Reinforcement Learning for Qualitative Group Behaviours Applied to Non-player Computer Game Characters

Jay Bradley

Doctor of Philosophy

Institute of Perception, Action and Behaviour
School of Informatics
University of Edinburgh

2010

Abstract

This thesis investigates how to train the increasingly large cast of characters in modern commercial computer games.

Modern computer games can contain hundreds or sometimes thousands of non-player characters that each should act coherently in complex dynamic worlds, and engage appropriately with other non-player characters and human players. Too often, it is obvious that computer controlled characters are *brainless zombies* portraying the same repetitive hand-coded behaviour. Commercial computer games would seem a natural domain for reinforcement learning and, as the trend for selling games based on better graphics is peaking with the saturation of game shelves with excellent graphics, it seems that better artificial intelligence is the *next big thing*.

The main contribution of this thesis is a novel style of utility function, *group utility functions*, for reinforcement learning that could provide automated behaviour specification for large numbers of computer game characters. *Group utility functions* allow arbitrary functions of the characters' performance to represent relationships between characters and groups of characters. These qualitative relationships are learned alongside the main quantitative goal of the characters. *Group utility functions* can be considered a multi-agent extension of the existing *programming by reward* method and, an extension of the *team utility function* to be more generic by replacing the sum function with potentially any other function. *Hierarchical group utility functions*, which are *group utility functions* arranged in a tree structure, allow character group relationships to be learned. For illustration, the empirical work shown uses the *negative standard deviation* function to create balanced (or equal performance) behaviours. This balanced behaviour can be learned between characters, groups and also, between groups and single characters.

Empirical experiments show that a *balancing* group utility function can be used to engender an equal performance between characters, groups, and groups and single characters. It is shown that it is possible to trade some amount of quantitatively measured performance for some qualitative behaviour using group utility functions. Further experiments show how the results degrade as expected when the number of characters and groups is increased. Further experimentation shows that using function approximation to approximate the learners' value functions is one possible way to overcome the issues of scale. All the experiments are undertaken in a commercially available computer game engine.

In summary, this thesis contributes a novel type of utility function potentially suitable for training many computer game characters and, empirical work on reinforcement learning used in a modern computer game engine.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jay Bradley*)

Thank you Mum, Dad, Zak and Jess.

Table of Contents

Chapter 1

Introduction

This thesis makes practical and theoretical contributions to further the case for using reinforcement learning to automate behaviour specification for non-player characters in modern computer games. Reinforcement learning is not commonly used in modern computer games at the date of writing [Manslow, 2006]. This thesis introduces a novel extension of the team utility function to a more general class of functions called *group utility functions*. A large number of experiments were run in a real-world computer game engine (shown in Figure 1.1) that show how to use group utility functions to combine quantitative and qualitative measures into the reward function. Theoretical and practical obstacles encountered when applying reinforcement learning to computer games are discussed. First, the motivating factors for this work are given.

1.1 Motivation

> *Although graphics technology allows the creation of games set in environments that look incredibly realistic, the behaviour of computer controlled characters, referred to as Non-Player Characters (NPCs), often leads to a shallow and unfulfilling game experience. The application of sophisticated AI techniques to the control of NPCs could rectify this situation and create more immersive games.* [Fairclough et al., 2001]

In the majority of modern computer games the non-player characters' behaviours are predetermined by hand-written scripts [Woodcock, 2002, Houlette and Fu, 2003, Rabin, 2003b, Ferguson et al., 2006] (discussed in Section 2.1.1) usually arranged as finite-state machines. This is an easily understandable and powerful method for designing behaviour. However, this method can lead to homogeneous character behaviour. As behaviour specification increases, so does the amount of human programming time.

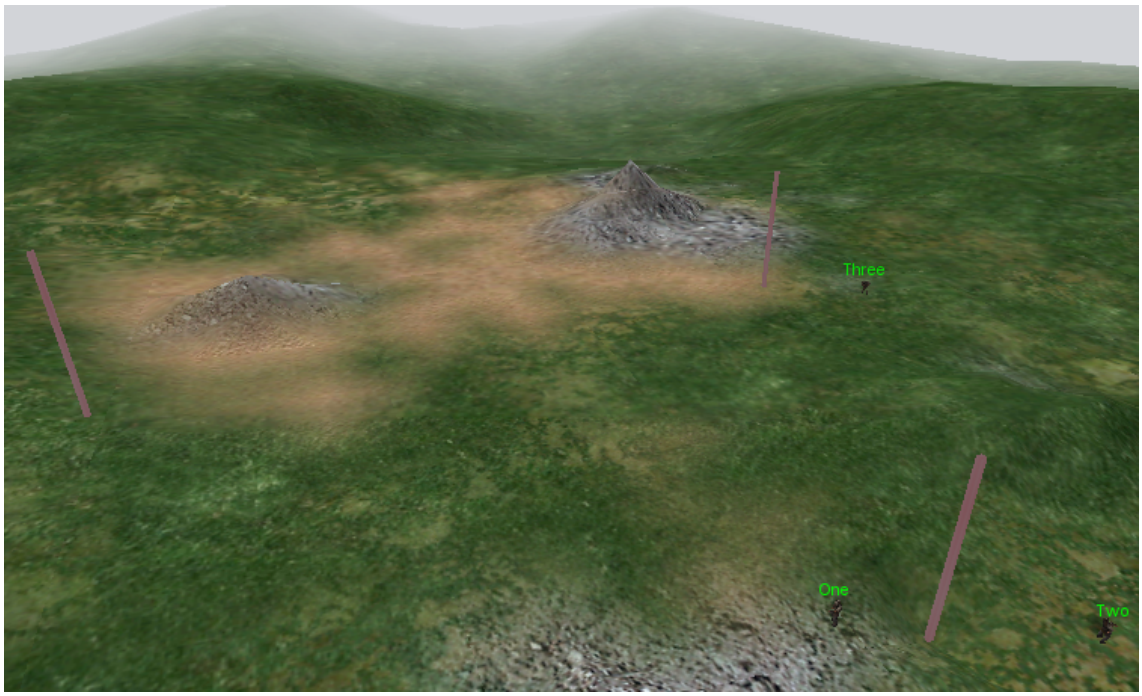


Figure 1.1: Torque [Garage Games, 2001] is used as the environment for each experiment presented herein.

This in turn leads to predictable or repetitive interactions and low expectations from the game player. This problem is more evident when many non-player characters are involved. If unique characters are required for the game then the game can suffer from a short game play time due to the effort required to specify those behaviours.

Modern computer games can be complex, soft real-time systems requiring efficient use of processing power for each non-player character. Heuristic decision systems, such as finite state machines or simple expert systems, meet this temporal criterion, are simple to put together and can be built with consideration and careful management to represent sophisticated behaviour. However, there is a linear or worse relationship between the time taken to design numerous character behaviours this way and the number of characters displaying individual behaviour. That is, the number of unique behaviours increases linearly or at a less than linear rate against the amount of human design time required to produce the behaviour specification. Hence, although in a typical computer game the characters may look different, many of them share the same or similar behaviour rules.

Much research sourced from commercial game institutions is based around how to get more complex or sophisticated behaviours from finite state machines. The best

one can hope for is that the behavioural rules are modularised and the modules are combined to create many acceptable distinct behaviours. For example, many characters in a game could share the same idle behaviour and only have unique behaviour when engaged in some way by a human character. Thus human design time is saved but the uniqueness of the characters' behaviours is compromised. That is not to say that some modern games do not have convincing character behaviour but the point is that human design time can be high for such games and such behaviour can be limited to a handful of characters. It should also be pointed out at this stage that few details are given for the AI methods used in commercial computer games (and the side of a retail box is not a reliable source of implementation details or performance metrics[1]). Some sources for details of computer game design and programming methods exist mainly on Internet sites such as Gamasutra [United Business Media, 2001] and AI Depot [Champandard, 2007b] and in the AI Game Programming series of books (Section 2.1).

Academic[2] research into traditional games such as chess and backgammon has a long and productive history with many well known successes, most famously for reinforcement learning in backgammon (through [Sutton and Barto, 1998]) and more generally famous in Deep Blue [Campbell et al., 2001]. These games bear little or no resemblance to modern commercial computer games and most of the research in these areas is of no interest to us here. For example, traditional games such as chess allow perfect knowledge in a reasonably small representation and have well understood predictable dynamics. This is unlike a massively-multiplayer persistent on-line game such as World of Warcraft [Blizzard Entertainment, 2009]. That is not to say that the problems faced by artificial intelligence for traditional games are simple (*go* in particular is a difficult problem) but they are different and as such not so relevant here. Research into commercial computer games is now becoming more common and Master's level dissertations and doctorate level theses on the subject of character behaviour and gameplay are beginning to appear [Hawes, 2000, SteveMcLean, 2003, Yannakakis, 2005].

[1] "Check out the revolutionary A.I. Drivatar^{TM} technology: Train your own A.I. 'Drivatars' to use the same racing techniques you do, so they can race for you in competitions or train new drivers on your team. Drivatar technology is the foundation of the human-like A.I. in Forza Motosport." [Laird and van Lent, 2005] "Your creature [in Black & White] learns from you the entire time. From the way you treat your people to the way you act toward your creature, it remembers everything you do and its future personality will be based on your actions." [Laird and van Lent, 2005]

[2] Throughout this thesis the term *academic* will refer to work coming out of research institutions usually appearing in journals or conference proceedings. The *commercial* research referred to is usually from web-sites presenting articles written by practising game designers or developers or from the Game AI Gems or AI Wisdom series of books featuring designers and developers from game studios (plus the occasional academic showing that crossover is happening).

Laird (from [Gold, 2005]) gives us some insight as to why doing academic research in computer games is finally gaining interest: home games consoles are becoming much better equipped for high processing requirements meaning that more involved AI techniques can be run in the soft-real time computer game environment; Master's students, doctorate and post-doctorate level researchers are young enough to have possibly grown up with a strong computer games influence; and now that high quality computer game graphics have saturated almost all new game releases, the clamour for the *next big thing* has moved to artificial intelligence, making this type of research commercially attractive. Woodcock agrees:

> *With the saturation of the computer game market with quality graphics, quality AI is now the next big thing for marketing and also on the production side, AI has finally "made it" in the minds of developers, producers and management. [Woodcock, 2000].*
>
> *AI programming teams are getting more processor time to run their code and the advancing power of home PCs and gaming consoles will inevitably allow us to use any sophisticated AI methods in future games* [Woodcock, 2003]

The number of non-player characters in computer games is increasing, for example, with the popularisation of on-line persistent worlds such as World of Warcraft [Blizzard Entertainment, 2009]. The proliferation of non-player characters showing remarkably similar behaviours is set to become more of a problem. The reasons for wanting characters to be able to learn have already been succinctly summarised:

- Improved Game Play
- Cheaper AI development - Avoid programming behaviors by hand
- Reduce Runtime Computation - Replace repeated planning with cached knowledge
- Marketing Hype

[Laird and van Lent, 2005] (which is a good introduction to all forms of machine learning for use in computer games.)

