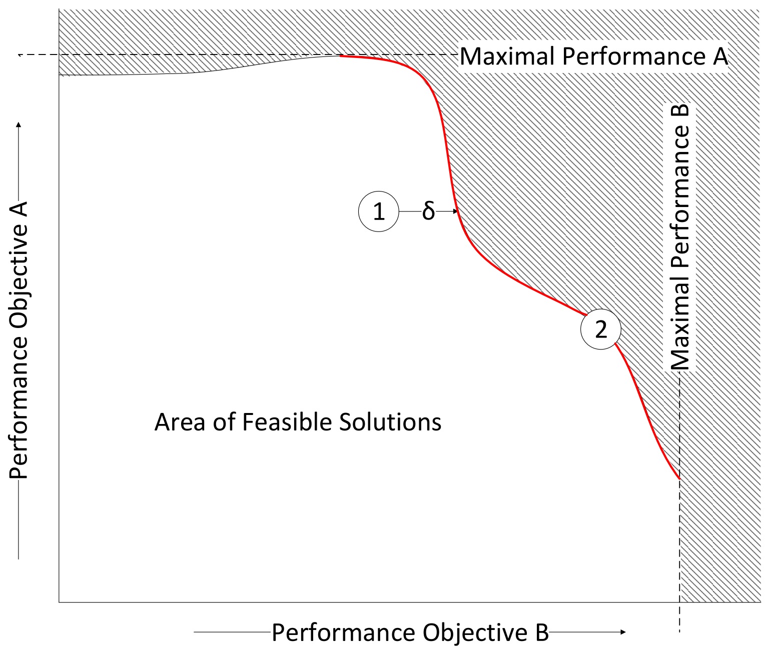


Figure 2.1 An Example Pareto Front.

Figure 2.1 shows an example Pareto front for a process with two objectives A and B. The hatched region represents infeasible solutions[1]. The curved line represents the boundary of the feasible region and the red line is the set of solutions in the front. The dashed lines show the particular solutions that have the best performance according to one objective, note that only one solution in the front is maximal in A and one in B, maximal B is where the red line meets the dashed line. The points directly below this are not Pareto optimal, as A can be improved without impacting B. The particular solution 1 is not optimal, as there a step δ by which performance in A remains constant, but it improves in B. The solution 2 has no such step, it is Pareto optimal. A step would have one of the four following non-Pareto optimal outcomes: (I) move it into the infeasible area, which is impossible; (II) feasibly increases its performance in A, which would move it to the left, reducing B; (III) feasibly increases its performance in B, which moves it

[1]Here infeasible means not existing rather than having been excluded based on some criterion.

down the vertical axis and reduces A's performance; or (IV) reduces performance in both A and B.

There are four commonly used ways to make this final selection from the Pareto front [Ruiz et al., 2014]. The first three types are to allow a human decision maker to select from the solutions based on other factors. They vary only in when the human interacts with the optimization process: A Priori (before optimisation), A Posteriori (after optimisation) and Interactive (during optimisation). The final type are No Preference Methods, in these no extra information is supplied and an effectively random Pareto optimal solution is chosen.

There are two main categories of approach to Multi-Objective Systems. The first is the naïve approach, which is to combine the multiple objectives into one single objective. Through doing this, the complexity of multiple objectives and their priorities can be removed. These approaches are commonly called Combined State-Space. It leaves no inter-objective arbitration to manage at run-time, however it means that the priorities can not be easily changed, as they are encoded by the objective-combining process [Deb, 2014]. This approach was attractive early in the development of Multi-Objective Systems as it allowed much of the work on Single-Objective Systems to be reused without alteration, but as more complex problems have been addressed it has been found wanting [Ehrgott et al., 2014; Karmellos et al., 2015; Srivastav and Agrawal, 2015].

Scalarisation is a more sophisticated version of the same approach [Yahyaa et al., 2014]. In it, more complex functions are used to merge the different objectives. By selecting a particular Scalarisation (i.e., a particular prioritisation of objectives), a Pareto optimal solution (or set thereof) can be found. If more than one solution is found, then some other process must select which to apply. The specific selection criteria for which solution to use from a Pareto front is sometimes encoded in the Scalarisation, so that the extra effort of calculating unused solutions is removed. If not, then a specific solution must be selected. Scalarisations that require human decision makers or the calculation of multiple control solutions are not practical. The requirement for human interaction makes these approaches cumbersome for large-scale systems with non-terminating control. They also effectively calculate many solutions to the control problem with can take an impractical amount of time given the scale and complexity of such systems.

The other approach is to give each objective its own representation. Each objective then makes a contribution to the decision or chooses not to. The contribution's form depends on the algorithm, but it is commonly a suggestion of what to do at a particular time [Banerjee and Sen, 2007; Bianchi and Bazzan, 2012]. This is particularly common in on-line learning. These approaches are called Arbitration-Based. The problem of optimal control in on-line Multi-Objective Systems is hard [Mannor et al., 2014; Mannor and Tsitsiklis, 2009]. It has been studied in games for some time and is known as Regret Minimisation [Hazan, 2012].

2.3 Machine Learning

2.3.1 Introduction

Previously, Chapter 1 discussed the need for learning in large-scale autonomic systems and particularly in MASs. Machine Learning is a category of techniques that use prior experience or examples to improve performance on a task [Alpaydin, 2004]. It can be broken down into four types based on how much feedback needs to be provided and what provides the feedback. The classes are quite broad and there can be overlap:

- **Supervised Learning** is learning with a teacher. The learning process is given a set of input data with correctly labelled output data, from this it deduces rules that can be applied to other unseen cases.

- **Unsupervised Learning** techniques tend to be statistical in nature, a process looks at data and based only on the data derives rules to be applied to unseen data. No correct labelling is provided. It groups similar data items based on features that it finds. From these features it can then classify new inputs.

- **Semi-Supervised Learning** falls somewhere between the two former types. A process is given a labelled input, but the labelling is incomplete and/or partially incorrect. This means the process must try to establish rules like Supervised Learning while validating the labelling similarly to how unsupervised approaches do.

- **Reinforcement Learning** is in some ways similar to Semi-Supervised Learning. A process has a function representing its goals which provides feedback based on performance. As performance can not necessarily be mapped to specific inputs or outputs, some extrapolation must be done.

Supervised Learning is generally applied to classification tasks, as it requires a teacher to label data, it can not be reasonably applied to on-line problems like control without significant off-line labelling of data [Barto et al., 1983]. When applied to control problems, it is usually coupled with other learning methods (e.g., Adaptive Dynamic Programming [Zhao et al., 2014] or Unsupervised Clustering [Wang et al., 2014]).

Unsupervised Learning has been applied to on-line systems [Banerjee and Basu, 2007; Furao et al., 2007; Kattan et al., 2015]. When used, it is normally in an incremental form or as a series of batched runs. Such methods can be effective, but they require large amounts of data to generate classes or predictions. This limits their usefulness as many large-scale systems exhibit some degree of variability[2] in data. This variability can introduce a heterogeneity into the data. This heterogeneity can be in accuracy, timeliness, type, source or accessibility. As the data can change, any training data will need to accommodate this, so that the classification is not skewed. Unsupervised learning in the face of unreliable data is a difficult problem. Decentralising an unsupervised process is complex as either the data must be gathered to be processed or the learning process must be distributed. The first approach introduces at least partial centralisation while the latter is limited by the interrelatedness of the data.

[2]Here variability means a change in the underlying process that generates the data, rather than a change between individual data items.

Semi-supervised learning suffers from combinations of the above problems depending on whether it is closer to Unsupervised or Supervised. RL too suffers from problems; for example, convergence to optimality is only guaranteed in some types of application and exploration can be inefficient. These are more manageable than those of other learning methods, however. Exploration time can potentially be addressed and optimality guarantees are lost in many large-scale systems due to the amount of potential solutions introduced by their scale.

2.3.2 Environments

How complex learning in a particular problem is depends on how its setting changes and how much of the environment that the learning process operates in can be known at any one time. The way that an environment changes affects learning processes. Change in the environment can be slow and gradual or sharp. Changes can follow a repeating pattern or not reoccur. The complexity of what must be learnt also depends on knowledge about the task and how the task is specified. The following list describes how various conditions affect environments [Anderson et al., 1986; Russel and Norvig, 2010]:

- **Time** can be treated as discrete or continuous. This changes the underlying representation of data and how performance is evaluated. Continuous formulations of time are nearly always more complex.
- **Task Repetition**, if the learning process is solving the same problem repeatedly, it is generally easier than a single non-terminating task.
- **Knowability** is how much of the problem can be understood at design-time. The more knowable a problem, the better the formulation that can be provide to the learner.
- **Observability** is how accurately the environment's current state can be known and how much of the environment can be seen. If an environment is fully observable, then the agent has complete and accurate knowledge. Partial observability

is when an agent has either inaccurate or incomplete information or both. Environments can also be unobservable.

- **Determinism** is how predictable the changes of an environment are. A deterministic environment is one in which the outcome of action is known and are always the same; in a stochastic one, action outcomes are defined by probabilities; non-deterministic environments are ones where action outcomes are not known or are unbounded.
- **Dynamicity** is a measure of how much an environment can change from sources other than the effects of the learning process over the time span of one action. For example, an agent trying to move forward in a static environment would expect to move forward, while in a dynamic environment the floor could move so that the agent maintains its position in spite of its action or it could be blocked by someone else in the environment.
- **Stationary** is a longer term version of dynamicity. If the results of an action or the performance they bring about can change from the previously correct historic expectation, then the environment is non-stationary.
- **Other Agents** can influence the environment and results of actions through their own actions. When multiple agents are in an environment, they can be cooperative, competitive, agnostic, unaware or any combination thereof.

These are not mutually exclusive criteria and ascribing any particular feature of an environment to a category is difficult. For example, in an environment with multiple agents, the actions of other agents could make a stationary, deterministic environment appear non-deterministic. Non-stationarity could cause a probabilistic, fully observable environment appear partially observable, if a shift in transition probabilities introduces a new possible outcome. As there can be uncertainty in this categorisation, the term simple environment will be used to describe environments which are fully observable, deterministic (possibly probabilistic) and stationary. When ‘simple environment’ is used in a MASs context, the only change in definition from the single agent case is the uncertainty due to other agents. A complex environment is one which is not simple. The use of these terms is only for clarity of expression, and where applicable, the terms from the above list will be used.

2.4 Reinforcement Learning

In behavioural psychology, the concept of Reinforcement on which Reinforcement Learning is based has the following definition:

> “Reinforcement is a consequence that will strengthen an organism’s future behaviour whenever that behaviour is preceded by a specific antecedent stimulus [Skinner, 1953].”

This concept is familiar in human and animal learning, good behaviour is rewarded, bad behaviour is punished and over time correct behaviour is learnt. When applied in Computer Science this is called Reinforcement Learning (RL) [Barto, 1998]. RL—regardless of the particular algorithm used—follows a basic pattern: observe the environment, select and execute an action, receive feedback (see Figure 2.2). After multiple iterations following this pattern, an understanding of what to do in each particular circumstance is developed, in short, the process has learnt.

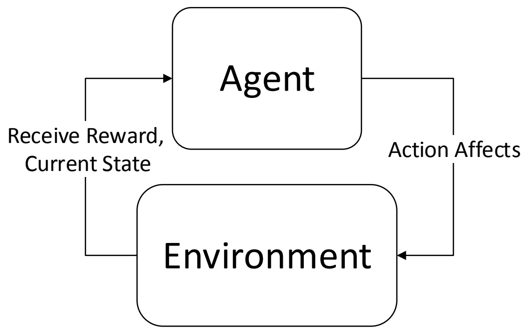


Figure 2.2 Agent/Environment Interaction Loop in Reinforcement Learning.

RL can be broadly split into on-policy and off-policy learning. A policy (π) is a set of rules for how act in an environment. A policy encodes the actions that satisfy an objective. In on-policy RL, an agent follows a policy until the application terminates—if it does—reinforcement is then received and if necessary the policy is changed. Off-policy RL is the process by which the agent learns a value for every action, not just specific sequences of them. In off-policy RL there is more learning to do, but it is much more flexible, as the agent can behave well if it finds itself in a situation not included in its current policy.

Agents typically represent the environment using a Markov Decision Process (MDP). An MDP is a set of states, $S = s_1, \ldots, s_n$, each with a set of actions $A_{s_i} = a_{s_i 1}, \ldots, a_{s_i m}$. A state is a single combination of the parameters that describe the environment. An action is anything an agent can do to affect either itself or the environment. The state-space is the set of all state-action pairs. Each action has a set of probabilities associated with it. These describe which states an action can transition to and how likely the transitions are. The transitions are rewarded by a function $R(s_i, s'_i, a_{s_i j})$. This function gives the immediate reward for the transition for taking the action $a_{s_i j}$ in the state s_i which caused the transition to state s'_i. MDPs are usually a discrete time formulation with transitions happening when the time changes. Ultimately, whether on-policy or off-policy, the agent's objective is to learn which sequences of actions lead to maximal reward. Assuming the reward function correctly encodes the system's goal, maximising reward is equivalent to finding the best way to achieve its goal. This is known as the optimal policy (π^*).

The example in Figure 2.3 shows an MDP with four states. Action 1 in State C does not cause a transition which is permissible in RL. In the example, if the agent starts in State A and takes Action 2, it would transition to State D, then Action 2 again to State C giving it the path <State A, State D, State C>. The reward accumulated by the path <State A, State B, State D, State C>, would be equivalent, but it would take one time-step longer to reach the high reward State C. This is an important feature of optimal policies, they are the most efficient way to satisfy objectives.

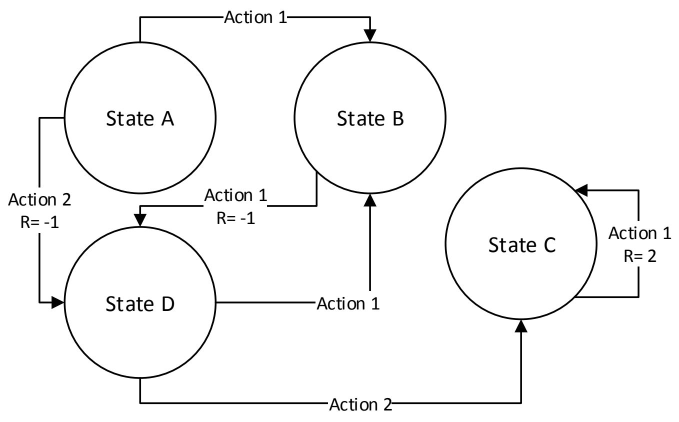


Figure 2.3 An Example Markov Decision Process.

To model a problem with an MDP, the following criteria must be satisfied [Bryson, 1975; Howard, 1960]:

- The **Markov Property** states that all state transitions are only affected by the environment, current state and current action. No prior states or actions influence transitions or have lingering effects.
- **Full Observability** is needed as an agent must know accurately what state it is in. MDPs have been extended to allow for continuous time and partial observability.
- **Stationarity** is required for the algorithms that use MDPs. They learn some representation of expected reward from a transition. For this expectation to be valid, both the transition probabilities and reward must remain constant[3].

When MDPs are being used in real-world systems, these criteria will not be as strictly met as in theory. For example, full observability can not be achieved if there is even the possibility of sensor error. In any MAS, the stationarity constraint is broken by the uncertainty inherent in the actions of others. In spite of this, MDPs are used in

[3]Here constant means the same as the algorithm has experienced while exploring. If previously a value was drawn from some distribution, then the distribution will not change.

real-world systems with the assumption that the errors introduced will be negligible or will average out [Kim et al., 2014; Peters et al., 2013; Wen et al., 2014].

While the transition probabilities can be encoded in the MDP (either explicitly or implicitly from the environment that the MDP is describing), some RL algorithms use an additional model to describe them. RL algorithms can be either model-based or model-free. A model is a description of how the environment changes in response to actions. Either describing or learning a model can involve considerable work, particularly in large-scale systems. In MASs, the model also needs to be able to account for the actions of other agents or it suffers from reduced accuracy. As others' actions can be entirely unpredictable, modelling will be, at best, inaccurate, so for this thesis model-free learning will be the focus.

The goal of RL is to learn a value function. A value function is a history of how good state-action pairs[4] have been, assuming that a given policy is followed. Each state-action pair has a (possibly inaccurate) value associated with it. Policy search approaches learn the value function by following a policy for some time until its value has converged[5]. It then changes the policy slightly and continues. This is the exploration phase. When an agent's performance is satisfactory, it moves to the exploitation phase, in which it uses the best policy learnt. This is on-policy learning. In off-policy learning, the agent does not follow a particular policy, but instead seeks out unexplored areas of the state-space and reward is assigned when it is received. It is usually coupled with temporal difference learning. Temporal difference learning assumes that the value function in a successor state is correct and uses it to update a state's value immediately, rather than waiting for reward to be received to assign credit. It can slow the process of learning considerably, because there is no guarantee that the successor state's value is correct. This can lead to transient inconsistency, but it will correct over time. Q-Learning [Watkins and Dayan,

[4]RL algorithms do not all assign reward to the same elements. Reward can be assigned to policies, states or state-action pairs. For clarity of expression, references to values of any of those will be equivalent. Regardless of how the value function is quantised, it learns the same way.

[5]This is when learning has finished, the value function contains the 'true' expected values for each state. The terms converged and finished learning will be used interchangeably. Convergence is not always possible.

1992] is a popular off-policy, temporal difference algorithm.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma(\max_A Q(S_{t+1}, A_{t+1})) - Q(S_t, A_t)) \tag{2.2}$$

Equation 2.2 is Q-Learning's update rule. In it, $Q(S_t, A_t)$ is the value function at the current state called the Q-Value. R_{t+1} is the reward received by transitioning from S_t to S_{t+1}. α is the learning rate, it affects how much new experiences change the Q-Value, γ is the discount factor, it influences how far the agent looks ahead. Higher values cause the agent to favour long term rewards more. Both α and γ are in the range $[0, 1]$. Q-Learning has been shown to reach the optimal policy in simple environments with a single agent [Kaelbling et al., 1996].

There are two common methods of selection actions. ϵ-Greedy, which chooses the action with the highest value with a probability $1 - \epsilon$, otherwise a random action is chosen. The other is Boltzmann action selection (or Softmax) which effectively makes each action as likely as its share of the sum of values for all actions.

$$P_t(A = a) = \frac{e^{Q(S_{t+1},a)/\tau}}{\sum_{i=1}^{m} e^{Q(S_{t+1},A_i)/\tau}} \tag{2.3}$$

Equation 2.3 is the Boltzmann action selection formula, $P_t(A = a)$ is the probability of action a being selected where τ is the temperature. Higher temperatures cause more exploration. Due to the limits on bitwidth of number representation, the use of exponential risks all actions getting the same probabilities due to overflow. Normally reward is limited or some approximating function is used. Regardless, the Boltzmann function has the property that even at low temperatures, some exploration will still occur. This is particularly attractive in non-stationary environments or those where learning can not be completed before exploration must begin (time sensitive applications for example). Both Boltzmann and ϵ-Greedy explore with some degree of exploitation as well. They are not purely random, the search is directed. This is important as in Q-Learning, for the value function to converge, successor states' values must be representative of the value of the optimal path through the state-space. If an agent always chooses a truly random action, the value function may never converge.

Reward in terminating tasks is generally received at the end of the task, in continuous problems it is given more frequently. The following terminology is used in Psychology and it clarifies the reward received well [Skinner, 1953]:

- **Positive Reinforcement** is the addition of a positive stimulus e.g., a reward function gives a positive number.
- **Negative Reinforcement** is a negative stimulus is removed or avoided e.g., an agent is prevented from entering a ‘bad’ state.
- **Positive Punishment** is a negative stimulus is applied e.g., a reward function giving a negative number.
- **Negative Punishment** is a positive stimulus is removed e.g., an agent is blocked from a ‘good’ state.

Due to the nature of RL, positive or negative reward is applied and can not be removed, so the negative terms are not applicable, therefore positive reinforcement and positive punishment will be used. From a theoretic point of view, the environment provides a reward to the agent. Many real-world environments lack the infrastructure to do so, which means the agents must generate their own reward based on their perception of the environment. This distinction is significant as the reward signal then requires the same properties as are needed to use MDPs. When they are not present in the environment, receiving a reward signal suffers from the same ambiguity that the agent does in mapping its state to the environment’s ‘true’[6] state. The convergence of RL is dependent of several factors. The reward at a state can not change or must come from a static distribution. The environments or other agents can not cause reward to shift.

[6]Even after infinitely many samples of an environment, the value function will only represent the expected value of a state-action based on historical experience. In complex environments, this may not be a correct expectation.

Effectively RL, takes steps toward the actual expected value for a state, if this expected value is moving then RL may never reach it.

2.4.1 Distributed W-Learning

In the previous Section, RL was discussed in the single agent, single objective case. When multiple agents and objective are used, the algorithm must be changed. There are few multi-agent, multi-objective RL algorithms. In Section 2.2, the two categories of approach to Multi-Objective Systems were discussed. Combining state-spaces has several drawbacks, so this thesis will focus on arbitration-based approaches, though it is equally applicable to combined state-spaces approaches. Distributed W-Learning (DWL) [Dusparic and Cahill, 2012] is used as a base, as it has several desirable features both as an algorithm and for using with other algorithms:

- Agents are capable of independent learning. This is advantageous when systems can suffer partitioning. Agents, therefore, have their own representation of knowledge which they alone affect.
- A learning process for each objective means that they can be added and removed as needed with minimal re-learning. It also separates concerns well, which allows each objective to be encapsulated.
- The relationships between objectives are dynamic, which allows their priorities to change over time. Having a separate representation for inter-objective priorities isolates the two main learnt types of information—what is best according to an objective and how important that objective is—to be learnt from one another.
- It has been shown to be effective in large-scale systems [Dusparic and Cahill, 2009].
- Information is represented in a common RL form, so it can be interchanged with any other RL algorithm with minimal effort.

DWL effectively gives each objective at each agent its own Q-Learning process and a W-Leaning process [Humphrys, 1996]. W-Learning works by providing each state a

W-Value for each of the agent's objectives in addition to its Q-Value. When an action is to be taken each objective's learning process selects an action as normal, but rather than being executed, the actions are arbitrated between. An example of this is in Figure 2.4. The arbitrator selects the action which "minimises the worst unhappiness"[7].. In practice this means the suggestion with the highest W-Value is executed. The W-Values represent how important a state is to an objective's learning process. If a state has a low W-Value, it means that the agent is not particularly affected if its suggestion (selected action) is obeyed or not. This could be because any possible result has little impact on performance or because historically its suggestion (or similar) has been obeyed for another objective's learning process. In the case where multiple objectives suggest the same action, only one can win and have its suggestion obeyed. The W-Values are updated when an objective's learning process is not obeyed. Updates use the following rule:

$$W(S_t) = (1 - \alpha)(W(S_t)) + \alpha(Q(S_t, A_t) - R_{t+1} - \gamma \max_A Q(S_{t+1}, A_{t+1})) \tag{2.4}$$

where $W(S_t)$ is the W-Value of the state that the agent was in when the losing action was suggested. R_{t+1} is the reward for the actual action that was executed, the remaining terms are the same as in the Equation 2.2, Q-Learning's update.

In W-Learning, an agent learns the Q-Value as normal. The value of the action that actually happens is updated, not what an objective suggested. Consider an agent with two objectives and two actions, A and B in some state. At a particular time, the first objective's learning process selects the action A, while the other selects B. Only one can win, so if action A happens, the first agent updates the Q-Value as normal, the other agent updates its Q-Value as though it selected A and updates the W-Value for the state in question.

DWL also uses remote policies. These are policies representing the objectives of neighbouring agents. They use the neighbour's state variables and the local agent's actions. Updates are then shared about how much reward is received by neighbours

[7]Other selection methods are discussed, but this is the best for collaborative systems[Dusparic and Cahill, 2012]

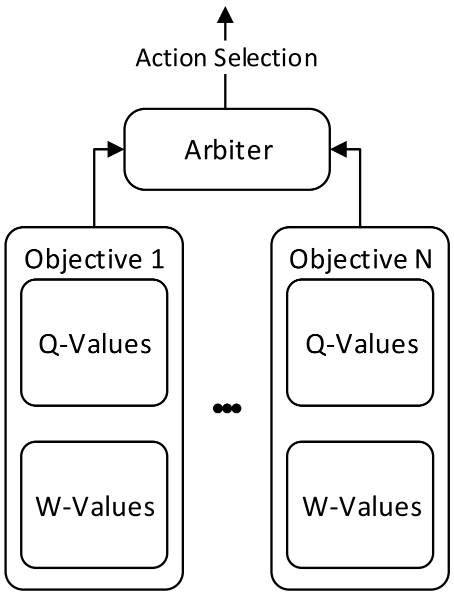


Figure 2.4 An Example W-Learning Process.

and in what state they are. This allows agents to learn how their actions impact on their neighbours. Remote policies—like local policies—suggest actions, which means that at any time, an agent can choose to do what is best for a neighbour rather than for itself. This arbitration is done the same way, each policy suggests an action and provides the associated W-Value. The only difference is that the W-Values of remote policies' suggestions are scaled by a value called the Collaboration Coefficient C. C is in the range $[0, 1]$, it can be set statically or learnt. By attaching a learning process to C, the best value can be found and it can change over time allowing the system to adapt to changing circumstance.

As W-Learning has a Q-Learning process for each objective's learning process, it can learn the optimal[8] policy for that objective. When an agent has multiple objectives there

[8]Optimal if the environment is effectively simple—agents not impacting each other etc.—otherwise optimality can not be guaranteed.

is no guarantee that the resulting interleaving of objectives' suggestions will be Pareto optimal even if each objective has learnt an independent optimal solution.

DWL uses W-Learning to arbitrate between objectives' learning processes. Each agent has a learning process for each of its objectives and one for each of its neighbours' objectives. This allows it to learn how an action impacts on other agents and it can therefore minimise the least unhappiness, not just locally but across several agents.

More formally a DWL system at a given time will have the following:

- A set of agents, $\{A \mid A_1, \ldots, A_Z\}$, where Z is the total number of agents in the system.
- A set of neighbours for each agent, $\{N^a \mid N_1^a, \ldots, N_Y^a\}$ where Y is the number of agents the agent A_a can communicate with.
- A set of local learning processes for every agent in the system, $\{LLP^a \mid LLP_1^a, \ldots, LLP_X^a\}$, where X is the number of objectives implemented by the agent A_a.
- A set of remote learning processes RLP for every agent in the system. Each local learning process in LLP^n of the agents in the neighbour set N^a is represented in RLP^a, such that $\{RLP^a \mid \overline{LLP_1^1}, \ldots, \overline{LLP_X^1}, \ldots, \overline{LLP_1^Y}, \ldots, \overline{LLP_X^Y}\}$, where the overline indicates the local policies LLP_x^y have to have their action-spaces replaced with that of the agent a.

An example of a DWL agent is given in Figure 2.5. In the figure, $V = \sum_{i \in N^a} X^i$ which is the summation of all A_a's neighbours' Xs.

2.5 Accelerating Learning

RL can take a long time to learn [Barto and Mahadevan, 2003]. This is because each state-action pair must be sampled several times to learn the correct value. In complex environments—those exhibiting non-stationarity or dynamism—more samples are

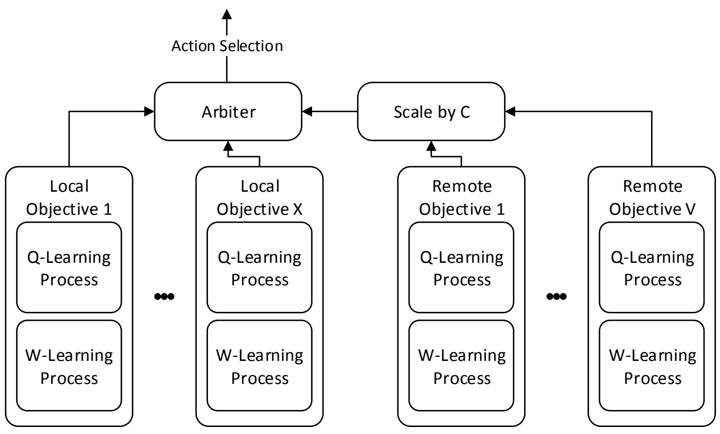


Figure 2.5 An Example Distributed W-Learning Agent with X Local Learning Processes and V Remote Learning Processes.

needed as single samples become less representative. The presence of multiple agents or objectives affects how representative samples are and thereby increases the amount needed to learn a representative value [Busoniu et al., 2008]. In arbitration-based approaches to multiple objectives, each objective has its own learning process, which increases the amount of learning that is needed as each objective's learning process needs to learn a representative value. While an agent is exploring, its performance is necessarily poor. To know that an action is sub-optimal, the action must be executed—probably several times—which will give sub-optimal performance. This means that the time spent exploring should be used as efficiently as possible to maximise performance over the agent's lifetime. This can be seen in Figure 2.6 which is a stylised performance graph. In it the red dashed line is a more rapidly learning process, the area under it is superior to that of the green solid line. It also achieves its maximum performance earlier, this means the system needs less time for its behaviour to normalise. In this example the two lines converge to the same level. This is not necessarily so, acceleration schemes can allow better performance to be achieved, particularly in complex environments where learning may not be able to find the optimal solution. Generally, acquiring samples of an environment to learn from is an expensive process, as it affects the environment and other agents, so it should be kept to a minimum. For these reasons, learning should be accelerated.

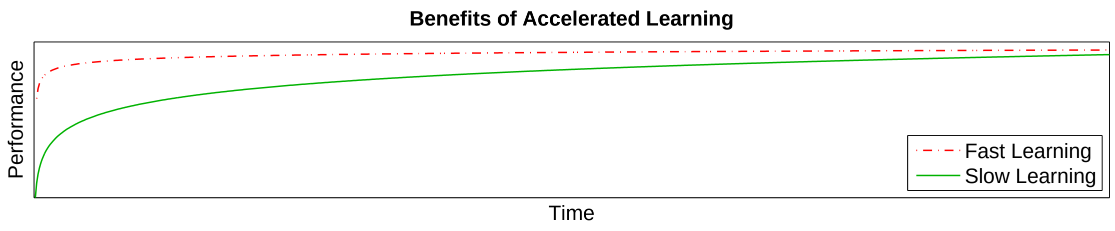


Figure 2.6 An Example of Performance with Different Learning Rates.

In RL, schemes to accelerate learning operate in two main ways; they can either improve the efficiency with which samples are used or they can generate new samples (either directly or virtually). Either method can fall in to one of the following three categories of approach to accelerating learning:

- **Algorithm Alteration** is changing the algorithm so that it requires less information to learn. These approaches aim to use data more efficiently; effectively learning more from less. Planning type algorithms [Hafez and Loo, 2015] are an example of this (further described in Section 2.5.1).

- **Generalisation** of experiences allows one experience to be representative of others. Conceptually, this means that a sample of reward from one state-action pair can be applied to several state-action pairs. Function approximation [Sutton et al., 2012] is an example of this (further described in Section 2.5.2).

- **Additional Knowledge** approaches involve providing an agent with some information from outside of its own experiences. This reduces the amount of learning to do. Transfer Learning [Brys et al., 2015] and Reward Shaping [Laud, 2004] are examples (further described in Section 2.5.3).

The different approaches have strengths and weaknesses which will be discussed in the following sections.

There are several problems that can exacerbate RL's inherently slow knowledge acquisition rate. Some approaches are more effective on particular problems than others.

These problems are often caused by fluctuations in a system's behaviour due to learning or inherent from large-scale autonomic systems. The following list encapsulates these problems. The categories here cover all causes of slow learning [Kaelbling et al., 1996; Sutton and Barto, 1998].

(I) **Credit Assignment** is a problem that occurs due to RL's update scheme[9]. If there is a large change in a successor state, it can take many iterations for this change to back propagate to all paths leading to it. This means that for a time, the prior states will be incorrect and require more learning.

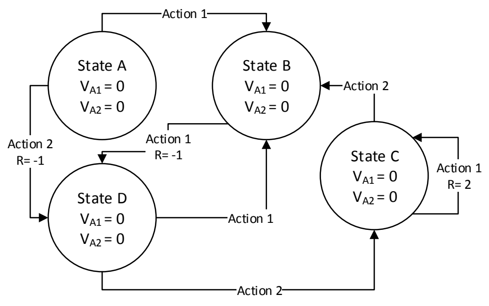
(II) **Sparsely Visited States** are those that will not be visited enough for learning to complete. This may be caused by insufficient exploration or if the agent can not move itself into the state.

(III) **Sample Variation** is how much an interaction with the environment can change between repetitions. The more a sample can change, the longer learning takes. This is difficult to address without an accurate model of the environment. It is affected by the complexity of the environment, which is discussed in Section 2.3.2.

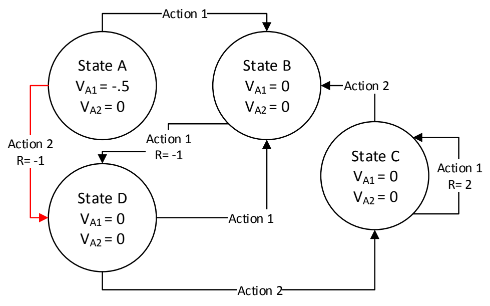
The credit assignment problem (I) is demonstrated in Figure 2.7. It shows the progression of an agent's value function using an off-policy temporal difference learning algorithm. The values of the states have already had the update for the selected action applied. For clarity the simple update rule in Equation 2.5 is used. In Figure 2.7d, Action 1 in State C is converged (effectively learning is finished there), but the action leading to State C has no knowledge of this and its value does not represent that it leads to the 'good' state. Until the transition from State D to State C occurs, the value of neither can be properly gauged by other states. The previous Figures 2.7a - 2.7c show its progress to the State C and changes in its value function.

$$V(S_t, A_t) = \frac{R_{t+1}}{2} + \frac{\underset{A}{\max} V(S_{t+1}, A_{t+1})}{2} \tag{2.5}$$

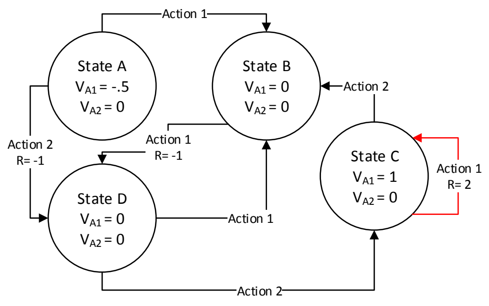
[9]The value function for all following states is typically estimated with a single state's value by temporal difference.



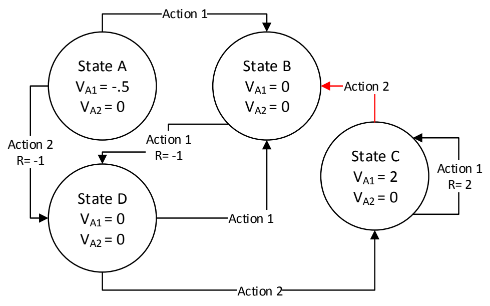
(a) An Example Learning Process.



(b) Action 2 Selected from State A



(c) Action 1 Selected from State C.



(d) After Many Iteration of Action in Figure 2.7c, State C converges.

Figure 2.7 An Example of Unequal Credit Assignment.

There was no discounting in this update rule which is commonly used in larger problems than this. In this small example, it is likely that any reasonable exploration scheme will adequately cover the states and the value will propagate back, but in larger MDPs the problem is obvious. Additionally, the sparsely visited states (II) problem is shown by State A. There is no way for the agent to get there by itself, it will only experience it when it is driven there by the environment. Seldom visited states like this can take a long time to learn in, due to the sparsity of their sampling.

The following sections will go through the various approaches to accelerating learning with respect to the identified problems.

2.5.1 Algorithm Alteration

There are many different RL algorithms which have inherently different rates of learning. This section will focus more on the generally applicable techniques rather than this variation. The problem of credit assignment (I) discussed above is often addressed by *Prioritised Sweeping* [Moore and Atkeson, 1993]. It works by maintaining a queue of states that might benefit from an update to their value function and a priority for each. Updates are then applied to the states with the highest priorities. The calculation for the priorities is based on whether a state might have changed and should be added to the queue, and it depends on the algorithm. In model-based ones, the model is used to find states that can lead to a changed state [Van Seijen and Sutton, 2013]. In model-free approaches, memory (effectively a history of which states lead where) is commonly used [Kaelbling et al., 1996]. This allows credit to be back-propagated more quickly and thereby state-action pairs achieve the correct value more rapidly. While Prioritised Sweeping is effective in model-based algorithms, it is less so in model-free approaches. The computational and memory cost of maintaining a pseudo-model of which states lead where is significant, particularly in model-free where there is no algorithmic model to support this process.

Prioritised Sweeping addresses the credit assignment problem (I) by directing updates to states where there may be misassigned credit. It does not address sparsely visited states (II). In complex environments, states can get added to the update queue and have their value changed as a result of a transient change in the environment or even from sample variation (III). This generates extra processing and potentially incorrect updates. However, it can react faster to a permanent change. The risk of overreacting to transient change is particularly acute in MASs where other agents are affecting a shared operating environment; a set of updates can be caused by the effect of another agent's action.

