University of Nairobi

School of Computing and Informatics

Integrating policy transfer, policy reuse and experience replay in speeding up reinforcement learning of the obstacle avoidance task

Miriti K.A. Evans

P80/84121/2012

Supervisor

Prof. Peter Waiganjo

December, 2014

A thesis submitted in partial fulfilment of the requirements of the doctorate degree of Computer Science, School of Computing and Informatics, University of Nairobi

Declaration

STUDENT:

This dissertation is my original work. It has not been presented for a degree in any other university. No part of this dissertation may be reproduced without the prior permission of the author or the University of Nairobi.

Signature ___________________________ Date________________

Miriti K.A. Evans

P80/84121/2012

SUPERVISOR:

The dissertation has been submitted for examination with my approval as the University Supervisor.

Signature ___________________________ Date________________

Prof. Peter Waiganjo

UNIVERSITY OF NAIROBI

Abstract

Obstacle avoidance is one of the key functionalities necessary for the proper functioning of an autonomous mobile robot. Smart & Kaelbling (2002) showed that the time required to learn this task using reinforcement learning can be unfeasibly long even for simplified versions of the task.

Knowledge transfer and experience replay are two techniques that have been suggested to speed up reinforcement learning. However, their application to the obstacle avoidance task is very limited. For instance Lin(1991) applied experience replay to an obstacle avoidance task in which the robot environment was bounded by a wall and therefore, the robot control agent needed to learn how to prevent the robot from colliding with the wall. Smart & Kaelbling (2002) applied a form of knowledge transfer known as teaching to the obstacle avoidance task, but there was only one obstacle in the environment.

Policy reuse and policy transfer are two knowledge transfer techniques that have been shown to improve learning performance when applied to the Keepaway sub problem of robotic soccer (Taylor & Stone, 2009; Fernández et al., 2010). These techniques can be used when there is the possibility of learning a simpler version of the intended task, then reusing the knowledge acquired in the simpler version of the task to bootstrap learning in the intended task. The simpler version of the task is called the source task while the intended task is called the target task. The obstacle avoidance task can be structured in this way. One way to achieve this is by creating an obstacle avoidance task containing fewer actions than the intended target task.

In this study, we investigated how policy transfer, policy reuse and experience replay can be combined to speedup learning in an obstacle avoidance task. We investigated the performance when the techniques were used in isolation and when they were used together. The experiments testing the performance of the techniques were setup in a robotics simulation environment. We used two sets of source and target task pairs. The first set comprised of two actions in the source task and three actions in the target task, while the second set comprised of three actions in the source task and six actions in the target task. The performance was measured by the average number of times the robot was able to reach the goal position under the guidance of the learned policy. The performance was calculated both at the initial level and at the asymptotic level.

In this study, our findings are that in the first pair of tasks, policy transfer outperformed policy reuse in as far as speeding up learning in the target task was concerned by about 30% at the initial level and about 10% at the asymptotic level. The combination of policy transfer with policy reuse was not found to lead to any significant improvement in the first pair of tasks, but it was found to lead to significant improvement in the second pair of tasks. The improved performance could be attributed to the fact that the six actions task is more difficult, hence it benefits more from bootstrapping. The combined techniques outperform policy reuse alone by 10% points in the six actions task based on the number of episodes that end at the goal region. The combination also overcomes the problem of declining performance observed in policy transfer from the three actions task to the six actions task.

It was also found that while experience replay led to significant improvement in learning the source task, it did not offer any improvement in learning when it was combined with knowledge transfer in the target task. In fact, it was seen to lead to degraded performance in most cases. For instance when experience replay was combined with policy transfer in the six actions task, the initial performance was 89% while the asymptotic performance was 59%. When policy reuse was combined with experience replay, the initial performance was 68% while the final performance was 59%. There was however some initial improvement when policy reuse was combined with experience replay, but this improvement was not sustained for long.

The main contribution of this work was a reinforcement learning framework that combines experience replay, policy reuse and policy transfer in learning the obstacle voidance task. We have also shown when each of these techniques can be most useful when applied to improve learning performance in reinforcement learning. These results will promote greater adoption and acceptance of the reinforcement learning techniques in the process of developing autonomous mobile robots.

Dedication

To my wife Alice and son Alvin: you bring meaning to my life. To my mum Leah and Dad Sebastian: you tried your best.

Acknowledgements

I want to express my appreciation and gratitude to the following:

To God almighty for giving me the strength to get through this challenging journey.

To my supervisor Prof. Peter Waiganjo for his guidance and support in all the phases of this work. Your suggestions and ideas greatly enriched this study.

To my colleagues at SCI who at one time or the other, were my companions through the journey. Your support and inspiration was pivotal in getting this work to its current position.

To all the administrative, technical and support staff at the School of Computing. Your support and cooperation was invaluable.

To the management of the University of Nairobi, and the School of computing and informatics. Thank you for providing a conducive environment in which one could focus on their research work.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

π_s – Source task policy

π_t – Target task policy

AHC - adaptive heuristic critic

ANNs – Artificial Neural Networks

A_s- Set of source task actions

A_t- Set of target task actions

CMAC – Cerebellar Model Articulator Controller

DARPA - Defence Advanced Research Projects Agency

DP – Dynamic programming

DSS - Decentralized Software Services

ER – Experience Replay

GR- Goal reward

MC - Monte Carlo

MDP - Markov Decision Process

MLP -Multi layer perceptron

MVSE - Microsoft Visual Simulation Environment

NREM - Non-rapid eye movement sleep

OCR – Obstacle collision reward

$Q^{\pi}(s,a)$ – state-action value function for state *s* and action *a* when following policy π

R(s,a,s') – The reward function. The reward obtained after transition from state ***s*** to state ***s'*** after taking action ***a.***

RDS – Microsoft Robotics Developer Studio

RL – Reinforcement Learning

SARSA(λ) - A combination of the SARSA algorithm with eligibility traces

SARSA(λ)-PR - A combination of the SARSA(λ) algorithm with policy reuse

SARSA(λ)-PT - A combination of the SARSA(λ) algorithm with policy transfer

SARSA(λ)-PT-PR - A combination of the SARSA(λ) algorithm with both policy transfer and policy reuse

SARSA(λ)-R - A combination of the SARSA(λ) algorithm with experience replay

SARSA(λ)-R-PR - A combination of the SARSA(λ) algorithm with both experience replay and policy reuse

SARSA(λ)-R-PT - A combination of the SARSA(λ) algorithm with both experience replay and policy transfer

SARSA(λ)-R-PT-PR - A combination of the SARSA(λ) algorithm with experience replay, policy transfer and policy reuse

S_s- Set of source task states

S_t- Set of target task states

T(s,a,s') - Transition function. The probability of transiting to state ***s'*** from state ***s*** given action ***a***

TBR –Time barred reward

t_{cal}- The t statistic calculated from the data

$t_{critical}$- The t statistic obtained from the students t distribution table for a given significance level.

TD – Temporal Difference

$V^{\pi}(s)$ – state value function for state *s* when following policy π

1. Introduction

1.1 Reinforcement Learning

In some environments, the use of supervised learning methods is nearly impossible. This is because supervised learning methods require that the trainer acquires an adequate number of examples with which to train the decision making agent. In some environments, this is not possible. Examples of such situations include: agents to control planetary rovers used for exploration of other solar system planets (Leffler, 2009), military robots that have to operate in enemy territory (Bruch et al., 2005), and robots that operate in the ocean floor. In some other cases, acquiring an adequate number of examples is just too tedious. As an example, if an agent is to be trained to control a motor vehicle is needed, the task of acquiring all the examples of road situations and actions that should be taken is a nearly impossible.

The other method that can used to solve these sorts of problems is via writing code to control the system. This sort of code would take the form of if then statements, which will give the actions to be taken for all possible states. In most of the cases, the programmer may not know all the possible states (Thrun, 1995), and in any case, the number of states may be infinite. In addition, the environment may change with time thus making it imperative that the control program be adjusted (Leffler, 2009). Examples of problems with huge state spaces include games like chess and backgammon.

Another problem that arises in some control problems is that, to solve a problem, a sequence of actions should be undertaken. It is not usually known if the actions in the sequence are the correct ones until at the end of the sequence. For instance in a game, we may not know if the actions taken are correct until the end of the game when the agent either losses, wins or draws in the game. The agent is then able to decide if the sequence of actions were the right ones. But even if the agent is able to know that the sequence of actions were the right ones, it may not know which actions contributed most to the win or loss. This is known as the credit assignment problem(Tesauro, 1992).

Reinforcement Learning (RL) is a learning paradigm that tries to address some of the learning challenges described above. It addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals. The agent is expected to learn based only on its experiences in its interaction with the environment. Every time an agent performs an action in the environment, it receives a reward or a penalty to indicate the desirability of the resulting state. As an example, a game playing agent may receive a positive reward for a winning state, a negative reward for a losing state, and a zero reward for all other states. The task of the agent is to learn from this indirect delayed reward, to choose sequences of actions that produce the greatest cumulative reward (Mitchel, 1997). Such learning agents can have many practical applications including: Manufacturing processes control, autonomous vehicle navigation, game development, and financial markets trading (Tesauro, 1992). Other applications include elevator scheduling and channel allocation in cellular networks (Peshkin, 2001).

The outcome of the agent learning is a control policy (π). The policy can be defined as a function π: S$\rightarrow$A that outputs an appropriate action *a*, given a state *s*, where S is the set of states, and A is the set of actions (Mitchel, 1997).

If the policy learnt by the agent enables it to choose actions that maximise its cumulative reward(overall reward), then the policy is said to be an optimal policy (π^*). Reinforcement learning methods specify how the agent changes its policy as a result of its experience.

1.2 Exploration-Exploitation Dilemma

One of the challenges faced by a RL agent is the Exploration-Exploitation trade-off (Sutton & Barto, 1998). In exploration, the agent has very limited knowledge on the value of actions or some of the actions available. In spite of the limited knowledge, it will still need to select the actions in order to see if they result in good rewards. By so doing, it foregoes selecting actions whose reward values it already knows, but gains more knowledge of the environment and the values associated with the actions it selects.

In exploitation, given several actions to choose from, the agent chooses the action with the highest value. It does not take the risk of trying to find out what the values of the other actions are (in case there values are not known, or are not known accurately).

For a Reinforcement learning agent to be successful, it will need to perform both exploitation and exploration in varying degrees (Sutton & Barto, 1998).

In RL, an agent gains knowledge about the value of its actions, and states in the environment through a trial and error process. An agent will require to experience each state or take an action a number of times before it can have a good estimate of its value (Sutton & Barto, 1998; Fernández et al., 2010). Since most environments have huge state spaces and action spaces, a lot of exploration would be required before the agent can have a good estimate of the value of each or state-action pair.

1.3 Knowledge Transfer

Tabula Rasa learning is the term applied to refer to reinforcement learning techniques that start off with a random policy (Taylor & Stone, 2009). An agent that starts off the learning process with a random policy will initially be capable of only choosing random actions. Such an agent is expected to have very poor initial performance as the agent explores the environment. It cannot be easily predicted how long the poor performance will last.

For a RL agent to be beneficial in a real life domain, the amount of exploration needs to be small (Leffler, 2009). This is because in most real life domains e.g. in robot domains, the exploration process is expensive in both time and energy. It is therefore imperative for RL algorithms to be data efficient in order to achieve acceptable performance after only a brief interaction with the environment (Adam et al., 2012). Touzet(1997) notes that the learning time must be as short as possible so as to reduce the development costs. A controller that performs poorly for a long period of time will never be accepted in practice even if it is believed that it will eventually yield optimal performance (Adam et al., 2012).

Kaelbling et al. (1996) points out that to solve highly complex problems, we must give up *tabula rasa* learning techniques and incorporate bias that will give leverage to the learning process. This is because in domains with large state-action spaces, the chances of finding rewards which lead to changes in the value function are very small. Smart & Kaelbling(2002) note that if sparse reward functions are used without including prior knowledge in the agent, the learning process is very unlikely to succeed. Taylor et al. (2007) pointed out that if RL agents start from scratch without any assistant, mastering larger tasks may be unfeasibly slow.

Knowledge transfer learning deals with the question of how a learned policy can be beneficial within a new task (Frommberger, 2010). It is a technique used in reinforcement learning to try and reduce the amount of experience required before an agent can achieve some pre-specified level of performance. Knowledge gained in learning one task is used to improve the learning speed in a different but related task (Taylor & Stone, 2009). The idea is to leverage knowledge gained in performing one task in learning how to perform another task, ensuring that the agent does not start from zero in learning to perform the new task.

Transfer can be beneficial when for instance an agent is moved to a new environment, or it is given a new goal. *Intra-domain* transfer occurs when the source task and the target task differ only in the reward function, while *cross-domain* transfer occurs when for instance the state space, action space and transition probabilities are different (Frommberger, 2010).

In knowledge transfer, the task in which we want to reduce the learning time is usually referred to as the *target task*. The task that provides the knowledge that we want to reuse is usually referred to as the *source task*. The target task is usually related in some way to the source task. Knowledge transfer is usually accomplished by initializing the target task agent with the knowledge from the source task hence the target task agent does not start from scratch. Usually, the source task knowledge will need to undergo some transformation so that it can be a form in which it can be used in the target task.

In most cases, transfer learning simplifies the learning of complex tasks by first solving simple tasks that are similar to the complex task, then transferring the knowledge to the more complex task (Knudson & Tumer, 2012).

A number of knowledge transfer techniques have been invented including: reward shaping (Laud, 2004), imitation (Smart & Kaelbling, 2002), directed training (Selfridge et al., 1985), learning from easier missions (Asada et al., 1994), hierarchical reinforcement learning (Singh et al., 2004), transfer via inter-task mapping for value functions (Taylor & Stone, July 2005), and policy reuse (Fernandez & Veloso, 2006). Each of these techniques has been applied to a very limited number of tasks hence a person working in a new task has to determine how a particular technique can be applied to the new task and if the performance obtained in the task of interest is significant.

1.4 Experience Replay

Experience replay is a technique that is used to speedup reinforcement learning by using experiences garnered in the environment to train the learning agent multiple times (Lin, 1992). Usually, in RL, an experience is utilised for training only once then discarded. Since in real life acting, experiences are usually expensive to obtain, they can be stored and reused later to train the agent. This is similar to training in supervised learning where the same examples are utilised many times until convergence occurs. Adam et al.(2012) found that experience replay (ER) speeded up learning in RL by up to four times in an inverted pendulum control problem. In addition, they found that the asymptotic performance achieved using ER is never attained by non ER reinforcement learning algorithms. Adam et al. (2012) also implemented ER in trying to solve a robotic arm manipulator problem and a robotic goalkeeper problem. The ER based reinforcement learning algorithms were found to perform well but the results were not compared to non ER based algorithms.

1.5 Obstacle Avoidance

Robot obstacle avoidance deals with the local observable aspect (within the robots perception horizon), where the robot may detect some unknown obstacles (real-time obstacles) on its path to an observable point called goal (Usher, 2006). This can be contrasted with robot navigation in which the robot is aware of the location of the obstacles, and it needs to come up with a path to reach the goal which may not be visible. Robot navigation methods require global knowledge of the state of the environment.

Obstacle avoidance is a subtask in robot navigation in that after the path planning algorithm comes up with global path, the robot will follow this path but will need to react to real time obstacles encountered along the path (Al-Jumaily & Leung, 2005). A vehicle capable of waypoint navigation, but without obstacle avoidance capabilities, will require to be remotely tele-operated by an operator who navigates the vehicle around obstacles (Bruch et al., 2005). This is likely to result in operator fatigue, and also keeps the operator occupied while he could be concentrating on other tasks. Another drawback of teleoperation is that over long distances, there will be a delay in the transmission of the sensory signal from the robot to the operator. As a result of this, the robot may end up running into obstacles before the operator sends the necessary command(s) for the obstacle avoidance manoeuvre. The objective of obstacle avoidance research is to reduce the level on human intervention in the mobile robot control hence increasing the autonomy of the robot.

Several attempts have been made to solve the obstacle avoidance problem using RL. These include: Sehad & Touzet(1994), Michels et al.,(2005) and Smart & Kaelbling(2002) among others. Smart & Kaelbling(2002) demostrated that learning the obstacle avoidance task using reinforcement learning could be unfeasibly slow even when the task was a in a greatly simplified form.

Smart & Kaelbling(2002) developed a two phase learning framework that used a form of knowledge transfer called imitation to speed up reinforcement learning in the obstacle avoidance task. The framework works by provision of example trajectories (provided by a human or hard coded control code) in the first phase of learning. The controller uses these example trajectories to improve its control policy. In the second phase of learning, the controller learns independently and generates its own trajectories which it uses to update its policy. The problem with this framework is the need for human intervention in the initial phase of training. This framework also requires that the task in the first phase of training be in the same as the task in the second phase of training.

1.6 Problem Statement

The obstacle avoidance feature is an important capability for control of mobile robots. It is one of the features necessary for autonomous navigation of mobile robots. Bruch et al. (2005) noted that vehicles capable of navigation from one location to another but without the obstacle avoidance feature will need to be tele-operated in order for them to be able to reach their destinations. The role of the operator will be to manoeuvre the vehicle around real time obstacles which the path planning algorithms were not aware of in generating the vehicle path.

Reinforcement learning is one of the methods that has been used inculcate the obstacle avoidance capability in mobile robots control agents. Smart & Kaelbling (2002) however showed that the time required for a control agent to learn the required behaviour may be unfeasibly long even for simplified versions of the task. For instance in Smart & Kaelbling (2002), it takes two hours of training in a simulator for a robot to reach the goal that is placed one meter from the starting position of the robot. This happens even when there are no obstacles in the environment. When the distance is increased to 2 meters, 6.24 hours of training are required before the robot can reach the goal once. The reinforcement learning frameworks that have been created to address this problem are insufficient in that they require human intervention in a process called teaching or imitation.

1.7 Objectives

The main objective of this study is to explore the use of knowledge transfer and experience replay to speed up learning in the obstacle avoidance task.

1.7.1 *Specific Objectives*

1) Review the state of the art in knowledge transfer, experience replay and application of reinforcement learning in the obstacle avoidance task
2) Propose framework for the application of knowledge transfer and experience replay in reinforcement learning in the obstacle avoidance task
3) Develop a control agent based on the proposed framework and interface the control agent with a robot simulation environment
4) Evaluate the performance of the control agent by performing simulations to determine the baseline performance, and the performance when knowledge transfer and experience replay are applied to the obstacle avoidance task
5) Update and validate the proposed learning framework

1.8 Significance of the Study

The following are some justifications for carrying out for this study:

1) Introduction of new ways of learning in obstacle avoidance that combines experience replay and knowledge transfer. This will show how knowledge transfer and experience replay can be integrated into learning and if they can result in improvement in performance.
2) Obstacle avoidance is one of the key features that need to be inculcated in autonomous vehicles and robots. Applications of such vehicles will include: clearing land mines, picking up injured soldiers from battle fields, fire fighting, surveillance and driverless cars among other things (Weinberger, 2012).

3) Weinberger (2012) notes that the level of autonomy of most robots that are currently being used is very limited. If reinforcement learning is used to train robots for obstacle avoidance, the knowledge acquired by one robot can be transferred to other robots so that they will not have to learn the task from scratch. An example of this would be the transfer of knowledge from a humanoid robot to a wheeled robot.
4) For some tasks, for example driving, initial bad performance can be dangerous.
5) Reduction of learning time may lead to greater acceptance and adoption of autonomous systems.

1.9 Chapter Summary

In this chapter, we introduced the concepts if reinforcement learning, knowledge transfer, experience replay and obstacle avoidance. We identified the main problem faced in learning the obstacle avoidance task using reinforcement learning. We then stated the objectives of the study which main deal with trying to find out how to apply knowledge transfer and experience replay to speedup learning in the obstacle avoidance task. We then listed some reasons why this study is important.

1.10 Organization of this thesis

The rest of this thesis is organized as follows:

Chapter two contains a review and analysis of literature related to reinforcement learning, knowledge reuse, experience replay and application of reinforcement learning to the obstacle avoidance task. The chapter also contains the proposed learning framework that integrates knowledge reuse techniques and experience replay in reinforcement learning. It also contains a listing of algorithms derived from the proposed framework and the research questions that need to be answered by determining the performance of the algorithms.

Chapter three is titled "Research Methodology" and describes how experiments were conducted to evaluate the framework proposed at the end of the literature review chapter. It starts by describing the use of simulation to conduct experiments and the justification for this choice. The nature of the task that was performed via simulation is then described. The chapter also contains a description of the environment in which the simulation experiments were carried out and the robotic capabilities it provided like sensors and actuators. The chapter also contains a description of the following issues: Performance measures used in evaluation the performance of the learning techniques, the general settings used in the experiments, steps taken to ensure validity of the results, and the statistical tests used in comparing the performance of different learning techniques.

Chapter four is titled “Simulation System Development Methodology”. It details the process used in the building of the simulation system used to conduct simulation experiments in this study. It starts by discussing some learning agent models. The system development methodology (evolutionary prototyping) used in building the system is then discussed. The system requirements section lists the functionalities that the system was expected to provide. The system design section details the various components used to build the system. It contains among other things: a description of the algorithms that were implemented in the learning system; a representation of the classes that make up the system and the relationships between them. The chapter concludes with a description of experiments carried out to select settings for the reinforcement learning parameters.

Chapter five contains details on the specific experiments carried out and the results obtained. Experiments were carried out to evaluate specific learning algorithms. The results are generally represented using line graphs that show the performance of the different algorithms that were being compared. The performance differences between the algorithms are then tested for their significance.

Chapter Seven contains a discussion of the results. The discussion is presented in a form that is designed to show the answers to the research questions. The chapter also contains a representation of the framework obtained by updating the proposed framework based on the results. Other items detailed in this chapter are: the achievements, the contributions, further work and the limitations of the study.

2. Literature review

In this chapter, the literature related to use of knowledge transfer and experience replay in reinforcement learning is extensively explored and analysed. Section 2.1 is titled "*The Goal of Reinforcement Learning*". It contains a general description of the learning problem that a learning agent faces and the nature of the solution that is expected to be provided when RL is used. The learning problem is described as the need to autonomously know how to select optimal actions in the situations that are encountered, while the solution is described in terms of the value function. The ensuing section titled "*Exploration Vs Exploitation*" describes why there is need to balance learning and use of the acquired knowledge. The subsequent section titled "*RL learning methods*" describes the techniques used to provide a solution to the learning problem usually by creating the value function. The equations used to update the value function are discussed. These equations are the basis of all reinforcement learning algorithms.

The section titled *"function approximations"* describes the function generalization methods used to approximate the value of states and | or actions that are not encountered during learning. Function approximation is necessary in RL because most problems of interest have infinite state and| or action spaces.

Due to the fact that RL agents are devoid of any knowledge when learning starts, learning has been found to take place very slowly. The section titled "*Learning speedup methods in reinforcement learning*" discusses some methods that have been applied in the past to try and speed up the learning process. The methods discussed are experience replay and knowledge reuse.

Also discussed in this chapter are: performance measures used in RL; a description of the obstacle avoidance task; and application of RL in obstacle avoidance.

In the section titled "*literature review summary*", the knowledge gaps identified from the review of literature are enumerated. Subsequent to this is a section titled "*Learning framework design*" in which a learning framework that incorporates knowledge transfer and experience replay in reinforcement learning is proposed. The reasons for the choices made in designing the framework are also discussed. The framework design section is followed by a listing of the algorithms derived from the framework. The chapter ends with a list of the research questions that should be answered by an evaluation of the framework.

2.1 The Goal of Reinforcement Learning

2.1.1 *Introduction*

Supervised learning is the most common learning technique used in machine learning (Sutton & Barto, 1998). It involves learning from examples provided by a knowledgeable external supervisor. In order for an agent that learns using supervised learning to be effective, it has to be provided with learning examples that characterise all the possible situations that it may encounter in its working environment. For some problems, it is often difficult to obtain an adequate number of examples that are representative of all situations in which an agent has to act. This is because the environment may be so large that the supervisor providing the examples may not be aware of all the possible situations or be able to enumerate them. It is also possible that the supervisor may have an inaccurate knowledge of the environment for instance when dealing with uncharted territory. Consequently, the examples they provide may not be completely accurate.

Another disadvantage of supervised learning is that the environment in which an agent is placed may change. This changed environment may have dynamics that are different from the environment in which the agent was trained. As an example, the outcome of actions may different from what the agent was 'told' by the supervisor. A supervised learning agent's knowledge is static and so it cannot adapt its knowledge to respond to the changes in the environment.

Lin(1992) argues that a human supervisor provides learning examples based on what they perceive. Given that humans and robots have different sensors, it is likely that the optimal action from the point of view of the supervisor may not be optimal for the robot. To provide an optimal example, the supervisor needs to see exactly what the robot is seeing.

An additional disadvantage of supervised learning is that human actions may not always be optimal. Humans for instance tend to make mistakes while driving. These examples of how not drive may also be fed into a learning agent during supervised learning.

Reinforcement learning is a learning technique that addresses some of the shortcomings of supervised learning. It addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals. Every time an agent performs an action in the environment, it receives a reward or a penalty to indicate the desirability of the resulting state. For example, a game playing agent may receive a positive reward for a winning state, a negative reward for a losing state, and a zero reward for all other states. The task of the agent is to learn from this indirect delayed reward to choose sequences of actions that produce the greatest cumulative reward (Mitchel, 1997). This way, the agent can learn the most optimal actions for the different situations that it encounters without waiting for the intervention of an external supervisor to provide a solution (Lin, 1992). The outcome of the agent learning is a control policy (π). The policy can be defined as a function π: S→A that outputs an appropriate action *a*, given a state *s*, where S is the set of states and A is the set of actions (Mitchel, 1997).

If the policy learnt by the agent enables it to choose actions that maximise its cumulative reward, then the policy is said to be an optimal policy (π*). Reinforcement learning methods specify how the agent modifies its policy as a result of its experience.

2.1.2 *State value and state-action value functions*

Reinforcement learning makes use of the concept of value function. Assuming that the agent is placed in a finite state environment with a finite number of actions, the agent is usually expected to learn *the utility value of each state* or *the value of each state–action pair*. Value here is defined in terms of expected sum of rewards the agent will receive from the state being considered until it reaches a terminal state.

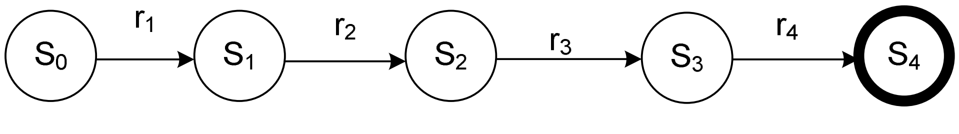


Figure 2.1: Example of state transitions to get to the final state

Using Figure 2.1 as an example of state transitions, then the value of state s_0 can be calculated as shown in Equation 2.1.

$V(s_0)=r_1+r_2+r_3+r_4$ *Equation 2.1*

Where s_4 is a terminal state and r_i is the reward obtained after transition *i*. Since the agent may take a different path to get to s_4 from s_0 (by choosing different actions, or by the actions being nondeterministic), $V(s_0)$ should be an average of the rewards received from of all paths that have been taken or can be taken to get to s_4 from s_0. Hence the term expected sum of rewards.

Rewards can also be discounted, such that immediate rewards are given more weight than future rewards. For instance, to calculate the value $V(s_0)$ using discounting, we use Equation 2.2.

$V(s_0)= r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4$ *Equation 2.2*

Where $0 \leq \gamma \leq 1$ is known as the *discount rate*.

2.1.3 *Markov Decision Process*

The specification of a sequential decision problem for a fully observable environment with a Markovian transition model and additive rewards is called a Markov decision process (MDP). An MDP is defined by the following three components:

i. Initial state S_0.
ii. Transition Model: T(s, a, s')
iii. Reward function: R(s)

In an MDP, the transition model and the reward function depend only on the current state and are not affected by previous states or actions (Sutton & Barto, 1998; Russell & Norvig, 2003). Reinforcement learning problems are usually specified as MDPs. A policy's value function assigns to each state or state-action pair the largest possible return achievable by the policy, starting from that state or state action pair, given that the agent uses the policy. The optimal value function assigns to each state or state-action pair, the largest expected return achievable by any policy that can be used (Sutton & Barto, 1998).

2.1.4 *Policy Evaluation and Policy Improvement*

Given a fixed policy, or a way of acting, then we may want to know the expected value of states when the agent follows this policy. This is called policy evaluation. Policy improvement on the other hand, tries to alter the current policy to find out if the resulting policy results in higher state values.

2.1.5 *Environment Model*

An agent in a new environment may not know how that environment works. Figure 2.2 illustrates this.

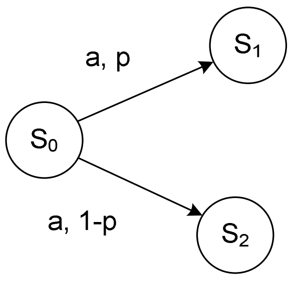


Figure 2.2: A nondeterministic environment

An agent that is new in the environment in Figure 2.2 may not know that choosing action *a* in state s_0 may result in a transition to either state s_1 or state s_2. It may also not know the probability of transitioning to either of the states. An agent may need to learn this information as part of its learning task. It might also not know the reward it will get by transitioning to either of the states. Learning this information is referred to as learning a model of the environment.

2.2 Exploration Vs Exploitation

One of the challenges faced by a RL agent is the Exploration-Exploitation trade-off (Sutton & Barto, 1998). In exploration, the agent has very limited knowledge on the values of actions or some of the actions available. It has to select the actions in order to see if they result in good rewards. By doing so, it foregoes selecting actions whose reward values it already knows, but gains more knowledge of the environment and the values associated with the actions. However, exploration without using the knowledge is of little use Russell & Norvig(2003).

In exploitation, given several actions to choose from, the agent chooses the action with the highest value. It does not take the risk of trying to find out what the values of the other actions are (in case their values are not known, are not known accurately). An agent that chooses to use an exploitation approach alone is called a *greedy agent*. According to

Russell & Norvig(2003) greedy agents seldom converge to the optimal policy. This is because the greedy agent never learns the true model of the environment.

An example of an approach that tries to balance exploration and exploitation is the **ε-greedy** strategy (Sutton & Barto, 1998). This method selects the greedy action most of the time but sometimes, with a small probability **ε** randomly selects any of the available actions with equal probability.

Another approach is the **softmax** action selection method in which the probability of selecting an action is directly proportional to the estimated value of that action (Sutton & Barto, 1998). This means that actions with higher estimated values are more likely to be selected but other actions also have a likelihood of being selected. The advantage of this approach is that if the best action is not selected, the second best action is the next most likely to be selected and the worst action the least likely to be selected. One common softmax method uses the Boltzmann distribution to generate the probability for selecting each of the possible actions as shown in Equation 2.3.

$$P(a) = \frac{\exp\left({Q_t(a)}/{\tau} \right)}{\sum_{b=1}^{n} \exp\left({Q_t(b)}/{\tau} \right)}$$ *Equation 2.3*

τ is referred to as the **temperature**. High temperatures cause the all the actions to have a nearly equal probability of being selected. Low temperatures cause the actions with high values to be preferred (Sutton & Barto, 1998).

Sutton & Barto(1998) note that ε-greedy approaches are usually easier to use than softmax approaches because it is easy to predict the resulting action selection behaviour when the ε-greedy method is applied. It is easy to tell the expected probability of selecting the greedy action, and the expected probability of selecting the other actions. For the softmax approaches, in order to predict the action selection behaviour, the value of the actions must be known. In addition, the effect of exponentiation on the value of the actions must also be estimated given the selected value of τ.

Russell & Norvig (2003) note that the the use of **optimistic initial values** can also be used to balance exploration and exploitation. An example of an algorithm that uses this strategy is the R_{max} algorithm (Brafman & Tennenholtz, 2002). States that have not been explored or have been visited very few times are given high rewards to encourage the agent to choose actions leading to those states hence exploring them. The disadvantage of this approach is that it will require the states to be discrete in nature. Each of the discrete states can then be assigned a high reward value.

In most tasks, a lot of exploration is usually required before the RL agent can gain adequate knowledge of its environment to perform reasonably well. This is because the RL agent gains knowledge about the value of its actions and states in the environment through a trial and error process. Most environments have huge state spaces and an agent will require to experience each state or take an action a number of times before it can have a good estimate of its value (Sutton & Barto, 1998; Fernández et al., 2010).

The TD-Gammon program that learns how to play the backgammon game had to play several hundred thousand games against itself before it could perform at strongly intermediate level (Tesauro, 1995). Tesauro(1995) notes that the initial random strategy is exceedingly bad, though the agent ultimately learns to perform at expert level after a lot of experience.

2.3 Reinforcement Learning Methods

2.3.1 *Direct Utility Estimation (Monte-Carlo Methods).*

Monte Carlo (MC) methods require only *experience* – sample sequences of states, actions, and rewards from on-line or simulated interaction with an environment (Sutton & Barto, 1998). An episode is defined as a sequence of state transition pairs starting at a given state s_0 and terminating at a final state.

Policy Evaluation

Suppose we wish to estimate $V^{\pi}(s)$, the value of a state *s* under policy π, given a set of episodes obtained by following π and passing through *s*. Each occurrence of state in an episode is called a *visit* to *s*. The *every-visit MC method* estimates $V^{\pi}(s)$ as the average of the returns following all the visits to *s* in a set of episodes. Within a given episode, the first time *s* is visited is called the *first-visit* to *s*. The *first-visit MC method* averages just the returns following first visits to s. In other words we get the total sum of rewards obtained in all state transitions from *s* until the final state is encountered in a given episode. We can the get the average of this sum for all episodes in which *s* is visited and this is stored as the value $V^{\pi}(s)$ for *s* under policy π.

Both *first-visit MC* and *every-visit MC* converge to $V^{\pi}(s)$ as the number of visits goes to infinity. *First-visit MC* is the method that is more commonly in use.

The MC method does not require a model P(s,a,s′) in order to carryout policy evaluation because the action is pre-specified by the policy. If the model was available, we would be able to estimate the state values without having to get samples from the environment using the dynamic programming approaches discussed later.

Though policy evaluation helps us know the values of the states when using a given policy, it does not enable us to determine if this policy is the best policy. To determine the optimal policy, we would have to carry out evaluation for all possible policies then select the one in which the states have the greatest values.

Instead of estimating the state value function $V^{\pi}(s)$, we can choose to estimate the state-action value function $Q^{\pi}(s,a)$ which is the value of choosing action *a* in state *s* and thereafter following policy π. If policy π is deterministic, only the value for one action from *s* will be learnt. Due to this limitation, MC methods are usually used to estimate $Q^{\pi}(s,a)$ for none deterministic policies hence each state-action pair has a none-zero probability of being selected. An example of such policies is the ε-greedy policies.

Policy Improvement

Given the current policy we may want to improve it so that it performs better by selecting actions which lead to states with greater rewards. For instance if in state *s*, $\pi(s)=a_1$, (meaning policy π selects action a_1 in state *s*) we might want to change this to $\pi(s)= a_2$ where we believe, based on the currently available information that a_2 is better than a_1. This is known as policy improvement. In MC methods, Policy improvement is obtained by making the policy greedy with respect to the current state-action value pairs. This is by selecting the action with the highest value in state *s* as shown in Equation 2.4.

$\pi(s)=\text{argmax}_a(Q(s,a))$ *Equation 2.4*

The main disadvantage of MC methods is that the value function update can only be effected at the end of the episode. Any useful information that is obtained before the end of the episode cannot be used immediately to update the value function. This means that in- episode improvement cannot occur.

2.3.1.1 Dynamic Programming

Dynamic programming (DP) can be used to compute the state value function or the state-action value function without collecting any samples form the environment. These can in turn be used to generate the optimal policies (Sutton & Barto, 1998). DP algorithms are obtained by turning the Bellman equations into assignments, that is, into update rules for improving approximations of the desired value functions. Equation 2.5 and Equation 2.6 are the Bellman optimality equations for state value functions, while Equation 2.7 and Equation 2.8 are the Bellman optimality equations for state-action value functions.

i. *For state value functions*

$$V^*(s) = \max_a E\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\}$$ *Equation 2.5*

$$V^*(s) = \max_a \sum_{s'} P^a(s,s')[R^a(s,s') + \gamma V^*(s')]$$ *Equation 2.6*

ii. *For state-action value functions*

$$Q^*(s,a) = \{r_{t+1} + \gamma \max_{a'}(Q^*(s_{t+1},a')) \mid s_t = s, a_t = a\}$$ *Equation 2.7*

$$Q^*(s,a) = \sum_{s'} P^a(s,s')[R^a(s,s') + \gamma \max_{a'} Q^*(s,a')]$$ *Equation 2.8*

Policy Evaluation

Computing the state-value function for an arbitrary policy π is called policy evaluation. When using dynamic programming, Equation 2.9 is used to obtain the updated state value.

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a(s,s')[R^a(s,s') + \gamma V^\pi(s')]$$

Equation 2.9

Where π(s,a) is the probability of choosing action *a* in state *s* under policy π. If the environmental dynamics are completely known, then we will have a system of |S| simultaneous equations with |S| unknowns which can be solved using algebraic methods.

Algebraic methods are usually tedious and computationally expensive and so an iterative method called *iterative policy evaluation* is used. This starts with arbitrary values for all $V^{\pi}(s)$ except the terminal state which, if any, may given a value which is equivalent to the goal reward. As the algorithm iterates, with the value in iteration *k* being subscripted as $V_k^{\pi}(s)$, $V_k^{\pi}(s)$ will tend to $V^{\pi}(s)$ as k tends to ∞. Usually, the iterations are stopped when successive changes to the state-value function are below a minimum value ε.

In the absence of an environment dynamics model, then, one can be built as the agent experiences the world. This gives rise to what is referred to as the Adaptive Dynamic Programming Agent (Russell & Norvig, 2003). $R^a(s, s')$ is the reward model and $P^a(s, s')$ is the transition model. The number of transitions from *s* to *s′* when action *a* is taken can be kept in a table, including the total number of times action *a* is taken in state *s*. $P^a(s, s')$ is given by dividing the total number of transitions from *s* to *s′* given action *a*, with the total number of times action *a* is taken in *s*. $R^a(s, s')$ is the average reward received when action *a* is taken in state *s* resulting in a transition to state *s′*.

Policy Improvement

Policy improvement is achieved by making changes to the current policy. The change is to make the policy greedy with respect to the current state-values similar to way state improvement was effected in MC methods. This means that if the current policy recommends action a_1 but there is an action a_2 that generates the greatest predicted value, then, change the policy so that it always selects a_2.

The main disadvantage of dynamic programming is that it requires that the number of states be finite. This way, these states can be used to create a model of the environment (if it does not exist). A model of the environment is required for Equation 2.5 to Equation 2.9 to be used.

2.3.1.2 Temporal Difference Learning

Equation 2.10 and Equation 2.11 are the update equations used in temporal difference learning.

i. *For state value functions*

Equation 2.10

$$V^{\pi}(s_t) = V^{\pi}(s_t) + \alpha[r_{t+1} + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)]$$

ii. *For state-action value functions*

$$Q^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) + \alpha[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) - Q^{\pi}(s_t, a_t)]$$ *Equation 2.11*

The algorithm that uses Equation 2.11 for updates is referred to as SARSA. α is referred to as the *learning rate* parameter, and it determines the size of the change in the state value or state-action value in each update.

TD methods base their updates on existing estimates and are therefore said to be bootstrapping methods. These methods do not require a model of the environment in order to carryout the policy evaluation. If action-values are estimated, then a model is not required for acting either. Using ε-greedy policies, SARSA converges to the optimal policy and action-value function as long as each state-action pair is visited an infinite number of times.

Another variant of temporal difference learning for the state-action value function uses Equation 2.12.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$ *Equation 2.12*

The algorithm that uses this equation is referred to as Q-learning. In this case, the learned action-value function Q, directly approximates Q* the optimal action-value function independent of the policy being followed (Sutton & Barto, 1998). The value of α is required to be less than 1 and greater than 0 for convergence to occur (Mitchel, 1997).

2.4 Function Approximations

2.4.1 *Introduction*

In the context of reinforcement learning, generalization is the use of prior experience to infer information about unseen states (Leffler, 2009). If the state space is small, then the calculated state value function or state-action value function can be stored in a table. But the time to convergence and the time per iteration for dynamic programming increases rapidly as the state space gets bigger (Russell & Norvig, 2003). Chess for instance has about 10^{50} states. Using table based representation, an agent would have to visit all these states (several times) to learn their state value under some policy. Since this is computationally infeasible, function approximation is used to represent the state value functions. The representation is viewed as approximate because it might not be the case that the true utility function can be represented in the selected form. For instance the state value function for chess can be represented as a weighted linear function of a set of features or basis functions as shown in Equation 2.13.

$$\hat{V}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s) \qquad \text{Equation 2.13}$$

A reinforcement learner can then learn the value of $\theta_1 \ldots \theta_n$ given some example experiences and the function $\hat{V}_\theta(s)$ can be used to estimate the true utility function. Methods such as the Widrow-Hoff rule which is based on the squared difference of the predicted value and the actual value can be used to adjust the values of the parameters given a training example. Other function approximators that can be used include neural networks, Kohonen maps, Tile Coding and decision trees. For instance in Tesauro (1995), a neural network is used as a function approximator in the TD-Gammon program that uses temporal difference learning to learn to play backgammon by playing against itself. The following is a discussion of some commonly used function approximators.

2.4.2 *Artificial Neural Networks (ANNs)*

Neural networks are computing models inspired by the structure of the biological nervous system. They consist of simple interconnected processing units where each unit receives a number of real valued inputs and produces a real valued output (Mitchel, 1997). The interconnection is achieved by having the outputs of some of the units being inputs to other units in the network. Neural networks usually learn by adjusting the weights of the connections between the processing units. Multilayer feed forward neural networks that are trained using the Backpropagation algorithm are one of the most common and practical ANNs structure (Mitchel, 1997).

In Reinforcement learning, ANNs are often used to represent the control policy. Tesauro uses a multilayer feed forward neural network with one hidden layer to represent the policy for selecting moves in a backgammon game (Tesauro, 1992). The network generates training experience by playing against itself. The TD(λ) algorithm is used to train the network. Given a particular board state, the network was supposed to output the probability of winning from that board state. The board state represented the number of pieces (for each player) in each of the possible board positions, the player whose turn it was to play, and the number of pieces captured for each player. The network was able to perform better in backgammon games than ANNs trained using supervised learning with training experience obtained from human expert games. It was found that the network performance seemed to improve with an increase in the number of hidden nodes.

Huang et al.(2005) use a multi layer perceptron in combination with the Q-learning algorithm in learning an obstacle avoidance task. The Backpropagation algorithm is used for training. The number of the outputs of the neural network is equal to the number of actions available. They show that neural network based Q-learning outperforms the alternative method which they refer to as normal RL though the actual representation used is not stated.

According to Mitchel(1997), several structures can be adopted when an ANN is used to represent the policy.

i. In structure 1, the network has only one output node that predicts the state-action value for a specified action. In this structure, the network inputs include the state plus an input node for each of the possible actions. At any one particular time when inputs are presented to the network, only one action node is set to 1 with all the other action nodes set to 0. The network will predict the state-action value for the selected action.

ii. In structure 2, the number of network outputs nodes is equal to the number of actions available. In this situation, when a given state is presented as the input, each output node produces the state-action value for its corresponding action.

iii. In structure 3, there is a separate ANN for each action, such that each ANN has only one output node. To select the best action, the current state is presented to each of the networks and the best action is associated with the network that generates the largest output.

Miriti et al.(2013) compared the first two structures in a task to drive a robot from a start position to a goal with no obstacles in the environment. It was found that the second structure outperformed the first structure by far. After 100 episodes, the average performance (the number of times the robot arrived at the goal) of the second structure was 80% while the average performance of the first structure was 20%. It was doubtful if there was any learning at all taking place in the first structure. The performance of the third structure was not evaluated.

2.4.3 *Kohonen Maps*

Kohonen maps are also known as self organizing maps and are classified as a form of neural networks. Touzet(1997) used Kohonen maps for the policy representation in an obstacle avoidance task. The Q-learning algorithm was used for the learning purposes and the resulting implementation was therefore referred to as Q-Kohon. In Q-kohon, each node represents a triplet of state, action and value information stored in the form of a vector. These nodes are arranged in a two dimensional grid. Each node also has a neighbourhood of 4 nodes. Nodes in a neighbourhood code for similar vectors.

During training, the input vector consisting of the state, action and Q-value is compared to the vectors stored at each of the nodes using a distance measure. The node whose distance from the input vector is the least is considered the winner. The winning node and its neighbours are modified so that their weights are brought closer to the winning vector. Equation 2.14 is used in updating the winning node i whose weights are represented by vector w_i.

$$w_i(t+1) = w_i(t) + \alpha(x - w_i) \quad \text{Equation 2.14}$$

And for the neighbouring nodes $j \in \{1,2,3,4\}$ the Equation 2.15 is used for update.

$$w_j(t+1) = w_j(t) + \beta(x - w_j) \quad \text{Equation 2.15}$$

Where x is the input vector and $0<\beta<\alpha<1$.
In selecting an action, only the current state is available, the Q-value is set to 1, therefore only one element of the triplet (the action is missing). The distance between this vector and all the nodes in the grid is still calculated in the usual way but the action part is excluded from the calculation. When the closest node is determined, the value at its action part of the vector is selected as the action that should be performed.

In tests using the Khepera miniature robot, Touzet(1997) showed that this method outperformed all other methods it was compared with in terms of performance and training time. Q-Kohon was observed to take a factor of 4 times less than the multi layer perceptron (MLP). It however needed more memory.

Macek et al. (2002) also use Kohonen maps in an obstacle avoidance task. In their implementation, the nodes only store the state vector. Each node is further associated with *n* actions values (where n is the number of actions in the task). Once an input has been associated with a given node, the action values associated with the node are compared and the action with the highest value is selected. This implementation allows storage of the eligibility traces, with an eligibility trace being associated with each action value. The authors report that the performance in a task involving corridor navigation was good but the performance measure is not very clear. Comparison with other techniques also does not seem to have been carried out.

2.4.4 *Tile Coding*

Tile coding is a form of linear function approximation in which the input space is divided into several segments, with each input falling into one of the segments. This is referred to as course coding (Sutton & Barto, 1998). Tile coding is also referred to as Cerebellar Model Articulation Controller (CMAC) (Skelly, 2004). In tile coding, a subdivision of the input space is referred to as a tiling and each of the subdivisions is referred to as a tile.

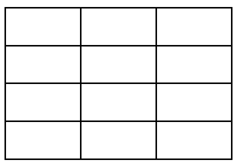


Figure 2.3: A tiling with 12 tiles representing a subdivision of a 2 dimensional input space

Each tile in a tiling is considered to be a feature and is associated with a weight which has to be learnt. If only one tiling is used, this will be equivalent to a table based representation of the value function. For a given input state, only one of the tiles in a tiling is activated, and the value associated with that tile is the value associated with the input. In practice however, we usually have several different segmentations of the input space resulting in several tilings. An input value will activate a single tile in each of the tilings. The values associated with each of the activated tilings will be summed to obtain the value associated with the input. For instance if we have only 2 tilings, for a given input only the weights associated with two tiles will be added (one from each of the tilings).

Training involves adjusting the weights in activated tiles towards the target value using an appropriate learning rate.

For tile coding, the memory requirements are usually exponentially dependent on the dimensions of the input space (Skelly, 2004). Hashing schemes may however be used to map high memory tile indices into the available memory space.

Tile coding has for instance been used as a function approximation mechanism in learning the Keepway subtask in (Stone & Sutton, 2001). The policy generated using tile coding resulted in much better performance compared to the hand-coded policy (pre-specified actions). This showed that tile coding schemes can indeed be used for generalization.

Sherstov & Stone(2005) investigated adaptive setting of parameters in using tile coding as a generalization technique. Two parameters that the user has to set are: the number of tiles in a tiling, and the size (width) of the tiles in the tilings. The resolution of the representation is a ratio w/t where w is the width of each tile and t is the number of tilings. To keep this ratio constant, if we increase w (resulting in bigger tiles), then t must also increase. This means that when we having larger tiles, then we should increase the number of tilings (t). If w reduces on the other hand, t has to reduce. We can reduce w up to the point where we have only one tiling (hence very small tiles). Sherstov & Stone (2005) investigated if it is preferable to have many tilings with large tiles, or few tilings with small tiles. They found that the setting in which there were many tilings with large tiles had better initial performance compared to setting where there were fewer tilings with small tiles. Asymptotically however, the setting with few tilings and smaller tiles outperformed the setting where there were many tilings with large tiles. The explanation for this performance is that small tiles make more precise value estimates while large tiles make better generalization because they represent a larger portion of the state space. Using this observation, it was suggested that w and t could be varied automatically starting off with a large w and t, then reducing both values with time.

2.4.5 ***Radial Basis Functions***

A radial basis function feature usually has the form $\phi_s(i) = e^{-\left(\frac{\|s-c_i\|^2}{2\sigma_i^2}\right)}$ where *i* implies that this is the *i*th feature and c_i and σ_i are the parameters for the *i*th feature (Sutton & Barto, 1998). Since *s* and c_i are vectors, $||s-c_i||$ represents the norm of the difference between the input vector *s* and the prototypical (centre) state associated with cluster *i* (c_i). These centres can be determined either heuristically or the can be obtained using supervised or unsupervised learning (Wettschereck & Dietterich, 1992). The variance values σ_i can also be learnt or set to fixed values. Assuming that there are *n* features, each of the features is weighted using adjustable weights in order to get the value for a given state (Menache et al., 2005). The purpose of learning is to determine the optimal values for the weights.

The value of a given state V(s) is obtained by summing the weighted features as shown in Equation 2.16.

$$V(s) = \sum_{i-1}^{n} \theta_i \phi_i(s)$$ *Equation 2.16*

Where θ_i are the weights associated with each feature. Radial basis functions are therefore categorised as a linear function approximation technique since the value is a sum of the values of the features. Radial basis functions have the advantage of being localised and therefore tend to be less likely to be affected by the unlearning problem (Cetina, 2008).

Papierok et al.(2008) used radial basis function networks in solving a robot navigation problem. The structure of the network was such that each of the actions had its own weight parameters but the actions shared the radial basis functions as illustrated in Figure 2.4.

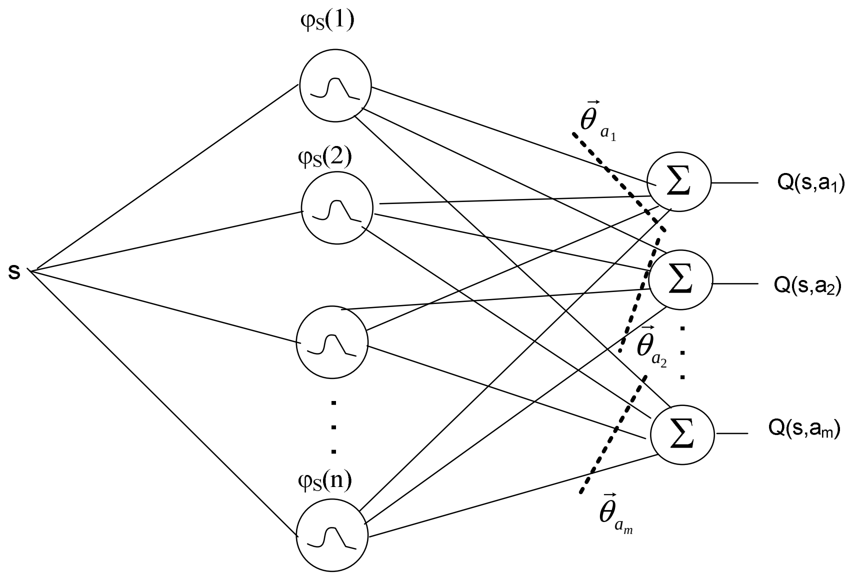


Figure 2.4: Radial Basis function network. Source (Papierok et al., 2008)

In Papierok et al.(2008), the radial basis function implementation was shown to result in better performance compared to course coding.