Potentially, learning means that characters could:

- tailor their level of proficiency to that of human players;
- continue to adapt over time to aid replayability and interest throughout the entirety of the game;
- show robust and complete behaviours even at edge cases (i.e. novel or uncommon situations);

- learn many unique behaviours based on their inherent abilities and experience;
- learn to fulfil complex roles given to them by a human designer that the human designer would have found difficult to have conceived of, let alone implement.

The problems associated with the use of learning in games include: loss of fine grained control over character behaviour, an investment in a new technology that affects design through to testing and a fear of using a technology that has not been proven to work in commercial computer games. These three drawbacks will, however, be removed with time. The loss of control is only really perceived - game designers should come to realise that they do not have to give over all control to learning methods as only certain controllable aspects of characters' behaviours are set through learning. Investment in new technology, combined with the uncertainty of unproven techniques, is an obvious difficulty to overcome for game software houses that most likely will have tight turnaround times. As more games use these methods to gain an advantage in the marketplace then both issues should be resolved as more developers emerge with the requisite skills and learning is proven to be of benefit to some game types.

There is a notable difference in opinion between *commercial* and *academic* practitioners of AI. The commercial opinion is that academic artificial intelligence methods are impractical in real games, that they are focused only on optimality and that they do not cater for game designers who need more control over behaviour than methods such as reinforcement learning can give. Of course, an academic researcher in the field of artificial intelligence might not agree that what constitutes artificial intelligence in computer games is truly artificial intelligence. For artificial intelligence in computer games the main requirements are that the algorithm be sufficiently efficient for home game consoles, the behaviour be somewhat convincing for human players and that the behaviour specification be done to a deadline. The behaviour of agents or characters in both fields is the focus but for computer games, how the behaviour is achieved is not such an issue, just so long as the characters perform their jobs to a reasonably acceptable level without affecting runtime performance. Academics generally have more time and resources to experiment with exotic learning algorithms. For example, reinforcement learning and genetic algorithms are biologically inspired. It may be reasonable to say that because of the disjunct between artificial intelligence in commercial computer games and in academia, artificial intelligence in the two fields should possibly be referred to using different terms and nomenclature. Gold noted this difference in the title of her paper, "Academic AI and Video Games" [Gold, 2005]. This is clearly

changing however and some academic methods such as genetic algorithms and forms of adaptation are creeping into computer games.

Reinforcement learning has been utilised in very few computer games (other than in traditional games such as backgammon which have been moved to the computer and similar games) and indeed most *academic* artificial intelligence methods have seldom been used either. Reinforcement learning (Section 2.2.4) is a search based optimisation technique, which puts it in the same family of algorithms as genetic algorithms, or more loosely evolutionary computing, which have been used in some computer games [Laramée, 2002, University of Texas at Austin, 2009]. Reinforcement learning differs from genetic algorithms as it attempts to adapt an agent's behaviour from the start to the end of its "life" whereas, evolutionary methods, as the name suggests, adapt between generations. At an abstract level, aside from the point at which behaviour is updated, the methods work in similar ways. They measure an agent's performance at regular intervals (after each time step for a learning agent and after each termination of an episode for genetic algorithms) and in some way adapt the agent to attempt to make it perform better over time with respect to some performance function. These methods require a way of evaluating the "worth" of an agent's performance and a representation of the agent's behaviour that can be altered. Reinforcement learning uses a reward function to measure the ability of the agent to perform well and, for representing an agent's behaviour, a map from state-action pairs to utility values is kept along with a policy for choosing actions based on that mapping. The map consists of a vector representing an agent's situation in the environment and the action it will choose to take from that situation mapped to a utility value - a real value indicating how rewarding it is to perform the chosen action from the situation[3]. Any reinforcement learning task is underpinned by a Markov decision process. A Markov decision process is any task where an agent chooses actions to move from state to state with a fixed probability distribution over the subsequent states given the chosen action. It is not imperative to be Markovian but the more Markovian a task is, the higher the chance of an agent learning to perform the task well and/ or in less time. Many modern computer games may have Markovian properties so reinforcement learning should perform well.

There is a potential issue when using search based optimisation methods. These methods produce agents with optimal or near optimal behaviour. However, optimal behaviour is most often not what is wanted from computer game characters. Imag-

[3]Some reinforcement algorithms map states to values and others (as used in this thesis) map combined state-action vectors to values.

ine a game where the human player loses every time without hope of outperforming the computer controlled characters, for example a racing game in which the player is humiliated by being continuously lapped. Such a game is rarely desirable. Though optimisation has proved useful in finding sweet spots in football games (Section 2.3.1) the ability to *craft* the non-player characters' behaviours is more important in game character behaviour specification. As discussed, however, hand-crafting is too time consuming to be possible with a large cast of characters in rich environments without serious compromises.

Reinforcement learning semi-automates the process of character specification. The designer must supply a reward function which, along with the agent's physical abilities in the world and their interactions whilst learning, determines the agent's final behaviour. This reward function is the primary means for a designer to determine an agent's eventual learned behaviour. Can we can put more control into the hands of the designer when setting the reward function, thus making reinforcement learning more suitable for use in computer games (Section 2.2.1)? How can we give the ability to specify not just quantitative maximisation goals but also qualitative behaviour in the reward function?[4]

1.2 Extending a Simple Method of Reinforcement Learning to Suit Computer Games

Chapter 3 describes the most significant novel contribution of this thesis - extending the team utility function to a more general form called group utility functions. These group utility functions are inspired by the view that the groups that a game character belongs to will determine its behaviour (just as agents under a team utility function behave as a team). For example, are they a "goody" or "baddy?" What type of "baddy" are they? What species are they? There are also usually groups of characters within other groups. The relationship between these groups provides the high-level structure of many games, obvious examples being sports teams and real-time strategy games. Which group or groups an agent belongs to usually determines its behaviour to a large extent [Reynolds, 2002]. It makes sense to work towards incorporating a group relationship structure into multi-agent reinforcement learning methods for use in computer games. As such the *team utility function* and *programming by reward* have been

[4] An example of a quantitative goal might be how fast a race was completed and an example of a qualitative goal in the same game would be how the computer chose to drive.

taken as inspiration for the approach of this thesis. The only existing simple group learning mechanism for multi-agent reinforcement learning is the well known team utility function (see [Wolpert and Lawson, 2002] and [Wolpert et al., 2001] for discussion and criticism). In *programming by reward* (Section 2.3.4) an agent's behaviour is determined by what traits they have. For example, the designer of a fighting game such as Mortal Kombat [Midway Games, 1992] may like a character to punch a lot, kick very little and have a strong desire to behead opponents. Through programming by reward the characters learn to behave appropriately for such a specification. This is very similar to the agent's behaviour being determined by which groups they have been assigned to. One could say for example that belonging to the "baddy" group is the same as having a sinister trait. Single agent experiments with programming by reward have shown the method to be effective at producing several different behaviours from otherwise homogeneous agents. It has been suggested that programming by reward could be used for computer game characters [Shapiro and Langley, 1999]. *Group utility functions* [Bradley and Hayes, 2005a, Bradley and Hayes, 2005b], detailed in this thesis, expand the team utility function to be more general - taking the concept of programming by reward and applying it to produce relationships instead of individual traits, thus helping to provide structure in computer game character societies.

The team utility function rewards each agent with the sum of all the agents' rewards. In this way a notion of teamwork is achieved with agents hopefully learning that what is good for the team is good for them. There exist a number of unexplored group utility functions other than the team utility function that provide a modified reward[5] to individual agents that is derived from several agents' performances. Figure 1.2 shows how the team utility and group utility functions relate. They differ in the way in which the individual agents' rewards are combined which determines the relationship they have with each other.

The team utility function sums all the agents' rewards which implies that each agent is on the same team (hence the name) with the same goals. The combining function that gives the negative standard deviation of the agents' rewards is introduced later. Using this combining function we can make agents in a group learn to perform approximately equally as well as each other. This could be useful in a computer game if we need several agents to compete and draw at some task. For example, we could use this combining function to provide non-player driving characters in a race game if

[5] In this thesis, the term *modified reward* is used to refer to the reward that an agent receives after any processing by team or group utility functions.

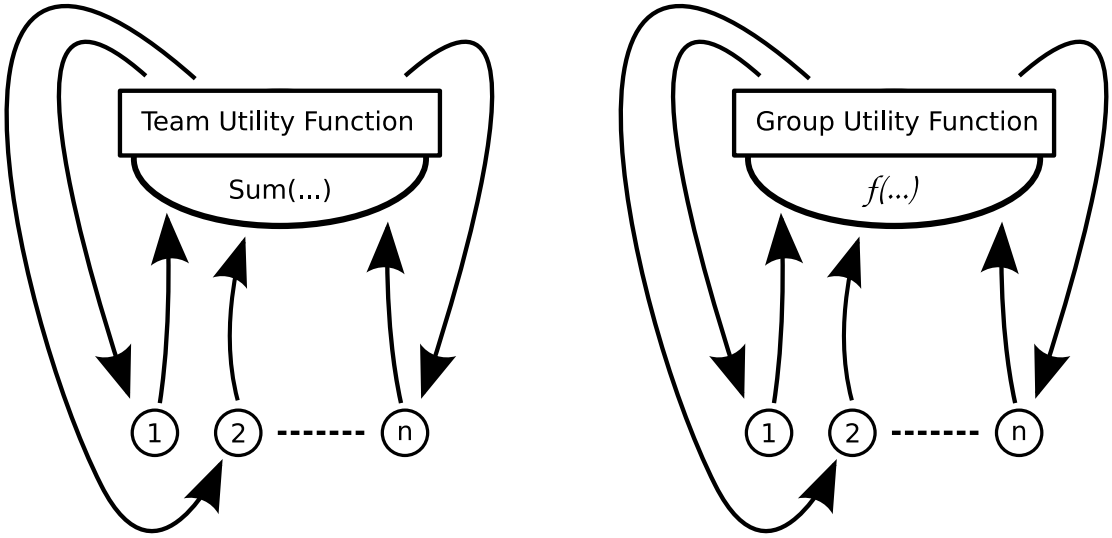


Figure 1.2: The difference between a team utility function and a group utility function i.e. the combination function of the agents' privately collected reward which is always the summation in the team utility function but can be an arbitrary function for group utility functions. The reward flowing *upwards* is the agents' unmodified, privately collected reward and the reward travelling *downwards* is the modified reward that an agent uses as its reward signal.

we wanted the result to be close. This balancing utility function is used as an example of a group utility function, a function other than summation that provides a useful relationship for computer game scenarios. One can imagine other simple but useful functions such as: less than; greater than; or a combination of both (some interesting functions are presented in Section 3.4). Further, more sophisticated functions that map more complicated relationships could be used. For example, if the agents' state representations could distinguish time then the functions could use calculus for time dependent relationships.