Prioritised Sweeping in model-based environments is closely related to *Dyna* [Sutton, 1990]. In Prioritised Sweeping, the model is used to target updates efficiently, in Dyna it is used to generate them. Interacting with the physical world takes time, so learning by doing so is slow when compared to the computational speed of whatever physical device is running RL. This can leave processing time available for other tasks. During this spare time, Dyna uses the model to generate virtual experiences and learns from them as well. Assuming the model is accurate, the virtual experiences are as good as real ones. It can be very effective at correctly assigning credit and thereby addressing the credit assignment problem (I). In sparsely visited states (II), as there are effectively sufficiently many experiences available to learn from, this problem can be completely addressed. If a complex environment is suitably well modelled, then the sample variation problem (III) can be reduced as well. If the environment is complex due to other agents, then it can not help as their actions are, from Dyna's point of view, unpredictable and can not be modelled. Generally, Dyna has a significant reliance on the model and its accuracy. If the model is inaccurate, then learning's effectiveness will be reduced. This makes it impractical for large-scale systems, due to the complexity in accurately modelling them.

Another approach to making learning faster is by the use of *Macro Actions* [McGovern et al., 1997]. These are effectively combinations of 'normal' actions which are used in series. This means that agents do not need to learn the relationship between all actions, but rather sets of actions. It is a particularly popular formulation in robotics. For example, a robotic arm with multiple actuatable joints could take a long time to learn how moving one affects the others, but if given Macro Actions like 'grasp' it will learn much faster [Amato et al., 2014]. It is only applicable in some applications and it limits how actions can be chained together to whatever blocks the Macro Actions are in.

2.5.2 Generalisation

In RL applications, the typical way to represent the value function is as entries in a table. This approach becomes problematic when the state-space is very large or continuous. A common way of addressing this problem is *Function Approximation* [Geist and Pietquin, 2011; Prashanth and Bhatnagar, 2011]. Rather than exactly defining the value function at every state, a function can be used to represent the state-space. The representation can be Neural Networks [Lee and Anderson, 2014], Decision Trees [Tan, 2014], Laplacian Eigenfunctions [Gatti, 2015] or any other function. Deep Neural Networks have shown very impressive results, but take huge amounts of training (about 38 days of real execution in this example) [Mnih et al., 2015]. The function needs to exhibit the Markov property and allow for non-stationarity. The functions used generally fit into the following categories [Geist and Pietquin, 2011]:

- **Bootstrapping** treats the approximation process as a Supervised Learning problem and uses an estimator to minimise some cost function. Gradient Descent methods are an example, and in these methods, the function is adjusted by taking a step which minimises an error term. This aims to refine the function used at each time-step. It handles non-stationarity by always moving down the error gradient, agnostic of its cause.
- **Residual Approaches** work by minimising the distance between actual value function and the approximated value function. These approaches minimise cost functions to achieve good approximations.
- **Projected Fixed Point Approximation** tries to find a point which minimises a cost function based on the optimal policy and its projection under the approximation.

To some extent, nearly all RL systems use function representation. There is always some mapping from an environment's 'true' state to the RL representation's state. Commonly, this is some form of quantisation of a variable into bins of ranges. Using a suitable function as an approximation, allows knowledge to be generalised. Effectively, an experience in what was previously one state is applied to other areas of the state-space. This means that each experience is more efficiently used. It can also be more memory efficient depending on the implementation, although there is extra complexity introduced to calculate and maintain the functional representation. The main drawback of such approaches is inaccurate generalisation. If the function and the 'true' value are significantly different for a particular state-action pair, then generalised knowledge will not be applicable and will not induce the correct behaviour in an agent. This in turn will reduce performance, which can occur if the function is poorly chosen or if its accuracy varies too much across the state-space. Function Approximation has little effect on credit assignment (I), as knowledge is not necessarily generalised to adjacent states in some trajectory. This means that changes to successor states are not necessarily propagated any quicker. It is effective in sparsely visited states (II), as a good functional representation can remove the isolation of sparsely visited states by linking them to more often visited ones. Assuming the function accurately represents the state-space, then sample variation (III) can be accounted for more quickly, but practically this is difficult.

An alternative approach to generalising knowledge is to use a *Feature-Based Representation* of knowledge [Sutton, 1996]. Feature-Based and Function Approximation can be seen as similar, as a function effectively maps states to features. Example features are distance to an object [Martínez et al., 2015], first-order logical statements [Palombarini and Martínez, 2012], and semantic similarly of words [Sánchez et al., 2012].

2.5.3 Additional Knowledge

Most approaches to RL start learning from scratch, assuming that no knowledge exists about a problem. This rather naïve assumption means that regardless of the problem or what is already known about it, an agent must always acquire all of its own knowledge. In reality, for most problems there is at least a partial solution. This prior knowledge could come from a domain expert, previous attempts to solve the problem, or just a designer's intuition. Regardless of the source of or the reliability of previous knowledge, it can be used to support the learning process. The intuition being, it takes less time to learn a solution from some partial solution than from nothing. This assumes that the partial solution being used as a basis is not 'wrong'[10]. If an agent is supplied with incorrect information, it must first 'unlearn' it, then learn as normal. Obviously this will increase learning time. The term target is used to describe the recipient of additional knowledge, while source is used to describe the process that supplies knowledge.

[10]Wrong here means that the value function derived from the partial solution is not an intermediate step to the actual value function of the target problem, not that it did not work in its own problem.

Any method of reusing knowledge requires two things, that the knowledge can be expressed and is readily available and that it can be supplied in some intelligible way to an agent. The former condition is only really problematic if the required knowledge is in some implicit form (such as an emergent behaviour). For the latter condition there are several approaches. They vary in how the information is supplied and in what form, but they all use the assumption that reuse of knowledge improves performance.

Knowledge can be provided in the initialisation of the value function, it can be provided through a reward function (actions thought to be good can be given higher than deserved reward, at least initially) and finally biassing the action selection function in some way can force the agent to explore those states thought to be good first. These are all used to affect the decisions taken by an agent, which in turn changes the way it perceives and explores the environment. Biassing the action selection affect its exploration directly, while both initialisation and reward shaping affects exploration indirectly.

2.5.3.1 Reward Shaping

Reward Shaping is the process of adding an extra signal to an agent [Ng et al., 1999]. This extra signal represents the knowledge that is being supplied. The risk when adding a second reward signal is that it will obscure the normal optimal policy by making some policy appear better than it actually is. To prevent this from happening, a technique called Potential-Based Reward Shaping can be used. The potential at a state is the difference between the shaping function at two different states. The basic idea is that the potential for a particular state can be applied in a similar way to how temporal difference works with value functions. When a transition occurs, *Potential-Based Reward Shaping* gives the discounted difference between the potentials of the states S_t and S_{t+1}. This effectively means if an agent is moving up the potential gradient it earns extra reward, if it is moving down the gradient it is slightly punished. It has been proven mathematically not to affect the optimal policy learnt by Q-Learning [Ng et al., 1999] as well as other RL algorithms [Wiewiora, 2003]. When used in MASs, it has been shown not to affect the Nash equilibria [Devlin and Kudenko, 2011]. While it is a weaker guarantee than the optimality guarantees that exist in single agent cases, most other approaches do not have any guarantees. Static shaping functions only allow knowledge that is held prior to execution to be supplied. This limits their usefulness in large-scale or complex environments. *Dynamic Shaping* can be used in these cases [Laud, 2004]. It has the nice feature that as long as the function generating the potential-based signal converges, then the agent's behaviour will converge. This is not a necessary condition, however, as behaviour can converge even if the potential function does not [Devlin and Kudenko, 2012]. This means that regardless of where the knowledge comes from, as long as it eventually converges, then the agent's behaviour will as well.

Reward Shaping alone can not accelerate learning, it is the mechanism by which knowledge is integrated into an agent. The knowledge needs to be acquired from somewhere. Without a way of doing so automatically, its usefulness in on-line applications will be limited. Automatically generating shaping functions has been attempted several times. Most approaches generate a more general version of the original MDP [Marthi, 2007]. This generalised MDP has fewer states, each of which represents several states in the original. Anything that is learnt in a state in the general MDP is then used as the shaping function for that state's sub-states. This approach to automating the shaping function's generation is based on the idea that the ideal shaping function is the 'correct' value function for the original MDP. The generalised MDP is used to learn an approximation of this. Since learning the value function to accelerate the learning of the value function is paradoxical, this approach used Dynamic Programming to solve the general MDP and then applies its knowledge as a shaping function. There are two defined steps, learn in source, then provide knowledge to the target. By using a model-free algorithm instead, knowledge can be provided continuously, rather than after learning in the generalised MDP is complete [Grzes and Kudenko, 2008]. These approaches are effectively using one environment with multiple learning processes. If the environment can be sufficiently well simulated, then an agent can learn in similar virtual problems (which are more simple) and use this knowledge to shape the main learning process. This can be done by learning how reward relates to its own actions and percepts (agent-space), rather than based on environmental parameters (environment-space). For example, given a robot that wishes to learn to solve a maze, if it is learning in an agent-space, then a proximity sensor reporting walls on three sides (a dead-end), should result in backtracking behaviour. In an environment-space, it might learn to backtrack from a specific set of coordinates. It is much easier to generalise from the agent-space, as the environment might change, but what the sensors perceive will not (given we assume the simulated environments are close to the actual environment). A similar approach uses a feature-based agent-space [Konidaris et al., 2012]. This allows the agent to pass not only its own agent-space representation but common environmental features as well, making it more general. More diverse tasks can be used when coupled with Transfer Learning and Inter-Task Mappings [Brys et al., 2015] (see Section 2.5.3.3). Rather than autonomously producing the input knowledge, Reward Shaping has been used as a vehicle for applying human knowledge to RL. This will discussed further in Section 2.5.3.4.

Shaping can only improve the credit assignment problem (I) if it shapes the agent towards states which correct it, similarly for sparsely visited states (II). Sample variation (III) can only be addressed if the agent is induced to visit a state more frequently, however this reduces visits to other states for a fixed training time. Its main method of improving learning is driving the agent to beneficial areas of the state-space earlier and thereby improving performance.

2.5.3.2 Selection Biassing

Rather than using a shaping function and affecting the value function to supply additional knowledge, action selection can be influenced [Fernández and Veloso, 2006]. For example, if a particular area of the state-space is known to be good, then actions leading to that area can be disproportionately selected. This will force the agent to explore the ‘good’ area earlier and more thoroughly than it may originally have done. Determining a priori which areas of the state-space an agent should be directed towards is a complicated task, as with all additional knowledge input selection. This section will focus on the automatic selection of knowledge, while human supported approaches will be discussed in Section 2.5.3.4.

There are two categories of information that can be used to bias action selection: information from prior tasks and information from the learning process [Bianchi et al., 2008]. The former requires that knowledge is available prior to the target task's execution, while the latter can be established at run-time. Both operate in a broadly similar manner; they try extract structure from the environment in much the same way as a learning model. This structure is then used to guide exploration. If a particular sequence of actions is found to transition to a goal state (or other 'good' area of the state-space), then biassing encourages the agent to explore this area preferentially [Bianchi et al., 2007]. This does not reduce the amount of exploration or the time it takes, it just prioritises it. The exploration that has the greatest effect on performance is done first. In applications with limited training time or resources, this allows good behaviour earlier, but the rest of the exploration will still need to be done to guarantee an optimal policy. In effect, *Selection Biassing* tries to get an agent to follow a (known or expected) good partial policy which is to some degree equivalent to Reward Shaping. Selection Biassing has no direct effect on any of the problems affecting learning (credit assignment problem (I), sparsely visited states (II) or sample variation (III)), it only addresses them by increasing the frequency of visits to given states which necessarily reduces visits to others.

2.5.3.3 Transfer Learning

An alternative method for applying additional knowledge is *Transfer Learning (TL)* [Taylor and Stone, 2009]. TL is based on an idea borrowed from psychology. When learning how to accomplish a task, knowledge from a related task is often used as a starting point. In terms of RL, this can be reusing policies, partial policies, models, state-action values, features or anything else that can be learnt. TL operates between a source task (the process that provides information) and a target task (the knowledge recipient). For transfer to occur, knowledge needs to be mutually intelligible. This is accomplished through an Inter-Task Mapping (ITM) denoted by χ. ITMs translate knowledge from the source—hopefully usefully—to the target. The ability to map data effectively presupposes that there exists some useful knowledge to share. This is not always true, as there is no way of knowing if there is useful information to share except by actually trying it. This gives rise to the concept of negative transfer. Negative transfer occurs when sharing information actually increases the learning time. This happens when incorrect knowledge is supplied and it must be unlearnt before the correct solution can be discovered. Generally the closer related two tasks are, the less likely it is that negative transfer will occur [Taylor, 2008]. Negative transfer can also occur between agents that have useful information to share if the mapping is inaccurate [Konidaris et al., 2012]. TL is an off-line process. A source task completes its learning, the information is mapped and provided to the target task which then learns.

Many attempts at TL require the designer to supply an ITM or limit the tasks to the same or very similar environments which render the mapping trivial [Taylor and Stone, 2011]. While these approaches can be effective in small applications, to be useful in real-world systems agents will need to be able to self-calculate or at the very least select from a library of possibilities. The selection of one ITM from many could be equivalent to learning a single ITM, if the set of possible ITMs are suitably good or there are enough others that a good one is likely to exist. Either of these possibilities would make it likely that selecting from multiple ITMs would lead to finding a single good ITM. While not the intention of this work, Fachantidis et al.'s approach could be used to do so [2015]. They use 48 ITMs in the largest environment they examine, but these are direct ITMs (the source and target have the same state-space), so they are effectively finding good pairs of agents to transfer between. If the source and target task were to differ, more so than in this work, then more ITMs or more complex ITMs would be needed.

In general the greater the source and target are allowed to differ, the more challenging calculating the ITM is. Tasks can differ in the following aspects [Taylor and Stone, 2009]:

- **Action-Space** can differ. Each agent can have more or less actions, different actions or the same actions with different effects (e.g., more powerful actuators).

- **Problem-Space** can vary while the agent remains identical. This is commonly used when an agent is trained in a more simple task, then ‘promoted’ to a more challenging one. Changing the number of objects in the problem is also possible, for example moving from learning to find one moving target to two.
- Changes in **Reward or Goals** can cause the agent’s value function to differ, even if everything else remains constant.
- **Entry or Exit Points** can differ which will invalidate (to some extent) prior information in the value function. For example, the direction an agent must solve a maze could be reversed.
- **Representation of States** can be changed. This then necessitates the mapping $\chi_{S_{old} \to S_{new}}$ for all states.
- **Transition Function** is the representation of how an environment responds to an action, this could be changed by addition of other agents etc.

For use in large-scale systems, an ITM must at the very least be able to accommodate changes in the transition function (brought about by other actors in the system). For most real-world systems with multiple stakeholders, ITMs will need to allow variation in all categories.

Case-Based Reasoning has been used to describe the differences between tasks and extract knowledge to transfer [Celiberto et al., 2011; Junior and Matsuura, 2011]. The use of Case-Based Reasoning allows the algorithm to function agnostically of the RL representation. An action selection biassing approach is used to provide the knowledge to the target agent. The problem with this approach is that it is highly dependent on the action-space and the ability to describe problems based on actions. In this case, the action mapping is hand-coded by a designer. While action-spaces are typically small—when compared to the state-space—having a designer produce ITMs is not scalable in on-line systems. It also limits transfer to identical agents. Ammar and Taylor describe an approach using common subspaces to facilitate transfer [2012]. Using a mapping for both the state and action-spaces relaxes the requirement for identical agents. The shared subspaces are generally of lower dimensionality than source or target and composed of shared features. The ITM is learnt based on the distance between pairs in the common subspace. Finally, a partial policy is found in the source task which is then translated to initialise the target task. This approach requires that the subspace is specified. Specifying all or part of a mapping is not possible given all the different combinations of agents that could occur in a large-scale system, so a completely automated approach is needed. For example, Chatzidimitriou et al. train a Neural Network to map between two tasks [2012]. In this example, they transfer information in the Neural Networks space (topologies and weights in an Echo State Network) rather than directly transferring RL features. This provides a degree of extraction which generalises the approach. Another autonomous approach is to produce a high dimensional space as an intermediary step [Ammar and Taylor, 2012]. The idea being that mapping to and from a midpoint is easier than mapping from source to target in one step. A high dimensional space is produced through Sparse Coding, the source and target are then projected into this space by solving an optimisation problem. Source to target pairs are then selected based on the Euclidean distance of their projections in the shared space. By applying a threshold to the projected distance, negative transfer can be avoided. Unsupervised Manifold Alignment has been used to generate the high dimensional space [Ammar et al., 2015]. The discovery of pairs is similar, but they are judged based on their closeness when projected back to the source task rather than in the intermediate space. Calculating ITMs is complicated and time consuming which limits their applicability to off-line problems or those where all training is done off-line. The ability to use arbitrary information to accelerate learning is potentially significant as it would enable TL and RL to be used practically in real-world systems.

Most work on TL has been in off-line applications. Here, the normal informational flow—learn in source task, transfer, learn in target—makes sense. However, when used in on-line or large-scale systems, this flow is impractical. Running a source task just to accelerate learning in each part of a large-scale system introduces significant overhead, exactly what accelerating learning is trying to avoid. In on-line systems, it is not always possible to run a source task before the target task. So, in these situations the way TL is applied has to change. Biassing the initial value function has been used to accelerate learning in MASs [Boutsioukis et al., 2012]. In this approach each agent is given initial information from an agent in either a single agent system or a MAS. They find little difference between single or multi-agent sources, but were limited to two agent sources.

In MASs, agents often learn broadly similar things. This repetition makes them likely to be good source tasks to each other, as closely related agents typically have mutually useful information. However, in most MASs, the inter-relatedness of the different parts makes them difficult to isolate, so one part can not easily be run before the others (this does not preclude transfer of information from other sources). Garant et al. propose a co-learning method to address this [2014]. This approach allows some agents to be supervisors to others. Supervisory agents manage the mapping and selection of which data to transfer and to whom. They use context features to relate agents with different state-spaces. Context features are specified by designers and represent sections of the transition model and reward received. The supervisor agents find pairs of agents that have similar context features and transfer between them. A similar scheme uses a hierarchical ontology to represent the state-space [Kono et al., 2014]. In this approach each agent's designer specifies an ITM to a central ontology. Each agent then has a way of representing information in a globally understandable way. The specification of the ontology and the mapping to it require designers to produce. The authors state a belief that there will always need to be humans to translate data due to the plurality of potential source task both inside and outside of a particular system. In many cases, humans will be involved in the learning process, so using them to aid in mapping is acceptable. However, it is not scalable and impractical in on-line tasks.

TL improves the credit assignment problem (I) by providing an estimate of the final value of a state, so less change is needed to get to the 'correct' value, therefore fewer samples are required. In sparsely visited states (II), it likewise provides an estimate of the value, so even if the agent can not visit the state sufficiently to learn, it will at least have a partial representation of what to do in it. Sample variation (III) too can benefit from extra knowledge, as long as the transferred data is representative of the variation, then the variability of a state can be represented in the value function without having necessarily experienced it.

2.5.3.4 Human Training

The source of additional knowledge can be a human ‘in the loop’. In this case, a domain expert can provide knowledge either before or after a target task has started. Human training can be practical for large-scale systems when good policies are already known (for example, in a previous control system) and are available prior to the system starting, but generally it is limited to smaller problems. Learning from an expert can be challenging, as specifying the additional knowledge in RL terms is non-trivial [Abbeel and Ng, 2004]. Instead, an agent can learn from the expert in the target problem and extract a reward function from the expert without it being specified. The *TAMER* framework provides an alternate method for integrating human knowledge [Knox and Stone, 2009]. The framework provides generalisation capabilities as well as allowing human training. This means that the interaction with the trainer can be minimised, as it is impractical for a trainer to be present for the entire learning process. It can also be used with a reward function and a human providing feedback [Knox and Stone, 2012]. The effect of trainer ability has been examined [Taylor et al., 2011]. While better training typically improves early performance, over time the performance reaches comparable levels.

Human Training can not really address the credit assignment problem (I), as the underlying algorithm is still operating the same way; it just has a better starting point or correction of poor performance provided, and it still has to represent this. If the trainer can directly affect the value function then the sparsely visited states problem (II) can be addressed. Variability of samples (III) can be addressed only if the trainer provides more sample to the agent. The main benefit of Human Training comes when the agent's exploration is limited to a subset of the problem-space by the trainer effectively reducing the area the agent has to search for a good solution.

2.5.3.5 Conclusion

To some degree all the different algorithms that apply additional knowledge are part of a larger single class of algorithm. The mechanism by which knowledge is supplied differs, but the underlying idea is the same. The quality and timeliness of the data is the most significant factor in the performance of all of them. The most complex requirement they have is the translation component where the supplied knowledge is cast into a form that is intelligible to the recipient. The production of these translation functions has typically been hard coded or the differences between tasks have been limited. This is the major barrier to more wide-scale adoption of these methods.

2.5.4 Summary

There are broadly three categories of approach to accelerating RL: altering the particular algorithm used to improve its performance; generalising locally learnt knowledge, so that it is applicable to more stations; and applying additional knowledge from external sources, so less learning is needed. These categories of approach and particular approaches within them have different strengths and weaknesses discussed previously. Table 2.1 summarises how the methods discussed above address the main problems that slow learning. A ✓ indicates the method addresses the problem well or directly, a (✓) is given if a method is partially effective or could address the problem in some circumstances, a ✗ indicates the problem is not addressed. None of these methods operate completely on-line. Reward Shaping and Selection Biassing have applied knowledge on-line, but not acquired it on-line. TL can not supply information on-line. This prevents them from effectively reusing knowledge in a MAS.

2.6 Performance Measures

To know if learning has been accelerated or not is not quite as simple as it may first seem. In RL, an algorithm aims to learn the value function for an agent. This would indicate that the time until the value function converges would be the best metric to

Method	**(I) Credit Assignment**	**(II) Sparsely Visited States**	**(III) Sample Variation**
Algorithm Alteration			
— Prioritised Sweeping	✓	✗	✗
— Dyna	✓	✓	(✓)
— Macro Actions	✗	✗	✗
Generalisation			
— Function Approximation	✗	✓	(✓)
— Feature-Based Representation	✗	✗	✗
Additional Knowledge			
— Reward Shaping	(✓)	(✓)	(✓)
— Selection Biassing	(✓)	(✓)	(✓)
— Transfer Learning	(✓)	(✓)	(✓)
Human Training			
— TAMER	(✓)	(✓)	(✓)

Table 2.1 Methods of Accelerating Learning's Impact on Problems.

use. However, it is perfectly plausible that the reward function or the environment will constantly generate different values, which means that regardless of the algorithm used, the value function may not converge. This is particularly true in large-scale systems and MASs.

Another option is the time for an agent's behaviour to reach a steady state. It is possible for an agent's action selection to stabilise, even if the value function is still changing. This can be because the changes in the value function are negligible or it can be that sections of the state-space finish learning before others. In systems with multiple objectives, behaviour can change when a different objective controls the agent regardless of if it has learnt or not. The behaviour of an agent can be dependent on other agents in a system, so its behaviour may never stabilise. These factors make agents' behaviour a poor indicator of learning speed.

With value functions and action selection impractical, it is difficult for an agent to tell if it has finished learning based only on its own operation. Application-dependent metrics are necessitated. Performance can be measured in reward received or in some application specific measure. Using performance can be a more holistic metric as it also represents the ‘quality’ of what was learnt. Learning quickly should not worsen performance in a given task. The following are the ways metrics can be judged [Taylor and Stone, 2009]:

- **Jump Start** is the initial improvement in performance when using a technique over not using it.
- **Asymptotic Performance** or final performance is how well an agent can do in a task given sufficient time to learn. Sufficient time is judged by when the level of reward attained becomes reasonably stable.
- **Time to Threshold** is how long it takes the agent to get a given level of performance.
- **Performance After Time** is given a specific amount of training how well can an agent do.
- **Total Reward** how much reward is accumulated over an agent’s lifetime. Faster learning agents should receive more reward over their lifetimes.

Each of these metrics has its advantages. No single metric covers all possible improvements (or disimprovements).

Another issue with calculating learning time is how should any additional calculation be apportioned. For example, with an additional knowledge-based approach, should the time to learn the additional information be included in the learning time for the target task or should it be thought of as part of the design process. A similar issue arises in how time is described. Typically performance is given with time-steps as the unit. However, not all time-steps are equal to one another. They can change based on how much ‘real time’ they represent. Consider a load balancing problem in a server farm, a time-step could be half an hour or a few seconds, obviously this affects how much calculation can be done per time-step. Different algorithms also value time-steps differently. A simple Q-Learning process only updates one state per time-step while a more complex memory based algorithm could update dozens of states. Some algorithms can also generate virtual experiences by sampling the reward function or model, this means for each 'real' experience there can be many virtual experiences. This makes it difficult to compare one approach with another. Typically the performance is calculated by base algorithm and improvements versus base algorithm alone [Ammar and Taylor, 2012; Lu et al., 2015; Pan and Yang, 2010] or different variants of the same approach [Hu et al., 2015]. There is a need for either a standard data set or a task generator, so that different approaches can be compared directly, however this is not provided in this thesis.

2.7 Discussion

Accelerating RL can be done in three main ways:

- **Algorithm Alteration** is a change to the way RL operates so that it converges more quickly or makes better use of data.
- **Generalisation** involves taking one interaction with the environment and making it applicable to other situations.
- **Additional Knowledge** accelerates learning by applying information from sources external to the agent.

Each of these methods have strengths and weaknesses when applied to MASs, as are outlined in Table 2.2. In the table, the method or category of method is analysed under the following headings:

- **MAS Agent Type** details what type of system the method can or has been applied to and what modifications are needed to do so. Single agent means it can not take advantage of multiple agents.
  - The **Homogeneous** column gives details on how it works in homogeneous systems.

  - The heading **Heterogeneous** provides details on how it works in heterogeneous systems.

- **Environment Type** is how the method can handle the respective environmental characteristics.
  - **Non-Stationary** is how the method manages change over time.
  - **Dynamic** explains how a method copes with dynamicity in the outcomes of actions.
- **On-Line** is whether the method does most of its calculation and improvement on-line or off-line.

Three of the main methods for Additional Knowledge are compared as well as Generalisation and Algorithm Alteration.

	MAS Agent Type		**Environment Type**		
Method	**Homogeneous**	**Heterogeneous**	**Non-Stationary**	**Dynamic**	**On-Line**
Algorithm Alteration	Single agent	Single agent	Unsupported	Unsupported	✗
Generalisation	Single agent	Single agent	Can lose accuracy	Can lose accuracy	✗
Additional Knowledge					
— Reward Shaping	✓	With mapping	Can track change	Less effective	(✓)
— Selection Biassing	✓	With mapping	Can track change	Less effective	(✓)
— Transfer Learning	✓	With mapping	Unsupported	Can address	✗

Table 2.2 Comparison of Methods for Accelerating Reinforcement Learning.

Algorithm Alteration and Generalisation both work in MASs as they would in single agent systems. They take no advantage of the presence and relatedness of additional agents in a system. This prevents them from *efficiently using knowledge* in MASs Additional Knowledge approaches can, however, exploit the repetition in learning in a MAS.

When faced with non-stationary environments, Algorithm Alteration and TL can only perform as well as the underlying RL algorithms does (called Unsupported in the Table 2.2). RL does not cope with non-stationarity or dynamism well; if either happen during the agent's exploration then what is learnt can partially account for them, otherwise they affect performance. This lack of *adaptability* limits usefulness in dynamic or non-stationary environments. Algorithm Alteration requires that its model accounts for the environmental change, which it generally will not.

TL's additional knowledge has already been supplied, so nothing can be done to track change in the environment. If the shift in the environment is significant enough to invalidate the representation used by Generalisation, then learning will become inaccurate. In these approaches most of the acceleration is done off-line, which prevents the acceleration scheme *adapt* to any *on-line* changes. The remaining Additional Knowledge approaches can address non-stationarity and on-line change, if the source of knowledge used remains applicable and changes to accommodate the new environment as knowledge is supplied consistently (the change does need to be detected and reacted to however).

None of the methods address dynamic environments better than RL alone does. Generalisation's representation can become less applicable if the dynamism is significant. Reward Shaping and Selection Biassing, through their effect on exploration, can prevent dynamism from being fully represented, as a dynamic state can be left insufficiently sampled. TL can address dynamism if the supplied knowledge provides a correct representation of the value for a state that is dynamic.

Only Shaping and Biassing are capable of any calculation *on-line* and this is usually only to apply a statically, off-line calculated signal at a particular time. To truly operate on-line and react to non-stationarity or dynamism, the source of information must also be selected for a particular circumstance. When applying additional knowledge to make *efficient use of information* in a MAS, *heterogeneity* needs to be accounted for as agents will always experience some degree of difference from each other.

In MASs—particularly those at large-scales—learning is slow, affecting performance, and the environments will be complex. If RL is to be used, both of these facts necessitate that some sort of acceleration scheme be used. If not, the performance of the system will suffer. The acceleration scheme will need to be generalisable and should be self-configuring, as it is impractical to have designers adjust the scheme for each change in an agent or the environment; the acceleration scheme should be *adaptable.* Schemes need to be practical for *on-line* learning to further enable this adaptability.

Algorithm Alteration generally requires greater knowledge of the environment (in the form of a model) to improve performance. The superior knowledge is used to ensure samples of the environment are used to their fullest potential. Having a model requires the environment to be modelled and modelable, which typically introduces a designer or extra computation to learn a model. This makes Algorithm Alteration a poor acceleration scheme in MASs. Similarly for Generalisation, a good approximation scheme must be designed or discovered through significant effort.

Human knowledge can be used, but considerable effort is required to identify how it should be applied [Knox and Stone, 2009]. Also, consulting a human after each change in an environment is impractical. Getting additional information from agents in other tasks is much more practical, as the information is more readily available and more likely to be closely related. For use in on-line learning, the informational flow will need to change, as running separate source tasks for type of *heterogeneous* agent prior to the target's execution is time consuming and requires that designers select relevant tasks for arbitrarily many agent types. Agents may also join the systems while it is running, so would not be present at the start, preventing designers selecting source tasks for them.

From this study the open requirements for the acceleration of learning in MASs are as follows:

(A) **Efficient Use of Knowledge** is important as learning in real-world systems is expensive. Acquiring knowledge impacts on performance and therefore should be minimised.

(B) **On-Line Improvement** is necessary as the operating environment can change over time experiencing fluctuations in behaviour and performance must be maintained. The longer it takes to adapt to changing circumstance, the worse overall performance will be.

(C) **Adaptiveness** is important as the way a system will change can not be anticipated, the acceleration scheme must account for this.

(D) **Heterogeneity Support** is a necessary condition as in large-scale systems there will always be some variation in agents.

Table 2.3 details how related work addresses these requirements. Algorithm Alteration can improve the efficiency with which knowledge is used (A). It does not however, offer on-line improvement in the face of change (B), nor does adapt to changes in the agent (C). It could be argued that it supports heterogeneity (D) by ignoring it. As it is necessarily a single agent process, the presence of heterogeneous agents has no impact, but it also provides no advantage. Generalisation addresses the requirements similarly. The Additional Knowledge approaches can all support heterogeneity (D) through the use of mappings from source to target. They differ in regards to the On-line improvement (B) requirement, which can not be done by TL. All of its improvement is at the start so it can not react to changing circumstances. It can however, make efficient use of knowledge (A), which other methods can not as they are limited in how the knowledge can be supplied.

Method	**Efficient Use of Knowledge**	**On-Line Improvement**	**Adaptiveness**	**Heterogeneity Support**
Algorithm Alteration	✓	✗	✗	(✓)
Generalisation	✓	✗	✗	(✓)
Additional Knowledge				
— Reward Shaping	✗	✓	✗	✓
— Selection Biassing	✗	✓	✗	✓
— Transfer Learning	✓	✗	✗	✓

Table 2.3 Related Work's Addressing of Requirements.

Chapter 3

Parallel Transfer Learning

The trouble with having an open mind, of course, is that people will insist on coming along and trying to put things in it.

Terry Pratchett

The previous chapters provided motivation for and an introduction to accelerating learning in Multi-Agent Systems (MASs). This chapter presents a novel algorithm to address the short comings of current work in accelerating learning in MASs that were identified previously. This thesis's contribution, Parallel Transfer Learning (PTL), accelerates learning in MASs by allowing agents to share knowledge and support each other's learning processes, thereby improving overall system performance.

3.1 Introduction

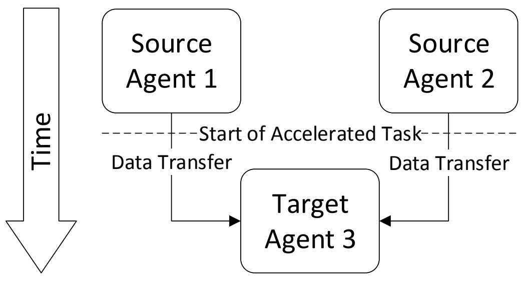
To accelerate learning in MASs a scheme will need the following characteristics to satisfy the requirements identified in the previous section: (A) Efficient Use of Knowledge, (B) On-Line Improvement, (C) Adaptiveness and (D) Heterogeneity Support. It must also address the main problems that impact on the rate of learning identified in Chapter 2: (I) Sample Variation, (II) Sparsely Visited States and (III) Credit Assignment. As also discussed in Chapter 2, no method currently has all of these characteristics. In homogeneous MASs, agents often learn similar or identical things. In heterogeneous MASs there is also potential for transfer, as they may have useful information to share. This makes additional knowledge-based approaches a logical way of accelerating the learning process. The first question is which scheme should be used? Selection biassing will have no effect on sparsely visited states, as an agent still has to be able to visit a state to use the knowledge. Biassing is a passive method of applying additional knowledge, as the agent still gains its own information, just receiving suggestions of what knowledge to seek. Reward Shaping is more of an active method as it does affect the value function, but the agent still needs to visit a state for the effect to happen. This leaves Transfer Learning (TL) as the best approach to use in MASs.

TL comes closest to meeting the requirements for acceleration in MASs, supporting heterogeneity (D) and efficiently using knowledge (A). Additionally, it is scalable; a single source of data can accelerate learning in potentially many targets and—aside from producing a mapping—it involves little calculation. Complex environments do not matter to TL, as the source task is agnostic of the target task's (the process operating in a complex environment) problem, learning is supported regardless. This makes it potentially adaptive (C), if it can be allowed to adjust the source knowledge used on-line. In MASs, all agents learn concurrently, this means no source information is available prior to the system's start. To leverage the relatedness of tasks in a MAS, an algorithm must operate on-line which TL does not.

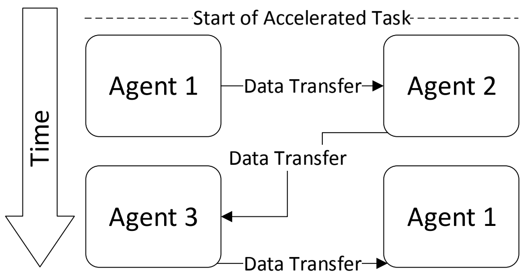
This chapter presents PTL, an algorithm that meets all of the requirements. First an overview of PTL is presented. After this the components of PTL will be introduced. The chapter closes with a discussion of how PTL addresses the problems with learning in Reinforcement Learning (RL) and how it meets the requirements for accelerating learning in MASs.

3.2 System Overview

Using TL in a MAS means there must be some good source information prior to the MAS's execution. This is a difficult requirement to satisfy, as running a source task beforehand for each agent or class of agents is time consuming and not particularly scalable. It also fails to take advantage of the relatedness of agents in a MAS. Running part of a MAS before the rest would allow relatedness to be leveraged, but it is not possible in many applications (e.g., in the Smart Grid (SG), agents effect a shared resource, running them separately prevents these effect being learnt) and impractical in others (e.g., in traffic control traffic lights at junctions could learn independently, but this would prevent them coordinating their behaviour which would reduce performance). So, TL needs to be moved on-line without impacting on its other capabilities. PTL allows this to happen (see Figure 3.1). Figure 3.1a shows the temporal dependency between source and target tasks. The target tasks execute, then the source task. Figure 3.1b shows possible transfers in PTL, where the strict ordering of source then target is removed, and any agent is capable of transferring to any other.



(a) Information Flow in TL.



(b) Information Flow in PTL.

Figure 3.1 Off-line Requirements of Transfer Learning.

PTL allows source and target tasks to learn simultaneously and transfer knowledge as and when they see fit to do so. As there is no knowledge at the beginning of the system, there is no initial boost in performance as there is with TL, but once this is overcome a sort of learning momentum is built. Moving TL on-line necessitates multiple transfers of information. This better captures dynamic changes and non-stationarity found in MASs than with a single transfer of information. This is particularly true with information about the inter-relatedness of objectives, which—even in the most simple of environments—can be highly variable.

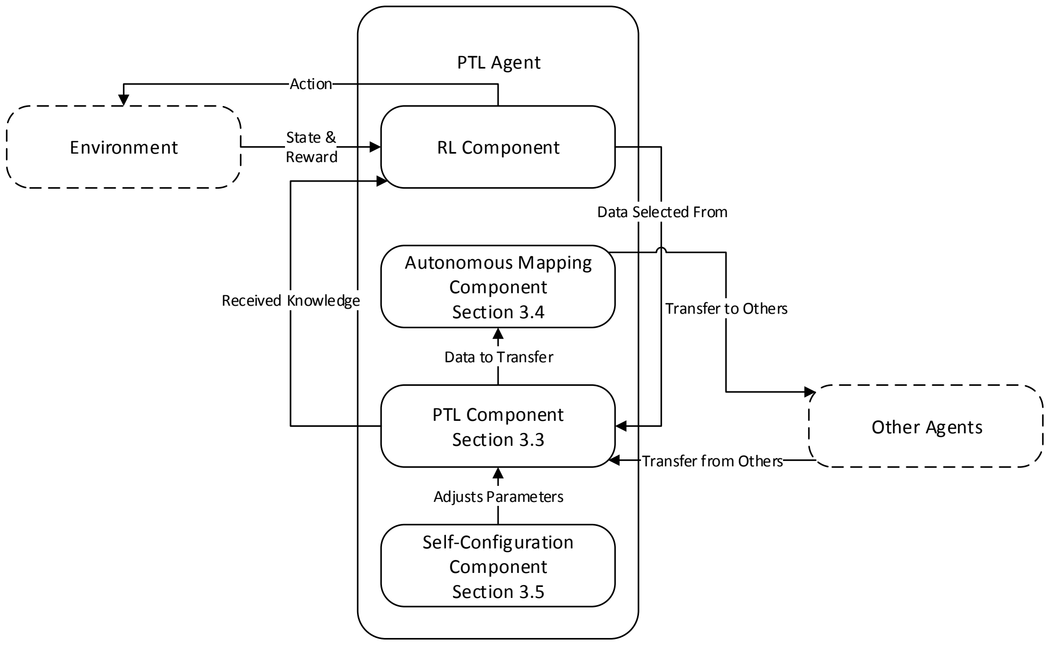


Figure 3.2 Parallel Transfer Learning Architecture Description Sections.