2.5 Learning Speedup Methods in Reinforcement Learning

It is necessary to speedup the learning process in RL in order to increase the likelihood of acceptance of autonomous agents based on RL learning and also to reduce the learning cost. This should also help agents adapt quickly to new situations. Several methods have been applied in an attempt to achieve the speedup. Two of these techniques are experience replay and knowledge reuse. The following elaborates further on these two methods and how they have been applied.

2.5.1 *Experience Replay*

2.5.1.1 Introduction

Classical reinforcement learning algorithms can be said to be wasteful since experience is presented to the learning mechanism just once then discarded (Lin, 1992). Experience Replay is a technique applied to speed up reinforcement learning by repeatedly presenting experienced data to the reinforcement learning algorithm (Adam et al., 2012; Lin, 1992). The objective of experience replay is for the control agent to achieve acceptable performance after only a brief interaction with the environment. Online learning occurs in the real world where every action takes time to complete and for its effects to be felt. Thus the learning rate is impacted by the time taken by events in the real world. When experience replay is used, the application of the learning experience to the learning algorithm is only affected by the processing speed of the computer.

A draw back of experience replay is that data from several episodes will need to be stored in order for it to be presented later to the learning algorithm (Adam et al., 2012). Experience replay may also suffer from the problem of over-training if the experiences are replayed too many times. In addition, if too many of the experiences are replayed, this may become too expensive computationally. Lin(1992) solved this problem by randomly selecting N samples from a list of 100 most recent experiences where N was a decreasing number between 12 and 4. Another drawback is that experience replay is useful where the laws governing the environment of the learning agent are stable (stationary environments). This means that the stored experiences might become irrelevant in an environment with new laws.

2.5.1.2 Experience Replay Examples

Adam et al.(2012) tested experience replay in the inverted pendulum task. Given the best performance achieved without ER, the algorithm applying ER was able to attain the best performance attained without using ER in about a quarter of the time. The ER algorithm went further and exceeded the best performance of the non-ER algorithm by far. This performance was replicated in two other test tasks which were: 1) stabilization of a two-link robotic manipulator and 2) Robotic soccer goalkeeper. In Adam et al.(2012), both the SARSA and the Q-Learning algorithms were tested. However, the policy was represented using linear parameters. It would therefore be desirable to know if a similar performance could be observed if neural networks are used to represent the policy.

Lin(1992) showed that experience replay achieved a speed up in learning time in a game like environment where the agent was supposed to locate food while avoiding enemies and obstacles. In the first 50 episodes alone, the ER based algorithm would get food an average of 10 times per play, while the non ER algorithm got food an average of 4 times per play. In addition, Lin combined experience replay with a form of knowledge transfer known as imitation, where the knowledge was transferred from a human expert called a teacher. The learning agent observed as the teacher acted in the environment. The agent then stored these experiences and replayed them later on when it started learning autonomously. A combination of experience replay with teaching was found to give better performance than when experience replay was used alone (Lin, 1992).

In the experiments in Lin(1992), the learning algorithms that were combined with experience replay were Q-Learning and adaptive heuristic critic (AHC) with the policy represented using neural networks. The SARSA algorithm was not used. The Q-Learning algorithm outperformed the AHC algorithm both when ER was used and when it was not used. It would therefore be desirable to find out if the SARSA algorithm would benefit as much from experience replay especially when neural networks are used for the policy representation.

Lin(1991) used experience replay several robotics tasks. These were: wall following, door passing and charger docking. Teaching(a form of knowledge transfer from a human expert) was also integrated into the learning. ER was used in all experiments so the performance of the algorithms in the tasks when ER was not used was not determined. In addition, only the Q-Learning algorithm was used in the experiments. The SARSA algorithm was not.

Kalyanakrishnan & Stone(2007) used ER in the Keepaway subtask of RoboCup with three keepers and two takers. ER was implemented in batch meaning that the learning agent would collect experiences for 50 episodes (this was reffered to as a batch of experiences); This batch of experiences were used to update the learning agent's function approximator in several iterations of training. The function approximators used to represent the value function were neural networks and CMAC. Experience replay using neural networks representation resulted in a better performance than experience replay using CMAC representation. Without use of experience replay the CMAC algorithm resulted in better performance than the neural network algorithm. In all cases the ER algorithms had better performance than the non-ER algorithms with the ER algorithms achieving performances in 400 episodes that the non-ER algorithms could not achieve in 5000 episodes.

In Kalyanakrishnan & Stone(2007), the SARSA learning algorithm was used for training. The neural network updates were performed using the backpropagation algorithm (Mitchel, 1997). Only the latest 50 episodes were used for training. This strategy runs the risk of forgetting experiences it had learnt in earlier batches. In addition, online learning did not take place. The update was performed only after all the fifty episodes and been collected so that they could be batched together and used to update the policy over several iterations. Online learning may be advantageous in that it ensures that the policy improves after every action.

2.5.1.3 Experience Replay in Animals

A significant amount of evidence has been put forward to support the idea that experience replay does occur in humans and animals (Wamsley et al., 2010). In Rodents, neural activities recorded during exploration are observed again in sleeping animals. This generally occurs during non-rapid eye movement sleep (NREM). This replay has been associated with better performance in memory recall tasks.

In animals, replay has been defined as "The sequential reactivation of hippocampal place cells that represent previously experienced behaviour trajectories" (Carr et al., 2011). These learned sequences are replayed in compressed timescales and are hypothesized to be essential to consolidate memory. The replay occurs during both awake and sleeping states. In the awake state, this is more likely to occur during periods of rest, grooming and consumption activities (Carr et al., 2011).

In animals, behavioural experiences can be replayed many times, even if the experience is encountered only once. After a novel behaviour that leads to attainment if a reward, the experience is seen to be replayed in reverse. Behaviour replay seems to link recently experienced sequences to their outcomes which may promote learning. Experience replay is also seen to occur more in novel environments and in sequences that lead to rewarding states (Carr et al., 2011). These can be seen to be salient events which are important to recall.

A recent study on replay of past experiences in rats (National Institute of Mental Health, 2012) found that blocking replay by blocking sharp-wave ripple (SWR) activity led to deterioration in performance in a maze task. The impairment was observed in the ability to link immediate and past experiences to rewards. The SWR were suppressed by using mild electrical simulation whenever ripple activity was detected.

The studies discussed above suggest that experience replay occurs in animals. One of its effects seems to be consolidation of memories for the purpose of recall. It also helps in thinking about future possibilities based on past experiences and helps in deciding what to do. In this regard, "stored associations can be retrieved to guide ongoing behaviour" Carr et al. (2011). Replay also seems to occur more when novel situations are encountered and in after salient events. This would imply a mechanism for determining what are the novel and salient events that would require more application of replay.

2.5.2 *Knowledge Transfer*

Tabula Rasa learning is the term applied to refer to reinforcement learning techniques that start off with a random policy (Taylor & Stone, 2009). An agent that starts of the learning process with a random policy will initially be capable of only choosing random actions. Such an agent's initial performance is expected to be very poor. This is because the agent can only arrive at the goal by chance (Lin, 1992). Knowledge transfer is a technique used to reinitialise the learning agent's policy so that it is not completely random at the beginning of learning. Usually, the knowledge used for initialisation can come from several sources including: A human mentor, heuristic knowledge incorporated into the learning algorithm, and knowledge obtained from a previously learnt task, among others.

Studies on human subjects have also provided a justification for the use of knowledge transfer. For instance Lam & Dietz (2004) showed that when a human subject learned to walk over obstacles on a level treadmill without visual information, the subject reused the knowledge in two related tasks viz. stepping over obstacles while walking down hill, and stepping over obstacles with a weight added to one leg.

Erni & Dietz (2001) investigated cross modal transfer in humans. Cross modal transfer occurs when skills learnt for performing a given motor task with specific stimulus information, are reused in performing the task when the stimulus information changes. They found that when the stimulus was changed from acoustic to somatosensory (touch based), full cross modal transfer of skills occurred. Change of stimuli from visual to other modalities, or from other modalities to visual did not result in significant cross modal transfer.

According to Taylor & Stone (2009), a reinforcement learning transfer agent will have to perform all of the following tasks:

i. Given a target task, select an appropriate source task or set of related tasks from which to transfer
ii. Learn how the source task and the target task are related
iii. Effectively transfer knowledge from source task to target task

Taylor & Stone (2009) list the following metrics to measure the benefits of transfer.

i. Jumpstart – related to improvement in initial performance of an agent in a task compared to the performance of an agent that does not use transfer.
ii. Asymptotic performance – The final learned performance of an agent in the target task may be improved via transfer.
iii. Total reward – total reward accumulated by an agent may be improved via transfer.
iv. Transfer ratio – The ratio of the total reward accumulated by the transfer learner and the total reward accumulated by the non-transfer learner
v. Time to threshold – Time needed to achieve pre-specified performance levels

The average reward obtained is another metric that can be used to quantify the performance of a knowledge transfer technique (Fernandez & Veloso, 2006).

According to Taylor & Stone (2009) the following are possible differences between tasks involved in Transfer Learning:

i. Transition functions

The transition function for a task is defined as **T(s,a,s′)** which is the probability of a transition from state **s** to state **s′** when action **a** is taken in state **s**. In a new task, this probability may be different.

ii. State spaces

The state space can be changed by for instance changing the domain of the state variables.

iii. Start states

The start state may change leading to a change in the task. For instance in a maze navigation task the goal may be fixed, but the starting position in the maze may vary leading to a change in the task (Fernandez & Veloso, 2006).

iv. Goal states

In a maze navigation task, the starting position may remain the same but the destination room may change.

v. State variables

The state space in a task can change for instance by introduction of more sensors in a robot control task. Other variables that can be introduced may include ability to recall some previous actions. Taylor & Stone (2009) suggest that the previous setting of the velocity of the car in the mountain car task may be introduced as a new state variable. Adam et al. (2012) simplify the robotic soccer goalkeeper problem by eliminating some of the state variables.

vi. Reward functions

The reward functions is usually represented as **R(s,a,s′)** which is interpreted as the reward obtained after taking action **a** in state **s** and transition to state **s′**. The task can be changed by having either a dense or a sparse reward function (Zhai et al., 2009). A task with a dense reward function is easier for the agent to learn when compared to a sparse reward function. Designing a dense reward function is a task that is left to the designer thus making the designers work more difficult.

vii. Action sets

A task can be changed by having more or less actions. A task with more actions will usually be more difficult and take longer to learn compared to an action with fewer actions. The number of possible policies grows exponentially with the number of possible actions hence limiting the number of actions makes the optimization problem simpler (Erez & Smart, 2008).

In Reinforcement learning, several techniques have been applied to achieve transfer of knowledge. Following is an elaboration on the most commonly used methods.

2.5.2.1 Reward Shaping

If an agent takes too long to obtain a reward after it has taken some action, it may fail to associate the reward with an action it took many steps in the past (Laud, 2004). This may cause the learning process to breakdown. Reward shaping involves training the agent on a different reward signal rather than R (Taylor & Stone, 2009). Laud (2004) defines reward shaping as a method that provides localized feedback based on previous knowledge to guide the learning process. In Reward shaping, the reward function is defined in such a way that progress towards the goal is rewarded (Laud, 2004). One way of achieving shaping is by decomposition of the task into subtasks, each with its own reward function (Singh, 1992). Another way of implementing reward shaping is by adding an artificial reward F to the native task reward.

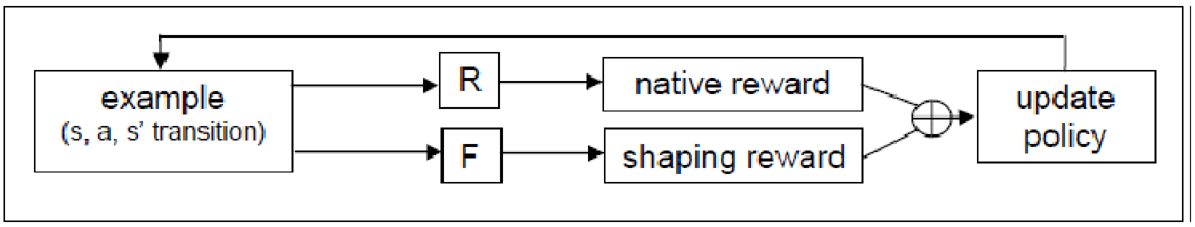


Figure 2.5: Implementation of reward shaping by addition of an artificial reward. Source (Laud, 2004)

F could for instance be derived from the potential function φ(s) that decreases with increase in distance from the goal as shown in Equation 2.17.

$$F = \varphi(s') - \varphi(s) \quad \textit{Equation 2.17}$$

The main disadvantage of the reward shaping strategy is that it is expected to be used in intra-domain transfer (Frommberger, 2010). If the state space or actions space changes, then this technique cannot be used.

2.5.2.2 Imitation

An agent can accelerate the reinforcement learning process through observation of an expert mentor or mentors. In the model referred to as *implicit imitation*, the agent observes the state transitions induced by the mentor's actions, and uses the observations gleaned from these observations to update the estimated value of its own states and actions (Price & Boutilier, 2003). A mentor can also focus the agent's attention to the most interesting areas for instance the areas that contain the goal.

Conn & Peters (n.d) use human supervision to train a robot to move from one corridor in a building, to a designated location in another corridor. This is achieved via teleoperation of the mobile robot. During the teleoperation, the agent learns a policy for accomplishing the task. Using this policy alone, the agent was always able to accomplish the task i.e. the agent could find the goal with a probability of one. Subsequently, the agent learns its own policy using an ε-greedy approach, but relies on the policy learnt from the human 50% of the time in selecting actions. New obstacles are also introduced in the environment to test agent's robustness to changes in its environment. The main result of this experiment was to show that higher learning rates (α) generate faster convergence to the optimal policy (derived from human training), but also, a method for use of supervision in reinforcement learning is introduced.

Smart & Kaelbling (2002) divide the learning task into 2 phases. In the first phase, example trajectories provided either by control code or by having a human being in control of the robot are used to guide the robot in the task. The reinforcement learning agent observes the actions, resulting states and rewards and uses them to update its value function. In the second phase, the reinforcement learning agent takes responsibility of guiding the robot. This method ensures that in the second phase, the control agent is able to move to rewarding states hence it is able to learn. Without the bootstrapping, the agent will hardly ever find any rewards and hence will take much longer to learn. The proposed learning framework is evaluated in an obstacle avoidance task and a corridor following task.

Lin(1991) used imitation (which he referred to as teaching) in several robotics tasks. These were: wall following, door passing and charger docking. Experience replay was was integrated into the learning algorithms. The experiments therefore showed the performance when ER was used alone or ER when was used together with teaching. The performance of the two algorithms was found to be the similar in the wall following task. In the door passing task, the algorithm that used teaching had better performance. In the charger docking task, the algorithm that did not use teaching could not learn the task, while the one that used teaching eventually learnt to perform the task.

The major limitation of imitation is that it requires a human trainer to provide some example lessons or experiences. For some problems, this may be necessary because the agent has a very small chance of ever finding the goal by chance. The charger docking task in Lin (1991) is an example of such a task. For tasks where the agent has a good chance of finding the goal by chance, other knowledge transfer methods that do not rely on a human trainer can be adopted.

2.5.2.3 Directed Training

Directed training is a technique to speedup learning whereby a human changes the task by modifying the transition function *T*. Using this method, a human supervisor can incrementally make the task more difficult, but the policy obtained from earlier training is used as the initial policy for the more difficult task (Taylor & Stone, 2005).

Selfridge et al. (1985) show the benefits of directed training in the pole balancing task. In the said task, a cart that moves only in one dimension in a space of limited length is supposed to learn to keep a pole upright. Increasing the mass of the pole is one way to make the task harder. The agent required less training when it trained on a lighter pole, then used the policy gained from the lighter pole to learn a policy for a heavier pole compared to when it learned the policy for the heavier pole from scratch. Similar benefits were observed when the task was adapted by making the pole shorter, with the longer pole length task being easier than the shorter pole length task. Great benefits (shorter training time) were also observed when the task was adapted by decreasing the length of space in which the cart moves. The longer the length of the space, the easier the task became.

This strategy is mostly suited for situations in which the task is changed by modifying the transition function. In such a case, the effect similar actions will be different in the source and the target task. For obstacle avoidance, this can be used for instance when the terrain is changed.

2.5.2.4 Learning from easier missions

Learning from easy missions allows a human to change the start state of the learner making the task incrementally harder. For instance an agent can be placed near the exit of the maze in the initial case and then placed further and further from the exit subsequently making the task harder. This enables the agent to spend less time learning to perform the final task (Taylor & Stone, 2005).

In Asada et al. (1994), a robot is trained to shoot a ball into a goal by initially having the robot learn to shoot from close range to the goal. This is after it was found that when the distance to the goal was too long, the robot could not learn the desired policy even after training for a long period of time. After initially learning to shoot from close range, the robot was eventually able to learn to score from longer distances.

This technique is also most situated for situations where the agent takes too much time in attaining the goal by chance. The task may be made easier for the agent to achieve the goal after which the more complex task can be introduced.

2.5.2.5 Hierarchical Reinforcement learning

Hierarchical RL allows transfer of learned subroutines. By analyzing two tasks, subroutines may be identified which are common to the two tasks, hence subroutines learnt in the first task can be reused in the second task hence reducing the learning time (Taylor & Stone, 2005).

Singh et al. (2004) point out that autonomous mental development should lead to development of reusable skills, where a skill corresponds to a subroutine that directs an agent's behaviour for a subset of the environmental states. Skills can be implemented using options where options have: 1) *an option policy* – directs the agent's behaviour for a subset of the environmental states, 2) *an initiation set* – consists of the states in which an option can be initiated and 3) *termination condition* – specifies the condition under which an option terminates. Because options can invoke other options as actions, this leads to development of a hierarchy of skills. The advantage of options is that they can be reused in different tasks. Another advantage is that each option can be learnt separately, subsequently, several of the options can be combined to perform an overall task. This can lead to learning to perform the overall task faster than if it was learnt as a single skill.

The MAXQ algorithm of Dietterich (1998) is hierarchical learning algorithm that decomposes a task into hierarchical subtasks. The main task can be accomplished by making use of a set of subtasks. The subtasks can are also accomplished by making use of a set of smaller subtasks. Thus there is a recursive definition of subtasks based on the subtasks that can be used to solve the subtask. This definition stops when we encounter a subtask that corresponds or can be solved by one primitive action. The programmer is responsible for defining the hierarchy of subtasks.

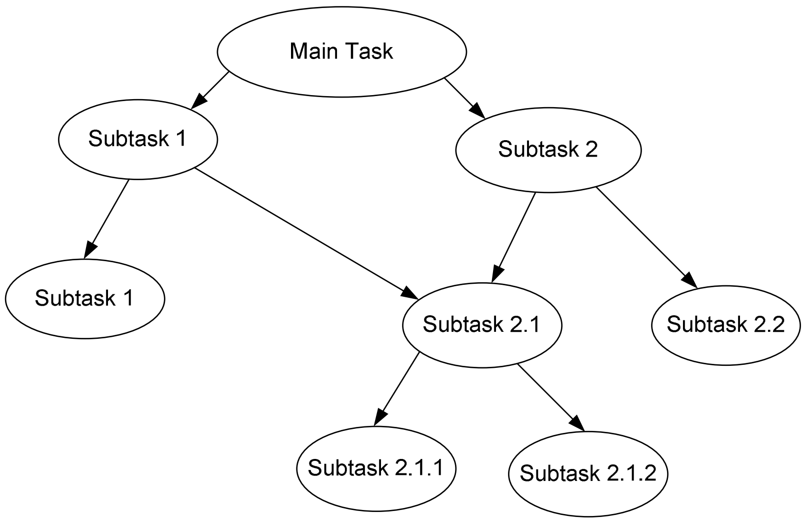


Figure 2.6: Example task hierarchy

Some subtasks are shared by subtasks higher in the hierarchy. E.g. subtask 2.1 in Figure 2.6. Thus the knowledge acquired in perfuming subtask 2.1 when subtask 1 is being learnt, can be reused when subtask 2 is being performed. MAXQ is an example of an intra-domain knowledge transfer technique. MAXQ has been shown to lead to much faster learning when compared to learning with no task structure imposed on the domain.

In Drummond (2002), a technique for accelerating RL by composing solutions of automatically identified subtasks is proposed. The proposed technique first learns the optimal policy for an initial example task in the domain. The state space and the value function are then partitioned with different parts of the value function being assigned different partitions of the state space. Partitions of the state space that are found in subsequent tasks can have the parts of the value functions associated with them being reused. This leads to faster learning in new tasks that have significant overlap with previously learned tasks. The main disadvantage of this technique is that it can only be applied in intra-domain knowledge transfer. In addition, it requires the tasks in the domain to have clearly identifiable subtasks that are shared by different tasks.

Hierarchical RL is well suited for tasks that can be decomposed into a hierarchy of subtask with shared subtasks within the bigger subtasks. Unfortunately, obstacle avoidance cannot be decomposed this way. This is because it has only two subtasks which are move to goal and avoid obstacles. Avoid obstacles is a subtask of move to goal but it is not needed in any other subtask.

2.5.2.6 Policy Transfer via Inter-Task mapping

Another technique for performing knowledge transfer is policy transfer via inter-task mapping (Taylor et al., 2007). This involves transforming the source policy to a form in which it is usable by the target task. The source policy (π_s) maps a state s in the source task ($s \in S_s$) to an action a in the source task ($a \in A_s$) ($\pi_s: S_s \rightarrow A_s$). Policy transfer involves transforming π_s so that it can have states in the target task (S_t) as inputs and actions in the target task (A_t) as outputs. Thus, π_s needs to be transformed to π_t which is a target task policy such that $\pi_t : S_t \rightarrow A_t$. This can be challenging because the state space for the source task may be different from the state space for the target task. This means that the set of inputs for the source task may be different from the set of inputs for the target task. The states may differ in several ways including: Having a different number of input variables, the number of variables may be the same but differ semantically, the variables may be the same but the allowed values for the variables may differ. The set of actions for the two tasks may also be different. For instance $|A_t|$ may be greater than $|A_s|$.

In Policy search algorithms, a policy is usually identified by specific assignment of values to a vector of parameters such that policy $\theta = (\theta_0, \theta_1, \dots, \theta_{n-1})$ (Peshkin, 2001). As an example, if a neural network is used for policy representation, θ corresponds to the set of weights in the links between the nodes of the neural network. These determine the output of the neural network for a given state and action. The output can be used to calculate the probability of selecting a given action.

Based on the parameterised representation of the policy, the source task policy may have different parameters from the target task policy. For instance in a neural network representation, if the target task has more input variables than the source task, the target task network will have more input nodes than the target task. Consequently, even if the number of hidden nodes are the same, the number of weights connecting the input layer to the hidden layer in both networks will differ. Thus the source task policy network cannot be directly used as the target task policy network.

Continuing with the neural network example, each of the input nodes (corresponding to an input variable) is connected to a subset of the weights which determine it influence on the final output. Policy search involves finding an assignment of values to the weights that result in a good performance in the agent. In a task where no learning has taken place, the values of the parameters are random and are unlikely to represent a well performing policy. Policy transfer usually involves initialising the parameters in the target task policy with the values of the parameters in the source task policy after learning has already occurred in the source task. Thus, given a parameter θ_{ti} in the target task, we need to know the parameter θ_{sj} in the source task should be used to initialise θ_{ti}. Since each input variable has its associated set of parameters, if there is a variable *x* that is common in both the source task and the target task, then the parameters associated with *x* in the source task, are used to initialise the parameters associated with *x* in the target task. The same applies for actions. If there is an action *a* that exist in both the source task and the target task, the parameters associated with *a* in the source task are used to initialise the parameters associated with *a* in the target task.

For those input variables and actions in the target task that do not exist in the source task (for instance the newly added variables), then more work needs to be done. For a variable *y* in the target task that does not have a matching variable in the source task, there is need to identify a variable *x* in the source task which is similar to *y* so that the values for the parameters associated with *x* can be used to initialise the parameters associated with *y*. This can simply be stated as the problem of finding a variable *x* in the source task that is similar to the variable *y* in the target task.

Taylor et al. (2007) use three methods to solve the problem of matching input variables and actions in the target task to similar input variables and actions in the source task. These are:

i. Hand coded mapping

Given a variable *y* in the target task that does not have a matching variable in the source task, a human expert who is knowledgeable in the task domain can decide which variable *x* in the source task is semantically most similar to *y*. This is the preferred method of mapping and was shown to produce the best results after transfer in the Keepaway subtask of robotic soccer. Note that several variables in the target task can be mapped into the same variable in the source task. This means that their parameters will be initialised with the same values.

ii. Incomplete mapping

For target task variables that do not have matching variables in the source task, initialise their parameters with random weights. This also gives better performance compared to when no transfer takes place. This is because the variables in the target task that have matching source task variables have their parameters values initialised with parameter values of corresponding source task variables, hence not all parameter values are random.

iii. Supervised learning

Supervised learning is used in an attempt to automate the variable mapping process. This involves training a classifier in the source task to identify variables and/or actions. For instance given a tuple $<s_t,a_t,s_{t+1},r_{t+1}>$ representing an experience in the source task such that s_t is the state at time *t*, a_t is the action taken at time *t*, s_{t+1} is the state transitioned to after taking action a_t in state s_t and r_{t+1} is the reward obtained after making this transition, a classifier can be trained which takes $s_t,,s_{t+1},r_{t+1}$ as inputs and outputs the associated action. This means that a_t becomes the target of the classifier. In the target task, a similar sample is obtained by taking action a_T which is a target task. The sample is fed as input to the classifier trained in the source task which outputs a source task action a_S. In this case, we can say that the classifier has mapped a_T to a_S. We will need to obtain several samples in the target task and carryout the mapping, while keeping count of how many times a_T is mapped to a_S or any of the other source actions. Ultimately, the source action that a_T is most frequently mapped to by the classifier will be the action most similar to a_T.

For mapping state variables using supervised learning, it may be necessary to group the state variables into clusters, where each cluster represents a semantic concept e.g. position. In the Figure 2.7 (representing a robotic arm), the state variables may represent the 3D position of the three labelled points.

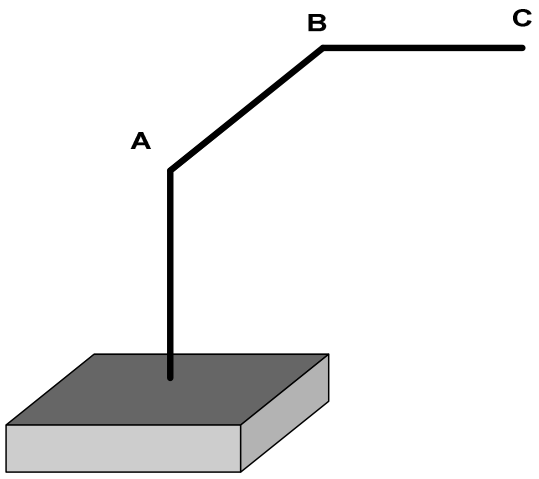


Figure 2.7: Robotic Arm

We will need to have nine state variables that is *(Ax, Ay, Az, Bx, By, Bz, Cx, Cy, Cz).* Thus, we can have a cluster to represent the concept of robotic joint position. Note that these could just be a subset of the state variables. We can add another variable to represent the weight of the load (if the robotic arm is for lifting loads), hence the state variables will be *(Ax, Ay, Az, Bx, By, Bz, Cx, Cy, Cz, W)* where W represents the weight of the load. This means that our state represents two concepts i.e. position and weight. The position concept has three possible objects, that is either *A, B* or *C*. Instead of training a classifier to identifier a particular state variable, it can be trained to identify the object i.e. either *A, B* or *C*. The classifier will be trained in such a way that given the values of the variables corresponding to the particular object e.g. *Ax, Ay, Az* for object *A* in both s_t and s_{t+1} then its output should be *A* (or an index for *A*). Given *Bx, By, Bz* then the classifier should output *B* and so on. If therefore the classifier is given *Ax, Ay, Az* and it outputs *B*, then this is a wrong classification and it needs further training. All this happens in the source task.

Lets say in the target task we have four joints i.e. *A, B, C* and *D*. Our goal would be to map *D* to either *A, B,* or *C*. i.e. the joint that is most similar to joint *D*. When the state variables associated with *D (Dx, Dy, Dz)* are fed to the classifier trained in the source task, the output will be either *A, B* or *C*. If we do this several times, then *D* will be mapped more times to either of *A, B,* or *C*. The object that *D* will be most frequently mapped to, will be considered to be the most similar to *D*.

Some amount of time will be spent training the classifier in the source task. Also, some time will be spent gathering samples in the target task to be used to perform the mapping. Taylor et al. (2007) argue that this time is negligible compared to the total training time.

Taylor & Stone (2005) use *Keepaway,* a subproblem of RoboCup soccer to test the performance of the transfer of the value function. The policy was represented using a Cerebellar Model Articulator Controller (CMAC). In the Keepaway task, there are two antagonistic teams. The first team is known as the keepers while the second team is known as the takers. The keepers' task is to keep the ball between themselves by passing to members in the same team. The takers try to take the ball away from the keepers or force it from the field of play. An episode ends when the takers successfully intercept the ball or the ball goes out of the filed of play. The takers use a hard coded policy while the keepers need to learn the optimal actions to take. For instance in a 3vs2 game, the actions available to a keeper is to pass the ball to any of its two team mates or continue holding the ball. A keeper needs to learn which of these three actions is optimal in any particular state.

When transfer is performed from a 3vs2 task to a 4vs3 task, a state action pair in the 4vs3 task is mapped to the most similar state action pair in the 3vs2 and the associated weights in the 3vs2 CMAC tiles are copied into the 4vs3 CMAC tiles.

Taylor & Stone (2005) show that the total training time needed to achieve a pre-specified performance (keepers retaining the ball for 9 seconds), is reduced when transfer is used compared to when the 4vs3 keepers have to be trained from scratch. There results also indicate that the initial performance (ball hold time) for the keepers that benefit from transfer does not differ from the keepers that benefit from transfer. The keepers that benefit from transfer however learn faster with training time being reduced from about 15 to 4 hours. When transfer is performed from 3vs2 to 4vs3 then to 5vs4 Keepaway, the total training time for 5vs4 Keepaway is reduced by 52%.

Taylor et al.(2007) define transfer via Inter-Task Mappings for Policy Search (TVITM-PS) for the case where the policy is represented using a neural network action. Link weights are copied from the source task and used to initialise the links of the target task neural network.

The evaluation of the target agent in the Keepaway and Server Job Scheduling tasks show that the target agent is able to learn faster when transfer is used than when the target task has to be learned from scratch using the time to threshold performance measure. This happens even when the time needed to train in the source task is taken into account. The initial performance of the agents is not reported.

Taylor & Stone (2005) and Taylor et al. (2007) show the usefulness of the policy transfer technique in the Keepaway task. The use of this technique in the obstacle avoidance task has not been studied. The obstacle avoidance task is different from keepaway in that in keepaway, the agent gets a reward for every time step (after every action) which results in keepers retaining the ball, while in obstacle avoidance, the agent gets a reward at the end of the episode. The temporal credit assignment problem is therefore more of a concern in the obstacle avoidance task. In addition, in Taylor et al. (2007), a neural evolutionary algorithm known as NeuroEvolution of Augmenting Topologies (NEAT) was used to create the neural network structure and updated the weights of the network. Another alternative method that has not been attempted is the use of policy transfer when the TDBackpropagation algorithm is used to update the neural network weights.

2.5.2.7 Policy reuse

Policy reuse is another technique used to reduce the learning time by providing policies learnt in previous tasks to be used in the new task (Fernandez & Veloso, 2006). Policy reuse can be applied in tasks within the same domain or to tasks in different domains. In policy reuse, the control agent may probabilistically either use the source policy or the target policy to select actions in the target task. The experience is used to update the target task policy. This means that learning is only taking place in the target policy.

i. *Tasks in the same Domain*

When the tasks are in the same domain, they have the same state and action space but may have different reward functions. Fernandez & Veloso (2006) show application of policy reuse in a maze navigation domain. A robot is expected to navigate a maze to reach to one of the rooms (the goal) in the maze (See Figure 2.8). The tasks are differentiated by changing the goal room. As an example, the optimal policy to reach room 1 will be different from the optimal policy to reach room 2. In this case no inter-task mapping needs to take place because the input of the source policy is the same as the input of the target policy. The actions available to the two policies are also the same.

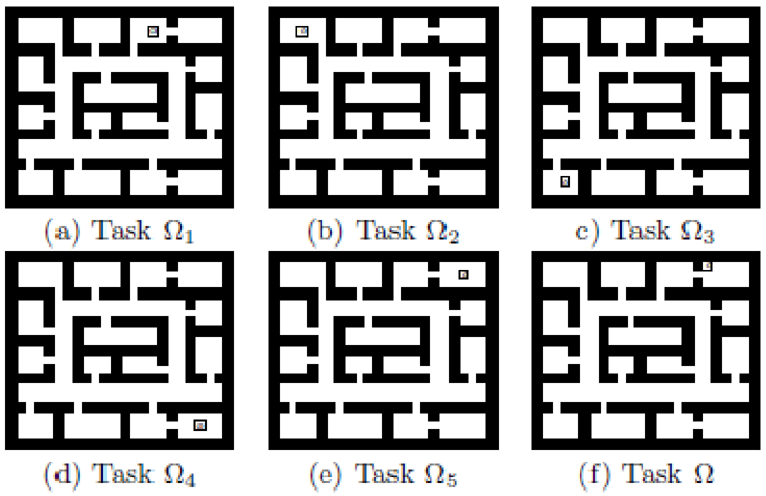


Figure 2.8: Several different tasks in a maze. The small rectangle in the maze shows the goal cell. Source (Fernández et al., 2010).

For instance given Figure 2.8, after learning task Ω_1, we can reuse the policy learnt in learning task Ω. Thus task Ω will not be learnt from scratch. Note that the new policy is not initialised with values from the past policy, however, it is hoped that the source policy will increase the probability of the right actions being selected. The result of these actions will be used for learning in the new policy.

Their experiments show that an agent benefits more by reusing a policy for getting to a room that is close to the goal room if it exists. The performance of policy reuse is compared to when the agent uses the ε-greedy strategy. Their results show that policy reuse provides significant gain in performance compared to the ε-greedy strategy after a few learning episodes when a policy close to the one for the current task is reused, but a deterioration in performance when a policy too far from the current task's policy is used. They define an algorithm π-reuse that can be used to reuse a policy.

Fernandez & Veloso (2006) also define the Policy Reuse Q-Learning Algorithm (PRQ-learning) that uses a library of policies to help in learning the policy for a new task. The algorithm is able to bias the learning so that it relies more on the policies in the library that lead to the highest gains in the new task.

The Policy Library through Policy Reuse Algorithm (PLPR) not only learns a new policy, but also adds it to the library if it is significantly different from the policies currently existing in the library. Thus the algorithm can be used for learning the core policies of a given domain.

The advantage of the policy reuse approach is that several policies can be re-used at the same time with each of the past policies being used probabilistically. It is also possible to compute a value referred to as reuse *gain* to determine which re-used policies are most beneficial. Consequently an agent can bias the selection of past policies such that the most beneficial policies are most frequently reused.

In Fernandez & Veloso (2006), the Q-Learning algorithm was used to learn the value function using policy reuse. The authors do not however indicate which function approximation mechanism was used. But since this was a grid based environment with a finite state space and actions space, it is very likely that table based method was used for the representation of the value function. It would therefore be desirable to determined the performance when the SARSA algorithm is used for learning, and also when other function representation methods are used for learning.

Bergamo et al. (n.d.) perform knowledge transfer by reusing abstract policies. In order to use abstract policies, abstract states are defined to represent similar semantic concepts. For instance if we have two rooms *class* and *office*, an abstract state s can be defined as s={inC(A), room(A)}: which means that the robot is at the centre of room A . *A* can then be instantiated by either *class* or *office*. Abstract actions can also be defined which are used to move from one abstract state to another. If for instance we have a predicate *door/2*(which identifies a door to a given room), we can define an abstract action to move to the door from the centre of the room. This can be represented as *{inC(A), room(A),door(A,D)}→moveCD(D).* With this representation, A can be initialised to any room, and D can be initialised to any door within the selected room. The assumption is that the once the *moveCD* action is learnt for one room, it will be applied in a similar way to other rooms.

Bergamo et al. (n.d) applied this technique to a robot navigation task which consisted several rooms linked by doors and corridors. The optimal policy was first learnt by defining a fixed start and goal room. This optimal policy was then converted to an abstract policy. The abstract policy was probabilistically reused when the start and goal rooms were changed. It shown that learning was significantly faster in learning new tasks when the abstract policies were used for learning than when normal reinforcement learning was used.

The main disadvantage of the technique introduced by Bergamo et al. (n.d) is that the state space needs to be finite or discretized. This is because the variables can only be instantiated with objects in the environment that can be uniquely identified. This method also requires that the task be decomposed into subtasks that can be shared among different higher level tasks. The obstacle avoidance task cannot be decomposed this way.

Reuse options (Bernstein, 1999), are another technique that can be used to reuse policies from previously learnt tasks. In reuse options, several policies from previously learnt tasks in the same domain are combined to create one policy called a mixed policy which is a type of reuse option. When a reuse option is used for acting, the number of steps in which only the reuse option is used for acting is specified. Thus the reuse option represents a macro action. In Bernstein (1999), in a given state, either a one step action is selected by ordinary Q-Learning or a macro action is selected using the mixed policy. The macro action selected using the mixed policy is called an option. The resulting algorithm is called macro Q-Learning.

Bernstein (1999) showed that macro Q-Learning significantly outperformed ordinary Q-Learning in a grid navigation experiment and in a reformulated darts game problem. It was also observed that the length of the options also affected the speed of learning with longer length options resulting in better performance. In this study, only discrete state and action spaces were considered. The application of the technique in continuous state spaces was not investigated. The main assumption of this technique is that options can be shared between tasks. In general, options represent solutions to subtasks. The obstacle avoidance task cannot easily be divided into subtasks because only one main task (move to goal) exists, with the avoid obstacles subtask which does not have another main goal it can be shared with.

ii. *Tasks in different Domains*

Tasks are said to be in different domains if they have different state spaces, different action spaces, or different state and action spaces. Fernández et al. (2010) show that policy reuse can be applied when the state and action spaces of the source task differ from those of the target task. This requires a mapping of the state and | or action spaces of the source task and the target task. In Fernández et al. (2010), the mapping is predefined by a human expert (hand coded mapping).

This mapping is defined as defined as $\rho=\langle\rho_s, \rho_a\rangle$. ρ_s is a function that takes as input a state in the target task and outputs a state in the source task ρ_s: $S_t \rightarrow S_s$. ρ_a is a function that takes an action in the source task and maps it into an action in the target task ρ_s: $A_s \rightarrow A_t$. If we have a source policy π_s that we want to reuse while learning a target policy π_t, we can probabilistically use π_s to select actions while learning in the target task. To generate an action, the policy needs the current state s^t_t(target state at time t). $s^t_t \in S_{target}$ cannot be used directly as an input to the source policy π_s and has to be mapped to the source policy state space S_s. We therefore use the following steps to reuse a past policy to select the next action a^t_t given the current state s^t_t *(note that the superscript t represents the current time)*

```
Probabilistically decide if to use the past policy $\pi_s$ or the new policy $\pi_t$
If the new policy is selected
    $a^t_t= \pi_t(s^t_t)$
Else if the past policy is selected
    $a^t_t =\rho_a (\pi_s(\rho_s(s^t_t)))$
```

$\rho_s(s^t_t)$ generates a source policy state from a target policy state.

$\pi_s(\rho_s(s^t_t))$ selects a source task action using the source task policy and the source task state returned by $\rho_s(s^t_t)$

$\rho_a(\pi_s(\rho_s(s^t_t)))$ maps the source task action produce by π_s into a target task action.

The advantage of this approach is that we need not know how to convert the source policy into a target task policy. We just need to define mappings of the state spaces and action spaces $\rho=\langle\rho_s, \rho_a\rangle$. Another advantage is that we can reuse several past policies while using the gain parameter to determine which of the past policies is the most useful. Reuse of source policies from domains that are different from the target task domain has also been found to considerably reduce learning time in learning the target task (Fernández et al., 2010).

Fernandez & Veloso(2006) demonstrate policy reuse using the Keepaway subproblem of robotic soccer. They reuse policies from the 3vs2 Keepaway in the 4vs3 Keepaway. They also reuse the 3vs2 and 4vs3 policies in the 5vs4 Keepaway. For example in transferring from the 3vs2 to 4vs3 Keepaway, states in the 4vs3 Keepway are mapped into the 3vs2 state space. An action *a* is then selected using the 3vs2 policy. This action is then mapped into an action in the 4vs3 set of actions. The 4vs3 agent probabilistically chooses either to use the action from the 3vs2 policy, or an action selected by its own policy which it is learning. The probability of choosing an action from the 3vs2 policy reduces with time.

When training the 5vs4 agent, it can probabilistically choose an action from the 4vs3 policy, or from the 3vs2 policy once again with the appropriate state space and action space mappings.

The major result reported by Fernández et al. (2010) is that buy using policy reuse, the time required to arrive at a pre-specified performance is reduced, compared to when learning from scratch.

In Fernández et al. (2010), the learning algorithm applied is SARSA while the value function is represented using CMAC. The performance of policy reuse when other mechanisms are used for function representation needs to be investigated.

The major disadvantage of policy reuse is that inter-task mapping will need to be performed if the two tasks are in different domains. However, this is also its main advantage in that knowledge can be transferred when the source and target tasks are in different domains. Examples of such a situation in the obstacle avoidance task are when the state space changes or the actions space changes. Another weakness of policy reuse is that initially, the target policy is random; consequently, when it is used to select actions, it is very likely that the wrong actions will be selected.

2.6 Performance measures in reinforcement learning

Performance measures are used to determine if a learning algorithm is resulting in improved behaviour in the learning agent and the rate at which this improvement is occurring. This can be used to compare several learning techniques.

Lin (1992) compared the performance of several algorithms and their variants in a grid world. This was a world in which the agent needed to avoid enemies and obstacles, while trying to move to cells where there was food. The performance measure that was used was the number of positive rewards that the agent obtained. This is the number of times the agent moved into cells which contained food after a given amount of training.

Smart & Kaelbling (2002) used the number of steps a robot took to reach the goal as the evaluation metric in a corridor following task and an obstacle avoidance task. This is a measure of the length of the path that the robot takes to reach the specified goal. Two different algorithms can be compared after a preset number of training iterations. An additional metric of the number of times the goal is reached from a given starting position is used to measure the performance.

Fernandez & Veloso (2006) use a performance measure referred to as the gain which is defined as the average reward obtained after a given number of episodes. This is in a maze navigation domain where the robot is rewarded for reaching the goal room. The episode can also end if the maximum allowed number of actions in an episode is reached before the robot arrives at the goal. Given that the reward for an episode that ends successfully is one, while the reward for an episode that does not end successfully is zero, this can be seen to be the percentage of episodes that end successfully. The maximum number of episodes is fixed at 2000. An algorithm that has the greatest gain when the maximum number of training episodes is reached can be taken to be the better training algorithm.

From the above examples, it can be seen that the number of times the robot arrives at the goal is a common measure of performance. To compare different algorithms, the maximum number of training trials is set and the rewards obtained by different algorithms after the maximum episodes are reached are compared.

An alternative method is to evaluate how long the algorithms take to reach a given performance level (Taylor et al., 2007). The disadvantage of this approach is that it may take too long before the desired performance is attained. In fact there is no guarantee that the required performance level will be reached.

2.7 The Obstacle Avoidance Task

In avoiding obstacles, the control agent needs to decide which way to turn and by how much (Touzet, 1997). That is, the agent must decide the direction of turning, and when the robot should stop turning and move forward.

In addition to avoiding obstacles, an agent also needs to maximize the speed of the robot (Knudson & Tumer, 2012). Thus, in avoiding obstacles, the robot should not follow a path that is unnecessarily long. The control agent should also select actions that make the robot move faster towards the goal.

In humans, when stepping over obstacles, the amount of foot clearance should be minimised (Lam & Dietz, 2004). This implies that in robot obstacle avoidance, the distance from the obstacle should also be minimized. This can also help in minimizing the amount of energy that is utilized in addition to increasing the speed of the robot.

The DARPA Robotics Challenge (DRC) is a prize competition funded by the Defence Advanced Research Projects Agency (DARPA) that aims to develop semiautonomous ground robots that can perform complex tasks (DARPA DRC Team, 2013). One of the tasks in the challenge is to drive a vehicle through a 250 feet long course in which obstacles have been placed. This is illustrated in Figure 2.9. The vehicle is expected to show autonomy in decision making and perception among other things. This shows that this is still an important area of research.

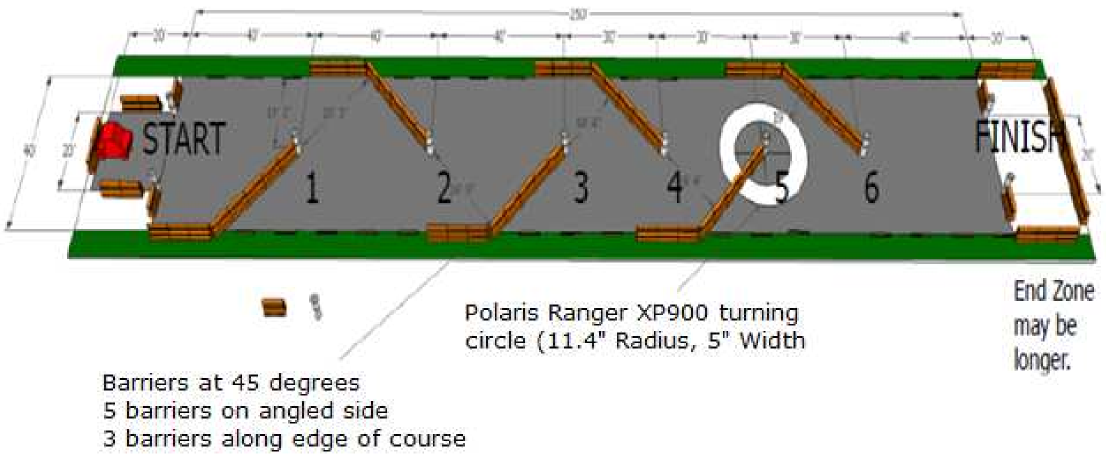

START
1
2
3
4
5
6
FINISH
Polaris Ranger XP900 turning circle (11.4" Radius, 5" Width
End Zone may be longer.
Barriers at 45 degrees
5 barriers on angled side
3 barriers along edge of course

Figure 2.9: DARPA Robotics Challenge 2013 track. Source (DARPA DRC Team, 2013)

2.8 Use of Reinforcement Learning in Obstacle Avoidance

Reinforcement learning has been used to train an agent that controls a robot in the obstacle avoidance task. The following are some examples:

i. Sehad & Touzet (1994) used Self organizing Maps (SOMs) together with Q-Learning to implement obstacle avoidance behaviour. The main focus of this work was generalization and showed that SOMs can be successfully applied as a generalization technique in reinforcement learning. In this study however, the obstacle avoidance behaviour was implemented in isolation. The robot just moved about its environment and avoided obstacles whenever they occurred with no particular goal in mind.

ii. Michels et al. (2005) used the PEGASUS policy search algorithm to generate a policy for steering, and speed control for a robot based on monocular vision. Monocular vision refers to the fact that a single camera was used to take an image. Visual cues e.g. texture were used to estimate the distance of the obstacles. The main focus of this work was to show that monocular vision can be used to provide sensory information for obstacle avoidance. In this study, the autonomously driven remote controlled vehicle was expected to maintain forward motion at all time except when it was necessary to avoid obstacles, in which case, steering commands were issued. The control agent did not have a specific goal to drive to.

iii. Touzet(1997) compared the performance of different implementations of the Q-learning algorithm in the obstacle avoidance task. In the task that is used for testing purposes, the robot by default executes the move forward behaviour unless it encounters an obstacle in which case it has to decide if to turn left or right or continue moving forward. In this implementation, the robot was not expected to move towards any particular goal, just keep moving forward while avoiding obstacles. The different implementations of Q-learning that were attempted included:

   a) Lookup table implementation
   b) Statistical clustering
   c) Use of weighted hamming distance for state space generalization
   d) DYNA Q-Learning
   e) Multilayer Perceptron
   f) Self Organizing maps

   The experiments showed that neural network approaches like multilayer perceptrons and self organizing maps led to better performance than the other generalizations techniques that were implemented.

iv. Mario et al. (2013) used Q-learning in a single and multi-robot obstacle avoidance task where the performance of Particle Swam Optimization and Q-learning are compared. The robot was expected to move forward while avoiding obstacles. The robot did not have any particular goal to move towards. In Q-learning, this led to the robot hovering in an area of the Arena that did not have obstacles.

v. Smart & Kaelbling (2002) solved the obstacle avoidance task by use of example trajectories. This is attained by having the learning taking place in two phases. In the first phase, the robot is either under the control of an example solution (hard coded) or a human. The reinforcement learning control agent observes the states, actions and rewards and uses them to learn. In the second phase, the reinforcement learning agent takes control of the robot. This bootstrapping of the learning process using examples of solutions to the task significantly increase learning rate. In this experiment, the direction and distance of the goal are given as inputs. The robot thus does not have the perceptual capability to perceive the goal. The distance of the obstacles is given by a laser range finder. Only one obstacle is placed in the environment.

2.9 Task Variation in Obstacle Avoidance

The obstacle avoidance task can vary in a multitude of ways. As an example, task variation can occur because the terrain over which the robot is travelling has changed (Leffler, 2009). An agent may learn to steer a robot on a sandy terrain, then it encounters a grass terrain. Different terrains have different levels of friction between the wheels and the ground. Very low friction can lead to the wheels not gripping the ground hard enough leading to the robot slipping. Another possible change is in the inclination of the terrain (Lam & Dietz, 2004).

The task of obstacle avoidance can also change by having different obstacles in the environment (Knudson & Tumer, 2012). The obstacle arrangements can change by having the obstacles being placed in different locations. The obstacle density can also change so that we have more or less obstacles in the environment. An agent should find it harder to learn a policy to control a robot in an environment with a higher density of obstacles.

The task can also change by having a different goal (Knudson & Tumer, 2012). This can also happen in navigation tasks (Fernandez & Veloso, 2006). If a robot is placed in a maze and it has to navigate to a target location, then, different tasks can be created by varying the location of the target. Some target locations may be harder to reach than others for instance if the density of obstacles around one target is higher than an alternative target. When the goal changes, the policy and value predictions will also need to change (Sutton, 1991).

The dimensions and mass of the robot can also change leading to a change in the robots dynamics (Hogg et al., 2001). Thus the robot with the changed dimensions can be seen as a different task and the policy learnt before the dimensions changed will not perform well after the change in dimensions. In Lam & Dietz (2004), one of the changes investigated in an obstacle avoidance task for human subjects was adding of weights to the left leg of the subjects. This shows that change in mass of the robot can be seen as a change of the robot obstacle avoidance task.

The number of distinct actions available to an agent is another factor that can vary in reinforcement learning tasks. Mobile robotics is an example of a task in which the total number of available actions is large (Touzet, 1997). The Khepera robot allows over 400 actions from which only one action is to be selected. This large number of actions occurs because the speed of each of the two control wheels can be set independently enabling turning manoeuvres. The robot is also able to move in reverse which introduces another set of actions. In Q-learning experiments of Touzet (1997), the total number of possible speeds per motor using the Khepera robot is reduced to five out of a possible total of ten, thus reducing the actions space. Since there are two motors, the total number of possible actions is set to 5×5=25. If the number of actions had not been reduced, the total number of available actions would be 10×10=100 actions. Mario et al.(2013) reduce the complexity of an obstacle avoidance task by having only 3 possible speeds per wheel using the Khepera robot in a simulator. The task can also be varied by having the robot move backwards while it had been trained to move in the forward direction.