The team utility function is used with a flat structure. That is, all the agents' rewards are summed by one team utility function and then fed to the agents as modified reward which they use to learn with. When using the team utility function, groups of agents can belong to different teams if they feed their individual rewards into separate summation functions. There is, however, no sensible use for any hierarchical organisation between teams when using the team utility function. Using group utility functions however, one can create a hierarchy of groups, each with their own utility function. The fact that there could be more than one type of relationship between the agents means

that when the group utility functions are combined in a hierarchy, more complex relationships are created[6]. As an example, using a balance inducing combining function (i.e. the negative standard deviation) with two team utility functions "beneath" it, it is possible to train two teams of agents to perform equally as well as each other.

This thesis also uses *group affiliation* values. These are scalar values that can be used to indicate the strength of an agent or group's attachment to a group utility function. To take the example above of being a "baddy," this means an agent with a group affiliation value of *one* between it and the baddy group utility function is, "a standard baddy." Whereas if the agent was attached to the baddy group with an affiliation value of *two* the agent would learn to be twice as bad.

The idea that agents' behaviours can be set by which group they belong to is analogous to programming by reward. Group utility functions are a novel take on programming by reward and when used with reinforcement learning they bring us closer to a multi-agent extension of programming by reward.

The computer game market is ever growing - $46.5 billion worldwide worth by 2010 [Rushe, 2007], $44 or maybe $54 billion by 2011 [McIlroy, 2008]. Many academic researchers are starting to use computer games as simulations for experiments and a few researchers and institutions are dedicating their research to them. At the same time, few computer game software houses are using the most recent methods of artificial intelligence. The size of the cast of characters in some popular genres of computer games is growing and the problem of hand coding non-player characters' behaviours will be more evident the larger they become.

There will be a greater need in the future for semi-automated behaviour specification in computer games, especially where: the world is persistent; where human players may get bored of static character behaviours; where the non-player character population is large and individual unique scripted behaviours would take too much human design time; or where the environment is rich in terms of many possibilities for interaction. Many games with very large populations of human players need many non-player characters to help keep the society sane - to provide ballast for the hedonistic or novice human players. Nobody wants to be the shopkeeper or the cannon fodder in the near-future massively multiplayer games with persistent universes but these roles may be needed [Kosak, 2002]. Hand-scripted behaviours, although always serving a dominant role, need to be supplemented with methods capable of produc-

[6]Note that this is entirely unrelated to what is normally called hierarchical reinforcement learning methods.

ing a large amount of unique, reasonably sophisticated behaviours. Tailoring existing reinforcement learning methods may be the start of the answer.

1.3 Research Hypothesis

I claim that by using group utility functions it should be possible to learn a primary quantitative task (foraging for as many rewards as possible given a time limit) whilst at the same time learning a secondary qualitative task (maintaining an equal number of rewards between agents, groups and agents and groups). The secondary task will come at some cost to the primary task. This effect should be observable by looking at graphs of the agents' and/or groups' performance in terms of number of rewards picked up per some time period and the absolute difference in the rewards picked up by the agents and/or groups during the same time period. This will be tested using six experimental configurations:

- Two agents;
- Two groups of two agents each;
- Two groups of three agents each;
- One group of two agents and a single agent;
- One group of three agents and a single agent.

The effect of the secondary, balancing task should be weak in the first three naturally balanced configurations and should be stronger in the last two configurations where the agents' resources are inherently unbalanced. It is predicted that both the primary and secondary parts of the task will be more difficult to learn as the number of agents increases because of the high signal-to-noise ratio, the curse of dimensionality and hidden state. It is predicted that using approximated value functions rather than tabular value functions should overcome some of this difficulty at the expense of asymptotic performance and possibly, stability.

1.4 Summary of Introduction

> *Game AI is not just neural networks and learning systems and complex mathematical structures, although it can be, but primarily game AI is*

> *about creating an environment and the appearance of thought from units. Game AI is behavioral, not scientific.* [Howland, 1999]

In order to adapt reinforcement learning methods to be more suitable for use in computer games, this thesis presents a simple value function that combines quantitatively optimal behaviour with qualitative behaviour in groups of agents (Chapter 3). Programming by reward has shown that this is possible with single agents but no existing work uses programming by reward with multiple agents. Group utility functions can create relationships between agents and groups of agents much like programming by reward creates single agent character traits. Whereas the team utility function sums all of the agents' individually collected reward, group utility functions can use any arbitrary function of a group of agents' individual rewards with the understanding that the function defines the relationship between the agents.

The main points made by this thesis are:

- By adapting a simple existing method of reinforcement learning game characters can learn relationships between themselves much as programming by reward has been used for individual character behaviours - Chapter 3.
- A *balancing* group utility function can induce a qualitative behaviour which, in this case, is equilibrium - Chapters 3 and 5.
- The method scales poorly using tabular value functions with more difficult tasks, more agents and larger state-action spaces but using function approximation can alleviate this - Chapters 5 and 6.

Besides these central points, several interesting theoretical issues arise when applying reinforcement learning to commercial computer games. These are:

- Should games be modelled as episodic or continuous tasks? - Section 2.2.4.2;
- How should the state and actions be represented for computer game environments? - Section 2.2.4.2;
- How can group based utility measures be used when agents are acting asynchronously? - Section 4.2.2.

The contributions of this thesis to further research in this area are:

- Group utility functions: an extension of the team utility function, an existing reward function used for reinforcement learning;

- Hierarchical group utility functions: the application of more than one group utility function in a hierarchy used for learning more than one relationship simultaneously;
- A large body of empirical work showing the performance of reinforcement learning in a computer game engine;
- Discourse on exploration/ exploitation balance in action selection policies, the fine line between episodic and continuous tasks and using a team utility like value function with asynchronous action steps.

Chapter 2 gives an overview of the background required for understanding the present use of artificial intelligence in commercial computer games and existing academic work that uses reinforcement learning in computer games. There are few resources detailing the use of reinforcement learning in computer games and so contributions from other areas such as reinforcement learning used for robotic football and the related method of genetic algorithms, are included. The reader will see from the background that:

- There is a separation between academic and commercial styles of artificial intelligence;
- Reinforcement learning seems well suited but has been used little in commercial computer games;
- The related method of genetic algorithms has had some success in bridging the divide between academic research and commercial computer games;
- The team utility function can be extended to more general group utility functions that are similar to the concept of *programming by reward* except that programming by reward has not yet been used with multiple agents;
- The idea of group utility functions fits well with the structure of some modern computer games.

Chapter 3 explains how the novel idea of hierarchical group utility functions was derived from the existing team utility function and the principle of programming by reward. The chapter also shows the equations for applying group utility functions to an existing reinforcement learning algorithm. Chapter 4 shows the results of initial

experiments in the chosen problem domain. Details of the environment, learning algorithm, state representation and possible agent actions are given in Chapter 5 along with results showing that two agents belonging to a group utility function with the negative standard deviation combining function learn to perform a task in equilibrium. This chapter proves the hypothesis that in reinforcement learning one can trade a slight drop in overall quantitative performance to gain some qualitative behaviour when using group and hierarchical group utility functions. Section 5.1 of this chapter is based on work previously published [Bradley and Hayes, 2005a]. The chapter goes on to show how the experiments scale with more agents and more difficult tasks. Section 5.2 is also based on previously published work [Bradley and Hayes, 2005b]. Chapter 5 also contains theoretical discussion addressing practical issues found when implementing reinforcement learning in a modern computer game and, how they can be overcome. Chapter 6 presents repeated explanation and experiments for using non-linear neural networks to approximate the agents' value functions instead of using a table based method, as in Chapter 5. Finally, Chapter 7 places this thesis in the context of existing and future research, and critiques and highlights contributions made by this thesis and potential further research.

Chapter 2

Background

This chapter starts by considering AI methods in existing computer games including Black & White, arguably the most influential game in terms of AI [Champandard, 2007e]. The academic artificial intelligence method of reinforcement learning is then discussed before four strands of research that have directly influenced this thesis.

Section 2.1 describes current AI practices in modern computer games, drawn from books such as the *AI Game Programming Wisdom* series, the *Game Programming Gems* series and internet sites such as *www.gamasutra.com*, *aigamedev.com*, *www.aiwisdom.com*, *www.gameai.com* and *www.ai-blog.net* and also from some academic conferences where commercial practitioners have attended. Technical details can be scarce in commercially sourced texts, likely to protect in-house work.

2.1 AI for Computer Games

> "It is anticipated that the widespread adoption of learning in games will be one of the most important advances ever to be made in game AI." [Manslow, 2002]

Human Level AI's Killer Application: Interactive Computer Games [Laird and van Lent, 2000], is a call to the academic community to use AI methods in computer games. It positions computer games as the ultimate research target for artificial intelligence techniques, likening them to robotic football while noting that the former have far more variety in behaviour. The paper points to the current trend in computer games for massively populated, persistent on-line worlds as evidence that game players have become dissatisfied with current non-player characters and instead

seek to play against many other humans. The popularity of these massively multi-player games is inspiring research into AI controlled characters. A large population of human players rarely organises itself well in a game environment as too many humans want to be the hero of the game rather than the cannon fodder. Cheap (in terms of processor cycles) artificially intelligent controllers are one way of providing a very large population of supporting characters for the human players such as opposition soldiers, racing car drivers who "crash and burn" and non-player character police [Kosak, 2002]. To quote from Laird's call to arms: "Adding other AI controlled support characters could help populate the games with interesting opportunities for interaction that guide the player along various plot lines. Since these characters need to exist in a virtual world and generally play a human role in this world, they provide a useful first step towards human-level AI." [Laird and van Lent, 2000]

In academic institutions, artificial intelligence research using computer games is now becoming more common (see [Fairclough et al., 2001, Niederberger and Gross, 2002] for an overview and [Bragge and Storgrds, 2007] for some statistical evidence). Use of artificial intelligence techniques in commercial computer games is growing also [Pottinger and Laird, 2000, Woodcock, 2000, Dybsand, 2001, Woodcock, 2003, Kirby, 2004, Dybsand, 2004], however the techniques are usually limited to finite state machines (see [Champandard, 2007a] for a critique of finite state machines in computer games). In *Promising Game AI Techniques* [Rabin, 2003c] Rabin says about current academic research, "What these techniques lack is not usefulness, but rather the impetus for game developers to take them seriously." He asserts that "things are changing" mainly due to the employment of recent artificial intelligence post-doctorates and that reinforcement learning is "extremely powerful." The academic research community needs to show working examples of techniques like reinforcement learning to convince commercial game developers of their power and practicality.