The PTL algorithm consists of three components. (1) The **PTL Component**, which is responsible for transferring knowledge between agents. It allows TL to happen on-line so that it can adapt to changes in the environment. It addresses the requirements for efficiency of knowledge use (A) and on-line improvement (B), and is presented in Section 3.3. (2) The **Autonomous Mapping Component** allows agents to learn how to translate knowledge on-line, addressing the heterogeneity support (D) requirement, and it is discussed in Section 3.4. (3) The **Self-Configuration Component** reconfigures the other components so that they can best operate in a particular environment. This allows PTL to meet the adaptiveness requirement (C), and its design is in Section 3.5.

Figure 3.2 shows how the components fit together and where they are discussed. The three components have numbered sections. The unnumbered components show how PTL is integrated in to an RL agent.

3.3 Parallel Transfer Learning

This section will first present a general description of the algorithm and then a more formal definition. Design considerations will then be discussed.

3.3.1 Algorithm Overview

The main problem with PTL is uncertainty about knowledge. An agent that is either sending or receiving knowledge can not be certain of its accuracy, so some caution must be taken. For this reason, source-driven transfer is considered better than target-driven transfer. Furthermore, rather than an agent asking others if they know anything about a given state as would be done in target-driven approach, agents transfer information when they feel it is representative. If target-driven approaches were used, it would be difficult to know when receiving a request for knowledge if the state is worth transferring, as it takes time to decide if information is ready to be transferred. In source-driven approaches, an agent can monitor a state over several samples and evaluate when that state is ready to be transferred. Having the source drive the process also removes the risk of a transfer being received too late to be useful, by removing the inter-dependence introduced by request-response interactions, PTL is effectively made asynchronous. In target-driven approaches, the target must determine it needs information, request it, the source must then respond and finally the target must process the response. In MASs, the agent can rarely look sufficiently far ahead for this process to be guaranteed to be completed before the information is needed. In source-driven approaches, the agent processes transfers received and applies them without the just-in-time constraint required by target-driven approaches. PTL generally happens after the ‘normal’ RL update, as new information is available to be transferred and there is time to receive messages from other agents and process them. Transferring information at this point in an agent’s operation ensures that new information will be available to transfer and any received information can support the next interaction with the environment.

From this point, the discussion of PTL will focus on underlying algorithms such as Q-Learning and W-Learning that have a distinct representation of knowledge for states and actions as well as objectives. There is no reason PTL can not be applied to other RL algorithms as transfer learning has [Taylor and Stone, 2009], but it is convenient for both discussion and implementation to separate the priorities of objectives from an individual action's value to a particular objective. Aside from the separation of objectives, most RL algorithms learn some form of discretised value function representation [Sutton and Barto, 1998], so little generality is lost for this clarifying assumption. The main exception to this is algorithms that use a functional representation. TL has been used with functional approximated RL [Taylor and Stone, 2009], so it is reasonable to assume PTL could be made to work as well.

When attempting to transfer information in PTL, an agent (or its designer) must answer several questions about the information it will send:

- **When to transfer?**
- **What to transfer?**
- **How to receive knowledge?**
- **To whom to transfer?**

These questions will be discussed in detail in following sections.

3.3.2 Definition of PTL

A MAS can be seen as a set of sets. As such, set notation is the easiest and clearest way to describe MASs and PTL. The following notation will be used to describe the necessary components for a system that implements RL and PTL. In general superscripts are the 'owner' of the particular set and subscripts indicate the index within a set. For example in a set of multiple packs of playing cards, $2^{\clubsuit}$ would be set of all the 2s of clubs and $2^{\clubsuit}_1$ would be the first 2 of clubs in that set. Uncapitalised names are an instance in the set and lower case names are specific instances of a type. In the playing card example, K would be the set of kings, k would be a specific king.

- A set of agents, $\{A \mid a_1, \ldots, a_Z\}$, where Z is the total number of agents in the system.
- A set of neighbours N for each agent $a \in A$, $\{N^a \mid n^a_1, \ldots, n^a_Y\}$ where Y is the number of agents the agent a can communicate with.
- A set of learning processes LP for every agent in the system $a \in A$, $\{LP^a \mid lp^a_1, \ldots, lp^a_X\}$, where X is the number of objectives implemented by the agent a.
- Agents have a parameter Transfer Size (TS), which determines how many states are shared per transfer for each learning process in LP^a.
- Agents use a Selection Method (SM) to determine which data to share for each learning process in LP^a.
- Merge Methods (MM) are used to add received information to an agent's own state-space for each learning process in LP^a.
- A set of mappings for each agent's learning processes in LP^a for $a \in A$ to each neighbour's learning processes in LP^n for each neighbour in N^a, $\{\chi^a \mid \chi^{a,1}_{1,1}, \ldots, \chi^{a,1}_{1,X}, \ldots, \chi^{a,X}_{Y,1}, \ldots, \chi^{a,X}_{Y,X}\}$. Where $\chi^{a,1}_{2,3}$ is the mapping from agent a's first policy to agent a's second neighbour's third policy.
- A set of transfers for each agent in the system T^a $a \in A$. T^a is composed of each learning process in LP^a's transfers to each neighbour's learning process in LP^n for each neighbour in N^a, $\{T^{LP^a} \mid T^{LP^a}_{1,1}, \ldots, T^{LP^a}_{1,X}, \ldots, T^{LP^a}_{Y,1}, \ldots, T^{LP^a}_{Y,X}\}$ for each time step.

An overview of PTL is provided in Algorithm 1, design considerations for addressing the questions identified above are discussed in sections following. As it is designed to operate between agents in a distributed system, there is no need for synchronisation between agents. This means that the ordering of operations needs to be flexible. Messages need to be sent and forgotten about; this allows agents to progress through the algorithm at whatever speed they are capable of. As a result, the functions operate in loose decoupled pairs. SelectKnowledge(TS,SM) chooses which knowledge to send and Merge($Q_{received}(S, A)$,MM) incorporates it in the state-space. The pair of functions SendKnowledge(*output*,*n*) and ReceiveKnowledge perform simple transmission and reception of the messages produced by PTL.

Algorithm 1 Parallel Transfer Learning

```
 1: After RL Update
 2: for all Agents $n \in N^a$ do
 3:     $output \leftarrow$ SelectKnowledge($TS$, $SM$)        ▷ Choose knowledge to send
 4:     $output \leftarrow \chi_{A_a \to n}(output)$          ▷ Translate knowledge
 5:     SendKnowledge($output$, $n$)
 6: end for
 7: $input \leftarrow$ReceiveKnowledge
 8: for all $Q_{received}(S, A) \in input$ do
 9:     Merge($Q_{received}(S, A)$, $MM$)         ▷ Incorporate new knowledge
10: end for
```

3.3.3 Data Selection

When an agent is selecting information to transfer, it must consider what information will be most useful to the target agent it is transferring to and when it will need that information. How frequently to transfer and what to transfer are closely related as one affects the other, and both are controlled by the selection method SM (see Section 3.3.3.1 and 3.3.3.2 for examples). The questions of what to transfer and when will be discussed separately to first present the issues involved in each, then selection methods will be presented to explain how what and when are combined into a single method.

3.3.3.1 When to Transfer

There is a spectrum of approaches to decide when—or how frequently—to transfer. The extremes of this spectrum are the two approaches that follow. Information can be transferred every time it changes. The advantage of this is that there is little chance of a target task not having information when it has to make a decision about the state to which the information pertains. This method requires more communications and there is a chance that what is being transferred may not be representative of the 'true' value of a state if it has been only sparsely sampled. The alternate method is to only transfer information to the target when a value appears to have converged at the source (i.e., that the value changes more slowly than it previously did or not at all). This approach will entail less message passing and has less risk of transferring inaccurate information, but it may not provide data in a timely manner. If information is transferred too infrequently, there is an opportunity cost at the target agent; the performance in a state could have been improved, but was not due to an over-cautious source.

No single method for how frequently to transfer will be suitable for all states all the time. The optimal frequency of transfer will depend on the frequency at which the state's value changes—either from the agent visiting it and learning something or receiving and merging a transfer—and how the environment changes inherently. It will also depend on how far into the learning process the agents are. At the beginning, information will be less reliable, but as agents are encouraged to explore early on, any information transferred will likely not be used to guide action selection (random exploration will instead), so there is less risk of reducing performance in the target agent. This means an initially high frequency of transfer can be used, which maximises the amount of information shared. As information becomes more reliable, fewer states need to be transferred, because the agent will likely have received them already or received them in some recent form which would be representative e.g., at the previous visit. This means the frequency can decrease over time, before tapering off when there is nothing left to learn. If the environment is non-stationary, then ideally the rate of transfer would correspond to the changing environment, which would allow any changes in the environment to be tracked and reacted to. With conventional TL, acceleration occurs as soon as the system begins, effectively there is learnt source information at the very beginning. In PTL, knowledge must be learnt before it can be transferred. This means that PTL loses the initial performance improvement that TL can provide, until sufficient knowledge is built up to allow transfer to become effective. This means that early transfers will not be very effective.

While what to transfer and when are closely related and impact on each other, the following are methods that can be categorised as selecting based on when to transfer rather than what:

- **Most Recently Visited** Transfer of the most recently visited states shares the freshest knowledge which is unlikely to have already been transferred. It will tend not to pass on received information, as received information is only forwarded when the state is visited. The selection method criterion SM is $\{T \mid Q(S_t, A_t), \ldots, Q(S_{t-TS}, A_{t-TS})\}$.
- **Converged States** Sharing the most converged states will provide the highest confidence information, but it will tend to select the same states repeatedly, again there is little flow of received information. The selection method criterion SM is $\{T \mid \min \|Q(S_{it}, A_{jt}) - Q(S_{it-1}, A_{jt-1})\|\}$ where t and $t-1$ are instances when the state was visited by this agent.
- **Visit Threshold** can be used on its own or in conjunction with other methods. The visits to a state are counted and once a set limit is reached it becomes eligible to be transferred. This means each transfer has a greater degree of confidence associated with it. The selection method criterion SM is $\{T \mid threshold \leq VisitCount(S_i, A_j)\}$.

All selection methods have the additional property that $|T| = TS$, that being that the cardinality of the set to be transferred is the correct size containing the right number of state-action pairs. Selection methods with indices i or j iterate through the state-space (without replacement) until the set T is full. Further methods are presented in the next section. Selection methods that chose data based on time will tend to perform better in environments that change over time as they focus on confidence in a value and timeliness of transfer rather than which states are important.

3.3.3.2 What to Transfer

Related to determining the optimal frequency of transfer is selecting what data to transfer. There are two different sources of data: locally learnt knowledge and transferred knowledge. Both must be shared, as not all agents will be neighbours of each other and knowledge should be fully propagated though the system. Fully propagated knowledge gives each agent the opportunity to learn from all others, even though there may not necessarily be any utility in this. There are two types of information to share: values that correspond to how an action affects one objective (e.g., value function information) and information about how objectives relate to one another (inter-objective information). A system must also consider what form the transferred information should take; it can be high level information like policies or lower level such as state-action values. Across the system there should be variation in what is transferred. If all agents share knowledge on the same subset of states then there is no improvement in learning rate in the other states. This can be particularly problematic if there is little variation in where the agents are in their respective state-spaces. This gives rise to the concept of diversity in data selection. The data transferred to a particular agent should provide the maximum amount of knowledge. In general, this means different states should be sent at each transfer—a particular transfer should be as different as possible from any other—but this is not always possible because there may not be sufficient new, good information available. The data selected should be a best estimate of the diversity criterion with respect to the quality of information shared.

3.3.3.2.1 Value Function Information

The set of states-action pairs that constitute a particular agent's transfer at the time t can be written as $\{T_t^a \mid Q_1^{SM}(S,A),\ldots,Q_{TS}^{SM}(S,A)\}$, where SM is the selection method used, $Q(S,A)$ is the value of the action A in state S and TS is the number of values per transfer. For clarity, it is assumed that a only has one policy, if there are more, then the set T_t^a is composed of the number of local learning processes X sets such that $\{T_t^a \mid T_t^{LP^1},\ldots,T_t^{LP^X}\}$. $T_t^{a\prime}$ is the same set mapped to the respective target's representation via $\chi^{a,l}_{n,m}$. This is the mapping from the agent a's l^{th} policy to a's n^{th} neighbour's m^{th} policy (discussed further in Section 3.4). The set of transfers T^a_t can be subdivided in subsets for each particular target in the set by rewriting the construction of the set as $\{T^a_t \mid T_t^{a \to N^a_1}, \ldots, T_t^{a \to N^a_Y}\}$. Regardless of how the set of transfers T^a_t is constructed, its constituent subsets can be empty sets. Not every potential transfer link needs to be used at every time-step. An agent can choose not to provide information to one or more of its neighbours. The main reason this may be done is if there is no good information available and the agent knows this. As early in their learning process the quality of what has been learnt is expected to improve, there is no reason not to transfer during initial learning. Not transferring will reasonably only occur if the environment has changed significantly enough to invalidate all of an agents knowledge or if learning has finished and all knowledge has been shared.

The preferable selection method for a particular system will depend on the nature and dynamics of that system. Selecting states by a particular property often dictates the frequency at which they can be transferred. Some example schemes are as follows, ordered based on the (estimated) diversity of the data they select (from lowest to highest):

- **Most Visited States** will also tend to share the same states repeatedly (assuming the agent gravitates to areas of higher reward), but it will also provide frequent estimates of the important (actually used) states. The selection method criterion SM is $\{T \mid \max VisitCount(S_i, A_j)\}$.
- **Best/Worst States** shares the states that are thought to be most important based on their value. It will pass on transferred information if it is important, but after a time the selection of states will stabilise to a fixed set. The selection method criterion SM for Best is $\{T \mid \max Q(S_i, A_j)\}$ and for Worst $\{T \mid \min Q(S_i, A_j)\}$.
- **Pass on Received** also can be used on its own or in conjunction with other methods. An agent forwards on any transfers it receives to other agents. The diversity this brings depends on the originating agent's selection.

- **Greatest Change** selects the state that has exhibited the greatest change between t and $t-1$. This is state at which the most significant learning has occurred. It will indiscriminately transfer received or local information. It can, however, neglect states with 'true' values closer to 0 as they appear to have changed less than higher reward states. The selection method criterion SM is $\{T \mid \max \|Q(S_{it}, A_{jt}) - Q(S_{it-1}, A_{jt-1})\|\}$.

- **Random States** will select data that is local or received but it takes no account of whether the state being transferred is useful or not. The selection method criterion SM is $\{T \mid Random(S_i, A_j)\}$.

All selection methods have the additional property that $|T| = TS$, that being that the cardinality of the set to be transferred is the correct size containing the right number of state-action pairs. Selection methods with indices i or j iterate through the state-space (without replacement) until the set T is full. Selection methods can either be set by a designer or automatically configured (see Section 3.5). Sparsely visited states are less of a problem in static environments, so the visit threshold preventing single instances of these states getting shared is less of an issue. In constantly changing environments, transferring a number of the best states is effective as transfer aims to maintain performance in important areas of the state-space in spite of the changes. In sharply changing environments, best states is also effective, but with smaller transfer size TS. The smaller transfer size makes the transfer more conservative, which is important as the value of state-action pairs may have been invalidated by the change. The value of these state-action pairs is then wrong for the changed environment and should not be shared. In general the greater the diversity a scheme maintains, the less confidence there is in the value of a state and the more likely a delay between experience and transmission is. The selection method (SM) is in the Algorithm 1 at Line 3. Selecting by Greatest Change is shown as an example in Algorithm 2.

Algorithm 2 Selecting Knowledge by Greatest Change

```
1: $output[TS]$                                        ▷ Storage Space
2: for all State-Action Pairs do
3:     $toTest \leftarrow$ GetChange
4:     if IsFull(output) then
5:         if $toTest \geq output[TS-1]$ then
6:             $output[TS-1] \leftarrow toTest$
7:             SortHighToLow(output)
8:         end if
9:     else
10:        $output \leftarrow toTest$
11:        SortHighToLow(output)
12:    end if
13: end for
14: return ouput
```

3.3.3.2.2 Non-Value Function Information

TL is not limited to transferring only value function information, nor is PTL. In Arbitration-based, multi-objective systems there will always be some way of representing the interrelatedness of objectives at each agent. This information can also be transferred. Transferring the interrelatedness information can further accelerate learning as it affects a different learning problem, the inter-objective learning problem. When learning in Arbitration-based, multi-objective systems, first an agent must learn what is best for each objective, then it learns which objective should be obeyed at which time. This second phase of learning can be targeted by the transfer of interrelatedness information. Transfer of interrelatedness information requires the agents be much less heterogeneous than ‘normal’ transfer, as inter-objective priorities are highly dependent on the particular set of objectives [Taylor et al., 2014].

The requirements for this type of transfer are somewhat different to ‘normal’ transfer. Typically, the relationships between objectives change rapidly as they are affected not only by the environment, but also by how frequently they are obeyed (see the update rule of W-Learning in Section 2.4). This makes the interrelatedness values more variable than Q-Values, which makes the effects of a particular transfer transient. Changing one objective's value at one state does not have the lasting effect that a change to Q-Values would. This makes the selection of interrelatedness transfer less involved. An agent can simply transfer the states at the extremes of the value spectrum (e.g., the current highest). If the agents are also collaborating through a method other than PTL (collaborative RL for example), then the priorities transfer can be the inverse of an agent's own. This will encourage action in states that the source agent does not care about, as it suggests the target give them high priority. It discourages action in states where the source has high priority by recommending a low priority. Other methods of collaboration particularly encourage collaboration at states in which the agent is not locally penalised [Jennings, 1993]. The merging of such information can be based on the target agent's priorities; do not merge if the state is locally important, otherwise do.

When transferring and merging interrelatedness information there are situations where it is important to maintain diversity in the values. If the MAS's problem is balancing use of a shared resource, then too similar interrelatedness values will cause agents to act the same way and affect the previously learnt balance. This effect is mitigated if the agents do not share the same current state, as variability is reintroduced through the state's variability. In the case of such scenarios, merging must be more cautious than normal or the interrelatedness transfer size kept small to prevent homogeneity.

3.3.4 How to Receive a Transfer

Upon receiving a transfer, an agent must decide if and how to incorporate this knowledge into its state-space. The simplest case is when the target agent knows nothing about a particular state, and accepts the knowledge as it has no better information. The more complex case is when the target has some knowledge of the state in question. In this case, a decision must be made as to which information to use. The receiving agent has the received information of indeterminate accuracy and its own information (which how it was gathered is known but not its accuracy with certainty). From this it must select from one source or combine them. The obvious way of reducing the ambiguity is to include metadata with each transfer. This could be how it was selected, visit count, original source, etc. The problem with metadata is that it is difficult to relate to confidence. If a state has been visited a given number of times by two agents independently, its value could vary significantly due to dynamism in the environment, different learning rates, different experience in successor states, etc. This makes metadata less reliable than one would expect. It also is very difficult to translate across heterogeneous tasks.

Alternatively, an agent can use its own knowledge as an estimator of the true value of a state. In simple environments, the value of a state tends to change in ever decreasing steps, which is less true in complex environments, but still effective. For example, an agent has a value of x at a state, having changed from $x - 2\delta$ on its previous update. It receives a transfer saying the value of that state is $x + \delta$. It can accept this as true since the received value is consistent with the progression of that state's value. In that example the transfer will have little value as it only made a small change. This is why using progression as a heuristic is better as an excluding criterion. If, instead, the transfer was $-x$, it would have been rejected.

If the transfer can not be fully accepted or rejected then it will have to be combined. A decaying linear combination allows agents to favour locally learnt information over received information once they have learnt. Equation 3.1 shows how this is done. Where $currentVisitCount$ is the number of samples of the state in question and $numberOfConfidenceVisits$ is an estimate of the number of experiences required for a good estimate without transfer, the agent discards transferred information after it has generated enough of its own experience. A nice feature of this approach is that the rate of merging does not need to be consistent across the state-space. In states where learning is slow—either because they are sparsely visited or particularly dynamic—the $numberOfConfidenceVisits$ can be adaptively increased allowing more additional knowledge to be used without impacting on 'normal' states.

$$\begin{aligned}
factor &= numberOfConfidenceVisits - currentVisitCount \\
scaledRecieved &= RecievedQ(S_t, A_t) * factor \\
scaledLocal &= Q(S_t, A_t) * currentVisitCount \\
Q(S_t, A_t) &= \frac{scaledRecieved + scaledLocal}{numberOfConfidenceVisits}
\end{aligned} \tag{3.1}$$

As the impact of transferred knowledge reduces over time the agent will prefer its own experiences instead of transfer. This allows transfer to support the learning without impacting on the final learnt value of a state.

Deciding how to merge knowledge of uncertain veracity is contingent on the sending agents' selection schemes. If the sending agents are cautious and only send well sampled information, then the merging process can be more permissive. The underlying environment will also impact upon the merging. If the environment—whether shared or not—is variable from agent to agent, then the value functions learnt will vary more, which in turn means merging should be more cautious. The way an agent merges does not need to be constant over time. In non-stationary environments after the environment changes, the agent may wish to be more permissive as it tries to adapt. With Equation 3.1 this could be accomplished by resetting visit counts. The merge method (MM) is in Algorithm 1 at Line 9.

If the agent has more information about its neighbours than just their transfer (for example, how long they have been learning, trustworthiness etc.), then it can add an extra step to merging information. This step uses the extra information to categorise neighbours. Merging decisions about knowledge from neighbours known to be reliable can be more permissive. The alternative is also useful, unreliable neighbours can be ignored. This additional step is represented in the activity diagram in Figure 3.3.

It is possible that an agent can receive a transfer from a neighbour or neighbours that is based solely on knowledge that it learnt locally and transferred. In this case there is an apparent risk of the value of a state diverging from its 'correct' expected value.

Consider the example of two homogeneous agents and a state that can have two possible successor states for an action. The first outcome occurs 90% of the time and has reward of 10, the remainder of the transitions yield a reward of -10. If Agent 1 experiences this state, receives reward of -10 and transfers this, Agent 2's value function will not represent the expected value of the state. Regardless of how many times the agents receive the less common reward without receiving the other and transfer to each other, their value functions will not have a value beyond the extreme of the reward, -10. If the more common reward is eventually received, as would be expected in this case, then that agent's value function will move towards the actual expectation and the significance of any received information will decay as that state has been visited more. This will reduce the impact of any future transfers moving it back to the reward's extreme, however it can still get there. As local information is always merged, the value of a state will move toward the 'true' expectation when using PTL as long as a state is sampled enough to trigger the heuristic preventing 'bad' information being merged. In short, the value of a state can be prevented from reaching its converged value by PTL in only the following situations:

- **Perverse Reward Sequences**, if an agent can receive non-representative reward several times in succession, this could be established RL's expectation and prevent the receiving of correct information. However, as long as the more representative value is experienced locally the agent's merging heuristic will be corrected and transfers can be applied correctly again.
- **Merge Amplification**, if one of the agents in a system is capable of merging a transfer by scaling it over 1. This would require an agent to implement PTL incorrectly or maliciously.
- **Incorrect Mapping**, if a state is not correctly mapped it can receive a transfer that is not representative. However, once the state is experienced locally, this incorrect value will begin to correct and the merging heuristic will prevent further transfers being applied.

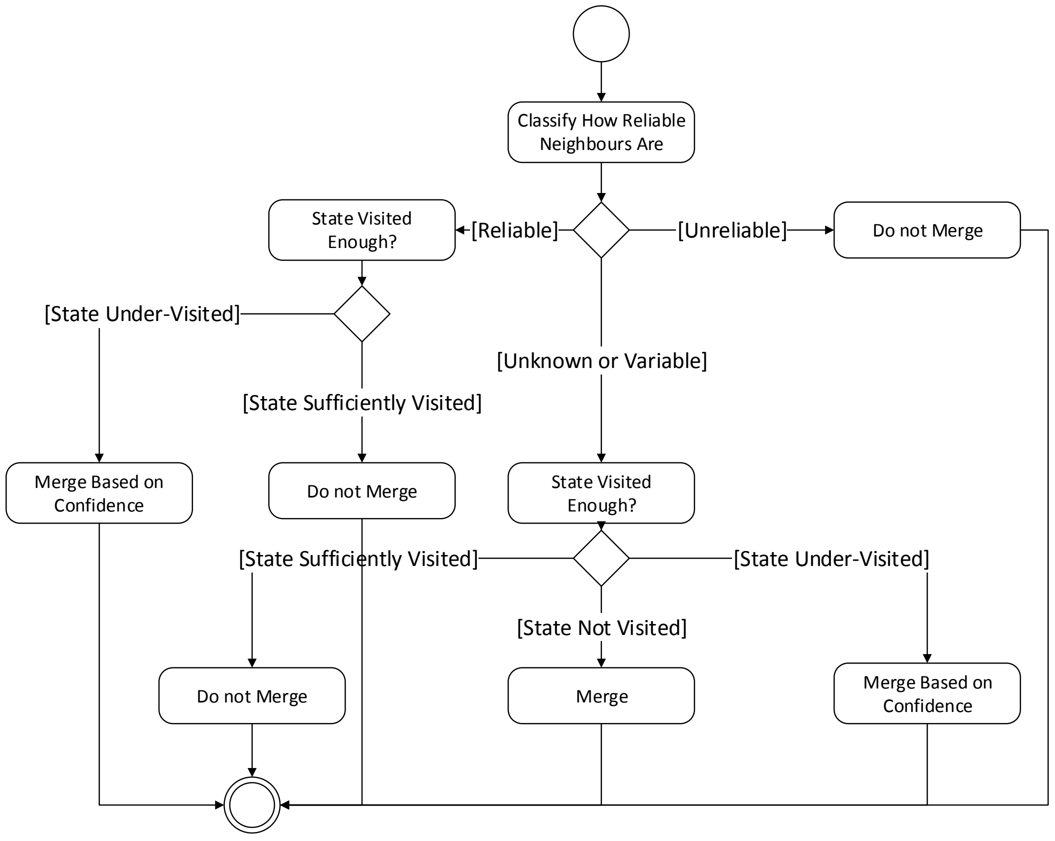


Figure 3.3 Activity Diagram of Reinforcement Learning, Parallel Transfer Learning and Self-Configuration.

3.3.5 To Whom to Transfer

In a system with multiple agents, an agent must choose to which agents to transfer. Ideally it would transfer to an agent that is exploring a different part of the state-space. In this case, it is likely that the target agent will have no knowledge in the area the source is exploring, thereby maximising the efficacy of the transfer and removing the need for complicated merging of knowledge. Determining which agents are exploring different parts of the state-space without sharing the entire state-space it is difficult, particularly so when heterogeneity is involved. Sharing the whole state-space is not scalable; it takes too many messages and too much processing to interpret. Instead some heuristic will have to be used to identify ‘good’ pairs of agents for transfer. In collaborative RL, agents will often share updates of their current state, and this update can be used to estimate where an agent is in its state-space (assuming heterogeneity). If it is in a similar area, it is likely to be a poor candidate for transfer and transfer should focus on other agents. However, if agents can join a system after it starts, then agents will have different ages. In that case, younger agents will benefit from transfers to important states as this will improve their performance more rapidly. Transfers received from an agent can also be used as an indicator of what states it knows about, and transfers can then be provided about other states. Finally, classes of agents can be used to generalise. If one agent of class A knows about a given state, then it can be assumed that all of class A have the potential to learn similar knowledge[1]. Younger agents can be targeted for transfer, as they can be expected to have incomplete information about their state-space.

Whatever agent selection scheme is used, the agents that are transferred to should vary to ensure that knowledge is well propagated and to maximise the potential for sharing useful information.

3.3.6 Summary

This section has presented the details of PTL, which is an algorithm that allows knowledge to be reused on-line. On-line knowledge reuse means that different parts of a MAS can support each other’s learning. This section also describes the design decisions that need to be taken.

The problems that affect the learning rate in RL are addressed by PTL through the sharing of knowledge. The sample variation problem (I) is improved by the multiple agents in a MAS effectively getting more samples of a state by having access to others’ experiences. For sparsely visited states (II) each experience any agent in the MAS has, can be provided to others in the system before they are likely to visit that state them-

[1]Complex environments can prevent this assumption holding, but it can be effective as a guiding heuristic, if it does not hold then the MM will correct it.

selves. This allows performance in these states to be improved. The credit assignment problem (III) also benefits from the sharing of knowledge.

PTL addresses the requirements for efficiency of knowledge use (A) by sharing information between agents in a MAS and on-line improvement (B) requirement as agents are constantly being supplied with new knowledge and can, therefore, adapt to fluctuations in the environment. As it has been presented here, PTL can only operate between agents that have a common representation of data. The source task sending information makes no attempt to map this knowledge to the target's representation. This means that only agents with representational homogeneity can transfer information. Enforcing this in MASs is impractical. This is compounded by the fact that representational homogeneity does not imply actual homogeneity. Two identical agents can learn different things if their environments differ or if they impact on each other. This means the requirement for supporting heterogeneity (D) is not met. This would considerably limit PTL's applicability in real-world systems. To allow heterogeneity to be supported agents must be capable of mapping information to their neighbours' representations. As discussed in Chapter 2, Inter-Task Mappings (ITMs) have always been calculated off-line, which is impractical with PTL for several reasons. In real-world systems, the combinations of agents in a system can not always be known a priori, so mappings can not be calculated for every combination of agents that may exist. Calculating mappings off-line before a system begins reduces the benefits of moving TL on-line, as considerable effort is needed before a system starts. The producing of mappings often requires that the knowledge is available when the mapping is being calculated, in a MAS the knowledge is not available until it is run. This means that the relatedness of tasks in a MAS can not be exploited when using off-line calculated mappings.

3.4 Autonomous Mapping

This section introduces on-line learnt ITMs. As PTL operates on-line, it is not practical to have ITMs provided off-line. The combinations of agents can not be known prior to the execution of a system. This necessitates the production of ITMs on-line. Learning ITMs is necessary for applying PTL between heterogeneous agents.

3.4.1 Learnt Mappings

Transferring information from one agent to another relies on the fact that knowledge can be made mutuality intelligible. If the agents are homogeneous, then no extra processing of knowledge is needed. If they are in some way non-homogeneous, then knowledge must be translated so that it can be understood by the target. The form of this translation depends on the difference between the agents in question. The knowledge is translated by a function called an ITM. $\chi_{source \rightarrow target}$ denotes the ITM from source to target. Generally the more different the agents, the greater the risk of negative transfer. This can be compounded by a poor ITM. If there is good, useful knowledge to share it must be transferred to the correct state to provide any benefit at the target agent. As discussed in Chapter 2, the production of ITMs has typically been done by designers or using computationally intensive methods, both of which are not scalable, and are executed off-line. To allow these mappings to be used with PTL, they must be calculated on-line. ITMs can be used to transfer any type of information (value function, model, policies, etc.), but PTL focuses on value function information and so will this section and by extension the ITMs produced.

As a single ITM is a complicated function that will be produced on-line, some clarification of terminology is required.

- **ITM** is the entire mapping. It can translate any value at the source to the target or deliberately stop translation (if a state has no match). In Figure 3.4, it is the set of dashed red lines.
- **Strand** is a single state-action pair's mapping to the target. An ITM is composed of many strands. In Figure 3.4 the dashed red lines are strands in an ITM, the black strands are potential ones but currently unused.

- **Feedback Tuple** a piece of feedback for a particular strand of the mapping describing if it was found to be effective by the target.
- **Effective** is defined as when a strand is properly matched and allows useful information to be exchanged.
- A **Correspondence** is used to mean when there is potential for an effective strand between two state-action pairs.

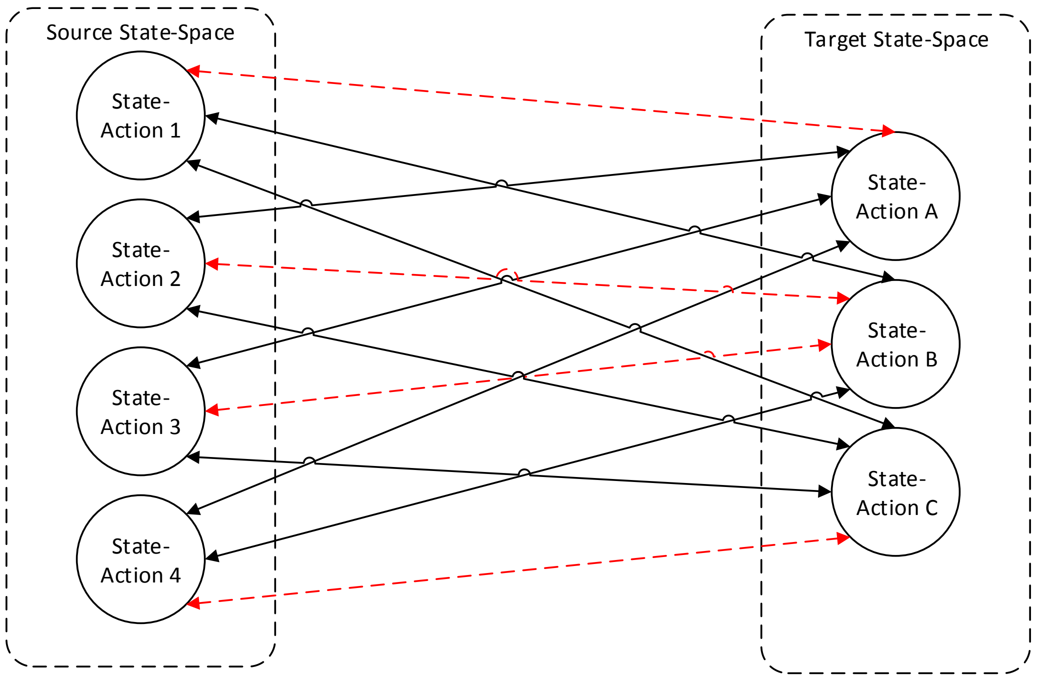


Figure 3.4 Inter-Task Mapping Terminology.

Figure 3.4 shows an example ITM (in dashed red) between two tasks. It also shows all potential strands. In this small example, the two tasks have state-spaces of 4 and 3 state-action pairs, which leads to 12 potential strands of which 4 are selected by the source task as it perceived there to be correspondences in the selected pairs. Between any two tasks there will be $|state\text{-}space_{source}| * |state\text{-}space_{target}|$ potential strands. In this example, the size of the source's state-space and the ITM are the same. This is not always the case, as multiple strands can lead to or from state-action pairs. Not all state-action pairs will have correspondences, so all strands to or from them will not be used, in other words they can be deliberately disconnected by the mapping.

As many MASs are distributed, it is impractical for the source and the target to both maintain a copy of the ITM. The constituent set of strands will change regularly—particularly early in the learning process—and maintaining multiple consistent versions at sources and targets is needlessly complex. The target never needs to know from which source state a particular transfer came. In addition, PTL is source-driven, and so it follows that the mapping should be as well. The source selects the data to transfer, so by having the source maintain the mapping, pre-translated information can be sent in transfers. The ability to translate information presupposes that the source agent has knowledge of the target agent's representation. This assumption is necessary as it is impossible to share information usefully without a common understanding. It is plausible as if the agents aim to improve each others' rate of learning they are necessarily cooperative. This cooperativeness provides a willingness to share the state-space. From a communication point of view, it is tractable as state-spaces are typically of the order of hundreds of state-action pairs. Each state-action pair is of the order of tens of bytes, so the entire state-space could reasonably be assumed to be under a MebiByte, which is plausibly communicable.

3.4.2 On-Line Mapping

When two agents are attempting to build an ITM without a common understanding of the environment in an on-line manner, there will not always be sufficient time to fully calculate an effective ITM prior to it being used, as current methods take considerable time to produce ITMs. So, an ITM must be built quickly and improved over time. The naïve approach is to produce an ITM that maps state-action pairs randomly (or some other starting point such as name-based similarities) from source to target. This random mapping must then be adjusted. Two methods to adjust the mappings on-line are proposed: Vote-Based Mapping and Ant-Based Mapping. The methods presented here were developed as they can be produced with comparatively little overhead. Only message passing is required, agents do not need to produce models or common subspaces as has been done previously. This reduces the communications and computational requirements. They also provide the flexibility to change over time, if either the source or target experience change.