The obstacle avoidance task can also be varied by having different types and numbers of sensors. Erni & Dietz (2001) showed that cross modal transfer is possible in human subjects between some modalities. For instance transfer occurs from somatosensory modalities to acoustic modalities but little cross modal transfer was noticed from visual modality to other modalities. In Robots, change of the sensors can lead to a change of the state space and also state space variables. The number of state space variables can for instance change when the resolution of the sensors either increases or decreases. The state space can change because different sensor modalities have different range. For instance a laser range finder will have a greater range than infrared sensors.

The reward function can also be varied in the obstacle avoidance task. Mario et al. (2013) note that incorporation of appropriate intermediate rewards can significantly speedup the learning process. These intermediate rewards are however hard to design and are usually not used in most tasks. Incorrect intermediate rewards may bias the robot so that it acquires a behaviour that is different from the desired.

The obstacle avoidance task can be varied by change of the distance from the goal. Smart & Kaelbling (2002) show that for a mobile robot, when the distance from the obstacle is 1 meter, the robot is able to arrive at the goal 46% of the times after one week of simulated training time. At 2 meters, the robot is able to reach the goal 25% of the times after one week of simulated training time. At 3 meters, it reaches the goal 18% of the time after one week of simulated training time. These results imply that the further the distance to the goal, the harder the task faced by the robot.

2.10 Literature Review Summary

Experience replay and knowledge transfer are two methods that have been used to improve learning performance in reinforcement learning. Both techniques have been found to improve the learning rate when applied.

There is very limited testing of the extent of improvement when ER is applied to the obstacle avoidance task. For instance in Lin (1991) the performance of the learning algorithm was not compared to the performance when experience replay was not used. In addition, the only obstacle in the environment was the wall which confined the robot to an enclosed environment. Thus the robot learnt wall following behaviour rather than true obstacle avoidance. In Smart & Kaelbling (2002) only one obstacle was placed in the environment. It is therefore necessary to investigate the extent of improvement when more obstacles are placed in the environment and the robot is not confined to a limited space. This means that the robot will have a greater range of experiences.

There is limited testing of the performance of ER algorithms as the learning task becomes more complex. For instance in Kalyanakrishnan & Stone (2007), ER was tested in the Keepaway task but only in the 3vs2 Keepaway. Though there was improvement in performance, it would have been interesting to investigate if the same trend would be maintained in the 4vs3 Keepaway.

The use of the knowledge transfer techniques in the obstacle avoidance task is very limited. Lin (1991) and Smart & Kaelbling (2002) used a form of knowledge transfer known as imitation (or teaching). In intra-domain transfer, only the reward function changes. It is easier than inter-domain transfer in which the state space and or the action space may change. It is therefore important to investigate the effect of inter-domain transfer techniques on the learning performance in the obstacle avoidance task.

Policy transfer and policy reuse are two knowledge transfer techniques that can be applied for inter-domain knowledge transfer. This is because they enable mapping of actions and or states between different but related tasks. The performances of these two techniques have been compared using the Keepaway task (Taylor & Stone, July 2005; Fernández et al., 2010). In the comparison, CMACs were used to represent the learned policy. Fernández et al.(2010) reported that similar levels of performance were obtained. There is need to compare the performance of the two techniques in a similar task when neural networks are used to represent the learned policy.

Lin (1991) found that in some tasks, for instance wall following, performance using experience replay alone was similar to performance using experience replay plus teaching. This means that in such a task, experience replay can be used alone without going to the extra trouble of adding knowledge transfer features to the learning agent. It would be useful to find out if this observation also applies to the obstacle avoidance task.

2.11 Learning Framework Design

In reinforcement learning research, learning frameworks are created to represent the conceptualised learning process. One way in which the proposed framework can be evaluated is by comparing it to the converse framework. For instance in Smart & Kaelbling (2002) a two phase learning framework is proposed. The framework uses Q-Learning with experiences being provided by a by a human expert in the first phase, and normal Q-learning in the second phase. This framework is compared to a framework that does not rely on the human expert.

An alternative approach is to create several frameworks and then compare the performance of all these frameworks. Lin (1991) and Lin(1992) use this approach. In Lin (1992) for instance, frameworks that combine either or both experience replay and teaching are created. These combinations result in several frameworks that are then compared. Two learning algorithms (Q-Learning and Adaptive Heuristic Critic) are then used to instantiate the frameworks. Each of these instantiation results in a different learning framework.

Another way to create additional frameworks from existing frameworks is to use different a policy function representation. For instance, Adam et al. (2012) uses experience replay but the learning policy is represented using radial basis functions. The framework is then instantiated with both the SARSA and the Q-Learning algorithms resulting in the ER-SARSA and the ER-Q-Learning frameworks. The performances of the resulting frameworks in different tasks are compared. Their performances are also compared to the normal SARSA and Q-Learning algorithms.

Following from the review of literature, we developed a learning framework applies both experience replay and knowledge transfer to reinforcement learning for obstacle avoidance. The framework considers the case where inter-domain transfer is required, in which case, the learning task is made more complex by altering either the state space, and| or the action space. The following learning techniques were found to be applicable to speeding up learning in the obstacle avoidance task:

2.11.1 *Experience Replay*

Several studies have shown that experience replay speeds up reinforcement learning. These include: Lin (1991), Lin (1992), Kalyanakrishnan & Stone (2007), and Adam et al. (2012). In addition, Lin (1991), Lin (1992), and Smart & Kaelbling (2002) combined experience replay with a form of with knowledge reuse known as teaching. Experience replay is therefore expected to improve performance when combined with inter-domain knowledge transfer techniques and applied to the obstacle avoidance task.

2.11.2 *Policy Transfer*

Policy transfer can be used for inter-domain knowledge transfer because it tries to map states and | or actions in the target task to similar state and | or actions in the source task. It has been shown to speed up learning in the Keepaway task (Taylor et al., 2007; Taylor & Stone, 2005). It is therefore expected to speed up learning in the obstacle avoidance task.

2.11.3 *Inter-task mapping*

Inter-task mapping refers to the criteria used to decide how the target task policy should be initialised with the source task policy during policy transfer. It relates the target task state variables and actions to the source task variables and actions. It also specifies how the target task variable parameters are initialised with the source task variable parameters. Among the methods available for inter-task mapping are: hand coded mapping, incomplete mapping and learning of the inter-task mapping (Taylor et al., 2007). Hand coded inter-task mapping results in the better performance compared to the other two methods.

2.11.4 *Policy Reuse*

This is a technique used to improve performance in reinforcement learning in a new task by using the final policy of another task that has already been learnt (source task) to select actions in a new task (target task). The experience gained using the source task policy is used to update the target task policy. The target task policy is also used to select actions. It is therefore necessary to decide which of the two policies will be in use at any one time. Policy reuse is applicable in inter-domain knowledge transfer because it supports inter-task mapping (Fernández et al., 2010).

2.11.5 *Combining Policy Transfer with Policy Reuse*

Policy transfer involves creation of a new policy for the target task in which the policy parameters values are initialised with policy parameter values from the source task. This new policy has no guaranteed performance levels because new actions have been added and there is no way to guarantee preference for the previous actions. The new parameter settings for the created new policy may bias the policy so that it prefers either the new actions or the old actions or a mixture of the two. The only reason that policy transfer is expected to work is that policy parameter values are assumed to be close to the "correct" parameter values of a good policy hence the search for the "correct" values will not take long since the initial values are close to the "correct" values.

Policy reuse on the other hand creates a completely new policy and initialises its parameters with random values. Knowledge transfer is attained by continuing to use the unmodified policy from the source task to select actions. However the experience obtained using the source task policy is used for learning in the new target task policy. Therefore, if only the actions have changed between the two tasks, the source task policy will continue to perform at the same level as it did in the source task. The initial performance of the target policy in policy reuse is however expected to be poor because its parameters are initialized with random values.

Given the above observations, we can use a learning scheme that utilises both policy transfer and policy reuse. In this scheme, the target policy is initialised with parameter values from the source task. The source task policy is however retained. In the target task, for a given episode, either the target task policy or the source task policy is selected for use with the experience attained being used for improving the target task policy. The two policies can continue to be used in parallel for as long as the source policy outperforms the target policy or until the target policy's performance is as good as the source policy's performance.

2.12 Learning Framework

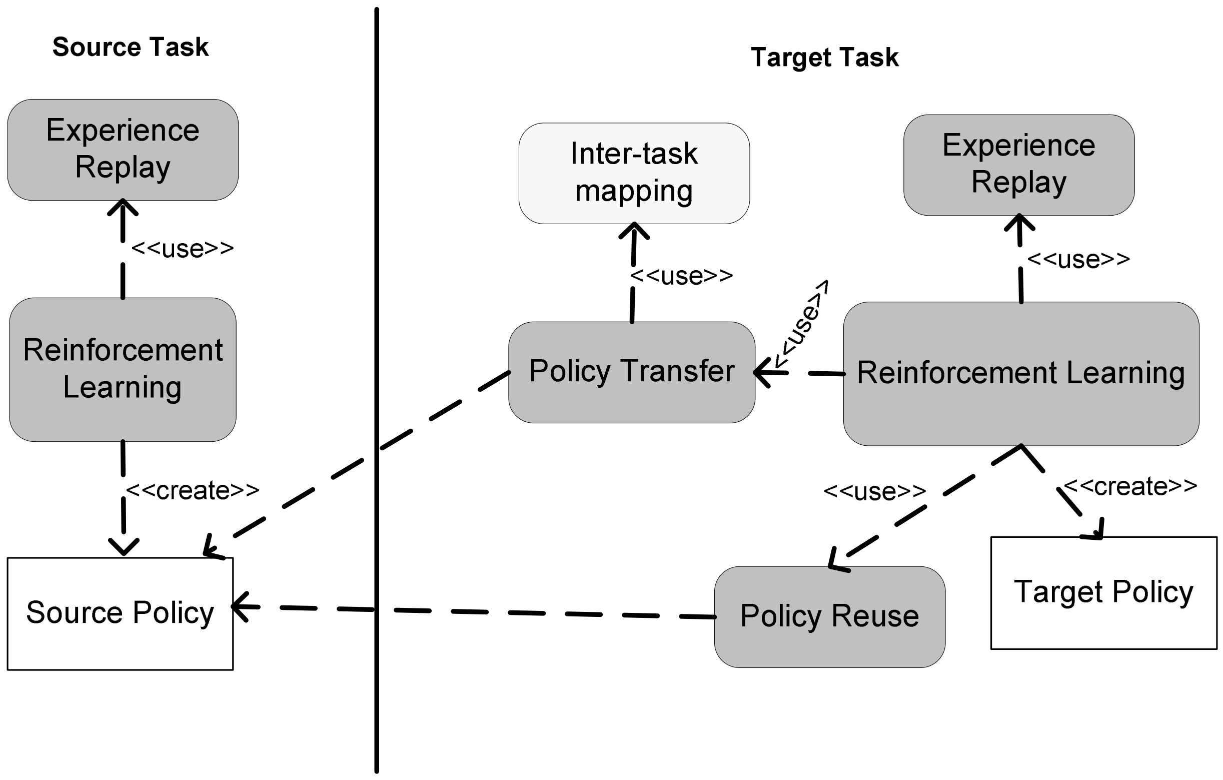


Figure 2.10: Learning Framework

In the proposed framework, the relationships between the techniques are depicted using UML notation for dependencies.

From the proposed learning framework, we can note the following:

i. Experience replay, policy reuse and policy transfer can be used to speedup reinforcement learning.

ii. During learning in the source task, experience replay will be used to speedup learning.

iii. During learning in the target task, we propose that experience replay, policy reuse and policy transfer should be used to speed up learning.

iv. We compare the efficacy of this framework by comparing it to the situation when any of the speedup techniques is not used in learning the target task. We use the term ON to represent a situation where a speedup technique is used, and OFF to represent a situation where a particular technique is not used.

v. The inter-task mapping is instantiated with hand-coded mapping

vi. The SARSA learning algorithm was used learn the value function. This is because it is less likely to diverge in complex tasks (Adam et al., 2012).

vii. Neural networks, which have been found to attain good performances in previous studies were used to represent the value function.

2.13 Learning Algorithms

Combining the above learning concepts in various forms results in the learning algorithms described in Table 2.1:

Table 2.1: Alternative task learning algorithms

Description	Algorithm	Details
Baseline	SARSA(λ)	This is the basic TD learning algorithm that is used in this study. It is used to provide the baseline performance. It uses Equation 2.11 to update the state-action value function. It can be used to learn both the source and the target task. It is the algorithm in use when experience replay, policy reuse and policy transfer are turned OFF
Proposed	SARSA(λ)-R-PT-PR	In this algorithm, policy reuse, policy transfer and experience replay are turned ON during learning the target task
Alternative 1	SARSA(λ)-PT-PR	In this algorithm both policy transfer and policy reuse are turned ON during learning in the target task
Alternative 2	SARSA(λ)-PT	In this algorithm, only policy transfer is turned ON during learning in the target task
Alternative 3	SARSA(λ)-PR	In this algorithm, only policy reuse is turned ON during learning in the target task
Alternative 4	SARSA(λ)-R-PT	In this algorithm, both experience replay and policy reuse are turned on during learning in the target task
Alternative 5	SARSA(λ)-R-PR	In this algorithm both policy transfer and experience replay are turned ON during learning the target task
Alternative 6	SARSA(λ)-R	In This algorithm, only experience replay is turned on in Learning the Target task. The algorithm can be used in learning both the source and the target task

2.14 Research Questions

Comparing the performance of the above algorithms will help us answer the following research questions:

i. Does the use of experience replay speedup learning in the obstacle avoidance task?
ii. Does the use of policy transfer speed up learning in the obstacle avoidance task?
iii. Does the use of policy reuse speed up learning in the obstacle avoidance task?
iv. Does combining policy transfer with policy reuse lead to greater speedup than when each of the techniques is used alone?
v. Does combining knowledge reuse with experience replay lead to greater speedup than when each of the techniques is used alone?

3. Methodology

In this chapter, the tasks to be undertaken in order to meet the objectives of the study are documented. Section 3.1 titled "*Introduction*" summarises the approach taken to attain each of the specific objectives listed in section 1.7. Section 3.2 titled "*Research Strategy*" discusses the justification for using the experimental strategy to conduct this study. Section 3.3 titled "*Experimentation Task Definition*" describes the obstacle avoidance task that the learning algorithms were expected to learn to perform. Section 3.4 is titled "*Experiment design*" and it details the main settings in the experiments that were performed. Section 3.4 also contains a description of the performance measures and the steps taken to ensure validity of the results. Section 3.5 is titled "*Analysis*" and contains a description of the statistical analysis techniques used in describing and comparing the results obtained in different experiments.

3.1 Introduction

The main objective of this study was to explore the use of knowledge transfer and experience replay to speed up learning in the obstacle avoidance task. Table 3.1 gives a summary of the approach taken in order to attain this objective; the specific objective that had to be achieved and the corresponding method used are listed.

Table 3.1: Methodology Summary

No	Objective	Method
1	Review the state of the art in knowledge transfer, experience replay and application of reinforcement learning in the obstacle avoidance task	Literature review
2	Propose framework for the application of knowledge transfer and experience replay in reinforcement learning in the obstacle avoidance task	Learning Framework Design
3	Develop a control agent based on the proposed framework and interface the control agent with a robot simulation environment	Evolutionary prototyping
4	Evaluate the performance of the control agent by performing simulations to determine the baseline performance, and the performance when knowledge transfer and experience replay are applied to the obstacle avoidance task.	Perform experiments using simulations in the obstacle avoidance domain. Compare the results of the performance of the proposed algorithms to the performance of the baseline algorithm

5	Update and validate the proposed learning framework	Analyse and discuss the results

The items numbered 1 and 2 in Table 3.1 are addressed in chapter 2. A learning framework that integrates experience replay and knowledge transfer (policy reuse and policy transfer) was designed and is depicted in section 2.12 of the literature review chapter. The algorithms corresponding to this framework, and alternative algorithms resulting from modifications of this framework as explained in section 2.12 are listed in Table 2.1. Objective 3 is addressed in chapter 4, objective 4 is addressed in this chapter and in chapter 5 and finally objective 5 is addressed in chapter 6. Attaining objective 4 will help us answer the research questions listed in section 2.14. This will consequently enable us to update the proposed learning framework.

3.2 Research Strategy

In order to answer the research questions, we had to evaluate the algorithm derived from the proposed framework, and compare its performance to the alternative algorithms resulting from modifications to the framework. An experimental strategy was adopted for this purpose. An experiment is an orderly procedure carried out with the goal of verifying or refuting a hypothesis. In experiments, specific factors can be isolated and their effects observed (Denscombe, 1999). The experimental strategy is appropriate for this study because several techniques are being compared. It is therefore possible to hold all other factors constant but for the application of the technique or techniques of interest. The experiments to evaluate the algorithms listed in Table 2.1 were performed through simulations in the obstacle avoidance domain.

A simulation is the imitation of the operations of a real-world process or system over time by hand or usually using a computer program. Simulation is also defined as the representation of a real system by another system that depicts the important characteristics of the real life system and allows experiments to be carried out (Hira, 2001). It leads to the generation of an artificial history of a system. The information in the artificial history can be used to draw inferences concerning the operational characteristics of the real system. In order to carryout the simulation, a model of the real-world system has to be built. A system, in general, is a collection of entities which are logically related and which are of interest to a particular application (Perros, 2009). When building a simulation model of a real-life system under investigation, one does not simulate the whole system. Rather, one simulates those sub-systems which are related to the problem at hand. This involves modelling parts of the system at various levels of detail (Perros, 2009).

In this study, simulations were used to conduct experiments for the following reasons:

1) The destructive nature of the experiments

Real robots are expensive to acquire and configure. Experiments will involve collision of the robots with obstacles. This would most likely lead to destruction of the real robot. It can also lead to accidents that damage other objects apart from the robot itself.

2) Experiments' running time

Reinforcement learning is an exploratory learning process. It can take thousands of episodes before the control agent learns the correct behaviour. Energy constraints would make it hard for a real robot to execute this many episodes.

3) Autonomous running of experiments

Real life robots would require full attention of the experimenter. For instance after an episode ends, the experimenter has to reset the environment for the next episode. In a simulator, this can be automated and the experiments can run unattended. This means the experiments can even run during the night resulting in more data being generated.

4) Replication of experiments

Multiple instance of the simulation can be run in different computers to test different aspects of the experiment resulting in generation of more data.

5) Common practices

It is a common practice to use simulation in robotics and reinforcement learning experiments.

3.3 Experimentation Task Definition

The goal of experimentation was to evaluate the performance of the algorithms listed in Table 2.1 in speeding up learning in the obstacle avoidance task when compared to the baseline learning algorithm. Doing this would help us in answering the research questions posed in section 2.14. The obstacle avoidance task can take many different forms: In Lin(1991) for instance, the only obstacle that the robot could encounter was the wall that also formed the boundary of the robot environment. The behaviour the robot was expected to learn was to follow the wall in either getting to a door or docking to a charger; In Smart & Kaelbling(2002), the robot was expected to move to a goal region while avoiding the one obstacle placed in the environment. The distance of the robot to the obstacle, the distance of the robot to the goal and the direction of both the obstacle and the goal relative to the heading of the robot were given as inputs; In Touzet (1997), though several obstacles were placed in the environment, the robot was just expected to keep moving forward while avoiding obstacles with no particular goal to move to. This was also the setup in Michels et al. (2005)

In this study a goal and four obstacles were placed in the enviroment. The obstacles were placed in such a way as to possbily block the progress of the robot towards the goal. The ability of the robot to move to the goal while avoiding obstacles was tested. This is the behaviour that was expected to be learnt by the agent controlling the robot using the learning algorithms listed in Table 2.1. The performance was evaluated based on how fast an algorithm enabled this behaviour to be learnt. The agent controlling the robot was known as the *control agent* and it represented the learnt behaviour in a control policy implemented using a multilayer neural network.

In order to test the performance of the knowledge transfer algorithms, the obstacle avoidance task was varied by changing the number of actions. Touzet (1997) and Mario et al. (2013) reduce the complexity of the obstacle avoidance task by reducing the number of steering actions available in the robot. The complexity of the task can thus be increased by increasing the number of actions.

The input to the control policy included: the distances to the obstacles, the directions of the obstacles relative to the heading of the robot, the direction of the goal relative to the heading of the robot, and the previous action selected. The distance to the goal was not be used as an input to the control policy. This was to enable the agent to move towards what it perceives as the goal rather than just take actions that reduce the distance to the goal. The agent was expected to figure out, based on rewards obtained from the environment, which of its inputs represents the goal and how to get to it. The task of the control policy was to select an action based on these inputs which represented the state of the environment.

The robot needed sensors to detect when a collision with an obstacle occurred. A means for detecting when the robot was close enough to the goal and thus could be deemed to have reached the goal was also needed.

Chapter 4 "simulation system development methodology" details the steps taken to develop the simulation system required to perform the evaluation experiments.

3.4 Experiment Design

Experiments were conducted for all learning algorithms being evaluated to generate data that was used to estimate their performance. The performances of the different techniques were then compared in three learning tasks which were:

i. A robot obstacle avoidance task with two steering actions
ii. A robot obstacle avoidance task with three actions
iii. A robot obstacle avoidance task with six actions

The experiments were performed in the Microsoft Robotics Developer Studio 4 (RDS) robotics simulation environment.

3.4.1 *General Task Description*

The general task was an obstacle avoidance task where a robot (a simulated version of Adept MobileRobots Pioneer 3DX) was randomly placed in one of 10 predefined starting positions. The control agent was responsible for guiding the robot to a goal position while avoiding obstacles by issuing steering commands. These steering commands constitute the set of actions that the control agent could direct the robot to perform. All the predefined starting positions were at a distance of 3 meters from the goal. At the start of an episode, the orientation of the robot towards the goal was random. This means that the robot was placed in any of the 10 predefined starting positions but the direction it was facing was random. Smart & Kaelbling (2002) also used ten pre-specified starting poses in training a mobile robot for obstacle avoidance using example trajectories. The robot was deemed to have reached the goal position when it was at a distance of one meter from the goal.

The Pioneer 3DX robot is equipped with several sensors that enable it to capture the state of the environment. These are:

1) Laser Range Finder (LRF)

The onboard simulated version of a SICK Laser Range Finder was used to determine the positions and the distances of obstacles. This LRF performs a 180^0 sweep of the scene at intervals of 0.5^0. This means that in one sweep, it returns 361 values. We processed these values in order to reduce the resolution to just eleven values. This was done to reduce the number of state variables. It was also informed by the fact that the same obstacles may be represented in several of the readings due to the high resolution. The scanning frequency of the SICK LRF is 75Hz which was more than adequate for our needs since we needed to get the state of the environment after every half a second. The range for the simulated LRF is 8 meters.

2) Contact Sensors

The onboard contact sensors were used to detect collisions with obstacles. The Pioneer 3DX has a ring of 13 simulated contact sensors (Moreno, 2007). For our experiments, we were only interested in finding out if a collision had occurred. The attached contact sensors were used to provide this information.

3) Webcam

The onboard simulated webcam captures a video of the scene at the front of the robot in colour. A frame grab enables the acquisition of the most recently captured frame (image). The dimensions of the returned image are 320×240 where 320 is the width and 240 is the height. In our experiments, we used this image to determine the direction of the goal relative to the heading of the robot. The image was divided into eleven vertical strips with strip six as the middle strip. If the goal object was found in strip six, then it was directly ahead of the robot. If the goal object was in strips one to five, then it was on the left of the robot. Otherwise if the goal object was on strip 7 to 11 then it was on the right of the robot. Therefore there were eleven possible goal locations. A blob detection algorithm (section 4.4.10.2) was used to identify the goal object. This algorithm is based on the colour of the image. In our simulation, the goal was the only pink object in the scene.

4) Simulated GPS Sensor

The simulated GPS sensor was used to detect the current location of both the robot and the goal. This means that there was one GPS sensor for the robot and another for the goal. The

GPS coordinates for the two entities were used to calculate the distance between the robot and the goal. This was useful for determining if the robot was within the goal region.

The Pioneer 3Dx is a differential drive robot which means that it has two wheels (one on either side of its body) that can be set to move at different speeds. If the two wheels are set to move at the same speed, then the robot moves in a straight line. If the left wheel has a greater speed than the right wheel, then the robot turns to the right. If the right wheel has the greater speed, the robot turns to the left. If the speed of one of the wheels is set to zero, while the other has a speed greater than zero, then the robot turns in place. Each wheel of the simulated Pioneer 3DX can be set to speed values ranging from -1 to 1. The negative values cause the wheel to rotate in reverse. If both wheels have negative values, then the robot will move in reverse. Table 3.2 gives the actions (wheel settings) used in the experiments performed in this study. Later on, the actions will frequently be referred to using the indexes assigned in Table 3.2.

Table 3.2: Actions used in the experiments

Action Index	Left and Right Wheel Power Values	Description
0	{0.1, 0.1}	Move straight
1	{0, 0.1}	Turn left
2	{0.1, 0}	Turn Right
3	{0.2, 0.2}	Move straight
4	{0.2, 0.1}	Turn Right
5	{0.1, 0.2}	Turn Left

A summary of the general task specification is given in Table 3.3

Table 3.3: Source task specification.

Description	Value
Distance from goal	3 meters
Goal region	1 meter from goal
Number of obstacles	4
State variables	24
Number of Actions	Variable
Maximum number of actions per episode	100
Goal reward (GR)	1
Obstacle collision reward (OCR)	-1

Time barred reward (TBR)	-1
Non terminal actions reward (NTR)	0

Eleven of the twenty two state variables represented the obstacle positions in relation to the heading of the robot. The first five variables represented positions of the obstacles that were to the left of the heading of the robot, the sixth variable was the position of obstacles directly ahead of the robot, while next five variables (7 to 11) represented positions of obstacles to the right of the robot with position 11 being the extreme right. The value of a given variable represented the distance of the detected obstacle that was closest to the robot in that direction. Since the maximum distance of the LRF is 8m, the values of these variables were divided by 8 hence a variable could have a minimum value of zero (robot is at zero distance from the obstacle) and a maximum distance of 1 (the closest obstacle is 8 meters or more from the robot).

The next 11 variables were used to denote the direction of the goal. The values for these variables were either 0 or 1. A value of 0 meant that the goal was not along the direction represented by that variable and a value of 1 implied that the goal was along the direction represented by the given variable. If all the eleven variables are zero, it implied that the goal was not visible. If the goal was visible, only one of these eleven variables had its value set to 1.

The next two variables represented the current speed of the robot (power settings) on both wheels.

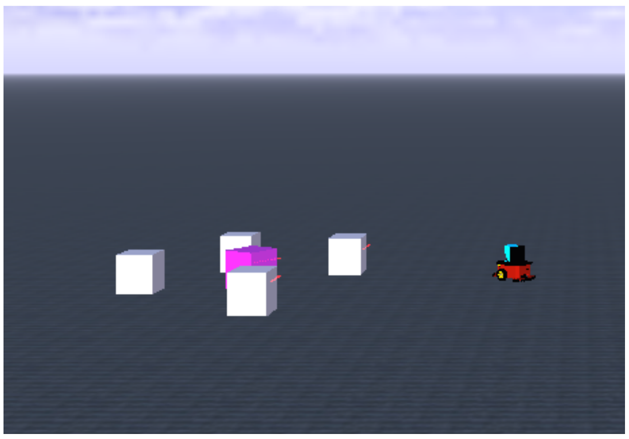


Figure 3.1: One of the starting positions of the robot also showing the obstacle positions.

All but two of the algorithms listed in Table 2.1 use a form of knowledge transfer. In knowledge transfer, both the source and the target task had to be specified. Table 3.4 shows the specification of two source task target task pairs. Other studies in the same area have a similar number of pairs of tasks in evaluating the performance of knowledge transfer techniques. Examples include Taylor & Stone(2005) and Fernández et al. (2010). In (Taylor et al.(2007) however, only one pair of tasks was used.

Table 3.4: Number of actions in the corresponding source and target tasks

	Number of Source task actions	Number of Target task actions
Setting 1	2	3
Setting 2	3	6

The two to three actions transfer setting was selected because at a minimum, the robot required two actions to able to undertake the task of moving from the start position to the goal location. This happens to be the case because by turning in place, the robot can make a 360^0 turn to face any direction. The three actions task represents the turning possibilities that human users are used to which are: turn left, turn right, and move straight. The six actions task is similar to the six actions task but the robot has access to faster turning and translation speeds. Every action in the six action task has a similar action in the three actions task that it can be mapped to. For instance the faster turn left action can be mapped to the slower turn left action in the three actions task.

During transfer from the two actions task to the three actions task, in the source task, the control agent had two actions with which it could control the robot. These were actions 0 and 1 in Table 3.2. These two actions were sufficient for the control agent to move the robot, while avoiding obstacles, from the start position to the goal region. The target task had got 3 actions. These were actions 0, 1, and 2 in Table 3.2.

During transfer from the three actions task to the six actions task, in the source task, the control agent had three actions with which it could control the robot. These are actions 0, 1 and 2 in Table 3.2. The target task had 6 actions: actions 0, 1, 2, 3, 4, and 5 in Table 3.2.

It was necessary to set the *Maximum number of actions per episode* because of the possibility of the robot failing to reach the goal or taking too long to reach the goal. The maximum number of actions per episode was used to terminate an episode whether the robot had reached the goal or not. This prevented situations where the robot got stuck in hopeless situations.

The goal reward (GR) is the reward that was given when the episode ended due to the fact that the robot was within the goal region. The obstacle collision reward (OCR) is the reward was given when the robot collided with an obstacle. The time barred reward (TBR) is the reward that was given when the agent exceeded the maximum number of actions per episode before reaching the goal or colliding with an obstacle. When an action taken did not lead to a terminal state, the Non terminal actions reward (NTR) was given. This reward was set to zero meaning that rewards were given only for terminal states.

3.4.2 *Random Number Generators*

Simulation involves generation of an artificial history of a system to be used to draw inferences about the real system (Perros, 2009). When simulation is applied, random number generators are used to model the different ways in which the system may behave. Table 3.5 gives a description the random number generators that we used in this study.

The artificial random number generators available in computers use what is known as a seed. The seed value used in creating a random number generator uniquely determines the sequence of random numbers that will be generated. This means that if two random number generators are initialised with the same seed, they will generate the same sequence of random numbers. Random number generators can be used to create task scenarios that are different and have different levels of difficulty. For instance given two *randomAngle* generators (see Table 3.5), if one generator produces values that orient the robot towards the goal most of the times, while the other generator produces values that orient the robot away from the goal, then the former generator will result in an easier task than the later.

Table 3.5: Random number generators

Index	Name	Description
1	*randomWeights*	A generator used to initialise the weights of the neural network. It generates a value between zero and one
2	*randomSign*	A generator used to set the weights of the neural network to values between -1 and 1 by setting the sign of the random value

		generated by *randomWeights*
3	*randomAction*	Used in ε-greedy strategies, to generate a random number such that if this number is less than ε, we select a random action rather than the currently best action
4	*randomActionSelect*	If we are to select a random action, then we need to generate another random number to enable random selection of any of the available actions
5	*randomStartingPosition*	Given the starting positions available, we need to select one randomly. This random number generator produces a random value used for this purpose
6	*randomAngle*	Used to generate a random orientation for the robot
7	*randomEpisode*	Used in experience replay to select a random episode to be replayed (used for training)
8	*randomReplace*	Used to select a random episode to be replaced by a new episode in experience replay when the maximum number of stored episodes has been reached
9	*randomPolicyReuse*	Used to randomly chose between the source and target policy in policy reuse

3.4.3 *Performance Measures*

3.4.3.1 What is measured

A learning algorithm can be said to speedup learning performance if it leads to attainment of higher performance levels after a given period of training compared to an alternative algorithm. Several performance measures have been used in previous reinforcement learning studies. As an example, Lin (1992) measured the number of positive rewards obtained. Smart & Kaelbling (2002) measured the number of steps taken to reach the goal. In addition, they measured the number of times the agent reaches the goal. Fernandez & Veloso (2006) measured the number of episodes that end with the robot at the goal region. This is similar to measuring the number of positive rewards obtained.

For this study we used the number of episodes that ended at the goal region as the measure of performance. This was the same as measuring the number of positive rewards obtained.

We therefore kept track of all the positive rewards obtained by the agent. This just entailed keeping track of the number of episodes that ended at the goal region. We in addition kept track of the number of episodes that terminated because of a collision with an obstacle, and the number of episodes that terminated because the agent was taking too long before it reached the goal or collided with an obstacle.

3.4.3.2 Performance Calculation and Representation

To determine the performance we calculated a value referred to as the *average reward (or average performance)* which was calculated at some point in time and showed the average performance over some interval of time corresponding to a number of continuous episodes. This interval need not start from the first episode. We therefore needed to define a variable *INTERVAL_SIZE* that determined the number of continuous episodes whose average performance we wanted to calculate. When creating a line graph of performance, we got the average performance at several points. There was the possibility of the intervals represented by these points overlapping. For instance if *INTERVAL_SIZE* is 200, we can calculate the average performance at point 500, representing performance from episode 301 to 500. For point 600, the episodes included in this interval are episode 401 to 600. Therefore, the graph will represent a *simple moving average reward*. In this study, the *INTERVAL_SIZE* was set to 300.

If the average performance is calculated at a point considered the end of the training, this is referred to as the *asymptotic performance*. If the average reward is calculated at a point considered the end of the initial phase of training, then this will be referred to as the *initial performance*. Taylor & Stone (2009) list five metrics that can be used to measure the benefits of knowledge transfer viz. jumpstart (initial performance), asymptotic performance, total reward, transfer ratio and time to threshold. The initial performance and the asymptotic performance were used in this study because they are more representative of the performance at a particular point in time. As an example, an agent may have a high initial performance and a low asymptotic performance. If the total reward measure is used, this trend will not be noticed. The same applies to the transfer ratio measure. The disadvantage of the time to threshold measure is that some techniques may not achieve the specified threshold.

To calculate the average reward, given the interval under consideration, we got the total number of episodes that have terminated in the goal region. We refer to this value as the *interval goal rewards* (IGR). The average reward for the interval is then given by Equation 3.1.

$$AR = \left(IGR \big/ INTERVAL\_SIZE \right) \times 100 \qquad \textit{Equation 3.1}$$

The multiplication by hundred just ensures that the average reward is viewed as a percentage for better visualization. Price & Boutilier (2003) use a similar measure of performance as we have described except that for graphing purposes their intervals do not overlap.

An alternative way to calculate the average reward at a given point is to count the number of episodes that end at the goal region since the trial started, up to the current point. The problem with this method is that it does not accurately represent the current performance at the point. This is because the value obtained is still affected by the performance that prevailed at the beginning of the trial. Therefore algorithms that have an initial poor performance will be disadvantaged in case of a comparison. Example of a studies that have used this measure are Fernandez & Veloso (2006), Taylor & Stone (2005) and Torrey et al. (2005).

3.4.4 *Experimental setup*

The main task that was be accomplished in all experiments was for the control agent to drive a simulated differential drive robot (Pioneer 3DX) that was to be placed at a distance of 3 meters from the goal (Figure 3.1) until the robot reached the goal region. The goal region was defined as the region that was within a radius of a one meter from the goal. The goal object was the only pink object in the simulated environment. There were some other four objects in the environment which constituted obstacles that the robot was to avoid.

Ten starting positions were manually defined such that the robot had a variety of starting positions. Four of the starting positions placed the robot directly in front of the obstacles. Four other starting positions placed the robot in a direct line to the goal with no obstacle in between. The other two starting positions placed the robot in a position where it could just brush an obstacle if it went directly towards the goal without any turning. When the robot was placed in a selected starting position, the direction which it was facing was randomly generated. This means that it could have been facing towards the goal, or away from the goal, or in any other direction.

The task facing the control agent was to select the sequence of actions that would move the robot from its starting position until it reached the goal region. In a given trial, the set of actions available was fixed. A trial was composed of several episodes. An episode started when the robot was placed in a random starting position, and ended when the robot either reached the goal region, collided with an obstacle, or the maximum number of actions that could be executed in an episode was reached. The maximum number of actions that can be executed in an episode was set to 100 where each of the actions was selected from the fixed set of available. When the episode ended, the control agent automatically placed the robot in a random starting position so that the next episode could start. The trial went on indefinitely unless it was manually terminated or if a power failure occurred.

The agent followed a *sense/action-selection/train/act* cycle as prescribed by the SARSA(λ) learning algorithm. The sequence of steps used in this cycle is given below:

i. The agent senses the state of the environment using the sensors available in the robot.
ii. The agent then uses the control policy to select the next action.
iii. The agent updates the control policy.
iv. The action selected in step *ii* is executed (sent to the robot) and the agent goes back to step *i* after 500 milliseconds

500ms is the time needed for an action to take effect. If time is not given for an action to take effect, the computer may be so fast that it may send the next command to the robot before the current action is executed. For instance, the control agent may issue a turn left action but before the robot can execute the command, the control agent may issue a different action because of the computer processing speed. This means that steps *i* to *iii* above can be executed in a negligible amount of time. The 500ms value was arrived at by experimenting with several values ranging from 50ms to 1000ms.

As a trial was proceeding, relevant information was captured in a log file. This information was largely episode based. For each episode we captured: The episode number, the sequence of actions selected in the episode, how the episode terminated. Other information was also captured depending on the technique that was used to create the control policy. Other information that was relevant to the trial was recorded in a hard copy log. This included: the set of actions used in the trial, and the learning techniques used to update the control policy.

An experiment consisted of 14 trials such that each trial could be run on its own computer. Alternatively several trials could be run sequentially in the same computer by starting a new trial after the currently executing trial is terminated. The sequential method would have taken much longer considering that one trial took at least 24 hours. 28 computers were available hence we could run two experiments simultaneously with each experiment taking up 14 computers hence 14 trials. While it would have been ideal to have one experiment running in 28 computers – hence 28 trials, frequent power failures meant that each experiment needed to be repeated almost five times. We therefore decided to run at least two experiments in parallel whenever power was available. While the choice of 14 trials was largely determined by the available resources, other similar studies have been based on approximately the same or less number of trials. For instance (Lin, 1992) run 7 trials for each algorithm that was tested in a study that investigated the performance of experience replay with each trial running for two days. In a study that compared different function generalization methods in an obstacle avoidance task (Touzet, 1997), performance for each method was calculated as an average of the performance in five trials.

In each experiment, each trial was allowed to run for 24 hours. It was found that typically after 24 hours, around 3,900 episodes would have been completed in a trial. Thus trial number 3,900 was taken as the final episode in a trial unless otherwise indicated.

3.4.5 *Validity*

To ensure validity, the following measures were put in place:

1. Use of scenarios

For this study, fourteen scenarios were defined. A scenario was defined by the set of integer seeds used to initialise the nine random number generators. For a given scenario, the set of seeds used to initialise the random number generators was fixed. Once a scenario was defined, it was used in all experiments with the same seeds. Each scenario had seeds that were different from any other scenario.

We indicated earlier that each experiment consisted of 14 trials. In our setup, trial one used the random number seeds in scenario one, trial two used the random number seeds in scenario 2 and so on.

This ensured that if we were comparing performance in say experiment one and two, trial one on both experiments had used the same set of random number seeds, trial two in both experiments had used the same set of seeds and so forth.

This strategy was adopted after it was found that use of different random number seeds resulted in different performances even if the training algorithm remained the same. As an example, if two different random number seeds were used to initialise two different neural networks, one of the initialization could result in a better performance than the other initialisation. It was therefore necessary to ensure that if we were comparing two algorithms, the neural networks were initialized with the same weights.

Table 3.6: Random number seeds for scenario one. The same set of seeds is used in trial one of all experiments

138119004, -876213882, -320144197, -76966129, 1962715483, -1977800493, -1696645572, 225649046, -1816068477

In Table 3.6 the first two values were used to initialise the random number variables used in initialising the weights of the neural network (randomWeights, and randomSign), the third number is used to initialise the randomStartingPosition variable, the fourth the randomAngle variable, the fifth the randomAction random variable, the sixth the randomActionSelect variable, the seventh the randomEpisode variable, the eighth the randomReplace variable, and the ninth the randomPolicyReuse variable (see Table 3.5). The random number seeds were generated using the Microsoft Visual studio Global Unique Identifier class (GUID).

The fact that each experiment consists of 14 trials (scenarios) ensures that the performance is averaged using 14 values representing different samples of the task. This ensures that the average performance is representative of the typical performance in the task.

2. Random starting position and orientation

The random starting position and orientation ensured that the task was not simplistic. The task the control agent was expected to learn was challenging task hence ensuring that the techniques tested can be used for other challenging tasks if they are found to lead to good performance.

3. Use of tasks with varying difficulty

The learning techniques were studied were tested in tasks with varying degrees of difficulty. These are control of a robot with two actions, control of a robot with three actions and control of a robot with six actions. Knowledge reuse was tested by reusing a two actions policy to a three actions task, and a three actions policy in a six actions task. We were therefore able to determine if the performance of a learning technique was consistent as the tasks changed.

3.5 Analysis

Statistical analysis techniques were used to test if the difference in performance between the different techniques that were used for learning was significant.

3.5.1 *Average Performance charts*

An average performance chart is used to show the change in performance with time. It can be used to show a visual representation of the difference in performance of two or more techniques. The average performance chart can show 1) The average initial performance and 2) The average performance after a predetermined number of episodes.

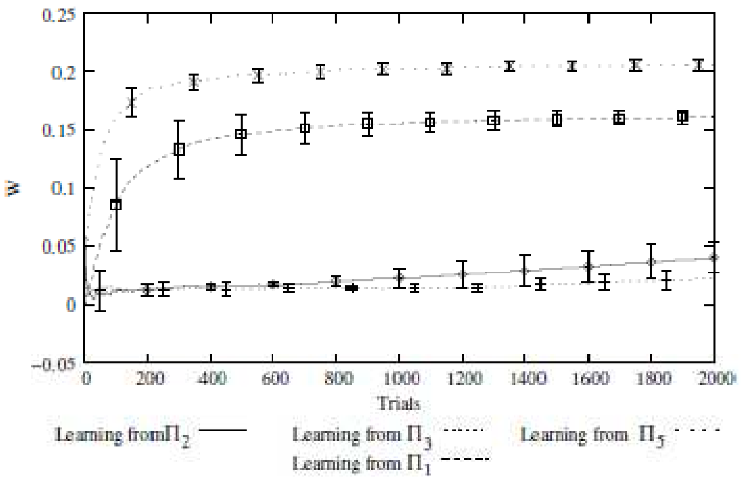


Figure 3.2: An average performance chart. The horizontal axis represents the number of trial (episodes) while the vertical axis represents the gain (performance). Source: (Fernandez & Veloso, 2006)

Several studies represent their results using this method including: Fernandez & Veloso (2006), Taylor & Stone (2005) and Torrey et al.(2005).

A variation of the average performance is to calculate the performance in the last 100 episodes (Price & Boutilier, 2003). This ensures that the performance graphed represents the current performance rather than the average performance from the starting episode to the current. This may suffer from having the performance chart being too jagged. To avoid this, we plotted graphs of moving averages as discussed in 3.4.3.

In this study the average performance was calculated over overlapping intervals of size INTERVAL_SIZE which was set to 300.

3.5.2 *Paired Student's t-test*

The Paired t-test is used when we are comparing two samples such that the elements from the two samples have been paired. Paired tests are tests in which hypothesis are evaluated on identical samples (Mitchel, 1997). That means that given two samples A and B, for every element $a \in A$, there is an element $b \in B$ such that a is paired with b. This can happen when the same sample has been tested twice with the first measurement being placed in sample A and the second measurement being placed in sample B. In comparing two algorithms using paired tests, any differences observed are due to the differences in the algorithms rather than due to the difference in make up between the two tests (Mitchel, 1997). Learning algorithms usually need to be compared when for instance they must learn in real time to perform the task of interest (Dietterich, 1998). We will need to compare the algorithms so we can select one of them to apply in the task.

The paired t-test can be used to test if the two samples are the same in which case they should have a mean difference of zero. The null hypothesis states that the mean of the difference between the two sets of samples is zero.

To use the paired t-test, the sample statistic *t* is calculated using Equation 3.2.

$$t_{calc} = \frac{\overline{X}_D - u_0}{S_D / \sqrt{n}} \qquad \text{Equation 3.2}$$

Where $\overline{X}_D$ is the mean of the differences between the paired elements, S_D is the standard deviation of the differences, n is the number of paired items and u_0 is the expected difference, which is zero if the two samples are expected to be the same. After getting t_{calc} we use the t-table to check if t_{calc} is greater than the *critical value* at the desired significance level. If t_{calc} is greater than the *critical value*, the null hypothesis is rejected. If t_{calc} is less than the *critical value*, then the null hypothesis is not rejected.

The paired t-test was ideal for comparing results from our experiments because we had defined a fixed number of scenarios (14 scenarios). Thus we had scenario 1 to scenario 14 where each scenario was defined by a specific allocation of seed values to the random number generators. Therefore for any two techniques (t_1 and t_2), we measured t_1's performance on scenarios 1 to 14 and also measured t_2's performance on scenario 1 to 14. Thus t_1's performance in scenario 1 was paired with t_2's performance in scenario 1 and so fourth. Each scenario represented a learning problem to be solved by the learning algorithms being considered.

The test for significance was performed at a significance level of 0.05. At this significance level, for a one tailed test, and 13 degree of freedom the critical value is 1.771. For a two-tailed test, the critical value is 2.16.

In this study, the *Data Analysis ToolPak* tool in Microsoft Excel was used to perform all the one tailed paired student's t-tests.

To use the paired student's t test, it is required that the differences between the samples follow a normal distribution. If this is not the case, it is recommended that a non parametric test such as the Wilcoxon Signed Rank test be used to compare the samples (McDonald, 2014). In this study, the Shapiro-Wilk test was used to test for normality. If the differences between the paired samples were found not to follow a normal distribution, the Wilcoxon Signed Rank test was used to compare the samples.

3.5.3 *Shapiro-Wilk test*

There are several tests for normality including Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests (Razali & Wah, 2011). Among these, Razali & Wah (2011) found the Shapiro-Wilk test to be the most powerful in that it is able to discriminate more accurately data that follows the normal distribution from data that does not follow the normal distribution. We therefore decided to use the Shapiro-Wilk test to test for normality of the differences of the paired samples at a significance level of 0.05. An advantage of the Shapiro-Wilk test is that it is quite sensitive even when the number of samples is small for instance less than twenty (Shapiro & Wilk, 1965). In the test, the W statistic is calculated. This value lies between 0 and 1. If the value is close to one, then the data is likely to be normally distributed and if it's close to zero, then it's most likely not normally distributed. After W is calculated, the corresponding p value is obtained from a significance levels table given in (Shapiro & Wilk, 1965). If the p value is less than 0.05, then we conclude that the data is not normally distributed. The cut off value of W that corresponds to a p value of 0.05 will depend on the number of values in sample. For 14 samples, the value of W that corresponds to a p value of 0.05 is 0.874. If the value of W obtained is less than 0.874, there is a more than 95% chance that the data is not from a normal distribution. Shapiro & Wilk (1965) outline the calculation of W as shown by the equations below.

$$W = \frac{b^2}{S^2} \qquad \textit{Equation 3.3}$$

$$S^2 = \sum_{i=1}^{n} (x_i - \overline{x})^2 \qquad \textit{Equation 3.4}$$

$$b = \sum_{i=1}^{k} a_{n-i+1}(x_{n-i+1} - x_i) \quad \text{Equation 3.5}$$

Where x_i i=1…n is the ordered set of samples and $\bar{x}$ is the mean of the samples, and a_{n-i+1} are coefficients provided in Shapiro & Wilk (1965).

$$k = \begin{cases} \frac{n}{2}, if\ \ n\ is\ even \\ \\ \frac{n-1}{2}, if\ \ n\ odd \end{cases} \quad \text{Equation 3.6}$$

When n is odd, the median value is not used in the calculation in Equation 3.5. For this study, we created our own excel tool that is used to perform the Shapiro-Wilk normality test. The test is carried out on the differences of the paired samples.

3.5.4 *The Wilcoxon Signed Rank Test*

This test can be used when the differences between the paired samples does not assume a normal distribution (McDonald, 2014). This test computes a test statistic W that is compared against an expected value at the selected significance level. The process for calculating W can be found in McDonald (2014). If W is less or equal to the expected value, then the null hypothesis is rejected. Alternatively, a p value can be calculated. If the p value is less than the selected significance level, then the null hypothesis is rejected.

When the p value approach is used, a z value is first calculated. Its corresponding p value is then looked up in a normal distribution table. This is possible because W follows a normal distribution when the number of samples used to calculate is greater or equal to 20. In this study, since the maximum number of samples is 14, the W statistic is used.

For a one-tailed test, the null hypothesis states that the median of the differences between the samples is zero (McDonald, 2014). The alternative hypothesis can either state that the median of the differences is greater than zero or that the median of the differences is less than zero depending on the direction of the test.

The following is the sequence of steps used in calculating the W statistic. The values in Table 3.7 will be used to illustrate.

Table 3.7: Sampled data for calculating the W statistic in the Wilcoxon Signed-Rank test calculation

col1	col2
33	40
51	49
76	72
86	80
85	85
65	9
97	60
14	15

1. Add a third column (col3) and populate it with the differences between the values in column 1 and column 2 (col1-col2). If any value in column 3 is equal to zero, it is excluded from any future calculations. n' is the total number of none zero values in column 3. $n' \leq n$ where n is the original number of pairs of values being compared.

2. Add a fourth column (col4) and populate it with the absolute values of the values in col3

3. Add a fifth column (col5) and populate it with the ranks of the values in col4 such that the smallest value in col4 has a rank of one, the second smallest has a rank of two and so on.

   Add a sixth column (col6), in this column, all the entries are equal to the entries in col5 except for entries that were equal in value in col4. For entries on col4 that are equal in value, get the sum of the ranks they are assigned in col5. Divide this sum by the number of equal values. Enter the result of the division in col6 for each of the equal values used to get the sum.

4. Add a seventh column (col7). The entries in col7 are equal in magnitude to the values in col6, but there signs (positive or negative) correspond to the signs of the values in column 3.

5. Get W1 as the sum of positive values in col7, and W2 as the absolute value of the sum of negative numbers in col7. W is the lesser of W1 and W2.

6. Obtain $W_{critical}$ (The expected value for W) from a table of critical values for the Wilcoxon Signed Rank test based on the value of n′ and the required significance level..

7. If $W \leq W_{critcal}$, Reject the null hypothesis. Otherwise the null hypothesis is not rejected.

Table 3.8: Calculation of W for the Wilcoxon Signed Rank test

col1	col2	col3	col4	col5	col6	col7
33	40	-7	7	6	6.5	-6.5
51	49	2	2	2	2	2
76	72	4	4	3	3.5	3.5
84	80	4	4	4	3.5	3.5
85	85	0				
65	60	5	5	5	5	5
97	90	7	7	7	6.5	6.5
14	15	-1	1	1	1	-1

W1=20.5, W2=7.5, W=7.5, n'=7, for a one tailed test, at a significance level of 0.05,$W_{critical}$=3. Therefore the null hypothesis is not rejected. This means that the median of the differences between the values in col1 and col2 is less than zero (the null hypothesis).

Stangroom (2014) provides a web based calculator that can be used to get the W statistic given the pairs of values being compared. We use this calculator to perform the Wilcoxon Signed Rank test in our study.