2.1.1 Finite-state Machines

In *AI programming Wisdom 2* there are two articles [Rabin, 2003b, Rabin, 2003c] by Steve Rabin detailing the current and future state of the art for AI in computer games. *Common Game AI Techniques* [Rabin, 2003b] describes finite-state machines as "a widely used software design pattern that has become a cornerstone of game AI." Fu and Houlette also state that, "Finite-state machines are without doubt the most commonly

used technology in game AI programming today."[Houlette and Fu, 2003]

2.1.2 Optimality

Academics could be accused of treating game AI as just another engineering problem, seeking only some optimal performance based on raw returns such as games won or opposition killed. "Needless to say, gameplay isn't something that today's AI researchers feel comfortable evaluating." [Pottinger and Laird, 2000] For a view on the prevailing opinions which supports the above statement see [ai-blog.net, 2007] in which Adam Russell from Lionhead Studios [Studios, 2002] writes about the divide. Upon hearing the proposals of those in traditional academic AI research for possible games industry collaborations at the inaugural meeting of the UK's EPSRC-funded AI/ Games Research Network he thought, "These guys have no idea about games! they're so naive!" The discussion of this blog-post is illuminating and there are contributions from several people of recognisable stature from both the academic and commercial worlds. One point repeats itself - that the two groups *are* now far apart but that in the future they could become closer which would benefit both parties. Others postulated that the distance is natural and will always exist because the two groups are dealing with different problems. The inaugural meeting obviously caught the attention of academics and commercial practitioners, gaining several commentators [Wardrip-Fruin, 2007, Champandard, 2007d, Togelius, 2007].

2.1.3 AI Game Programming Wisdom

The *AI Game Programming Wisdom* series is indispensable to anybody looking to do research into AI for computer games. The *AI Game Programming Wisdom* series consists of four books containing short articles aimed at the practising programmer wanting to use AI in their computer game. Most of the authors are based in commercial computer game companies but more academic based researchers are featuring as the series progresses - evidence perhaps that academics and game programmers are coming closer together. The first *AI Programming Wisdom* book, published in 2002, contains seventy-one articles on computer game AI, ten of which address some form of learning or adaptation [Rabin, 2002]. *AI Game Programming Wisdom 2* contains sixty-seven articles of which nine have some concern with learning or adaptation [Rabin, 2003a]. *AI Game Programming Wisdom 3* contains fifty-three articles of which eight concern adaptation in some way [Rabin, 2006]. The recently published *AI Game*

Programming Wisdom 4 contains fifty-five articles, six concerning learning or adaptation of some sort with two of those about reinforcement learning [Rabin, 2008]. Most of the material in the following section is largely based on the *AI Game Programming Wisdom* series.

2.1.4 Learning

> "There seems to be so much scope to apply learning methods in games, and yet successful examples of this in video games are few and far between." [Lucas, 2009]

Manslow makes a case for the importance of reinforcement learning in future games [Manslow, 2002]. He classifies two types of learning: *indirect* and *direct*. *Indirect* learning refers to learning information about human players or the game which can be used by AI controlled characters to modify their behaviour. Manslow uses *direct* learning to refer to methods which work on the behaviour of a computer controlled character explicitly. Under Manslow's definitions, reinforcement learning is classed as direct learning. Indirect learning is probably more commonly called *adaptation*. Almost all learning in existing games is indirect learning. The main creature in Black & White is a good example of indirect learning as it adapts its behaviour based on user feedback but does not learn any new behaviours *per se*. Manslow claims, with good reasoning, that direct learning (i.e. reinforcement learning) is impractical without constraints. On those constraints, Manslow recommends that direct learning should only be allowed to alter a limited subset of character behaviour that would not result in catastrophe should the character develop odd behaviour. This is only an issue if the learning is left on during game playing rather than used prior to game delivery where the behaviours can be checked. He also goes on to say that direct learning is "perhaps the ultimate AI." But more usefully he recommends that *a priori* knowledge should be used as much as possible and that, because reinforcement learning usually takes a long time, we need to train agents against agents[1] and that this is only possible when "there is an exact symmetry between the roles of the AI and the player."

Manslow [Manslow, 2003] goes on to reiterate the common use of rule based controllers and gives four reasons why reinforcement learning should be used in computer games:

- "The same RL engine can be used to solve a wide variety of unrelated prob-

[1] Often referred to as self-play.

lems, ranging from producing a competent Chess player, controlling an aircraft to fly as low as possible to avoid radar without crashing, controlling AI vehicles to follow paths specified as a series of way-points, controlling the movements of dogfighting aircraft, to producing a competent player of a real-time strategy game."

- "Provided that the problem is set up correctly, RL is likely to find a close to optimal solution with minimal development effort."
- "It can find optimal behaviours even in situations where the effect of an action might not be immediately apparent - either because its effect has a strong stochastic component, or because its effect is delayed. Either of these can make the manual creation of even moderately competent AI extremely difficult."
- "It learns as an AI interacts with the game world, making it suitable for use both during development and to facilitate in-game learning once a game is complete."

Like Manslow, Baekkelund believes that reinforcement learning is powerful and well suited to use before game release [Baekkelund, 2006]. Furthermore, that a "very small but information-rich representation of the state space" needs to be found for learning to be possible. The state representation used in this thesis attempts to satisfy these ideals. Importantly, Baekkelund says that neural networks cannot be used to approximate the value function for reinforcement learning due to the fact that they can "often produce erratic behaviour." This function approximation is precisely what is experimented with in Chapter 6.

RETALIATE [Lee-Urban et al., 2008] is a system that uses reinforcement learning to obtain better-than-hand-coded high-level strategies for dominance games in Unreal Tournament. In Unreal Tournament, dominance games are won by holding several geographical positions for longer than the other teams in the game. RETALIATE uses reinforcement learning only to choose which of the dominance areas each agent should move towards whilst the characters themselves are still controlled at a low-level by their finite-state machine. RETALIATE is shown to quickly learn to outperform some example Unreal Tournament bots at the dominance game. Also, policies learned against one example set of bots can then be quickly adapted to combat other example bots (with different behaviours) through reinforcement learning during a game.

Kirby [Kirby, 2004] puts forward an interesting idea when he highlights the potential of learning in massively-multi-player network games because of the wide exposure

of characters to human players with different behaviours. Hence, the state-space is adequately covered by the learning agents. Therefore some form of distributed reinforcement learning could be implemented.

For an introduction to reinforcement learning in computer games see *A Practical Guide to Reinforcement Learning in First-Person Shooters* [McPartland, 2008] which covers the basics of reinforcement learning, sarsa (covered in Section 2.2.4.1) and state representation together with some good results for a simple navigation task and some poor results for a combative task. The combative task failed to perform well because the state representation did not include enough information to perform the task adequately. The state representation was small which should mean that the task would be learned well but the value function was still changing by significant amounts towards the end of the experiments. This is indicative of a state space that has been kept too small and so does not contain enough information to distinguish between states sufficiently enough to learn the correct value function. That is stochasticity was too high which means that certain reward values for the same state-action pairs will vary significantly because of the hidden state. This is a partially observable Markovian decision process. The partial observability refers to the fact that one state may in actuality cover more than one significant state for the agent. Thus more than one different real state is reported by the state representation as the same state.

2.1.5 Rule Based Control

In *AI Game Programming Wisdom* there are two articles concerning group control. Van der Sterren [van der Sterren, 2002b, van der Sterren, 2002a] gives us two systems: *decentralised emergent group behaviour* and *centralised planned control*. The "emergent" group behaviour requires that agents share their observations and intended actions with other members of their group. Individual agents use this expanded state to match against scripted actions. The expanded state contains more information so that more rules can be written to match against more situations that concern the whole group so group behaviour can be achieved. The centralised approach requires that a group state be kept which contains an abstract representation of "observations, historic data and experience..." Fuzzy rules determine which scripted group action should be executed according to this group state and to stop state thrashing. The transitions between group manoeuvres are controlled by a finite state machine. Both these systems are rule based and will suffer in terms of interesting unique behaviours as complexity

and/ or the number of characters grows.

2.1.6 Testing

Testing is as important in games as in any other software release and adding non-determinism may have implications for detecting issues. Barnes and Hutchens [Barnes and Hutchens, 2002] offer some practical advice saying that the testing phase of development must be planned from the start with learning in mind and the state of any learning entities should be saved to journals for later debugging. They also say that testing the learning system must go hand in hand with testing the look and feel of the game. That is, the learned behaviour is only useful if it adds to the game in an appropriate manner. They state that the creatures in Black & White (Section 2.1.8) have innate behaviour constraints that will never allow the creatures to do certain, *undesirable things*. The *undesirable things* are left unstated, but they would be highly dependent on the role of the character. For instance a friendly character should probably be forced not to harm the human player. With reinforcement learning the characters' final behaviours will still be massively constrained by the designer so the possibility that the learning may do something completely unpredicted in the game world is not of serious concern. It is a concern that will persist until the use of reinforcement learning is commonplace in games.

2.1.7 Planning

The article *Team Member AI in an FPS* [Reynolds, 2003] is typical of current rule based approaches. It uses several hand-coded rules that restrict the actions of agents so that they do not get in each other's way when advancing or shooting at the enemy. This is not really planning, but it does give the appearance of coordination between agents which is the most important thing. Wallace [Wallace, 2003] gives a more in-depth discussion on agent planning. He suggests using hierarchical task networks for planning at different levels of abstraction and briefly mentions coordinating the networks between agents. Orkin [Orkin, 2003] gives a rallying call for the use of multi-agent planning in games: "By letting go of the reins and allowing games to formulate plans at runtime, we are handing over key decisions to those who are in the best position to make them - the characters themselves." He goes on to describe a simple blackboard system used in No-one Lives Forever 2 [Monolith Productions, 2002]. The characters can post requests or intentions; however, there is no specific group coordination.

F.E.A.R. is a recent success in the application of "academic" artificial intelligence to computer games. The artificial intelligence in F.E.A.R. uses a STRIPS style planner with modifications to make it more suitable for use in a modern computer game [Champandard, 2007c]. The use of a planner "relieves the programmer or designer of a burden that gets bigger with each generation of games," [Orkin, 2006] and is a more efficient use of manpower. In Orkin's own words:

> The motivation to explore planning came from the fact that we had only one A.I. programmer, but there are lots of A.I. characters. The thought was that if we can delegate some of the workload to these A.I. guys, we'd be in good shape. If we want squad behavior in addition to individual unit behavior, it's going to take more man-hours to develop. If the A.I. are really so smart, and they can figure out some things on their own, then we'll be all set! [Orkin, 2006]

Even taking F.E.A.R. into account, there is only one game that has truly ridden to success on the back of advanced artificial intelligence and that is Black & White.