3.4.2.1 Vote-Based Mapping

From the starting point of a randomly populated ITM between a source and target, agents need to improve the ITM as it is almost certainly inaccurate. One way of doing so is to have the target provide feedback to the source. Upon receiving a transfer from a source, the target makes its merging decision using a merge method MM (discussed in Section 3.3). The target agent then produces a feedback tuple based on this decision. The outcomes are binary; to merge or not to merge. This is encoded as ± 1. If the target agent chose not to merge a particular transfer, then it will send back negative feedback to the source. Alternatively, it can reinforce that particular strand of the ITM by providing positive feedback. When the source agent receives feedback, it accumulates it for the strand in question. If the strand's value falls below a threshold (its initial value works well), then it is deemed not effective and remapped to some other state. If there are states in the target without strands leading to them, then these are preferred. For example, if the target agent receives a transfer to some state-action pair $Q_{target}(S, A)$ and it decides not to merge this transfer (i.e., the received information does not agree with local information), the target agent will send the feedback tuple $[Q_{target}(S, A), -1]$ indicating that particular strand of the mapping that lead to $Q_{target}(S, A)$ is inaccurate and should be changed. Alternatively, if the transfer was merged this strand can be reinforced with the feedback tuple $[Q_{target}(S, A), 1]$. If a perfect mapping[2] exists, over many iterations the bad strands will be removed and only good ones will remain. The more likely case is that only some of the source's state-action pairs will find correspondences in

[2]One in which each state-action pair finds a corresponding pair in the target where its knowledge is merged.

the target agent's state-space. In this case the sufficiently trained mapping will consist of good strands and fluctuating strands, where the fluctuating strands are constantly being rejected and trying new state-action pairs. As their transfers will not be merged, they have no impact on learning performance, which means they need not be removed which removes the complexity of checking if each strand has tried each of the target agent's state-action pairs. Leaving the fluctuating strands prevents accidental removal of potentially useful transfer routes. This could happen if the target learnt something that was not representative in a state (i.e., the value was not representative of the 'true' value) and its correct strand was rejected based on this wrong information. Allowing the fluctuating strands to retry states prevents this.

As it can take a considerable amount of time to find a correct mapping by voting, the mappings can be produced collaboratively by several agents. In this case, the sources must be mutually homogeneous as must the targets, as a single ITM can only translate information between one class and another. Regardless of whether single or multiple agents are involved in the learning the mapping, it fits into the PTL algorithm the same way (see Algorithm 3). The starting section of the algorithm (marked by the right brace) is PTL as described in Section 3.3. The extension to allow ITMs to be learnt is from line 11. FeedbackToSource gets whether or not the received states were merged, it then sends the feedback tuples for these states to the respective agents that sent them. If the Boolean variable *learningMapping* is true, then ReceiveFeedback gets any feedback tuples for the current agent to apply to its ITM. These tuples are then iterated through and applied. If the strand is determined to be not effective by its source's state's feedback score (accessed by getTotalFeedback($Q_{sent}(S, A)$)) and the $feedbackThreshold$, then RemapState($Q_{sent}(S, A)$) is used to replace it.

Vote-Based mapping requires each state-action pair to be visited at least once—probably several times—to correctly allocate strands between states. Learning the mapping in this way takes time. It also requires that the mapping is trained when PTL could have been providing knowledge and improving performance. It delays any benefits that PTL can bring. This makes it only practical when the time required to produce the

Algorithm 3 Learning an Inter-Task Mapping by Votes

```
 1: After RL Update
 2: for all Agents $n \in N^a$ do
 3:     $output \leftarrow$ SelectKnowledge($TS$, $SM$)        ▷ Choose knowledge to send
 4:     $output \leftarrow \chi_{A_a \to n}(output)$              ▷ Translate knowledge
 5:     SendKnowledge($output$, $n$)
 6: end for                                                 PTL algorithm
 7: $input \leftarrow$ReceiveKnowledge                         as in Section 3.3.
 8: for all $Q_{received}(S, A) \in input$ do
 9:     Merge($Q_{received}(S, A)$, $MM$)                      ▷ Incorporate new knowledge
10: end for
11: FeedbackToSource                          ▷ Feedback in case neighbours are learning
12: if $learningMapping$ then                              ▷ Learn mapping component
13:     $feedback \leftarrow$ReceiveFeedback
14:     for all $[Q_{sent}(S, A), value] \in feedback$ do
15:         ApplyFeedback($Q_{sent}(S, A), value$)           ▷ value either $-1$ or $+1$
16:         if GetTotalFeedback($Q_{sent}(S, A)$) $<$ $feedbackThreshold$ then
17:             RemapState($Q_{sent}(S, A)$)                       ▷ Was a bad strand
18:         end if
19:     end for
20: end if
```

learnt mapping can be reduced by sharing it between agents; agents could collaboratively produce a single mapping. It can also be useful if a learnt mapping can be recycled for agents joining the system after it has started. Generally, Vote-Based mapping will take too long to be practically learnt, so a faster method is needed.

3.4.2.2 Ant-Based Mapping

An alternate approach to mapping takes advantage of the fact that in most real-world systems the environment does not actually provide reward—as expected in the normal RL pattern—but agents self-calculate the ITM based on the state of the environment. This distinction is significant as it means that the agent has access to a function that approximates its value function, its reward function. The target task can share this with potential source tasks, that can use this to generate virtual experiences. These virtual experiences are then provided to ants for an Ant-Colony Optimisation [Dorigo et al., 1996]. In Ant-Colony Optimisation, lots of individual ants (i.e., agents with limited capabilities) traverse a search-space. While doing so, they lay down pheromone (i.e., a representation of how recently an ant has passed this point in the search-space, which decays over time), ants prefer to follow strong trails of pheromone. This means most of the ants will follow known routes through the search-space. Occasionally one will deviate and explore, and if this ant finds a shorter path, its pheromone trail will be stronger and other ants will follow it. Ant-Colony Optimisation finds shortest paths through search-spaces. Ants try to find good correspondences between the virtual sample of the target agent's reward function and the source agents own. If a good correspondence is found, it will likely be a good state to transfer to, as the value functions are likely similar as well. Similar reward is a good indicator that the states will have similar values when the transition function and environmental variability are accounted for. Conceptually this approach is related to Dyna [Sutton et al., 2012], but does not require a model. It too can be used to collaboratively produce a mapping, as long as the collaborative sources are heterogeneous as are the targets. The number of ants used and what threshold are required for good correspondences, depend on the application. Obviously the greater the number of ants, the better the chance of finding correspondences quickly. The smaller the threshold of difference between reward, the more difficult correspondences are to find.

When used with PTL, Ant-Based mapping can happen in parallel to the normal RL/PTL execution (assuming parallel execution is supported, otherwise it can be interleaved with PTL in any way). A state in the target is chosen, its reward function sampled and the ants are started. When the ants finish, the scores for the states they tried are compared to each other and to the threshold. A strand is selected based on the result. The threshold is used to prevent ineffective strands being included in the mapping. Algorithm 4 shows this process. In it, the use of subscript s or t indicates which state-space has a state or action, source or target. The process is not limited to only mapping unmapped states, it can remap states as well. This is particularly important when there are few ants used, as it is unlikely to find the best strand when only testing a small subset of strands.

Algorithm 4 Learning an Inter-Task Mapping by Ants

```
1: $targetReward \leftarrow \textsc{GetTargetReward}((S_t, A_t))$
2: for all Ants do
3:     $sourceReward \leftarrow \textsc{GetSourceReward}((S_s, A_s) \in StateSpace_{source})$
4:     $antScore \leftarrow \textsc{AbsoluteValue}(sourceReward - targetReward)$
5: end for
6: $\textsc{ChooseBestAntAndMap}((S_t, A_t), threshold)$
```

3.4.3 Summary

The learning of ITMs on-line has been presented in this section. Two different methods of learning ITMs were presented. One, Vote-Based mapping, can always be applied, but it requires more communications and is potentially slower. The other method, Ant-Based mapping, requires access to the reward functions of agents, which could limit its applicability.

Allowing agents to learn mappings on-line allows the requirement for heterogeneity support (D) to be met. This could have been done by calculating ITMs off-line, but this would have severely limited PTL's ability to provide on-line improvement. It would also limit the applicability of PTL to systems with combinations of agents known prior to their execution.

PTL using learnt mappings can now meet three of the requirements; efficient use of knowledge (A), on-line improvement (B) and support for heterogeneity (D). This leaves one requirement to be met; adaptiveness (C). Regardless of how PTL is configured and whether or not it uses a mapping, its performance is dependent on the environment and how the environment changes. If an environment changes, then the configuration of PTL will need to adapt. There are two ways this can be achieved, either the parameters can be learnt for each particular environment or they can be selected based on the environment. Learning the parameters will prove too slow to be practical in on-line systems, so a detection-based approach will be used to allow PTL to configure itself.

As the ITM production methods described here produce ITMs on-line, they will experience some degree of change as the agents at either end learn. Once the agents have finished learning and the ITMs are stable any non-stationary change in the environment may completely or partially invalidate them. If this happens the previously effective mapping will be a good start point for the new changed version, so the current mapping should be adapted. This is done by setting a maximum confidence in a particular strand. If a strand is known to be good and is regularly used, it will accumulate significant positive feedback. To prevent it being reinforced to a point where it could never practically be reallocated, a maximum score is set (in the current implementation 100 is used). Limiting the confidence in a particular strand means that if it is invalidated by change it can be corrected once its previous confidence is unlearnt. As there is some inertia in relearning ITMs, the response to change in systems using learnt ITMs will be slower that statically produced mappings. However, their facility to adapt offsets this.

3.5 Self-Configuration

This section introduces the Self-Configuration component, the purpose of which is to make PTL more effective in changing environments by allowing it to adapt. This component is responsible for setting PTL's parameters. It does this by monitoring the environment and how the agent is performing, and if either of these things deviate from expectations, then parameters are changed to remediate this.

3.5.1 Environments and PTL

As discussed in Section 2.3.2, there are three sources of change that could affect the representativeness of a state's value: dynamicity, non-stationarity and other agents. RL—and by extension PTL—learns a value for a state-action pair that includes the variability of the environment. A state's value can be seen as having four constituents as follows, with the latter three capturing variability:

(1) **Inherent Part** is the value that the state would have gotten if it was in a simple environment. It includes any natural variability in the reward received in the state.

(2) **Neighbour Part** is the change in value due the effects of other agents.

(3) **Dynamic Part** is the effect of any dynamic change in the environment.

(4) **Non-Stationary Part** is the share of the value that is due to changes in the environment that have happened since time zero (i.e., all changes over the lifetime of the system).

Of these, the non-stationarity part is the most significant, as it can completely invalidate the other parts, which increases the amount of learning to do and may reduce the effectiveness of PTL.

Different methods of selecting data (SM) and merging (MM) work well in different environments. Choosing the correct approach is important as PTL's performance is affected by these methods. It must be possible to change the method used, as in non-stationary environments, the best method to use may change over time. Equally, natural progression though the learning process may change, which is the best scheme to use; knowledge reuse will differ from early in the learning process to the end of it. There are two ways this can be approached. The first is the system can detect and categorise the environment that it is in and based on the resultant categorisation, select appropriate parameters. The other method is to attach learning processes to PTL's parameters and have them learnt by agents. Either method opens a further possibility for transfer as agents can transfer effective sets of parameters for their environments. Adding an extra level of learning will increase the time taken to learn by too much for any benefit to come from accelerating learning, so the predictive version will be used here. The learning-based version would also require that the characteristics of an environment's change can be accurately represented at a high level, so that this meta-learning process can know its state. It would also need several samples of each type of change to learn what is best for each one, which is impractical unless the environment is highly periodic or episodic.

3.5.2 Environment Detection

Detecting change in the environment is a complex issue and not the main topic of thesis, so a well-known approach will be used. Further work could be done to detect other types of change and to do so more accurately. For example, predictive modelling [Marinescu et al., 2015] or statistical techniques [Raza et al., 2015] can be used.

The first step in reacting to changing environments is to monitor the environment. Good monitoring of the environment allows divergence to be identified and reacted to. Agents have limited perception of the environment, even in fully observable environments. Generally, an agent will only know its own state, actions, reward and value function. These are the only ways that an agent can perceive the environment without requiring extensive additional modelling, so one or more of them must be used to detect change in the environment. Using state alone only tells the agent where in the environment it is, not what caused it to get there. It also is not able to provide information about value changes. The value function and reward can be used to detect change, but as a state's value is not equal to reward because of the update rule, it can be a poor indicator if there is natural variability in the reward. Actions provide no information about the environment directly, but comparing results with previous instances can provide information. This is problematic too if the actions themselves have variable results. This leaves reward as the best metric to use to detect change. The reward received in a state is a representation of how good that state is at the present moment. A change in the environment that requires remedial action will affect reward before any of the other sources of information available to the agent. So, if an agent maintains a history of the reward received at each state-action pair and checks for divergence, then state-action pairs can be categorised as either static or changing. If sufficiently many states are categorised as changing, then it is known that the environment is changing.

3.5.3 Environment Categorisation

Once it is known that a state has changed, an agent needs to decide how many states need to be categorised as changing for the environment to be so categorised. If only one state is found to be changing, then there is a risk that it is a detection error. Additionally, it is unlikely a change in the environment will only affect one state and if it does there will be minimal impact on performance. If an agent waits for all states in the environment to report change, then there is a significant delay in the agent's reaction to change. Each state needs to be experienced to sample the reward and thereby determine if it is changing. So, the threshold of changed states must be set to reduce the risk of reacting to errors, while still providing a timely response. Obviously this will differ between environments, so it needs to be set by the agent. In PTL, the agent tracks the number of times it visits states for both knowledge selection and merging, and this count can be used to identify states that are frequently visited. States that are frequently visited are particularly beneficial for both change detection and categorisation for several reasons. (1) Reward has been sampled more, so the model is better than in other states. (2) They are likely important states, so any change in them will have a greater impact on performance. (3) They are visited more regularly and will take less time to visit again, so change detection can happen faster. What counts as 'frequently visited' will depend on the environment, but any state representing over 10% of visits works well. This is a design decision based on the expected frequency of future visits to these states. In environments with very few states or very many, no states or only one may reach this mark. This reintroduces the susceptibility to reacting to errors, as in these cases the most visited states representing approximately 33% of all visits can be used regardless of their absolute value of visit count or its relationship to other states. The figure of 33% gives an expectation that one of the states in this set will be visited every 3 time-steps, which gives the potential for timely change detection.

Having categorised an environment as changing, the type of change needs to be identified. There is no hard set of rules that can be used to categorise change, as it can vary both spatially and temporally. Agents will have to estimate the type of change they are experiencing and react accordingly. The following categories are useful and commonly found in environments:

- **Environment Based Change**
  - **Slow Change** is change that has little impact on the reward received over short time periods, but continues until it impacts significantly.
  - **Fast Change** is significant over one time-step and continues for several.
  - **Sharp Change** is significant change that happens within one time-step and does not continue afterwards.
- **Other Sources of Change**
  - **Neighbour Change** is when the impact of neighbours' behaviour in aggregate changes from its previous level.
  - **Self Change** is when the agent is the source of change, this could be the addition of a new policy, more powerful actions etc.
  - **Finished Learning** can cause a change in what is required from PTL as an agent no longer needs information to support learning.

Differentiating between the two sources of change is beyond the capabilities of reward-based detection and requires more sophisticated detection. However, those changes in the second category appear as being the first from a reward point of view. Which means they can still be reacted to without the correct categorisation, though the reaction may not be optimal, it can still react. Assigning change to one of these categories depends on the detection method, but here it is based on the magnitude of the deviation from the modelled reward and the time period it occurs over.

3.5.4 Change Mitigation

Once the environment is found to be changing, the agent must decide how to react and what parameters to change, if any. Regardless of how the environment is categorised, when an agent reacts to a particular category it can change one or more of the following:

- **Knowledge Selection Method** (SM), how data is selected can be changed, if the environment is not changing the agent can be less cautious with its transfer, sharing more less-frequently-sampled-states.
- **States per Transfer** (TS) will vary based on the number of potential transfer targets. More agents mean more states may have changed, so more information should be sent. Transferring too many states can lead to sharing states that have not been recently sampled, which risks sharing outdated information.
- **Merging Period or Method** (MM) will change with the reliability of the received information and as an agent becomes more confident in its own state.
- **Between Which Agents** will change if particularly good pairs of agents are found.
- **RL parameters** can allow the agent to be more permissive with its learning and potentially learn to adapt to its new circumstance. With more extreme change, a completely new learning phase can be triggered[3].

In general, when the environment is currently changing, the agent should be more cautious. This will prevent misleading information from being shared. After change has finished, more information should be shared, so the agent can relearn more quickly. When the environment is not changing they should settle to their normal behaviour, whether this is designer encoded or configured based on how long the agent has been learning.

3.5.5 CUSUM

The approach to environmental change detection used here is CUSUM. CUSUM is a statistical method that has been used in industrial quality control [Hawkins, 1987]. It uses a model of the mean and standard deviation of a discrete time series process and parameters for false positives and negatives to detect when a process is out of control.

[3]Not considered here, as it is out of the scope of PTL.

It operates using a cumulative sum, so it can detect small changes in mean. It typically operates on processes with a known or targeted mean and standard deviation.

As in RL the mean and variation of the reward for a state can not be known a priori, the first samples are used to build a model of them. This requires that the process's initial rewards are representative of those provided by the current environment. This will almost always be the case. If not, CUSUM will learn a model that is not representative (what RL learns will also be not representative in this case), then it will detect a change as the environment goes back to 'normal'. The model can then be retrained on the correct reward. This means that regardless of whether the model is built when change occurs, it can be detected. This allows the model building to be done over several time-steps. The longer the model is built for, the better it can represent the inherent part (1) of the environment's variability in the environment, as it has more samples to model.

Retraining the model is needed after every change in the environment, as there is no guarantee that the environment will return to its previous reward distribution. If the model is not retrained, it will be impossible to detect further change, as the wrong model will consistently report change. For example, a model trained with the sequence of reward $\{A \mid 10, 15, 20\}$ will have a mean of 15 and a standard deviation of 5, if a change in the environment then give rewards of $\{B \mid 100, 150, 200\}$, this would be detected as a change. If the model was not retrained and reward kept being drawn from B, each instance would be reported as a change.

CUSUM is basically an upper and lower bounds on where a discreet value drawn from a modelled distribution can fall. It is recursive, so even small (within the standard deviation) changes can be detected over time. The upper bound is specified by Equation 3.2 and the lower by Equation 3.3.

$$S_{hi}(t) = \max(0, S_{hi}(t-1) + x_t - \hat{\mu} - k) \tag{3.2}$$

$$S_{lo}(t) = \max(0, S_{lo}(t-1) - x_t + \hat{\mu} - k) \tag{3.3}$$

x_t is the sample for the distribution at time t, $\hat{\mu}$ is the estimated mean and k is a design parameter. There is an additional parameter h which is used to determine if S_{hi} or S_{lo} are too large. If either is greater than h, then it has changed. The procedure on which CUSUM is based is called V-Mask [Montgomery, 2007]. In it, the parameters h and k are the slopes of the mask. If a point is found to be outside the mask, it is out of control. The selection of h and k can be rather opaque, so they have been related to three other parameters, which are as follows:

- $\boldsymbol{\alpha}$ is the allowable false positive rate. It must be greater than 0. This α is distinct from RL's α.
- $\boldsymbol{\beta}$ is the false negative rate.
- $\boldsymbol{\delta}$ is the deviation required to be detected. It is expressed as a multiple of the standard deviation.

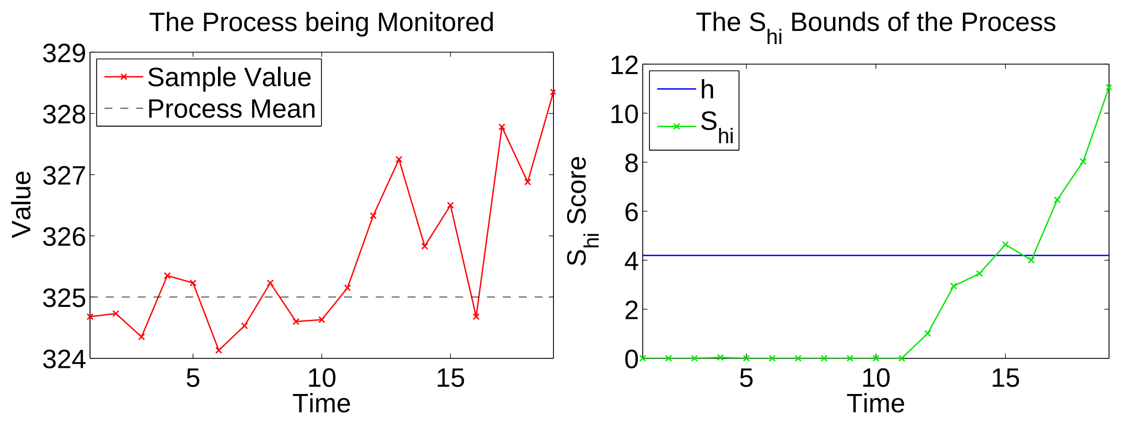
To convert from the easier to work with parameters to h and k, the Equations 3.4, 3.5 and 3.6 are used.

$$d = \frac{2}{\delta^2} \ln\left(\frac{1-\beta}{\alpha}\right) \tag{3.4}$$

$$h = dk \tag{3.5}$$

$$k = \frac{\delta\sigma_x}{2} \tag{3.6}$$

σ_x is the model's standard deviation and in Self-Configuration's use of CUSUM, is therefore an estimate and can be replaced with $\hat{\sigma_x}$. The selection of the actual parameters used can be reasonably permissive and risk false positives, as there is the extra step of aggregating states reporting change before making a decision. This provides some resilience to false positives. If δ is set low enough, β is not too high and $\hat{\sigma_x}$ is correctly modelled, then any samples that fall on the border of detection (where false negatives occur most frequently) and are not detected will likely not have been caused by very significant changes in reward [Natrella, 2010]. This adds a factor of safety for missing changes that will significantly affect performance. This can be seen in Figure 3.5[4] at times 14 $\rightarrow$ 16 where the process is out-of-control and in-control in quick succession. The figure shows the output of the upper bound in Equation 3.2 for a series of values. The lower bounds is omitted as this series does not deviate downward significantly. There is no need to build the model as described above, as the true mean is known to be 325, so no sample mean is needed as an estimate. From time 11 the process being monitored has begun to vary significantly, but not sufficiently to cross the detection threshold. The value of the process being controlled is in Figure 3.5b. The cumulative effect of the deviations prior to time 16 prevent the rather large drop in the process's output at that time from inducing a significant drop in the CUSUM's output. This 'memory' feature is desirable, as the changes to the mean are what should be tracked rather than variations in individual samples.



(a) Samples Drawn. (b) CUSUM Response.

Figure 3.5 CUSUM Change Detection with Samples from a Distribution (Mean = 325) Going Out of Control.

The magnitude of CUSUM's output grows with the process's deviation from the mean. This allows the type of change to be categorised. A CUSUM output of small magnitude is likely the result of slow or small change, while larger magnitudes indicate more significant change. In this example, the current S_{hi} rises at approximately one unit

[4]Figure based on an example by Natrella with slight modification [2010].

per time-step in response to a final sample five standard deviations from the previous mean (controlled standard deviation calculated over the first twelve samples and $\sigma = 5.1$). This is a relatively small change and could be categorised as such. There are no single values for S_{hi} or S_{lo} that can be used as a cut off between change types, as they are only loosely defined and application dependent, but approximately $10h$ and $20h$ work well as points to cut off from slow to fast and fast to sharp respectively.

Change in the reward is unlikely to be proportional everywhere in the state-space, reward in some states may increase while others may fall. This means some states may report change through S_{hi}, while others with S_{lo}. Both these two numbers need to be used to categorised change. As each state can only have one at a time—reward in a particular state that has changed can only be higher or lower than previously—and both are positive values that grow with increased deviation, they can be combined with an average of their magnitudes. This can be done irrespective of the direction of the change, as remedial action is based on the significance of the change, rather than the new value that will be learnt. The number of states exhibiting change is also an important heuristic for error detection. If only one state-action pair has changed and no others appear to have changed when sampled, it is likely the state-action pair with the changing value was incorrectly modelled. While if several values change, action may need to be taken. Errors can be corrected by remodelling the reward. CUSUM will only work for changes that affect reward. It is entirely possible that a change in the transition function of an environment could occur which would leave reward unchanged, but that would affect performance. For example, in a maze with reward only at the goal state, the walls could move changing the 'correct' path. This change would not affect the reward at states, but would affect the value function. In situations like this more sophisticated detection methods would need to be used.

Algorithm 5 from Line 21 is the Self-Configuration component, the previous lines are as presented in Section 3.4. UPDATECUSUM($Q_{RL}(S, A)$, *reward*) adds another reward to the monitoring process for the state-action pair (S, A). The CUSUM is then calculated for this to see if there has been change. The number of changed states and the average magnitude of change are then used to determine if a remedial action should be taken.

While the most effective PTL configuration for a particular type of change is evaluated in Chapter 5, some basic heuristics can be applied. When an agent's environment is not changing and it is early in its learning process, it should transfer states that it has visited the most, as these are more reliably sampled. Later in the learning process, states with the most impact on performance should be shared, for example, those with the highest value. When the environment is changing slowly, newer or received information will be best, as it likely has not been invalidated by change. During fast change transferring no information or only the most recently sampled will be best, as all previously learnt information may have been invalidated. This is visualised in Figure 3.6.

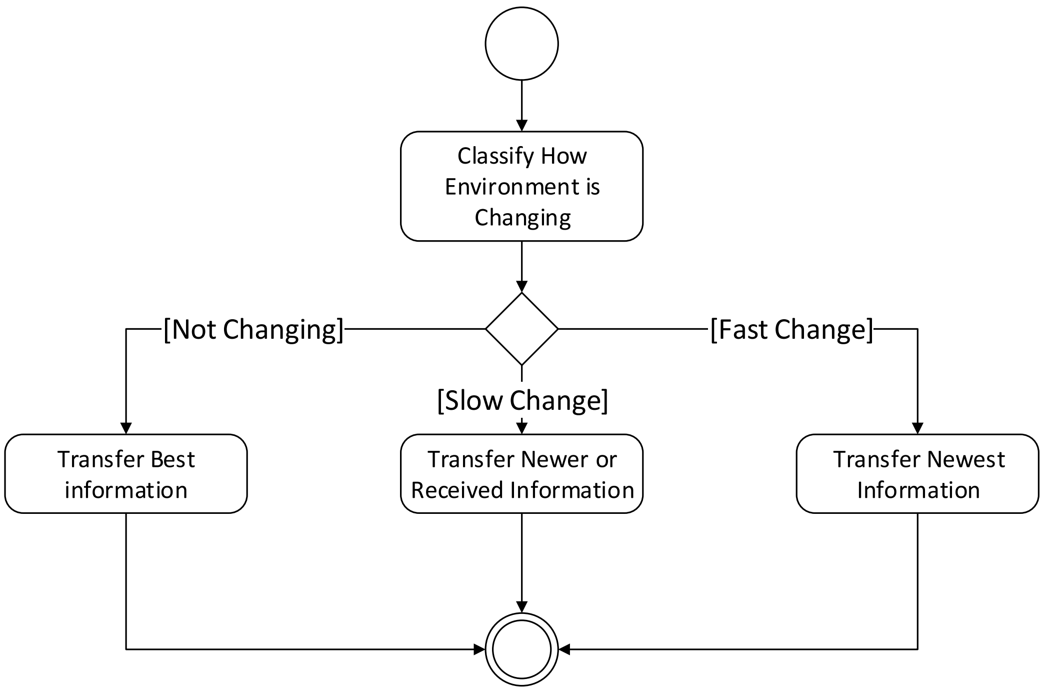

Classify How Environment is Changing
[Not Changing]
[Slow Change]
[Fast Change]
Transfer Best information
Transfer Newer or Received Information
Transfer Newest Information

Figure 3.6 Activity Diagram Self-Configuration.

Algorithm 5 Parallel Transfer Learning, Learnt Mapping and Self-Configuration

```
1: After RL Update
2: for all Agents $n \in N^a$ do
3:     $output \leftarrow$ SelectKnowledge($TS$, $SM$)        ▷ Choose knowledge to send
4:     $output \leftarrow \chi_{A_a \rightarrow n}(output)$        ▷ Translate knowledge
5:     SendKnowledge($output$, $n$)
6: end for                                          PTL algorithm
7: $input \leftarrow$ReceiveKnowledge                       as in Section 3.3.
8: for all $Q_{received}(S, A) \in input$ do
9:     Merge($Q_{received}(S, A)$, $MM$)        ▷ Incorporate new knowledge
10: end for
11: FeedbackToSource        ▷ Feedback in case neighbours are learning
12: if $learningMapping$ then        ▷ Learn mapping component
13:     $feedback \leftarrow$ReceiveFeedback
14:     for all $[Q_{sent}(S, A), value] \in feedback$ do
15:         ApplyFeedback($Q_{sent}(S, A), value$)        ▷ value either $-1$ or $+1$
16:         if GetTotalFeedback($Q_{sent}(S, A)$) $< feedbackThreshold$ then
17:             RemapState($Q_{sent}(S, A)$)        ▷ Was a bad strand
18:         end if
19:     end for
20: end if
21: if $detectingChange$ then        ▷ Change detection component
22:     UpdateCUSUM($Q_{RL}(S, A)$, $reward$)
23:     $numStates \leftarrow$GetNumberOfChangedStates1
24:     $changeMag \leftarrow$GetAverageMagnitudeOfChange
25:     if $numStates \geq$GetNumberOfRegularlyVisitedStates then
26:         if $changeMag \leq fastThreshold$ then
27:             ReactToSlowChange
28:         else if $changeMag \leq sharpThreshold$ then
29:             ReactToFastChange
30:         else
31:             ReactToSharpChange
32:         end if
33:     end if
34: end if
```

3.5.6 Summary

The ability to reconfigure PTL while it is running in response to changes in the environment is provided by the Self-Configuration component presented in this section. It monitors reward and if it diverges changes how PTL operates. This addresses the requirement for adaptiveness (C). It allows PTL to react to changes in the environment and thereby continue performing well in changeable environments. It does this without impinging on the other requirements that have been met by the PTL and learnt mapping components.

3.6 Overall Summary

PTL is designed to accelerate learning in MASs. In Chapter 2 requirements for accelerating learning and problems that affect learning rate were identified, Table 3.1 shows how PTL addresses them. The algorithm, PTL, allows knowledge to be shared between

Problem	**Reference**	**Addressed by**
Efficient Use of Knowledge	(A)	PTL component shares knowledge so it can benefit multiple agents
On-Line Improvement	(B)	Knowledge constantly shared, so change can be reacted to
Adaptiveness	(C)	Self-Configuration component can change how PTL operates if the environment changes
Heterogeneity Support	(D)	Learnt mapping component translates knowledge between agents
Sample Variation	(I)	Additional knowledge reduces samples needed at variable states.
Sparsely Visited States	(II)	Neighbours help by providing extra information.
Credit Assignment	(III)	States are moved to their correct value without backpropagation.

Table 3.1 Characteristics of Parallel Transfer Learning.

agents on-line. Allowing the source and target to run simultaneously in this manner allows the relatedness of agents in a MAS to be leveraged to accelerate learning. A pictorial representation of the algorithm is in Figure 3.7 in addition to the textual version presented previously.

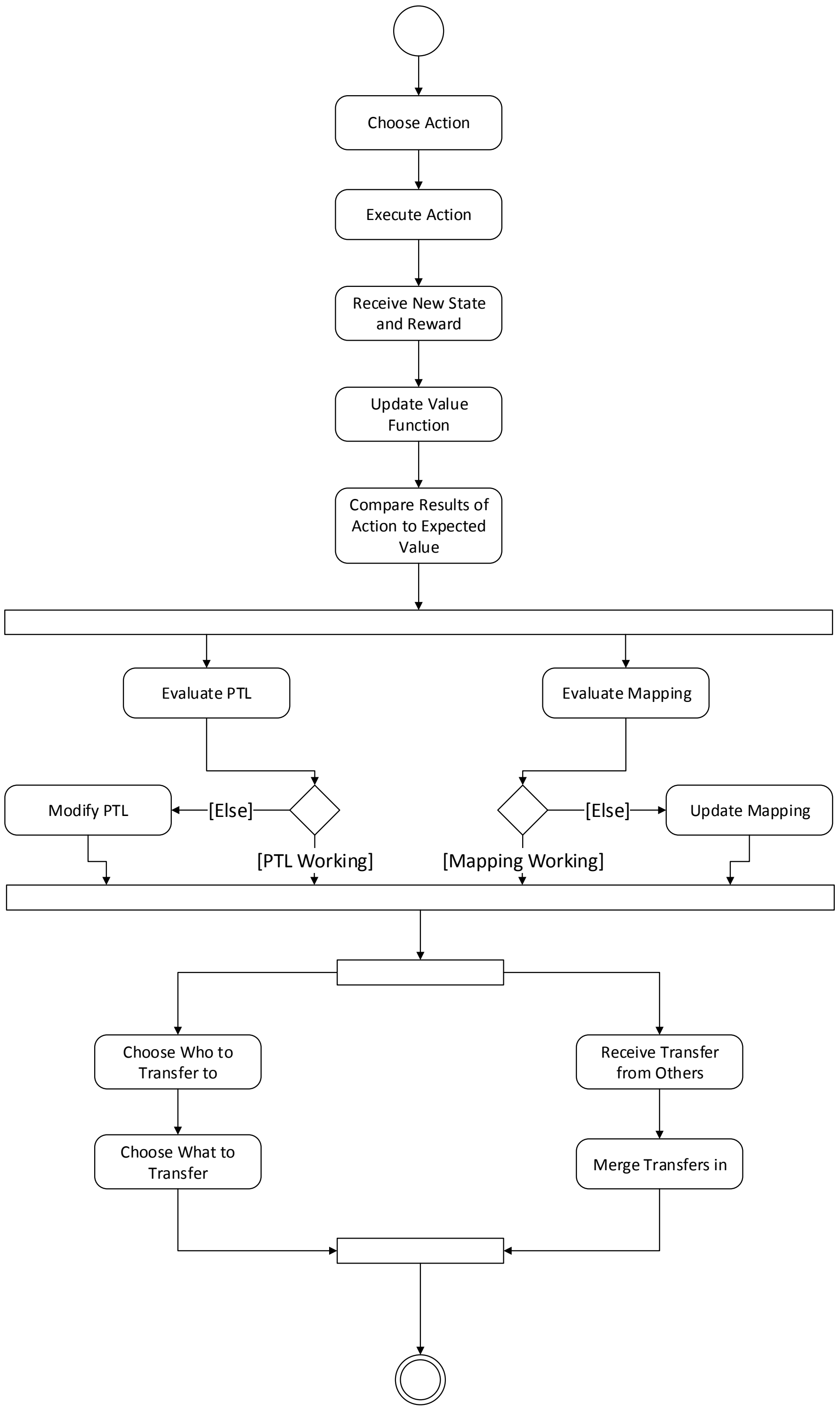


Figure 3.7 Activity Diagram of Reinforcement Learning, Parallel Transfer Learning and Self-Configuration.

Chapter 4

Implementation

You have to have an idea of what you are going to do, but it should be a vague idea.

Pablo Picasso

This chapter describes the Reinforcement Learning (RL) library used and its extension for Parallel Transfer Learning (PTL). It will also introduce the simulators used in Chapter 5 and their integration with the library. This chapter contains no new information about PTL and as such is not crucial to understanding it. However, it would be needed for the replication of results, and the set-up is important for the evaluation in Chapter 5. The code is available on GitHub[1] with further instructions on how to run it at `github.com/tayloral/GridLAB-D_RL`.

4.1 Library Design

Due to the way most RL libraries stored data and handled time, they were unsuitable to build PTL on, so one was written from scratch. The library is written in C++, for easy integration with one of the main simulators, GridLAB-D [Chassin et al., 2008]. Consideration was also given to deploying agents on real-world embedded devices. Many can run C++ which is not true of many other languages. Aside from this, it is well-known, commonly used and reasonably platform independent.

[1]GitHub is a register trademark (see `github.com`).

The main design goals were that agents could be easily created for new applications and integrated into them. The structure was kept general enough, so that other algorithms could be integrated into the same structure as long as they operate in discrete time. To achieve this, a top down approach was taken based on the requirements of the Distributed W-Learning (DWL) algorithm. DWL was used as an exemplar algorithm as it requires support for multiple agents, objectives and environment, each of which needs to be implemented separately. This gives a structure into which other RL algorithms could be implemented. DWL requires that the following steps happen:

- **Select Action** chooses what action will be executed.
- **Execute Action** applies the selected action to the environment.
- **Update Local Objectives** informs an agent about how the environment changed.
- **Pass Messages** informs neighbours of what has happened.
- **Update Remote Objectives** updates an agent based on what the neighbours have said happened to them.
- **Transfer Information to Others** selects and sends PTL messages to neighbours.
- **Translate Knowledge** uses an Inter-Task Mapping (ITM) to translate knowledge to the recipient's representation.
- **Receive Information from Others** receives and merges PTL messages from neighbours.
- **Update Mapping** happens if the agent wishes to learn an ITM.
- **Self-Configure** adjusts the parameters of PTL.

The communications are all asynchronous, but tighter requirements can be added. There is no requirement that these steps all happen or in any set order[2]. These steps allow

[2]Obviously to learn anything reasonable, an agent should learn from the effect of its own action which implies a causal ordering, but this is not enforced by the library.

most discrete time algorithms to operate, so are a good basis around which to build the library. It also facilitates the integration of simulators as an external process to the agent. The execute action step is simply replaced by a time-step of the simulator. Similarly, the pass messages step is decoupled from actual communications and simply serialises messages for some other process to deliver. All of these specialisations are handled by the actual use of the library and allow the library's core functionality to be agnostic of a particular deployment. The library was built from scratch as existing RL libraries were not flexible enough to use with the external simulators without considerable effort. Use of external libraries can also limit deployment on some embedded platforms.