3.5.5 *Simulation time difference*

We can compare two learning techniques by getting the difference between the amount time taken by each of the techniques to get to a specified performance level. The time taken by a learning algorithm to arrive at a specified performance can be calculated based on the number of episodes it takes to arrive at the performance. The number of episodes can be converted to actual simulation time. This is based on the following formula.

$$\text{time taken} = \text{No of episodes} \times \text{actions per episode} \times \text{time per action (in seconds)} \quad \textit{Equation 3.7}$$

Although the maximum number of actions in an episode is set to 100, it was found that the average number of actions per episode was 40. The time per action was set to 0.5 seconds. The time taken can therefore be divided by 3600 to convert the value to hours.

4. Simulation System Development Methodology

This chapter begins by introducing some learning agent models. Learning agent models generally suggest the components and the structure of learning agents. Evolutionary prototyping, the system development methodology adopted in this study is then discussed. This is followed by a section that list the requirements that the system developed was expected to have. Following this is a system design section that discusses how the system was structured in order to meet the requirements. The design section contains the following subsections: the overall system architecture; the selection of the robotics simulator; a detailed description of the SARSA(λ) algorithm particularly as it used with gradient descent based function approximation methods; Descriptions of the modifications to the SARSA(λ) algorithm to include policy transfer, policy reuse and experience replay; a description of the TDBackpropagation algorithm as used for neural network updates in this study; and the UML diagrams including package diagrams, and class diagrams of the components of the system. The chapter ends with a description of experiments conducted to set the learning parameters for the SARSA(λ) algorithm, neural networks function approximator, the policy reuse technique, and the experience replay technique.

4.1 Learning Agent Models

An agent can be defined as an entity that perceives its environment using sensors and acts on that environment using actuators (Russell & Norvig, 2003). In reinforcement learning, learning is supposed to take place via interaction with the environment (Sutton & Barto, 1998). The main tasks of the agent in the reinforcement learning problem include:

1) Perceiving the state of the environment using sensors
2) Perceive the reward generated by the environment for being in a particular state
3) Make decisions on how to act
4) Learn from the interaction on how to make better decisions to order to maximize the long-term reward signal

Figure 4.1 represents this learning model.

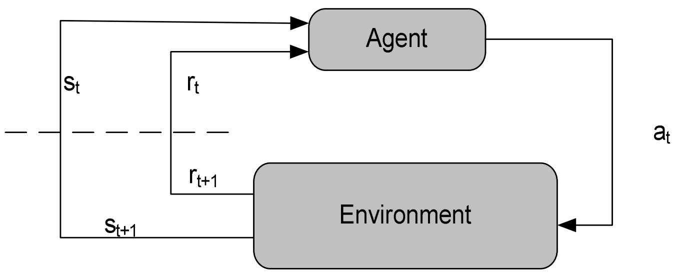


Figure 4.1: Agent/Environment Interaction in Reinforcement learning: Source (Sutton & Barto, 1998)

From the above model, the acting, sensing cycle occurs at discrete time intervals. This cycle is initiated by acquisitions of percepts and the reward signal from the environment at time *t* (*s_t and r_t*), the agent makes a decision on how to act and initiates an action also in time *t* (*a_t*). The next interaction cycle then begins and it will be assigned time *t+1*. It will start with the agent receiving percepts and the reward signal from the environment (*s_{t+1} and r_{t+1}*).

In this design, a robot that is controlled by the agent is part of the environment. However, there is an interface through which the agent can send commands to the robot. The sensors are also part of the environment, but the agent can read the state of the sensors via the robot agent interface (Sutton & Barto, 1998).
Russell & Norvig(2003) Provide a more detailed view of a learning agent as shown in Figure 4.2.

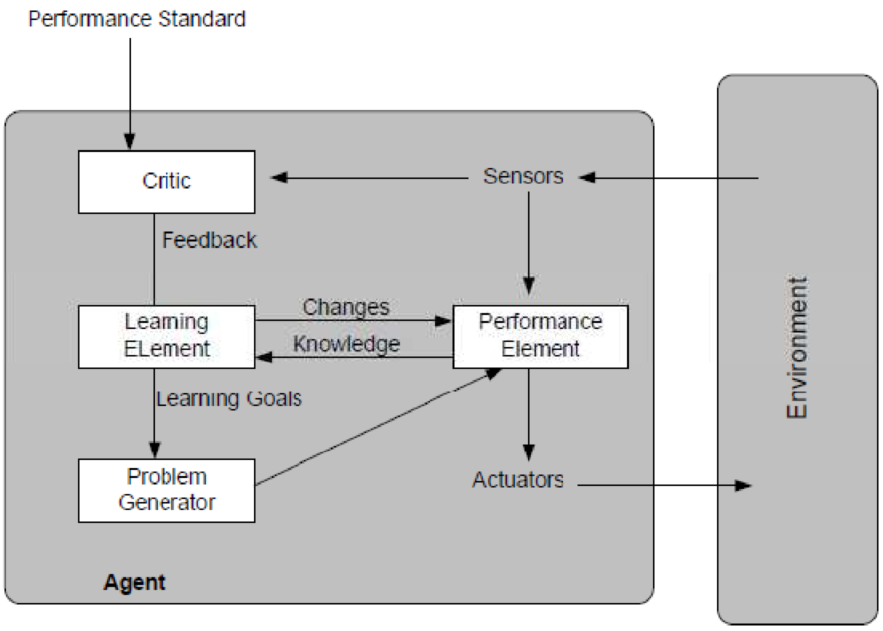


Figure 4.2: Model of a learning agent. Source (Russell & Norvig, 2003)

In the model in Figure 4.2, the sensors and the actuators are seen as being part of the robot. This model emphasizes the aspect of autonomy in addition to learning. In the model in Figure 4.1, the agent and the robot need not share the same physical body as long as they can communicate in some way. In the later model, the agent and the robot would need to share the same physical body.

The two models can be reconciled in that as long as actuators whether situated in an external robot or in the agent are used to act on the environment, while the sensors are used to perceive the state of the environment. If the notions of self-sufficiency and complete autonomy Pfeifer (1997) are relaxed, then the sensors and actuators can be removed from inside the agent box in Figure 4.2.

Russell & Norvig (2003) define the different elements in Figure 4.2 as follows:

Performance Element: This component is responsible for decision making. Given the current state of the environment, it selects the action that should be performed.

Learning element: This component is responsible for making improvements to the performance element given the feedback from the critique on how the agent is performing.

Critic: This component tells the agent how well it is performing in respect to some predefined performance standard.

Problem Generator: This component is responsible for selecting actions that will lead to novel experiences.

In the context of reinforcement learning, the performance element is the *control policy* while the learning element is the mechanism that decides how the policy should be updated. The critic is the reward processing mechanism while the problem generator is inbuilt into the control policy by ensuring that it is ***stochastic***. The problem generator can also consist of a set of random tasks which the agent can experience and from which it can learn.

4.2 Evolutionary Prototyping

Evolutionary prototyping is one of several software development processes that are categorised under evolutionary process models (Aggarwal & Singh, 2008). These can be used in projects that use new technologies that may not be well understood and documented. They are also suitable in situations where the requirements may not be very clear at the beginning of the project. Aggarwal & Singh (2008) give the following as some of the conditions under which evolutionary prototyping can be selected as the software development process:

i. Requirements are not easily understandable and defined
ii. Requirements change quite often
iii. Requirements are indicating a complex system is to be built
iv. The team has less experience in building similar systems
v. The team has less domain knowledge
vi. The team has less experience on the tools to be used

Figure 4.3 shows the steps in the evolutionary prototyping process.

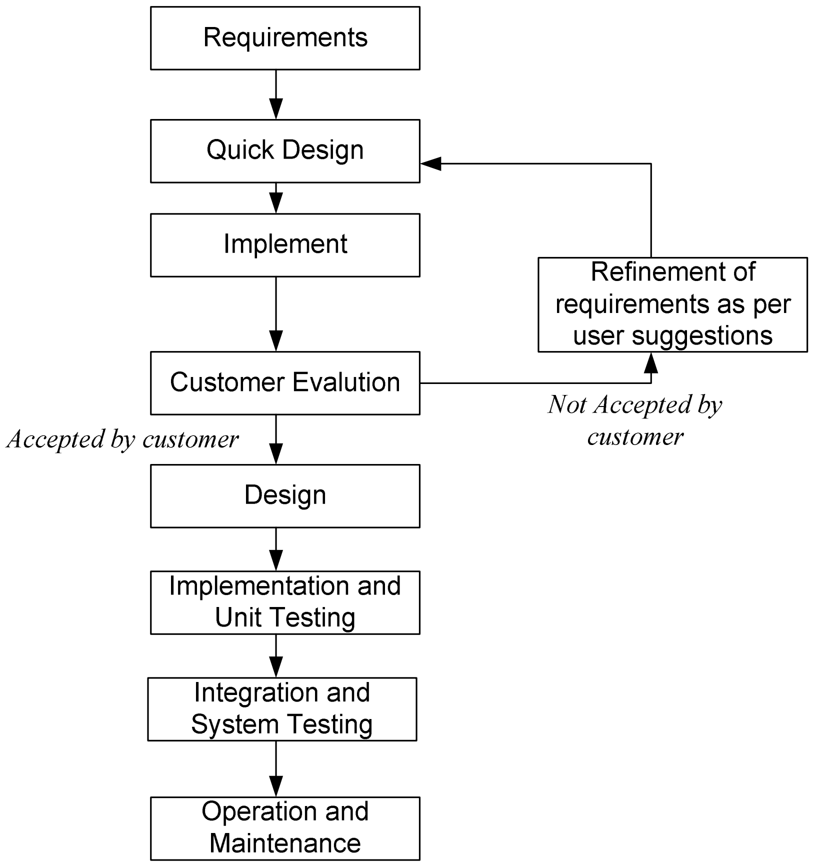

Requirements
Quick Design
Implement
Refinement of requirements as per user suggestions
Customer Evalution
Not Accepted by customer
Accepted by customer
Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maintenance

Figure 4.3: Prototyping Model. Source (Aggarwal & Singh, 2008)

The prototyping process may result in what is referred to as a *throwaway prototype* or an *evolutionary prototype* (Ghezzi et al., 2003). The throwaway prototype is a product developed with the goal of clarifying requirements after which the system specification document can be created(Sommerville, 1992). The evolutionary prototype on the hand is progressively transformed into the final product (Ghezzi et al., 2003). Sommerville (1992) points at artificial intelligence (AI) systems as examples of systems that may be difficult to specify from the beginning. According to Ghezzi et al. (2003), developers usually have only a vague idea of how AI systems are going to work.

4.3 System Requirements

Given the learning models in Figure 4.1 and

, and put into consideration the objectives of the study, the following requirements were identified.

i. A robot with sensors and actuators
ii. A robot control agent that
    a) Gets the state of the state of the environment by reading the robot sensor data. The sensor sensors will need to provide signals that show the positions of obstacles and the position of the goal. The sensors will also need to provide signals that show if a collision with an obstacle has occurred.
    b) Issues commands to the robot to control its behaviour
    c) Is able to evaluate the behaviour of the robot in the environment based on the reward signal received from the environment
    d) Modifies its action selection mechanism to improve its performance in the environment.
    e) Has mechanisms to store the performance information of the robot to be used for comparison of different learning algorithms
    f) Generates problem tasks (Scenarios) from which the agent can learn

4.4 System Design

4.4.1 *System Architecture*

The robot features were provided by a robotic simulator. Components to provide other features and also to interface with the robot simulator therefore needed to be developed.

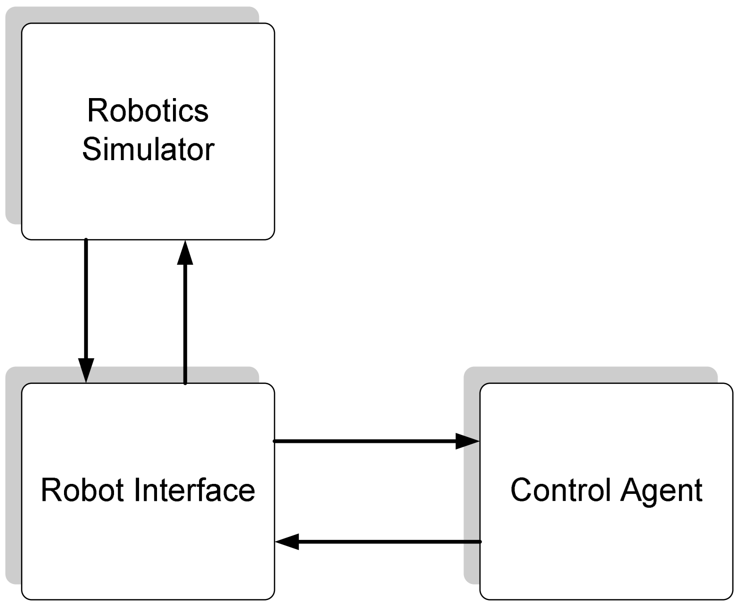


Figure 4.4: Robot Obstacle Avoidance System Architecture

The *Robotics Simulator* component is a software simulation of a robots environment. It enables definition of a robot that is situated in this environment. The robot defined in this environment should have adequate sensors to capture environmental status information. For our purposes, this included: object sensors e.g. laser range finder, camera, bump sensors; and location sensors (GPS sensor). The robot should also respond to steering commands so that its movement in the environment can be controlled.

The *Robot Interface* component interacts with the robot simulation environment through the interfaces it exposes in order to get sensor readings. After getting the sensor readings, it processes them as is necessary in order to extract state information. This component is also responsible for sending steering commands to the robot.

The *control agent* component decides what steering commands are to be sent to the robot (via the *Robot Interface*) given the current state of the environment. It receives the environment status information from the *Robot Interface* component. This component is also responsible for extracting reward information from the state information it receives from *the Robot Interface*. It contains the *performance element* and the *learning element* described in section 4.1. The *learning element* decides how the *performance element* is to be updated given the state information and the rewards from the environment. This component also contains the *problem generator* that generates novel tasks that lead to learning experiences.

4.4.2 *Selection of a Robotics Simulator*

It is usually prudent to perform simulations on mobile robots to investigate some phenomena of interest before performing experiments on real robots (Michael, 2004). Compared to real robots, simulations are easier to setup, cheaper, faster and the parameters that represent the status of the simulation can be displayed on the screen (Michael, 2004).

Robotics simulators are software programs that allow users to create virtual worlds in which natural phenomena like gravity, friction, and lighting conditions can be defined. These environments also allow creation of objects (entities) in the world that can be affected by physical laws and forces of nature. Thus, the objects will realistically interact with the environment and with each other. Usually, these environments support 2D or 3D rendering of the environment and the objects in the environment for visual demonstration of the evolution or changes of the environment with time (Palmisano, ND). Simulators also usually contain a *physics engine* which is a software that provides approximate simulation of some physical phenomena e.g. collision detection, gravity and friction. There are several robotics simulation software that are used in robotics research including Webots (Cyberbotics , 2014), Gazebo (gazebosim.org, ND) and Robotics Developer Studio (RDS) (Microsoft, 2011). The Microsoft Robotics Developer studio was selected for the following reasons:

1) It is available for free unlike *Webots*
2) It has better documentation has compared to Gazebo.

   Even though documentation exits for Gazebo, it is spread out over various locations on the internet and it may be difficult to locate the specific information that one seeks. RDS documentation on the other hand exists in one main document the RDSUserguide which is available on the web and can also be downloaded for local storage.

   RDS also provides numerous tutorials and samples that can be used as a starting point to develop and run simple robotics applications.
3) RDS uses the Visual Studio development environment (version 10) which is a mature environment providing tools for RDS project development, building, installation, debugging, and GUI development among others.
4) The DSS runtime environment in which RDS services run outputs descriptive error messages in case of errors or exceptions in the execution of the RDS application.

4.4.3 *The SARSA(λ) Learning algorithm*

The SARSA(λ) algorithm is the basic algorithm used for learning in this study. The version used when gradient descent based function approximation methods are used to represent the value function is known as gradient-descent SARSA(λ)Sutton & Barto (1998) and is given below.

```
1.  Initialise $\vec{\theta}$ randomly
2.  **Repeat** for each episode
3.      $\vec{e} = \vec{0}$
4.      Initialise *s*
5.      Use the stochastic policy $\pi$ (such as ε-greedy) derived from $\vec{\theta}$ to select the
        first action *a* of the episode
6.      **Repeat** for each step of the episode
7.          Take action *a*, observe the next state **s′** and reward *r*
8.          Use the stochastic policy $\pi$ (such as ε-greedy) derived from $\vec{\theta}$ to
            select the next the next action *a′* of the episode
9.          $Q(s,a) = \vec{\theta}(s,a)$
10.         $Q'(s',a') = \vec{\theta}(s',a')$
11.         $\delta = r + \gamma Q'(s',a') - Q(s,a)$
12.         $\vec{e} = \lambda\gamma\vec{e} + \nabla_{\vec{\theta}} Q(s,a)$
13.         $\vec{\theta} = \vec{\theta} + \alpha\vec{e}\,\delta$
14.         s=s′,    a=a′
15.         **until s** is terminal
16. **GOTO** step 2
```

Algorithm 1: The gradient-descent SARSA(λ).

In step 1, the parameters of the function approximation technique are initialised. $\vec{\theta}$ may for instance represent the weights of a neural network. In step 3, the eligibility traces are initialised to 0 (at the beginning of each episode). Each parameter in $\vec{\theta}$ has an eligibility trace in $\vec{e}$ associated with it. The eligibility trace represents by how much the parameter it is associated with is responsible for the TD error in step 11. A parameter with a big trace value will under go greater change (increment or reduction) because it will bear most blame for the error.

In step 4, the initial state of the episode (the start state is selected). In step 5, the first action of the episode is also selected. In step 7, the currently selected action is executed and the next state ***s'*** and reward ***r*** are observed and recorded. In step 8, since now we have the next state ***s'*** we can select the next action ***a'***. The tuple **<** ***s, a, r, s', a'*****>** contains the items required to update the value function.

Step 9 to 10 represents the calculations that are used to update the value function. In this case, the value function is updated by making changes to the parameters in $\vec{\theta}$. In step 9, the value function represented by $\vec{\theta}$ is used to calculate the value associated with the current state ***s*** and action ***a***. in step 10, the same function is used to calculate the value associated with the next state **s'** and action **a'**. In step 11, the TD error is calculated using the values obtained in step 9 and 10, and the reward ***r*** obtained in step 7.

In step 12, the eligibility traces $\vec{e}$ are updated. γ is the learning rate and λ is the trace decay rate. Note that the output of the last term in the equation is a vector that consists of the derivative of the output of the value function given the current state and action, with respect to the vector of parameters. In step 13, the function parameters are updated using their current values, and a product of the learning rate α, the eligibility traces $\vec{e}$, and the TD error δ.

In step 14, the next state becomes the current state and the next action becomes the next action. In step 15, if the current state is not a terminal state, the execution moves to step 7 to continue with the current episode. Otherwise the current episode ends, and in step 16, execution moves to step 2 to start a new episode. This is repeated for ever to achieve lifetime learning.

4.4.4 *Incorporating Experience Replay into SARSA(λ) algorithm*

In order to incorporate experience replay, we need to remember that to perform an update in steps 9 to 13 of Algorithm 1, we need the tuple <s, a, r, s′,a′>. Such a tuple is known as an experience. In normal learning, each such tuple is used only once to update the value function. When ER is in use, these tuples are saved and used multiple times to perform updates. The sequence of tuples are saved per episode. Table 4.1 shows what the saved tuples may look like.

Table 4.1: Experiences stored for experience replay

Episode	Experience
1	$<s_0,a_0, r_0>, <s_1,a_1, r_1>,\ldots,<s_{n1},a_{n1}, r_{n1}>$
2	$<s_0,a_0, r_0>, <s_1,a_1, r_1>,\ldots,<s_{n2},a_{n2}, r_{n2}>$
⋮	⋮
N	$<s_0,a_0, r_0>, <s_1,a_1, r_1>,\ldots,<s_{nN},a_{nN}, r_{nN}>$

N represents the number of episodes whose associated experiences we are willing to store. Different episodes may have different lengths, hence n_i represents the length of that particular episode. s_{ni} represents the state before the terminal state in an episode. The terminal state is never used for training. To perform experience replay, we can use all the episodes in the table or randomly select a few. We then update the value function using the experiences selected from the tuples associated with a selected episode. For instance to use episode 1 for training, we can create the following tuples

$<s_0, a_0, r_0, s_1, a_1>, <s_1, a_1, r_2, s_2, a_2>, \ldots$

To use the last experience $<s_{n1},a_{n1}, r_{n1}>$ for replay, in step 11 of Algorithm 1, the second term is left out of the equation.

The replay can happen between steps 15 and 16 in Algorithm 1, which is at the end of the current episode. The replay can be done at the end of each episode, or after several episodes have passed. Replay is accomplished by converting steps 9 to 13 of Algorithm 1 into a function which can be called from the replay position, with the replay tuple provided as the input.

Experience replay statements can be enclosed in selection statements which, if the condition of the selection statement is true, ER statements will be executed, otherwise they will be ignored. This essentially turns ER either ON or OFF.

4.4.5 *Incorporating Policy transfer into SARSA(λ) algorithm*

In step 1 of Algorithm 1, the function parameters $\vec{\theta}$ are initialised randomly. In policy transfer, there exists another task which has already been learnt. The policy for this function is represented by a vector of parameters $\vec{\theta}_s$. Instead of initialising $\vec{\theta}$ randomly, we can initialise $\vec{\theta}$ using the values of $\vec{\theta}_s$. Inter task mapping is necessary in order to decide which parameters in $\vec{\theta}$ will be initialised with which parameter in $\vec{\theta}_s$. Generally, if a parameter in $\vec{\theta}$ is not mapped to any parameter in $\vec{\theta}_s$, it will be initialised randomly. To avoid this, we can have more than one parameter in $\vec{\theta}$ being initialised with the value of the same parameter in $\vec{\theta}_s$. Once the initialisation is done $\vec{\theta}_s$ can be discarded.

Just like in experience replay, policy transfer statements can be enclosed in selection statements which, if the condition of the selection statement is true, policy transfer statements will be executed, otherwise they will be ignored. This essentially turns policy transfer either ON or OFF.

4.4.6 *Incorporating Policy reuse into SARSA(λ) algorithm*

In policy reuse, there exists another task which has already been learnt. The policy for this function is represented by a vector of parameters $\vec{\theta}_s$. In steps 5 and 8 of Algorithm 1, we use the policy derived from $\vec{\theta}$ to select actions. When policy reuse is implemented, in some episodes, we will be using a policy derived from $\vec{\theta}_s$ to select actions instead the policy derived from $\vec{\theta}$. Generally, we will have to probabilistically decide which of the two policies to use in each episode. The probability for using $\vec{\theta}_s$ should reduce as the performance of $\vec{\theta}$ improves. Either way, only $\vec{\theta}$ is updated in steps 9 to 13 of the algorithm.

Just like in experience replay, policy transfer, policy reuse statements can be enclosed in selection statements which, if the condition of the selection statement is true, policy reuse statements will be executed, otherwise they will be ignored. This essentially turns reuse either ON or OFF.

4.4.7 *TDBackpropagation Algorithm*

TDBackpropagation is a learning algorithm that combines temporal difference learning with the Backpropagation algorithm used in training of feed-forward multilayer neural networks (McClellad, 2013). It provides a way to modify neural network weights based on the error obtained by getting the difference between the predicted value and the target value. It is an implementation of steps 9 to 13 of Algorithm 1 for multilayer neural perceptrons. While in supervised learning using neural networks the target value is provided, in reinforcement learning, the target value is not provided and only becomes available at the end of the episode for episodic tasks. This target value that becomes available at the end of the episode is referred to as the *return* and it is the discounted sum of rewards obtained from a given state to the final state. For an episodic task, the return can be defined as shown below (Sutton & Barto, 1998):

$$R_t = r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{k-1} r_{t+k}$$ Equation 4.1

The parameter γ, $0 \le \gamma \le 1$, is referred to as the *discount rate* and it determines the value of future rewards. If γ is zero, then future rewards are not important, but if γ is one, future rewards are as important as immediate rewards.

Where *k* is the number of states encountered after the state encountered at time t (s_t) and $r_{t+1} \ldots r_{t+k}$ are the rewards encountered after making a transition to states ($s_{t+1} \ldots s_{t+k}$). State s_{t+k} is a terminal state. In some tasks, all intermediate rewards may be zero except r_{t+k}, the reward obtained after transitioning to the final state.

Sutton & Barto (1998) give the following equations for update of a vector of parameters defining the function approximator using the SARSA(λ) algorithm.

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \beta_t \vec{e}_t$$ Equation 4.2

Where

$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}} Q(s_t, a_t)$ *Equation 4.3*

$\beta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$ *Equation 4.4*

$\vec{e}_t$ is a value referred to as the eligibility trace with each parameter element (weight) having its own trace. The eligibility trace for a given weight element determines the how much the element is responsible for the error observed currently (β_t) hence the weight of the update in relation to the current error (β_t). β_t is the error as calculated in SARSA(λ) algorithm. γ and λ are the *discount rate* and the *trace decay* parameters as used in SARSA(λ).

A 3 layer feed-forward neural network will have one output layer, one hidden layer and one input layer. We will therefore need to derive the equation to update the weights elements for each particular node in either the hidden layer and the output layer. In the following discussion, weights incident on a particular node are taken to belong to that node. In order to define the update rule, we need to define the output (activation) function for the hidden and output units. One popular choice is the sigmoid function $o_n = \frac{1}{1 + e^{-\sigma net_n}}$ where *net*$_n$ is the sum of inputs to the neural network unit *n* (Mitchel, 1997). With this function we have $0 \le o_n \le 1$. σ is a slope parameter usually referred to as the *gain* (Nawi et al., 2009). It adjusts the rate at which the function moves from 0 to 1, with higher values giving a steeper slope (Rajasekaran & Vijayalakshmi, 2003). This function may not be the most appropriate for an obstacle avoidance task where rewards of values less than zero e.g. -1 are given when the robot collides with an obstacle and a reward of a value greater than 0 is awarded for successfully reaching the goal.

The function chosen for this task was the bipolar sigmoid function defined as $o_n = \tanh(\sigma net_n)$. With this function we have $-1 \le o_n \le 1$.

Given an input *x*, the derivative for this function with respect to *x* is *1-tanh²(x)*. If *x* is multiplied with a constant (σ) then the derivative is given by Equation 4.5.

$\nabla_x \tanh(\sigma x) = \sigma(1 - \tanh^2(\sigma x))$ *Equation 4.5*

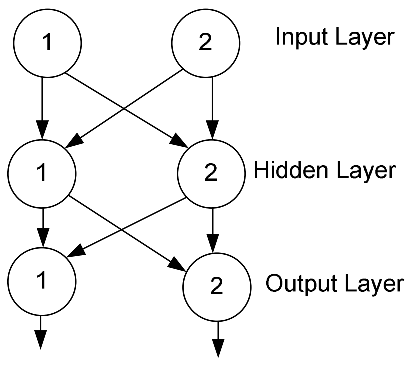


Figure 4.5: Example 3 layer neural network

In the following discussion we will refer to the output of an input node (an input to the network) as O_{Ii} where *i* is the index of the node and *I* indicates that it is an input node, the output of an hidden layer node as O_{hi} and output of an output layer node as O_{oi}. The sum of inputs to a hidden layer node *i* will be referred to as sum_{hi} and the sum of inputs to an output node as sum_{oi} where *i* represents the index of the node. A weight link from an input layer node to a hidden layer node will be referred to as w_{hji} where *h* specifies that the hidden layer node owns the weight, *j* is the index of the input node and *i* the index of the hidden layer node. A weight link from a hidden layer node to an output node will be referred to as w_{oji} where *j* is the index of the hidden layer node, *i* the index of the output layer node, *o* indicates that the output layer node owns the weight.

4.4.7.1 Update for an output layer node

In the following discussion we will use Figure 4.5 to illustrate. Given a node in the output layer, it has as many inputs as there are hidden nodes plus the bias. Hence forth, we will assume that the bias is just another node except for the fact that it always outputs a value of 1. Using Figure 4.5 as an example, node 1 in the output layer has only 2 inputs (from node 1 and 2 in the hidden layer). There also two weights associated with these inputs that that we can call w_{o11} and w_{o21}.

To get the sum of inputs to output node 1, we use Equation 4.6.

$$sum_{o1} = w_{o11}O_{h1} + w_{o21}O_{h2} \qquad \textit{Equation 4.6}$$

The actual output for output node 1 is calculated using Equation 4.7.

$$O_{o1} = \tanh(\sigma sum_{01}) \qquad \textit{Equation 4.7}$$

A gradient will have to be obtained with respect to all the weights. For the weight w_{o11} from the first hidden layer node to the first output layer node, the gradient is obtained Equation 4.8.

$$G_{o11} = \sigma(1 - O_{o1}^2)O_{h1} \qquad \textit{Equation 4.8}$$

Where G_{o11} indicates that this is a gradient of output node 1 with respect to the weight connected to the first hidden layer node. O_{h1} is the output from the hidden layer node 1 which is multiplied with w_{o11} in Equation 4.6 in getting the sum. For the second weight of the same node (w_{21}), the gradient is calculated using Equation 4.9.

$$G_{o21} = \sigma(1 - O_{o1}^2)O_{h2} \qquad \textit{Equation 4.9}$$

We note that that $\sigma(1 - O_{o1}^2)$ is common in the gradients for both weights. This value is usually referred as the *delta* (δ) and is the same in calculating the gradient for all the weights of a given output node (McClellad, 2013). Hence for a given output node *n*, we calculate the delta (δ_{on}) once, then to get the gradient associated with a given weight, we multiply the delta with the input (output of an hidden node) that was multiplied with that weight in the sum (Equation 4.6). Note however that the deltas for different output nodes are different.

Equation 4.10 is the general equation used to calculate the delta for any output node *n.*

$$\delta_{on} = \sigma(1 - O_{on}^2) \qquad \textit{Equation 4.10}$$

And the gradient with respect to a given weight w_{ji} where *j* is an hidden node, *i* is an output node and w_{ji} is the weight between them is given by *Equation 4.11*.

$$G_{oji} = \delta_{oi}O_{hj} \qquad \textit{Equation 4.11}$$

For any output node *n,* we will have as many gradients as there are hidden nodes. For instance output node 1 in Figure 4.5 will have one delta and 2 gradients.

We use Equation 4.3 to calculate the eligibility trace for a specific output node weight.

For instance for w_{o11} the eligibility trace is calculated using Equation 4.12.

$$e_{o11} = e_{011}\gamma\lambda + G_{o11} \qquad \textit{Equation 4.12}$$

And for the second weight of the first output node we use Equation 4.13.

$e_{o12} = e_{012}\gamma\lambda + G_{o12}$ Equation 4.13

In general, the eligibility trace update for weight w_{ji} from node j in the hidden layer to node i in the output layer is obtained using Equation 4.14.

$e_{oji} = e_{0ji}\gamma\lambda + G_{oji}$ Equation 4.14

All eligibility traces are initialised to zero at the beginning of an episode since no events have occurred and no state action pair is eligible for update.

To update the weight, we use Equation 4.2 from which we derive Equation 4.15 for use in updating first weight in the first output node.

$w_{o11} = w_{o11} + \alpha e_{o11}\beta_1$ Equation 4.15

In general, for any output node weight, Equation 4.16 is used for update.

$w_{oji} = w_{oji} + \alpha e_{oji}\beta_i$ Equation 4.16

β_i is obtained using Equation 4.4. β_i can therefore not be known until after the current action a_t has been executed, the next state s_{t+1} obtained, the reward r_{t+1} obtained, the next action a_{t+1} selected, and the value of the next state action pair $Q(s_{t+1},a_{t+1})$ obtained. $Q(s_t, a_t)$ is already known and corresponds to O_{oi} the output of the output node under consideration given state s_t as the input.

4.4.7.2 Update for an Hidden Layer Node

For a given output layer node weight, the gradient is the derivative of the output of a given output layer node with respect to some weight element from a hidden layer node to the output layer node.

In the hidden layer, the weights come from the input layer to the hidden layer. We are supposed to get the derivative of the output of a given output layer node with respect to a weight element in a link between an input node j and a hidden node i (w_{hji}). McClellad (2013) shows how this can be done for the sigmoid function. We will show how it can be done for the bipolar sigmoid function. For a given weight between an input layer node and an hidden layer node, say w_{h11} (the weight from the first input node to the first hidden node), if we have say two output nodes as shown in Figure 4.5, we have to get the gradient with respect to the w_{h11} for the output of each of the output layer nodes. We will therefore have 2 gradients for this one weight. If we select an output node, say O_1, we need to obtain $\nabla w_{h11} O_{o1}$, the gradient of the first output node with respect to w_{h11}.

Equation 4.17 is used to get the sum of inputs to hidden layer node 1 (sum_{h1}).

$$sum_{h1} = w_{h11} I_1 + w_{h21} I_2 \qquad \textit{Equation 4.17}$$

Equation 4.18 gives the output of hidden node 1.

$$O_{h1} = \tanh(\sigma sum_{h1}) \qquad \textit{Equation 4.18}$$

The output of output node 1 (O_{o1}) is given in Equation 4.7. Equation 4.19 is used to get the gradient of the output of the first output node with respect to w_{h11}.

$$\nabla w_{h11} O_{o1} = \frac{\partial \tanh(\sigma sum_{o1})}{\partial w_{h11}} \qquad \text{Equation 4.19}$$

Using the chain rule, this simplifies to

$$\frac{\partial \tanh(\sigma sum_{o1})}{\partial w_{h11}} = \frac{\partial \tanh(\sigma sum_{o1})}{\partial sum_{o1}} \times \frac{\partial sum_{o1}}{\partial O_{h1}} \times \frac{\partial \tanh(\sigma sum_{h1})}{\partial sum_{h1}} \times \frac{\partial sum_{h1}}{\partial w_{h11}} \qquad \textit{Equation 4.20}$$

$$\frac{\partial \tanh(\sigma sum_{o1})}{\partial w_{h11}} = \sigma(1-\tanh^2(\sigma sum_{o1})) \times w_{o11} \times \sigma(1-\tanh^2(\sigma sum_{h1})) \times O_{I1} \qquad \textit{Equation 4.21}$$

$$\frac{\partial \tanh(\sigma sum_{o1})}{\partial w_{h11}} = \sigma(1-O_{o1}^2) \times w_{o11} \times \sigma(1-O_{h1}^2) \times O_{I1} \qquad \textit{Equation 4.22}$$

This then is the gradient of the output of output layer node 1 with respect to the first weight of the first hidden layer node. The gradient of the output of the first output layer node with respect to the second weight of the first hidden layer node is given by Equation 4.23.

$$\frac{\partial \tanh(\sigma sum_{o1})}{\partial w_{h12}} = \sigma(1-O_{o1}^2) \times w_{o11} \times \sigma(1-O_{h1}^2) \times O_{I2} \qquad \textit{Equation 4.23}$$

Note the difference between Equation 4.22 and Equation 4.23 in the last value multiplied to the equation (O_{I1} and O_{I2}). This represents the input associated with the weight used for differentiation. The value $\sigma(1-O_{o1}^{2})\times w_{o11}\times\sigma(1-O_{h1}^{2})$ is referred to as the delta for the hidden node. A hidden node has a delta associated with each of the output nodes.

For instance hidden node 1, has 2 deltas one associated with the first output node, and another associated with the second output node. The equation for the second delta (associated with the second output node) is given in Equation 4.24 for the sake of comparison.

$$\sigma(1-O_{o2}^{2})\times w_{o12}\times\sigma(1-O_{h1}^{2})$$ *Equation 4.24*

Note also that the value $\sigma(1-O_{o1}^{2})$ (the first item in equation for delta) is nothing more than the delta for the output node obtained earlier as we calculated the gradient for the output node in *Equation 4.8*.

So to calculate the delta for a hidden node *j* with respect to output node *i*, we get the product of the delta for output node *i*, the weight from node *j* to *i*, and the value $\sigma(1-O_{hj}^{2})$ where O_{hj} is the output of hidden node j. This is summarised in Equation 4.25.

$$\partial_{hji}=\partial_{oi}w_{ji}\sigma(1-O_{hj}^{2})$$ *Equation 4.25*

For a given hidden node, there will be as many deltas as there are output nodes.

To get the gradient for the output of output node *i* with respect to weight w_{kj} from an input node *k* to hidden node *j*, we multiply the output of input node k, with the delta in hidden node *j* associated with output node *i* as depicted in Equation 4.26. So a weight w_{kj} will have has many gradients as there are output nodes. For instance in Figure 4.5, w_{h11} will have 2 gradients, one associated with output node 1and the other associated with output node 2.

$$G_{hkji}=\partial_{hji}O_{Ik}$$ *Equation 4.26*

In the gradient subscript, *h* implies that the gradient is associated with a weight in an hidden layer node, *k* and *j* are the input and hidden nodes linked by the weight, *i* is the output node since the gradient is of the output of some output node with respect to the weight.

Equation 4.27 is used to update the eligibility trace for e_{hkji} associated with weight w_{hkji} with the subscripts meaning the same as in Equation 4.26.

$$e_{hkji} = e_{hkji}\gamma\lambda + G_{hkji} \quad \text{Equation 4.27}$$

Equation 4.28 is then used for the weight update.

$$w_{hkj} = w_{hkj} + \alpha\sum_{i=1}^{N} e_{hkji}\beta_i \quad \text{Equation 4.28}$$

Where N is the number of output nodes, and β_i is the error associated with each output node calculated as shown in *Equation 4.4*. The same value (β_i) is also used in updating the weight for the output node in Equation 4.16.

4.4.8 *Class Identification*

The unified modelling language (UML) was used in the design process. UML is a modelling language providing a set of notations to create models of systems (Mall, 2003). According to Mall (2003), there are several advantages associated with the use of object oriented techniques including:

i. Code reuse by use of predefined class libraries
ii. Code reuse via the process of inheritance
iii. Abstraction of the system into entities that are more intuitive and easier to understand

According to Mall (2003), when the object oriented approach is used, a system is defined as consisting of a set of interacting classes, where each class may represent a tangible real world entity or some conceptual entity e.g. a scheduler or a controller. A class is a prototype of a set of objects that have similar properties.

The following classes were identified for the system:

i. ControlAgentService class

This is a control class that controls other classes that are responsible for action selection and learning. This class also specifies the mechanism for problem generation.

ii. ControlPolicy class

This is a conceptual class representing the performance component which is responsible for action selection. This component represents the policy and it has mechanism for policy improvement and action selection.

iii. ExperienceReplay class

This class is mostly responsible for storing past experience data and providing the data for use in training.

iv. MainWindow class

This provides A GUI from which the user can interact with the application

v. RobotInterfaceService class

This class is responsible for interacting with the simulation engine. It can send commands to entities (robots) in the simulation engine, and also get sensor information from the sensors currently in the simulated environment. It makes use of some predefined service classes in order to accomplish its functionality.

There may be other classes not explicitly stated here but which will appear in the class diagrams.

4.4.9 *Package Diagrams*

A package diagram helps in visualization of packages and the dependencies between them. A package helps to organize model elements into coherent logical groupings. In the following section *Microsoft.Robotics* is used to group together the components provided by Robotics developer studio for use in this project.

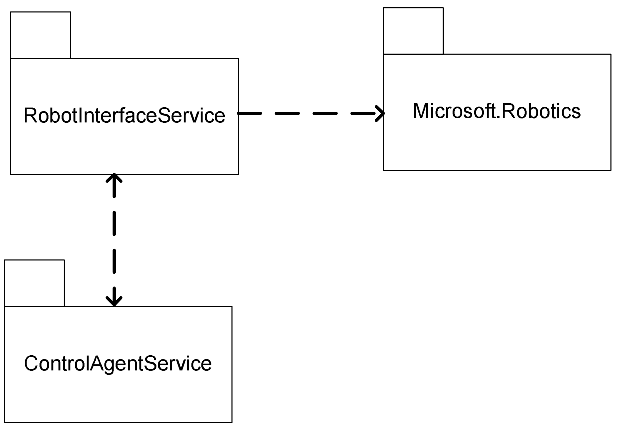


Figure 4.6: Mobile Robot Control System Package Relationship Diagram.

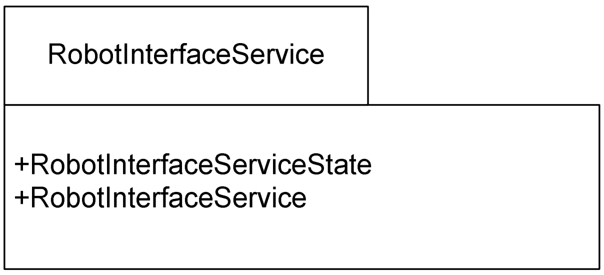


Figure 4.7: Robot Interface Service Package

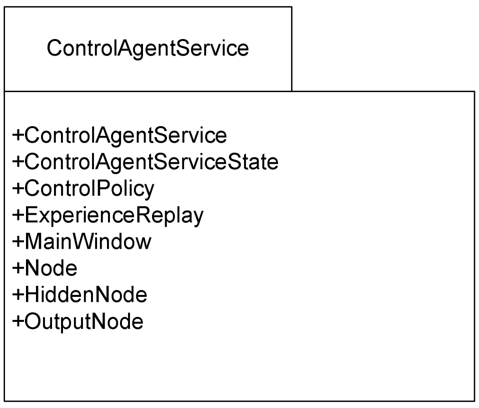


Figure 4.8: ControlAgentService Package

4.4.10 *Class Diagrams*

Class diagrams are used to represent the static structure of a system (Mall, 2003). It consists of a number of class diagrams and their dependencies. It shows classes and their relationships including generalization, aggregation and associations. The class themselves are depicted by a box with three compartments with the top compartment showing the name of the class, the middle compartment the attributes of the class, and the last (bottom) compartment showing the operations that can be performed by an instance of that class (Schach, 2004). In the following diagrams, for classes (services) imported from RDS, only the names compartment is filled.

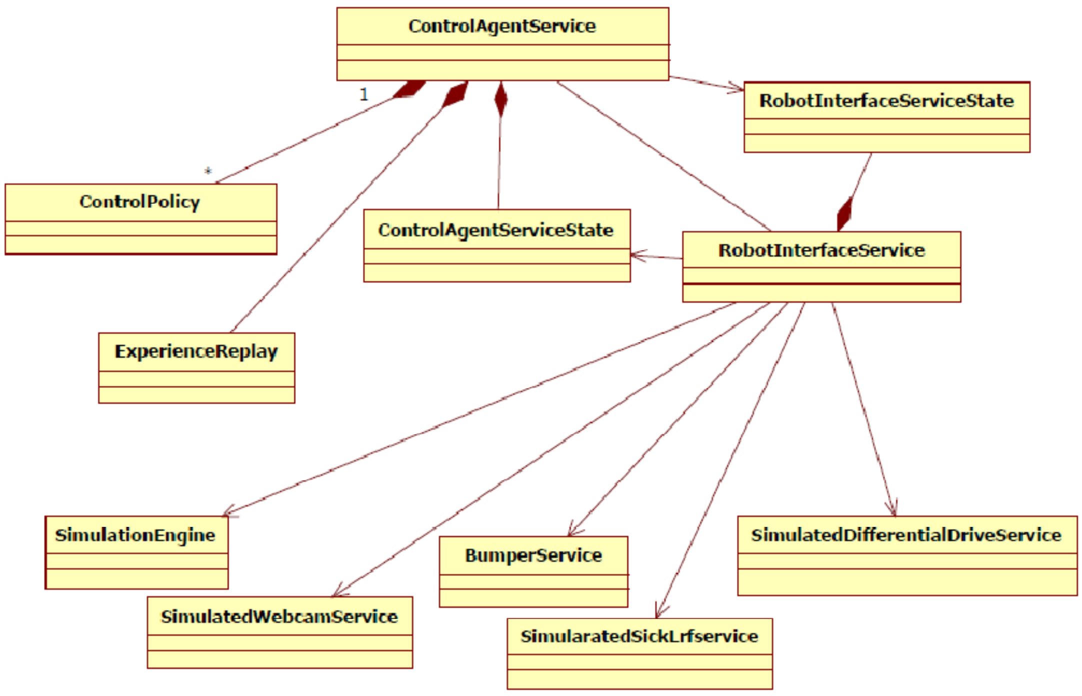


Figure 4.9: Robot Control System Main Class Diagram

The *ControlAgentService* is the main control class. It takes care of getting state information from the *RobotInterfaceService* which it converts to a form suitable for input to an instance of the *ControlPolicy* class. It also extracts the reward information from the state information received from the *RobotInterfaceService.* The *ControlPolicy* instance uses the state information to select the next action. It also uses the state information, previous actions taken and reward information to learn in order to improve performance.

The *ExperienceReplay* class is used to store previous experiences, which the *ControlAgentService* occasionally presents to the *ControlPolicy* for learning purposes.

Actions selected by the *ControlPolicy* are passed to the *RobotInterfaceService* by the *ControlAgentService.* The *RobotInterfaceService* passes these commands to the *SimulatedDifferentialDrive* which is responsible for setting the wheel speeds.

4.4.10.1 The ControlAgentService class

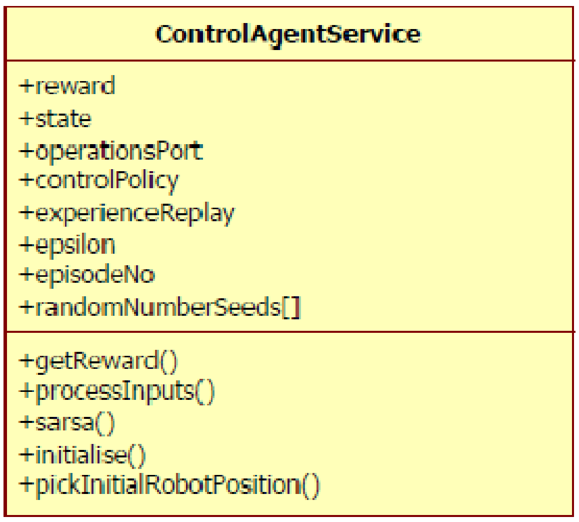


Figure 4.10: ControlAgentService class diagram

This class drives the operations of the whole application using the *sarsa* method. The sarsa method is an implementation of the SARSA(λ) algorithm. The following is the general sequence of steps followed by the sarsa method.

1) Send a message to the *RobotInterfaceService* asking for state information. Wait until a response is received before going to step 2
2) Get state information from the message received in step 1,
3) Get reward information from the message received in step 1
4) Get the next action and its value from the *ControlPolicy* by sending the state information to the *ControlPolicy*
5) Update the *ControlPolicy* using the previous state information, the current reward, the previous action and the next action value.
6) Set the state information to become the previous state information
7) Set the next action as the current action and send it to the *RobotInterfaceService*

The above sequence may vary if :

i. The goal has been reached, in which case the *pickInitialRobotPosition* method is used to select a random starting position for the next episode. This selected position is then sent to the *RobotInterfaceService*

ii. The action being taken is the first in an episode. In this case, the previous state and the previous action do not exist hence no training can occur.

iii. If experience replay is being used, the state, actions and reward will also need to be stored in the *experienceReplay* object.

4.4.10.2 RobotInterfaceService class

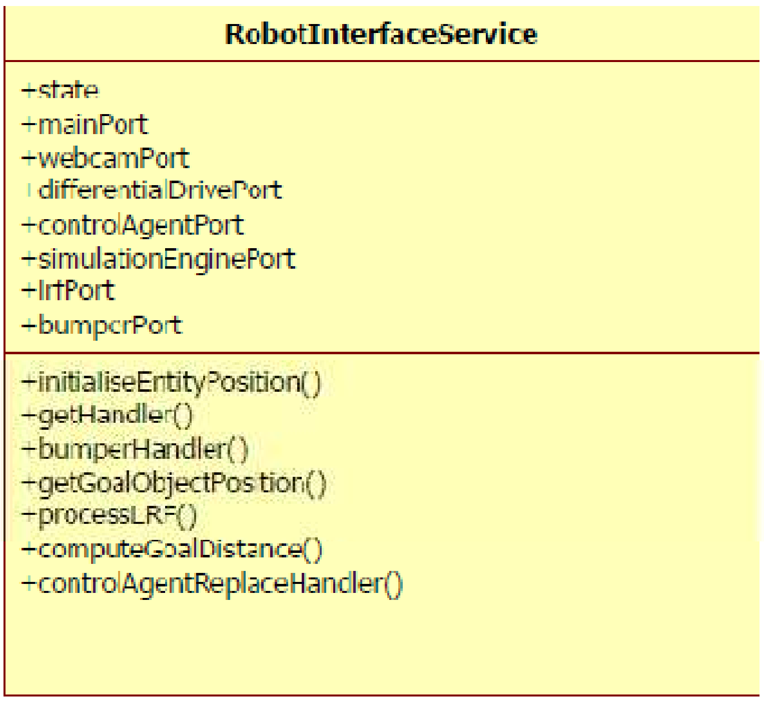


Figure 4.11: RobotInterfaceService class diagram

The *RobotInterfaceClass* interacts with the simulated sensor services, the simulation engine and the simulated differential drive. The data from the sensors is processed by the different methods and the obtained state information is stored in the *state* attribute. In the distributed software services (DSS) model, each service has a port through which it can receive messages from other services. The *mainPort* attribute in the class diagram represents this port.

The *initialiseEntityPosition* method is used to set the position of an entity in the simulated environment. For instance it can be used to set the initial position of the robot. This method interacts with the simulation engine service via the *simulationEnginePort*.

A service may receive a message from another service requesting that the receiving service sends its state to the requesting service. In DSS, the requesting service will need to send a

Get message to the receiving service. For instance the control agent may send a *Get* message to the *RobotInterfaceService*, the *RobotInterfaceService* will respond by sending its state variable to the control agent service. The *getHandler* method serves the purpose of responding to *Get* message request from other services.

The *bumperHandler* method receives messages from the simulated bumper service via the *bumperPort*. The received message indicates if the bump sensors have been pressed or not indicating that an obstacle collision has or has not occurred.

The *getGoalObjectPosition* method interacts with the simulated webcam service via the *webcamPort*. It requests for an image from the webcam and the webcam responds with a bitmap image. This method implements a blob identification algorithm based on the colour of the goal object. The algorithm looks for the object in the image that has a pre-specified colour. It does this by determining how close the colour of each pixel is to the specified goal object colour. The region with the most goal object colour pixels is taken to be the goal object region. This region must however contain at least 100 pixels to be considered as containing the goal. If no region has more than 100 goal object pixels, the goal is considered to be invisible. The method looks for the goal object in the image, and also its position in the image. The image is divided into eleven vertical stripes. The 6th stripe corresponds to the centre of the image, stripes 1 to 5 corresponds to position on the left of the centre with position 1 being the extreme left. Stripes 7 to 11 corresponds to positions on the right of the centre with position 11 being the extreme right. The centre stripe also represents the heading of the robot. So if the goal is in the centre stripe, the goal is directly ahead of the robot. Otherwise the goal is either on the left or right of the robot. The goal may also not be in the line of site of the robot in which case, the goal object will not be found in any of the 11 vertical stripes.

The *processLRF* method interacts with the laser range finder service via the *lrfPort*. The laser range finder is used to detect the positions of obstacles. It performs a 180 degrees sweep of the distance ahead and makes a reading at every 0.5 degrees interval. It therefore has a resolution of 361 measurements. This resolution was too high for our needs and so approximately every 33 consecutive readings were grouped together as one reading with the smallest value used as the representative reading. This represented the obstacle that was closest in the direction represented by the 33 consecutive readings. In total, we ended up with 11 readings from the LRF.

The *computeGoalDistance* method interacts with the simulation engine service. It gets the location of the goal, and the location of the robot, and uses these two 3D points to calculate the distance of the robot from the goal. It useful in determining if the robot as reached the goal region.