2.1.8 Black & White

Black & White [Lionhead Studios, 2001] is likely the only known commercial game of note that has used learning as it is understood by the academic community. Black & White is a "god game" which allows a human player to teach one character in the game through a series of discouraging or encouraging actions realised by mouse gestures. A god game puts the player in the position of an all seeing god able to alter the lives and behaviours of their world's population whilst not actually directly controlling any characters. Aside from giving direction to mostly reactive inhabitants, the human player can stroke or slap one main creature giving positive or negative reward respectively for the creature's current action. The creature will also assume that tasks often asked of it by the player are worthy and will repeat them when similar situations reoccur. Lastly, the creature will adopt actions displayed by other characters in the game with the assumption being that if god (the player) sanctions the behaviour then it must be desirable. If other characters are asked to do jobs by the player then this is seen by the creature as positive reinforcement for that character's actions. The main creature develops throughout gameplay into one that assists the human player's tactics. Based on the BDI (Belief, Desire, Intention) architecture, the creature learns opinions which relate desires to states or objects [Evans and Lamb, 2002]. The creature holds beliefs about objects and types of objects represented as a list of attribute-value pairs

and decision-trees respectively. Desires are represented as perceptrons. It is not stated how the intents of the creature are represented. The beliefs about types of objects are kept as a decision tree built dynamically from the experience gained after attempting an action on a type of object.

There are many other characters in the game other than the main learning creature. These characters are organised into groups bound by social activities. Their actions are influenced by each social activity they are currently engaged in (worshipping for example). The characters are not entirely autonomous, with some of their control coming from a central decision process. The reasoning behind this is that the number of characters is potentially unlimited and to have them decide on cooperative actions (the social activities) could be costly in processor usage.

The small amount of adaptation and potential for non-deterministic behaviour of the main creature was described as a "gamble" by the game's designer, Peter Molyneux [Molyneux, 2001]. This highlights the perceived and probably very real difficulty in having characters that can change their behaviours post release. Denzinger points out that it is an odd problem; fearing that methods to make character behaviour less predictable for the game-player will make character behaviour less predictable for game designers [Denzinger and Winder, 2005]. The most obvious difficulty with such characters is that you must change the way that play testing is done as the characters will be less deterministic.

The main creature in Black & White shows signs of intelligence and, more impressively, adaptation. However, it is the other "incredibly simple-minded" [Wexler, 2002] characters in Black & White that could benefit most from reinforcement learning. With the application of reinforcement learning, these characters could each show individual behaviours across a large range of situations in a complex dynamic world with possibly similar or less human design time.

2.2 Artificial Intelligence

How can one have control over anything if it is automated?

2.2.1 Control

When designers of character behaviours want more input, or control, some automation has to be relinquished. For example, by using "options" a designer can express some

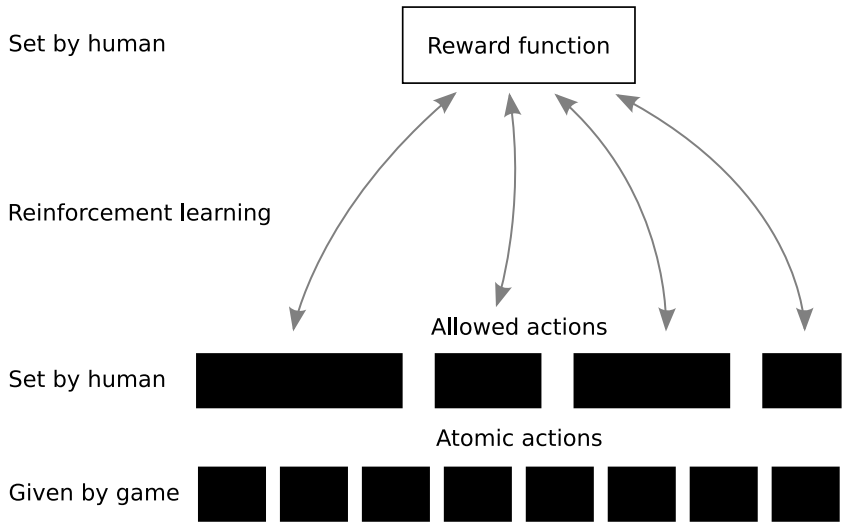


Figure 2.1: Reinforcement learning joins high and low level human specification.

parts of an agent's behaviour with meta-actions and use reinforcement learning to join these segments up. That is, the designer can specify both the high-level goals in a reward function (see Section 2.2.4.5) and the low-level actions. Reinforcement learning will automatically join them together, filling in the middle ground between the high and low levels of specification. Figure 2.1 shows the middle ground filled in by reinforcement learning. For example if a designer wishes that a character can only play a game using two methods then these ways are set down as meta-actions (containing more than one atomic action in a sequence). When learning the agent can only choose one of these meta-actions rather than their elemental actions individually. Though this may lead to sub-optimal asymptotic behaviour if the meta-actions are themselves sub-optimal, the point is that the characters behave in the way we wish at both high and low levels. In general, more human control takes us away from the benefits of automated learning (or any search based optimisation method) and can lead to problems such as specifying implicit games (as discussed in Section 2.2.4.5).

2.2.2 Situatedness

Although computer game characters can be thought of as embodied agents, acting in real-time in a well defined environment, there are several restrictions one does not have to deal with that affect truly situated agents (see [Konidaris and Hayes, 2005] for a good explanation of what it means for an agent to be truly situated). A truly situated agent is one embodied in some form for which it must overcome *real world* forces to

achieve its actions. Robots are truly situated; they require power to broadcast messages if within range and must use force to move around their environment. One could think of a Mars rover robot as the canonical situated agent. For computer game characters (as opposed to truly situated agents), there is negligible cost for communication between agents and centralised communication is easily possible. In addition, whereas truly situated agents can have a strict time limit for learning (i.e. real time), game agents can be run in simulated games for as long as is necessary for learning before game release. Although there are limits to the amount of time a designer is willing to wait to see the resultant behaviour. If one wanted, the agents could have perfect knowledge. However, we do not utilise or propose the use of perfect knowledge in this field of work because of the high processing requirements and in order to maintain perceptual honesty [Isla et al., 2001] (what is referred to as *epistemic verisimilitude* in Black & White [Evans, 2001]). Games are supposed to be implemented on home machines, not super computers. The processing power of home games machines, whilst increasingly fast, is still limited but of course Moore's law will eventually overcome this issue. Secondly, adversarial characters in computer games are often accused of "cheating" by human players because they clearly have use of more information than the human player. By not gathering perfect knowledge we deny our agents one way of becoming cheats. Ideally human players would recognise the decisions made by the characters based on their own thinking.

2.2.3 Markov Decision Processes - The Keystone of Reinforcement Learning

Reinforcement learning theory is underpinned by that of Markov decision processes [Zubek, 2006][2]. A Markov decision process is a temporal chain of sense-think-act steps and for an environment to satisfy the Markov property the state must encapsulate enough information to summarise all past states together with the current state.

Formally, equation 2.1 must hold for each possible state.

$$P\{s_{t+1} = s' | s_t = s, a_t = a, s_{t-1} = s_i, a_{t-1} = a_i, s_{t-2} = s_{ii}, a_{t-2} = a_{ii}, \ldots\} = P\{s_{t+1} = s' | s_t = s, a_t = a\} \quad (2.1)$$

[2]There are many citations for Markov decision processes, but this citation is interesting as it appears in the AI Game programming Wisdom Series of books which is aimed squarely at commercial AI game development.

Equation 2.1 states first, that the probability distribution over the next state, s_{t+1}, depends upon the entire history of the task from s_0, a_0 to s_t, a_t. And secondly, that the probability distribution over the next state, s_{t+1}, should be identical whether given the entire history or just s_t and a_t.

Whether a task can be described as *being Markovian* depends on the state representation and the environment. A task can be said to be *more or less* Markovian and it is desirable for a task to be as Markovian as possible. That is, being Markovian is not a boolean property but a sliding scale and it may be sufficient to be "Markovian enough" to learn a task well. If equation 2.1 holds true for all states and actions then the task is fully Markovian but if the equation does not hold for just one or a few cases then the Markov property is lost but the task may still be learnable. Sutton and Barto say to think of every state as an approximation to a Markov state [Sutton and Barto, 1998]. One can see just how easy it is to break the Markovian property by imagining multiple agents learning in the same environment. When there are multiple agents adapting their policies simultaneously, then one agent's state transition via a given action has high potential to be different on each repetition as the other agents may act differently each time. Learning in a non-stationary environment like this is more difficult but not impossible. The Markovian property is a useful measure of difficulty and a useful setting for reinforcement learning tasks.

The fact that the next state is not deterministic but occurs with a probability is called stochasticity. A stochastic environment can slow learning compared to a deterministic transition function. Reinforcement learning deals with this by averaging over the expected return from each possible next state given an action choice and a current state.

If the state and action spaces for a task are finite then the task can be called a finite Markov decision process. Any task that can be formulated as a finite Markov decision process should be a good candidate for reinforcement learning.

2.2.4 Reinforcement Learning

> *Learning* - "The act, process, or experience of gaining knowledge or skill." [Laird and van Lent, 2005]
>
> *Learning in computer games* - Allowing computer game characters to explore and interact with the game world and its inhabitants in order to ascertain the best way to fulfil their given roles in the game.

Reinforcement learning is clearly biologically inspired but it is an engineered

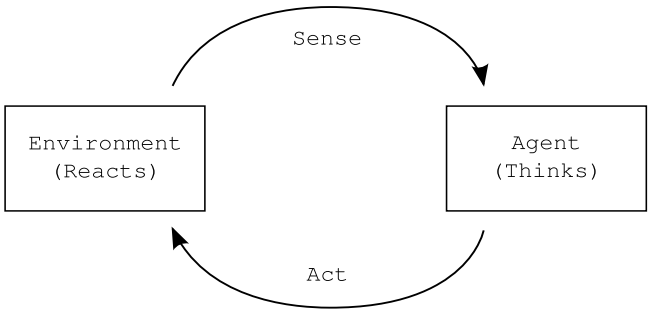


Figure 2.2: The simple iterative flow of an agent's life.

method that bears little resemblance to how humans learn. It can, however, be a useful analogy for understanding and explaining. Reinforcement learning is a search based optimisation method for the control of agents who are situated in possibly rich and dynamic environments. This thesis uses a well known, often used method of reinforcement learning called sarsa which is part of a group of algorithms called TD (temporal difference) methods. Sarsa is described more fully below in Section 2.2.4.1 but some of this more general introduction applies to sarsa (and sometimes only sarsa or TD methods). Reinforcement learning agents follow a simple sense-act-reward cycle at each discrete time-step as shown in Figure 2.2. That is, for each time-step an agent builds an approximation of the environment which is stored as its *state*[3], then based on this state the agent chooses an action according to its policy (its method of choosing an action given its value function). Once the action is finished the agent receives some reward (which is a real number) from the environment. This cycle either ends at a terminal state (episodic task) or continues for an indeterminate time period (continuous task). Each run from start to finish is referred to as an episode (or alternatively an epoch).