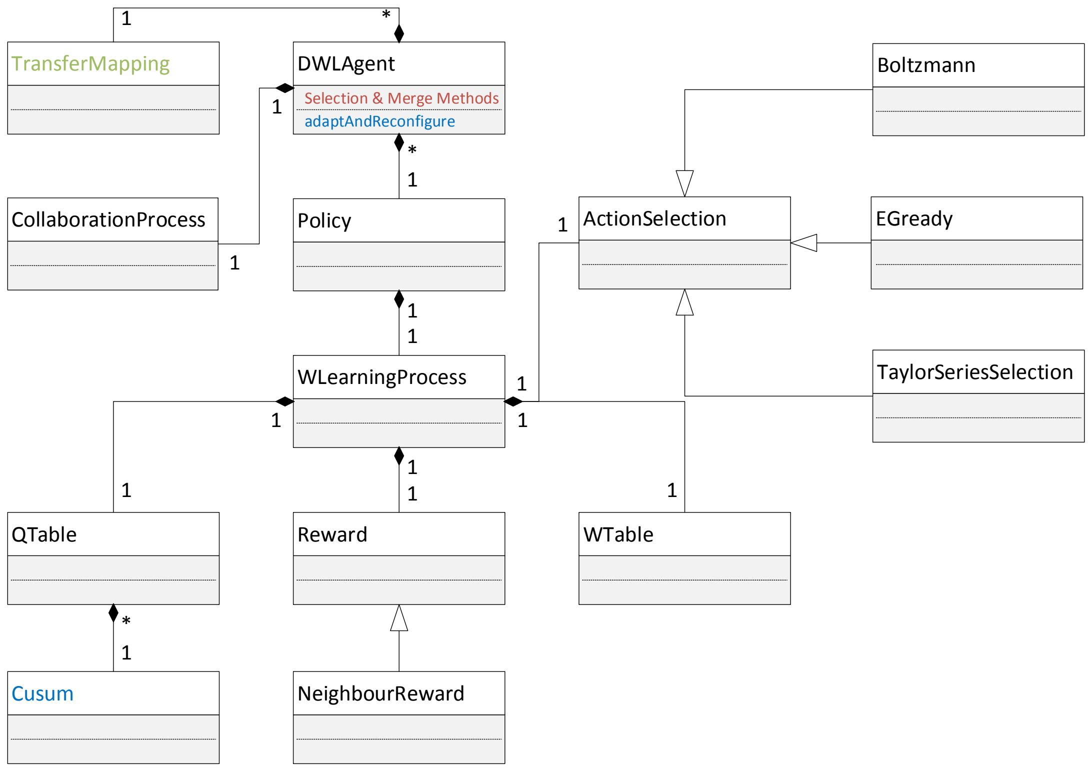


Figure 4.1 Class Diagram of the Parallel Transfer Learning Library.

4.2 Library Structure

The library is organised, so an agent (DWLAgent in Figure 4.1) contains everything needed to perform RL. The location of the main methods for each component are identified by colour. Red for PTL component, green for Learnt Mapping component and blue for Self-Configuration component. A Policy is a container for a single objective, if an agent is learning to meet this objective with any Q-Learning-based algorithm it will need a WLearningProcess. If the agent only has one objective, the fact that W-Learning is used does not matter, as there is no inter-objective arbitration that can take place; there is only one action to choose from at any time. In the case of multiple objectives (in DWL for example), policies can be either local or remote. The local policy represents and agents own objective, while remote ones represent neighbours' objectives. These three classes are the main parts of the required structure. The environment drives what an agent must do at any one time.

When an agent is required to select an action by the environment, it polls all its policies via the nominate method in DWLAgent. Nominate asks each policy for an action suggestion. Action suggestions are the action that an objective would like to be executed at the given time. In DWL, each action suggestion comes with a W-Value (weight based on its importance). These W-Values are maintained by the W-Table. The actual action contained in a suggestion is chosen by a class that extends ActionSelection. ActionSelection gets the possible actions for the current state from the QTable (which also maintains the Q-Values). Once nominate has the action suggestions from all policies, the one with the highest W-Value is provided to the environment. The TaylorSeriesSelection class provides a directed but somewhat random search based on temperature much like Boltzmann does, but without the problem of exponentials exceeding bitwidth. The temperature is on the range $[0, 1000]$, where 0 is purely exploitation and 1000 is random selection. Anything in between these points uses the temperature as a probability of using random selection. For example, a temperature of 400, would choose a greedy action

60% of the time and a random exploratory action the remainder of the time. This is a novel method to avoid the bitwidth limitations of Boltzmann.

Once the environment has executed an action, DWLAgent's updateLocal method is called. This provides each of the local policies with the environment's new current state. Each local policy calls its own update function and performs Q-Learning and W-Learning updates. The reward received can be passed in from the environment or some other external process. If the environment lack the ability to calculate reward, then the WLearningProcess can self-calculate its own reward.

DWL requires update messages be sent and received prior to calling updateRemote. updateRemote then processes these messages and learns how the agents' actions affect its neighbours. Transfers from PTL also use the message passing interface. It selects data to transfer with TransferToAll and readTransferedInfoIn merges it. Both of these methods are affected by the Self-Configuration component. Self-Configuration is encapsulated in DWLAgent's adaptAndReconfigure method and uses the CUSUM value that each state-action pair has and calculates with its attached Cusum class. Importantly, a function called finishRun in DWLAgent must be called to end each time-step. It does several things including removing old action suggestions and messages. Most of the data structures used are C++'s Standard Template Library containers, so minimal effort is required for others to use the library [Plauger et al., 2000]. As the Standard Template Library is used, memory must be managed by the agent, this is done by finishRun releasing old messages once they have been processed. This sequence of interactions between is shown in Figure 4.2.

While the current implementation of PTL is built on top of DWL, it would be relatively easy to adapt to other RL algorithms. The agent structure allows different temporal difference algorithms to be added with only a small change to the update rule. Adding support for functional approximation or continuous state-spaces would require changes in the underlying RL representation, but the transfer infrastructure would need only minimal changes to accommodate the new representation. The most work would be required if support for policy-search were required, as both the representation and update rule would need to be changed. The information exchanged may also need to change, as policy-search algorithms do not necessary have state-action values (which the code currently shares). If policy values have to be passed rather than state-action values, then a small change would have to be made to the transfer infrastructure so it could interpret policy values. Other than this there are few changes needed to the code to make it more generalisable.

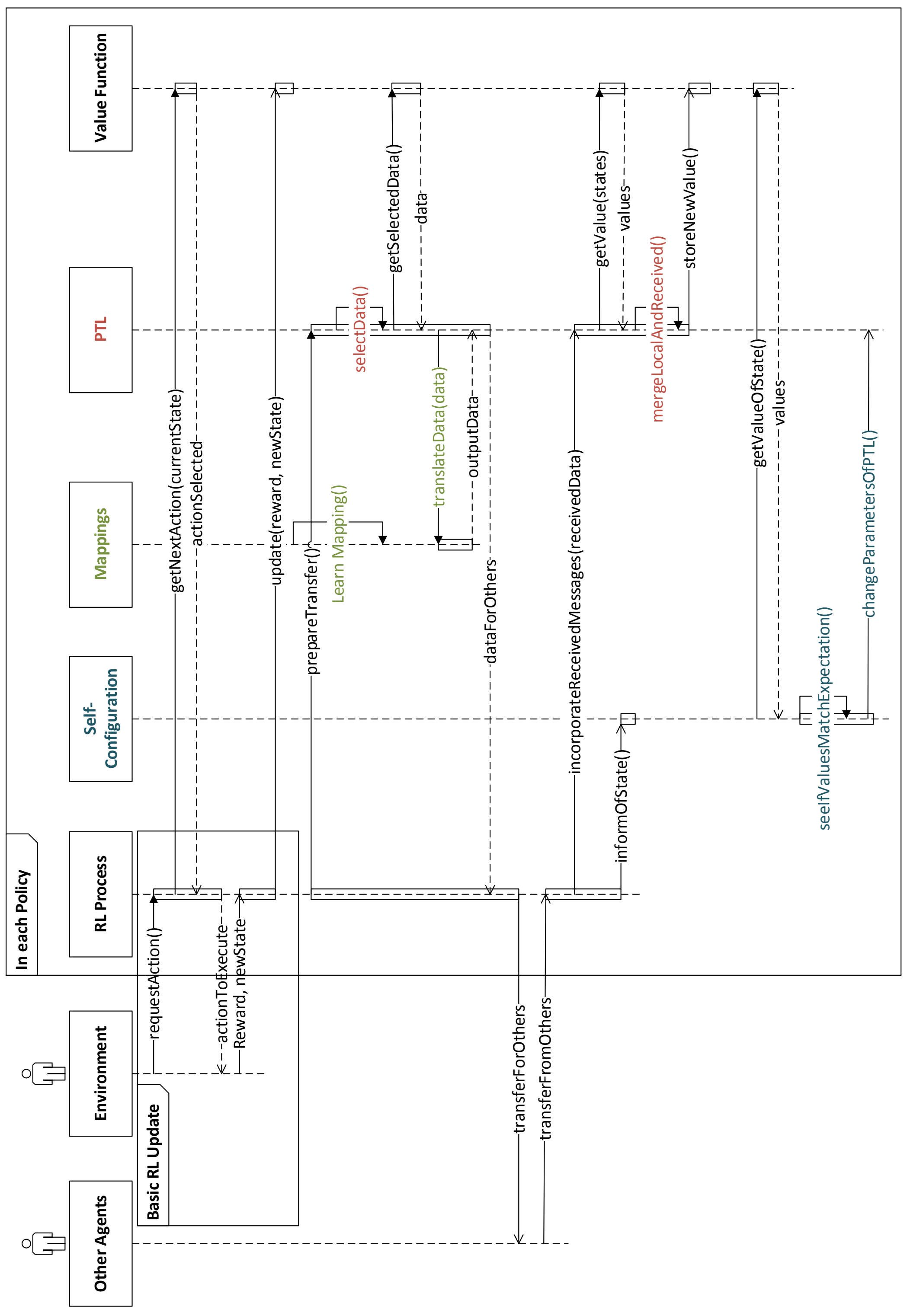


Figure 4.2 Sequence Diagram of the Parallel Transfer Learning Library.

4.3 Library Use

To implement a DWL agent using the library the first three things are needed, the fourth is needed to implement PTL:

- **Extend DWLAgent** to produce a specialised agent for a particular application. This will need to add policies with sets of states and actions.
- **Extend Reward** if the environment is not capable of providing it. Each policy needs a separate reward function, assuming that different policies encode different objectives.
- **Provide Interpretation** for the agent. This will need to turn the agent's action representation into something understood by the environment, the environment's state will need to be translated to the agent's representation. Communications infrastructure should be provided as well.
- **Transfer Set-Up**, if the agent is not learning an ITM, then one will need to be provided for PTL to occur. Pairs of agents for transfer also need to be specified, as there is no way to identify good transfer pairs otherwise. Pair selection can be done by the library, but not in a particularly intelligent way.

While the library can be further configured: parameters set, neighbours selected or other components added; the above is all that is required.

4.4 PTL Component

As was mentioned in Chapter 3, at a high level, PTL is two loosely coupled pairs of methods. The first is the selection method SM and the merge method MM, the second is the communications wrappers for these. When an agent wishes to transfer to its neighbours, it calls the transferToAllFromAll method. The flow of gathering information is shown in Figure 4.3. As two components are involved in the transmission of knowledge, the colour scheme is kept the same as Figure 4.2. Green indicates the Mapping component; red, the PTL component and black are generic methods. This polls all of its objectives for the knowledge they want to transfer. This is done through the transferToAllFromOne method. This method actually applies the selection method SM to the value function for the objective. Applying the selection method iterates through the value function stored in the QTable and gathers TS state-action pairs. These pairs are then passed to the Mapping component to be translated. The translated version of the knowledge is then passed to the communications. The communication is done by serialising the knowledge to transfer into an XML-like format before transmission. The XML-like format is used so that knowledge can human readable for debugging purposes. The library supports sockets, shared memory and direct method calls to provided communication, but only at a low level, there is no routing or discovery or other high level services. This allows the library to be used across networks, in multi-threaded applications and in a single process. The sequence of actions involved in the merging of received knowledge is shown

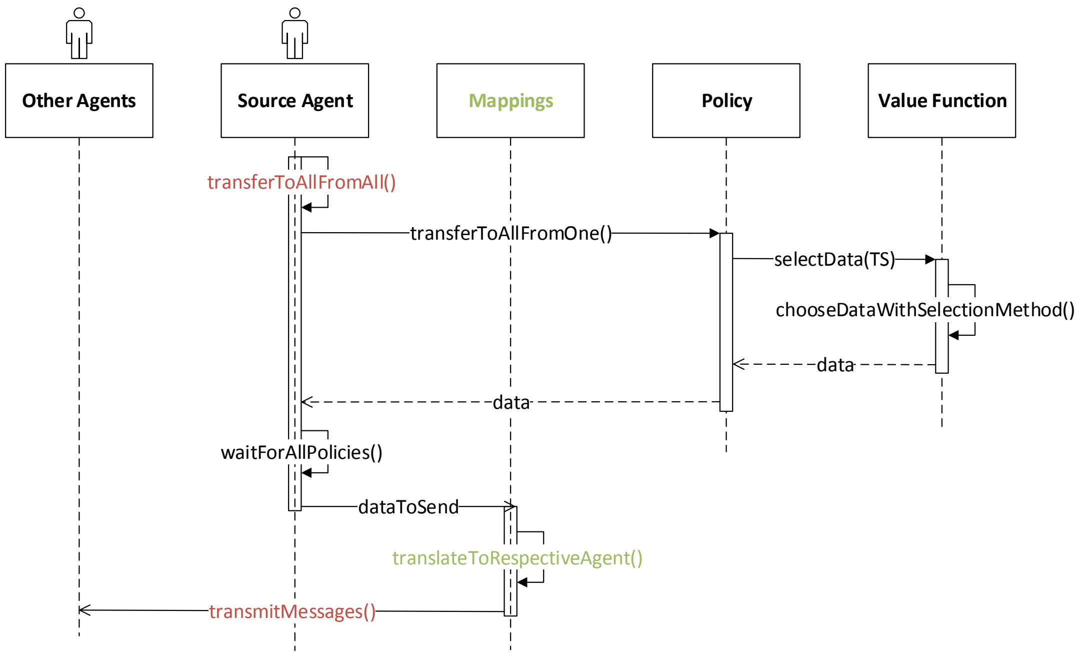


Figure 4.3 Sequence Diagram of Parallel Transfer Learning Knowledge Selection.

in Figure 4.4. Again, red indicates the PTL component and black are generic methods.

When a transfer is received by the communications interface it sits in a buffer until PTL is ready to process it. Once it is ready, it parses the message and splits it into sections for different policies. These sections are then passed to their respective policies. The merge method MM now coordinates the actual integration into the state-space stored in the QTable. It gets the current value for the state-action pair and the metadata about the state. It then uses this information to determine if it should be merged and how. First, the heuristic that estimates if the transfer agrees with the expected converged value is used (discussed in Chapter 3). If it passes this test, then the new value is calculated by weighting the local knowledge and received knowledge based on the amount of times the state has been visited. Finally the result of merging is stored.

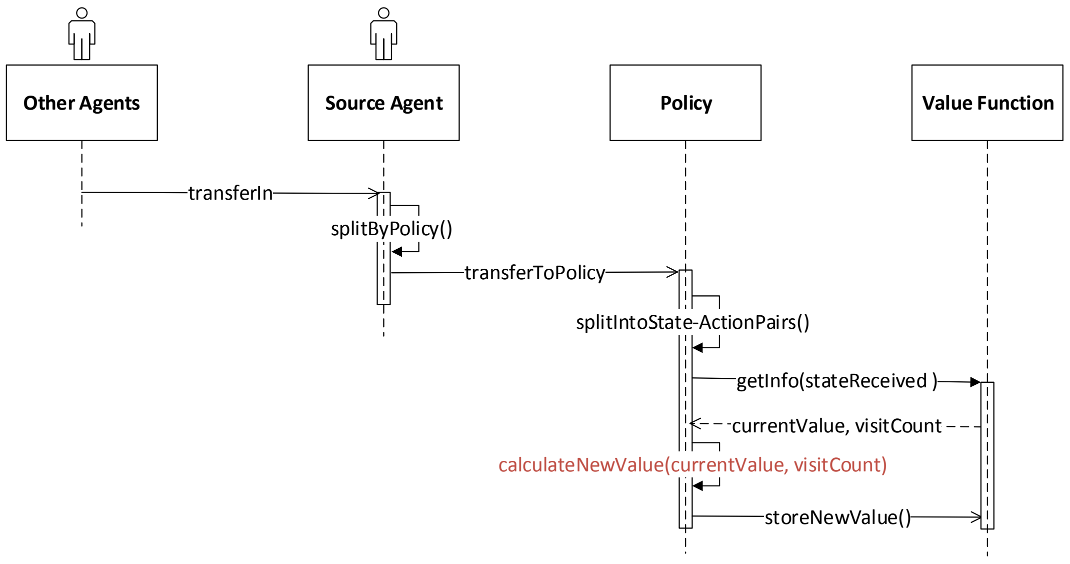


Figure 4.4 Sequence Diagram of Parallel Transfer Learning Knowledge Merging.

4.5 Learnt Mapping Component

The two different methods of mapping operate in similar sequences; Vote-Based is shown in Figure 4.5. All of the methods used in it are part of the Learnt Mapping component, so no colouring is applied. Once the PTL component gives the Learnt Mapping component some knowledge to translate and an agent's objective to map it to, it finds the correct mapping then it begins searching the set of strands for the state-action pair that is to be mapped to see if it can translate the knowledge. If it fails to find a strand, it checks to see if that strand has been mapped before, and if not, it maps it randomly and translates the knowledge. If the state is not in the mapping and has been tried before it is assumed to be unmappable and is not mapped. Once this is done for all state-actions to be sent, the result is returned. The recipients of this knowledge will send back feedback when they apply their merge method MM to the mapped knowledge. This feedback is then sent to the source agent, and upon receiving this, the second half of the sequence diagram happens. The feedback is applied and stored by the TransferMapping class. The entire ITM is then scanned for any strands whose feedback score falls below the threshold for good mapping (usually zero, which means that any bad feedback causes the state to be remapped). Any strands that are identified as requiring remapping are removed and their state-action pairs added to the unallocated pools (unless they were in other strands as well). The unallocated pools are sets of state-action pairs which are not connected by strands. When an ITM needs a new strand, it first looks in the unallocated pools for potential strands. The rationale for this is that any state-actions that are mapped effectively will be left as they are while the rest of the mapping is developed.

Instead of the feedback process used in the Vote-Based mapping, the Ant-Based mapping has an update step using ants. The ants use the same structure as Vote-Based mapping to store the pheromone, the closer relate two state-action pairs are, the more pheromone 'votes' that strand receives. The Ant-Based mapping is then checked the same way as the Vote-Based, any strands below a threshold are removed and added to unallocated pools.

4.6 Self-Configuration Component

The implementation of the Self-Configuration component is less discrete than the other components. Its main action is shown in Figure 4.6. The self-configuration process

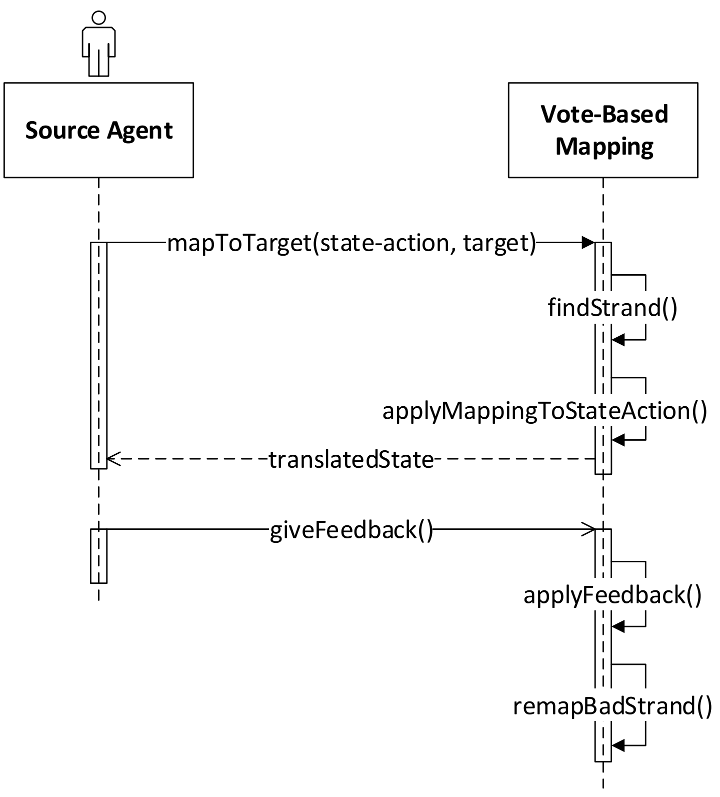


Figure 4.5 Sequence Diagram of Vote-Based Mapping.

does not have the same temporal requirements as the rest of PTL. RL needs some temporal ordering as the update must follow the action to learn correctly. With PTL, there is an expectation that knowledge will be shared and merged regularly so that it can improve performance. Self-configuration is likely to happen much less frequently, as change in the environment is rare compared to the frequency of RL or PTL. When there is no change in the environment PTL's configuration is kept static, so the Self-Configuration component only has to do periodic monitoring until it detects change. The regular updates to the CUSUM are done following an RL update. As mentioned previously, each state-action pair has its own CUSUM process that monitors if it is changing or not. The latest sample of reward is added to CUSUM's model by default following an update. The compareToThreshold method takes the result of polling all of the CUSUMs and determines if change is occurring based on the number of changed states. If sufficiently many states are changing, then it will be categorised as fast, slow or sharp (see Chapter 3). If fast change is happening, then the greatest change method is used with a small transfer size TS. This means only new information is shared, this information could have come from local information or from a transfer. If slow change is detected, the most visited and converged states are transferred as these have a greater degree of confidence than the other potentially changed states. If sharp change happens, only the most recently visited state or received states are transferred as all information is assumed to be potentially invalid. If any type of change is detected, the merge method MM has its visit counts reset. This effectively drops the confidence in each state and the agent becomes willing to merge information that does not agree with its local knowledge.

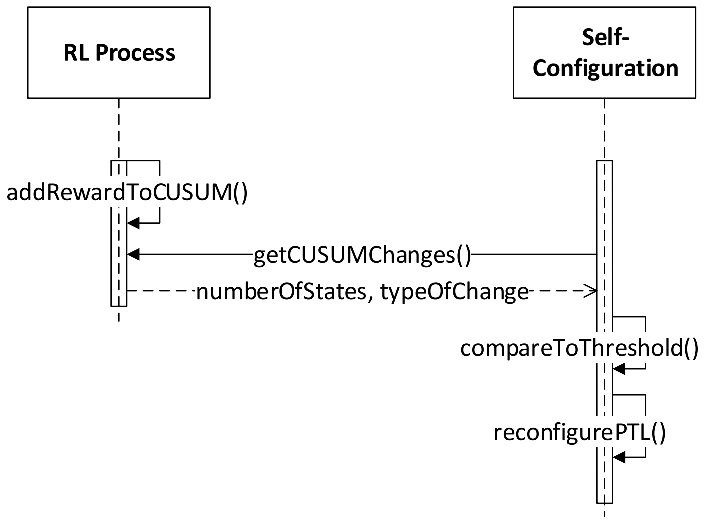


Figure 4.6 Sequence Diagram of the Self-Configuration Component.

4.7 Simulators

The simulators used will be described here from an implementation point of view. Their selection and particular scenarios used will be presented in Chapter 5.

4.7.1 Mountain Car

The Mountain Car is well-known in RL [Knox et al., 2011]. In it an agent must learn to drive a car up one side of a valley. Its engine is insufficient to do this, so it must first reverse up the opposite side to gain enough potential energy to complete the task. The parameters used are based on those used in [Sutton and Barto, 1998]. They are detailed in Tables 4.1 and 4.2. The reward was changed to this form, as the more common form, $-1 + height$, directly encodes the knowledge that height is good which is one less thing for the agent to learn. It also creates two local maxima (one at each side of the valley), which makes the agent's goal somewhat ambiguous. There is also a change to the scaling factor on the velocity update, so more ground is covered by one time-step. This makes learning more difficult, as it covers more ground in one time-step which means each action has a greater effect on the environment. The Mountain Car is a Markovian environment as the effects of actions at previous times are encapsulated in its current state variables. This means that anything that happened at $T - n \forall n > 0$ is completely included in the current state and there are no lingering effects.

Parameter	**Value**	**Description**
Position	$[-1.2, 0.6]$	Quantised in to 4 states variables with equal ranges with one additional goal state.
Goal position	0.6	" "
Velocity	$[-0.07, 0.07]$	Range split into 4 states variables covering $\frac{1}{3}, \frac{1}{6}, \frac{1}{6}, \frac{1}{3}$ of the range respectively.

Table 4.1 Mountain Car State Variables.

Parameter	**Value**
Actions	Left -1, Neutral 0 or Right 1
Reward	10 if at the goal, -1 otherwise.
Velocity Update	$Velocity + action * -0.00375 * \cos(3 * Position)$
Position Update	$Position + Velocity$

Table 4.2 Mountain Car Update Rule and Reinforcement Learning Settings.

4.7.2 Cart Pole

The Cart Pole or Inverted Pendulum is similarly well-known [Sutton and Barto, 1998]. In it, a cart must move horizontally so as to balance a straight inflexible rod that is attached to it but free to rotate. The parameters used are in Table 4.3. The pole was lengthened slightly to make the problem more challenging. Its state variables and reward are described in Table 4.4. The Equations 4.1 - 4.6 are used to update the state of the system. The physical interpretation of the parameters is in Figure 4.7. Additionally τ is the length of a time-step, $\dot{x}$ is acceleration of the cart, $\dot{\theta}$ is the pole's acceleration and m_c is the cart's mass. The agent's goal is to keep the pole balanced for as long as possible. The Cart Pole is Markovian for similar reasons to the Mountain Car, there are no effects from previous actions.

$$L = \frac{l * m}{2} \tag{4.1}$$

$$x = \tau * \dot{x} \tag{4.2}$$

$$\dot{x} = \tau * \frac{forceComponent - L * \dot{\theta} * \cos(\theta)}{m + m_c} \tag{4.3}$$

$$forceComponent = \frac{f * action * L * \dot{\theta}^2 * \sin(\theta)}{m + m_c} \tag{4.4}$$

$$\theta = \tau * \dot{\theta} \tag{4.5}$$

$$\dot{\theta} = \tau * \frac{(g * \sin\theta - \cos(\theta) * forceComponent)}{\frac{\frac{4L}{3} - m * \cos(\theta)^2}{m + m_c}} \tag{4.6}$$

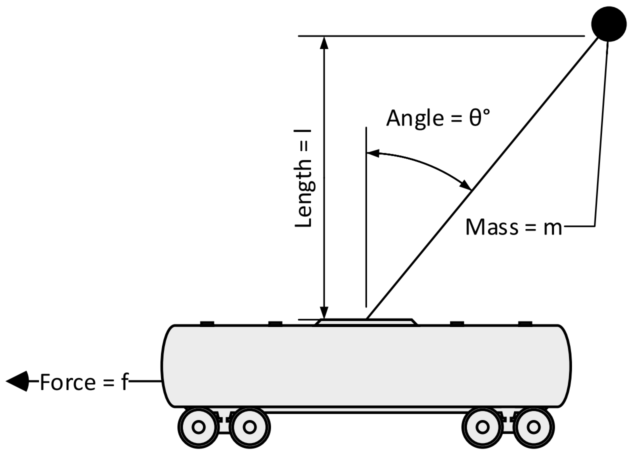


Figure 4.7 Cart Pole Parameters.

Parameter	**Value**
Actions	Left -1 or Right 1
Reward	-20 if the pole falls, 0 otherwise.
Gravity	9.8 ms^{-2}
Mass of Cart	1 kg
Mass of Pole	0.1 kg
Length of Pole	1.4 m
Force Applied	± 10 N
Simulation Time-Step	0.01 s

Table 4.3 Cart Pole Parameters and Reinforcement Learning Settings.

4.7.3 GridLAB-D

GridLAB-D is an electrical grid simulator produced by the U.S. Department of Energy ar Pacific Northwest National Laboratory [Chassin et al., 2008]. It simulates the grid with sub-second accuracy of the behaviour of electricity. It is a finite difference based simulator, which means that elements of the simulation can be synchronised at different rates depending on how much they change. It is relativity easy to change the structure

Parameter	**Value**	**Description**
Cart Position	$[-2.4, 2.4]$	Quantised in to 3 states variables with equal ranges.
Cart Velocity	$[-1, 1]$	Quantised in to 4 states variables with equal ranges.
Pole Position	$[-90°, -6°, -1°, 0°, 1°, 6°, 90°]$	Quantised in to 6 states variables as indicated.
Pole Velocity	$[-\infty°, -50°, 0°, 50°, \infty°]$	Quantised in to 4 states variables as indicated.

Table 4.4 Cart Pole State Variables.

of the grid and data logging is inbuilt. Most importantly, it is open source, so it can be integrated with the PTL library.

GridLAB-D's integration with the library is more involved than integration with the previous applications. The other simulators and the PTL library could interact directly, as they were the only components involved. The simulation could progress without requiring any temporal synchronisation, as either PTL was happening or the simulation and the other was waiting. This effectively kept their method calls synchronised, so RL and PTL were occurring at the right times to learn from the results of their actions. In GridLAB-D there are other components that need to be synchronised to accurately simulate electricity flow, most notably Powerflow and Core (see Figure 4.8). Figure 4.8 shows the required classes to connect PTL to GridLAB-D. The Core component controls the simulation, requesting components synchronise when they are needed. This is why PTL and the simulator needed to be so closely linked. With core controlling the time synchronisation of method calls, the connecting code needed to force PTL and the Residential component to happen together so that the correct sequence of action selection, learning and transfer could happen. For the PTL library to be used it required an interpretor, this converts the actual values for load and battery charge to states. Additionally, Reward needs to be extended for each objective, so that the agents can determine how well they are doing. In GridLAB-D's Residential component, the class representing the particular device we are using (EVCharger is an Electric Vehicle (EV)) is altered so that it uses PTL or RL. There are other classes in the Residential component, but they are not impacted on by integration with PTL.

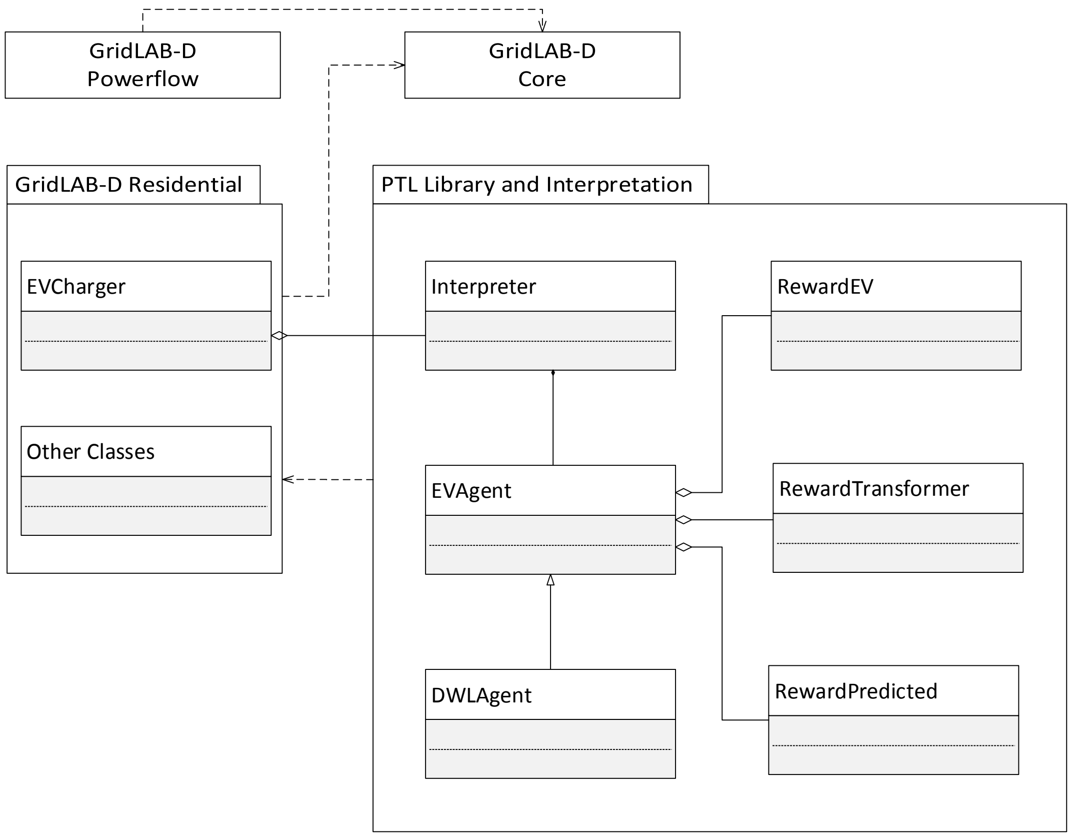


Figure 4.8 The Relationship between Parallel Transfer Learning and GridLAB-D.

The Smart Grid (SG) is a new model for electrical grids by which flexibility is introduced through new technologies most notably computing and intelligent control [Farhangi, 2010]. Since the first power grids in the late 19th century the electrical grid has operated one way, from large central power plants to end-users. Demand was reasonably stable and the main concern for grid operators was providing economical, reliable power. In the early 21st century several changes began happening that have affected the problem of control in the electrical grid [Fang et al., 2012]:

- **Distributed Generation** changes the way electricity flows in the grid, coming from end-users rather than just to them.
- **Electric Vehicles** not only increase demand (as energy comes from the grid rather than internal combustion), but provide capacity for distributed energy storage.
- **Abundant Sensors** make it easier to know the current state of the grid and inform predictions about future states.
- **Renewable Generation** is generally less flexible than conventional generation, so the grid must accommodate this. For example, the output of an oil fired power plant can be easily increased or decreased, but a wind turbine's output can only be decreased by its operators.

While conventional control of the grid could accommodate these things, the SG's aim is to improve the efficiency of the grid, which would be curtailed by inexact control schemes. Aside from the variability in the grid, the number of different stakeholders means each device can have multiple different objectives to be balanced. These factors make the SG an interesting venue for applying learning-based control.

Residential Demand Response (RDR) is a method of solving many problems in SG applications [Forouzandehmehr et al., 2015; Mohsenian-Rad et al., 2010]. Rather than as was traditionally done, increasing electricity generation to match demand, in RDR electricity demand is increased or decreased to equal generation. There are several reasons why this approach is preferable to supply side methods:

- **Ancillary Services** are electrical concerns (such as maintaining correct frequency) that can impact on the quality of the electricity produced. There can be rapid and dynamic changes that must be addressed in real time. A central solution can take time to propagate to their location and affect other areas of the grid as well.
- **Uncontrollable Supply** is generally from renewable energy and can lead to over production of electricity. This surplus would otherwise be wasted. The disposal of this excess energy is known as curtailment and it can be expensive. Renewable

energy sometimes has to be curtailed because of the guarantees of energy quality provided, which can not be maintained by ancillary services otherwise [Harris et al., 2014]

- The **Cost** of meeting high demand peaks is generally significant, as markets buy the cheapest energy first. If the peak can be shifted to some other time more cheaper energy can be bought.
- **Grid Resilience** can be achieved through RDR's load shedding. If there is some unpredictable event increasing demand or lowering supply, a reduction in load can protect grid stability.
- **User Benefits** are potentially available in energy bill reduction. As there is considerable benefit to the grid there will be cost incentives to participate in such programs.
- **Hardware Life** can be prolonged by reducing the amount of times a device is pushed close to its limit and by smoothing demand profiles.

To implement RDR, devices must be flexible and able to control their own operation. When demand is less than supply, extra demand can be introduced either by activating more devices (for example, turning on storage devices) or increasing demand in currently operating ones (for example, charging EVs at a higher rate). If supply is less than demand, the demand can be shed though the opposite operations.

The degree to which individual devices can contribute to RDR programs depends on their operational requirements and electricity draw. Devices with an expectation of timeliness (ovens for example), lack the required flexibility to be useful. Devices with a small power draw do not have a significant impact. Devices with large power requirements and little or no energy storage (such as clothes dryers) are really only useful for adding demand. Devices with storage and large power draws that are both flexible and have an effect on the grid are most useful for delivering RDR (for example, EVs, heating systems, water heaters, etc.). The storage can be thermal, chemical, electrical or any other form, all that is required is that operation can be rescheduled within some time window without affecting users overly.

RDR tends to be referred to using the specific terminology depending on whether load is being raised or lowered. Lowering demand is called peak reduction or peak shaving, raising demand is called valley filling. Both of these can be achieved independently or as a result of load shifting. Figure 4.9 shows a sample of normal residential demand over one day. Typically residential energy use follows this pattern, low usage overnight and in the early morning, followed by higher usage until the afternoon, before peaking in the evening. Ideally this evening peak would be reduced and the night time valley filled, achieving load shifting and flattening the demand profile. This demand profile without the reschedulable load is commonly called base load. Generally, the base load and the well scheduled reschedulable load together will produce a flat line, as a stable level of output is preferable for large-scale generation. Other shapes can be produced if RDR has other goals (e.g., to maximise renewable energy generation the shape will correspond to the level of generated energy).