In the DSS model, when the state of a service changes, it can send a *replace* message to subscribed services informing them its state has changed. The *state* attribute of the *ControlAgentService* represents the current commands for the robot from the control agent. The replace message sent from the control agent to the *RobotInterfaceService* has the actuator commands for the robot. It may also have the robot position and orientation. The replace message is handled in the *RobotInterfaceService* by the *controlAgentReplaceHandler* method. It extracts the robot wheel power values from the replace message and sends them to the *simulatedDifferentialDrive* via the *differentialDrivePort*. If necessary (at the start of an episode), it extracts the robot pose from the message and instructs the simulation engine to place the robot in the specified position and orientation through the *simulationEnginePort*.

4.4.10.3 The ControlPolicy, Node and Node derived classes

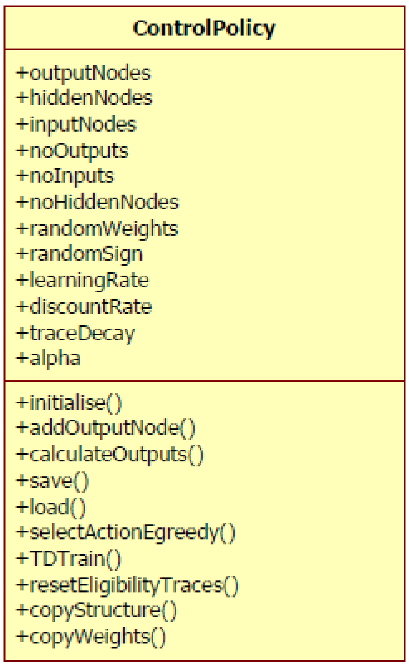


Figure 4.12: ControlPolicy Class diagram

The *ControlPolicy* class is a neural network learning element and therefore contains output and hidden nodes in a composition relationship as shown in the diagrams below.

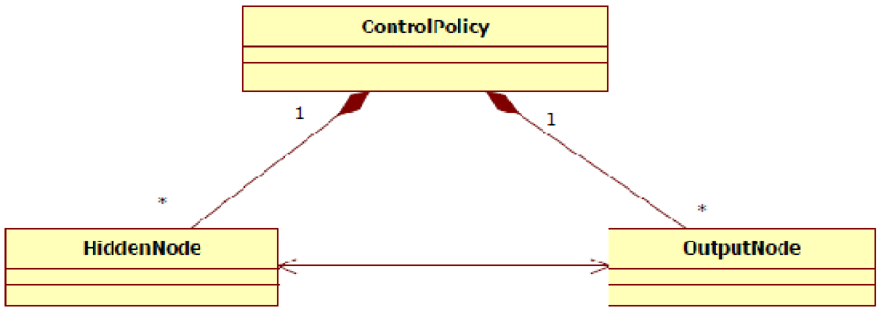


Figure 4.13: The relationship between the ControlPolicy class with the HiddenNode and OutputNode classes.

The *HiddenNode* and the *OutputNode* classes are specializations of the Node class as shown in Figure 4.14.

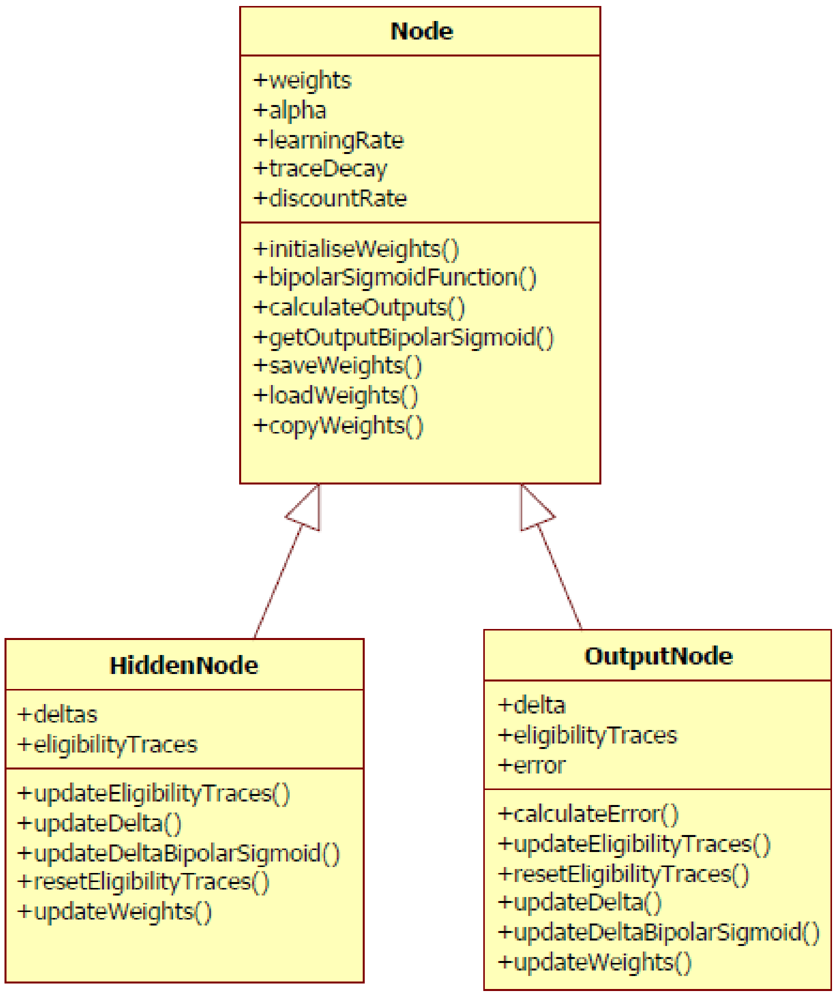


Figure 4.14: Node class specializations

The *ControlPolicy* class is an implementation of a feed-forward neural network so it has the common attributes found in such a network. It implements a 3 layer neural network with nodes in the input layer which do not transform their inputs in any way; nodes in the hidden layer that have bipolar sigmoid function output unit, and nodes in the output layer that also implement a bipolar sigmoid function output unit. The output nodes also correspond to the actions. If for instance a task has 3 actions, it will also have 3 output nodes. The *randomWeights* and *randomSign* attributes are random number generators used to initialise the weights in hidden and output nodes to values between -1 and 1. The *learningRate* corresponds to the value α discussed in section 4.4.7, while *discountRate* corresponds to γ, *traceDecay* to λ, and *alpha* to the gain (σ)all discussed in the same section.

The *initialise()* method instantiates the required number of input nodes, hidden nodes and outputs nodes and initialises their weights with random values.

The *addOutputNode()* methods adds a node to the current neural network consequently increasing the number of actions. This is used in the policy transfer. The method expects a source node to be specified. The source node is the node whose weights will be used to initialise the weights of the new node. The weights may also be initialised with random values or with average values (the average weight values of existing output nodes). The method may be called repeatedly to add more than one new node.

The *calculateOuputs()* method calculates the output of the output nodes, given the supplied values for the input nodes, first by calculating the output values of the hidden nodes, then using these as inputs to the output nodes.

The *save()* method is used to save a *ControlPolicy* class to a file while the *load()* method instantiates a *ControlPolicy* class from a file created using the *save()* method.

The *copyStructure()* and *copyWeights()* are used in the process of creating a new *ControlPolicy* object that is identical to an existing *ControlPolicy* object.

The *selectActionEgreedy()* takes the current state as input, and a small value ε , it then uses the *calculateOuputs()* method to get the outputs of all the output nodes. It can either return the index of the output node with the highest value (corresponding to the greedy action), or with probability ε randomly select the index of any of the output nodes (actions) and return it.

The *TDTrain()* method implements the *TDBackpropagation* algorithm discussed in section 4.4.3. It requires as input the value of the next action that is going to be executed O_{t+1}, the previous action that was executed a_t, the state in which the previous action executed was selected s_t, and the reward obtained after the previous action was executed r_{t+1}. It then goes through the following steps

1) Using s_t as inputs calculate the network output.
2) Set the delta term for all output nodes to zero
3) Call the output node corresponding to a_t (the current action) to calculate its delta term as described in section 4.4.7.1, (*updateDelta()*)
4) Call on each hidden node to calculate its delta term (*updateDelta()*)
5) Call on all output nodes to update their eligibility traces (*updateEligibilityTraces()*)
6) Call on all hidden nodes to update their eligibility traces (using *updateEligibilityTraces()*)
7) Call on the output node corresponding to a_t to calculate its error given O_{t+1}, r_{t+1} and its own output(*calculateError()*)
8) Set the error for all other output nodes to be the same as the error calculated in 7
9) Call on all output nodes to update their weights as discussed in section 4.4.7.1 using (*updateWeights())*
10) Call on all hidden nodes to update their weights as discussed in section 4.4.7.2 (*updateWeights()*)

4.4.10.4 ExperienceReplay and Related Classes

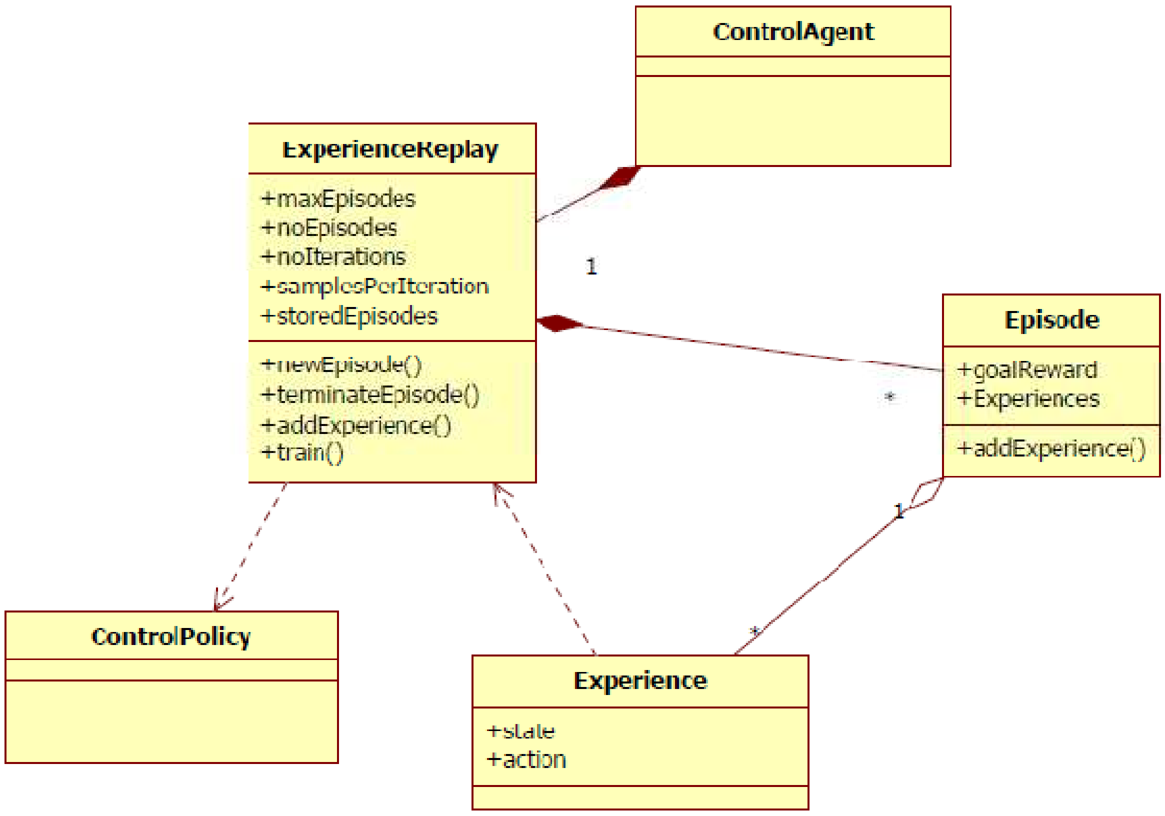


Figure 4.15: The ExperienceReplay class and the other classes it relates with

As the name implies, the *ExperienceReplay* class performs the experience replay function. This involves storing past experiences and reusing them to train the network at specified times.

An experience is defined as a state (environment state) in the form that is used as input to the *ControlPolicy* class, and the action taken in that state. Thus it can be said to consist of the pair (s_t, a_t). A sequence of experiences make up an episode. That is for a given episode, we have experiences (s_0, a_0), (s_1, a_1), (s_2, a_2),… (s_{k-2}, a_{k-2}), and state s_{k-1} which is the goal state in which we do not take any action. A sequence of experiences is stored in the *experiences* member of the *Episode* class. The *Episode* class also contains the *goalReward* member that species the value of the reward obtained at the end of the episode (1 or -1). The *Experience* class would have had an additional member (*reward),* but this is not necessary since all rewards are zero except when the goal state is reached. So in general, an *Episode* class stores the sequence of states encountered in an episode, the actions taken in each state, and the reward obtained at the end of the episode.

The constant *maxEpisodes* defines the maximum number of episodes that can be stored by an *ExperienceReplay* object. The *noEpisodes* member stores the actual number of episodes currently stored (this is useful if they are less than *maxEpisodes*).

The *newEpisode* method creates a new *Episode* object into which experiences can be added. Every new state action pair encountered is added to the *Episode* object by the *addExperience()* method. After the robot reaches the goal state, the *terminateEpisode()* method is used to set the *goalReward* member of the *Episode* object, and also to add the episode into the set of episodes (*storedEpisodes)*. In case the *storedEpisodes* is equal *to maxEpisodes*, *terminateEpisode()* implements either a random or oldest replace strategy that is used when any of the existing stored episodes is replaced by the newly completed episode.

The *ControlAgent* class is responsible for telling the *ExperienceReplay* class when to create a new episode, provision of experiences to add to the episode, and also when to terminate the episode. It is also responsible for telling the *ExperienceReplay* class when to use the stored experience to train the *ControlPolicy.*

The *train()* method in *ExperienceReplay* randomly picks *samplesPerIteration* episodes and uses them to train the control policy. This is repeated *noIterations* times.

When an episode is being used to train, two consecutive experiences are necessary. The first experience represents the state s_t that is the input to the network, and a_t represents the action hence the output node whose output value we are interested in. With the next experience, (s_{t+1}) as input to the network, we can get the output associated with a_{t+1} hence obtaining the target value. Therefore the first experience enables us to obtain $Q(s_t,a_t)$, while the second experience enables us to get $Q(s_{t+1},a_{t+1})$. These are sufficient to train the network as described in section 4.4.10.3.

4.5 Parameter Settings

When learning using reinforcement learning, several learning parameters need to have their values set to optimal values in order for the learning to occur successfully. Most researchers perform test experiments to decide on suitable values that should be assigned to these parameters. For instance in deciding the value of epsilon(ε), Sutton & Barto(1998) compared values of 0.01 and 0.1, and found that a value of 0.1 gives a better performance. The choice of the values for these parameters is described next.

4.5.1 *Setting the gain*

In this section the gain parameter refers to one of the neural network parameters that is found in the activation function as discussed in section 4.4.3. The gain value was set in a trial and error fashion. In the initial testing of the control policy neural network, the gain was set to one. It was observed that most of the outputs were very close to one. Table 4.2 shown below can be used to explain the reason for this behaviour. It can be seen than for any value of x greater than 3, a value of 1 is returned. Since the input to the function is the sum of weighted inputs to a node, if the sum has a value of 3 and above, a value of 1 is returned. This means that the network would not be able to differentiate between states that result in weighted sums of 3 and above.

Table 4.2: The behaviour of the bipolar sigmoid function

x	0.1	0.4	0.6	0.8	1	2	3	4	5	10	15	20
tanh(x)	0.1	0.38	0.54	0.66	0.76	0.96	1	1	1	1	1	1

The gain can be used to alleviate this problem. For instance if we have a gain of 0.2, the following result is obtained for the inputs in Table 4.2.

Table 4.3: Application of the gain parameter to inputs of the bipolar sigmoid function

x	0.1	0.4	0.6	0.8	1	2	3	4	5	10	15	20
0.2 * x	0.02	0.08	0.12	0.16	0.2	0.4	0.6	0.8	1	2	3	4
tanh(x)	0.02	0.08	0.12	0.16	0.2	0.38	0.54	0.66	0.76	0.96	1	1

Table 4.3 shows that the outputs span a greater range hence the network can be able to differentiate the different inputs. We experimented with a gain value of 0.2 and it was found to produce acceptable results. This value was therefore adopted for this study. The lesson learnt here was that in setting the gain, it is necessary to approximate the expected maximum sum of inputs to a node. The gain should be a value that scales the maximum sum to be approximately equal to 3.

4.5.2 *Setting the discount rate*

In the SARSA(λ), the discount rate (γ) determines how much future rewards contribute to the value of the state encountered at time t *(s_t)*. The discount rate value ranges from zero to one. If γ is zero, then, only the immediate reward contributes to the expected value of $V^{\pi}(s_t)$ or $V^{\pi}(s_t,a_t)$ for the current policy π. If the discount rate is one, then both immediate and future rewards have equal weights in their contribution to the value of the current state using the current policy π. For values between zero and one, the contribution of future rewards to the value of the current state is weighted. The closer γ is to zero, the lesser the weight (importance) associated with future rewards.

In our experiments, the agent got a reward of zero for all states and actions except for the actions that led to a terminal state. At that point, the agent got a reward of -1 if the terminal state was not a goal state and 1 if it was a goal state. This meant that future rewards were useful as the reward obtained at the end is the only useful feedback that was available to the agent.

The discount rate was determined by experimentally trying out several possible values in order to determine the most appropriate setting. These were: 0.2, 0.4, 0.8, 0.8, 0.9 and 1. To determine the performance when a given discount rate value is used, the control agent was used to drive the robot using action index 0 and 1 in Table 3.2 from a distance of 2 meters from the goal until it reached the goal region (one meter from the goal) while avoiding obstacles. One such attempt is called an episode. The way in which an episode ended (robot at goal or collides with an obstacle or time barred) was recorded in a log file. One trial consisted of 1000 such episodes. One starting position was used for all episodes although the direction in which the robot was facing was random. The following graph shows the results obtained for an average of five trials for each value. The success rate refers to the percentage of episodes that ended at the goal region.

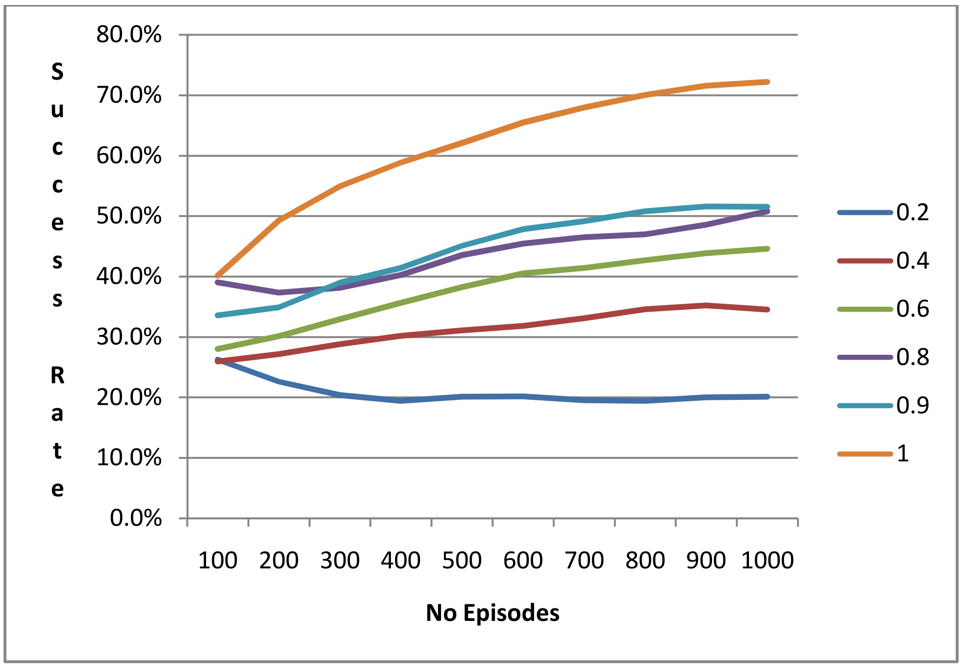


Figure 4.16: Success rate for different values of discount rate.

Given the results obtained a discount rate of 1 was selected for use in our experiments. Fernández et al. (2010) also used a discount rate value of 1 in their experiments in policy reuse and reported that this give accurate results. We however went further and compared different settings and we can conclude that a discount rate of 1 is indeed reasonable for an episodic task where the rewards are only obtained at the end of the episode.

4.5.3 *Setting the trace decay parameter*

In the SARSA(λ) algorithm, the λ stands for the *trace decay* parameter which is used in eligibility traces. Eligibility traces are used to perform updates on previously encountered states when an error is encountered during an update on the value of the current state. Equation 2.11 (repeated below) is used to perform updates in the SARSA(λ) algorithm, the difference between the underlined values is the TD error , where the first underlined value is the desired state-action value estimate, and the second underlined value is the current state-action value estimate.

$$Q^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) + \alpha[\underline{r_{t+1} + \gamma Q^{\pi}(s_{t+1}, a_{t+1})} - \underline{Q^{\pi}(s_t, a_t)}]$$

Eligibility traces means that this error is used to adjust the values of state action pairs encountered earlier in the episode i.e. s_{t-1}, s_{t-2} and so on. A value of **zero** for λ means that previous a state action values are not adjusted while a value of **one** means that previous state action pairs are adjusted as much as the current state action pair. The adjustment for previous state action pairs gets smaller as λ gets closer to zero and also with the temporal distance of the previous state action pair to the current state action pair.

We tested the performance of 6 different trace decay rate values. These are: 0.2, 0.4, 0.6, 0.8, 0.9 and 1. To determine the performance when a given trace decay value was used, the control agent was used to drive the robot using action index 0 and 1 (Table 3.2) from a distance of 2 meters from the goal until it reached the goal region (one meter from the goal) while avoiding obstacles. The way in which an episode ended (robot at goal or collision with an obstacle or time barred) was recorded in a log file. One trial consisted of 1000 episodes. One starting position is used for all episodes although the direction in which the robot was facing was random. Figure 4.17 shows the results obtained for an average of five trials for each value.

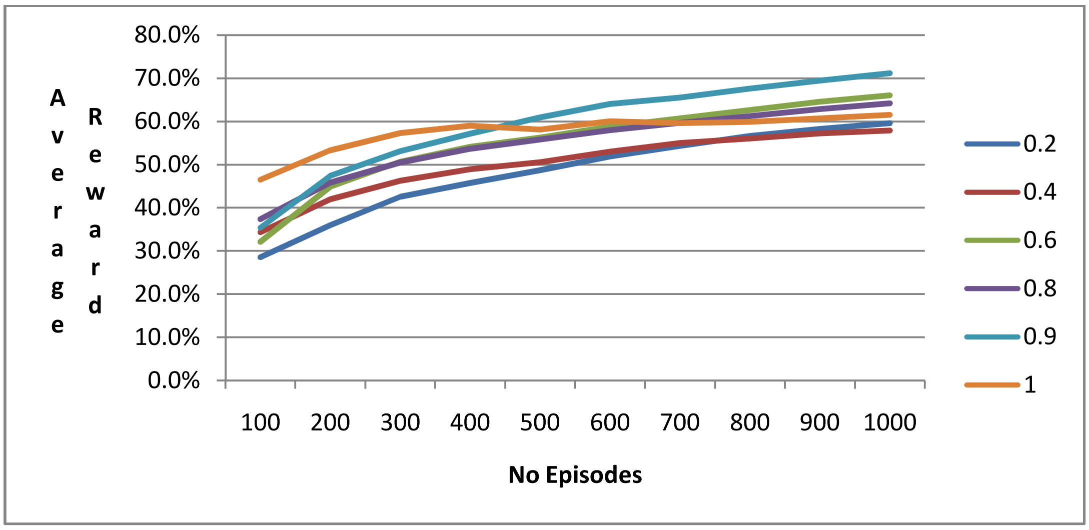


Figure 4.17: Success rate for different values of trace decay rate.

The results seem to indicate that the choice of λ does not have a very significant effect on the overall outcome. However, high values of λ seem to lead to better initial performance. This is expected because it leads to greater propagation of the error to states visited earlier in the episode. As the state action value becomes accurate, propagating the error value exactly as it is to states and actions experienced earlier does not lead to improvement in performance. This may be because these states and actions may not be so much responsible for the error anymore. Tesauro (1992) recommends setting λ to a high value during initial learning and reducing it to a smaller value when the predictions become more accurate. Given the results obtained, a trace decay of 0.9 was selected for use in our experiments since it is high enough to use in the initial learning and also asymptotically resulted in the best performance of the values tested.

4.5.4 *Setting the learning rate*

In Equation 2.11, the learning rate is represented by the α parameter. It determines the rate at which the current state action value is adjusted given the error. If the learning rate is zero, no learning takes place (Rajasekaran & Vijayalakshmi, 2003). Larger values of α will cause the state action value to oscillate rather settling at the correct value. For Temporal difference algorithms to converge, then α ought to be sufficiently small. The value of α can also be set to a high value then reduced gradually with time.

We tested the performance of 6 different learning rate values. These are 0.01, 0.05, 0.1, 0.15 0.2 and 0.25. To determine the performance when a given learning rate value was used, the control agent was used to drive the robot using action index 0 and 1 (Table 3.2) from a distance of 2 meters from the goal until it reached the goal region (one meter from the goal) while avoiding obstacles. The way in which an episode ended (robot at goal or collision with an obstacle or time barred) was recorded in a log file. One trial consisted of 4100 episodes. One starting position was used for all episodes although the direction in which the robot was facing was random. Figure 4.18 below shows the results obtained for an average of five trials for each value.

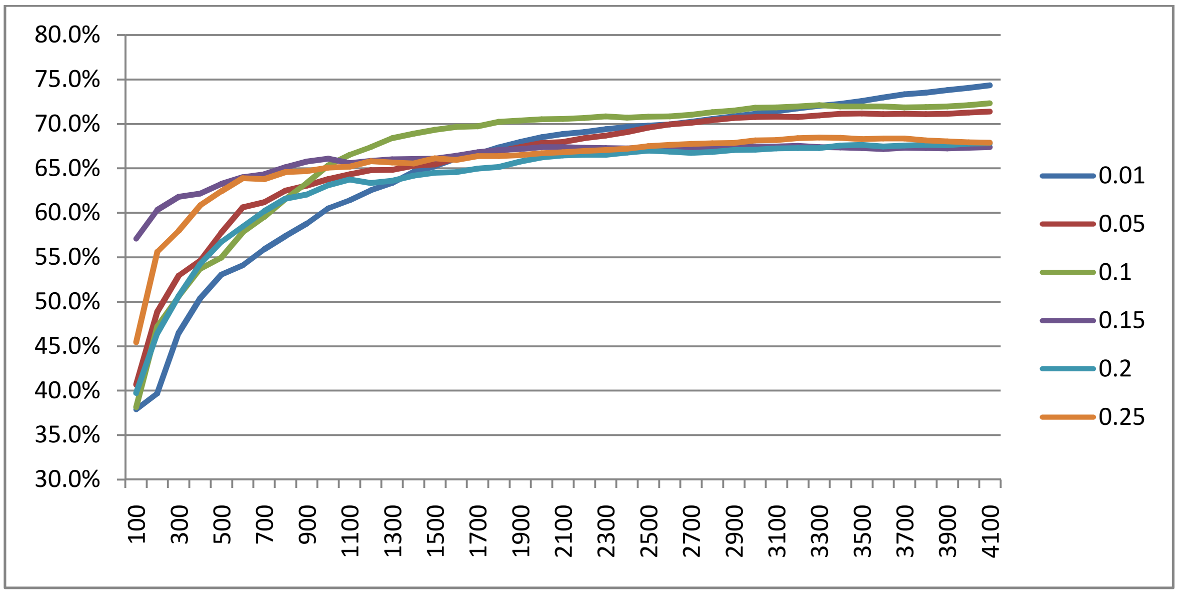


Figure 4.18: Performance for different settings of the learning rate.

The results show that higher values of the learning rate result in better performance in the early episodes, but lower settings result to better performance as the number of episodes increases, with the lowest setting of 0.01 giving the best performance asymptotically. We therefore settled on an initial learning rate of 0.15 which reduces by 0.001 after every ten episodes but stops reducing when the value reaches 0.01.

4.5.5 *Setting epsilon*

The epsilon (ε) parameter is used in ε-greedy algorithms to determine the probability of selecting a random action. High value of ε lead to greater exploration hence the agent can learn the values of more state-action pairs. The trend is to set a high value for ε in the beginning which reduces with time.

We tested 6 initial values of ε that is 0.1, 0.2, 0.3, 0.4, 0.5, and 0.6. The value for ε reduced by 0.005 after every 10 episodes until a minimum value of 0.1 was obtained. To determine the performance when a given epsilon value was used, the control agent was used to drive the robot using action index 0 and 1(Table 3.2) from a distance of 2 meters from the goal until it reaches the goal region (one meter from the goal) while avoiding obstacles. The way in which an episode ended (robot at goal or collision with an obstacle or time barred) was recorded in a log file. One trial consisted of 2500 episodes. One starting position was used for all episodes although the direction in which the robot was facing was random. Figure 4.19 below shows the results obtained for an average of five trials for each value.

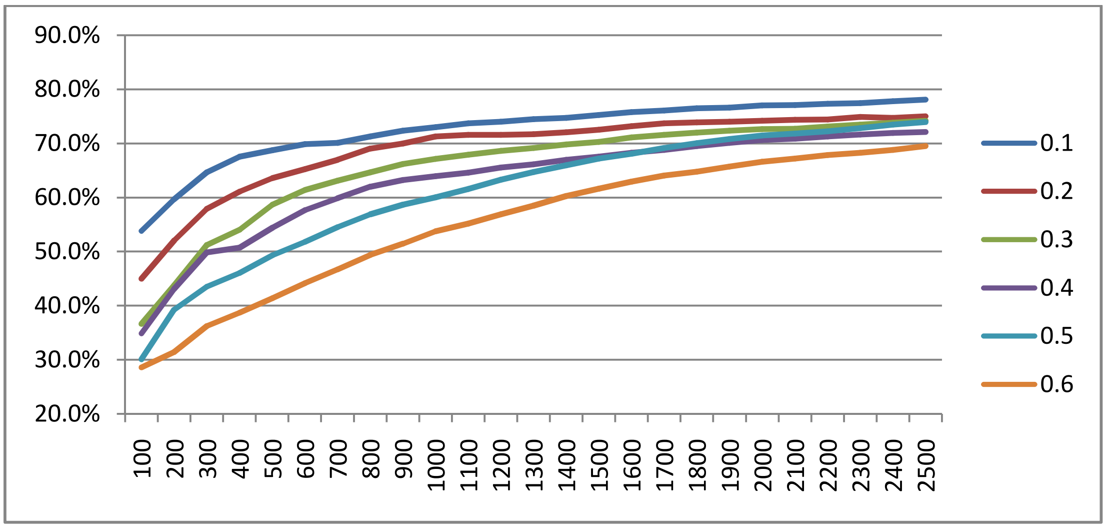


Figure 4.19: Comparing different value of epsilon(ε).

The results indicate that lower values of ε which have less exploration give the best performance. This is possibly because the number of actions is small and the agent learns the value of each of the actions early in the learning process. We therefore settled on a constant value of 0.1 for our experiments.

4.5.6 *Setting the number of episodes replayed*

When experience replay is used, it is necessary to determine how many episodes are replayed (*n*) in every replay session. This means that when we reach a point at which we have to replay past episodes corresponding to past experiences, *n* episodes are randomly picked from the list of stored past episodes and replayed.

The performance for six different settings for n was tested. These were 10, 20, 40, 60, 80 and 100. When a given value for *n* was used, the control agent was used to drive the robot (using action index 0 and 1) from a distance of 2 meters from the goal until it reached the goal region (one meter from the goal) while avoiding obstacles. The way in which an episode ended (robot at goal or collision with an obstacle or time barred) was recorded in a log file. One trial consisted of 1500 episodes. One starting position was used for all episodes although the direction in which the robot was facing was random. Figure 4.20 below shows the results obtained for an average of five trials for each value.

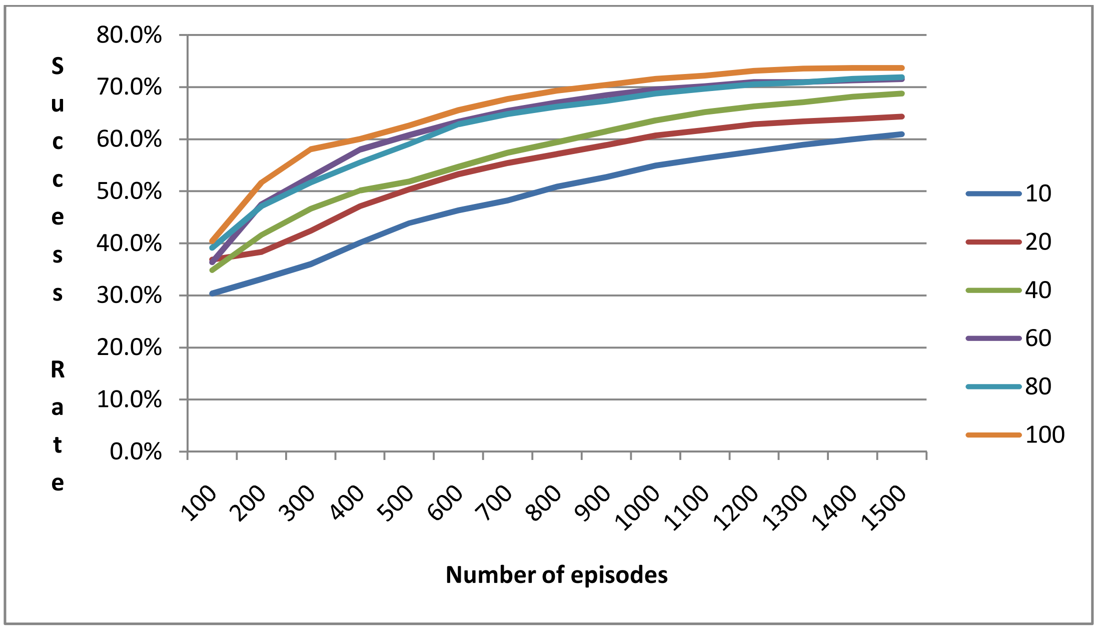


Figure 4.20: Results of different values for number of past episodes replayed

From the graph, we can see that the more the number of past episodes replayed, the better the performance. If the time taken to play episodes is not a major issue in the experiment, then we should prefer replaying more of the stored episodes. In our case, we set the number of replayed past episodes to 100.

4.5.7 *Setting the experience replacement strategy in experience replay*

Experience replay requires that past experiences are stored. These are later represented to the learning mechanism at frequent intervals hence ensuring that past experience is reused several times for training resulting in a speedup of learning. Since only a limited number of past experiences can be stored, once the maximum possible number of past experiences have been stored, some technique is needed to select a stored experience to be replaced by a new experience that needs to be stored. Lin (1992) suggests the use of replace oldest strategy. We compared the replace oldest strategy with the random replace strategy. The random replace strategy may lead to better performance since it ensures that the stored experience span a wider time interval rather than just the most recent experiences. Thus there is a wider variety of experiences among the stored experiences.

To determine which strategy to use in our experiments, we compared the performance of the replace oldest and the replace random strategies. Five trials were run for each of the strategies. Each trial consisted of 2000 episodes such that in each episode, the control agent was used to drive the robot (using action index 0 and 1) from a distance of 2 meters from the goal until it reached the goal region (one meter from the goal) while avoiding obstacles. The way in which an episode ended (robot at goal, collides with an obstacle, time barred) was recorded in a log file. One starting position was used for all episodes although the direction in which the robot was facing was random. Figure 4.21 below shows the results obtained.

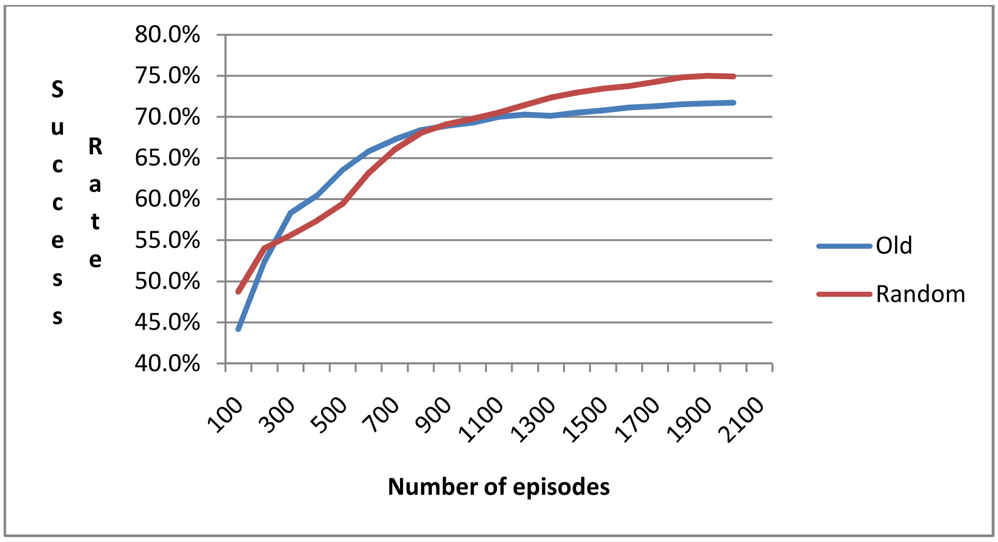


Figure 4.21: Comparison of experience replacement strategies

It can be seen that at the end of 2000 episodes, the replace random strategy outperforms the replace oldest experience strategy. The replace oldest strategy success rate is at 71.7% while the replace random success rate is at 74.9%. We therefore chose to use the random replace strategy in this study.

4.5.8 *Policy Selection in policy reuse*

In policy reuse, the control agent has to decide which among the new *target policy* and the *source policy* to use in an episode. Where the source policy is the *final policy* of the *source task* and the *target policy* is created for the *target task*. Fernández et al. (2010) proposes the use of a technique based on the Boltzmann distribution to decide if to use the *source policy* or the *target policy*.

Assuming that the probability of using the source policy is P_s and the Probability of using the target policy is P_t then these probabilities are calculated as shown below.

Let

S_c – be a count of the number of times the source policy has been used

S_g – Be a count of the number of episodes that have ended in the goal region while under the control of the source policy

T_c – be a count of the number of times the target policy has been used

T_g – Be a count of the number of episodes that have ended in the goal region while under the control of the target policy

The success rate of the source policy S_r is given by $S_r=S_g/S_c$

The success rate of the target policy T_r is given by $T_r=T_g/T_c$

The probability of using the source policy (Ps) is given by

$$p_s = \frac{e^{tS_r}}{e^{tS_r} + e^{tT_r}} \qquad \textit{Equation 4.29}$$

And the probability of using the target policy is given by

$$p_t = \frac{e^{tT_r}}{e^{tS_r} + e^{tT_r}} \qquad \textit{Equation 4.30}$$

Or

$$P_t = 1 - P_s \qquad \textit{Equation 4.31}$$

We tested the use of this strategy in reusing a final 2 actions policy in a 3 actions task. In test experiments with a temperature value of 2, use of this equation showed that the control agent tended to prefer the source policy over the target policy since the source policy was already capable of guiding the robot to the goal. The target policy was still random hence its performance was initially bad leading to its being seldom selected. We therefore proposed a new policy selection strategy explained below.

Let $normalised_s$ and $normalised_t$ be the normalised performances of the source and target policies calculated using Equation 4.32 and Equation 4.33 where S_r and T_r are the success rates of the source policy and the target policies respectively.

$$normalised_s = \frac{S_r}{S_r + T_r} \qquad \textit{Equation 4.32}$$

$$normalised_t = \frac{T_r}{S_r + T_r} \qquad \textit{Equation 4.33}$$

The probabilities P_s and P_t of using the source policy and the target policy are calculated using Equation 4.35 and Equation 4.34 as shown below

$$P_s = normalised_s(1 - normalised_t)$$ Equation 4.35

$$P_t = 1 - P_s$$ Equation 4.34

Equation 4.32 and Equation 4.33 calculate an initial probability of using the source policy (*normalised_s*) and an initial probability of using the target policy (*normalised_t*) based on their success rates. The more successful policy will have a higher normalised probability. The effect of *Equation 4.35* is to reduce the probability of selecting the source policy based on how successful the target policy is. Whatever the value of the second term in *Equation 4.35*, P_s will be smaller than the initial value (*normalised_s*). However, the more successful the target policy, the smaller the value of the second term in *Equation 4.35* hence the more the probability of choosing the source policy decreases.

This method introduces a way to discourage use of the source policy in a controlled way as the target policy improves. However, due to the first term in *Equation 4.35* the higher the success rate of the source policy, the higher its probability of being used. We call the new method *target policy preferred policy selection(tppps)*.

To determine which strategy to use in our experiments, we compared the performance of the Boltzmann distribution based method and the *tppps* strategy. The control agent was initially trained using two actions until a success rate of 75% was reached after 2500 episodes to obtain the source policy. Five trials were then run for each of the strategies. Each trial consisted of 2500 episodes such that in each episode, the control agent was used to drive the robot using action index 0, 1 and 2 (Table 3.2) from a distance of 2 meters from the goal until it reached the goal region (one meter from the goal) while avoiding obstacles. The way in which an episode ended (robot at goal, collides with an obstacle, time barred) was recorded in a log file. One starting position was used for all episodes although the direction in which the robot was facing was random. Figure 4.21 shows the results obtained.

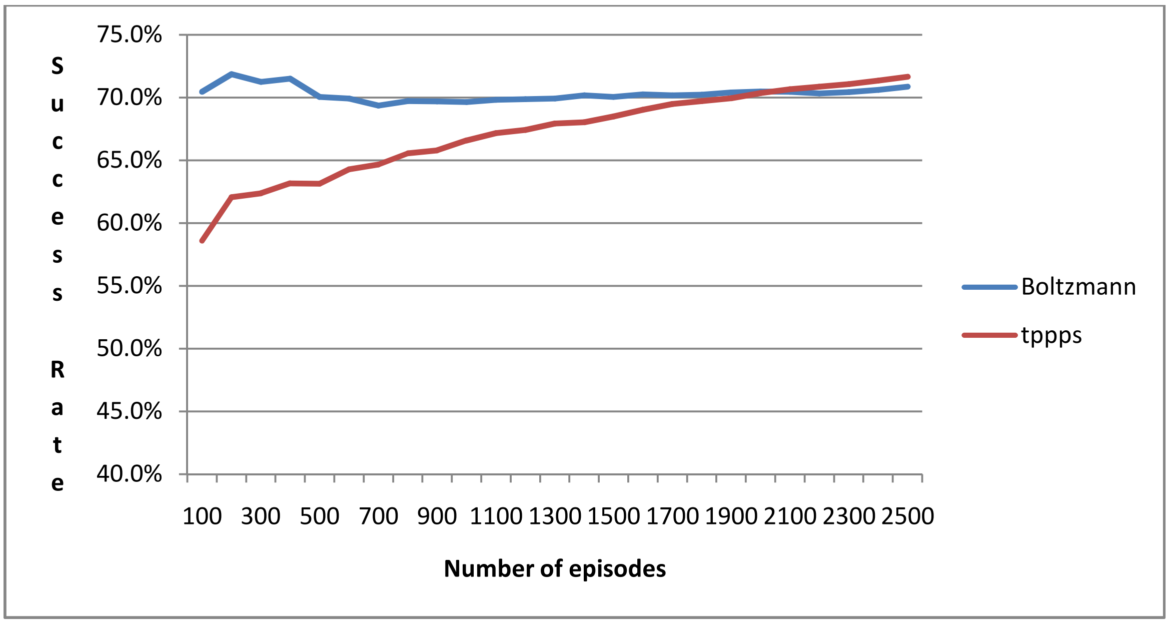


Figure 4.22: Combined performance of the source and target policies using the Boltzmann distribution based method and the new tppps method

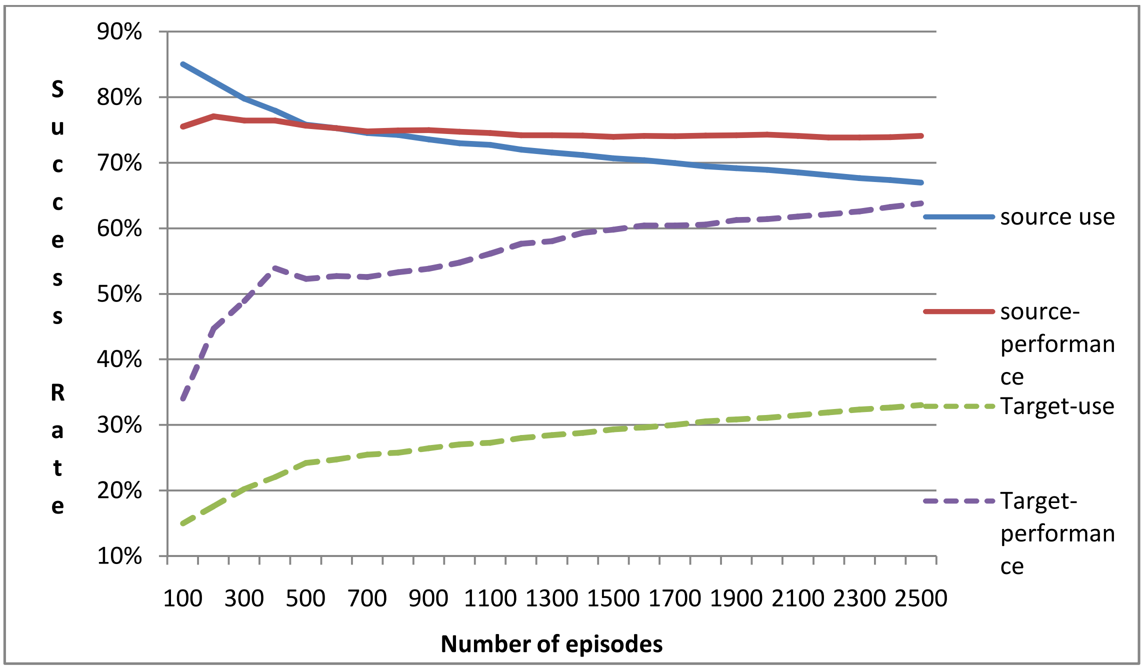


Figure 4.23: Performance of the Boltzmann distribution based method.

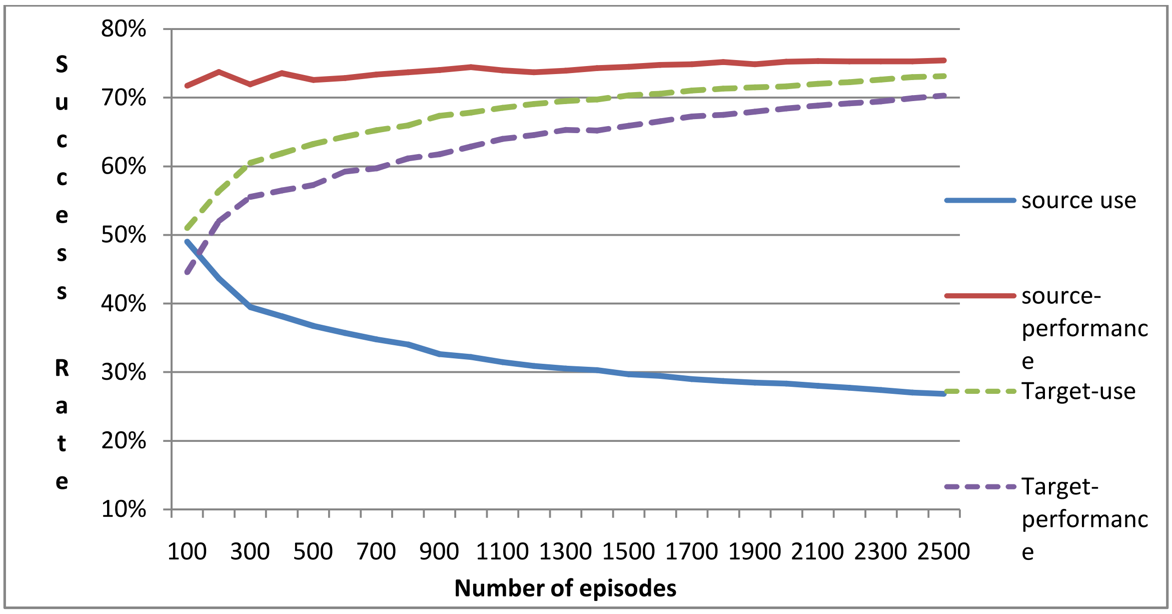


Figure 4.24: The performance of the tppps method.

From Figure 4.22 we can see that asymptotically the *tppps* outperforms the Boltzmann distribution based method. However, the initial performance of the Boltzmann distribution based method is better. This is because the source policy is used most of the time hence the performance approaches that of the source policy.

From Figure 4.23 which illustrates the performance of the source and target polices using the Boltzmann distribution based policy selection method, we can see that the performance of the source policy (about 75%) hardly changes through out the trial. This is expected because only the target policy is updated. The use rate of the target policy is low and increases to about 33% by episode number 2500. The performance of the target policy by the same episode is at 64% from an initial performance of about 34%.

From Figure 4.24 which illustrates the performance of the source and target polices using the *tppps* policy selection method, we can see that the performance of the source policy (about 75%) hardly changes through out the trial. This is expected because only the target policy is updated. The use rate of the target policy is high and increases to about 73% by episode number 2500 from an initial use rate of 51%. The performance of the target policy by the same episode is at 70% from an initial performance of about 45%.

Based on these results, we decided to use the *tppps* strategy for out study.

5. Results

In this chapter, the actual experiments performed and the results attained are detailed. The experiments are performed to test the performance of the algorithms listed in Table 2.1. These are the baseline learning algorithm, the algorithm resulting from a direct implementation of all the techniques suggested in the learning framework illustrated in Figure 2.10, and the alternative algorithms resulting from suggested modifications to the learning framework. The experiments were grouped into three main categories which comprise the subsections of this chapter. Each subsection contains a listing and a brief description of the all experiments performed and the algorithms used; followed by a comparison of the results attained using different algorithm. The comparisons try to establish if the difference in performance attained using different algorithms is significant.

Section 5.1 is titled *"learning performance in the two actions task"*. The experiments detailed in this section involve trying to learn the task of controlling a robot in which two actions are allowed to move from pre-specified start locations to a specific goal location. Only two experiments are performed for the two actions task.

Section 5.2 is titled *"learning performance in the three actions task"*. The experiments performed in this section involve trying to learn the task of controlling a robot in which three actions are allowed to move from pre-specified start locations to a specific goal location. Eight experiments are performed in this section in which the performances of all the algorithms listed in Table 2.1 are tested.

Section 5.3 is titled *"learning performance in the six actions task"*. The experiments performed in this section involve trying to learn the task of controlling a robot in which six actions are allowed to move from pre-specified start locations to a specific goal location. Eight experiments are performed in this section in which the performances of all the algorithms listed in Table 2.1 are tested.

5.1 Learning performance in the two actions task

In learning the two actions task, no other task had been learnt from which knowledge would have been transferred. Consequently, only the two algorithms that did not use knowledge transfer were used in learning this task. These were SARSA(λ) and SARSA(λ)-R.

5.1.1 *Experiments*

5.1.1.1.1 *Experiment 1: Learning to control a two actions robot using SARSA(λ)*

In this experiment, the control agent had two actions which it used to control the robot from the initial start position until the robot reached the goal region. The two actions were the move straight and turn left actions which are actions 0 and 1 in Table 3.2. Each trial in this experiment used the first six random numbers in Table 3.5.

5.1.1.1.2 *Experiment 2: Using SARSA(λ)-R to learning to control a two actions robot*

In this experiment, the control agent had two actions which it used to control the robot from the initial start position until it reached the goal region. The two actions were the move straight and turn left actions which are actions 0 and 1 in Table 3.2.

The maximum number of episodes stored for experience replay was 100. After every ten episodes, the policy neural network was trained using all of the stored episodes.

5.1.2 *Results*

The objective of the comparison was to find out the extent of improvement in learning the two actions tasks that resulted from use of experience replay. Figure 5.1 shows the average performance in the task using the two techniques.