At the core of TD learning algorithms is an update which alters an agent's value function according to the reward signal from the environment after every step. A designer sets the reward function and it is important that the reward function tells the agent what to do not how to do it. If the reward function is for a game of chess then it should give reward at the end states of win, lose or draw rather than after taking opposition pieces, as that would increase the chances of introducing an *implicit game* and crippling the learning ability. See Section 2.2.4.5 for the definition of an implicit game and background on the reward and value functions.

[3]The state can also be viewed as a signal originating in the environment and received by the agent.

A designer chooses the state representation (Section 2.2.4.2), the actions available (Section 2.2.4.3), and the environment. The environment also determines the state transitions. The reward function is often nominally considered to be part of the environment too. All of these factors will affect the agents' learning ability and their final learned behaviours and so must be carefully considered.

Reinforcement learning can be difficult to get the best performance from and, "in reinforcement learning one typically has to settle for approximate solutions." [Sutton and Barto, 1998]. This is not of *great* concern in computer games where some random behaviour is often considered a good thing because it introduces some changeability to the gameplay and, with probabilistic action selection (Section 2.2.4.4) any random action will be graded so that very bad actions are very unlikely to be taken whereas sub-optimal but still viable actions may be taken very occasionally.

2.2.4.1 Sarsa

All the agents in the experiments in this thesis use the well known and well documented *sarsa* learning algorithm (see [Sutton and Barto, 1998] for a comprehensive discussion of sarsa and its background and uses). Sarsa is relatively easily understood and relatively simple to implement which makes it a good option for a game programmer's first foray into reinforcement learning. In practice, in the experiments in this thesis the sarsa algorithm regularly converges to a satisfactory solution. Group utility functions should work with other reinforcement learning algorithms such as Q-learning or Monte Carlo methods. Reinforcement learning is a well established technique and this thesis does not alter the fundamental sarsa learning algorithm (shown in Equation 2.2). The contributions of this thesis lie in the novel utility functions. This thesis also serves as a practical example of reinforcement learning in game settings and contributes theoretical discourse on several problems faced.

Sarsa is called an on-policy learning algorithm. This is because it updates estimated values for each state-action pair using experience actually gathered as it follows its policy (see Section 2.2.4.4). For each step, a sarsa agent takes the received reward at that step and uses it to update the previous state-action pair that it visited according to Equation 2.2 (From [Sutton and Barto, 1998]). Because sarsa updates its value function using a state-action pair that only has an estimated expected return value, it is said to bootstrap. Bootstrapping refers to the way that the algorithm uses estimates to learn with. The agent uses its value function to learn even if its value function has not yet converged, and indeed even if its value function has just been initialised with ran-

dom values. Typically a sarsa learning agent's policy will be greedy or ε-greedy when selecting actions based on its value function. This means that as soon as the agent's value function is updated, the agent's behaviour will reflect that update. In addition, because sarsa is an on-policy learning algorithm, updates are applied to exactly the areas of state-action space that are visited. The fact that sarsa bootstraps and is on-policy means that agents adapt their behaviour as quickly as possible as learning takes place. This quick changing behaviour in the areas visited means that agents will quickly learn a reasonable action to take in often encountered situations. Furthermore, sarsa is suited to computer games because agents can learn as they act in real-time in the game world. Like all temporal difference learning methods sarsa is able to learn during an episode and not just at the end of episodes. This opens up the opportunity for in-game learning. As well as this potential, sarsa learns good control solutions when given as much time as is needed to reach some predefined performance level.

The backup equation is key to any reinforcement algorithm. The backup algorithm for sarsa is shown in Equation 2.2. The backup equation determines how an agent updates its value function given one time step of experience. The full algorithm for sarsa is given in Section 4.2.4. As the reader can see from Equation 2.2 the backup operation uses the current state and action, the reward at $t+1$, and the state and action at $t+1$ in its update calculation. These five parts of the calculation ($s_t, a_t, r_{t+1}, s_{t+1} and a_{t+1}$) are what gives sarsa its name.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.2}$$

In Equation 2.2 Q is used to represent the value function. $Q(s_t, a_t)$ denotes the current estimated return for being in state s and taking action a at the discrete time step t and following the current policy afterwards (see Section 2.2.4.4 for a description if an agent's policy). α denotes the learning rate which is a percentage indicating by how much an agent should update its expected return values. An α of zero means no learning and α of 0.0001 means slow learning compared to an α of 1. High α values can lead to oscillation and hence learning does not converge whereas, a low value for alpha will take a long time to learn and may only find local maxima. Often learning experiments start with an α of say 0.1, and multiply the α value by a constant value of say 0.01 as the experiment continues or, set the α value as α/t where t represents the time step. Degrading an α value means local maxima should be overcome and also that learning will eventually converge. Some experiments that continually need to keep learning run with a constant low α value of say 0.01. Mileage may vary and trying a

number of values for α is recommended. Reinforcement learning is robust and α and γ need not be set precisely for agents to learn well. γ denotes the discount rate which scales the effect the rewards received in the future. If $\gamma = 0$ then the agent will only be concerned with immediately collectable reward whereas, the higher γ becomes the more far sighted an agent will be when choosing an action. A γ of 0 means the agent will by myopic and will fail to learn tasks where reward is delayed, or possibly beyond negative reward space. A γ of 1 will lead to an agent that is only concerned with the ultimately returned reward. A γ value of 1 does not make sense for continuous tasks where a γ value less than 1 should be used. For computer games characters γ can be used as a sliding scale from dumb characters that cannot plan into future situations at $\gamma = 0$ to "clever" characters that can "understand" consequences as γ approaches (but, for continuous tasks, does not reach) 1.

2.2.4.2 State Representation

> "Choose your state well and you have (half) solved your problem." - *A wise supervisor*

The *state*, as far as reinforcement learning is concerned, is a data structure holding the information the agent knows about the environment it is situated in. Ideally an agent's state representation should satisfy the Markov property as discussed in Section 2.2.3. The state representation is the way in which the information about the state is kept. The state representation is usually updated at each time step (the sense part of the sense-think-act cycle). The larger the state representation (i.e. the more information it stores), the greater the size of the state space, and hence the time to explore each state will be longer. For this reason a designer must carefully consider what should appear in the state. Too much will slow learning (possibly so much that convergence will not take place in any reasonable amount of time) and too little may deprive the agent of information it needs to learn. When too little information is represented in the state there is said to be *hidden state*. This hidden state means that instead of a Markov decision process the agent is actually trying to solve a partially observable Markov decision process. Attempting to find a solution for a partially observable Markov decision process using reinforcement learning will likely fail if the hidden state obscures significant features crucial to understanding the dynamics of the environment. Methods do exist to use with reinforcement learning to overcome hidden state if one wished to use them. However, mentioning hidden state here just serves to illustrate what a state representation should contain.

As a simple illustrative example, a blindfolded person, wearing ear plugs and attempting to play snooker suffers greatly from hidden state. They can feel the table edges and the cue but would find it difficult to learn to play snooker. With a large, dynamic environment as may be found in a computer game featuring many other characters, hidden state will be a problem. There are two solutions:

- the state representation can be expanded to take into account the distinctions needed to cover the hidden state. This will increase the state space size which will make learning take exponentially longer[4].
- learning can succeed with hidden state and it may be beneficial to trade off some accuracy lost to hidden state for the benefits of a smaller state space (note that this approach failed with the RETALIATE system in Section 2.1.4).

At present there are few examples of state representations used for computer game characters. It is robotic football that provides the most useful existing ideas on how to represent the state of an agent in a highly dynamic environment. Stone and Sutton [Stone and Sutton, 2001] cleverly limit the state-space to encourage short convergence times by restricting a player's state to a certain number of nearest features arranged by distance. In their state representation, a player is only aware of the nearest two other team-mates, the nearest two other opponents and the ball. In this way players are able to play the game reasonably well without an impossibly large state-space. This style of state representation is used in all the experiments in this thesis. This approach of labelling features (in this case players) by their distance allows agents to better utilise their experience. So long as the team-mates act similarly to each other and the opposition players act similarly to each other then the agents can apply what they learnt about one team-mate or opposition player to other team-mates or opposition players. It can cause problems if the number of features represented is inadequate for the task being learned. For the robotic football example above, a player may not be able to consider passing the ball to a team-mate that is only the third closest. In doing this the designers may have severely limited the player's ability to play football. There is a lot of hidden state caused by the actions of the players not in the state representation. As long as these hidden players do not significantly affect the optimal decisions, then the agents will still be able to learn to play football well. This is a calculated trade off between the quality of the solution found and learning speed. Computer games can

[4]Although there is the possibility of approximating the value function and escaping the curse of dimensionality as in Section 6.

often be similar to robotic football with many characters that are in the same class. For example, in the World of Warcraft [Blizzard Entertainment, 2001] game one may calculate that if all dwarf warrior's act similarly enough to treat them as one type, then the state representation could only consider the nearest n number of dwarf warriors to make the learning easier. Of course, all dwarf warriors may not act the same but the trade off in learning speed may be worth it.

The idea of an agent-centred state representation filtered by some measure of distance allows one to trade the possibility of lower asymptotic performance for easier and/ or quicker learning. This will be a useful method in computer games and so it is used for the experiments in this thesis.

2.2.4.3 Actions

For reasons of simplicity, grid-worlds have been the dominant environment used in reinforcement learning research. A grid-world is a set of two-dimensional squares connected by the actions north, east, south and west. These worlds are very useful for highlighting problems to be solved with reinforcement learning. Grid-worlds bear little resemblance, visually or in dynamics, to modern computer games which are often complex continuous spaces. Using the traditional north, south, east and west actions is aesthetically unrealistic since characters do not generally move crab-like on a grid, but approach features directly, as a human might.

Deictic actions are actions that reference a feature in the agent's environment. An example of a deictic action definition might be, *move to the left of the rock*, rather than just, *move left*. Computer games allow us to use these high-level deictic actions to make the agents actions appear much more realistic. These actions are higher level and may need to rely on a navigation layer such as the ubiquitous A* algorithm.

We can assume that Orkin uses deictic actions in F.E.A.R. from an academic paper pre-dating its release [Orkin, 2004]. In it Orkin gives a pseudo-code example of a plan which is:

> "EliminateEnemy - Goal: Goto(pipe), PickupWeapon(pipe), Goto(target), Attack(target)" [Orkin, 2004]

The reliance on grid-world environments can affect the ability to use some learning algorithms in computer games. As an illustrative example COIN (COllective INtelligence) would be difficult to use in a continuous world because of its reliance on

absolute coordinates (sometimes referred to as GPS coordinates) and absolute actions. COIN and the problems associated with it when applied to computer games are discussed in Section 2.3.3.

One can see that reinforcement learning's association with grid-worlds cannot carry over to computer games. There are many reinforcement learning examples used without absolute states and actions, but the designer must still keep in mind that some algorithms may be shackled by their creation in a grid-world.