From a RL point of view, the SG is non-Markovian. While a single-agent version would be Markovian, the ambiguity of other agents introduce the possibility of temporal dependencies. The simple Markovian 'test' of would a prediction be improved by knowledge of previous time steps, in this case would be failed. Knowledge of what other agents do and how they decide on it would greatly improve prediction.

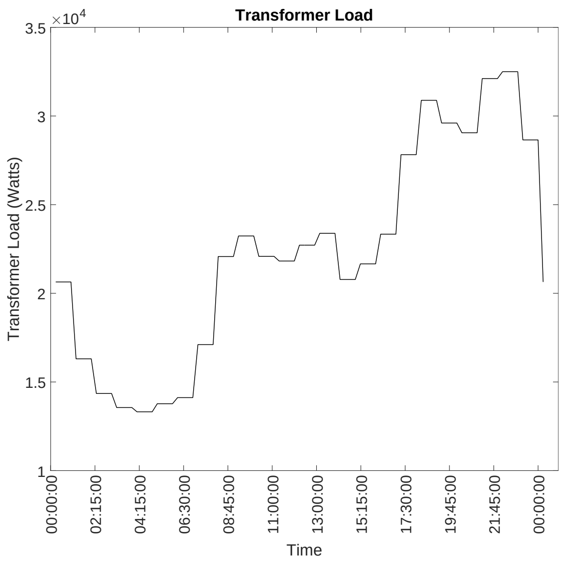


Figure 4.9 A Single Day's Electricity Usage.

Chapter 5

Evaluation

No experiment is ever a complete failure. It can always be used as a bad example.

Paul Dickson

This chapter presents the evaluation of Parallel Transfer Learning (PTL). It was done using three simulators and a range of environments. First, the simulators are introduced, then the requirements for the experiments are discussed. The first set of experiments evaluate the effects of parameter selection. Using the results of this, the effectiveness of PTL is evaluated. Following this, the Learnt Mapping and Self-Configuration components are evaluated. All experiments are run a minimum of 10 times and averaged. The bars shown represent the 95% confidence interval. For visual clarity bars are omitted on data that has already been presented.

5.1 Application Areas

PTL has a number of characteristics to be investigated, these are investigated in different applications and environments. This section provides a description of the environments and applications used with a particular focus on why these simulators were used. Chapter 4 discussed lower level concerns of the applications used such as parameter settings and state-space design.

5.1.1 Mountain Car

As outlined in Section 4 in the Mountain Car problem, an agent must learn to drive a car up one side of a valley [Knox et al., 2011]. Its engine is insufficient to do this, so it must first reverse up the opposite side to gain enough potential energy to complete the task. It aims to minimise the time required to do this, so better performance leads to less time required to complete the problem. The parameters used are based on those used in [Sutton and Barto, 1998]. They are detailed in Tables 4.1 and 4.2 in Chapter 4.

As this problem only requires a single agent, two independent instances are run simultaneously to use it to evaluate a multi-agent algorithm. The first instance uses Reinforcement Learning (RL) and functions as a source task, while the second uses PTL and receives information as a target task. Unless otherwise stated, these agents use a 'perfect' mapping. Each state-action pair is mapped to its exact correspondence. For clarity and to distinguish this from learnt mappings, this type of mapping is called Statically Mapped. This application is used as it allows the efficacy of PTL to be investigated without the added complexity of agents affecting the same environment. It is also a simple environment, there is no dynamism of non-stationarity for learning to navigate. The word episode will be used to describe a single attempt at reaching the goal. There will be exploration episodes in which the agent aims to gain knowledge and exploitation episodes in which it aims to perform as well as possible. All experiments are run 10 times and averaged.

5.1.2 Cart Pole

In the Cart Pole or Inverted Pendulum problem, a cart must move horizontally so as to balance a straight inflexible rod that is attached at one end but free to rotate [Sutton and Barto, 1998]. The parameters used are in Table 4.3 in Chapter 4. Its state variables and reward are described in Table 4.4 in Chapter 4. The update equations can also be found in Chapter 4. The agent's goal is to keep the pole balanced for as long as possible. This means that greater amounts of time spent balancing the pole, indicates better performance. As with the Mountain Car, Statically Mapped agents are used unless stated and two simultaneous simulations are run. Like the Mountain Car, it is a simple environment where agents do not affect each other through their actions. Two such applications are needed so that the mapping of information between heterogeneous tasks can be investigated. All experiments are run 10 times and averaged.

5.1.3 Smart Grid

GridLAB-D is used to run Smart Grid (SG) experiments [Chassin et al., 2008]. In these experiments, agents aim to achieve Residential Demand Response (RDR) in the SG. Unless otherwise stated, the scenario used for evaluation are at a residential neighbourhood scale. Each has an uncontrollable base load, which is the electricity drawn by non-controlled devices, that can not be rescheduled or influenced by agents. It varies over time in a realistic manner, as it is based on real-world data [Commission for Energy Regulation, 2011]. The data was collected from a smart meter trial in Ireland. It is representative of residential electricity usage in a developed country with a temperate climate without extremes of temperature. The real-world data was used to produce representative electricity demand profiles which are then provided to GridLAB-D and integrated into the scenarios. The simulations were run from 00:00 20th August to 00:00 1st September. This period in Ireland (typically) does not require heating or cooling, which are large, somewhat unpredictable electrical demands. This means that the base load can be learnt by the agents. The only major variability is due to the actions of agents. The agents control the charging of Electric Vehicles (EVs), which are a significant electrical load. The neighbourhood has 9 houses each of which has an EV. EVs are required to travel 60 miles (96.56 km) a day, the battery has a capacity of 30 kWh and an efficiency of 3 km/kWh, which means each day 32.19% of the battery is used. To recharge this, 6.89 hours of charging are required, as decisions are made every 15 minutes, this is effectively 7 hours. EVs are available for charging for 14 hours. There is variation in when these hours are between weekdays and weekends. Distributed W-Learning (DWL) without remote policies is used as the base algorithm. The use of RL-based algorithms such as DWL in the SG has shown promising results, but takes considerable time to perform well [Dusparic et al., 2015]. Each EV's agent implements three objectives:

(i) **User Objective**: the EV must be sufficiently charged for its journey when required. Battery charge is in the range $[0, 1]$, reward is $(currentCharge - 0.5) * 1000$.

(ii) **Transformer Load Objective**: the total load of all EVs and the base load must be kept as low as possible. Load is quantised into eight ranges evenly. If load is in the lower three bins, reward is $bin * 100$, otherwise it is $bin * -100$.

(iii) **Predicted Load Objective**: attempts to smooth the load out. If the load is predicted to rise, then off is preferred in the current time-step. Using the current load as a predictor of future load, the reward given if the predicted load is in the lower three bins is 500, in the upper three, it is -500, otherwise it is 0. The accuracy of the prediction only vary rarely affects the reward, as it takes reasonably significant errors to change a future state from one category to another.

Two metrics are used for evaluating performance in SG.

(1) **Discharged EVs -** If an agent is still discharging EVs completely during its exploitation phase, then it has not learnt sufficiently in the EV objective. This value should be low as there is a large punishment for completely discharging an EV, as the user could be extremely inconvenienced. This metric is preferable to average battery charge, as the way GridLAB-D updates battery charge would lead to inaccurate values. When the battery is being discharged (i.e., the EV is in use), the value for battery charge is only updated when the EV arrives home. This means that the leaving battery charge is over sampled. It also does not punish late charging, which other metrics may.

(2) **Mean Average Deviation (MAD) -** This encompasses the other two objectives. The ideal load shifting behaviour is a flat line in the third load bin[1]. Higher MAD

[1]According to the reward structure used here, in general a flat low load may not be desirable, the preferred shape is more application dependent.

means less load shifting has occurred, as generally the line is less close to its own average. Deviation from its average is typically due to peaks and valleys (see Figure 4.9 for example). It can also occur due to jitter when devices are regularly turned off (which the predicted load objective aims to reduce).

For all experiments, exploration is done by a guided method similar to Softmax without the exponential function because of bitwidth concerns. When exploitation occurs, the temperature is set to its minimum and the best action for a state is chosen. The GridLAB-D experiments have 20 days of exploration and 15 days of exploitation. These are averaged over 10 runs.

5.2 Requirements

This section provides the rationale for the selected experiments. Various claims have been made about the components of PTL throughout the previous chapters. The requirements outlined in Chapter 3 for the PTL component are repeated in Table 5.1. The scalability requirement for PTL is addressed by its agent-based nature and the use of neighbour sets. Neighbour sets allow the control of the number of agent that one agent must transfer to. This requirement is evaluated in the Effects of Scale experiments in Section 5.3.6. PTL's ability to operate in complex environments is evaluated in the SG. The effects of other agents in these experiments makes them dynamic. The variability of the base load the agents experience independently introduces dynamism. Some degree of non-stationarity is introduced by other agents. The Effects of Self-Configuration experiments in Section 5.3.5 directly investigates how PTL performs in the face of different types of non-stationary change. This section also evaluates how the Self-Configuration component reacts to an environment and thereby how PTL operates without designer input. All the experiments show PTL's on-line learning effects and it exploiting relatedness of tasks. There is an additional implicit requirement; that PTL accelerates learning. This is addressed through the remainder of this chapter, but particularly in the Effects of Learning Time experiments in Section 5.3.2.

The use of Transfer Learning (TL) as a base algorithm introduces the requirement that knowledge can be made mutually intelligible. This is achieved using learnt Inter-Task Mappings (ITMs), which are evaluated in Section 5.3.4. The intention for PTL to operate in real-world systems also introduces a requirement; that multiple objectives can be supported. This is investigated in Section 5.3.7.

Requirement	**Addressed by**
Efficient Use of Knowledge (A)	PTL component shares knowledge so it can benefit multiple agents
On-Line Improvement (B)	Knowledge constantly shared, so change can be reacted to
Adaptiveness (C)	Self-Configuration component can change how PTL operates if the environment changes
Heterogeneity Support (D)	Learnt mapping component translates knowledge between agents

Table 5.1 Characteristics of PTL to be Evaluated.

5.3 Experiments

This section evaluates the performance of PTL and its components in a variety of scenarios. It begins by investigating parameter selection, before evaluating the PTL component, Learnt Mapping component and the Self-Configuration component. Table 5.2 shows how these components and the requirements are addressed by the experiments

Component	**Requirements**	**Experiments**
PTL	Efficient Use of Knowledge (A)	Section 5.3.3, Section 5.3.6
	On-Line Improvement (B)	Section 5.3.2
Learnt Mapping	Heterogeneity Support (C)	Section 5.3.4, Section 5.3.7
Self-Configuration	Adaptiveness (D)	Section 5.3.5

Table 5.2 Experimental Roadmap.

5.3.1 Parameter Selection

To allow experiments' results to be more easily compared, a standard set of parameters are used where possible. This section evaluates various combinations of parameters to establish this standard set.

5.3.1.1 RL Parameters

The performance of RL-based algorithms is contingent on the selection of their parameters. In Q-Learning-based algorithms (as are used here), the parameters α and γ are used, where α controls how much a value changes in response to new information, and γ controls the weighting of future states (see Chapter 2 for further detail). For clarity of expression, in this section the γ's axis is called the X-axis, α's the Y-axis and performance is on the Z-axis.

Figure 5.1 shows the effect of α and γ on the Mountain Car. In the Mountain Car, the less time taken to reach the goal is better, so lower values indicate better performance. Lower values of γ are better than higher, with 0.1 performing particularly well regardless of the α value. This suggests that the value of future states is not that important. This is supported by the fact that the states in the Mountain Car are quite large. It takes several actions to traverse the distance of one state, so only a small fraction actually leave a state and hence have a different future value. Overall there is a slight trend for lower αs to be better, particularly at higher γs. This trend is reversed in the channel at $\gamma = 0.1$ where increasing α improves performance to a maximum at $\alpha = 1$, $\gamma = 0.1$. These values are used for all experiments using the Mountain Car.

Figure 5.2 is the effect of parameters on the Cart Pole. In the Cart Pole, the agent aims to maximise the time spent balancing the pole, so the longer the time spent exploiting, the better the performance is. In general, the Cart Pole is less sensitive to α and γ than the Mountain Car. Higher values of α and lower values of γ perform best. $\alpha = 0.9$, $\gamma = 0.2$ is the maximum performance and will be used.

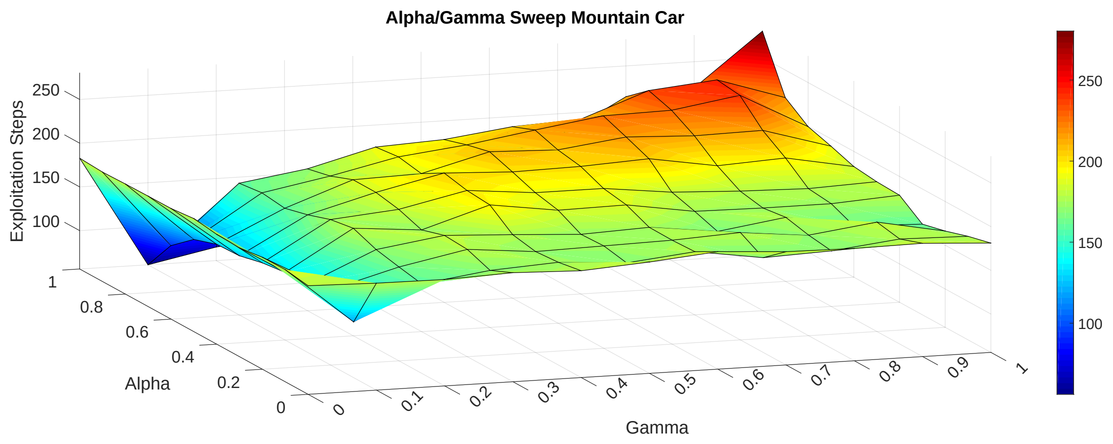


Figure 5.1 The effect of α and γ on Reinforcement Learning in the Mountain Car.

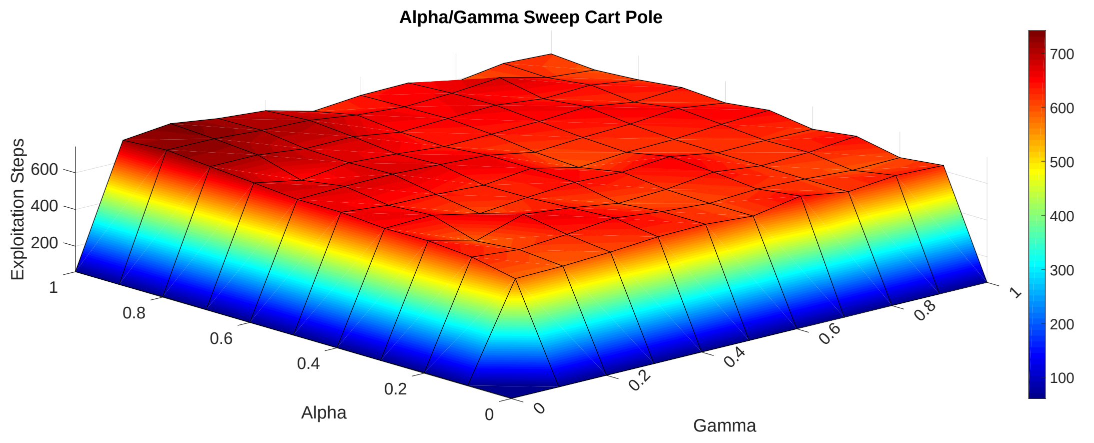


Figure 5.2 The Effect of α and γ on Reinforcement Learning in the Cart Pole.

Figure 5.3 shows the α and γ sweep for the SG problem. In this case there are multiple objectives, so performance is not evaluated on a single metric. The two metrics for the SG are combined. They are only be combined for visual clarity in the parameter sweep, as doing so can obscure information in other settings. The values for each combination of parameters is divided by the minimum for that metric across all parameters. This transforms them into relative metrics with a minimum value of 1. After this transformation the EV metric and the MAD are added, giving a minimum score of 2.

Obviously with this metric as with both of its constituents, lower values for performance are better performing combinations.

Generally, there are areas of good performance and areas of bad. Performance is reasonably stable on lines of constant α (with the exception of $\gamma= 0$). It varies more on lines of constant γ. This is because the α has much more impact on performance. α of 0 is worst, followed by the area in the range $\gamma= [0.4, 0.8]$. The area around $\alpha= 0.2, 0.3$ is best. This is a much lower learning rate than was found to be best in the other problems. This is because of the influence of other agents and the natural variability in the SG problem, single experiences become less representative of the converged value and therefore should be given less weighting, hence lower values of α. There is a slight trend for higher γs to improve performance, which is probably a feature of the problem. Rescheduling demand requires balancing of actions across longer time scales than other problems, so attaching greater weight to future states can be beneficial. The best particular combination of parameters is $\alpha= 0.3$, $\gamma= 0.7$. All agents use the same parameters.

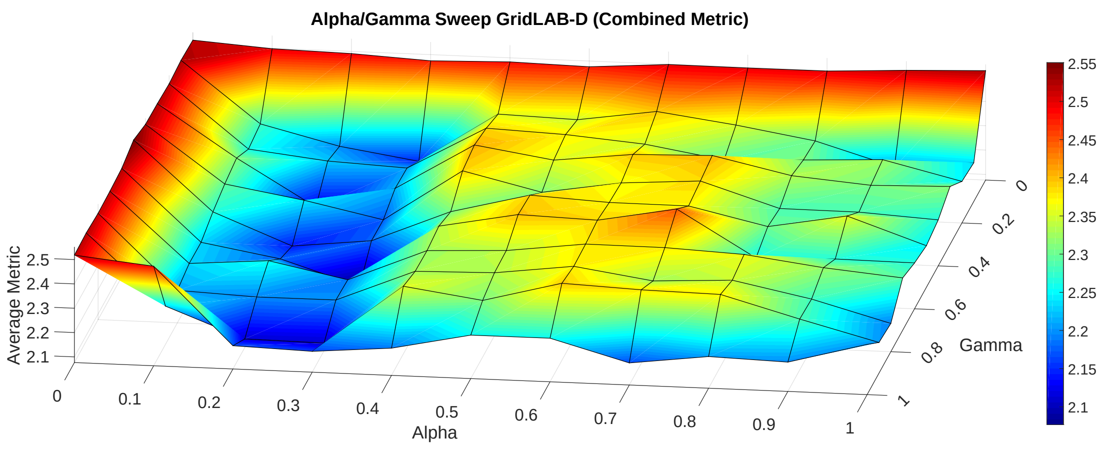


Figure 5.3 The Effect of α and γ on Reinforcement Learning in the Smart Grid.

5.3.1.2 PTL Parameters

As with RL, the selection of parameters for PTL is important. The evaluation of the Self-Configuration component investigates automatically selecting parameters and varying them on-line, but for most experiments, statically selected parameters are used. There are three parameters that need to be set for each of the applications; the selection method SM, the merge method MM and the transfer size TS. These parameters are evaluated in two tables for each application. One in which MM's number of confidence visits is varied and TS is kept constant at 5 while changing the selection method SM. In the other, the best merge method for each selection method is used and the transfer size TS is varied. In the variable merge method tables, the selected MM is marked with an asterisk. In the variable SM tables, an asterisk indicates the combination of parameters used for other experiments.

Table 5.3 shows the effects of varying MM on different SM in the Cart Pole. The difference column is the improvement in target over source (greater time indicates better performance). The table shows that PTL is reasonably sensitive to MM's number of confidence visits. Table 5.4 show that it is less sensitive to TS, with more performing well than poorly. It is difficult to pick out trends due to the dependence on the quality of source information relative to the maximum achievable performance. The maximum performance achieved by any run was 767.68 by a source task in Table 5.3 using Most Visits. As this is difficult to better even with much greater training, the selected parameters tend to be those that have brought about good target performance with poor source performance. The Cart Pole uses Converged Most Visits with 10 confidence visits and a $TS = 5$.

Table 5.5 shows the effects of varying the number of confidence visits used by the merge method in the Mountain Car. Only the Best State method of selection improves performance for all parameters. This is because 100 exploration episodes are used which gives RL enough time to learn a good policy most of the time. When this happens, if PTL is still affecting the value function it can skew the policy used in exploitation and reduce the final performance of RL. While most of PTL's benefits are early in the learning process (see Section 5.3.2), it is important to select parameters that do not

[2]The values for the target's runs with Greatest Change confidence visits 7 and 10 are actually identical, it is not a typo. They are composed of different individual values, only the averages are identical.

Method	**Confidence Visits**	**Source**	**Target**	**Difference**
Best State	1	639.11	632.81	−6.3
	*4	625.82	697.15	71.33
	7	681.06	623.69	−57.37
	10	624.14	647.92	23.78
Converged Most Visits	1	689.15	664.53	−24.62
	4	679.38	645.34	−34.04
	7	700.07	731.59	31.52
	*10	561.71	747.05	185.34
Greatest Change	1	713.21	671.11	−42.1
	*4	649.4	697.91	48.51
	7	703.98	512.77	−191.21
	10	665.13	512.77	−152.36
Many Visits	*1	596.94	649.26	52.32
	4	690.19	667.11	−23.08
	7	703.25	631.93	−71.32
	10	749.76	621.67	−128.09
Most Visits	1	729.32	688.49	−40.83
	*4	533.69	670.73	137.04
	7	767.68	674.59	−93.09
	10	649.52	686.37	36.85

Table 5.3 Effect of Varying MM on Parallel Transfer Learning in the Cart Pole[2].

impinge on final performance. For this reason Best States is used for Mountain Car experiments. Table 5.6 shows the effects of transfer size on selection. A $TS = 5$ and 1 confidence visit is best.

Table 5.7 shows the effect of MM on the SG problem. Rather than using the difference column as previously, RL (with 20 days exploration time) is included as a baseline, so that it is easier to compare the two metrics. For most parameter values, the EV metric is met and all EVs are charged. There is a slight trend for larger numbers of confidence visits to be better. This is because in the SG problem, there is greater variability, so more time is needed to learn the variability. This trend is mirrored in Table 5.8, where larger TS are better allowing for more samples to be shared. The Converged Most Visits method tends to perform poorly, as it disproportionately transfers

Method	TS	**Source**	**Target**	**Difference**
Best State	*1	603.15	702.46	99.31
	5	625.82	697.15	71.33
	10	686.96	668.03	−18.93
	20	584.92	642.14	57.22
Converged Most Visits	1	711.38	699.22	−12.16
	*5	561.71	747.05	185.34
	10	688.25	578.35	−109.9
	20	737.66	424.15	−313.51
Greatest Change	1	669.57	620.08	−49.49
	5	665.13	512.77	−152.36
	10	610.59	644.93	34.34
	*20	644.26	711.03	66.77
Many Visits	*1	523.95	708.77	184.82
	5	596.94	649.26	52.32
	10	650.24	652.2	1.96
	20	698.11	683.94	−14.17
Most Visits	1	675.94	757.66	81.72
	5	729.32	688.49	−40.83
	*10	570.83	767.59	196.76
	20	704.34	743.63	39.29

Table 5.4 Effects of Varying TS on Parallel Transfer Learning in the Cart Pole.

states the agents experience when they arrive home with a discharged or near-discharged battery. These states are frequently visited but not beneficial, which means that much of the capacity for transfer is being used on states where the target will have learnt the same thing. The parameters selected were Best States with a TS= 20 and 10 confidence visits.

Method	**Confidence Visits**	**Source**	**Target**	**Difference**
Best States	*1	569.45	382.29	187.16
	4	616.59	511.93	104.66
	7	476.96	451.52	25.44
	10	529.69	526.05	3.64
Converged Most Visits	1	524.04	525.52	-1.48
	*4	660.17	397.79	262.38
	7	401.83	514.88	-113.05
	10	527.03	491.44	35.59
Greatest Change	1	416.9	514.75	-97.85
	*4	475.32	483.81	-8.49
	7	518.3	539.8	-21.5
	10	500.17	492.49	7.68
Many Visits	1	496.63	502.91	-6.28
	4	418.08	438.75	-20.67
	7	534.75	511.33	23.42
	*10	508.7	444.08	64.62
Most Visits	1	438.3	514.18	-75.88
	4	364.85	441.4	-76.55
	*7	472.09	523.09	-51
	10	403.16	559.53	-156.37

Table 5.5 Effect of Varying MM on Parallel Transfer Learning in the Mountain Car.

Method	TS	**Source**	**Target**	**Difference**
Best States	1	555.27	551.56	3.71
	*5	569.45	382.29	187.16
	10	447.93	433.38	14.55
	20	563.23	483.44	79.79
Converged Most Visits	1	452.31	528.98	−76.67
	5	401.83	514.88	−113.05
	*10	498.97	489.06	9.91
	20	476.63	562.5	−85.87
Greatest Change	1	446.93	549.34	−102.41
	5	475.32	483.81	−8.49
	10	423.2	572.18	−148.98
	*20	536.1	461.77	74.33
Many Visits	1	417.17	577.64	−160.47
	5	418.08	438.75	−20.67
	*10	452.96	444.32	8.64
	20	454.91	555.9	−100.99
Most Visits	1	523.13	687.95	−164.82
	5	403.16	559.53	−156.37
	*10	527.52	467.65	59.87
	20	472.53	525.28	−52.75

Table 5.6 Effects of Varying TS on Parallel Transfer Learning in the Mountain Car.

Method	**Confidence Visits**	**EV Discharges**	**MAD**
RL		0.3889	8596.6
Best States	1	0	7142.5
	4	0	6922.1
	7	0	7199.2
	*10	0	6825.9
Converged Most Visits	1	0	7388
	4	0	7565
	7	0	7461.8
	*10	0	7198.4
Greatest Change	1	2.7778	5901.1
	4	0.3	7082.1
	7	0	7444.6
	*10	0	7234.4
Many Visits	*1	0	7105.7
	4	0	7312.6
	7	2.7444	5943.2
	10	0	7359.4
Most Visits	1	0	7411.7
	4	0.0111	7559.7
	*7	0	7288.2
	10	2.8111	5992.5

Table 5.7 Effect of Varying MM on Parallel Transfer Learning in Smart Grid.

Method	TS	**EV Discharges**	**MAD**
RL		0.3889	8596.6
Best States	1	0	7382.6
	5	0	7199.2
	10	0	7466
	*20	0	7182.7
Converged Most Visits	1	0.0222	7058.5
	*5	0	7388
	10	0.0111	7306.6
	20	2.6	5961.6
Greatest Change	1	2.7	5913.9
	5	0	7444.6
	*10	0	7243.5
	20	0	7431.4
Many Visits	1	0	7285.8
	5	0	7359.4
	10	0	7372.8
	*20	0	7155.7
Most Visits	1	0	7359.8
	5	0	7411.7
	*10	0	7165.2
	20	0	7248.3

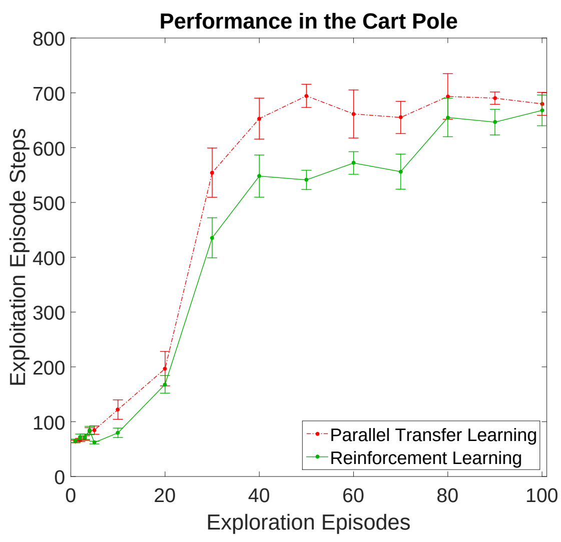
Table 5.8 Effects of Varying TS on Parallel Transfer Learning in Smart Grid.

5.3.2 Effects of Learning Time

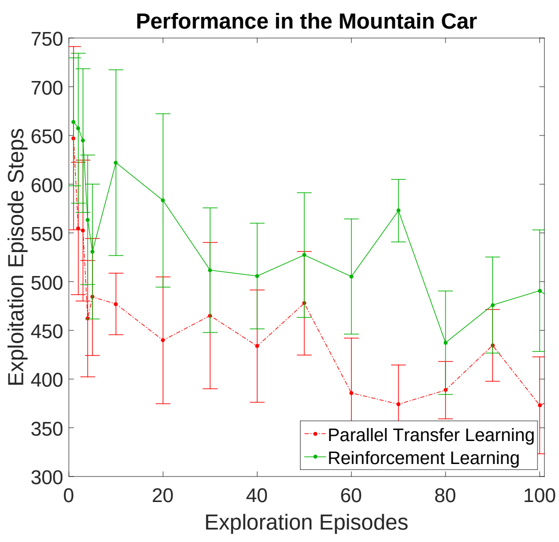
This experiment investigates if PTL is of any benefit while agents are still learning. RL for both accelerated learning and baseline used the same reward and parameters (described previously). Transfer Size TS was set to 20, the Best States scheme was used for the SG experiments. Merging was done using the adaptive method described in Chapter 3 with the number of confidence visits set to 10. These parameters where chosen based on testing combinations of settings (described previously), however they can also be selected by intuition. In this scenario the agents are being rewarded based on their performance, so they will see more benefit in transferring information about performance-critical states, which Best States does. As it takes several experiences of a state for a representative value to be developed, the merge parameter needs to be sufficiently large to allow transferred knowledge to be incorporated until local knowledge alone is sufficient. The experiments were run ten times and averaged.

In the Cart Pole, Transfer Size TS was set to 5 and the Most Converged selection method was used. Again, this was selected by evaluating the range of possibilities, but can be determined based on the problem. In the Cart Pole, it takes a reasonable amount of learning for performance to increase significantly (note the 'S' shape in Figure 5.4a). This is because a single poor action can make the pole unstable and cause it to fall, ending the episode and preventing further learning. As a result, information transferred should be much more accurate, to prevent misleading the target into taking a poor action. Merging was probabilistic, 50% of the time information was merged, the rest of the time it was discarded. As probabilistic merging was used, transfer continued regardless of the amount of learning time, allowing PTL to affect performance regardless of how much training is done. The Mountain Car used the same TS, but MM was set to 1 and the Best States scheme was used. The Mountain Car is much less sensitive to poor actions, so can share less reliable information. The values are much less variable, so a single sample is more representative, hence 1 confidence visit.

Figure 5.4 shows the Cart Pole and Mountain Car's results. The horizontal axis is the number of episodes that the agent had to explore, the vertical axis is its performance



(a) Effect of Learning Time on Cart Pole.



(b) Effect of Learning Time on Mountain Car.

Figure 5.4 Learning Time's Impact on Performance.

when it used the knowledge learnt. In the Cart Pole the vertical axis is how long it was able to balance the pole, higher values indicate better performance. For the Mountain Car, the vertical axis is how long it took to successfully climb the mountain to its goal. Lower values are better, as they indicate the mountain has been climbed more quickly. To provide source information to PTL in an inherently single agent problems, two simulations were run simultaneously. The first used RL and was the source for the second using PTL.

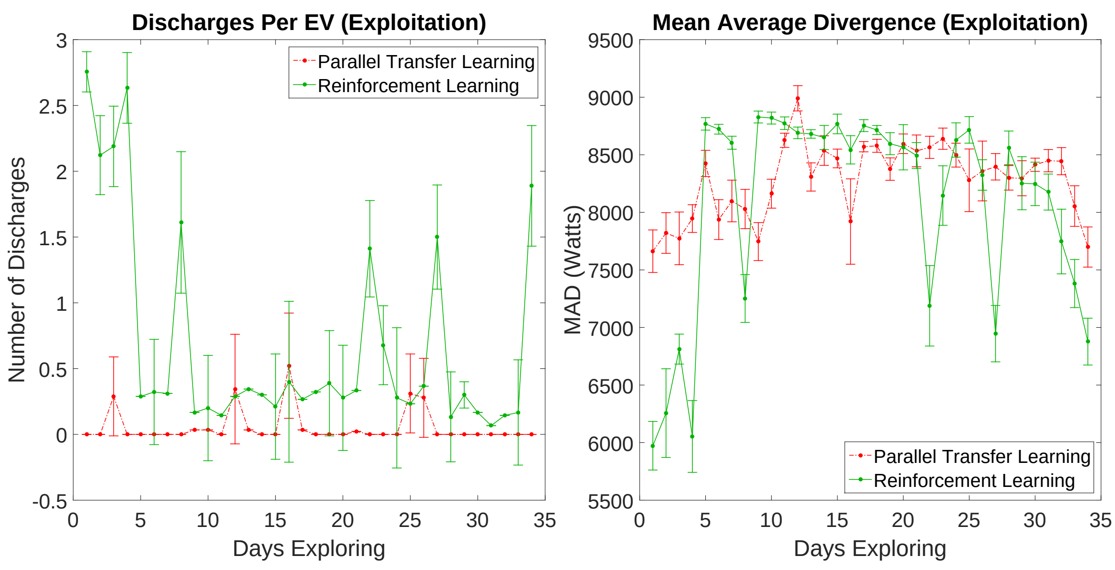
Figure 5.4a shows the performance of the Cart Pole in which the agent aims to balance the pole for as long as possible, so higher numbers of steps are better. When there is very few training episodes (< 10), little has been learnt, so there is no good knowledge to transfer. This causes PTL's performance to be similar or slightly worse than RL's. Once some knowledge is acquired and performance begins to improve, PTL outperforms RL for the same amount of training (two-tailed P value $= 0.0376$ at 30 learning episodes); it is using knowledge more efficiently. PTL reaches its asymptotic performance after 50 exploration episodes, while it takes RL an additional 50 for its performance to reach the same level. PTL reaches performance levels comparable to RL's final level (after 100 episodes) after 40 exploration episodes. This is an improvement of 60% over RL.

Figure 5.4b shows agent performance in the Mountain Car problem. Fewer steps are better as the agent wants to minimise the time to solve the problem. PTL's performance benefits in the Mountain Car occur earlier than in the Cart Pole, the gap between RL and PTL is significant after 10 exploration episodes (two-tailed P value = 0.0117 at 10 learning episodes). RL's performance plateaus after 90 episodes[3], while PTL's performance stabilises after 60. PTL's final performance is better than RL's, as well as getting to it more rapidly. PTL is able to achieve the same performance as RL's final performance (90 episode performance) after only 10 exploration episodes which is 11.11% of the time. The final performance is improved by 27% after 150 exploration episodes.

It is interesting to see the difference in the rate at which knowledge is built up to be used by PTL. In the Cart Pole, a relatively large amount of experience is needed before PTL becomes effective (the steep climb at beginning at 20 steps), in the Mountain Car it takes considerably less. When the Mountain Car agent is performing badly it takes a long time and generates a lot of experiences ($\sim$ 650 steps) to learn from, the Cart Pole's poor performance limits its training experiences ($<$ 100 steps). This indicates that the quality of the whole solution learnt is not too important for PTL to be effective, more that subsections of it are reusable and representative. This has implications for non-episodic tasks. In non-episodic tasks (like control in the SG), an agent generates many experiences regardless of performance, so there should be benefits in them from PTL.

Figure 5.5 shows the results of different learning times for the SG Scenario. Figure 5.5a shows the average number of times each EV completely discharges over a 3 day sample at the end of the run (during exploitation). Figure 5.5b shows the MAD which encompasses the other objectives.

[3]The spike in the PTL data at 90 exploration episodes is caused by two failures of the agent to climb the mountain in one experiment. For these it was given 5000 steps and the run ended. Removing this experiment and recalculating the average yields 389 steps.



(a) EV Objective Results. (b) Transformer and Predicted Results.

Figure 5.5 Learning Time's Impact on Performance.

Regardless of the amount of training done, PTL performs better according to the EV objective. It maintains fewer EV discharges than RL alone in all but four of the experiments (in which it is very close). A better MAD is maintained, particularly early on (two-tailed P value = 0.013 at 3 learning episodes). This improvement decays as PTL stops affecting the process due to the expiry of the merge parameter. After this parameter expires, in the experiments with longer learning times, RL keeps affecting the value function and thereby reduces the MAD back to RL's own level. RL slowly improves the MAD over time as it learns in the Transformer and Predicted Load objectives. The occasional sharp drops in the MAD correspond to times when EVs are not being charged enough. As less power is being used in these instances, it is easier to achieve load shifting and there are corresponding drops in the MAD. They are more frequent when RL is insufficiently trained and only occur in RL not PTL. This is because PTL is learning the constraints required by EV objective (sufficient charge required) more quickly, it can then adjust when it charges to better suit the other objectives.

The requirement for on-line improvement (B) was necessary to leverage the relatedness of tasks in a Multi-Agent System (MAS). This section has shown that PTL can reuse knowledge that is learnt on-line to improve performance. It can improve both performance from a specific amount of training and final asymptotic performance.

5.3.3 Exploring Different Areas of the State-Space

This experiment investigates the claim that PTL is particularly beneficial when agents explore different parts of the state-space. To do this, agents must be forced to experience different areas of the state-space. It is not practical to do this in most applications, as preventing an agent entering a state will impact greatly on performance as well as preventing the correct value function being learnt. To allow agents be forced into different areas of the state-space without affecting their performance, each area of the state-space must allow the agent to perform identically to normal. This is achieved by creating a new state-space which contains two copies of the old state-space, which are distinguished from each other using an additional parameter; colour. For example, State A in the original state-space will become State A-red and State A-blue. Half of the agents will explore the blue half of the state-space, the others the red half. After 20 days (the normal amount of exploration), the agents will be moved to the other half of the state-space where they will exploit. If PTL is transferring knowledge well then the agents learning in one colour will provide knowledge to those in the other colour as well as those in their own colour, which will allow good performance to be achieved in the previously unexplored area of the state-space.