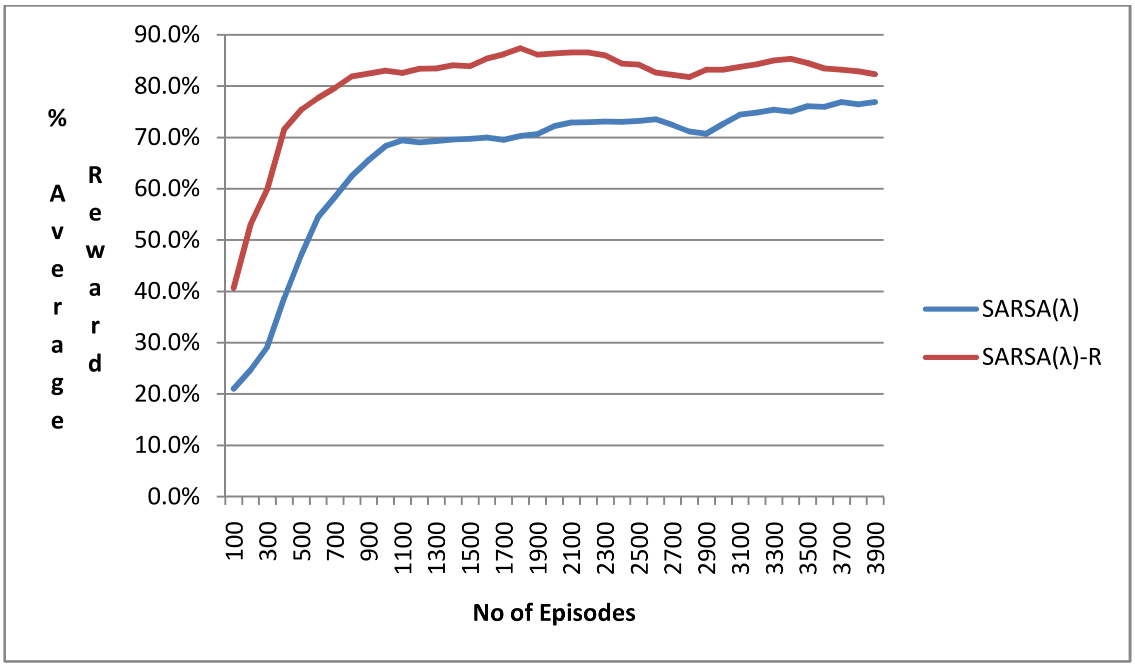


Figure 5.1: Performance of SARSA(λ)-R and SARSA(λ) in the two actions tasks

Table 5.1: Episode termination type for the algorithms used in the two actions task

		Algorithm	
Episode No	Episode Termination Type	SARSA(λ)	SARSA(λ)-R
100	Goal	21.0%	40.6%
	Time Barred	42.0%	22.5%
	Collision	37.0%	36.9%
3900	Goal	76.9%	82.3%
	Time Barred	3.2%	5.1%
	Collision	19.9%	12.6%

Table 5.1 shows the manner in which the learning episodes terminated both at the initial and at the asymptotic levels. From the table, we can see that while both algorithms did eventually learn to move towards the goal, they had a harder time learning how to avoid obstacles.

For the SARSA(λ)-R algorithm to be said to speedup the learning, it needs to have better performance than SARSA(λ) at either the initial level, or at the asymptotic level or both. We used significance test to determine if the performance of SARSA(λ)-R was significantly better than that of SARSA(λ). The tests were performed at both the initial and at the asymptotic levels.

a) Initial performance

Table 5.2 shows the performance of both algorithms after 100 episodes in the 14 paired scenarios.

Table 5.2: Performances of SARSA(λ)-R and SARSA(λ) after 100 episodes of learning in the two actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-R	8	56	46	46	24	72	46	54	50	33	49	18	38	29
SARSA(λ)	2	10	32	34	16	43	7	10	6	36	42	26	30	0

Using Shapiro-Wilk test, the value of the W statistic obtained was 0.91 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. We therefore used the paired Student's t-test to compare the performance of the two techniques.

At a significance level of 0.05, and 13 degrees of freedom, $t_{critical} = 1.771$, and $t_{calc} = 3.99$ which is greater than $t_{critical}$. We therefore concluded that the initial performance of SARSA(λ)-R in the two actions task was significantly greater than that of SARSA(λ).

To illustrate the difference in performance in terms of simulation time we observe from Figure 5.1 that the initial performance of SARSA(λ)-R was 40.6%. This was attained after 100 episodes corresponding to 0.56hrs. SARSA(λ) took about 400 episodes to arrive at the same performance. This is approximately 2.2hrs of simulation time. We therefore conclude that to arrive at a performance of about 40%, SARSA(λ) took 1.7hrs more than SARSA(λ)-R.

b) Asymptotic performance

Table 5.3 shows the performance of both algorithms after 3900 episodes in the 14 paired scenarios.

Table 5.3: Performances of SARSA(λ)-R and SARSA(λ) after 3900 episodes of learning in the two actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-R	71	84	88	85	82	80.3	83.3	89.7	68.3	77.7	87	83.7	85.7	86
SARSA(λ)	76.7	75	80	75	74	74.7	85.3	75.7	74	68.7	82.3	74.7	79.3	80.3

Using Shapiro-Wilk test, the value of the W statistic obtained was 0.86 which is less than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were not normally distributed. We therefore used the Wilcoxon Signed Rank test to determine if the median asymptotic performance of SARSA(λ)-R was greater than the median asymptotic performance of SARSA(λ) in the two actions task.

Using the Wilcoxon Signed Rank, the value obtained for W was 11 with n'=14. The sum of positive ranks was 94 whereas the sum of negative ranks was 11. For a one tailed Wilcoxon Signed Rank test at a significance level of 0.05, and n'=14, $W_{critical}=25$. Since W is less than $W_{critical}$, we concluded that the median asymptotic performance of SARSA(λ)-R was greater than the median asymptotic performance of SARSA(λ) in the two actions task.

To illustrate the difference in performance in terms of simulation time we observe from Figure 5.1 that the average asymptotic performance of SARSA(λ) was 76.9%. This was attained after 3900 episodes corresponding to 21.7hrs. SARSA(λ)-R took about 600 episodes to exceed this performance. This is approximately 3.3hrs of simulation time. We therefore conclude that to arrive at a performance of about 76.9%, SARSA(λ) took 18.3hrs more than SARSA(λ)-R.

5.2 Learning performance in the three actions task

5.2.1 *Experiments*

All the algorithms in Table 2.1 were used in learning the three actions task. Six of the algorithms used knowledge transfer. Two algorithms, that is SARSA(λ) and SARSA(λ)-R did not use knowledge transfer. Actions 0 ,1 and 2 in Table 3.2 were used by the control agent to control the robot.

For the algorithms that used knowledge transfer, the control agent was first trained in the two actions task using SARSA(λ)-R until a performance of 89% was attained after 900 episodes. The final policy attained in the two actions source task was used as the source policy. It was used to initialise the 3 actions task policy to attain policy transfer. To implement policy reuse, the source policy was used to select actions in some of the episodes of the 3 actions task. The following experiments were performed:

Experiment 3: Using SARSA(λ) to learn to control a three actions robot

Experiment 4: Using SARSA(λ)-R to learn to control a three actions robot

Experiment 5: Using SARSA(λ)-PT to learn to control a three actions robot

This algorithm required initialisation of the three actions task policy with the two actions task source policy. Table 5.4 shows how the three actions task actions were mapped into the two actions task actions.

Table 5.4: Mapping action 2 to action 1 in transfer of the two actions policy to the three actions policy

Target Action	Source Action
0	0
1	1
2	1

When action *i* in the target task was mapped to action *j* in the source task, the weights for action *j* were used to initialise the weights for action *i* in the target task policy.

Experiment 6: Using SARSA(λ)-PR to learn to control a three actions robot

In this experiment, the target task policy (neural network) was initialised randomly but it was updated based on the actions selected by either the source task policy or the target task policy.

Experiment 7: Using SARSA(λ)-PT-PR to learn to control a three actions robot

The policy transfer was achieved by using a final two actions policy to initialise the policy used in performing the three actions task. Policy reuse on the other hand was achieved by probabilistically using the source policy to select actions in some episodes of the target task. Table 5.5 shows how the weights were mapped for policy transfer.

Table 5.5: Mapping action 2 to action 1 in policy transfer and policy reuse from the two actions task to the three actions task

Target Action	Source Action
0	0
1	1
2	1

Experiment 8: Using SARSA(λ)-R-PT to learn to control a three actions robot

The policy transfer was achieved by using a final two actions policy to initialise the policy used in performing the three actions task. Table 5.6 shows how the weights were mapped with the new action mapped to action 1 in the source task.

Table 5.6: Initialising action 2 with action 1 in policy transfer from 2 actions to 3 actions task with experience replay

Target Action	Source Action
0	0
1	1
2	1

Experiment 9: Using SARSA(λ)-R-PR to learning to control a three actions robot

Policy reuse was achieved by probabilistically using a final two actions policy to select actions in some episodes during execution of the task.

Experiment 10: Using SARSA(λ)-R-PR-PT to learning to control a three actions robot

Policy reuse was achieved by probabilistically using a final two actions policy to select actions in some episodes during execution of the task. Policy transfer was used to initialise the target task policy. Table 5.7 shows the inter-task action mapping that was used.

Table 5.7: Mapping action 2 with action 1 in combining policy transfer, policy reuse and experience replay in the three actions task

Target Action	Source Action
0	0
1	1
2	1

5.2.2 *Results of learning the three actions task using the proposed, baseline and alternative algorithms*

Figure 5.2 shows the performance of all the learning algorithms evaluated in the three actions learning task. From the graph, we notice that SARSA(λ)-PT which uses policy transfer and SARSA(λ)-PT-PR which use both policy transfer and policy reuse result in the best performance. In learning without knowledge transfer, SARSA(λ)-R which uses experience replay performs better than SARSA(λ) the baseline learning algorithm. Therefore if the three actions task is to act as the initial source task, SARSA(λ)-R should be used in learning it.

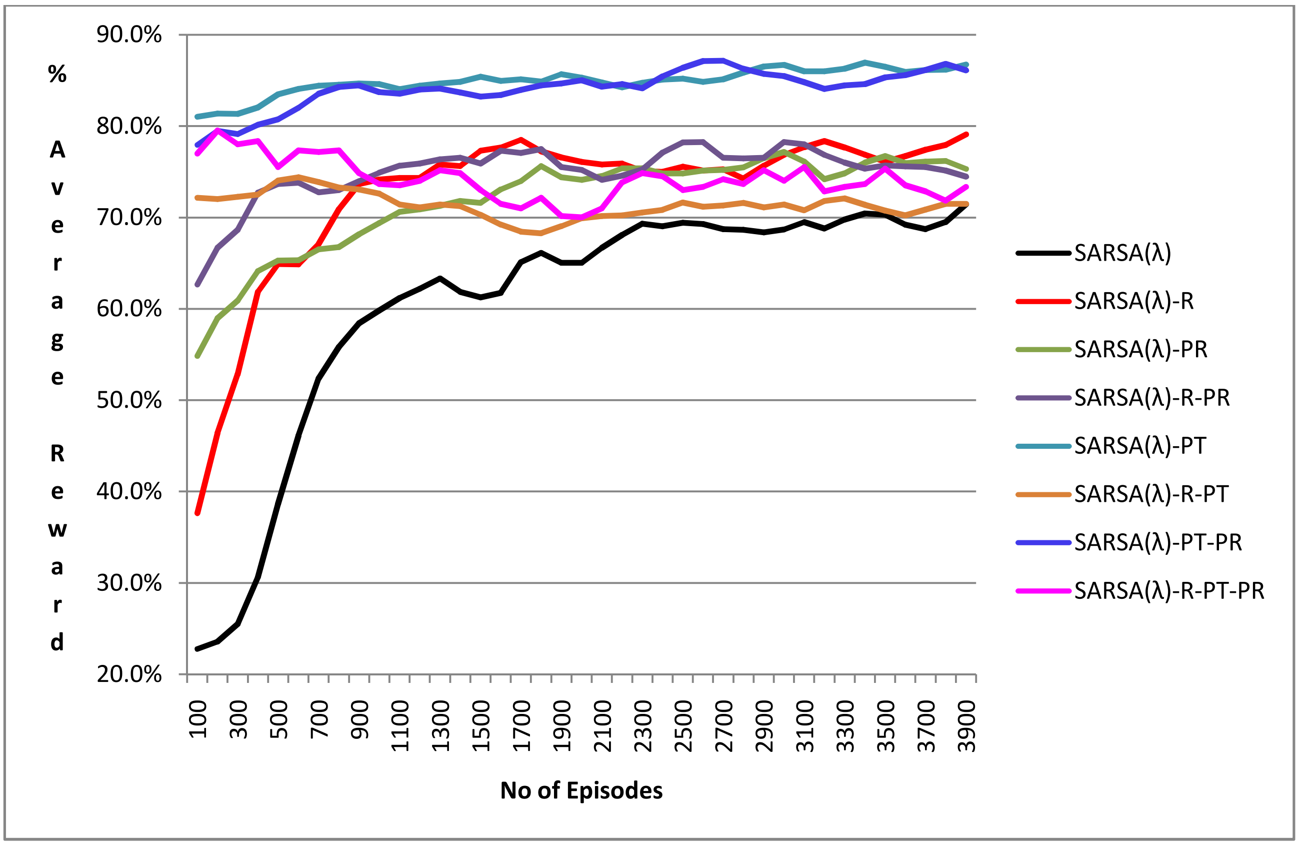


Figure 5.2: Performance of all the learning algorithms in the three actions task

Combining the three techniques: policy transfer, policy reuse and experience replay (SARSA(λ)-R-PT-PR) does not result in good performance as expected. The initial performance is high but then the performance starts to decline after the first few episodes. The most likely explanation for this is overfitting which occurs if a neural network is trained too much with the same examples. Combining policy transfer with experience replay seems to result in similar behaviour. However when policy reuse is combined with policy experience replay SARSA(λ)-R-PR, there seems to be some benefit in the initial performance compared to when SARSA(λ)-PR is used alone. This may be attributed to the initial low performance of SARSA(λ)-PR.

Table 5.8 shows the episode termination type for episodes in the three actions task. Just as in the two actions task, the learning algorithms eventually learnt how to move towards the goal but had a harder time learning how to avoid obstacles.

Table 5.8: Episode termination type for the algorithms used in the three actions task

		Algorithm				
Episode No	Episode Termination Type	SARSA(λ)	SARSA(λ)-R	SARSA(λ)-PR	SARSA(λ)-PT	SARSA(λ)-PT-PR
100	Goal	23.0%	37.8%	54.9%	81.0%	78.0%
	Time Barred	33.9%	23.5%	19.9%	4.8%	6.5%
	Collision	43.1%	38.7%	25.2%	14.2%	15.5%
3900	Goal	71.5%	79.1%	75.3%	86.7%	86.1%
	Time Barred	5.1%	5.7%	6.5%	1.6%	2.8%
	Collision	23.4%	15.2%	18.2%	11.7%	11.1%

5.2.3 *Comparing Individual Learning Algorithms in learning the three actions task*

The individual algorithms were compared to find out if the differences in performance were significant.

5.2.3.1 Comparing performance of SARSA(λ) with the performance of SARSA(λ)-R in learning the three actions task

SARSA(λ) is the baseline learning algorithm. If the three actions task is the initial source task, only the two algorithms SARSA(λ) and SARSA(λ)-R can be used for learning. It is therefore important to find out which of them leads to faster learning.

From Figure 5.2, we can see that experience replay resulted in improved learning performance in the 3 actions task. We performed significance tests on the initial and asymptotic performances of the two algorithms to determine if the performance of SARSA(λ)-R was significantly greater than that of SARSA(λ).

a) Initial performance

Table 5.9 shows the performance of both algorithms after 100 episodes in 14 paired scenarios.

Table 5.9: Performances of SARSA(λ)-R and SARSA(λ) after 100 episodes of learning in the three actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-R	20	41	42	26	52	54	27	38	56	40	32	35	36	30
SARSA(λ)	22	1	21	42	39	30	21	18	27	11	14	33	25	18

Using Shapiro-Wilk test, the value of the W statistic obtained was 0.98 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. We therefore used the paired Student's t-test to compare the performance of the two techniques.

At a significance level of 0.05, and 13 degrees of freedom, $t_{critical}$ =*1.771*, and t_{calc}= 3.84 which is greater than $t_{critical.}$. We therefore concluded that the initial performance of SARSA(λ)-R in the three actions task was significantly greater than that of SARSA(λ).

To illustrate the difference in performance in terms of simulation time we observe from Figure 5.2 that the initial performance of SARSA(λ)-R was 37.6%. This was attained after 100 episodes corresponding to 0.56hrs. SARSA(λ) took about 500 episodes to attain this performance. This is approximately 2.7hrs of simulation time. We therefore conclude that to arrive at a performance of about 37.6%, SARSA(λ) took 2.2hrs more than SARSA(λ)-R.

a) Asymptotic performance

Table 5.10 shows the performance of both algorithms after 3900 episodes in 14 paired scenarios.

Table 5.10: Performances of SARSA(λ)-R and SARSA(λ) after 3900 episodes of learning in the three actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-R	85.7	79	94.3	83.7	86.3	75.7	69.7	86.3	78.3	78.7	62.7	54.7	80	92.3
SARSA(λ)	79	74.7	71.3	64.7	78	67.3	70.7	76.7	70.7	67.7	66	61.3	75.7	76.7

Using Shapiro-Wilk test, the value of the W statistic obtained was 0.97 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. We therefore used the paired Student's t-test to compare the performance of the two techniques.

At a significance level of 0.05, and 13 degrees of freedom, $t_{critical}$ =*1.771*, and t_{calc}= 3.5 which is greater than $t_{critical.}$ We therefore concluded that the asymptotic performance of SARSA(λ)-R in the three actions task was significantly greater than that of SARSA(λ).

To illustrate the difference in performance in terms of simulation time we observe that the best average performance obtained using SARSA(λ) was 71.5%. This was obtained after 3900 episodes, which translates to 21.7hrs. It took 900 episodes for SARSA(λ)-R to get to the same performance, which translates to 5hrs. Therefore on average, it took SARSA(λ) 16.7hrs more than SARSA(λ)-R to get to a performance of 71.5% in the three actions task.

5.2.3.2 Comparing the learning Performance of SARSA(λ)-PT-PR with that of SARSA(λ)-PT in the 3 actions tasks

In Figure 5.2, these algorithms were seen to result in the best performance in the three actions task. We compared the performance of the two algorithms in order to determine which of the two algorithms should be preferred in learning the three actions target task.

a) Initial performance

Table 5.11 shows the initial performances attained in both techniques. The initial performance is the percentage of episodes that ended in the goal region in the first 100 episodes.

Table 5.11: Initial performances of SARSA(λ)-PT-PR, and of SARSA(λ)-PT in the three actions task

Scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT-PR	63	75	80	75	72	80	72	84	85	80	78	80	79	89
SARSA(λ)-PT	89	61	80	83	80	87	81	81	78	88	78	80	83	85

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.93 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to determine if the initial performance of SARSA(λ)-PT-PR was greater than the initial learning performance using SARSA(λ)-PT in the three actions task.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}=1.771$, $t_{calc}= -1.19$ whose magnitude is less than the critical value. We therefore concluded that the performances of SARSA(λ)-PT and SARSA(λ)-PT-PR in the three actions task were not significantly different.

b) Asymptotic Performance

Table 5.12 shows the asymptotic performances achieved in both techniques after 3900 episodes. This is the percentage of episodes that had ended in the goal region within the last 300 episodes (INTERVAL_SIZE was set to 300; see section 3.4.3.2).

Table 5.12: Asymptotic performances of SARSA(λ)-PT-PR, and of SARSA(λ)-PT in the three actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT-PR	70.7	71	90.7	84	93.3	81.7	86	90.3	90	92.7	89.7	91.7	84.7	89
SARSA(λ)-PT	88.7	57.3	89	91	90	87.7	89	86	85.3	90.7	93	88	91	87.7

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.94 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to determine if the asymptotic performance using SARSA(λ)-PT-PR was greater than the asymptotic performance using SARSA(λ)-PT in learning the three actions task.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}=1.771$, $t_{calc}= 0.32$ which is less than the critical value. We therefore concluded that the difference in performance between the two techniques was not significantly different.

5.2.3.3 Comparing Learning Performance of SARSA(λ)-PT with learning performance of SARSA(λ)-PR in the three actions task

Policy reuse (SARSA(λ)-PR) and policy transfer (SARSA(λ)-PT) are two alternative techniques for reusing knowledge from the source task in the target ask. Figure 5.2 shows that SARSA(λ)-PT which uses policy transfer outperforms SARSA(λ)-PR which uses policy reuse. We want to find out if this difference in performance is significant.

We compared the performance of SARSA(λ)-PR with the performance of SARSA(λ)-PT at both the initial and asymptotic levels to determine if the differences in performance were significant.

a) Initial performance

Table 5.13 shows the initial performances of both SARSA(λ)-PT and SARSA(λ)-PR after 100 episodes.

Table 5.13: Initial performances of SARSA(λ)-PT and of SARSA(λ)-PR in the 3 actions task.

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT	89	61	80	83	80	87	81	81	78	88	78	80	83	85
SARSA(λ)-PR	58	66	55	41	50	47	64	55	59	54	59	57	40	63

Using the Shapiro-Wilk test, the value obtained for W statistic was 0.912 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to determine if the asymptotic performance SARSA(λ)-PT was greater than the asymptotic performance of SARSA(λ)-PR.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}=1.771$, $t_{calc}= 7.9$ which is greater than the critical value. We therefore concluded that the initial performance SARSA(λ)-PT was greater than the initial performance of SARSA(λ)-PR.

From Figure 5.2, we observe that the initial performance of SARSA(λ)-PT was 81%. This was reached in episode 100 which translates to 0.56hrs. In 3900 learning episodes (21.7hrs) SARSA(λ)-PR does not get to SARSA(λ)-PT's initial performance. We therefore conclude that to reach a performance of 81%, SARSA(λ)-PR took 21.1hrs more than SARSA(λ)-PT.

b) Asymptotic Performance

The asymptotic performances of both techniques are shown in Table 5.14. This is the performance that was attained after 3900 episodes of learning in the 14 scenarios tested. We did not have to consider the time spent in learning the source task since both techniques made use knowledge reuse.

Table 5.14: Asymptotic performances of SARSA(λ)-PT and of SARSA(λ)-PR in the 3 actions task.

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT	88.7	57.3	89	91	90	87.7	89	86	85.3	90.7	93	88	91	87.7
SARSA(λ)-PR	77.3	73.7	77.7	76	71.7	75.3	77.3	78	81	75.3	72	74.3	77	76.7

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.74 which is less than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were not normally distributed. The Wilcoxon Signed rank test was therefore used to determine if the asymptotic performance using SARSA(λ)-PT was greater than the asymptotic performance of SARSA(λ)-PR in the three actions task.

The calculated value for W was 12 with n′=14. The sum of positive ranks was 93 whereas the sum of negative ranks was 93. For a one tailed Wilcoxon Signed Rank test at a significance level of 0.05, and n′=14, $W_{critical}$=25. Since W is less than $W_{critical}$, we conclude the median asymptotic performance for SARSA(λ)-PT was greater than the median asymptotic performance for SARSA(λ)-PR in the three actions task.
From Figure 5.2 , we observe that the best performance of SARSA(λ)-PR was 77%. This was reached in episode 3000 which translates to 16.7hrs. It took 100 episodes for SARSA(λ)-PT to exceed this performance. This translates to 0.56hrs. Therefore, to reach a performance of 77%, SARSA(λ)-PR took 16.1hrs more than SARSA(λ)-PT.

5.2.3.4 Comparing SARSA(λ)-PT with SARSA(λ)-R

The comparison helped us to determine which of experience replay or policy transfer led to the greatest speedup in learning the three actions task. This comparison was necessary so as to determine which of the two algorithms should be used in learning the three actions target task.

a) Initial Performance

To compare the initial performances, we needed to consider the fact that 900 episodes were spent in learning the source task when SARSA(λ)-PT was used in the target task. We therefore took the performance of SARSA(λ)-PT after 100 episodes in the target task, and compared this with the performance of SARSA(λ)-R after 1000 episodes of learning in the target task. Table 5.15 shows the performance after the said number of episodes for the two techniques.

Table 5.15: Initial performances of SARSA(λ)-PT and of SARSA(λ)-R in the 3 actions task.

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT	89	61	80	83	80	87	81	81	78	88	78	80	83	85
SARSA(λ)-R	72.3	82.3	76.7	72	90	76	72.7	67.7	65	74.3	68	65	70.3	85.3

Using the Shapiro-Wilk test, the value obtained for the W static was 0.78 which is less than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were not normally distributed. The Wilcoxon Signed rank test was therefore used to determine if the initial performance using policy transfer from the two actions task to the three actions task was greater than the initial performance when the three actions task was learnt using experience replay.

Using the Wilcoxon Signed rank test, the value obtained for the W static was 19.5 with n′=14. The sum of positive ranks was 85.5 whereas the sum of negative ranks was 19.5. For a one tailed Wilcoxon Signed Rank test at a significance level of 0.05, and n′=14, $W_{critical}$=25. Since W is less than $W_{critical}$, we concluded the median initial performance for SARSA(λ)-PT was greater than the median initial performance for SARSA(λ)-R in the three actions task.

To further illustrate the difference in performance in terms of simulation time, we observe that the average initial performance of SARSA(λ)-PT was 81%. This was obtained by episode 100 (see Figure 5.2), which translated to 5.6hrs (when the 900 episodes spent in learning the source task were included). After 3900 episodes (21.7hrs), the average performance of SARSA(λ)-R was 79% which was still below the initial performance of SARSA(λ)-PT. Therefore we concluded that the difference between the average time it takes SARSA(λ)-PT to get to a performance of 81% and the average time it takes SARSA(λ)-R to get to the same performance in the three actions task was greater than 16.1hrs.

b) Asymptotic Performance

We got the asymptotic performance after 3000 episodes of learning in the transfer setting (because 900 episodes are used in learning the source task) and 3900 episodes in the replay setting since replay trials ended after 3900 episodes. The performances after the said number of episodes are shown in Table 5.16.

Table 5.16: Asymptotic performances of SARSA(λ)-R and of SARSA(λ)-PT in the 3 actions tasks.

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT	86.3	69.7	89.3	90	87.7	90.3	91.3	86.7	85.7	87.7	85.7	84.3	90	89
SARSA(λ)-R	85.7	79	94.3	83.7	86.3	75.7	69.7	86.3	78.3	78.7	62.7	54.7	80	92.3

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.96 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to determine if the asymptotic performance using policy transfer from the two actions task to the three actions task was greater than asymptotic performance when the three actions task was learnt using experience replay.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$*=1.771,* t_{calc}= 2.5 which is greater than the critical value. We therefore concluded that the asymptotic performance using policy transfer from the two actions task to the three actions task was greater than asymptotic performance when the three actions task was learnt using experience replay.

5.2.3.5 Comparing Learning Performance of SARSA(λ)-PR with learning performance of SARSA(λ) in the three actions task

Comparison of the two techniques was to help us determine if policy reuse speeds up learning in the three actions task. Figure 5.2 shows that the performance of SARSA(λ)-PR was better than that of SARSA(λ) both at the initial level and at the asymptotic levels. However, in the graph, the fact that some time had been spent in learning the source task before using SARSA(λ)-PR in the target task is not put into consideration. We compared the two algorithms at both the initial and the asymptotic levels while putting the time spent in learning the source task into consideration.

a) Initial performance

Since SARSA(λ)-PR benefited from learning in the source task, we compared the performance after 1000 episodes in using SARSA(λ) against the performance at episode 100 using SARSA(λ)-PR. This is because 900 episodes had been spent in learning in the source task which adds up to 1000 episodes for SARSA(λ)-PR. Table 5.17 shows the performances of the two techniques after the aforementioned number of episodes. Given these values, SARSA(λ) outperforms SARSA(λ)-PR in 12 out of the 14 episodes. We hypothesize that SARSA(λ) has a significantly better performance.

Table 5.17: Initials performances of SARSA(λ)-PR and of SARSA(λ) in the 3 actions task

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PR	58	66	55	41	50	47	64	55	59	54	59	57	40	63
SARSA(λ)	63.7	9.7	56.7	55	65.3	61.3	57	66.7	68.3	67.3	67.3	58.3	73.3	66.7

Using the Shapiro-Wilk test, the value for obtained for the W statistic was 0.74 which is less than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were not normally distributed. The Wilcoxon Signed rank test was therefore used to determine if the initial performance using the baseline SARSA(λ) was significantly better than the performance obtained using SARSA(λ)-PR .

Using the Wilcoxon Signed rank test, the value obtained for the W static was 19 with n′=14. The sum of positive ranks was 86 whereas the sum of negative ranks was 19. For a one tailed Wilcoxon Signed Rank test at a significance level of 0.05, and n′=14, $W_{critical}$=25. Since W is less than $W_{critical}$, we concluded the median initial performance for SARSA(λ) was significantly greater than the median initial performance for SARSA(λ)-PR in the three actions task.

From Figure 5.2 we note that the average performance of SARSA(λ) after 1000 episodes is 59.8%. The performance of SARSA(λ)-PR after 1000 episodes (900 in the source task and 100 in the target task is) 54.9%. The average performance of SARSA(λ) after 1300 episodes is 63.3%. The performance of SARSA(λ)-PR after 1300 episodes (900 in the source task and 400 in the target task is) 64.1%. The most likely reason for this observed performance is that SARSA(λ)-PR starts with a random target policy in the three actions task. After 1000 episodes, SARSA(λ) has already learnt a policy which is better than the random policy initially used by SARSA(λ)-PR. However by bootstrapping the target policy with the two actions policy, learning of the target task policy using SARSA(λ)-PR is accelerated such that it does better than SARSA(λ) after a few episodes of learning in the target task.

b) Asymptotic performance

We compared the performance after 3900 episodes using the baseline SARSA(λ) algorithm against the performance at episode 3000 using the SARSA(λ)-PR algorithm since 900 episodes were spent in learning in the source task.

Table 5.20 shows the asymptotic performances of the two techniques after the aforementioned number of episodes. SARSA(λ)-PR has the better performance in 10 of the fourteen tasks. We performed significance tests to determined if the asymptotic performance of SARSA(λ)-PR was significantly greater than that of SARSA(λ).

Table 5.18: Asymptotic performances of SARSA(λ)-PR and of SARSA(λ) in the 3 actions task

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PR	76.7	77	81.3	77	77	78.3	77.3	72.7	77.3	78.3	79	79	72	77
SARSA(λ)	79	74.7	71.3	64.7	78	67.3	70.7	76.7	70.7	67.7	66	61.3	75.7	76.7

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.93 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to determine if the asymptotic performance of SARSA(λ)-PR was greater than the asymptotic performance of SARSA(λ)-R.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}=1.771$, $t_{calc}= 3.01$ which is greater than the critical value. We therefore concluded that SARSA(λ)-PR has a significantly greater asymptotic performance compared to SARSA(λ).

From Figure 5.2, the best performance attained by SARSA(λ) is 71.5% at episode 3900 (21.7hrs), SARSA(λ)-PR exceeds this performance after 1800 episodes (10hrs) putting into consideration the time spent in learning the source task. Therefore, to attain a performance of 71.5%, SARSA(λ) needs 11.7hrs more than SARSA(λ)-PR.

5.2.3.6 Comparing Learning Performance of SARSA(λ)-PR with learning performance of SARSA(λ)-R in the three actions task

Comparison of the two techniques was to help us determine if we could apply policy reuse instead of experience replay in learning the three actions task. Figure 5.2 shows that though SARSA(λ)-PR initially performed well, SARSA(λ)-R eventually caught up with SARSA(λ)-PR. Since some time had already been spent in learning the source task, it seemed like experience replay performance was better than the performance of policy reuse. We performed significance tests to verify this.

a) Initial performance

Since SARSA(λ)-PR benefited from learning in the source task, we compared the performance after 1000 episodes in using SARSA(λ)-R against the performance at episode 100 using SARSA(λ)-PR since 900 episodes were spent in learning in the source task. Table 5.19 shows the performances of the two techniques after the aforementioned number of episodes.

Table 5.19: Initials performances of SARSA(λ)-PR and of SARSA(λ)-R in the 3 actions task

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-R	72.3	82.3	76.7	72	90	76	72.7	67.7	65	74.3	68	65	70.3	85.3
SARSA(λ)-PR	58	66	55	41	50	47	64	55	59	54	59	57	40	63

Using the Shapiro-Wilk test, the value for obtained for the W statistic was 0.94 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to determine if the initial of performance SARSA(λ)-R was greater than the initial performance of SARSA(λ)-PR in the three actions task.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}=1.771$, $t_{calc}= 7$ which is greater than the critical value. We therefore concluded that use of experience replay in the 3 actions task resulted in significantly greater speedup in initial performance that use of policy reuse.

From Figure 5.2, we can see that the initial performance of SARSA(λ)-PR was 55%. This was achieved after 100 episodes. Considering that 900 episodes were spent in learning the source task, the total number of episodes executed to arrive at this performance was 1000. This translates to about 5.6hrs. SARSA(λ)-R reached the same performance after 320 episodes, which translates to 1.8hrs. We therefore conclude that to arrive at a performance of 55%, SARSA(λ)-PR took 3.8hrs more than SARSA(λ)-R.

b) Asymptotic performance

We compared the performance after 3900 episodes in the experience replay tasks against the performance at episode 3000 in the policy reuse task since 900 episodes were spent in learning in the source task.

Table 5.20 shows the asymptotic performances of the two techniques after the aforementioned number of episodes:

Table 5.20: Asymptotic performances of SARSA(λ)-PR and of SARSA(λ)-R in the 3 actions task

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PR	76.7	77	81.3	77	77	78.3	77.3	72.7	77.3	78.3	79	79	72	77
SARSA(λ)-R	85.7	79	94.3	83.7	86.3	75.7	69.7	86.3	78.3	78.7	62.7	54.7	80	92.3

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.90 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to determine if the asymptotic performance of SARSA(λ)-R was greater than the asymptotic performance of SARSA(λ)-PR.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}=1.771$, $t_{calc}= 0.64$ whose which is less than the critical value. We therefore concluded that in the three actions task, the asymptotic performance of SARSA(λ)-R was not significantly greater than that of SARSA(λ)-PR.

5.2.3.7 Comparing SARSA(λ)-PT-PR with SARSA(λ)-R

Figure 5.2 shows that SARSA(λ)-PT-PR outperforms SARSA(λ)-R at both the initial and at the asymptotic levels. We would like to ascertain that this difference in performance is significant in order to be able to decide which of the algorithms should be used in learning the three actions target task.

a) Initial Performance

To compare the initial performances, we needed to consider the fact that 900 episodes were spent in learning the source task when SARSA(λ)-PT-PR was used in the target task. We therefore took the performance of SARSA(λ)-PT-PR after 100 episodes in the target task, and compared this with the performance of SARSA(λ)-R after 1000 episodes of learning in the target task. Table 5.21 shows the performance after the said number of episodes for the two techniques.

Table 5.21: Initial performances of SARSA(λ)-PT-PR and of SARSA(λ)-R in the 3 actions task.

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT-PR	63	75	80	75	72	80	72	84	85	80	78	80	79	89
SARSA(λ)-R	72.3	82.3	76.7	72	90	76	72.7	67.7	65	74.3	68	65	70.3	85.3

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.94 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to determine if the initial performance of SARSA(λ)-PT-PR was greater than the initial performance of SARSA(λ)-R.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}=1.771$, $t_{calc}=1.4$ whose which is less than the critical value. We therefore conclude that the initial performance of SARSA(λ)-PT-PR is not significantly greater than that of SARSA(λ)-R in learning the three actions task.

b) Asymptotic Performance

We got the asymptotic performance after 3000 episodes of learning in the transfer setting (because 900 episodes are used in learning the source task) and 3900 episodes in the replay setting since replay trials ended after 3900 episodes. The performances after the said number of episodes are shown in Table 5.22.

Table 5.22: Asymptotic performances of SARSA(λ)-PT-PR and of SARSA(λ)-R in the 3 actions tasks.

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT-PR	75	73	90	85	91	85.7	86	87.3	89.3	91.7	86	89.3	77.7	88.7
SARSA(λ)-R	85.7	79	94.3	83.7	86.3	75.7	69.7	86.3	78.3	78.7	62.7	54.7	80	92.3

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.94 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to determine if the asymptotic performance of SARSA(λ)-PT-PR was greater than the asymptotic performance of SARSA(λ)-R in the three actions task.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}=1.771$, $t_{calc}=1.9$ which is greater than the critical value. We therefore concluded that the asymptotic performance using SARSA(λ)-PT-PR in the three actions task was greater than asymptotic performance when the three actions task was learnt using experience replay.

5.2.3.8 Comparing the performance of SARSA(λ)-PT with the performance of SARSA(λ)-R-PT

This comparison was necessary in order to find out if combining experience replay with policy transfer led to more speedup in performance compared to when policy transfer was used alone. Figure 5.2 shows that policy transfer plus experience replay (SARSA(λ)-R-PT) resulted in worse performance than when policy transfer (SARSA(λ)-PT) was used in isolation.

a) Initial Performance

Table 5.23 shows the performances of the two techniques after 100 episodes.

Table 5.23: Initial performances of SARSA(λ)-PT and of SARSA(λ)-R-PT in the three actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT	89	61	80	83	80	87	81	81	78	88	78	80	83	85
SARSA(λ)-R-PT	72	74	62	73	82	67	66	64	66	78	76	81	82	67

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.9 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to determine if SARSA(λ)-PT resulted in greater speedup in initial learning performance compared to SARSA(λ)-R-PT.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}=3.35 which is greater than the critical value. We therefore concluded that SARSA(λ)-PT led to greater speedup in initial learning performance compared to SARSA(λ)-R-PT.

b) Asymptotic Performance

Table 5.24 shows the performances of the two techniques after 3900 episodes.

Table 5.24: Asymptotic performances of SARSA(λ)-PT and of SARSA(λ)-R-PT in the three actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT	88.7	57.3	89	91	90	87.7	89	86	85.3	90.7	93	88	91	87.7
SARSA(λ)-R-PT	82.3	70.3	64	70	73.3	72.3	63.3	82.3	74.3	85	77.7	65.7	62.7	68

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.92 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used determine if SARSA(λ)-PT led to greater speedup in asymptotic learning performance compared to SARSA(λ)-R-PT.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}=4.9 which is greater than the critical value. We concluded that SARSA(λ)-PT led to greater speedup in asymptotic learning performance compared to SARSA(λ)-R-PT.

In addition we also noted that the initial mean performance (after 100 episodes) using SARSA(λ)-R-PT was 72.1% while the asymptotic performance (after 3900 episodes) was 71.5% . This generally implied that combining policy transfer with experience replay led to declining performance.

5.2.3.9 Comparing performances of SARSA(λ)-PR with performance of SARSA(λ)-R-PR

This comparison was necessary in order to find out if combining experience replay with policy reuse led to more speedup in performance compared to when policy reuse was used alone. Figure 5.2 shows that policy reuse plus experience replay (SARSA(λ)-R-PR) resulted in speedup in initial performance than when policy reuse (SARSA(λ)-PR) was used in isolation.

a) Initial Performance

Table 5.25 shows the initial performances of both techniques after 100 episodes.

Table 5.25: Initial performances of SARSA(λ)-PR and of SARSA(λ)-R-PR in the three actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-R-PR	58	67	61	58	63	67	63	57	67	53	62	66	67	68
SARSA(λ)-PR	58	66	55	41	50	47	64	55	59	54	59	57	40	63

Using the Shapiro-Wilk test, the value obtained for the W statistic 0.89 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. We therefore used the paired student's t test to determine if the speedup in initial performance in the three actions task using SARSA(λ)-R-PR was greater than the speed up achieved using SARSA(λ)-PR.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}=3.4 which is greater than the critical value. We therefore concluded that the speedup in initial performance using SARSA(λ)-R-PR was greater than that achieved using SARSA(λ)-PR.

From Figure 5.2, we can see that the initial performance of SARSA(λ)-R-PR was 63%. This was achieved after 100 episodes. This translates to about 0.56hrs. SARSA(λ)-PR reached the same performance after 350 episodes, which translates to 1.94hrs. We therefore conclude that to arrive at a performance of 55%, SARSA(λ)-PR takes 1.39hrs more than SARSA(λ)-R-PR.

b) Asymptotic Performance

Table 5.26 shows the asymptotic performances of both techniques after 3900 episodes.

Table 5.26: Asymptotic performances of SARSA(λ)-PR and of SARSA(λ)-R-PR in the three actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-R-PR	69	79	79.3	73.7	79.3	71.7	78	72.7	65.7	64.3	78	76.3	83.7	72
SARSA(λ)-PR	77.3	73.7	77.7	76	71.7	75.3	77.3	78	81	75.3	72	74.3	77	76.7

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.95 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to determine if the speed up in asymptotic performance using SARSA(λ)-R-PR was greater than the speedup achieved using SARSA(λ)-PR

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}=-0.78 whose magnitude is less than the critical value. We therefore concluded that the speedup in asymptotic performance achieved using the SARSA(λ)-R-PR is not significantly different from the speedup achieved using SARSA(λ)-PR.

5.2.3.10 Comparing the performance of the SARSA(λ)-R-PT-PR Algorithm with the performance of the SARSA(λ)-PT-PR algorithm

Figure 5.2 shows that SARSA(λ)-PT-PR outperforms SARSA(λ)-R-PT-PR both at the initial phase and at the asymptotic level. SARSA(λ)-R-PT-PR combines the three learning speedup techniques viz. experience replay, policy transfer and policy reuse. It has a high initial performance (77%). The asymptotic performance at 73.3% is lower than the initial performance. It therefore does not lead to a speedup in learning performance as expected. This algorithm is derived from the SARSA(λ)-PT-PR algorithm by addition of experience replay. The SARSA(λ)-PT-PR has an average initial performance of 77.9% and an average asymptotic performance of 86.1%. While the initial performances of the two algorithms are similar, the asymptotic performance of SARSA(λ)-PT-PR is much better than that of SARSA(λ)-R-PT-PR. This suggests that the addition of experience replay to the SARSA(λ)-PT-PR results in a decline rather than a speedup in performance. We performed significance tests to determine if these differences in performance were significant.

a) Initial performance

Table 5.27 shows the initial performances of both techniques after 100 episodes.

Table 5.27: Initial performances of SARSA(λ)-R-PT-PR and of SARSA(λ)-PT-PR in the three actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT-PR	63	75	80	75	72	80	72	84	85	80	78	80	79	89
SARSA(λ)-R-PT-PR	79	78	75	82	77	70	73	74	77	81	76	80	76	78

Using the Shapiro-Wilk test, the value obtained for the W statistic 0.95 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. We therefore used the paired student's t test to determine if the difference in initial performances obtained using the two algorithms were significantly different.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}=0.56 which is greater than the critical value. We therefore concluded that the initial performances achieved using the SARSA(λ)-R-PT-PR and SARSA(λ)-PT-PR are not significantly different.

b) Asymptotic Performance

Table 5.28 shows the asymptotic performances of both techniques after 3900 episodes.

Table 5.28: Asymptotic performances of SARSA(λ)-R-PT-PR and of SARSA(λ)-PT-PR in the three actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT-PR	70.7	71	90.7	84	93.3	81.7	86	90.3	90	92.7	89.7	91.7	84.7	89
SARSA(λ)-R-PT-PR	76.3	73	70	74.3	71	72.7	71	74.3	69.7	71.3	72.3	70.7	70.3	73

Using the Shapiro-Wilk test, the value obtained for the W static was 0.83 which is less than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were not normally distributed. The Wilcoxon Signed rank test was therefore used to determine if the difference in asymptotic performance between SARSA(λ)-R-PT-PR and SARSA(λ)-PT-PR was significant.

Using the Wilcoxon Signed rank test, the value obtained for the W static was 3 with n′=14. The sum of positive ranks was 102 whereas the sum of negative ranks was 3. For a one tailed Wilcoxon Signed Rank test at a significance level of 0.05, and n′=14, $W_{critical}$=25. Since W is less than $W_{critical}$, we concluded the median performance of SARSA(λ)-PT-PR was greater than the median performance for SARSA(λ)-R-PT-PR in the three actions task. This means that addition of experience replay to SARSA(λ)-PT-PR did not speed up its learning performance. In fact it was detrimental to the learning process.

5.3 Leaning performance in the six actions task

5.3.1 *Experiments*

All the algorithms in Table 2.1 were used in learning the six actions task. Six of the algorithms used knowledge transfer. The two algorithms that did not use knowledge transfer were SARSA(λ) and SARSA(λ)-R. Actions 0 to 5 in Table 3.2 were used by the control agent to control the robot.

For the algorithms that used knowledge transfer, the control agent was first trained in the source task using SARSA(λ)-R until a performance of 87% was attained after 1500 episodes. The final policy attained in the three actions source task was used to initialise the 6 actions task policy to attain policy transfer. To implement policy reuse, the policy was used to select actions in some of the episodes. The following experiments were performed:

Experiment 11: Using SARSA(λ) to learn the six actions task

Experiment 12: Using SARSA(λ)-R to learn the six actions task

Experiment 13: Using SARSA(λ)-PT to learn the six actions task

Table 5.29 shows how the actions were mapped from the three actions task policy to the six actions task target policy.

Table 5.29: Action mapping in policy transfer from the three actions task to the six actions task

Target Action	Source Action
0, 3	0
1, 4	1
2, 5	2

Experiment 14: Using SARSA(λ)-PR to learn the six actions task

Experiment 15: Using SARSA(λ)-PT-PR to learn the six actions task

The action mapping given in Table 5.29 was used in implementing policy transfer.

Experiment 16: Using SARSA(λ)-R-PT to learn the six actions task

The action mapping given in Table 5.29 was used in implementing policy transfer.

Experiment 17: Using SARSA(λ)-R-PR to learn the six actions task

Experiment 18: Using SARSA(λ)-R-PR-PT to learn the six actions task

The action mapping given in Table 5.29 was used in implementing policy transfer.

5.3.2 *Results of learning in the six actions task using the baseline, proposed and alternative algorithms*

Figure 5.3 shows the performance of all the learning algorithms evaluated in the six actions learning task. From the graph, we notice that SARSA(λ)-PT-PR which uses both policy transfer and policy reuse results in the best performance. In learning without knowledge transfer, both SARSA(λ)-R and SARSA(λ) the baseline learning algorithm have their strengths and weaknesses; between the two, SARSA(λ)-R performs better initially while SARSA(λ) performs better asymptotically.

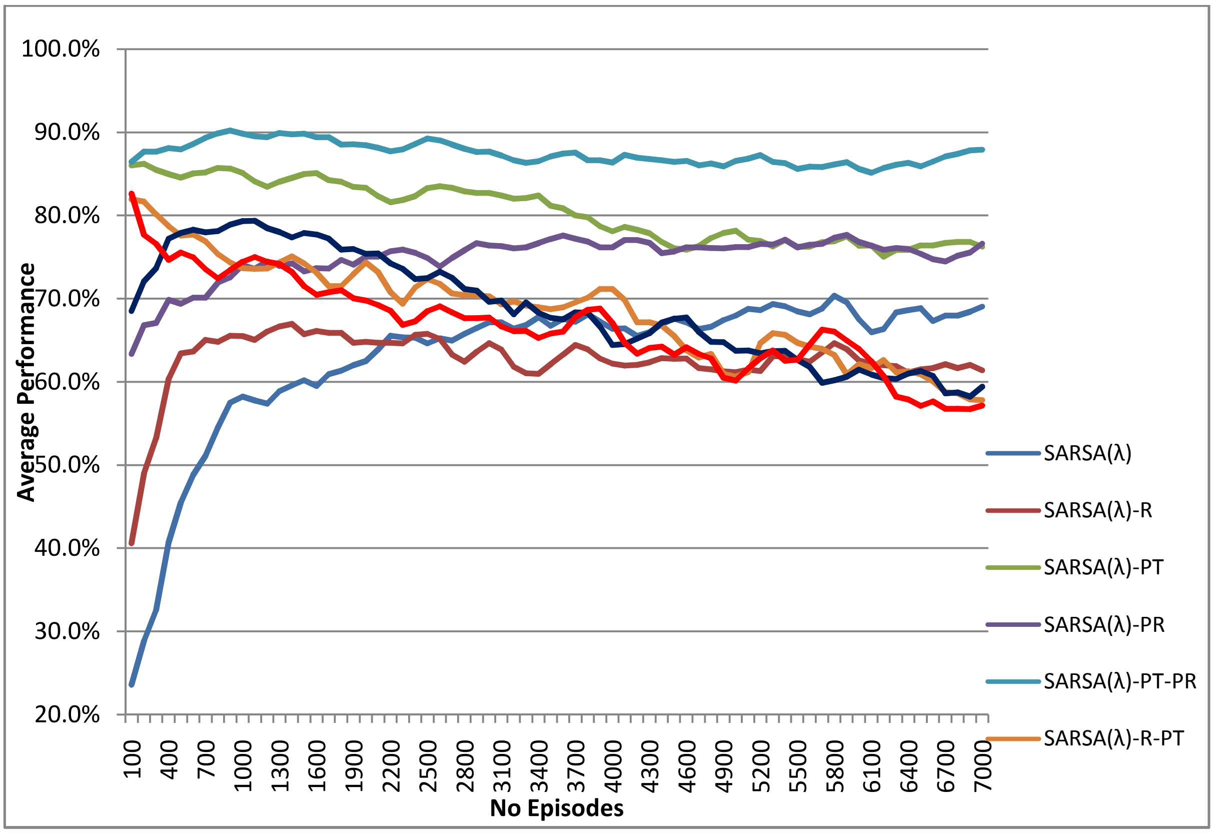


Figure 5.3: Performance of all algorithms in the six actions task

Combining the three techniques: policy transfer, policy reuse and experience replay (SARSA(λ)-R-PT-PR) does not result in good performance as expected. The initial performance is high but then the performance starts to decline after the first few episodes. The most likely explanation for this is overfitting which occurs if a neural network is trained too much with the same examples. Combining policy transfer with experience replay seems to result in similar behaviour. However when policy reuse is combined with experience replay SARSA(λ)-R-PR, there seems to be some benefit in the initial performance compared to when SARSA(λ)-PR is used alone. This may be attributed to the initial low performance of SARSA(λ)-PR.

Table 5.30 shows the episode termination type for episodes in the six actions task. Just as in the three actions task, the learning algorithms eventually learnt how to move towards the goal but had a harder time learning how to avoid obstacles.

Table 5.30: Episode termination type for some algorithms used in the six actions task

		Algorithm				
Episode No	Episode Termination Type	SARSA(λ)	SARSA(λ)-R	SARSA(λ)-PR	SARSA(λ)-PT	SARSA(λ)-PT-PR
100	Goal	23.6%	40.6%	63.4%	86.0%	86.4%
	Time Barred	35.5%	14.4%	13.8%	5.0%	3.0%
	Collision	40.9%	45.0%	22.8%	9.0%	10.6%
3900	Goal	68.8%	61.4%	76.6%	76.2%	87.9%
	Time Barred	4.8%	10.7%	2.5%	8.9%	4.0%
	Collision	26.4%	27.9%	20.9%	14.9%	8.1%

5.3.3 *Comparing the individual learning algorithms in learning the six actions task*

5.3.3.1 Comparison of the SARSA(λ) to the performance of SARSA(λ)-R

SARSA(λ) is the baseline learning algorithm. Both SARSA(λ) and SARSA(λ)-R can be used for learning if the six action task is considered to be the initial source task. This comparison was used to verify if experience replay would speed up learning in the six actions task.