2.2.4.4 Action Selection

An agent's *policy* dictates what action it should take based on its current state and previous experience. The policy, often denoted by π in literature, chooses between possible next actions given the value function. The *policy* refers to an agent's policy at any one time as it may change over that time but the term is also used more generally to refer to methods of choosing actions. For example, a common policy is simply taking the action with the highest expected reward. However, this does not take into account the trade off between *exploration* and *exploitation* that is so important to optimising the agent's behaviour. If the agent is purely greedy in selecting the action with the highest expected reward then it may never investigate any other actions other than the first action it takes that gives a reasonable return. There are two common methods of ensuring sufficient exploration whilst achieving a good asymptotic performance. The simplest is the ε-greedy method. ε is scaled from an initial value (commonly 1) at an agent's first step to (usually) zero at some point in the future. At each step the agent generates a random number between zero and one, and if this number is greater than ε, the agent will choose the action with the highest expected reward. If the random number is below ε then the agent will select an action randomly. By using the ε-greedy method we can ensure that an agent explores enough before exploiting their experience. The initial value, end value and scaling speed must be set for each problem with some trial and error.

Agents can choose actions probabilistically, as in for example, *softmax* action selection. Softmax action selection uses (for example) the Boltzman distribution to chose action a from state s at step t with probability:

$$\frac{e^{Q_t(s,a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(s,b)/\tau}} \tag{2.3}$$

where τ, called the *temperature*, is a positive value controlling how greedy or ran-

dom action selection is. High temperatures mean action choice is practically random and selection becomes greedy as τ tends to zero. Setting the temperature with any degree of certainty is difficult for not well understood tasks. The effect of the temperature is dependent on the final resting utility values of each action from each state. Without knowing these converged values it is hard to achieve the effect one wishes when setting the temperature.

ε-greedy action selection is sufficient in many reinforcement learning tasks. It is a simple method that chooses either the top rated action or randomly amongst the other actions. Softmax selection is more refined as it chooses actions probabilistically based on their utility values. This allows for less than optimal actions to be taken occasionally which would be ideal for avoiding repetitive behaviours in computer game characters with little risk of random actions spoiling consistent behaviours. However, using softmax requires a solid understanding of the powers of *e* and knowledge of the expected value function. Some form of softmax action selection would be good for computer games but this thesis experiments with the utility functions and so it is near impossible to predict the expected value function making setting the temperature guess-work. And so, the experiments in Chapters 4, 5 and 6 use ε-greedy action selection.

2.2.4.5 Utility Functions

Reward and value functions both indicate how useful it is to perform each action from each state[5]. Both functions are often referred to as utility functions as they provide a level of usefulness given a state-action pairing

The reward function is usually considered to be (nominally) part of the learning environment and is responsible for giving reward signals to agents after each time step. The reward function is normally human designed and is sparse in nature with only points of interest noted. For example, if a goal is to reach a certain square in a grid world then all squares could be set to zero reward except for the goal square which could be set to one. Sometimes such a reward function might be set to negative one at each non-goal square and zero at the goal square. This encourages agents to find the goal square as quickly as possible. However, with an appropriate discount rate for an episodic task, this would not be necessary as the agent should find the quickest route. An example reward function for such a task is shown in Figure 2.3. Sutton and Barto advocate that the designer should use the reward function to tell agents what task

[5]This is the case when action-state values are being learned but, sometimes only the state values are learned so the utility functions would only give values for states.

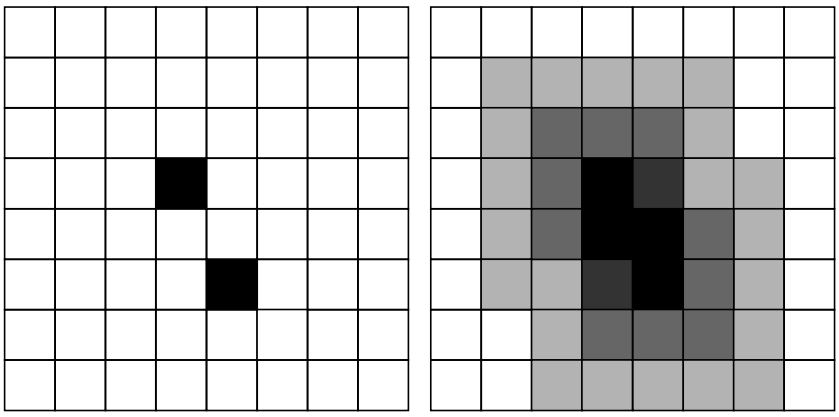


Figure 2.3: The sparse reward function on the left and the more populated value function on the right.

to learn, not how to perform that task. If a designer attempts to give advice on how the task should be performed in the reward function, they risk introducing an implicit game. An implicit game occurs when a human designer accidentally asks an agent to learn a different task than the one desired. It is best for the designer to set the reward at important points in the state-action space and let the agents do the laborious work of filling in the details. Each agent reads the reward signal from the reward function for learning and stores this knowledge in its own value function. The process of observing the reward and storing it in a value function is central to learning.

Everything an agent knows about the task is kept in its value function which can be thought of as a local version of the reward function which, through learning, changes from its initial blank or random state to a less sparse version of the reward function. One can see an example value function in Figure 2.3, where the reward has spread across the state-action space.

An agent's value function is updated after each time step using a *backup* function. For sarsa, this function is shown in Equation 2.2. The sarsa backup equation updates the value for the current state and action ($Q(s_t, a_t)$) using the value of the current state and action ($Q(s_t, a_t)$), the reward value received after the transition (r_{t+1}) and the value of the next state and action taken by the agent ($Q(s_{t+1}, a_{t+1})$). As the reader can see, the function that updates the value function uses the value function itself in its calculations. The use of itself to update itself is called bootstrapping (as discussed earlier in Section 2.2.4.1). The value function must be initialised in some way and the three most common initial values are:

- All values set to a neutral value - most commonly zero. One must keep in mind that as sarsa bootstraps and is on-policy, the value function is used in updating itself. A neutral value should not affect the learning in any way.
- All values set to positive and negative random numbers. Most likely the most neutral method of initialising the value function.
- All values set optimistically - this is an interesting case as it may affect an agent's exploration if desired.

If the designer chooses, they can set the initial values of the value function to be higher than the likely maximum reward the agent could encounter during the task. Thus, as the agent explores the state-action space the value function is "pulled down" from their optimistic values leaving unexplored areas at a high value, i.e. the agent is optimistic regarding what it expects to find. If the agent uses its value function to select actions while learning (as *on-policy* algorithms such as sarsa do) then it will not cease exploring until it has explored most of the state-action space even if it is entirely greedy in its action selection policy. Note that some of the state space may remain unexplored even using optimistic starting values if the agent approximates the value function using methods listed in Section 2.2.4.7. This is because with many forms of function approximation the surrounding areas of an updated state-action pair will be affected too.

Most simply the value function for an agent can be kept as a (usually large) two-dimensional table with one dimension each for the state and action. The the entries are the utility values. Of course, as the size of the state-action space grows the value table becomes very large and grows exponentially when new dimensions are added to the state representation. Approximating the value function using methods described in Section 2.2.4.7 is the way to overcome the crippling effect on processing resources of a large value table. The experiments shown in Chapter 5 use a table to represent the value function. The experiments in Chapter 6 approximate the value function.

Some specific reward functions such as the team utility function are well known and are discussed next. The wonderful life utility function is a reward function used in COIN (COllective INtelligence) which is covered in Section 2.3.3. Section 2.2.4.7 gives some common methods of approximating the value function which is beneficial in two ways. As already stated, keeping the value function as a table does not scale well, as the size of the table grows exponentially with the number of dimensions of the state representation. Secondly, using an approximation method makes better use of

learning experience as an update to the value function also updates neighbouring areas of the state-action space.

2.2.4.6 The Team Utility Function

The team utility function is a well known, well understood reward function. Each agent in the environment is given as its reward, the sum of all of the agents' individual rewards. Using the team utility function, agents can learn that what is good for the team is good for them and it can enable multiple agents to learn to work as a team. The team utility function has some well known deficiencies. Chief amongst them is its poor signal-to-noise ratio. The reward signal can be weak compared to the noise of the function. For example, imagine a game of football where every player on the team is rewarded when a goal is scored despite the fact that one of the players may have done nothing at all to contribute or may even have been detrimental to the team effort. The noise makes learning more difficult and can make learning take longer to converge to a reasonable solution.

The other deficiency with the team utility function is that it does not encourage cooperation. Rather it encourages competition between agents to do the best for the team. This problem is highlighted in [Wolpert and Tumer, 2000] and belies the fact that learning agents do not consider, they just find reward. Designers must be careful if, as in this thesis, they experiment with novel reward functions. The agents may not be learning what the designer thought they would.

The team utility function creates a relationship between all of the agents; one of team-ship. By using the sum of the agents' individually received rewards as each agent's reward to learn with, the agents learn to be team-mates. The central novel idea behind the goal of altering reinforcement learning to allow more control is to swap the summation function for other mathematical functions. Other functions of the agents' reward will produce other relationships. The team relation is possibly useful in games and this thesis introduces an equality (or balanced) relationship. The new relationships and the functions behind them are shown in Chapter 3.

2.2.4.7 Approximating the Value Function

> "All bets are off" when it comes to using non-linear methods to approximate the value function. [Durnan, 2004]

Reinforcement learning agents often store their value function as a table with each row holding a state representation, a representation of an action and the actual expected

reward value. As the size of the combined state-action space increases, the potential size of the table increases exponentially. This is referred to as *the curse of dimensionality* or *state space explosion*. The size of the state space may need to increase to represent more information required by an agent in order to learn a more complex task. In the experiments in Chapter 5 it was necessary to use a larger state space than in the experiments in Chapter 4 so that the agents could learn the set task. Agents must visit and revisit states many times to learn a reasonably optimal policy thus it becomes increasingly difficult to do this with a very large state-action space and hence table. This table needs to be stored in memory for each agent in an experiment which quickly runs out as the computer game also takes up memory. Aside from memory the time taken to visit each and every state a large number of times becomes too great to be practical. The standard way to alleviate these scale problems is to use some form of function approximation.

It is possible to replace a tabular value function with a parametrised function which approximates the value function being learned. When using a table based value function, the time and size requirements grow exponentially with the size of the state-action space. A parametrised approximation, providing it is sufficient to represent the required complexity, might not grow at all. Function approximation also brings with it another benefit: *generalisation*. When an update is applied to a table based value function, only the expected value for one state-action pair is updated whereas any function approximation method update will affect not just the current state-action pair but its neighbours state-action pairs as well. Function approximation can be thought of as a rubber surface in n-dimensions which when updated at one point moves and stretches in nearby areas.