Method	**EV Discharges**	**MAD**
RL (Uncoloured)	0.3889 ± 0.3913	8596.5 ± 475
RL (Coloured)	2.8889 ± 0.124	5915.7 ± 51.57
PTL (Uncoloured)	0	8375.7 ± 100.2
PTL (Coloured)	0	6757.5 ± 156.4

Table 5.9 Effects of Exploring Different Areas of the State-Space.

Table 5.9 shows the results of this experiment. RL and PTL results from uncoloured experiments are included to provide some context for the performance. As would be expected, the performance of RL drops significantly when colour is introduced. When RL exploits in an unexplored area of the state-space, the value function is always 0, so action selection is essentially random. This leads to EVs not being sufficiently charged. It does achieve some load shifting however, as the charging that is done is uniformly distributed by the random process of action selection. This, in conjunction with less electricity being used to charge EVs, causes the drop in MAD. PTL is able to sufficiently charge all EVs when the colour parameter is introduced, which shows the benefit of exploring different areas of the state-space. In this case, the agents were able to perform well in states they had never visited before. Interestingly, the colour parameter actually improves performance according to the MAD metric. This is because the agents' behaviour is learnt solely from transferred knowledge, so charging is more distributed. Normally, each agent learns a series of actions to take which lead to its EV being charged. Each agent's sequence will differ a little, but will be broadly similar. Combining sections of these sequences together through PTL, leads to new sequences that are more varied than the 'normally' learnt set. The more varied the solutions, the more RDR agents can provide and hence, better performance according to the MAD. When an agent learns 'normally', it has a series of sequential choices, to charge or not. This means the probability of agents starting charging in the first few time-steps after arriving home is relativity high[4], this attracts reward and is reinforced. A better solution is for the agents to spread their collective charging evenly over the available time, this is difficult to achieve without explicit coordination. In this case, it can be approximated, as agents are told through PTL to charge in a reasonable distribution of transformer load levels. This occurs as PTL does not share whole policies, just sections of them. For example, one agent may receive information telling it to charge at low load, while another is told to charge at medium load. Due to the load profile, this means the former will charge in the early morning and the later agent will charge around midnight. This introduces a natural staggering of charging schedule. This is one of the situations in which PTL is preferable to TL even if there is sufficient source data available to use. The diversity

[4]While time is not explicitly encoded in the state-space, it is partially represented through the temporal variability of the base load and the lower level of charge when an EV arrives home.

produced by PTL's intermixing of partial solutions could potentially be produced by TL methods if multiple sources where used, but without the on-line validation that the solutions produced are good, many of these multi-source solutions would fail.

This section has shown that the efficient use knowledge (A) can be achieved by PTL. Performance can be improved in unexplored areas of the state-space through reusing knowledge learnt by other agents in a system.

5.3.4 Effects of Mapping

To evaluate the effectiveness of the Learnt Mapping component, the techniques used are first used to learn mappings between homogeneous tasks. This removes the added complexity of having to determine whether there is useful information to transfer. It is also easier as there are the same number of states in the source and the target. All of the states will have correspondences, so there should be no unmapped states.

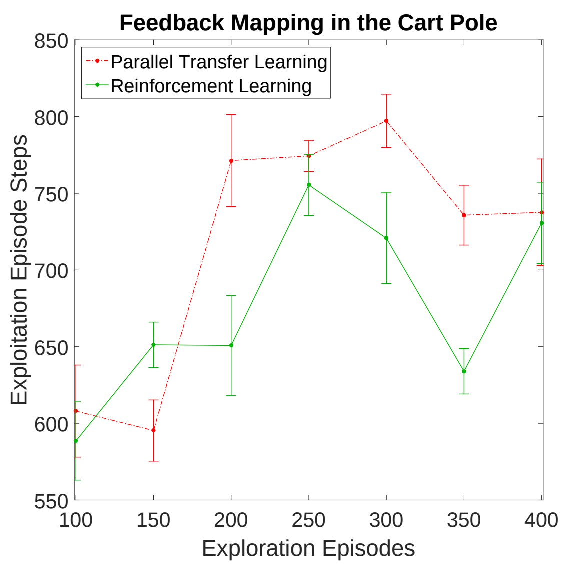
5.3.4.1 Homogeneous Tasks

This experiment is to test the feasibility of learning mappings on-line. The learning parameters for these experiments are similar to the previous ones, except that the ITM used here is no longer statically one to one. Previously, each state-action pair was mapped to its exact correspondence, for this one it is randomly populated. Over time, effective links are reinforced, while poor links are reallocated. The reallocation is done if the state is not merged. As the mapping must be learnt as well as knowledge, accelerating learning can not be as effective, so this experiment requires more exploration, hence the longer time scale in the figures.

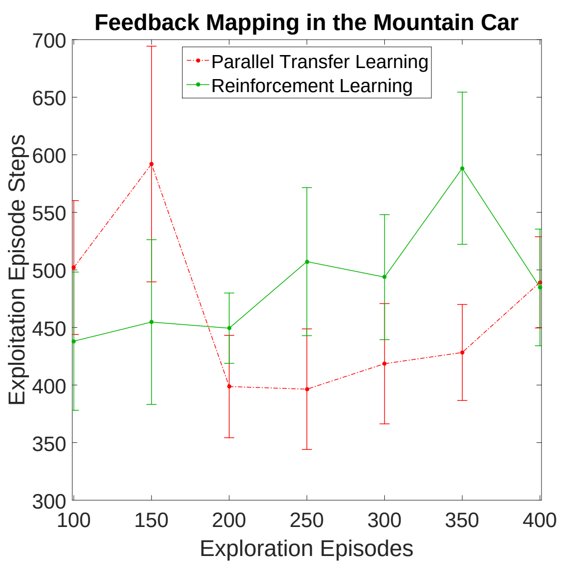
5.3.4.1.1 Vote-Based Mapping

Figure 5.6 shows the results of Vote-Based mapping in the single agent applications. The PTL parameters used are the same as those above, only the mapping varies.

Figure 5.6a shows the Cart Pole's results (where higher numbers are better). The RL line keeps increasing from the level established in the previous experiment as would be



(a) Vote-Based Mapping in the Cart Pole.

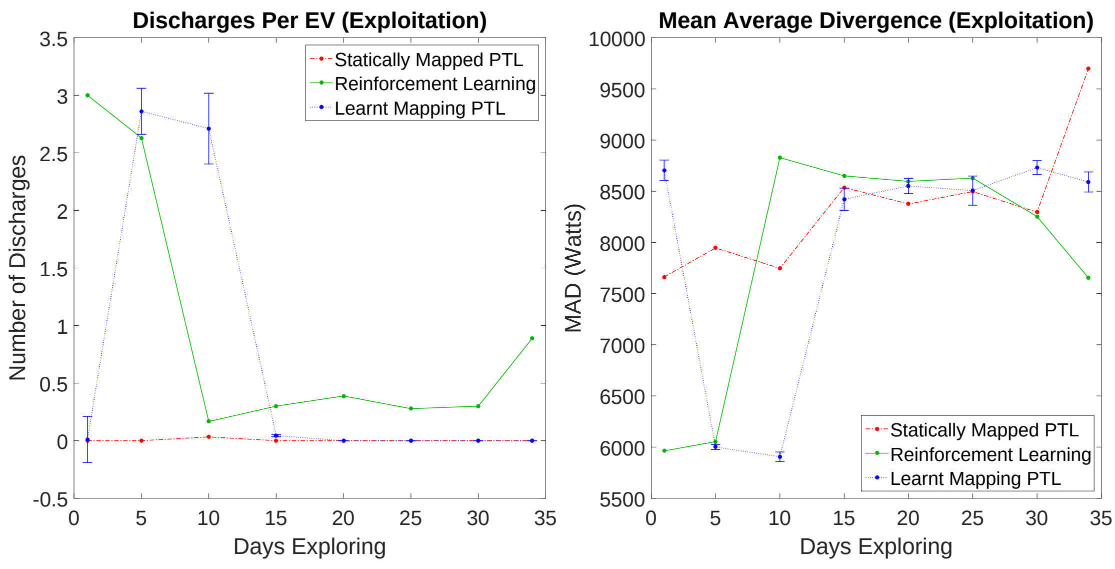


(b) Vote-Based Mapping in the Mountain Car.

Figure 5.6 Learning Time's Impact on Vote-Based Mapping Performance.

expected as it is unaffected by mapping. The PTL line is effected by mapping, and sees a reduction in performance from the non-mapped experiment. At 100 exploration steps, comparable performance to the one to one mapping version is not achieved, and there is no improvement over RL. Later, at 150 there is a significant improvement (two-tailed P value = 0.013), however, as after this, the mapping has had time to stabilise and performance settles to a reasonably static level (around 750).

Figure 5.6b (where lower numbers are better), there is a similar trend, after 200 exploration episodes performance stabilises. The mapping has been adjusted enough from random to perform well. PTL with mapping is not particularly sensitive to the quality of the source information when mapping is used. This can be seen at 150 and 350 when performance of one method is particularly poor without the other corresponding. The stabilised performance level shows that PTL does not achieve the same level of performance as it did without mapping (see Figure 5.4b). This is likely because the mapping learnt is not perfect. There are only a few 'correct' strands; ones where identical state-action pairs are matched. Most of the mapping strands produced are pairs with similar reward, in these cases mapped pairs are approximately functionally equivalent, rather than exact matches. This explains why in the Cart Pole, the extra time allowed improvement over the performance achieved in the non-mapped experiment, while the Mountain Car did not. The state-space in the Cart Pole is effectively symmetrical about the vertical axis, so there are more correspondences that are functionally equivalent than in the Mountain Car (which is not symmetrical). In the Cart Pole, the mapping learnt was sufficient to allow performance to be improved with significant extra time, while in the Mountain Car it was not.



(a) EV Objective Results. (b) Transformer and Predicted Results.

Figure 5.7 Learning Time's Impact on Vote-Based Mapping Performance.

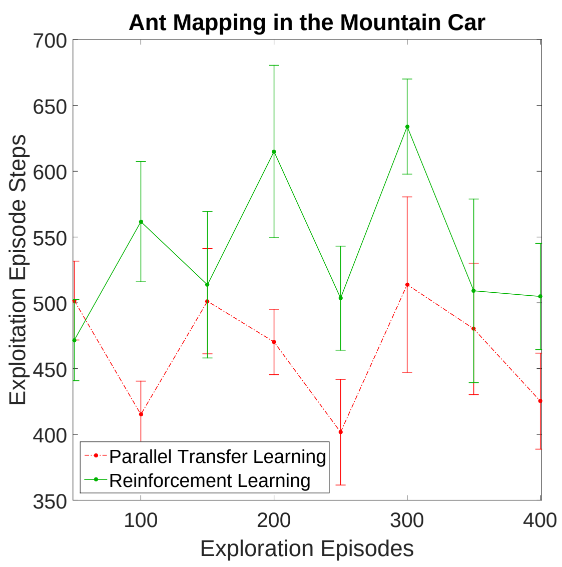
Figure 5.7 shows the performance of Vote-Based mapping in GridLAB-D. As each agent functions as both a source and a target, there is no natural baseline to use, so the RL and PTL results from the first experiment are used. The old PTL results are referred to as statically mapped PTL. This allows statically mapped PTL with a perfect mapping to be compared with the learnt mapping version. There are fewer learnt mapping experiments than statically mapped ones, so the statically mapped results for only the corresponding learning times are included.

As would be expected from previous mapping experiments, learning the mapping reduces performance with lower amounts of training. It performs poorly on one of the metrics until after it has had 15 days of training. After this the mapping is suitably effective and performance becomes comparable to the statically mapped PTL. PTL—regardless of how it is mapped—maintains noticeably better performance on the EV objective. This is because the method of transfer used is Best States which prioritises state-action pairs that receive high reward. Doing so causes charging to happen more frequently as it attracts more reward than not doing so. As a result, the charge action converges more quickly and performance is improved directly.

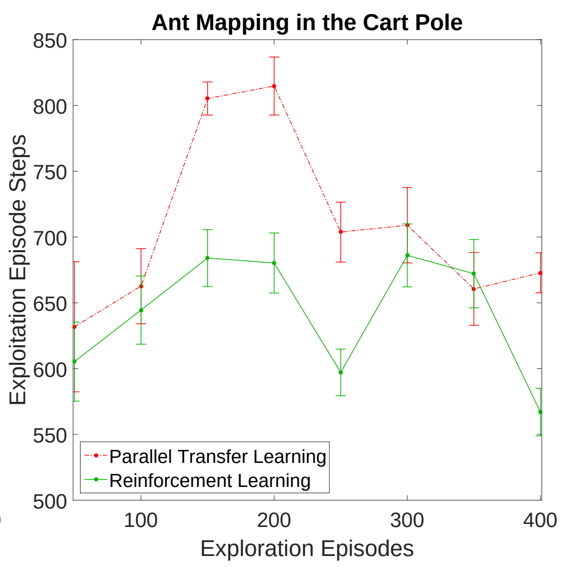
5.3.4.1.2 Ant-Based Mapping

Figure 5.8 shows the Ant Colony inspired on-line mappings. The ants attempt to find correspondences between reward at state-action pairs in the source and the target, if they do the strand between the corresponding state-action pairs is reinforced, otherwise it is remapped. Each time-step 100 ants are used, each tries one source and target. It is feasible to use more ants, but would require more computational resources (100 could plausibly be run on an embedded system). The threshold for a good correspondence was 1, if reward differs by less than that the strand of mapping was kept, otherwise it was reallocated.

In the Mountain Car (Figure 5.8b) where lower numbers indicate better performance, PTL performs better after 100 exploration episodes. Figure 5.8a illustrates the performance of the Cart Pole with the Ant-Based Mapping. In it PTL has had little effect, sometimes RL is better, others not and there is no relationship between the amount of exploration and performance. The performance remains variable throughout both experiments which indicates that the Ant-Based Mappings are less reliable. This is explained by the reward structure, only the terminal states have different reward to the other states, while their values differ significantly. The reason the Mountain Car performs better than RL when the mappings struggle to find correct source and target pairs, is because more of the environment is sampled in exploration as PTL actually reduces performance when training, so more samples are gained. If the reward received at each state differed more, it would be easier to find good correspondences.

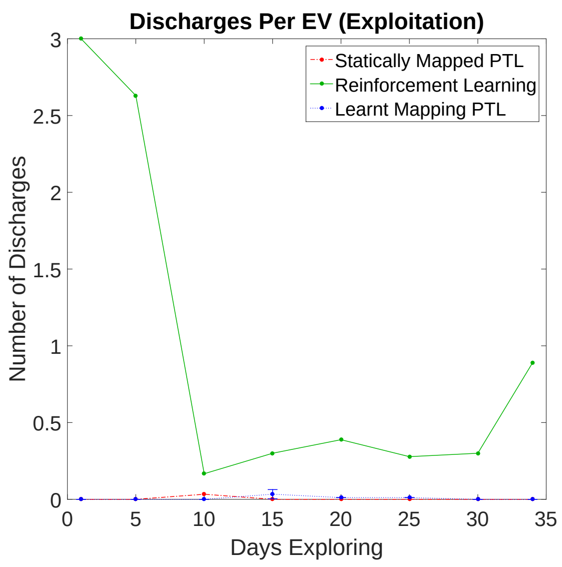


(a) Ant-Based Mapping in the Cart Pole.

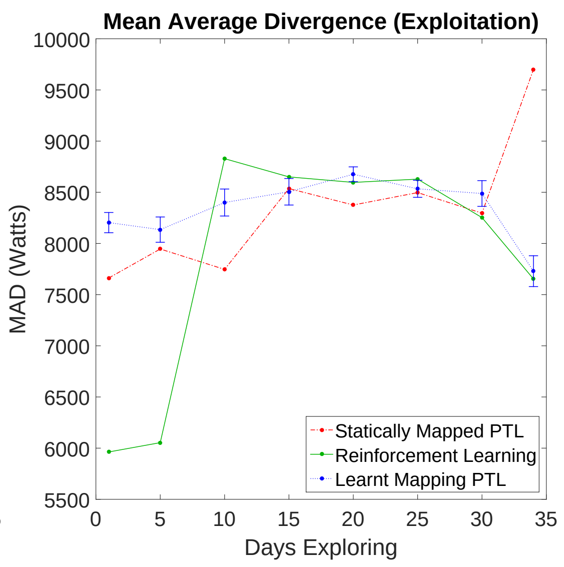


(b) Ant-Based Mapping in the Mountain Car.

Figure 5.8 Learning Time's Impact on Ant-Based Mapping Performance.



(a) EV Objective Results.



(b) Transformer and Predicted Results.

Figure 5.9 Learning Time's Impact on Ant-Based Mapping Performance.

The effect of Ant-Based Mapping in the SG is shown in Figure 5.9. In the EV objective (Figure 5.9a) the performance of mapped and statically mapped PTL is comparable and superior to RL. In Figure 5.9b, the performance in the other objectives is shown.

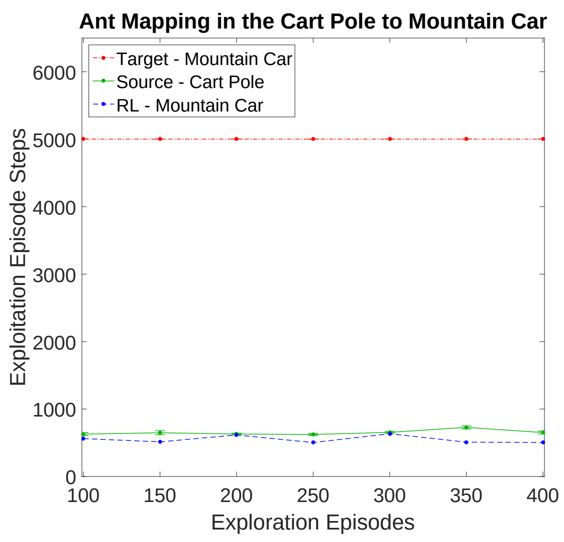
Interestingly, mappings performance is reasonably constant regardless of training time. This is because the objectives used in this experiment have smaller state-spaces than in the Cart Pole and Mountain Car. Smaller state-spaces mean that there is less potential source to target pairs for the mapping to choose from. In addition to this, the reward is more variable so each state-action pair is more distinctive and its correspondence is easier to find. The performance of the Ant-Based Mapping is better than statically mapped PTL, which is rather counter-intuitive, as the 'perfect' mapping would be expected to produce the best results. The fact that is does not is because of incorrectly mapped strands in the Ant-Mapped version. These strands introduce some variation in what the agents learn. This variation manifests itself as charging EVs at slightly different times, which in turn lowers the peaks and raises valleys and improves the MAD.

Selecting between the two methods of mapping, Vote-Based mapping and Ant-Based mapping, is difficult as they are effective in different problems. Ant-Based mapping works best when the states experience different reward to each other so that they can be uniquely identified (like in the SG). Vote-Based mapping works well in problems where much of the time is spent in a subset of states that are critical to performance. In these situations, Vote-Based mapping is able to find correspondences in the important states more quickly as its search is directed by the agent while Ant-Based mapping just finds correspondences randomly.

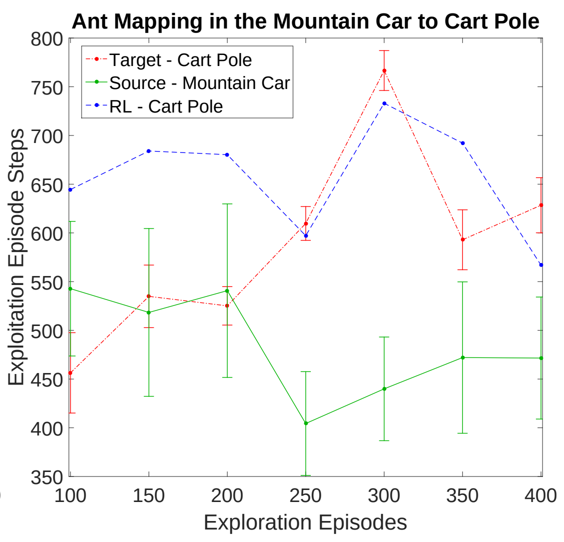
5.3.4.2 Heterogeneous Mapping

To evaluate how feasible on-line mapping between heterogeneous tasks is, the following experiments were run. The parameters were as used previously in Section 5.3.4.1.2, the homogeneous Ant-Based mapping experiments. Ant-Based mapping is used for this experiment, as it showed generally better performance than Vote-Based mapping. Additionally, as Ant-Based mapping checks multiple potential strands per time-step, if it does not work then there is no expectation that Vote-Based will, as it only tries one strand per time-step. Figure 5.10a shows using the Cart Pole as a source task for the Mountain Car. The blue line, like in previous experiments, is to provide a reference level of performance. It shows the performance of RL in the target problem, the Mountain Car. As the Mountain Car is a minimisation problem, lower numbers are better, so, ideally, the red line of PTL's performance would be below RL's. The fact that it is not, shows that there is a considerable amount of negative transfer occurring. PTL is severely impacting on performance. To keep the length of time taken to run experiments tractable, they are stopped after 5000 time-steps. In the case of the Mountain Car, this limit is where it is assumed that learning will never succeed, so the episode is finished. This is what happens here without exception, PTL is preventing the Mountain Car from finishing its task. For each run of the experiment a new mapping is made from a random start. This means that from 70 different ITMs which were changed over time, no useful information could be transferred. If there was some knowledge that could accelerate learning between the Cart Pole and the Mountain Car, some evidence would have been expected. This indicates that it is not the mapping preventing transfer, but no commonality in the knowledge that is being transferred. Effectively nothing could be transferred regardless of how it was mapped.

Figure 5.10b shows this experiment run the opposite way; from Mountain Car to Cart Pole. Again the blue line is providing a benchmark; it is RL's performance in the Cart Pole, in which higher numbers are better. The green line is the source task; the Mountain Car. As would be expected it improves its performance over time. The red line is the target Cart Pole, it shows improvement over time. It is particularly interesting that for three amounts of training it is able to out-perform RL, as this means that it has successfully mapped information from the Mountain Car to the Cart Pole. While it has not been able to achieve the same level of performance as when using a homogeneous source task, it has shown improvement with additional training. As the source's performance remains reasonably stable with additional training, the relativity large increase in target performance must be due to better mapping of source knowledge. The source has learnt the same thing in each run, but performance improves, so the only variable is the mapping.



(a) Mapping from the Cart Pole to Mountain Car.



(b) Mapping from the Mountain Car to Cart Pole.

Figure 5.10 Heterogeneous Mapping Performance.

This section has shown that the Learnt Mapping component can allow the heterogeneity requirement (D) to be met in certain circumstances. Generally, the effectiveness of mapping is contingent on how different the source and target tasks are, but the difference in performance between Cart Pole to Mountain Car and vice versa, shows that the representation plays an important role. Some ways of representing knowledge must be more conducive to having knowledge extracted, as if the Mountain Car can support learning in the Cart Pole, it stands to reason there is some commonality of knowledge between the tasks. Therefore, the representation of knowledge used in the Cart Pole must be in some way incomparable with the Mountain Car. Regardless of the representation of knowledge, even when agents are fully homogeneous (different application with different actions, states etc.) mapping can at least match RL's performance. This experiment was not run in the SG, as the complexity of the different electrical devices required to provide heterogeneity and different environment would be too much to expect an ITM to be able to account for.

More generally, the mapping of information between heterogeneous tasks will require careful task selection. While a heterogeneous mapping scheme can find a mapping from source to target, there is no guarantee that it will provide beneficial information. Between two arbitrary tasks there is not necessarily any commonality of knowledge, so regardless of how effective the mapping is, useful knowledge could not be transferred. There is no way to detect this potential for negative transfer a priori. The best heuristic known is to confine transfer to task that have some similarity e.g., running and walking share the same basic mechanics or Mountain Car and Cart Pole share a similar reward distribution across their state spaces.

5.3.5 Effects of Self-Configuration

To evaluate the effectiveness of the Self-Configuration component, changeable non-stationary environments are needed. Of the applications used, only the SG can be used for these experiments, as all of the agents share a common environment. If the agents are in separate environments—as with the Cart Pole and Mountain Car—then the system's resultant performance after a change can not be fairly assessed. This is because the two different environments necessary for making these problems multi-agent must be given exactly the same change at the same time. This can be unfair as one of the agents may be in a different part of the state-space from where the change occurred and hence will not experience it for some time. This delay in observing the change will affect the other agent's performance, as the other agent is not being supplied with information about the change. This could misrepresent the magnitude or extent of the change. The performance of a system that has experienced change is a function of all the agents in the systems' reaction to change. The agents' performance is interdependent.

To generate change that effects performance in the SG environment, the change needs to impact on all of the objectives. Changing the base load would affect the Predicted Load and Transformer Load objectives, but it would not affect the EV objective. The best way to effect all objectives is to change the amount of charge required by the EVs. To change the amount of charge required, the battery capacity can be increased. This effects the reward received in a state for the EV objective, as lower charge states become able to provide for the full daily journey. This changes the value function at each state significantly. It also impacts on the electrical objectives, as more load is placed on the transformer as the agent tries to reach the previously established ‘good’ states. The other change that can be made is to the length of the daily journey. Increasing this reduces the battery charge that EVs arrive home with, which directly effects the EV objective. The increased amount of charging needed affects the other objectives through the increased transformer load. When the daily journey is increased significantly, capacity increases are necessary to allow the EVs to meet their requirements.

When slow change is required, each day the daily journey increases by 2 miles (3.21 km) and every 5 days the capacity of the battery increases by 1 kWh. Slow change is change that has little impact on the reward received over short time periods, but continues until it impacts significantly. For fast change, which is significant over one time-step and continues for several, 6 miles (9.65 km) are added a day, while the capacity in increased by 3 kWh a day. Sharp change happens and stops. In it, only the daily journey is increased, the increase is 14 miles (22.53 km). The change starts (or happens in the case of sharp change) after 15 days of learning, which is 5 days before learning stops. This gives RL and PTL some time to learn in the changed environment before they stop generating new knowledge and exploit.

Method	**Confidence Visits**	TS	**EV Discharges**	**MAD**
RL			3 ± 0	5576.2 ± 154.22
Best States	1	4	3 ± 0	5970.8 ± 62.452
Converged Most Visits	7	1	3 ± 0	5888.9 ± 44.478
Greatest Change	7	1	3 ± 0	5988.3 ± 72.1
Many Visits	1	1	3 ± 0	6005.2 ± 67.311
Most Visits	4	1	3 ± 0	6045 ± 73.22
Adaptive Configuration			3 ± 0	6569.9 ± 45.34

Table 5.10 Performance of Parallel Transfer Learning with Slow Change.

Table 5.10 shows the effect of slow change on performance. The named PTL methods use a single configuration throughout the run and do not adapt to change. PTL was run with all parameters (similarly to the experiments in Section 5.3.1.2), only the best results are shown (although they were all similar). The continuous nature of the change prevents useful knowledge being learnt and as such there is nothing to transfer. Regardless of if an action is good or bad, its value is invalidated quite quickly with even slow change. Table 5.11 shows similar poor performance when fast change is happening. Again, this is because knowledge is invalidated too quickly for performance to improve.

Method	**Confidence Visits**	TS	**EV Discharges**	**MAD**
RL			3 ± 0	5720 ± 127.973
Best States	7	1	3 ± 0	6167.7 ± 84.62
Converged Most Visits	1	10	3 ± 0	6135.7 ± 40.176
Greatest Change	1	1	3 ± 0	6064.3 ± 58.4
Many Visits	1	1	3 ± 0	6052.1 ± 59.794
Most Visits	1	10	3 ± 0	5977.2 ± 59.52
Adaptive Configuration			3 ± 0	6467.6 ± 55.32

Table 5.11 Performance of Parallel Transfer Learning with Fast Change.

Table 5.12 shows how RL and PTL react to sharp change. RL fails to meet the EV objective, as it does with the other types of change. As it is using less electricity, it is easier to keep the MAD low. The methods of PTL which do not adapt to the change perform better than RL except for the Converged Most Visited method. The Converged Most Visited method keeps transferring the same states, so it does not share information in the changed states. Performance is better on the EV objective and similar in the MAD for the rest of the statically configured PTL methods. When the Self-Configuration component is used, PTL can adapt to change, resulting in much better performance in the EV metric. It is able to use the knowledge built up in the system in the short learning period after the change more efficiently and thereby, maintain its performance at

Method	**Confidence Visits**	TS	**EV Discharges**	**MAD**
RL			3 ± 0	5956 ± 45.372
Best States	7	1	1.8889 ± 0.213	6141.9 ± 59.18
Converged Most Visits	10	5	2.1 ± 0.326	6018.7 ± 73.601
Greatest Change	4	5	1.9333 ± 0.388	6110.3 ± 52.293
Many Visits	7	5	1.8556 ± 0.388	6034.4 ± 68.365
Most Visits	1	1	1.9889 ± 0.407	6001.4 ± 54
Adaptive Configuration			0.7111 ± 0.213	6707.2 ± 99.92

Table 5.12 Performance of Parallel Transfer Learning with Sharp Change.

an acceptable level. The Self-Configuration component was able to improve performance here as there was time for some knowledge about the change in the environment to be built up, which was not true with the other types of change.

This experiment has shown that PTL's Self-Configuration component can react to change in the environment as long as the change stops and some knowledge can be built up following the change. In terms of PTL's requirements, the adaptiveness (C) requirement is partially met. If the environment stabilises, then PTL can adapt to change in the environment. If it is constantly changing, then it can not. However, if knowledge could be taken from another source with information about the change, then perhaps performance could be improved in constantly changing environments. This is an interesting possibility for future work and is discussed further in Section 6.

5.3.6 Effects of Scale

For PTL to be useful in real-world MASs, it needs to be scalable. The distributed nature of the algorithm means that the computational, memory and communications costs only vary with the size of the set of neighbours. More neighbours requires more selection of data, more ITMs and more messages. As observed above, the overall quality of a solution is not overly important when using PTL, so there is no need to ensure every agent has the best performing agent as a neighbour. If performance was more strongly linked to overall solution quality, then all agents would need to be neighbours with each other to ensure that a good knowledge source was available to all. As it is not, neighbour sets can be kept small. This means that regardless of the number of agents in a MAS, PTL requires the same amount of work per agent (as long as neighbour set size is constant). The neighbour sets size used in this experiment (and all others) is 6.

Method	EV Discharges	MAD
RL	0.2649 ± 0.114	100670 ± 169.995
PTL	0 ± 0	84202 ± 950.04

Table 5.13 Effects of Scale on PTL.

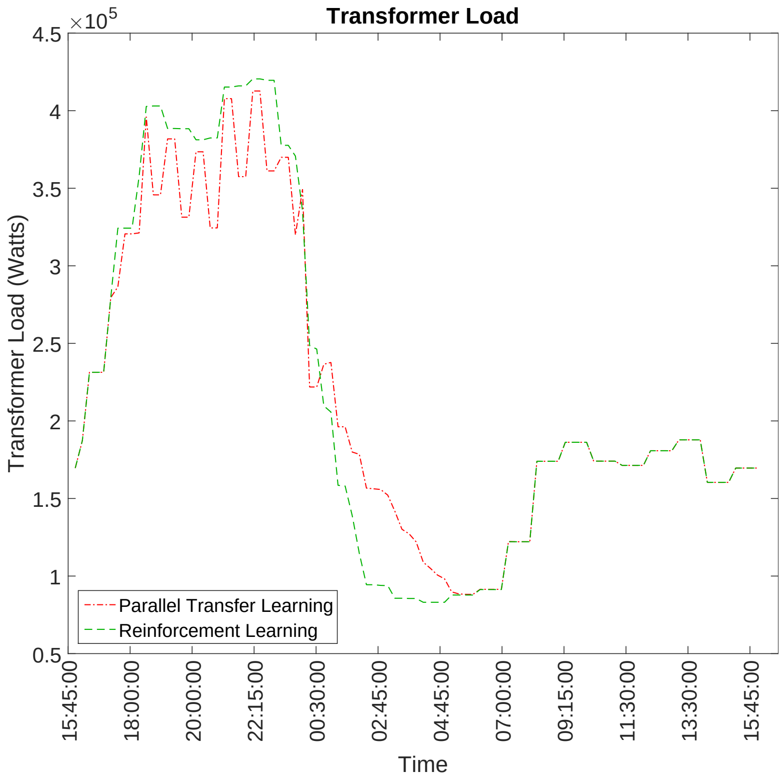


Figure 5.11 Transformer Load for 90 Electric Vehicles.

Table 5.13 shows the effect of running RL and PTL for 90 houses in the SG. The rest of the experiment is the same, each has a EV and base load, learning parameters are as before, as are the objectives. RL performs slightly better than it did in the 9 EVs experiments on the EV metric (see Figure 5.5a at 20 days exploration). This is because the effect of individual agents on the transformer load is lessened, which means that the sequence of states experienced is more predictable and a better policy can be learnt. The MAD can not be compared across scales in the same way, as the number of agents cause the magnitude of the deviation to be much greater at scale. It can however be compared to PTL's MAD. PTL performs better according to it, indicating that more load shifting is occurring. This can be seen additionally in Figure 5.11. In it, PTL is shifting load from the peak on the left of the figure to the valley in the centre. After this point, EVs start leaving for work, so they must be charged by then and after it are no longer available to charge.

This experiment has shown that the efficient use of knowledge (A) can occur at scale when using knowledge from small groups of agents, indicating that knowledge in a system can be shared between agents in a scalable way which does not require all agents to communicate with each other.

5.3.7 Multi-Objective Transfer

This experiment investigates the effect of transferring inter-objective information (W-Values), the previous experiments transferred information pertaining to one objective (Q-Values). It aims to determine if there is any utility to be gained by sharing the inter-objective information of states. The inter-objective information for a state represents the importance of that objectives action suggestion being obeyed. For these experiments, PTL did not transfer any single objective information (Q-Values). This means that PTL affects the equilibrium developed by RL and not the performance according to a single objective as previously. The evaluation of the effects of parameters is similar to that in Section 5.3.1.2. First, different merge methods MM are evaluated and then the best of them (marked with an asterisk) are used to evaluate transfer size TS. To provide a base line, RL's performance is provided. GridLAB-D is used to run these experiments. The same policies are used as detailed previously. Each agent's inter-objective information for a particular policy can be transferred to its neighbours corresponding policies.

The methods used are as follows:

- **Highest W-Value** selects the state with the highest W-Value. This is the state with the highest priority either due to its importance or not winning in inter-objective arbitration. The selection criterion is $\{T_w \mid \max W(S_i)\}$.
- **Random W-Value** selects a random state. This will tend to select a more diverse set of states. The selection criterion is $\{T_w \mid Random(S_i)\}$.
- **Inverted W-Value** selects the state with the highest W-Value. It then multiplies this by -1. This effectively forces the target agent to select the opposite action to the source at this state as it will cause one of the other objectives to win. The aim of this is to provide better load shifting by coordinating actions between the sources and targets. The selection criterion is $\{T_w \mid (\max W(S_i)) * -1\}$.

The merge methods used are only received information and a weighted linear combination with confidence visits set to 10. When using Inverted selection, only received information will be used as combining it would reduce the intended coordinating effect.

Method	MM	**EV Discharges**	**MAD**
RL		0.3889 ± 0.3913	8596.5 ± 475
Highest W-Value	*Received Only	2.7333 ± 0	5943.7 ± 41.252
	Adaptive 10	2.8272 ± 0.1	5882.9 ± 31.269
Random W-Value	*Received Only	2.7444 ± 0.133	6006.9 ± 58.477
	Adaptive 10	2.7556 ± 0.11	5954.7 ± 49.231

Table 5.14 Effects of Varying MM on Transferring Inter-Objective Information.

Table 5.14 shows the effect of varying the merge method MM. A transfer size of 5 was used. None of the methods used are able to achieve RL's performance. They are all similar to each other, indicating there is little difference between the different methods of section for inter-objective information. The worsening of performance according to the EV metric from RL's is due to the equilibrium between objectives being upset. For both selection methods used, Received Only will be used to evaluate the effects of TS.

Method	TS	**EV Discharges**	**MAD**
RL		0.3889 ± 0.3913	8596.5 ± 475
Highest W-Value	*1	2.6333 ± 0.166	5914.3 ± 37.821
	5	2.7333 ± 0	5943.7 ± 41.252
	10	2.8 ± 0	5922.8 ± 34.09
	20	2.7333 ± 0	5978.6 ± 53.897
Random W-Value	1	2.7111 ± 0.163	5917.1 ± 46.673
	5	2.7444 ± 0.133	6006.9 ± 58.477
	10	$2. \pm 0.133$	5907.1 ± 62.454
	20	2.7 ± 0.213	6014.1 ± 41.862
Inverted W-Value	1	2.7222 ± 0.134	5938.6 ± 38.83
	5	2.7111 ± 0.153	5928.3 ± 61.983
	10	2.8222 ± 0.1	5908.2 ± 31.27
	20	2.7667 ± 0	5846 ± 25.115

Table 5.15 Effects of Varying TS on Parallel Transfer Learning in Smart Grid.

Table 5.15 shows how TS impacts on the performance of PTL. Again none of the methods perform well. There is little difference between any of the sizes. This indicates that altering which objective is selected in a particular state worsens overall performance. In this application, one objective tries to charge most the time (EV objective) and the others prefer not to charge. Shifting the balance between these policies prevents them both from being effectively interleaved. The transfer of inter-objective information may be more effective in applications where multiple policies can be satisfied simultaneously rather than having to establish a trade-off. In this case, there is little flexibility in how the objectives interact. The lack of flexibility means any improvement of one objective comes at the expense of another. If there was more flexibility, objectives could be partially satisfied to benefit other objectives. In other words, rather than just one effective equilibrium between objectives, there are likely to be more if the objectives are less mutually exclusive.