Figure 5.3 shows that SARSA(λ)-R led to greater performance in the early episodes of the task. As the number of episodes increased, SARSA(λ)-R's performance improvement slowed down to such an extent that SARSA(λ)'s performance became better than that of SARSA(λ)-R. We compared both the initial and the asymptotic performances of the algorithms to determine if the differences in performance were significant.

a) Initial Performances

Table 5.31 shows the initial performances obtained using the two techniques in the 14 tested scenarios. The performances were recorded after 100 episodes. We compared the performances to determine if the initial learning performance using SARSA(λ)-R in the six actions task was significantly greater than the performance of SARSA(λ) in the same task. The values shown in Table 5.31 are the percentage of episodes that ended in the goal region.

Table 5.31: Initial performances of SARSA(λ)-R and SARSA(λ) in the six actions task

Scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-R	38	56	43	40	52	47	21	19	49	39	40	31	39	55
SARSA(λ)	23	56	44	12	22	21	17	11	12	16	22	18	23	34

Using the Shapiro-Wilk test, the value of the W statistic obtained was 0.97 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. We therefore used the paired student's t test to determine if the initial learning performance using SARSA(λ)-R in the six actions task was significantly greater than the performance when SARSA(λ) was used.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}=5.55 which is greater than the critical value. We therefore concluded that the initial learning performance using SARSA(λ)-R in the six actions task was greater than the performance when SARSA(λ) algorithm was used.

From Figure 5.3, we notice that the initial performance using SARSA(λ)-R is 40%, which was attained by episode 100 translating to 0.56hrs. SARSA(λ) attained the same performance after 400 episodes translating to 2.2hrs. Thus to get to an initial performance of 40% SARSA(λ) uses 1.7hrs more than SARSA(λ)-R.

b) Asymptotic Performances

Table 5.32 shows the performances of both techniques after 7000 episodes. The values shown are the percentage of episodes that ended at the goal region. SARSA(λ) has a better performance in 9 of the 14 scenarios.

Table 5.32: Asymptotic performances of SARSA(λ)-R and SARSA(λ) in the six actions task

Scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)	60	57	80	67	82	71.3	85.3	60	57.7	66	80	71.7	69	56
SARSA(λ)-R	44.7	63	58	78.3	68.3	54	74.7	67.7	69.3	51	50	72.3	62.7	45.3

Using the Shapiro-Wilk test, value obtained for the W statistic was 0.94 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. We therefore used the paired student's t test to determine if the asymptotic learning performance using SARSA(λ) in the six actions task was significantly greater than the performance when the SARSA(λ)-R algorithm was used.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}=2.14 which is greater than the critical value. We therefore concluded that the asymptotic learning performance using SARSA(λ) in the six actions task was significantly greater than the performance when the SARSA(λ)-R algo rithm was used.

5.3.3.2 Performance of SARSA(λ)-PT in the six actions task

Figure 5.3 shows that SARSA(λ)-PT which uses policy transfer had a good initial performance(87.7%), but subsequently, performance declined rather than increased with time. The final average asymptotic performance is 76.2%. This implies that the use of policy transfer in this task might be detrimental to learning. Table 5.33 shows both the initial and asymptotic performances of SARSA(λ)-PT in the fourteen scenarios tested. The initial performance was calculated after 100 episodes while the asymptotic performance was calculated after 7000 episodes.

We performed a significance test to determine if the decline in performance was significant.

Table 5.33: Initial and asymptotic performance of SARSA(λ)-PT in the six actions task

Scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Initial	82	89	81	84	88	81	87	85	87	84	90	87	87	92
Asymptotic	2	82.3	56	91	86.7	65	89.3	88	86.7	82.7	80.3	91	82	84.7

Using the Shapiro-Wilk test, the value obtained for the W static was 0.64 which is less than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were not normally distributed. The Wilcoxon Signed rank test was therefore used to determine if the initial performance of SARSA(λ)-PT was significantly greater than the asymptotic performance in the six actions task.

Using the Wilcoxon Signed rank test, the value obtained for the W static was 24 with n′=13. The sum of positive ranks was 81 whereas the sum of negative ranks was 24. For a one tailed Wilcoxon Signed Rank test at a significance level of 0.05, and n′=14, $W_{critical}$=25. Since W is less than $W_{critical}$, we concluded the median initial performance of SARSA(λ)-PT was significantly greater than its median asymptotic performance. Therefore SARSA(λ)-PT is detrimental to learning the six actions task.

5.3.3.3 Performance of SARSA(λ)-R-PT-PR in the six actions task

Figure 5.3 shows that SARSA(λ)-R-PT-PR which uses experience replay, policy transfer and policy reuse had a good initial performance(82.6%). The asymptotic performance was 57.2% which indicates that there was a large decline in performance. This implies that the use of SARSA(λ)-R-PT-PR in this task might be detrimental to learning. Table 5.34 shows both the initial and asymptotic performances of SARSA(λ)-R-PT-PR in the fourteen scenarios tested. The initial performance was calculated after 100 episodes while the asymptotic performance was calculated after 7000 episodes.

We performed a significance test to determine if the decline in performance was significant.

Table 5.34: Initial and asymptotic performance of SARSA(λ)-R-PT-PR in the six actions task

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Initial	70	93	76	85	72	86	82	88	84	88	92	85	86	66
Asymptotic	49.7	68.7	48	49.7	53	56.3	53	70	54.7	62.7	70	52.7	84.7	58

Using the Shapiro-Wilk test, the value obtained for the W static was 0.91 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. We therefore used the paired student's t test to determine if the initial learning performance was significantly greater than the asymptotic learning performance.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}=9.2 which is greater than the critical value. We therefore concluded that the initial learning performance using SARSA(λ)-PT-PR in the six actions task was significantly greater than the asymptotic learning performance. This algorithm is therefore detrimental to learning in this task.

5.3.3.4 Performance of SARSA(λ)-R-PT in the six actions task

Figure 5.3 shows that SARSA(λ)-R-PT which uses experience replay and policy transfer had a good initial performance(81.9%). The asymptotic performance was 57.8% which indicates that there was a large decline in performance. This implies that the use of SARSA(λ)-R-PT in this task might be detrimental to learning. Table 5.35 shows both the initial and asymptotic performances of SARSA(λ)-R-PT in the fourteen scenarios tested. The initial performance was calculated after 100 episodes while the asymptotic performance was calculated after 7000 episodes.

We performed a significance test to determine if the decline in performance was significant.

Table 5.35: Initial and asymptotic performance of SARSA(λ)-R-PT in the six actions task

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Initial	79	89	78	82	79	89	76	82	92	85	87	75	75	79
Asymptotic	49	48	57	54	55.7	68.7	55.3	58	68.3	47.3	80.7	60.3	53.7	53.3

Using the Shapiro-Wilk test, the value obtained for the W static was 0.95 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. We therefore used the paired student's t test to determine if the initial learning was significantly greater than the asymptotic learning performance.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}=10.4 which is greater than the critical value. We therefore concluded that the initial learning performance using SARSA(λ)-R-PT in the six actions task was significantly greater than the asymptotic learning performance. This algorithm is therefore detrimental to learning in this task.

5.3.3.5 Performance of SARSA(λ)-R-PR in the six actions task

Figure 5.3 shows that SARSA(λ)-R-PR which uses experience replay and policy reuse had a initial performance of 68.5%. The asymptotic performance was 59.4% which indicates that there was a large decline in performance. This implies that the use of SARSA(λ)-R-PR in this task might be detrimental to learning. Table 5.36 shows both the initial and asymptotic performances of SARSA(λ)-R-PR in the fourteen scenarios tested. The initial performance was calculated after 100 episodes while the asymptotic performance was calculated after 7000 episodes.

We performed a significance test to determine if the decline in performance was significant.

Table 5.36: Initial and asymptotic performance of SARSA(λ)-R-PR in the six actions task

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Initial	77	71	67	70	69	60	74	70	68	61	67	65	66	74
Asymptotic	54.7	81.3	56	75.5	53	60	47	84.3	48.3	49.3	44.7	52	44.7	80

Using the Shapiro-Wilk test, the value obtained for the W static was 0.91 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. We therefore used the paired student's t test to determine if the initial learning performance was significantly greater than the asymptotic learning performance.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}=2.5 which is greater than the critical value. We therefore concluded that the initial learning performance using SARSA(λ)-R-PR in the six actions task was significantly greater than the asymptotic learning performance. This algorithm is therefore detrimental to learning in this task.

5.3.3.6 Comparing SARSA(λ)-PR with SARSA(λ)-R

The policy reuse technique was compared to the experience replay technique at both the initial level and at the asymptotic level. This was to determine if knowledge transfer using policy reuse led to a significant speedup in learning performance when used to learn the six actions task compared to when experience replay was used.

a) Initial performance

SARSA(λ)-PR required the reuse of the final three actions task source task policy together with the new six actions task policy in acting in the 6 actions task. Since 1500 episodes were spent in learning the 3 actions task policy, we compared the initial performance after 1600 episodes had been spent in learning the six actions task using SARSA(λ)-R, with the performance of SARSA(λ)-PR after 100 episodes had been spent learning the same task. Table 5.37 shows the performance for both techniques after the said number of episodes.
We wanted to determine if the initial learning performance using SARSA(λ)-PR in learning the six actions was greater than the initial performance when SARSA(λ)-R was used in learning the same task.

Table 5.37: Initial performances of SARSA(λ)-PR and of SARSA(λ)-R in the six actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-R	60.3	76	72.7	79	74.3	63.7	64.7	85	70.7	35	52	53.7	74	64.7
SARSA(λ)-PR	66	67	64	58	61	60	63	68	69	64	62	64	62	60

Using the Shapiro-Wilk test, the value obtained for the W static was 0.94 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to perform the comparison.

For a one tailed t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}=1.771$, $t_{calc}= 0.77$ which is less than the critical value. We therefore concluded that the initial learning performance using SARSA(λ)-PR in learning the six actions task was not significantly different from the initial performance obtained when SARSA(λ)-R was used in learning the same task.

b) Asymptotic performance

Again we considered the fact the 1500 episodes were spent in learning the 3 actions. We compared the performance obtained after 7000 episodes had been spent in learning the six actions task using SARSA(λ)-R, to the performance obtained after 5500 episodes had been spent in learning the same task using SARSA(λ)-PR. Table 5.38 shows the performance for both techniques after the said number of episodes.

We wanted to determine if the asymptotic learning performance using SARSA(λ)-PR in learning the six actions was greater than the asymptotic performance when SARSA(λ)-R was used in learning the same task.

Table 5.38: Asymptotic performances of SARSA(λ)-PR and of SARSA(λ)-R in the six actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PR	80	72.7	84.3	72	76.7	75.7	82	72.3	74	78.3	65.7	79	75.7	77.7
SARSA(λ)-R	44.7	63	58	78.3	68.3	54	74.7	67.7	69.3	51	50	72.3	62.7	45.3

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.95 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to perform the comparison.

For a one tailed t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}=1.771$, $t_{calc}= 4.56$ which is greater than the critical value. We therefore concluded that the asymptotic learning performance using SARSA(λ)-PR in learning the six actions task was greater than the asymptotic performance when SARSA(λ)-R was used in learning the same task.

5.3.3.7 SARSA(λ)-PR with SARSA(λ)

Figure 5.3 shows that in learning the six actions task without experience replay, a better asymptotic performance was obtained than if experience replay was used. We decided to compare the performance of SARSA(λ)-PR and SARSA(λ) both at the initial and at the asymptotic levels, to determine if SARSA(λ)-PR resulted in significant increase in performance compared to the baseline algorithm.

a) Initial Performance

We still considered the fact that 1500 episodes spent in learning the three actions source task. The performance of SARSA(λ)-PR in episode 100 was compared to the performance of SARSA(λ)at episode 1600.

We sought to determine if the initial learning performance using SARSA(λ)-PR in learning the six actions was greater than the initial performance when SARSA(λ) was used in learning same task. Table 5.39 shows the initial performances of both techniques.

Table 5.39: Initial performance of SARSA(λ)-PR and of SARSA(λ) in the six actions task

Scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PR	66	67	64	58	61	60	63	68	69	64	62	64	62	60
SARSA(λ)	53.8	57.5	54.9	42	54.3	52.2	55.8	43.5	48	52.8	51.3	55	54.5	49.7

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.80 which was less than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were not normally distributed. The Wilcoxon's signed rank test was therefore used to perform the comparison.

Using the Wilcoxon's signed rank test the valued obtained for the W statistic was 0 with n′=14. The sum of positive ranks was 105 whereas the sum of negative ranks was 0. For a one tailed Wilcoxon Signed Rank test at a significance level of 0.05, and n′=14, $W_{critical}$=25. Since W is less than $W_{critical}$, we concluded the median initial performance of SARSA(λ)-PR was significantly greater than the median initial performance for SARSA(λ) in learning the six actions task.

From Figure 5.3, we can see that the initial performance obtained using SARSA(λ)-PR in the six actions task was 63%. This was achieved after 100 episodes. If we add the 1500 episodes spent in learning the source task we get a total of 1600 episodes. This translates to about 8.9hrs. Using SARSA(λ), the same performance was obtained after 2000 episodes which translates to 11.1hrs. Therefore to attain a performance of 63%, SARSA(λ) used 2.2hrs more than SARSA(λ)-PR.

b) Asymptotic Performance

Considering the 1500 episodes spent in learning the source task, the performance of SARSA(λ)-PR in episode 5500 was compared to the performance of SARSA(λ) at episode 7000.

We sought to determine if the asymptotic learning performance using SARSA(λ)-PR in learning the six actions was greater than the asymptotic performance when SARSA(λ) was used in learning the same task. Table 5.40 shows the asymptotic performances of both techniques.

Table 5.40: Asymptotic performance of SARSA(λ)-PR and of SARSA(λ) in the six actions task

Scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PR	80	72.7	84.3	72	76.7	75.7	82	72.3	74	78.3	65.7	79	75.7	77.7
SARSA(λ)	60	57	80	67	82	71.3	85.3	60	57.7	66	80	71.7	69	56

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.96 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to perform the comparison.

For a one tailed t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}= 2.72 which is greater than the critical value. We therefore concluded that the asymptotic learning performance using SARSA(λ)-PR in learning the six actions was greater than the asymptotic performance when SARSA(λ) was used in learning the same task.

From Figure 5.3 we can see that the best performance obtained using SARSA(λ) in the six actions task was 71%. This was achieved after 5800 episodes, which translates to 32.2hrs. SARSA(λ)-PR arrived at the same performance after 600 episodes. If we add the 1500 episodes spent in learning the source task we get a total of 2100 episodes. This translates to about 11.7hrs. Therefore to attain a performance of 71%, SARSA(λ) used 20.6hrs more than SARSA(λ)-PR in learning the six actions task.

5.3.3.8 Comparing SARSA(λ)-PT-PR with SARSA(λ)-PR

Since both algorithms were not detrimental to learning in the six actions task, we compared their performance in order to find out which of the two algorithms had better performance.

a) Initial performance

We sought to determine if the initial performance of SARSA(λ)-PT-PR with was greater than that of SARSA(λ)-PR in the six actions task. Table 5.41 shows the initial performances of both techniques.

Table 5.41: Initial performances of SARSA(λ)-PT-PR, and of SARSA(λ)-PR in the six actions task

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT-PR	86	94	85	88	86	83	87	87	83	87	81	85	90	88
SARSA(λ)-PR	66	67	64	58	61	60	63	68	69	64	62	64	62	60

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.97 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to perform the comparison.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}= 19.62 which is greater than the critical value. We therefore concluded that the initial performance of SARSA(λ)-PT-PR was significantly greater than the initial performance of SARSA(λ)-PR in the six actions task.

b) Asymptotic performance

We sought to determine if the asymptotic performance of SARSA(λ)-PT-PR was greater than the asymptotic performance of SARSA(λ)-PR in the six actions task with knowledge transfer from the three actions task to the six actions task. Table 5.42 shows the asymptotic performances of both techniques.

Table 5.42: Asymptotic performance of SARSA(λ)-PT-PR, and of SARSA(λ)-PR in the six actions task

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT-PR	85.7	89	85.3	86.7	94	88.7	88.3	85	89.7	84.7	89	89.7	88.7	86
SARSA(λ)-PR	75.5	79	80.3	73	85	76.3	88	72	74	78.7	65.3	67.7	74.7	83.7

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.97 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to perform the comparison.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}= 6.28 which is greater than the critical value. We therefore concluded that the asymptotic performance of SARSA(λ)-PT-PR with hand coded mapping was significantly greater than the asymptotic performance of SARSA(λ)-PR in the six actions task.

From Figure 5.3, we can see that the least performance of SARSA(λ)-PT-PR with hand coded mapping was 85.1%. This performance was obtained by episode 6100 which translate to 33.9hrs. We find that SARSA(λ)-PR never reached this level of performance even after 7000 episodes (38.8hrs). We therefore concluded that to get to a performance of 85.1% in the six actions task SARSA(λ)-PR took at least 5hrs more than SARSA(λ)-PT-PR with hand coded mapping.

5.3.3.9 SARSA(λ)-PT-PR verses SARSA(λ)

Since the initial performance of SARSA(λ)-PR is significantly better that the initial performance of SARSA(λ), and the initial performance SARSA(λ)-PT-PR is significantly better than the initial performance of SARSA(λ)-PR we can infer that the initial performance of SARSA(λ)-PT-PR is significantly better than the initial performance of SARSA(λ). To compare the two algorithms, we perform significance tests only at the asymptotic level.

We sought to determine if the asymptotic performance of SARSA(λ)-PT-PR with was greater than the asymptotic performance of SARSA(λ) in the six actions task with knowledge transfer from the three actions task to the six actions task. Table 5.43 details the asymptotic performances of both algorithms.

Table 5.43: Asymptotic performance of SARSA(λ)-PT-PR, and of SARSA(λ) in the six actions task

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT-PR	85.7	86	82	80.7	90	81.7	87	85.7	89.3	83.3	89.3	89.7	81	86
SARSA(λ)	60	57	80	67	82	71.3	85.3	60	57.7	66	80	71.7	69	56

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.93 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to perform the comparison.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}= 6.1 which is greater than the critical value. We therefore concluded that the asymptotic performance of SARSA(λ)-PT-PR with was significantly greater than the asymptotic performance of SARSA(λ) in the six actions task.

From Figure 5.3, we can see that the least performance of SARSA(λ)-PT-PR was 85.1%. This performance was obtained by episode 6100 which translate to 33.9hrs. We find that SARSA(λ) never reached this level of performance even after 7000 episodes (38.8hrs). We therefore concluded that to get to a performance of 85.1% in the six actions task SARSA(λ) took at least 5hrs more than SARSA(λ)-PT-PR.

5.3.3.10 SARSA(λ)-PT-PR verses SARSA(λ)-R

Since the asymptotic performance of SARSA(λ)-PT-PR is significantly better that the asymptotic performance of SARSA(λ), and the asymptotic performance SARSA(λ) is significantly better than the asymptotic performance of SARSA(λ)-R we can infer that the asymptotic performance of SARSA(λ)-PT-PR is significantly better than the asymptotic performance of SARSA(λ)-R. To compare the two algorithms, we perform significance tests only at the initial level.

We sought to determine if the initial performance of SARSA(λ)-PT-PR significantly greater than the initial performance of SARSA(λ)-R in the six actions task. Table 5.44 details the initial performances of both algorithms.

Table 5.44: Initial performance of SARSA(λ)-PT-PR, and of SARSA(λ)-R in the six actions task

Scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-PT-PR	86	94	85	88	86	83	87	87	83	87	81	85	90	88
SARSA(λ)-R	60.3	76	72.7	79	74.3	63.7	64.7	85	70.7	35	52	53.7	74	64.7

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.92 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that the differences in performance were normally distributed. The paired student's t test was therefore used to perform the comparison.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}= 6.2 which is greater than the critical value. We therefore concluded that the initial performance of SARSA(λ)-PT-PR with was significantly greater than the initial performance of SARSA(λ)-R in the six actions task.

5.3.3.11 Performance of SARSA(λ)-R-PR and SARSA(λ)-PR in the six actions task

Figure 5.3 shows that combining policy reuse with experience replay seemed to proffer some benefit at the initial learning phases, we tested if this gain in performance in the initial phases of learning was significant.

We sought to determine if the initial learning performance of SARSA(λ)-R-PR in the six actions task was greater than the initial performance of SARSA(λ)-PR. Table 5.45 shows the initial performance of the two techniques after 100 episodes.

Table 5.45: The initial performance of SARSA(λ)-PR and of SARSA(λ)-R-PR in the six actions task

scenario	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SARSA(λ)-R-PR	78.7	71.7	71	73	77	69.7	75	77.3	76.7	71.7	70.3	74	69.3	75.3
SARSA(λ)-PR	68	61.3	66.7	68.3	69.3	63.3	61.7	70	72	72	67.3	67	65	67.1

Using the Shapiro-Wilk test, the value obtained for the W statistic was 0.98 which is greater than the critical value of 0.874 at a significance level of 0.05 for n=14. This implied that differences in performance were normally distributed. We therefore used the paired student's t test to perform the comparison.

For a one tailed paired Student's t-test at a significance level of 0.05 and 13 degrees of freedom, $t_{critical}$=1.771, t_{calc}= 7 which is greater than the critical value. We therefore concluded that the initial learning performance of SARSA(λ)-R-PR in the six actions task was greater than the initial performance of SARSA(λ)-PR.

From Figure 5.3, we can see that the initial performance of SARSA(λ)-R-PR was 68%. This performance was obtained by episode 100 which translate to 0.6hrs. We find that SARSA(λ)-PR reached the same performance after 350 episodes (1.9hrs). We therefore conclude that to get to an initial performance of 68% in the six actions task, SARSA(λ)-PR took 1.4hrs more than SARSA(λ)-R-PR.

6. Discussion and Conclusion

In this chapter, section 6.1 is used to answer the research questions posed in section 2.14. In answering the research questions, the performance of the techniques explored vis-à-vis the baseline algorithm's performance is discussed. For a particular technique, we analyse if its performance is superior to the performance of the baseline algorithm, in addition, we analyse its performance vis-à-vis the performance of other learning techniques.

In section 6.2, the updated learning framework is illustrated. This framework resulted from the modification of the framework proposed in section 2.12. The section also contains explanations on the reasons for the modifications of the framework based on the results obtained in this study. In section 6.3, experiments performed to validate the updated framework and the results achieved are described.

Section 6.4 lists the achievements of this study based on the objectives given in the introduction chapter. In section 6.5, the contributions of this study are discussed. These are categorised into theoretical contributions, methodological contributions and technical contributions.

Section 6.6 contains a listing of the limitations of the study. Section 6.7 contains suggested further research work that can be used to extend the findings of this study. Section 6.8 contains a summary remark that describes the work carried out and the results obtained.

6.1 Answering the research questions

The main objective of this study was to explore the use of *knowledge transfer* and *experience replay* to speed up learning in the obstacle avoidance task. In order to attain this objective, the following five specific objectives were formulated:

1) Review the state of the art in *knowledge transfer*, experience replay and application of reinforcement learning in obstacle avoidance
2) Propose framework for the application of knowledge transfer and *experience replay* in reinforcement learning in the obstacle avoidance task
3) Develop a control agent based on the proposed framework and interface the control agent with a robot simulation environment

4) Evaluate the performance of the control agent by performing simulations to determine the baseline performance, and the performance when *knowledge transfer* and experience replay are applied to the obstacle avoidance task.
5) Update and validate the proposed learning framework

The first objective was dealt with via literature review in chapter 2. In that chapter we reviewed the different methods used for *knowledge transfer* and reflected on their applicability to the obstacle avoidance task. We found that application of reinforcement learning to the obstacle avoidance task was limited to more simplified versions of the task. We also found that there were few examples of use of inter-domain *knowledge transfer* and *experience replay* in learning the obstacle avoidance task.

The second objective was addressed by the formulation of a learning framework that combined *knowledge transfer* techniques and *experience replay*. The two *knowledge transfer* techniques were *policy reuse* and *policy transfer*. We then proposed seven learning algorithms. One was based on use of the learning framework as was designed, while the rest resulted from suggested possible modifications on the framework. We then formulated research questions that would be answered by the evaluation of the proposed learning algorithms. In chapter 3, we described the methodology we were going to follow to address the third, fourth and fifth objectives.

In chapter 4, we addressed the third objective by developing a simulation system that implemented the proposed algorithms. In order to address the fourth objective, the alternative algorithms were tested and their individual performance compared in chapter 5. In this section, we answer the research questions posed in section 2.14, based on the results obtained in chapter 5. This will help in the refinement of the learning framework proposed in chapter 2.

These research questions were:

i. Does the use of *experience replay* speedup learning in the obstacle avoidance task?
ii. Does the use of *policy transfer* speed up learning in the obstacle avoidance task?
iii. Does the use of *policy reuse* speed up learning in the obstacle avoidance task?

iv. Does combining *policy transfer* with *policy reuse* lead to greater speedup than when each of the techniques is used alone?
v. Does combining *knowledge reuse* with *experience replay* lead to greater speedup than when each of the techniques is used alone?

For an algorithm to be said to speedup learning in a task, it just needed to have a better performance than the baseline algorithm at either the initial level or the asymptotic level or both. If the algorithm is faster at the initial level, it can be used at the initial levels of learning, and when it is deemed to be no longer beneficial, the baseline algorithm can take over control of the learning process. If it is faster at both the initial and at the asymptotic levels, then it should be used throughout the learning process. If it is faster only at the asymptotic level, then the baseline algorithm can be used for learning at the initial level, and then replaced with the alternative algorithm from the point when the alternative algorithm is likely to lead to greater speedup in performance.

6.1.1 *Use of experience replay in learning*

This section sought to answer research question *i* which states *"Does the use of experience replay speedup learning in the obstacle avoidance task?"*.

In order to answer this question the performance of SARSA(λ)-R was compared with the performance of the baseline algorithm SARSA(λ) in 3 tasks. These were the *two actions* task, the *three actions* task and the *six actions* task. In each task, each algorithm was executed in fourteen different scenarios. In summary, *experience replay* was found to speedup learning in all the tasks.

In the *two actions* task, compared to SARSA(λ), SARSA(λ)-R had significantly better performance at both the initial and at the asymptotic levels. Table 6.1 summarises the initial and asymptotic performances of the two algorithms in this task.

Table 6.1: Average Initial and asymptotic performances of SARSA(λ) and SARSA(λ)-R in the two actions task

	Algorithm & Performance	
No of Episodes(time)	SARSA(λ)	SARSA(λ)-R
100 (0.6hrs)	21%	40.6%
3600(20hrs)	76.9%	82.3%

In the *three actions* task, compared to SARSA(λ), SARSA(λ)-R had significantly better performance at both the initial and at the asymptotic levels. Table 6.2 summarises the initial and asymptotic performances of the two algorithms in the *three actions* task.

Table 6.2: Average Initial and asymptotic performances of SARSA(λ) and SARSA(λ)-R in the three actions task

	Algorithm & Performance	
No of Episodes(time)	SARSA(λ)	SARSA(λ)-R
100 (0.6hrs)	22.8%	37.6%
3600(20hrs)	71.5%	79.1%

In the *six actions* task, compared to SARSA(λ), SARSA(λ)-R had significantly better initial performance but a significantly worse asymptotic performance. Table 6.3 summarises the initial and asymptotic performances of the two algorithms in the *six actions* task.

Table 6.3: Average Initial and asymptotic performances of SARSA(λ) and SARSA(λ)-R in the six actions task

	Algorithm & Performance	
No of Episodes(time)	SARSA(λ)	SARSA(λ)-R
100 (0.6hrs)	23.6%	40.6%
7000(38.9hrs)	69.0%	61.4%

In the *six actions* task, 67% was the best performance attained using SARSA(λ)-R after which performance started to decline. By episode 2100 both SARSA(λ) and SARSA(λ)-R had a performance of 64.7% which means that SARSA(λ) had caught up with SARSA(λ)-R . At the end of the trial (7000 episodes), the average performance of SARSA(λ) was 69% while that of SARSA(λ)-R was 61.4% which shows that SARSA(λ) had overtaken SARSA(λ)-R in performance.

Other studies have shown that *experience replay* speeds up learning in various task. For instance in Kalyanakrishnan & Stone(2007), in the 3vs2 keepaway task, after 100 episodes, the Q-learning algorithm that used ER had a hold time of about 7.7 seconds, while the Q-learning algorithm that did not use ER had a hold time of 5 seconds (In keepaway, longer hold times imply better performance). After 400 episodes, Q-learning with ER had hold times of 10.2 seconds. For Q-learning without ER, after 5000 episodes of training, the hold time was 8.5 seconds meaning it had not reached the performance that the ER based algorithm had be able to attain after 400 episodes.   These results compare well with our findings in the *two actions* task and the *three actions* task.

Lin(1992) showed that asymptotically, Q-learning performed as well as Q-learning with *experience replay*. This compares well with our findings in the *six actions* task.

Lin (1992) hypothesizes that the slowing down in performance of ER based algorithms may be due to overfitting. Measures that can be taken to avoid this include turning off ER and reducing the number of experiences and iteration used during ER.

In learning using the of SARSA(λ) algorithm in the *six actions* task, the performance attained by episode 3000 is 67.1%. At the end of 7000 episodes, the performance is 69%. This small change in performance is achieved after 4000 episodes. Since SARSA(λ) is not affected by overfitting , these results imply that unlearning Cetina (2008) might be taking place. This would occur if agent starts to collide with obstacles which it could previously avoid successfully. This could explain the high number of collisions detailed in Table 5.1, Table 5.8 and Table 5.30. This is a weakness of neural network learning methods since the weights do not completely stabilize. It may however be partially dealt with by reducing the learning rate as the performance improves.

Given these results, our major contribution is in showing the change in performance of ER as the task becomes progressively difficult. This had not been shown before. The effect of this on the learning framework is to make it necessary to actively monitor learning performance during learning in the source task. If performance fails to improve or starts to decline after a given number of episode, then ER can be switched off.

6.1.2 *Use of Policy Transfer in learning*

This section sought to answer research question *ii* which states *"Does the use of policy transfer speed up learning in the obstacle avoidance task?"*.

6.1.2.1 Comparing the policy transfer algorithm with the baseline learning algorithm

In order to answer this question the performance of SARSA(λ)-PT was compared with the performance of the baseline algorithm SARSA(λ) in 2 tasks. These were the *three actions* task, and the *six actions* task. In each task, each algorithm was executed in fourteen different scenarios. For the algorithm to be said to speedup learning in either of the tasks, it just needed to have a better performance either at the initial level, or at the asymptotic level or both.

Other studies in the same area have a similar number of pairs of tasks in evaluating the performance of *knowledge transfer* techniques. Examples include Taylor & Stone (2005) and Fernández et al. (2010). In (Taylor et al. (2007) however, only one pair of tasks was used.

In the *three actions* task SARSA(λ)-PT had significantly better performance compared to SARSA(λ) at both the initial and at the asymptotic levels. Table 6.4 summarises the initial and asymptotic performances of both algorithms. Note that the 900 episodes spent in learning the source task are put into consideration hence the performance of SARSA(λ)-PT after 100 episodes, can only be compared with the performance of SARSA(λ) after 1000 episodes of learning in the 3 actions task.

Table 6.4: Average Initial and asymptotic performances of SARSA(λ) and SARSA(λ)-PT in the three actions task

	SARSA(λ)		SARSA(λ)-PT	
	No Episodes	Performance	No Episodes	Performance
Initial	1000(5.6hrs)	59.8%	100(5.6hrs)	81%
Asymptotic	3900(21.7hrs)	71.5%	2900(21.7hrs)	86.5%

These results compare well with those achieved in Taylor et al. (2007) where in the transfer from the 3vs2 to 4vs keepaway, the algorithm using transfer attained a hold time of 8.5 seconds after about 75hrs of training, while the algorithm that did not use transfer required over 450hrs of training to achieve the same performance. In our case, the algorithm using transfer attains a performance of 81% after 5.6 hrs of training. The algorithm that does not use transfer does not reach this performance even afer 21.7hrs of training.

In the *six actions* task, the SARSA(λ)-PT algorithm was seen to be detrimental to learning. This is because the average initial performance was significantly greater than the average asymptotic performance. Table 6.5 illustrates the initial and asymptotic performance of SARSA(λ)-PT and SARSA(λ).

Table 6.5: Average Initial and asymptotic performances of SARSA(λ) and SARSA(λ)-PT in the six actions task

	SARSA(λ)		SARSA(λ)-PT	
	No Episodes	Performance	No Episodes	Performance
Initial	1600	59.5%	100	86%
Asymptotic	7000	69%	5500	76.5%

These results imply that it may be preferable not to use *policy transfer* in reusing the *three actions* task policy in the *six actions* task.

We can hypothesize that the 6 outputs network represented a function that was very different from a 3 outputs network. The 3 outputs network weights may therefore not have been suitable for initializing the 6 actions network. As a general rule therefore, if many new nodes are to be added to the target network, it is not advisable to use *policy transfer*.

Given these results, our major contribution is in showing that if the target task network is very different from the source task network, then *policy transfer* can be detrimental to learning in the target task. The effect of this on the learning framework is that if the source task and the target task are not very different, *policy transfer* can be used and it will speed up the learning. Otherwise some other technique should be used in learning the target task. Since determining if the target task is very different from the source task may not be easy, the performance of SARSA(λ)-PT can be monitored and if it is found to be declining, an alternative algorithm can be selected to take control of the learning.

6.1.2.2 Comparing the policy transfer algorithm with other learning algorithms

6.1.2.2.1 *Comparing the policy transfer algorithm with the experience replay algorithm*

The algorithm that applied the *policy transfer* technique was SARSA(λ)-PT. The algorithm that applied the *experience replay* technique was SARSA(λ)-R. Any of these two algorithms can be used in learning the target task. By comparing, we can decide which of the two should be used for learning in the target task.

In the *three actions* task SARSA(λ)-PT was found to outperform SARSA(λ)-R both at the initial and at the asymptotic levels. Table 6.6 shows the average initial and asymptotic of both algorithms in the 3 actions task.

Table 6.6: Average Initial and asymptotic performances of SARSA(λ)-R and SARSA(λ)-PT in the three actions task

	SARSA(λ)-R		SARSA(λ)-PT	
	No Episodes	Performance	No Episodes	Performance
Initial	1000	74.1	100	81%
Asymptotic	3900	79.1	2900	86.5%

In the *six actions* task, SARSA(λ)-PT was found to be detrimental to learning. SARSA(λ)-R sped up the initial performance but not the asymptotic performance as explained in section 6.1.1.

Based on these results, we can conclude that when the target task is not very different from the source task, then SARSA(λ)-PT will led to greater speed up in learning the target task compared to SARSA(λ)-R. If the target is significantly different from the source task then SARSA(λ)-PT should not be used, but SARSA(λ)-R can be used in the initial phases of learning and switched off when it no longer leads to improvement in performance.

Given these results, our contribution is in the fact that no comparisons of the two algorithms have been performed previously. These results show when it is preferable to use either of the algorithms and when they might be detrimental to performance.

6.1.2.2.2 *Comparing the policy transfer and the policy reuse algorithms*

Section 5.2.3.3 show that *policy transfer(*SARSA(λ)-PT*)* significantly outperformed *policy reuse* (SARSA(λ)-PR)when these two techniques were used in learning the *three actions* task. Table 6.7 shows the average initial and asymptotic performances of both algorithms.

Table 6.7: Performance of policy reuse and of policy transfer in the three actions task

	Knowledge Transfer Type & Performance	
No of Episodes(time)	Policy transfer	Policy Reuse
100 (0.6hrs)	81%	54.9%
3900(21.7hrs)	86.7%	75.3%

In reusing the *three actions* policy in the *six actions* task, the results achieved were very different from those achieved in reusing the *two actions* task policy in the *three actions* task. Table 6.8 shows a comparison between the performance of *policy reuse* and *policy transfer*. The results show that the performance of *policy transfer* declined with time until the two algorithms had similar performances by episode 7000.

Table 6.8: Performance of policy reuse and of policy transfer with incomplete mapping in the six actions task

	Knowledge Transfer Type & Performance	
No of Episodes(time)	Policy transfer	Policy Reuse
100 (0.56hrs)	76%	63.4%
7000(38.9hrs)	76.2%	76.6%

Fernández et al.(2010), used policy resue in learning the 4vs3 and the 5vs4 keepaway. He stated that the results achieved were similar to those achieved by Taylor et al.(2007). However, in Fernández et al. (2010) the CMACs were used for policy representation while neural networks were used in Taylor et al.(2007).

The initial poorer performance of policy reuse may be attributed to the fact that the target policy is randomly initialised in policy reuse. As the control mechanism uses either the source policy or the target policy for controlling the robot, in the initial phases, when the target policy is in use, the probability of reaching the goal will be low. This makes a case for initialising the target policy using policy transfer, and still continuing to apply policy reuse.

The main contribution of our work is in comparing the two techiques in a simular task and with the same policy representation (neural networks).

The results achieved imply that if the target task is not very different from the source task, it is preferable to use *policy transfer* and use *policy reuse* when the target task is very different from the source task.

The theoretical implication is that as the difference between the source task and the target task increases, *knowledge transfer* by suggesting the right actions to take in observed situations may be preferable to *knowledge transfer* which directs how the target policy parameters are to be initialized.

6.1.2.2.3 ***Comparing the policy transfer algorithm, with the policy transfer plus policy reuse algorithm.***

SARSA(λ)-PT-PR is the algorithm that uses both *policy transfer* and *policy reuse*. In the *three actions* task, section 5.2.3.2 showed that the difference in initial and asymptotic performances between SARSA(λ)-PT and SARSA(λ)-PT-PR were not significant. Table 6.10 summarises the average initial and asymptotic performances of both algorithms. These results are expected because it shows that with the target policy performing so well, SARSA(λ)-PT-PR prefers to use only the target policy (which was initialized with the *two actions* policy), while hardly using the source policy.

Table 6.9: Performance of policy reuse and of policy transfer in the three actions task

	Knowledge Transfer Type & Performance	
No of Episodes(time)	SARSA(λ)-PT	SARSA(λ)-PT-PR
100 (0.6hrs)	81%	77.9%
3900(21.7hrs)	86.7%	86.1%

Table 6.10 summarises the average initial and asymptotic performances of both algorithms in the *six actions* task. In this task, the asymptotic performance of SARSA(λ)-PT is significantly lower than its initial performance (see section 5.3.3.2). This implies that this algorithm is detrimental to learning in this task. SARSA(λ)-PT-PR's asymptotic performance is however higher than its initial performance. This implies that this algorithm can be used for learning in this task.

Table 6.10: Performance of policy reuse and of policy transfer with incomplete mapping in the six actions task

	Knowledge Transfer Type & Performance	
No of Episodes(time)	SARSA(λ)-PT	SARSA(λ)-PT-PR
100 (0.6hrs)	86%	86.4%
7000(38.9hrs)	76.2%	87.9%

The main contribution of our work combining the *policy transfer* and *policy reuse* techniques. This approach has not been used before. We have shown that this combinations can either be expected to performance as well as when *policy transfer* is used alone (in the *three actions* task) or to result in a better performance (in the *six actions* task).

The effect of these results on the learning framework is that in learning the target task, SARSA(λ)-PT-PR can used instead of SARSA(λ)-PT since it has been shown to result in better performance.

6.1.3 *Use of Policy Reuse in learning*

This section sought to answer research question *iii* which states *"Does the use of policy reuse speed up learning in the obstacle avoidance task?"*.

6.1.3.1 Comparing the policy reuse algorithm with the baseline learning algorithm

In order to answer this question the performance of SARSA(λ)-PR was compared with the performance of the baseline algorithm SARSA(λ) in 2 tasks. These were the *three actions* task, and the *six actions* task. In each task, each algorithm was executed in fourteen different scenarios.

In the *three actions* task SARSA(λ) had a significantly better initial performance compared to SARSA(λ)-PR. At the asymptotic level, SARSA(λ)-PR had significantly better performance. Table 6.11 summarises the initial and asymptotic performances of both algorithms. Note that the 900 episodes spent in learning the source task are put into consideration hence the performance of SARSA(λ)-PR after 100 episodes, can only be compared with the performance of SARSA(λ) after 1000 episodes of learning in the 3 actions task.

Table 6.11: Average Initial and asymptotic performances of SARSA(λ) and SARSA(λ)-PR in the three actions task

	SARSA(λ)		SARSA(λ)-PR	
	No Episodes	Performance	No Episodes	Performance
Initial	1000	59.8%	100	54.9%
Asymptotic	3900	71.5%	2900	77.1%

These results imply *policy transfer* does speed up learning in the *three actions* task.

In the *six actions* task, the SARSA(λ)-PR significantly outperformed SARSA(λ) at both the initial and at the asymptotic levels. Table 6.5 illustrates the initial and asymptotic performance of SARSA(λ)-PR and SARSA(λ).

Table 6.12: Average Initial and asymptotic performances of SARSA(λ) and SARSA(λ)-PR in the six actions task

	SARSA(λ)		SARSA(λ)-PR	
	No Episodes	Performance	No Episodes	Performance
Initial	1600	59.5%	100	63.4%
Asymptotic	7000	69%	5500	76.1%

These results imply *policy transfer* does speed up learning in the *six actions* task.

Fernández et al. (2010) showed that *policy reuse* sped up learning in the Keepaway task. Their results showed that SARSA(λ)-PR sped up the initial performance but asymptotically, SARSA(λ) did catch up and overtake SARSA(λ)-PR. However, these authors did not take into consideration the time spent in learning the source task.

The fact that SARSA(λ) has a better initial performance in the *three actions* task than SARSA(λ)-PR might imply that too much time was spent in learning the source task. However we note that despite the lower initial performance, SARSA(λ)-PR catches up and overtakes SARSA(λ) in about 400 episodes.

Our main contribution is in showing that SARSA(λ)-PR speeds up learning in the obstacle avoidance task even when the time spent in learning the source task is put into consideration. The effect of this on the learning framework is that SARSA(λ)-PR should be used in learning the target task in the absence of any other faster algorithms.

6.1.3.2 Comparing the policy reuse algorithm with other learning algorithms

6.1.3.2.1 *Comparing the policy reuse algorithm with the experience replay algorithm*

Any of these two algorithms can be used in learning the target task. By comparing, we can decide which of the two should be used for learning in the target task.

We showed in section 5.2.3.6 that in the *three actions* task SARSA(λ)-R outperformed SARSA(λ)-PR. At the asymptotic level, the performance of both algorithms were not significantly different. Table 6.13 shows the average initial and asymptotic of both algorithms in the 3 actions task.

Table 6.13: Average Initial and asymptotic performances of SARSA(λ)-R and SARSA(λ)-PR in the three actions task

	SARSA(λ)-R		SARSA(λ)-PR	
	No Episodes	Performance	No Episodes	Performance
Initial	1000	74.1	100	54.9%
Asymptotic	3900	79.1	2900	77.1%

In the *six actions* task, SARSA(λ)-PR and SARSA(λ)-R had similar performances at the initial level. At the asymptotic level SARSA(λ)-PR, had a significantly better performance. The initial and asymptotic performances of both algorithms are illustrated in Table 6.14.

Table 6.14: Average Initial and asymptotic performances of SARSA(λ) and SARSA(λ)-PR in the six actions task

	SARSA(λ)-R		SARSA(λ)-PR	
	No Episodes	Performance	No Episodes	Performance
Initial	1600	66.1%	100	63.4%
Asymptotic	7000	61.4%	5500	76.1%

These results imply that as the task gets more complex, then SARSA(λ)-PR can be preferred in learning the target task instead of SARSA(λ)-R. Our contribution is in comparing the performance of these two algorithms which as far as we know has not been attempted before.

6.1.3.2.2 *Comparing the policy reuse, with the policy transfer plus policy reuse algorithm*

In the *three actions* task, SARSA(λ)-PT-PR had significantly higher initial and asymptotic performances compared to SARSA(λ)-PR. Table 6.15 shows the average initial and asymptotic performances of both algorithms.

Table 6.15: Performance of SARSA(λ)-PR and of SARSA(λ)-PT-PR in the three actions task

	Knowledge Transfer Type & Performance	
No of Episodes(time)	SARSA(λ)-PR	SARSA(λ)-PT-PR
100 (0.6hrs)	54.9%	77.9%
3900(21.7hrs)	75.3%	86.1%

In the *six actions* task SARSA(λ)-PT-PR outperformed SARSA(λ)-PR at both the initial and asymptotic levels as illustrated in Table 6.16.

Table 6.16: Performance of SARSA(λ)-PR and of SARSA(λ)-PT-PR in the six actions task

	Knowledge Transfer Type & Performance	
No of Episodes(time)	SARSA(λ)-PR	SARSA(λ)-PT-PR
100 (0.6hrs)	63.4%	86.4%
7000(38.9hrs)	76.6%	87.9%

The main contribution of our work is combining the *policy transfer* and *policy reuse* techniques. This approach has not been used before. We have shown that this combinations can either be expected to perform better than when *policy reuse* is used alone.

The effect of these results on the learning framework is that in learning the target task, SARSA(λ)-PT-PR can used instead of SARSA(λ)-PR in the absence of any other faster algorithm.

6.1.4 *Combining policy transfer and policy reuse in learning*

This section sought to answer research question *iv* which states *"Does combining policy transfer with policy reuse lead to greater speedup than when each of the techniques is used alone?"*.

6.1.4.1 Comparing the combined policy transfer and policy reuse algorithm with the baseline algorithm

In order to answer this question the performance of SARSA(λ)-PT-PR was compared with the performance of the baseline algorithm SARSA(λ) in 2 tasks. These were the *three actions* task, and the *six actions* task. In each task, each algorithm was executed in fourteen different scenarios.

In the *three actions* task, we concluded that SARSA(λ)-PT-PR had a significantly better initial and asymptotic performance compared to SARSA(λ) based on the fact that SARSA(λ)-PT-PR and SARSA(λ)-PT had similar performances and SARSA(λ)-PT had higher initial and asymptotic performances compared to SARSA(λ). Table 6.17 summarise these results.

Table 6.17: Average Initial and asymptotic performances of SARSA(λ) and SARSA(λ)-PT-PR in the three actions task

	SARSA(λ)		SARSA(λ)-PT-PR	
	No Episodes	Performance	No Episodes	Performance
Initial	1000	59.8%	100	77.9%
Asymptotic	3900	71.5%	3000	86.1%

In the *six actions* task, in section 5.3.3.9, we showed that SARSA(λ)-PT-PR significantly outperformed SARSA(λ) at both the initial and at the asymptotic levels. Table 6.18 illustrates the initial and asymptotic performance of SARSA(λ)-PT-PR and SARSA(λ) in this task.

Table 6.18: Average Initial and asymptotic performances of SARSA(λ) and SARSA(λ)-PT-PR in the six actions task

	SARSA(λ)		SARSA(λ)-PT-PR	
	No Episodes	Performance	No Episodes	Performance
Initial	1600	59.5%	100	86.4%
Asymptotic	7000	69%	5500	85.6%

These results imply that the combination of *policy transfer* and *policy reuse* does speedup learning in the *six actions* task.

Our main contribution is in showing that SARSA(λ)-PT-PR speeds up learning in the obstacle avoidance task even when the time spent in learning the source task is put into consideration. The effect of this on the learning framework is that SARSA(λ)-PT-PR should be used in learning the target task in the absence of any other faster algorithms.

6.1.4.2 Comparing SARSA(λ)-PT-PR with other learning algorithms

6.1.4.2.1 *Comparing SARSA(λ)-PT-PR with SARSA(λ)-R*

Any of these two algorithms can be used in learning the target task. By comparing, we can decide which of the two should be used for learning in the target task.

We showed in section 5.2.3.7 that in the *three actions* task, the initial performance of SARSA(λ)-PT-PR was not significantly greater than that of SARSA(λ)-R. The asymptotic performance of SARSA(λ)-PT-PR was however significantly greater than that of SARSA(λ)-R. Table 6.19 shows the average initial and asymptotic of both algorithms in the 3 actions task.

Table 6.19: Average Initial and asymptotic performances of SARSA(λ)-PT-PR and SARSA(λ)-R in the three actions task

	SARSA(λ)-R		SARSA(λ)-PR	
	No Episodes	Performance	No Episodes	Performance
Initial	1000	74.1	100	77.9%
Asymptotic	3900	79.1	2900	85.5%

In the *six actions* task, SARSA(λ)-PT-PR had significantly better performance than SARSA(λ)-R at both the initial and at the asymptotic levels.. The initial and asymptotic performances of both algorithms are illustrated in Table 6.20.

Table 6.20: Average Initial and asymptotic performances of SARSA(λ)-R and SARSA(λ)-PT in the six actions task

	SARSA(λ)-R		SARSA(λ)-PT-PR	
	No Episodes	Performance	No Episodes	Performance
Initial	1600	66.1%	100	86.4%
Asymptotic	7000	61.4%	5500	85.6%

These results imply that as the task gets more complex, the more useful *knowledge transfer* techniques become.

Our contribution is in comparing the performance of these two algorithms which as far as we know has not been attempted before. In addition, SARSA(λ)-PT-PR combines *policy reuse* and *policy transfer*. We have shown in previous sections that this algorithm can lead to significantly better performance than when each of the techniques is used alone.

6.1.5 *Combining experience replay with knowledge reuse*

This section helps in answering research question six *"Does combining knowledge reuse with experience replay lead to greater speedup than when each of the techniques is used alone?"*. In summary, a combination of *experience replay* with *knowledge reuse* seemed to have a negative impact on performance. This especially so when *experience replay* was combined with *policy transfer*. When *policy reuse* was combined with *experience replay*, the initial learning performance improved, but the asymptotic performance declined.

In the reuse of the *two actions* task policy in the *three actions* task, section 5.2.3.8 shows that there was significant decline in performance when *experience replay* was combined with *policy transfer* both at the initial and asymptotic levels. There was however a significant gain in the initial performance when *policy reuse* was combined with *experience replay*. Table 6.21 and Table 6.22 give a summary of the results.

Table 6.21: Results of combining policy transfer with experience replay in the three actions task

	Learning technique & performance	
No of Episodes(time)	Policy transfer	Policy transfer plus experience replay
100 (0.6hrs)	79.3%	72.1%
3900(21.7hrs)	84.6%	71.5%

Table 6.22: Results of combining policy reuse with experience replay in the three actions task

	Learning technique & performance	
No of Episodes(time)	Policy reuse	Policy reuse plus experience replay
100 (0.6hrs)	55%	62.6%
3900(21.7hrs)	75.3%	74.5%

Section 5.3.3.3 also shows that combining *policy transfer* with *experience replay* in reusing the *three actions* task policy in the *six actions* task was detrimental to learning since the initial performance was greater than the asymptotic performance. When *experience replay* was combined with *policy reuse,* there was some significant benefit in performance at the initial level. This benefit was sustained for very short time (1100) episodes after which performance started to decline. Table 6.23 and

Table 6.24 give a summary of the results.

Table 6.23: Results of combining policy transfer with experience replay in the six actions task

	Learning technique & performance	
No of Episodes(time)	Policy transfer	Policy transfer plus experience replay
100 (0.6hrs)	86%	82.6%
7000(38.9hrs)	76.2%	57.2%

Table 6.24 Results of combining policy reuse with experience replay in the six actions task

	Learning technique & performance	
No of Episodes(time)	Policy reuse	Policy reuse plus experience replay
100 (0.6hrs)	63.4%	68.5%
1100(6.1hrs)	73.6%	79.4%
1200(6.7hrs)	74.5%	78.5%
7000(38.9hrs)	76.6%	59.4%

The combination of *experience replay*, *policy reuse* and *policy transfer* was also found to be detrimental to learning in both the *three actions* and the *six actions* task. This is illustrated in Table 6.25 and Table 6.26.

Table 6.25: Results of combining policy transfer policy reuse and experience replay in the three actions task

	Learning technique & performance	
No of Episodes(time)	SARSA(λ)-R-PT-PR	SARSA(λ)-PT-PR
100 (0.6hrs)	77.9%	77%
3900(21.7hrs)	73.3%	86.1%

Table 6.26: Results of combining policy transfer policy reuse and experience replay in the six actions task

	Learning technique & performance	
No of Episodes(time)	SARSA(λ)-R-PT-PR	SARSA(λ)-PT-PR
100 (0.6hrs)	82.6%	86.4%
7000(38.9hrs)	57.2%	87.9%

From these results, we can see that *experience replay* seems to be most beneficial when the target policy has a very poor initial performance. In *policy reuse*, the target policy is randomly initialized. Therefore *experience replay* ensures that it learns the target function very fast. However after the target function has been sufficiently learnt, over fitting seems to set in rather quickly leading to the observed decline in performance. The recommended course of action would be to switch off *experience replay* when sufficient learning has been achieved. This can be detected when the increase in performance plateaus or starts to decline. We are also sufficiently justified in observing that *experience replay* was the only cause of the decrease in performance considering that when *policy reuse* was used in isolation, the decline in performance was not observed.

Based on these results, our contribution is in showing that *policy reuse* and *experience replay* can be combined in learning the target task, but performance should be monitored and *experience replay* switched off when declining performance is observed. Another contribution is in showing that *experience replay* should not be combined with *knowledge transfer* techniques that result in very high initial performances in the target task. This combination results in declining performance due to overfitting.

6.2 Updated Learning Framework

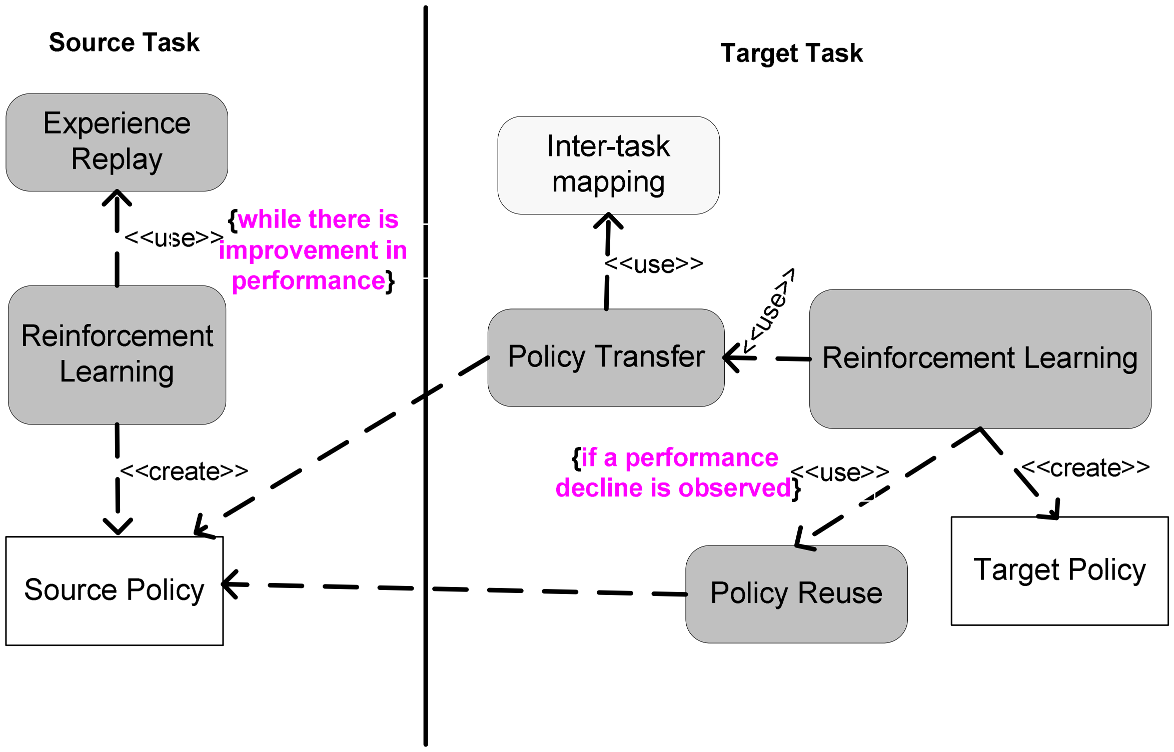


Figure 6.1: Updated Learning Framework

Given the results obtained the learning framework was modified as depicted in Figure 6.1. In the updated framework, the relationships between the techniques are depicted using UML notation for dependencies. The text in between curly braces specifies constraints that must be met for the relationship to remain active. We note the following from the updated learning framework.

i. In the source task, learning will take place using *experience replay*. The performance should be monitored. For instance given *2N* consecutive episodes, say starting from episode *y*, the performance at the end *y+2N* can be compared with the performance at the end of episode *y+N*, where the performance at a given episode is calculated using the results of the current episode and the previous *N-1* episodes. If a decline in performance is observed, then *experience replay* can be terminated. This is based on the observation in the *six actions* task that when *experience replay* is in use, at some point, there is no more improvement in performance. In fact, performance starts to decline. At this point, *experience replay* can be switched off.

This can also mark the point at which learning in the source task terminates considering that very little improvement can be expected in using the baseline algorithm without *experience replay*. For instance in the *six actions* task, SARSA(λ)-R achieves the best performance of 67% after 1400 episodes. SARSA(λ) reaches the same performance after 3000 episodes, and only manages to get to the highest performance of 69.4% after 5300 episodes. More episodes do not result in better performance.

ii. In learning the target task, combining *policy transfer* with *experience replay* was found to lead to declining performance in both the *three actions* task and the *six actions* task. A combination of *policy transfer* and *policy reuse* with *experience replay* was also found to result in declining performances in both the *three actions* and the *six actions* tasks. When *experience replay* was combined with *policy reuse*, there was a speedup in learning in the initial phases of learning. Since we recommend use of either *policy transfer* or a combination of *policy transfer* and *policy reuse* in the target task, *experience replay* should not be used since it was found to lead to declining performance when combined with these two techniques.

iii. In the target task, initially, *policy transfer* should be used. This is because in the *three actions* task, *policy transfer* performed as well as when a combination of *policy transfer* and *policy reuse* was used. Performance should then be monitored (as explained in *i* above) and if a declining performance is observed *policy reuse* should be switched on which leads to use of a combination of *policy transfer* and *policy reuse*. This is because in the *six actions* task, *policy transfer* was found to lead to declining performance. However when *policy transfer* was combined with *policy reuse*, performance did not decline. A combination of the two techniques also performed significantly better than when *policy reuse* was used alone in the *six actions* task.

6.3 Evaluating the updated Learning framework

To evaluate the learning framework, ten trials that involved transfer of knowledge from the 3 actions task to the *six actions* task were setup. Unlike in previous experiments in which one source policy was used to initialise knowledge for the all the *six actions* trials, each trial was setup such that the source task was learnt using experience replay. When a performance decline was experienced in learning the source task using *experience replay*, the source task was terminated and a *six actions* task policy created using the final *three actions* task policy. This realized the *policy transfer* part of the learning framework. The *three actions* task policy was also retained. Learning performance in the target task was then monitored and if a decline in performance was observed, *policy reuse* was switched on and the source policy was used probabilistically in some episodes of the *six actions* task. This implemented the *policy reuse* part of the framework. Table 6.7 summarises the results obtained in the framework validation trials.

Table 6.27: Results attained using an implementation of the validated framework

Description	Value
Average Episode in which source task was terminated	700
Average Performance at end of source task	75.6%
Final Performance	80.2%

These results confirmed our expectations of the learning performance of the algorithm based on the updated learning framework. The final value of 80.2% which is lower than the final value of 87.9% depicted in Table 6.16 for the SARSA(λ)-PT-PR is due to the fact that one source policy which had a very high performance was used for *policy transfer* and *policy reuse* in all previous *six actions* target task trials.

6.4 Achievements

The main objective of this study is to explore the use of *knowledge transfer* and *experience replay* to speed up learning in the obstacle avoidance task. The specific objectives were to:

i. Review of the state of the art in *knowledge transfer*, *experience replay* and application of reinforcement learning in obstacle avoidance;
ii. Propose framework for the application of *knowledge transfer* and *experience replay* in reinforcement learning in the obstacle avoidance task;
iii. Develop a control agent based on the proposed framework and interface the control agent with a robot simulation environment and
iv. Evaluate the performance of the control agent by performing simulations to determine the baseline performance, and the performance when *knowledge transfer* and *experience replay* are applied to the obstacle avoidance task.

v. Update and validate the proposed learning framework

Given the above objectives, the following were the achievements of this study:

1) We extensively reviewed the existing literature on the use of *knowledge transfer* and *experience replay* in speeding up reinforcement learning. Of the several techniques reviewed we found that *policy reuse* and *policy transfer* were the most appropriate for use in inter-domain *knowledge transfer* since they allowed the source task and the target task to come from different domains. We also found that the application of *experience replay* to the obstacle avoidance task was limited to simplified versions of the task. In addition, we found that though *experience replay* had been combined with a form of *knowledge transfer* called teaching, no attempts had been made to combine *experience replay* with either *policy reuse* or *policy transfer*.

2) We created a framework that combined *experience replay* with *policy reuse* and *policy transfer* in learning the obstacle avoidance task. This framework proposed the use of *experience replay* in learning the source task and a combination of *experience replay*, *policy transfer* and *policy reuse* in learning the target task.

3) We developed a control agent that implemented the proposed learning framework. The control agent in addition, implemented other algorithms that resulted from suggested modifications to the learning framework. The control agent was integrated with Microsoft's robotics developer studio simulation environment to enable the testing of control agent algorithms.

4) We tested the algorithms implemented in the control agent in three robot obstacle avoidance task. These tests enabled us to compare the performance of the alternative algorithms. We were then able to refine the proposed learning framework into a final framework that can be used to apply *experience replay*, *policy reuse* and *policy transfer* in learning the obstacle avoidance task.

6.5 Contributions

We have made a number of contributions regarding the application of *policy reuse, policy transfer* and *experience replay* to speed up learning in reinforcement learning. The results of our study can be used in deciding which of these techniques to use, and when to combine them in a reinforcement learning task such as obstacle avoidance. The following is a summary of our key contributions:

6.5.1 *Theoretical Contributions*

1. We developed a reinforcement learning framework that can be used to apply *experience replay*, *policy reuse* and *policy transfer* to speed up learning in the obstacle avoidance task. Though this framework was tested in the obstacle avoidance task, it is expected that it could also be applicable to other similar reinforcement learning tasks. The framework suggest the use of *experience replay* in learning the source task; then initially using *policy transfer* alone in learning the target task; then monitoring the performance, and if a decline in performance is noticed, activating *policy reuse* so that *policy reuse* and *policy transfer* are used together in learning the target task.

2. We compared to *policy reuse* and the *policy transfer* algorithms with both techniques using neural networks as the function approximation technique. We demonstrated the situations when each of the two algorithms can be expected to be most useful in speeding up learning. When the source task and the target task are not very different, then it is advisable to use *policy transfer* rather than *policy reuse* to transfer knowledge from the source task to the target task. We showed that in transferring knowledge from the *two actions* task to the *three actions* task, *policy transfer* outperforms *policy reuse* by far. Using *policy transfer*, the best asymptotic performance in the *three actions* task was seen to be at 86.7% while it stood at 76.7% when *policy reuse* was used. The percentage performance indicates the average number of times the goal was reached.

   When the source task and the target task are very different, then it is advisable to use *policy reuse* to transfer knowledge since it had a consistent improvement in performance in the target task. When *policy transfer* is used, we were able to show that it may lead to an asymptotic decline in performance. In the *policy transfer* from the 3 actions task to the *six actions* task, the performance drops from an initial rate of 86% to an asymptotic rate of 76.5% after 7000 episodes. For *policy reuse*, the initial performance is 63% while the asymptotic performance is 76.6%, showing a consistent improvement from the beginning.

3. We were also able to show that if *policy reuse* and *policy transfer* are combined, they lead to better performance than if any of the techniques is used alone. This was demonstrated in the transfer of knowledge from the *three actions* task to the *six actions* task. When *policy transfer* is combined with *policy reuse*, the initial performance is 86.4% while the asymptotic performance is 87.9%. The asymptotic performance is calculated after 7000 episodes. As observed earlier, *policy transfer* alone leads to declining performance while the best performance achieved using *policy reuse* in the *six actions* task is 76.6% after the same number of episodes.

4. We were also able to show that it may be detrimental to combine *knowledge transfer* with *experience replay*. We showed that when *policy transfer* from the 2 actions task to the *three actions* task is combined with *experience replay* a decline in performance is observed. The initial performance is seen to be 72.1% while the asymptotic performance is 71.5%. The performance is worse than when *policy transfer* is used alone without replay; where the initial performance is 81% and the asymptotic performance is 86.7%. This behavior was also observed in *policy transfer* from the *three actions* task to the *six actions* task, where when *policy transfer* is combined with *experience replay*, the initial performance is 81.9% and the asymptotic performance after 7000 episodes is 57.8%.

   Combining *policy reuse* with *experience replay* on the other hand was shown to result in improvement in the initial performance in the *knowledge transfer* from the 2 actions task to the *three actions* task. The initial performance is observed to be 63% while the asymptotic performance is 75%. When replay is not used, the initial performance is 55% while the asymptotic performance is 75%. For the transfer from the *three actions* task to the *six actions* task, combining *policy reuse* with *experience replay* shows an improvement in the initial performance but this improvement is very short lived, after which performance drops drastically. The initial performance is observed to be 69%. After the first 1000 episodes, this jumps to 79%. The performance then starts to decline until it reaches 59% after 7000 episodes. Hence showing the need to actively monitor if the *experience replay* is beneficial and to have it automatically switched off it is not.

5. We were also able to show that *experience replay* leads to the greatest improvement in performance when the policy being used is performing very poorly. As the performance improves, *experience replay* was shown to sometimes lead to a decline in performance. For instance in the *six actions* task, we showed that the technique that did not use *experience replay* asymptotically overtook the technique using *experience replay* in performance. The initial and final performances using the SARSA(λ) algorithm without either *experience replay* or *knowledge transfer* are seen to be 23.6% and 69% respectively. When *experience replay* is added, the initial and final performances are 40.6% and 61.4% respectively. This once again showed the need to monitor performance when *experience replay* is in use and to disable it or modify its parameters if a decline in performance is noticed. These observations are applicable even when *experience replay* is combined with other techniques, for instance *policy reuse*.

6.5.2 *Methodological Contributions*

1. In comparing the learning performance of algorithms that use *knowledge transfer* with those that do not use *knowledge transfer*, we put the time (number of episodes) spent in learning the source task into consideration. For an algorithm that did not use *knowledge transfer*, we ensured that it had spent *x*+*y* episodes in learning the target task for it to be compared with an algorithm that had spent *x* episodes in learning the target task and *y* episodes in learning the target task. We therefore ensured that no algorithm had spent more time than the other in learning.

2. Since the experiments were performed using simulations, we used test scenarios to create a variety of situations of different complexity on which to test a given algorithm. For instance in the *two actions* tasks, we had fourteen scenarios. A given algorithm was tested on all of these scenarios and the performances on all the scenarios were combined to obtain the average performance of the algorithm. A scenario was defined using random number seeds which determined things like: the way the neural network weights were initialized, the selection of random starting positions, the selection of the orientation of the robot at the start of an episode, the decision on if to use the optimal action or a random action, the decision on if to use

the source policy or the target policy. This ensured that if algorithms A and B were compared, in scenario 1, they used the same set of random number streams, in scenario two they also used another set of random number streams which were the same for the two algorithms but different from the streams used in scenario one, and so forth.

6.5.3 *Technical Contributions*

1. We also created a system that can be used to test other reinforcement learning algorithms. The system consists mainly of the *RobotInterfaceService* and the *ControlAgentService*. The *RobotInterfaceService* receives action commands from the *ControlAgentService* and sends them the robot in the simulation environment. It also receives sensor information from the simulated environment, which it processes to get the environmental state which it then forwards to the *ControlAgentService*.

   The *ControlAgentService* contains the *ControlPolicy.* After receiving state information from the *RobotInterfaceService*, the *ControlAgentService* decides on the next action to be performed using the *ControlPolicy*. It also decides when learning should take place in the *ControlPolicy*.

   To test a new algorithm, minimal changes only need to be made to the *ControlPolicy* and the parts of the *ControlAgentService* responsible for initiating learning in the *ControlPolicy*.

6.6 Limitations

1) In the transfer from *three actions* to *six actions*, Figure 5.3 shows an asymptotic decline in performance. It would have been interesting to run these experiments for longer to determine if this trend would change and after how many training episodes. This would help us determine if to completely rule out the use of *policy transfer* as the target task becomes more complex.

2) We have observed the trends when knowledge is transferred from a *two actions* task to a *three actions* task, and from a *three actions* task to a *six actions* task. Other combinations are a possible, for instance from a *three actions* task to a *four actions* task. Or from a *six actions* task to a *nine actions* task or *twelve actions* task. These other combinations would have

helped to illuminate more on the behavior of *policy reuse* and *policy transfer* and under what conditions they are likely to be most beneficial

3) Figure 5.3 shows that asymptotically *experience replay* may result in a decline in performance. Since this is attributed to overfitting, it would have been interesting to test if asymptotic reduction in the number of experiences replayed in a replay session, and a reduction in the frequency *of experience replay* would have ameliorated this problem.

6.7 Further Work

1) Despite the fact that neural networks are able to generalize well when the number of states is huge or infinite, the huge state space makes learning much harder. Clustering states in the state space so that the agent will only need to learn the values for the clusters may lead to faster performance. The early learning episodes may be spent learning the clusters with some mechanism to ensure that as much of the state space is sampled so that the clusters formed are a good representation of the state space.

2) Clustering may also be used to attain model based learning. This will allow the selection of not only the best action, but a prediction of the next state and its value.

3) In some situations, unlearning (Cetina, 2008) was found to occur in that the agent stops taking optimal actions in some situations where it was previous taking optimal actions. This occurs because changes in neural network weights may adversely affect previous better settings for some states. This is an instance of what is usually referred to stability-instability dilemma (Rajasekaran & Vijayalakshmi, 2003) where the agent needs to be stable enough to preserve learnt knowledge, but plastic enough to acquire new knowledge. This limitation can be addressed if clusters can be formed from which rules that constitute a form of knowledge with less flux can be derived.

4) We used eleven variables to represent the position of obstacles, that is 5 variables for positions in the left, 5 variables for positions on the right, and one variable for straight ahead position. The choice of these setting was guided by convenience. More work can be done to determine if it would be preferable to have more or less position variables. The same case can be made for the goal position since we had 11 variables to represent the possible positions of the goal object.

5) It has been observed that *experience replay* is mostly beneficial when the performance of the control agent policy is very poor for instance in the initial stages of learning. After the performance of the policy exceeds certain levels (depending on the task), use of *experience replay* was shown to lead to a decline in performance. This may imply that after some performance levels are attained, or when the performance starts to decline, *experience replay* can be disabled. There is need therefore to investigate if stopping *experience replay* results in a resumption of performance improvement. This can also be compared to other techniques for instance reduction of the number of *experiences replay*ed in a replay session.

6) It has been shown that *policy transfer* can lead to deterioration in performance in the target task. This was shown in the transfer of the 3 actions task policy to the *six actions* task policy. Since this has been shown to occur in the obstacle avoidance task, it would be interesting to see of this result can be replicated in other tasks e.g. the *Keepaway* task.

7) The learning framework was only evaluated in the obstacle avoidance task and specifically by changing the number of actions in the task. The framework can be tested in a case where for instance the state space is changed. It can also be evaluated in other reinforcement learning tasks.

8) In chapter four we observed that the control agent received much more negative rewards for colliding with obstacles than for being time barred. This meant that it learnt how to move towards the goal but it had trouble learning how to avoid obstacles. In our setup, the agent received a reward of -1 for being time barred and -1 for colliding with obstacles. Since it has a harder time learning how to avoid obstacles, the two rewards can be assigned different values say -0.5 for time barred and -1 for obstacle collision.

6.8 Conclusion

This work investigated *experience replay*, in combination with two methods of *knowledge reuse*, that is *policy reuse* and *policy transfer* as applied in reinforcement learning. We found that they can be combined in various ways to improve performance. For instance, in learning the source task, *experience replay* should be used but the performance should be actively monitored and when it starts to deteriorate, then *experience replay* can be stopped or reduced to avoid overfitting. In learning the target task however, if *knowledge transfer* techniques are used, then they should not be combined with *experience replay* as this leads to worse performance than when *knowledge transfer* is used alone. In *knowledge transfer policy transfer* should be used alone at first, and if a declining performance is observed, *policy reuse* should be switched on so that both techniques are used together. This helps to deal with the problem of poor initial performance in *policy reuse*, and also deals with the problem of declining performance observed *in policy transfer*. These results are applicable in domains like mobile robot control where learning can start with a simpler task for instance, a task containing less actions or states. After learning has occurred, the knowledge can be transferred to a task with more actions and / or states. They are also applicable in the games domain where a game can be simplified to create a simpler version. After learning the simpler version, the knowledge can be reused in learning the full game.

References

Adam, S., Busoniu, L. & Babuska, R., 2012. Experience Replay for Real-Time Reinforcement Learning Control. *IEEE Transactions on Systems, Man, And Cybernetics—Part C : Applications and Reviews*, 42(2), pp.201- 212.

Aggarwal, K.K. & Singh, Y., 2008. *Software Engineering*. 3rd ed. New Delhi: New Age International (P) Ltd.

Alam, K.M.R., Huda, M.S., Al-asad, M.M.H. & Zaman, M.M., 2004. A Comparison of Different Constructive Algorithms to Design Neural Networks. In *3rd International Conference on Electrical & Computer Engineering*. Dhaka, Bangladesh, 2004. [Online].Available at: http://www.buet.ac.bd/icece/pub2004/P059.pdf.[Accessed Jan,2014].

Al-Jumaily, A. & Leung, C., 2005. Wavefront Propagation and Fuzzy Based Autonomous Navigation. *International Journal of Advanced Robotic Systems*, 2. Online. Available at [http://arxiv.org/ftp/cs/papers/0601/0601053.pdf]. [Accessed 21 Aug 2013].

Asada, M., Noda, S., Tawaratsumida, S. & Hosoda, K., 1994. Vision-Based Behavior Acquisition for a Shooting robot by Using a Reinforcement Learning. In *IAPR/IEEE Workshop on Visial Behaviours*., 1994.

Bergamo, Y.P., Matos, T., Silva, V.F.d. & Costa, A.H.R., n.d. *Accelerating reinforcement learning by reusing abstract policies*. [Online] Available at: http://www.each.usp.br/valdinei/Papers/enia2011.pdf [Accessed Jan 2015].

Bernstein, D.S., 1999. *Reusing old Policies to Accelerate Learning on new MDPs*. Technical Report. University of Massachusetts. Available through: CiteSeer[Accessed 5th Jan 2015].

Brafman, R.I. & Tennenholtz, M., 2002. R-max - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning. *Journal of Machine Learning*, pp.213-31. [online] Available at: http://jmlr.org/papers/v3/brafman02a.html. [Accessed 2 may 2012].

Bruch, M.H., Lum, J., Yee, S. & Tran, N., 2005. Advances in autonomy for small UGVs. In Gerhart, G.R., Shoemaker, C.M. & Gage, D.W., eds. *SPIE: Unmanned Ground Vehicle Technology VII*. Orlando, Florida, USA , 2005. Available through: CiteSeerX. [Accessed March, 2014].

Carr, M.F., Jadhav, S.P. & Frank, L.M., 2011. Hippocampal replay in the awake state: a potential substrate for memory consolidation and retrieval. *Nature Neuroscience*, 14(2). Online. Available From: http://www.nature.com/neuro/journal. [Accessed 19 Aug 2013].

Cetina, V.U., 2008. Multilayer Perceptrons with Radial Basis Functions as Value Functions in Reinforcement Learning. In *European Symposium on Artificial Neural Networks - Advances in Computational Intelligence and Learning*. Bruges (Belgium), 2008. Available Through: CiteSeerX. [Accessed May 2013].

Conn, K. & Peters, R.A., n.d. Reinforcement Learning with a Supervisor for a Mobile Robot in a Real-World Environment.

Cyberbotics, 2014. *Features*. [Online] Available at: http://www.cyberbotics.com/features [Accessed February 2014].

Cyberbotics, 2014. *Webots overview*. [Online] Available at: http://www.cyberbotics.com/overview [Accessed February 2014].

Cyberbotics, 2014. *Webots User Guide*. Avaliable Online. [Lass Accessed: February 2014].

DARPA DRC Team, 2013. *DARPA Robotics Challenge DRC Trials 2013*. DARPA. Online. Available. [Accessed 27 Aug 2013].

Denscombe, M., 1999. *The Good Research Guide for Small-Scale Social Research Projects*. New Delhi: Vinod Vasishtha.

Dietterich, T.G., 1998. Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms. *Neural Computation*, 10, pp.1895-923. Avaliable through: CiteSeer [Accessed 10 Jan 2014].

Dietterich, T.G., 1998. The MAXQ Method for Hierarchical Reinforcement Learning. In *Fifteenth International Conference on Machine Learning*., 1998. Morgan Kaufmann. [Onlinee] [http://citeseerx.ist.psu.edu] [Accessed 2 Aug 2013].

Drummond, C., 2002. Accelerating reinforcement learning by composing solutions of automatically identified subtasks. *Journal of Artificial Intelligence Research*, 16, pp.59-104.

Erez, T. & Smart, W.D., 2008. What does Shaping Mean for Computational Reinforcement Learning? In *7th IEEE International Conference on Development and Learning*. Monterey, Canada, 2008. Online. Availabe at : [Accessed 02 Sept 2013].

Erni, T. & Dietz, V., 2001. Obstacle avoidance during human walking: learning rate and cross-modal transfer. *Journal of Physiology*, 534(1), pp.303-12. Online. Availale From: The Journal of Physiology. [Accessed 23 Aug 2013].

Fahlman, S.E. & Lebiere, C., 1990. The Cascade-Correlation Learning Architecture. In *Advances in Neural Information Processing Systems 2*. San Francisco, CA, 1990. Available through: CiteSeerX. [Accessed June 2012].

Fernández, F., García, J. & Veloso, M., 2010. Probabilistic Policy Reuse for inter-task transfer learning. *Robotics and Autonomous Systems*, 58(7), pp.866-71. Available Through: CiteSeer [Accessed September 2011].

Fernandez, F. & Veloso, M., 2006. Probabilistic Policy Reuse in a Reinforcement Learning Agent. In *Fifth International Joint Conference on Autonomous Agents And Multiagent Systems*. Hakodate, Hokkaido, Japan, 2006. Available Through: CiteSeer [Accessed September 2011].

Frommberger, L., 2010. *Qualitative Spatial Abstraction in Reinforcement Learning*. Berlin Heidelberg: Springer-Verlag.

gazebosim.org, 2013. *Tutorials/1.0/plugins/overview*. [Online] Available at: http://gazebosim.org/wiki/Tutorials/1.0/plugins/overview [Accessed February 2014].

gazebosim.org, ND. *About Gazebo*. [Online] Available at: http://gazebosim.org/about.html [Accessed February 2014].

Ghezzi, C., Jazayeri, M. & Mandrioli, D., 2003. *Fundamentals of Software Engineering*. 2nd ed. New Dehli: Prentice-Hall of India.

Hira, D.S., 2001. *System Simulation*. New Delhi: S. Chand and Company Ltd.

Hogg, R.W. et al., 2001. Sensors and Algorithms for Small Robot Leader/Follower Behavior. In *SPIE 4364, Unmanned Ground Vehicle Technology III.*, 2001. [Online] Available< http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=912531>. [Accessed 24 Aug 2013].

Huang, B.-Q., Ciao, G.-Y. & Guo, M., 2005. Reinforcement Learning Neural Network to the Problem of Autonomous Mobile Robot Obstacle Avoidance. In *Fourth International Conference on Machine Learning and Cybernetics*. Guangzhou, China, 2005. [Online]. Avaiable at:http://www.ice.ci.ritsumei.ac.jp/~ruck/CLASSES/INTELISYS/NN-Q.pdf. [Accessed March 2013].

Kaelbling, L.P., Littman, M.L. & Moore, A.W., 1996. Reinforcement Learning: A survey;. *Journal of Artficial Intelligence Research*, 4, pp.237-85.

Kalyanakrishnan, S. & Stone, P., 2007. Batch Reinforcement Learning in a Complex Domain. In *The 6th International Conference on Autonomous Agents and Multiagent Systems*. New York, USA, 2007.

Knudson, M. & Tumer, K., 2012. Efficient State Spaces and Policy Transfer for Robot Navigation. In L. Garcia, ed. *Advanced Robotics*. Concept Press. [Online] Available at. [Accessed 20 Aug 2013].

Koenig, N., 2014. *Gazebo supports 4 physics engines*. [Online] Available at: http://gazebosim.org/news/gazebo-supports-4-physics-engines.html [Accessed February 2014].

Lam, T. & Dietz, V., 2004. Transfer of Motor Performance in an Obstacle Avoidance Task to Different Walking Conditions. *Journal of Neurophysiology*, 92, pp.2010-16. Online. Available From: The Fournal of Physiology. [Accessed: 20th Aug 2013].

Laud, A.D., 2004. *Theory and Application of Rewad Shaping in Reinforcement Learning*. PhD Thesis. University of Illinois at Urbana-Champaign.

Leffler, B.R., 2009. *Perception-Based Generalization in Model-Based Reinforcement Learning*. PhD Thesis. Graduate School, New Brunswick Rutgers, The State University of New Jersey.

Lin, L.-J., 1991. Programming robots using reinforcement learning and teaching. In *AAA1*., 1991.

Lin, L.-J., 1992. Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching. In *Machine Learning*. Boston: Kluwer Academic Publishers. pp.293-321. Online. Available From: CiteSeer. [Accessed 30 Aug 2013].

Maček, K., Petrović, I. & Perić, N., 2002. A reinforcement learning approach to obstacle avoidance of mobile robots. In *7th International Workshop on Advanced Motion Control*., 2002. Available through: IEEEEXplore. [Accessed June 2013].

Mall, R., 2003. *Fundamentals of Software Engineering*. 2nd ed. New Delhi: Prentice-Hall of India.

Mario, E.D., Talebpour, Z. & Martinoli, A., 2013. A Comparison of PSO and Reinforcement Learning for Multi-Robot Obstacle Avoidance. In *2013 IEEE Congress on Evolutionary Computation*. CANCUN, 2013. Online. Available From: IEEE Xplore Digital Library. [Accessed 27 Aug 2013].

McClellad, J.L., 2013. *Explorations in Parallel Distributed Processing: A Handbook of Models,Programs, and Exercises*. Available Online. Last Accessed Jan, 2014.

McDonald, J.H., 2014. *Handbook of Biological Statistics*. 3rd ed. Baltimore, Maryland: Sparky House Publishing. [online].Avaliable At [Accessed Sept 2014].

Menache, I., Mannor, S. & Shimkin, N., 2005. Basis Function Adaptation in Temporal Difference Reinforcement Learning. *Annals of Operations Research*, 134, pp.215-38. Available through: CiteSeerX. [Accessed Jan 2013].

Michael, O., 2004. Webots: Professional Mobile Robot Simulation. *International Journal of Advaced Robotic Systems*, 1, pp.39-42. Available Online. [Last Accessed: 14/01/2014].

Michels, J., Saxena, A. & Ng, A.Y., 2005. High Speed Obstacle Avoidance using Monocular Vision and Reinforcement Learning. In *22nd International Conference on Machine Learning*. Bonn, Germany, 2005.

Microsoft, 2011. *Robotics Developer Studio, Getting Started*.

Microsoft, 2012. *DSS Introduction*. [Online] Available at: http://msdn.microsoft.com/en-us/library/bb483056.aspx [Accessed February 2014].

Miriti, E., Waiganjo, P. & Mwaura, A., 2013. Comparing Action as Input and Action as Output in a Reinforcement Learning Task. *International Journal of Computer Applications*, 76(1), pp.24-28. [online]. Available at: http://www.ijcaonline.org/archives/volume76/number1/13212-0593. [Accessed Jan 2014].

Mitchel, T., 1997. *Machine Learning*. Singapore: McGraw-Hill.

Moreno, R.A., 2007. *Simulated Pioneer 3DX Bumper*. [Online] Available at: http://www.conscious-robots.com/en/robotics-studio/robotics-studio-services/simulated-pioneer-3dx-b.html [Accessed 6 Dec 2013].

National Institute of Mental Health, 2012. *Awake Mental Replay of Past Experiences Critical for Learning*. [Online] NIMH Available at: http://www.nimh.nih.gov/news/science-news/2012/awake-mental-replay-of-past-experiences-critical-for-learning.shtml [Accessed 20 Sept 2013].

Nawi, N.M. et al., 2009. The Effect of Gain Variation In Improving Learning Speed Of Back Propagation Neural Network Algorithm on Classification Problems. In *Symposium on Progress in Information and Communication Technologies (SPICT'09)*., 2009. Available through: CiteSeer [Accessed, Dec 2013].

Ng, A.Y. & Jordan, M.I., 2000. PEGASUS: A policy search method for large MDPs and POMDPs. In *Sixteenth Conference on Uncertainty in Artificial Intelligence*. Stanford, California, 2000. Available Online. [Last Accessed 5th Dec 2013].

Nilsson, N.J., 1998. *Artificial Intelligence: A New Synthesis*. New Delhi: Elsevier.

Palmisano, J.S., ND. *Programming - Robot Simulation*. [Online] Available at: http://www.societyofrobots.com/programming_robot_simulation.shtml [Accessed February 2014].

Papierok, S., Noglik, A. & Pauli, J., 2008. 1st International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems. Patras, Greece, 2008. [Online]. Available at: http://www.is.uni-due.de/fileadmin/literatur/publikation/papierok08erlars.pdf. [Last Accessed Jan 2014].

Perros, H., 2009. *Computer Simulation Techniques: The Definitive Introduction*. Available online. [Last Accessed May 2013].

Peshkin, L., 2001. *Reinforcement Learning by Policy Search*. PhD Thesis. MIT.

Pfeifer, R., 1997. Design principles for autonomous agents: a case study of classification. *Artificial Life Robotics*, pp.43-46. Available From CiteSeerX.[Last Accessed: 03-02-2014].

Price, B. & Boutilier, C., 2003. Accelerating Reinforcement Learning through Implicit Imitation. *Journal of Artificial Intelligence Research*, 9, pp.569-629. Available Online. [Last Accessed 15 Nov 203].

Price, B. & Boutilier, C., 2003. Accelerating Reinforcement Learning trhough Implicit Imitation. *Journal of Artificial Intelligence Research*, 19, pp.569-629.

Rajasekaran, S. & Vijayalakshmi, G.A.P., 2003. *Neural Networks, Fuzzy Logic, and Genetic Algorithms: Synthesis and Applications*. New Delhi: Prentice-Hall of India.

Razali, N.M. & Wah, Y.B., 2011. Power Comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. *Journal of Statistical Modeling and Analytics*, 2(1), pp.21-33. [Online]. Available from [Last Accessed Sept 2014].

Russell, S. & Norvig, P., 2003. *Artificial intelligence: A modern Approach*. Singapore: Pearson Education Inc.

Schach, S.R., 2004. *Introduction to Object-Oriented Analysis and Design with UML and the Unified Process*. New Delhi: Tata McGraw-Hill.

Sehad, S. & Touzet, C., 1994. Self-organizing Map for Reinforcement Learning: Obstacle-Avoidance with Khepera. In *From Perception to Action Conference*. Laussane, Switzerland, 1994.

Selfridge, O.G., Sutton, R.S. & Barto, A.G., 1985. Training and Tracking in Robotics. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*., 1985.

Shapiro, S.S. & Wilk, M.B., 1965. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4), pp.591-611. Available Through: JSTOR [Accessed Sept 2014].

Sherstov, A.A. & Stone, P., 2005. Function Approximation via Tile Coding: Automating Parameter Choice. In *the 6th international conference on Abstraction, Reformulation, and Approximation*., 2005. [Online]. Available at http://www.cs.ucla.edu/~sherstov/pdf/sara05-tiling.pdf. [Last accessed June 2013].

Singh, S.P., 1992. Transfer of Learning by Composing Solutions of Elemental Sequential Tasks. In *Machine Learning*. Online. Available From CiteSeer. [Accessed 02 Sept 2013].

Singh, S., Barto, A.G. & Chentanez, N., 2004. Intrisically Motivated Reinforcement Learning. In *18th Annual Conference on Neural Information Processing Systems*. Vancouver, B.C., Canada, 2004.

Skelly, M.M., 2004. *Hierarchical Reinforcement Learning With Function Approximation for Adaptive Control*. PhD Thesis. Cleveland: Case Western Reserve University.

Smart, W.D. & Kaelbling, L.P., 2002. Effective Reinforcement Learning for Mobile Robots. In *IEEE International Conference on Robotics and Automation*. Washington, D.C, 2002. Online. Available From IEEE Xplore Digital Library. [Accessed 26 Aug 2013].

Smart, W.D. & Kaelbling, L.P., n.d. Effective Reinforcement Learning for Mobile Robots.

Sommerville, I., 1992. *Software Engineering*. 4th ed. Addison-Wesley Publishing Company.

Stangroom, J., 2014. *Wilcoxon Signed-Rank Test Calculator*. [Online] Available at: http://www.socscistatistics.com/tests/signedranks/Default2.aspx [Accessed 12 September 2014].

Stone, P. & Sutton, R.S., 2001. Scaling Reinforcement Learning toward RoboCup Soccer. In *The Eighteenth International Conference on Machine Learning*. Williamstown, MA, 2001. Available Through: CiteSeerX. [Accessed 21 May 2013].

Sutton, R.S., 1991. Reinforcement Learning Architectures for Animats. In *The First International Conference on Simulation of Adaptive Behavior: From Animals to Animats*., 1991. Online. Available from. [Accessed 26 Aug 2013].

Sutton, S.R. & Barto, G.A., 1998. *Reinforcement Learning: An Introduction*. London: MIT Press.

Taylor, M.E. & Stone, P., 2005. Behavior Transfer for Value-Function-Based Reinforcement Learning. In *Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*. Utrecht, The Netherlands, 2005.

Taylor, M.E. & Stone, P., 2009. Transfer learning for Reinforcement Learning Domains: A survey. *Journal of Machine Learning*, 10, pp.1633-85.

Taylor, M.E. & Stone, P., July 2005. Behavior Transfer for Value-Function-Based Reinforcement Learning. In *Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*. Utrecht, The Netherlands, July 2005.

Taylor, M.E., Whiteson, S. & stone, p., 2007. Transfer via InterTask Mappings in Policy Search Reinforcement Learning. In *The Autonomous Agents and Multi-Agent Systems Conference*. Honolulu, Hawaii, 2007.

Tesauro, G., 1992. Practical Issues in Temporal Difference Learning. Kluwer Academic Publishers. Available from CiteSeer [Last Accessed 28 Nov 2013].

Tesauro, G., 1995. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38(3).

Thrun, S., 1995. An approach to learning mobile robot navigation. *Robotics and Autonomous Systems*, 15, pp.301-19. Available Through: CiteSeer[Accessed Jan 2013].

Torrey, L., Walker, T., Shavlik1, J. & Maclin, R., 2005. Using Advice to Transfer Knowledge Acquired in One Reinforcement Learning Task to Another. In *Sixteenth European Conference on Machine Learning*. Porto, Portugal, 2005. Available From CiteSeerX. [Last Accessed 15 Jan 2014].

Touzet, C., 1997. Neural Reinforcement Learning for Behaviour Synthesis. *Robotics and Autonomous Systems*, 22, pp.251-81. Online. Available Through: CiteSeer [Accessed 23 Aug 2013].

Touzet, C., 1997. Neural Reinforcement Learning for Behaviour Synthesis. *Robotics and Autonomous Systems*, 22, pp.251-81. Available Through: CiteseerX. [Accessed Jan 2013].

Usher, K., 2006. Obstacle avoidance for a non-holonomic vehicle using occupancy grids. In MacDonald, B., ed. *Conference on Robotics and Automation*. Auckland, Newzealand, 2006.

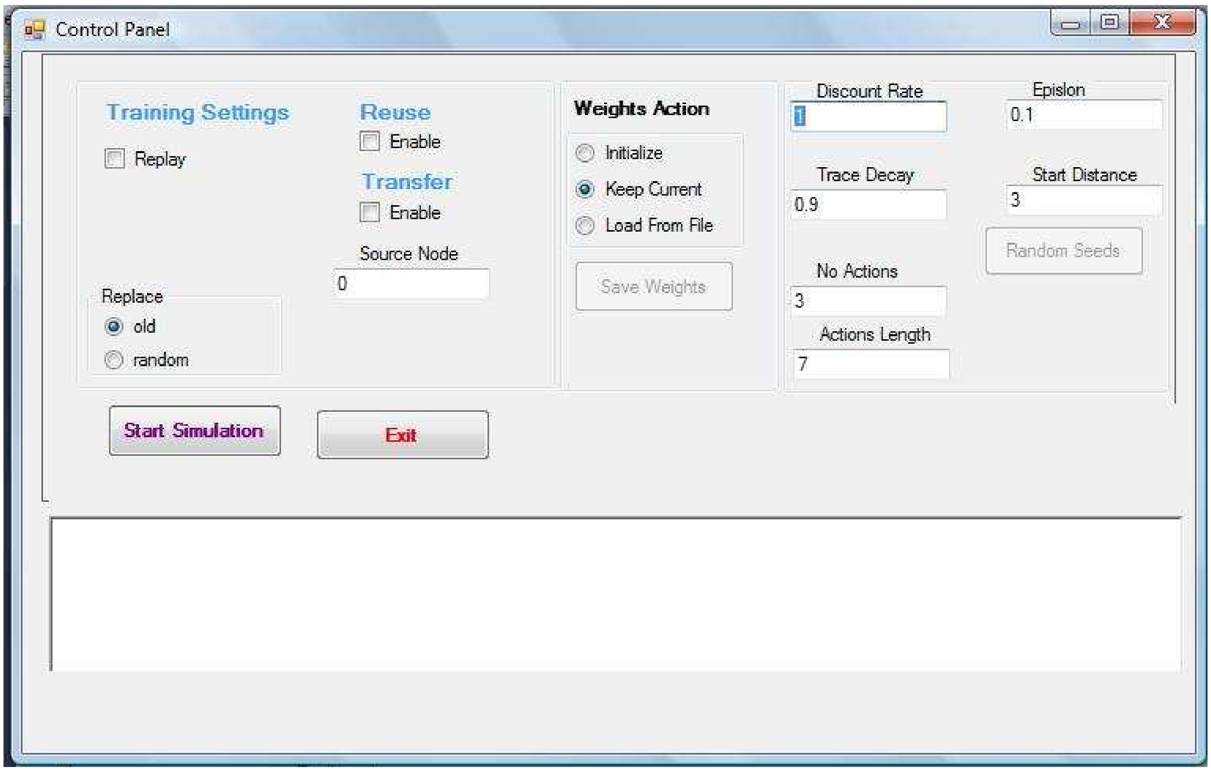
Wamsley, E.J. et al., 2010. Cognitive Replay of Visuomotor Learning at Sleep Onset: Temporal Dynamics and Relationship to Task Performance. *Sleep*, 33(1). Online. Available from: http://www.journalsleep.org/. [Accessed 20 Aug 2013].

Weinberger, S., 2012. *Next generation military robots have minds of their own*. [Online] Available at: http://www.bbc.com/future/story/20120928-battle-bots-think-for-themselves [Accessed September 2012].

Wettschereck, D. & Dietterich, T., 1992. Improving the Performance of Radial Basis Function Networks by Learning Center Locations. In *Advances in Neural Processing Systems 4*. San Mateo, 1992. Available through: CiteseerX. [Accessed June 2013].

Zhai, Z., Chen, W., Li, X. & Guo, J., 2009. A Modified Average Reward Reinforcement Learning Based on Fuzzy Reward Function. In *International MultiConference of Engineers and Computer Scientists*. Hong Kong, 2009.

Appendix 1: System User Interface



Control Panel

Training Settings

- [ ] Replay

Replace
- (●) old
- ( ) random

Reuse
- [ ] Enable

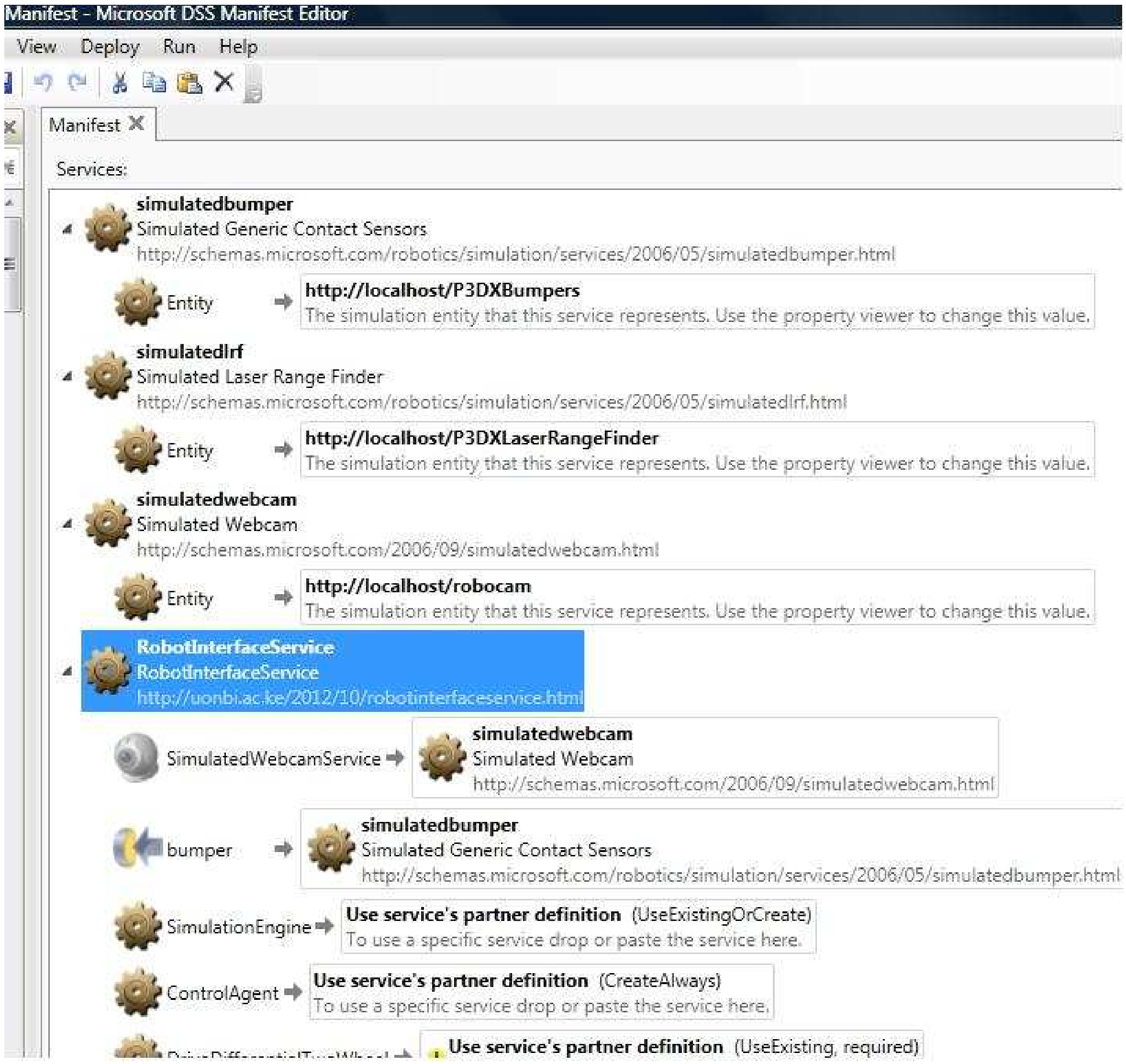
Transfer
- [ ] Enable

Source Node: 0

Weights Action
- ( ) Initialize
- (●) Keep Current
- ( ) Load From File

Discount Rate	1
Trace Decay	0.9
No Actions	3
Actions Length	7
Epislon	0.1
Start Distance	3

Appendix 2: Composition of Services


Manifest - Microsoft DSS Manifest Editor

View Deploy Run Help

Manifest

Services:

simulatedbumper
Simulated Generic Contact Sensors
http://schemas.microsoft.com/robotics/simulation/services/2006/05/simulatedbumper.html

Entity → **http://localhost/P3DXBumpers**
The simulation entity that this service represents. Use the property viewer to change this value.

simulatedlrf
Simulated Laser Range Finder
http://schemas.microsoft.com/robotics/simulation/services/2006/05/simulatedlrf.html

Entity → **http://localhost/P3DXLaserRangeFinder**
The simulation entity that this service represents. Use the property viewer to change this value.

simulatedwebcam
Simulated Webcam
http://schemas.microsoft.com/2006/09/simulatedwebcam.html

Entity → **http://localhost/robocam**
The simulation entity that this service represents. Use the property viewer to change this value.

RobotInterfaceService
RobotInterfaceService
http://uonbi.ac.ke/2012/10/robotinterfaceservice.html

SimulatedWebcamService → **simulatedwebcam**
Simulated Webcam
http://schemas.microsoft.com/2006/09/simulatedwebcam.html

bumper → **simulatedbumper**
Simulated Generic Contact Sensors
http://schemas.microsoft.com/robotics/simulation/services/2006/05/simulatedbumper.html

SimulationEngine → **Use service's partner definition** (UseExistingOrCreate)
To use a specific service drop or paste the service here.

ControlAgent → **Use service's partner definition** (CreateAlways)
To use a specific service drop or paste the service here.

Use service's partner definition (UseExisting, required)

The Microsoft DSS Manifest Editor is used to put together a group of services that will run together to provide the required functionality.

Appendix 3: Sample Code

Getting a random start position

```
private void computeRobotPosition()
        {
            try
            {

                    int index = randomDistance.Next(noPositions);
                    _state._robotPosition.X = positions3M[index, 0];
                    _state._robotPosition.Z = positions3M[index, 1];

            }
            catch (Exception e)
            {
                Console.Write("\n" + e.ToString() +" " +positions3M.Length);
            }
        }
```

Set of start positions

```
float[,] positions3M = { { 3.4f, -8 }, { 1, -6 }, { 3.5f, -10f }, { -2f, -10.7f }, { -
2.2f, -7.5f }, { 3.2f, -10.5f }, { 2f, -11.6f }, { 1, -12 }, { 0.5f, -12 }, { 0, -12 }
};
goalPosition={0.5f,-9}
```

Set of actions

```
double[,] actions = { { 0.1, 0.1 }, { 0.1, 0 }, { 0, 0.1 }, { 0.2, 0.2 }, { 0.2, 0.1
}, { 0.1, 0.2 }, { 0, 0 } }; //two values per entry. left wheel power and right wheel
power
```

Selecting the action in the control policy

```
public int selectActionEGreedy(decimal[] state, double epislon, Random rProb, Random
rAction, int noActions)
        {

            // Console.Write("\nInputs {0} \n", state[0]);
            int action = -1;
            decimal max = decimal.MinValue;
            int i;

            try
            {

                if (epislon > 0)
                {
                     //select a random action
                    if (rProb.NextDouble() < epislon)
                    {
                        action = rAction.Next(0, noActions);
                        return action;
                    }
                }
```

```
      //else select the action with the greatest value for the given state
        calculateOutputs(state);
        for (i = 0; i < noOutputs; i++)
        {

            if (outputNodes[i].output > max)
            {
                max = outputNodes[i].output;
                action = i;
            }
            Console.Write("\n{0}: {1}", i, outputNodes[i].output);
        }
        //action = state[0] == 1 ? 0 : 1;
    }
    catch (Exception e)
    {
        Console.Write("\n" + e.ToString());
    }
    return action;

}
```