5.3.7.1 Different Policy Configuration

To test whether homogeneity of objectives has any effect on transfer, an experiment is run where the agents in a MAS have different sets of objectives. The normal SG scenario is used with modifications to the objectives implemented by agents. Half of the agents will not use the Predicted Load objective. This means the inter-objective information for the other two objectives will change. The agents will transfer inter-objective information agnostically of this. The same metrics will be used.

Method	EV Discharges	MAD
RL	0.3889 ± 0.3913	8596.5 ± 475
PTL for Same Objectives	2.6333 ± 0.166	5914.3 ± 37.821
Different Policies	2.7 ± 0.1633	5985.7 ± 49.895

Table 5.16 Effects of Different Objectives on Parallel Transfer Learning in Smart Grid.

The Table 5.16 shows the results of the different objective experiment with the same objective and RL experiments as base lines. As with the other multi-objective experiment, there is little change in performance. It is worse than RL's performance alone. This reinforces the reasoning for the poor performance of transfer of inter-objective information in the previous experiment as well. In this experiment there was greater variability in the actual values transferred (due to the heterogeneity in the system), yet the performance level is reasonably stable. This is because few of the values actually get used without being overwritten by another agent or having a transfer go to a different objective's value for the same state. If transfers go to each of the objective's values for a particular state, then the original equilibrium between objectives is destroyed. In this case the action selected at that state becomes effectively random which leads to poor performance.

The transfer of inter-objective information is not effective in this application. The balance that must be achieved between objectives is too fragile for the extra variability induced by inter-objective transfer to improve performance. It could be useful in applications where there is greater scope for collaboration between objectives. In this application either the EV objective is obeyed and charging occurs or one of the other objectives is obeyed and no charging happens. This is an oversimplification of what is actually learnt, but it illustrates the problem. If there was some action which could partially satisfy multiple objectives, then it is plausible that PTL transferring inter-objective information could find a different equilibrium between objectives, rather than modifying the single effective equilibrium found by RL.

The experiments in this section were designed to evaluate if inter-objective information could be efficiently used (A) and policy heterogeneity supported (D). Due to the effect PTL had on the equilibrium between objectives, in this application it was not able to improve learning's performance. This does not impact on the previously established benefit of transferring information related to one objective. The transfer of inter-objective information could be more beneficial if it could promote collaboration to a shared goal. This is further discussed in future work in Chapter 6.

5.4 Summary

The following requirements for PTL were identified in previous chapters: (A) Efficient Use of Knowledge, (B) On-Line Improvement, (C) Adaptiveness and (D) Heterogeneity Support. The former two requirements are met. PTL can effectively use the knowledge learnt by one agent to accelerate the learning of others. This can be done on-line, allowing the relatedness of tasks in a MAS to be exploited. The latter two requirements are met only in certain circumstances. The Self-Configuration component that provides PTL's adaptiveness, can only improve learning performance in response to change that stops. The length of time the change needs to stop for will depend on the application and how representative samples of the new environment are. For example, in GridLAB-D approximately a third of the normal learning period was sufficient. There needs to be some knowledge about a particular environment for PTL to leverage. PTL's Learnt Mapping component produces ITMs which allow heterogeneity to be supported. The ITMs can be learnt effectively in homogeneous tasks. In heterogeneous tasks it is more difficult, one of the configurations tried led to negative transfer and a worsening of performance, while the other was unable to bring performance to the same level as PTL had done using a homogeneous source. However, the degree of difference between the tasks is unlikely to occur within a single MAS, so the performance of mappings will be closer to the homogeneous case than the heterogeneous case.

This chapter has shown that PTL can accelerate learning in both large-scale autonomous MASs and in more simple applications.

Chapter 6

Conclusions

I am turned into a sort of machine for observing facts and grinding out conclusions.

Charles Darwin

This chapter summarises the thesis and reviews the most relevant contributions and achievements. Following this, it closes by identifying interesting open research issues that relate to the main contribution, Parallel Transfer Learning (PTL).

6.1 Contributions

This thesis's main aim was to accelerate Reinforcement Learning (RL) in Multi-Agent Systems (MASs). Chapter 1 provided motivation for this and introduced the systems for which PTL is designed. Learning-based control in large-scale autonomous systems allows them to be adaptive and operate in changeable environments. MASs are commonly used in these systems. The use of MASs increases the variability in the environment and affects the amount of time it takes for RL to learn to perform well. While RL is learning its performance is necessarily poor, which means the performance over the lifetime of a system will be increased by reducing the learning time.

Chapter 2 presented the state of the art in accelerating RL. It outlines three main ways of accelerating learning: (1) Algorithm Alteration, (2) Generalisation and (3) Additional Knowledge. Of these, Additional Knowledge is determined by this thesis to be the most applicable for use in MASs. In MASs, agents often learn similar things, so there is potentially useful knowledge to share. The fact that agents in a MAS learn broadly similar things makes them likely to be good sources of additional knowledge for one another, which can be used to accelerate each other's learning. If the agents in a MAS are homogeneous, then there is useful information to share.

The novel approach used in this thesis, PTL, is presented in Chapter 3. PTL has three components that enable it to make efficient use of knowledge that is learnt on-line in a MAS. It is capable of adapting to changes in the environment whether they are inherent or due to fluctuations in agent behaviour. It can also cope with heterogeneous agents, allowing them to support each other's learning. The components are as follows: (1) The **PTL component** is responsible for selecting and merging knowledge that can be used to accelerate learning. (2) The **Learnt Mapping component** provides heterogeneous agents with a mechanism by which they can translate the knowledge that they have learnt so that it can be understood by their neighbours. (3) The **Self-Configuration component** makes the system more adaptable to change. It monitors how well PTL is performing and if notices a drop in performance, it reacts to the change in performance by reconfiguring PTL so that it can try to mitigate the change that caused the performance drop.

Chapter 4 provides details of how PTL is implemented and introduces the application simulators that it is integrated into. These simulators are used in the evaluation in Chapter 5. The evaluation uses two well-known RL applications to investigate fundamental aspects of PTL and a further application to investigate how it behaves in MASs. The experiments show that PTL can accelerate learning in all of the applications considered. In one of the applications, the Cart Pole, it can achieve its final performance in half the time of unaccelerated RL. In another application, the Mountain Car, it improves the final performance over what RL can achieve by 27%. Additionally, it reaches comparable performance to RL's final performance in a tenth of the time. In the final application, a Smart Grid (SG) problem in which the charging of Electric Vehicles (EVs) and transformer load must be balanced, PTL is able to better RL's performance and regardless of training time does not fail to meet the constraint of having the EVs charged which RL fails to do on several occasions.

The Learnt Mapping component performs well in all the applications when homogeneous source tasks are used. Using two different methods, Vote-Based and Ant-Based mapping, the Learnt Mapping component is able to produces effective Inter-Task Mappings (ITMs) which allow knowledge to be transferred. When heterogeneous source and target tasks are used, the mapping performance is reduced. In one case, this is because there is no knowledge to transfer, in the other the knowledge from a heterogeneous source is not as good as from a homogeneous source. In this latter case, the performance is able to reach the level of RL alone, so it at least does not reduce performance. Generally, this means that source tasks should be selected carefully so that there is useful information to transfer. This can be seen by the results in the SG experiment, where the agents are more different from each other than in the other homogeneous experiments, but performance can still be improved.

The Self-Configuration component, which allows PTL to adapt to change, has been evaluated with different types of change; slow continuous change, fast continuous change and sharp change. The evaluation shows that if the change is not continuous and stops for long enough for agents to learn something, then PTL can leverage the knowledge learnt in the changed environment to perform better. If the change is continuous, PTL can not improve performance over RL's level as there is no useful knowledge to share. However, if a system is expected to continuously change, learning is a poor approach to take in the first place, as what is learnt is immediately invalidated.

PTL aimed to accelerate RL's learning in MASs and thereby improve performance, and has been shown to do this in three different applications. While it is somewhat limited in how changeable the environment can be and between which agents it can be applied, it is a suitable method to accelerate learning and improve resultant performance. It is particularly applicable in MAS environments like the SG where there is a great deal of repetition in what is learnt. In the SG there are many individual devices, but they fall into only a few classes (e.g., EVs, water heaters, etc.). What one device learns will be similar to what all of the class learns, this give significant potential for PTL to accelerate learning. There is also repetition in learning over time (e.g., day/night cycles, seasons, etc.), which gives potential to reuse knowledge from different times. Many large-scale systems exhibit this task repetition. For example, traffic management systems control traffic lights at many junctions. There are finite amount of junction layouts and traffic volumes. In this scenario, agents controlling traffic lights could transfer information to accelerate learning at other junctions of the same or similar type. Conversely, PTL would not be applicable in systems with little similarity in what is learnt. This could occur if agents are operating at a high level in a system. For example, in a whole grid solution, an agent operating in an industrial area would have little in common with a residential agent.

More generally, PTL will work in environments where samples are in some way expensive. In these environments the extra computational cost and risk of misleading information is offset by reducing the time spent performing poorly. If environments are easy to model or experience in the environment is easy to generate, then PTL will be less applicable as more relevant information can be generated without the overheads involved in transfer. If large quantities of prior knowledge exist then conventional Transfer Learning (TL) should be preferred. It can use this prior knowledge to achieve an initial improvement in performance that PTL can not. In many of these cases PTL would work, but it is not the preferred method as another method of accelerating learning works better. In situations where PTL would not work because of no commonality of knowledge, mapping could potentially work to allow PTL to perform well. This means that the only situations where PTL would not work are those where knowledge can not be mapped. This could be because it is implicit and can not be expressed (e.g., an emergent behaviour), or because the state-spaces are in some way incompatible (e.g., radically different numbers of actions or states etc.).

6.2 Future Work

The future work stemming from this thesis can be split into two parts: work related to addressing weaknesses in PTL and work that would be required before PTL can be deployed in the real-world. First the weakness in PTL related work will be presented.

- Identifying if there is useful information to share between agents is important for determining whether a pair of agents should transfer information. If there is no good information to share, then only negative transfer can result. This is particularly important when using learnt ITMs. If no useful knowledge exists, then this lack of information could be mistaken for a poorly performing mapping. This mapping could have considerable effort put into improving it when success is impossible.

- The significant effort required to produce mappings could be offset by collaboratively producing them. If a single mapping were produced from one class of agents to another, then the effort to learn a mapping could be shared between the agents, which would reduce the time required to produce them.

- It would be interesting to investigate the effect of representation of mapping. Reducing the size of a state-space by combining states can improve performance in a single task, but it can obfuscate knowledge which may make it non-transferable and thereby impact target performance. Determining if there is a type of representation that can facilitate transfer could improve performance and potentially allow greater difference between agents.

- Negative transfer could potentially be introduced after a change in the environment, so it is may important to be cautious with changing a previously good mapping afterwards. Analysing the risk of change introducing negative transfer is interesting and potentially important when learning mappings in a MAS.

- Aside from the problem of producing individual mappings, there is potential research around identifying when they are necessary. For example, given two agents, one with objectives A and B, the other with objectives A and C. When trying to map knowledge between the two objectives A, would it be necessary to use a learnt mapping, or would a simple matching of state names suffice? In this case, the objectives A are the same, but they may be affected by the other dissimilar objectives at each agent. There are more subtle aspects to this problem. In the case of different sets of objectives, it is relatively easy to know the single objectives may not be fully homogeneous, but it may not be in general.

- The factors that can cause homogeneous agents to differ can be more difficult to identify. For example, if the neighbours around a pair of homogeneous agents differ from one to the other, then what is learnt could differ too. This would effectively make anything transferred from one to the other without mapping potentially wrong. The risk of this problem occurring in a particular system would need to be estimated by agents. From this, they could then decide if they need to use a mapping or not. There will also be a trade-off between the potential drop in utility of transfer from this subtle heterogeneity and the cost of producing a mapping.
- The inter-objective information could be used to identify subtle heterogeneity. Directly applying transferred inter-objective information was found to negatively affect performance. Transfer of the inter-objective information could be improved by allowing agents to take action in which they choose to collaborate. The collaborate action would then allow an agent to do what is best for their neighbours. Using collaboration like this would be possible if the agents were sharing inter-objective information to mitigate subtle heterogeneity.

Deploying PTL in a real-world system the following would need to be done:

- The experiments for the Self-Configuration component identified the problem of continuously changing environments preventing good performance. This would require work before a system could be used effectively in a real-world applications. It is unlikely a learning-based approach could perform well regardless of how quickly it learns, so when this type of change is detected a system could switch to some other form of control (e.g., predefined by an expert, prediction based, etc.). The integration between RL and other forms of control that may be used would be important.

- It is possible that the other forms of control could be used as a source of information for PTL to transfer from. This raises a wider area of work, how can PTL operate in a system with algorithmic heterogeneity? In any deployment of PTL, there will be uncontrolled entities and those controlled by other algorithms. Modelling them as simply an input that PTL needs to accommodate (as is done here with base load in the SG) is naïve, as they will vary over time particularly when they are intelligently controlled. It also misses an opportunity to learn from or collaborate with other sources of knowledge in an environment. The extension of PTL to function in the presence of other algorithms is of interest.

In conclusion, the result of this thesis illustrate that PTL has considerable potential impact on MASs which use RL. It can in certain circumstances, reduce the time spent learning, and thereby improve overall performance. With further research it would be able to better operate in changing non-stationary environments, and could then be used in more real-world systems.

Bibliography

Abbeel, P. and Ng, A. Y. (2004). Apprenticeship Learning via Inverse Reinforcement Learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM.

Alpaydin, E. (2004). *Introduction to Machine Learning*. MIT press.

Amato, C., Konidaris, G. D., and Kaelbling, L. P. (2014). Planning with Macro-Actions in Decentralized POMDPs. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 1273–1280. International Foundation for Autonomous Agents and Multiagent Systems.

Ammar, H. B., Eaton, E., Ruvolo, P., and Taylor, M. E. (2015). Unsupervised Cross-Domain Transfer in Policy Gradient Reinforcement Learning via Manifold Alignment.

Ammar, H. B., Eaton, E., Taylor, M. E., Mocanu, D. C., Driessens, K., Weiss, G., and Tuyls, K. (2014). An Automated Measure of MDP Similarity for Transfer in Reinforcement Learning. In *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*.

Ammar, H. B. and Taylor, M. E. (2012). Reinforcement Learning Transfer via Common Subspaces. In *Adaptive and Learning Agents*, pages 21–36. Springer.

Anderson, J. R., Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (1986). *Machine Learning: An Artificial Intelligence Approach*, volume 2. Morgan Kaufmann.

Banerjee, A. and Basu, S. (2007). Topic Models over Text Streams: A Study of Batch and Online Unsupervised Learning. In *SDM*, volume 7, pages 437–442. SIAM.

Banerjee, D. and Sen, S. (2007). Reaching Pareto-Optimality in Prisoner's Dilemma using Conditional Joint Action Learning. *Autonomous Agents and Multi-Agent Systems*, 15(1):91–108.

Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT press.

Barto, A. G. and Mahadevan, S. (2003). Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(4):341–379.

Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike Adaptive Elements that can Solve Difficult Learning Control Problems. *Systems, Man and Cybernetics, IEEE Transactions on*, (5):834–846.

Batista, G. E., Prati, R. C., and Monard, M. C. (2004). A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data. *ACM Sigkdd Explorations Newsletter*, 6(1):20–29.

Bazzan, A. L. (2009). Opportunities for Multiagent Systems and Multiagent Reinforcement Learning in Traffic Control. *Autonomous Agents and Multi-Agent Systems*, 18(3):342–375.

Berenji, H. R., Chen, Y.-Y., Lee, C.-C., Jang, J.-S., and Murugesan, S. (2013). A Hierarchical Approach to Designing Approximate Reasoning-Based Controllers for Dynamic Physical Systems. *arXiv preprint arXiv:1304.1124*.

Bianchi, R. and Bazzan, A. (2012). Combining Independent and Joint Learning: a Negotiation Based Approach. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1395–1396.

Bianchi, R. A., Ribeiro, C. H., and Costa, A. H. (2008). Accelerating Autonomous Learning by using Heuristic Selection of Actions. *Journal of Heuristics*, 14(2):135–168.

Bianchi, R. A., Ribeiro, C. H., and Costa, A. H. R. (2007). Heuristic Selection of Actions in Multiagent Reinforcement Learning. In *IJCAI*, pages 690–695.

Bonfè, M., Castaldi, P., Mimmo, N., and Simani, S. (2011). Active Fault Tolerant Control of Nonlinear Systems: The Cart-Pole Example. *International Journal of Applied Mathematics and Computer Science*, 21(3):441–445.

Boutsioukis, G., Partalas, I., and Vlahavas, I. (2012). Transfer Learning in Multi-Agent Reinforcement Learning Domains. In *Recent Advances in Reinforcement Learning*, pages 249–260. Springer.

Brook, D. and Evans, D. (1972). An Approach to the Probability Distribution of CUSUM Run Length. *Biometrika*, 59(3):539–549.

Brys, T., Harutyunyan, A., Taylor, M. E., and Nowé, A. (2015). Policy Transfer using Reward Shaping. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 181–188. International Foundation for Autonomous Agents and Multiagent Systems.

Bryson, A. E. (1975). *Applied Optimal Control: Optimization, Estimation and Control.* CRC Press.

Busoniu, L., Babuska, R., and De Schutter, B. (2008). A Comprehensive Survey of Multiagent Reinforcement Learning. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 38(2):156–172.

Celiberto, L. A., Matsuura, J. P., López de Mantaras, R., Bianchi, R., et al. (2011). Using Cases as Heuristics in Reinforcement Learning: A Transfer Learning Application.

Chassin, D., Schneider, K., and Gerkensmeyer, C. (2008). GridLAB-D: An Open-Source Power Systems Modeling and Simulation Environment. In *2008 IEEE/PES Transmission and Distribution Conference and Exposition.*

Chatzidimitriou, K. C., Partalas, I., Mitkas, P. A., and Vlahavas, I. (2012). Transferring Evolved Reservoir Features in Reinforcement Learning Tasks. In *Recent Advances in Reinforcement Learning*, pages 213–224. Springer.

Coelingh, E. and Solyom, S. (2012). All Aboard the Robotic Road Train. *Spectrum, IEEE*, 49(11):34–39.

Commission for Energy Regulation (2011). Demand Side Vision for 2020: Decision Paper. `http://www.poyry.com/sites/default/files/imce/sem-11-022_demand_side_vision_for_2020.pdf`, Retrieved 10/28/14.

Deb, K. (2014). Multi-Objective Optimization. In *Search methodologies*, pages 403–449. Springer.

Devlin, S. and Kudenko, D. (2011). Theoretical Considerations of Potential-Based Reward Shaping for Multi-Agent Systems. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 225–232. International Foundation for Autonomous Agents and Multiagent Systems.

Devlin, S. and Kudenko, D. (2012). Dynamic Potential-Based Reward Shaping. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 433–440. International Foundation for Autonomous Agents and Multiagent Systems.

Di Marzo Serugendo, G., Gleizes, M.-P., and Karageorgos, A. (2005). Self-Organization in Multi-Agent Systems. *The Knowledge Engineering Review*, 20(02):165–189.

Dorigo, M., Maniezzo, V., and Colorni, A. (1996). Ant System: Optimization by a Colony of Cooperating Agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41.

Dusparic, I. and Cahill, V. (2009). Distributed W-Learning: Multi-Policy Optimization in Self-Organizing Systems. In *Self-Adaptive and Self-Organizing Systems, 2009. SASO'09. Third IEEE International Conference on*, pages 20–29. IEEE.

Dusparic, I. and Cahill, V. (2012). Autonomic Multi-Policy Optimization in Pervasive Systems: Overview and Evaluation. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(1):11.

Dusparic, I., Harris, C., Marinescu, A., Cahill, V., and Clarke, S. (2013). Multi-Agent Residential Demand Response Based on Load Forecasting. In *Technologies for Sustainability (SusTech), 2013 1st IEEE Conference on*, pages 90–96. IEEE.

Dusparic, I., Taylor, A., Marinescu, A., Cahill, V., and Clarke, S. (2015). Maximizing Renewable Energy Use with Decentralized Residential Demand Response. In *Proceedings of the 2015 International Smart Cities Conference (ISC2)*. IEEE.

Ehrgott, M., Ide, J., and Schöbel, A. (2014). Minmax Robustness for Multi-Objective Optimization Problems. *European Journal of Operational Research*.

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.

Fachantidis, A., Partalas, I., Taylor, M. E., and Vlahavas, I. (2015). Transfer Learning with Probabilistic Mapping Selection. *Adaptive Behavior*, 23(1):3–19.

Fang, X., Misra, S., Xue, G., and Yang, D. (2012). Smart Grid ? The New and Improved Power Grid: A Survey. *Communications Surveys & Tutorials, IEEE*, 14(4):944–980.

Farhangi, H. (2010). The Path of the Smart Grid. *Power and Energy Magazine, IEEE*, 8(1):18–28.

Fernández, F. and Veloso, M. (2006). Probabilistic Policy Reuse in a Reinforcement Learning Agent. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 720–727. ACM.

Forouzandehmehr, N., Esmalifalak, M., Mohsenian-Rad, H., and Han, Z. (2015). Autonomous Demand Response Using Stochastic Differential Games. *Smart Grid, IEEE Transactions on*, 6(1):291–300.

Furao, S., Ogura, T., and Hasegawa, O. (2007). An Enhanced Self-Organizing Incremental Neural Network for Online Unsupervised Learning. *Neural Networks*, 20(8):893–903.

Ganek, A. G. and Corbi, T. A. (2003). The Dawning of the Autonomic Computing Era. *IBM systems Journal*, 42(1):5–18.

Garant, D., da Silva, B. C., Lesser, V., and Zhang, C. (2014). Accelerating Multi-Agent Reinforcement Learning with Dynamic Co-Learning.

Gatti, C. (2015). Reinforcement Learning. In *Design of Experiments for Reinforcement Learning*, pages 7–52. Springer.

Geist, M. and Pietquin, O. (2011). Parametric Value Function Approximation: A Unified View. *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 9–16.

Grzes, M. and Kudenko, D. (2008). Plan-Based Reward Shaping for Reinforcement Learning. In *Intelligent Systems, 2008. IS'08. 4th International IEEE Conference*, volume 2, pages 10–22. IEEE.

Hafez, M. B. and Loo, C. K. (2015). Topological Q-Learning with Internally Guided Exploration for Mobile Robot Navigation. *Neural Computing and Applications*, pages 1–16.

Hahn, J. and Zoubir, A. M. (2015). Inverse Reinforcement Learning using Expectation Maximization in Mixture Models. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 3721–3725. IEEE.

Harris, C., Doolan, R., Dusparic, I., Marinescu, A., Cahill, V., and Clarke, S. (2014). A Distributed Agent Based Mechanism for Shaping of Aggregate Demand on the Smart Grid. In *Energy Conference (ENERGYCON), 2014 IEEE International*, pages 737–742. IEEE.

Hawkins, D. M. (1987). Self-Starting CUSUM Charts for Location and Scale. *The Statistician*, pages 299–316.

Hazan, E. (2012). The Convex Optimization Approach to Regret Minimization. In *Optimization for Machine Learning*, pages 287–305. MIT press.

Hernandez, L., Baladron, C., Aguiar, J. M., Carro, B., Sanchez-Esguevillas, A., Lloret, J., Chinarro, D., Gomez-Sanz, J. J., and Cook, D. (2013). A Multi-Agent System Architecture for Smart Grid Management and Forecasting of Energy Demand in Virtual Power Plants. *Communications Magazine, IEEE*, 51(1):106–113.

Howard, R. A. (1960). Dynamic Programming and Markov Processes.

Hu, Y., Gao, Y., and An, B. (2015). Learning in Multi-Agent Systems with Sparse Interactions by Knowledge Transfer and Game Abstraction. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 753–761. International Foundation for Autonomous Agents and Multiagent Systems.

Huebscher, M. C. and McCann, J. A. (2008). A Survey of Autonomic Computing – Degrees, Models, and Applications. *ACM Computing Surveys (CSUR)*, 40(3):7.

Humphrys, M. (1996). Action Selection Methods using Reinforcement Learning.

Jamshidi, M. (1996). Large-Scale Systems: Modeling, Control, and Fuzzy Logic.

Jennings, N. R. (1993). Specification and Implementation of a Belief-Desire-Joint-Intention Architecture for Collaborative Problem Solving. *International Journal of Intelligent and Cooperative Information Systems*, 2(03):289–318.

Junior, L. A. C. and Matsuura, J. P. (2011). Investigation in Transfer Learning: Better Way to Apply Transfer Learning Between Agents. In *Machine Learning and Data Mining in Pattern Recognition*, pages 210–223. Springer.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement Learning: A Survey. *Journal of artificial intelligence research*, pages 237–285.

Kara, E. C., Berges, M., Krogh, B., and Kar, S. (2012). Using Smart Devices for System-Level Management and Control in the Smart Grid: A Reinforcement Learning Framework. In *Smart Grid Communications (SmartGridComm), 2012 IEEE Third International Conference on*, pages 85–90. IEEE.

Karmellos, M., Kiprakis, A., and Mavrotas, G. (2015). A Multi-Objective Approach for Optimal Prioritization of Energy Efficiency Measures in Buildings: Model, Software and Case Studies. *Applied Energy*, 139:131–150.

Kattan, A., Fatima, S., and Arif, M. (2015). Time-Series Event-Based Rrediction: An Unsupervised Learning Framework Based on Genetic Programming. *Information Sciences*.

Kephart, J. and Chess, D. (2003). The Vision of Autonomic Computing. *Computer*, 36(1):41–50.

Kim, B.-G., Zhang, Y., van der Schaar, M., and Lee, J.-W. (2014). Dynamic Pricing for Smart Grid with Reinforcement Learning. In *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, pages 640–645. IEEE.

Knox, W. B., Setapen, A. B., and Stone, P. (2011). Reinforcement Learning with Human Feedback in Mountain Car. In *AAAI Spring Symposium: Help Me Help You: Bridging the Gaps in Human-Agent Collaboration.*

Knox, W. B. and Stone, P. (2009). Interactively Shaping Agents via Human Reinforcement: The TAMER Framework. In *Proceedings of the fifth international conference on Knowledge capture*, pages 9–16. ACM.

Knox, W. B. and Stone, P. (2012). Reinforcement Learning from Simultaneous Human and MDP Reward. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 475–482. International Foundation for Autonomous Agents and Multiagent Systems.

Kober, J. and Peters, J. (2012). Reinforcement Learning in Robotics: A Survey. In *Reinforcement Learning*, pages 579–610. Springer.

Konidaris, G., Scheidwasser, I., and Barto, A. G. (2012). Transfer in Reinforcement Learning via Shared Features. *The Journal of Machine Learning Research*, 13(1):1333–1371.

Kono, H., Kamimura, A., Tomita, K., and Suzuki, T. (2014). Transfer Learning Method Using Ontology for Heterogeneous Multi-agent Reinforcement Learning. *International Journal of Advanced Computer Science & Applications*, 5(10).

Lange, S., Riedmiller, M., and Voigtlander, A. (2012). Autonomous Reinforcement Learning on Raw Visual Input Data in a Real World Application. In *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pages 1–8. IEEE.

Laud, A. D. (2004). *Theory and Application of Reward Shaping in Reinforcement Learning*. PhD thesis, University of Illinois at Urbana-Champaign.

Lee, M. and Anderson, C. W. (2014). Convergent Reinforcement Learning Control with Neural Networks and Continuous Action Search. In *Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), 2014 IEEE Symposium on*, pages 1–8. IEEE.

Liu, W., Gu, W., Sheng, W., Meng, X., Wu, Z., and Chen, W. (2014). Decentralized Multi-Agent System-Based Cooperative Frequency Control for Autonomous Microgrids with Communication Constraints. *Sustainable Energy, IEEE Transactions on*, 5(2):446–456.

Lu, J., Behbood, V., Hao, P., Zuo, H., Xue, S., and Zhang, G. (2015). Transfer Learning using Computational Intelligence: A Survey. *Knowledge-Based Systems*.

Mannor, S., Perchet, V., and Stoltz, G. (2014). Approachability in Unknown Games: Online Learning Meets Multi-Objective Optimization. *arXiv preprint arXiv:1402.2043*.

Mannor, S. and Tsitsiklis, J. N. (2009). Approachability in Repeated Games: Computational Aspects and a StackelbergVariant. *Games and Economic Behavior*, 66(1):315–325.

Marinescu, A., Dusparic, I., Taylor, A., Cahill, V., and Clarke, S. (2015). P-MARL: Prediction-Based Multi-Agent Reinforcement Learning for Non-Stationary Environments. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 1897–1898. International Foundation for Autonomous Agents and Multiagent Systems.

Marler, R. T. and Arora, J. S. (2004). Survey of Multi-Objective Optimization Methods for Engineering. *Structural and multidisciplinary optimization*, 26(6):369–395.

Marthi, B. (2007). Automatic Shaping and Decomposition of Reward Functions. In *Proceedings of the 24th International Conference on Machine learning*, pages 601–608. ACM.

Martínez, D., Alenyà, G., and Torras, C. (2015). Relational Reinforcement Learning with Guided Demonstrations. *Artificial Intelligence*.

McGovern, A., Sutton, R. S., and Fagg, A. H. (1997). Roles of Macro-Actions in Accelerating Reinforcement Learning. In *Grace Hopper celebration of women in computing*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-Level Control Through Deep Reinforcement Learning. *Nature*, 518(7540):529–533.

Mohsenian-Rad, A.-H., Wong, V. W., Jatskevich, J., Schober, R., and Leon-Garcia, A. (2010). Autonomous Demand-Side Management Based on Game-Theoretic Energy Consumption Scheduling for the Future Smart Grid. *Smart Grid, IEEE Transactions on*, 1(3):320–331.

Montgomery, D. C. (2007). *Introduction to Statistical Quality Control*. John Wiley & Sons.

Moore, A. W. (1990). Efficient Memory-Based Learning for Robot Control.

Moore, A. W. and Atkeson, C. G. (1993). Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time. *Machine Learning*, 13(1):103–130.

Natrella, M. (2010). NIST/SEMATECH e-Handbook of Statistical Methods.

Ng, A. Y., Harada, D., and Russell, S. (1999). Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. In *ICML*, volume 99, pages 278–287.

Northrop, L., Feiler, P., Gabriel, R. P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., et al. (2006). Ultra-Large-Scale Systems: The Software Challenge of the Future. Technical report, DTIC Document.

Olfati-Saber, R., Fax, A., and Murray, R. M. (2007). Consensus and Cooperation in Networked Multi-Agent Systems. *Proceedings of the IEEE*, 95(1):215–233.

Palombarini, J. and Martínez, E. (2012). SmartGantt–An Intelligent System for Real Time Rescheduling Based on Relational Reinforcement Learning. *Expert Systems with Applications*, 39(11):10251–10268.

Pan, S. J. and Yang, Q. (2010). A Survey on Transfer Learning. *Knowledge and Data Engineering, IEEE Transactions on*, 22(10):1345–1359.

Panait, L. and Luke, S. (2005). Cooperative Multi-Agent Learning: The State of the Art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434.

Pareto, V. (1906). Manuale di Economica Politica, Societa Editrice Libraria. *(Translated into English by A.S. Schwier as Manual of Political Economy, edited by A.S. Schwier and A.N. Page, 1971. New York: A.M. Kelley).*

Peters, M., Ketter, W., Saar-Tsechansky, M., and Collins, J. (2013). A Reinforcement Learning Approach to Autonomous Decision-Making in Smart Electricity Markets. *Machine learning*, 92(1):5–39.

Pipattanasomporn, M., Feroze, H., and Rahman, S. (2009). Multi-Agent Systems in a Distributed Smart Grid: Design and Implementation. In *Power Systems Conference and Exposition, 2009. PSCE'09. IEEE/PES*, pages 1–8. IEEE.

Plauger, P., Lee, M., Musser, D., and Stepanov, A. A. (2000). *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

Prashanth, L. and Bhatnagar, S. (2011). Reinforcement Learning with Function Approximation for Traffic Signal Control. *Intelligent Transportation Systems, IEEE Transactions on*, 12(2):412–421.

Raza, H., Prasad, G., and Li, Y. (2015). EWMA Model Based Shift-Detection Methods for Detecting Covariate Shifts in Non-Stationary Environments. *Pattern Recognition*, 48(3):659–669.

Ruiz, A. B., Saborido, R., and Luque, M. (2014). A Preference-Based Evolutionary Algorithm for Multiobjective Optimization: the Weighting Achievement Scalarizing Function Genetic Algorithm. *Journal of Global Optimization*, pages 1–29.

Russel, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach, Third Edition*. Prentice Hall.

Sánchez, D., Batet, M., Isern, D., and Valls, A. (2012). Ontology-Based Semantic Similarity: A New Feature-Based Approach. *Expert Systems with Applications*, 39(9):7718–7728.

Selfridge, O. G., Sutton, R. S., and Barto, A. G. (1985). Training and Tracking in Robotics. In *IJCAI*, pages 670–672. Citeseer.

Sichman, J. S. and Coelho, H. (2014). Autonomous Agents and Multi-Agent Systems.

Skinner, B. F. (1953). *Science and Human Behavior.* Simon and Schuster.

Srivastav, A. and Agrawal, S. (2015). Multi Objective Cuckoo Search Optimization for Fast Moving Inventory Items. In *Advances in Intelligent Informatics*, pages 503–510. Springer.

Stone, P. and Veloso, M. (2000). Multiagent Systems: A Survey from a Machine Learning Perspective. *Autonomous Robots*, 8(3):345–383.

Sutton, R. S. (1990). Integrated Architectures for Learning, Planning, and Reacting based on Approximating Dynamic Programming. In *Proceedings of the seventh international conference on machine learning*, pages 216–224.

Sutton, R. S. (1996). Generalization in Reinforcement Learning: Successful Examples using Sparse Coarse Coding. *Advances in neural information processing systems*, pages 1038–1044.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*, volume 1. MIT press Cambridge.

Sutton, R. S., Szepesvári, C., Geramifard, A., and Bowling, M. P. (2012). Dyna-Style Planning with Linear Function Approximation and Prioritized Sweeping. *arXiv preprint arXiv:1206.3285.*

Sycara, K. P. (1998). Multiagent Systems. *AI magazine*, 19(2):79.

Tan, M. (2014). Learning a Cost-Sensitive Internal Representation for Reinforcement Learning. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 358–362.

Taylor, A., Dusparic, I., Galván-López, E., Clarke, S., and Cahill, V. (2014). Accelerating Learning in Multi-Objective Systems through Transfer Learning. In *Neural Networks (IJCNN), 2014 International Joint Conference on*, pages 2298–2305. IEEE.

Taylor, M. E. (2008). Autonomous Inter-Task Transfer in Reinforcement Learning Domains.

Taylor, M. E. and Stone, P. (2009). Transfer Learning for Reinforcement Learning Domains: A Survey. *The Journal of Machine Learning Research*, 10:1633–1685.

Taylor, M. E. and Stone, P. (2011). An Introduction to Intertask Transfer for Reinforcement Learning. *AI Magazine*, 32(1):15.

Taylor, M. E., Suay, H. B., and Chernova, S. (2011). Integrating Reinforcement Learning with Human Demonstrations of Varying Ability. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 617–624. International Foundation for Autonomous Agents and Multiagent Systems.

Tumer, K. and Agogino, A. (2007). Distributed Agent-Based Air Traffic Flow Management. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 255. ACM.

Van den Berg, J., Litjens, R., Eisenblätter, A., Amirijoo, M., Linnell, O., Blondia, C., Kürner, T., Scully, N., Oszmianski, J., and Schmelz, L. C. (2008). Self-Organisation in Future Mobile Communication Networks. *Proceedings of ICT Mobile Summit*, 2008.

Van der Hoek, W. and Wooldridge, M. (2008). Multi-Agent systems. *Foundations of Artificial Intelligence*, 3:887–928.

Van Seijen, H. and Sutton, R. S. (2013). Planning by Prioritized Sweeping with Small Backups. *arXiv preprint arXiv:1301.2343*.

Wang, J., Belatreche, A., Maguire, L., and McGinnity, T. M. (2014). An Online Supervised Learning Method for Spiking Neural Networks with Adaptive Structure. *Neurocomputing*, 144:526–536.

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

Weiss, G. (1999). *Multiagent Systems: a Modern Approach to Distributed Artificial Intelligence*. MIT press.

Weiss, G. M. and Provost, F. (2003). Learning when Training Data are Costly: the Effect of Class Distribution on Tree Induction. *Journal of Artificial Intelligence Research*, pages 315–354.

Wen, Z., O'Neill, D., and Maei, H. R. (2014). Optimal Demand Response using Device Based Reinforcement Learning. *arXiv preprint arXiv:1401.1549*.

Wiewiora, E. (2003). Potential-Based Shaping and Q-Value Initialization are Equivalent. *J. Artif. Intell. Res.(JAIR)*, 19:205–208.

Yahyaa, S. Q., Drugan, M. M., and Manderick, B. (2014). The Scalarized Multi-Objective Multi-Armed Bandit Problem: An Empirical Study of its Exploration vs. Exploitation Tradeoff. In *Neural Networks (IJCNN), 2014 International Joint Conference on*, pages 2290–2297. IEEE.

Zhao, D., Hu, Z., Xia, Z., Alippi, C., Zhu, Y., and Wang, D. (2014). Full-Range Adaptive Cruise Control based on Supervised Adaptive Dynamic Programming. *Neurocomputing*, 125:57–67.