There are many approaches to function approximation but the most common are tiling the state representation with overlapping regions and using neural networks. Tile coding is a non-linear function approximation method in which the state or state-action space is split into many tilings. Each tiling is an exhaustive pattern of tiles across the state or state-action space. The tiles can be any shape in the dimensions of the state (state-action) space. The tiles can be shaped to best reflect the direction of generalisation wanted. For example, two-dimensional concentric circles around a point would generalise across direction. The direction would be ignored and only the distance would be significant. One could then place a separate tiling "on top" of the concentric circles using a cross-hair shape to split the two-dimensional plane into quadrants thus adding some information about direction but still generalising within the four

quadrants. In this way it is possible to control the direction of generalisation across dimensions and the resolution with the size of tiles and number of tilings. Each tile is a binary feature and has an associated parameter. The value function's output is calculated as in Equation 2.4 where n is the number of parameters (and the number of features as there is one parameter per feature), $\phi_{s,a}$ is the state-action feature and $\theta_t(i)$ is its corresponding parameter at time t. It is the θ vector that is updated to approximate the value function. The gradient to follow to improve the approximation is also simple and is shown in Equation 2.5. The parameters are updated according to some value of α, the *learning rate*, as in the tabular version of Section 2.2 and the neural network version explained below.

$$V_t(s,a) = \sum_{i=1}^{n} \theta_t(i)\,\phi_{s,a}(i) \tag{2.4}$$

$$\nabla_{\vec{\theta}_t} V_t(s,a) = \vec{\phi}_{s,a} \tag{2.5}$$

Neural networks are too large a subject to be discussed fully here. However, it is sufficient to say that neural networks can be, and have been used successfully to approximate value functions during reinforcement learning. A neural network is a hierarchical collection of nodes and associated weights. The nodes are arranged in rows, with a row for all the inputs to the network, any number of hidden rows, and a row for the outputs of the network. When used with reinforcement learning, the inputs are the state-action space encoded in some way to be acceptable (i.e. they must be quantitative not qualitative values as well as following some other rules of thumb) and the output row normally consists of one node whose output is the expected return value. Each node has an associated weight, and each row has a bias weight which is not attached to any node. When learning an approximate value function a standard back-propagation algorithm is used. When presented with a vector of inputs and an expected output, the error between the expected output and the actual output is calculated. This error is often called the TD - temporal difference - error in reinforcement learning. This TD error is back propagated through the network of nodes and, at each node the associated weight is altered so as to minimise the error seen with the same input vector and expected output. The weights are altered part-way along the calculated error vector by scaling the update by the learning rate (often referred to as α) - a scalar between zero and one. The error is only corrected part of the way to allow for smoother learning of an approximation (not an exact representation which is pointless

to aim for) and stochastic variations. Over many iterations of presented input vectors, expected output values, and back-propagation of the error values, the weights should converge to optimal or near optimal values so that the value function is approximated in a reasonable way. There are a great deal of variables which may affect the application and success of neural networks for value function approximation in reinforcement learning. The network topology, starting weights, number of training iterations, back-propagation algorithm and function to be approximated, all have an effect. There are rules of thumb for network topology that tend to work in a wide variety of experiments (see [Sarle, 2009]). For example it is usually unnecessary to use more than one hidden layer and, the number of nodes in the hidden layer should be enough to make sure generalisation happens but not too many to over-fit the value function. In particular this advice concerning the number of hidden nodes and generalisation properties relies heavily on the number and type of training iterations. These are just rules of thumb and much experimentation is usually warranted to find values that work.

In a typical neural network, each hidden and output node takes as its inputs the weighted sum of the previous layer's node outputs and a weighted bias node with a constant output of one (that is, one, which is then weighted). This weighted sum is then passed through each node's activation function such as that shown in Equation 2.6 - a squashing, sigmoidal function. Squashing means that the output of the function is squashed to between zero and one and sigmoidal refers to the function's *s* shape - it is actually the *logistic function*. By altering the weights through back-propagation of the TD error the network can be trained.

$$\frac{1}{1+e^{-input}} \tag{2.6}$$

Much success has been had using non-linear methods such as tile coding and neural networks. Most of the guarantees for TD-learning gradient descent methods using function approximation apply to on-policy learning using linear approximation methods. The advantage of non-linear methods is that they are potentially more capable of representing complex target functions. For example, if a value function relies on interactions of more than one state feature then the combination of these features must be made explicit as an additional input to the vector of inputs for the linear function approximator. This might occur in a situation where A and B are both in the state vector and high values of A are a good indicator only for certain values of B otherwise, they are an indication of low expected return. Non-linear methods of function approximation are often able to handle interdependence between state features without explicit

help. There is a trade off between the theoretical guarantees of linear methods and the increased power to represent complex functions of non-linear methods. In practice it is fair to surmise that it can be more difficult to train non-linear than linear function approximation but that reinforcement learning using non-linear function approximation can converge to usable value functions. The experiments in Chapter 6 use neural networks, a powerful, non-linear function approximation method.

Probably the most famous reinforcement learning agent - Tesauro's *TD-Gammon* (through [Sutton and Barto, 1998]) - used a non-linear neural network to approximate the value function. The backgammon game has a state-action space far too large to use a table-based value function. TD-Gammon used a standard neural network much like that employed in Chapter 6 - three layer (one hidden) network with one output node representing the expected return[6], forty hidden nodes and one hundred and ninety-eight input nodes representing the board state (in its initial incarnation). TD-Gammon emphatically shows that for some extremely difficult tasks reinforcement learning with a neural network approximating the value function can prove effective.

However, reinforcement learning using approximated value functions can be difficult to get working and, the fact that both reinforcement learning and neural networks are opaque systems makes it difficult to track down problems with the implementation or design. This can result in worse performance or odd learning behaviour when compared to the relative simplicity of tabular value functions [Burrow and Lucas, 2009] and users are advised to start with small problems using table based value functions and then move on to using function approximation when the problem is well understood [Sutton, 2001]. Otherwise, much time can be wasted attempting to analyse the system through its results which, from experience, can lead one down many blind alleys. In summary, using function approximation to approximate value functions:

- can be used to generalise experience across the state-action space and reduce the size of the value function representation;
- is a powerful method as evidenced by TD-Gammon;
- is difficult.

[6] Actually three output nodes were used. One to estimate the expected return, and the two extra nodes to estimate the chance of the *gammon* or *backgammon* finishing states.

2.3 Related Research

2.3.1 Finding Sweet Spots

Reinforcement learning agents do not pay attention to how the game should be played (i.e. how it was designed to be played), rather they learn how to achieve as much reward as possible. They can detect potential opportunities for human players to capitalise on unthought-of simple ways to succeed at games that, if known about prior to release would have been closed by the game designers. The most obvious example of these "sweet spots" can be seen in football games where there are certain places to shoot from that will more than likely guarantee a goal. Sweet spots, whilst initially being very exciting for the player that discovers one, usually strip the game of all purpose and ruin the game playing experience.

Work has been done to find these sweet spots in the commercial game FIFA'99 [Chan et al., 2004, Denzinger and Winder, 2005]. The work uses the term sweet spots as well as *unwanted behaviour*, indicating that the sweet spots are to be found and removed or at least checked by a human designer. In the research, evolutionary methods are used to detect sequences of actions which likely lead to a goal being scored with a high probability. The algorithm finds sequences of actions such that if they were repeated again and again by a human player, would more than likely guarantee a landslide victory every match.

This research shows that search based optimisation can be used in a commercial game. However, it is not an obvious use and we are yet to see common use of evolutionary methods or reinforcement learning in commercial games. One academic group has taken a different approach - rather than trying to get academic artificial intelligence into a commercial game they have produced a "commercial quality" game based entirely on an academic artificial intelligence method - evolutionary methods applied to neural networks in the game NERO.

2.3.2 rtNEAT and NERO

rtNEAT is an on-line, evolving neural network methodology that allows the weights and topology of the networks to change [Stanley et al., 2005]. NEAT has been proven to work well in several domains but most pertinent here is its real-time adaptation (the "*rt*" in rtNEAT) for use in NERO, a game that is built on the ability of multiple agents to be trained by a human in real-time. NERO allows the fitness function to be par-

tially set by the game player by providing a set of sliders which rate the importance of certain behaviours such as avoiding being shot or, shooting other characters. NERO's agents do not require a state representation *per se* but do require inputs into their neural networks. These inputs are similar to a state representation used in reinforcement learning. If NERO's agents had a state representation it would appear that it would be be agent centred.

The rtNEAT system is somewhat of a landmark case in academic computer game development. It was, and still is being developed, entirely by students from undergraduate to doctoral level at the University of Austin, Texas and has produced a commercial standard game called NERO [University of Texas at Austin, 2009]. The project shows that it is possible to successfully combine academic artificial intelligence and game development. rtNEAT is novel artificial intelligence research that "could not have been accomplished without an off-the-shelf game engine as a test-bed." [Gold, 2005]. That off-the-shelf game engine is Torque [Garage Games, 2001] and has been used for the experiments in this thesis. rtNEAT's researchers give five reasons why reinforcement learning techniques should not be used for real-time adaptation in computer games [Stanley et al., 2005]:

- *Large state/action space means that RL can take a long time to converge.* This is true, with any highly complex task with a large state and action space reinforcement learning will take a long time to converge. However, evolutionary approaches are not immune to complexity issues. There are ways in which to reduce the state/action space in reinforcement learning. For example, composite actions reduce the action space and, function approximation can generalise an agent's experience to provide more efficient use of the search time and manage large state spaces. A large state-action space means there is a greater area to search and, as genetic algorithms and reinforcement learning are both search based optimisers, they should be affected in similar ways.

- *Diverse behaviours.* The authors maintain that homogeneous reinforcement learning agents will all converge to the same behaviour as there are no explicit attempts to maintain diversity. Again in evolutionary methods there is no explicit maintenance of diverse behaviours[7]. There are ways of learning diverse behaviours using reinforcement learning and in this thesis programming by reward is advocated for this reason (Section 2.3.4).

[7]Using evolutionary methods is often a trade-off between diversity and fitness

- *Consistent individual behaviours. Characters should behave realistically during their lifetime which means not taking random exploratory actions as in reinforcement learning.* It is true that reinforcement learning agents must explore the state/action space. This is also true of evolutionary methods. Both techniques are search based optimisers and must therefore perform sub-optimally for some time during exploration. The characters in NERO do act consistently throughout their lives but their lives are extremely short. Fast evolution is only achieved by the quick turnover of agents in the game world. Also, the ability for an agent to adapt during their lifetime could be beneficial and a small amount of exploratory actions could be seen as more realistic behaviour. Especially if reinforcement learning agents are used with probabilistic action selection, as those exploratory actions are likely to be from states where the obvious best action is not clear. A common theme in commercial game research and in particular in the articles of AI Programming Wisdom 2 [Rabin, 2003a] is filtered, or controlled, randomness. That is behaviour that is *just random enough* but not entirely random. Optimal action choice throughout an agent's lifetime may not be needed for game characters.

- *Slow adaptation.* The slow speed at which RL converges is well known, is related to the first point above and can be improved by reducing the state-action space or using methods mentioned above. Convergence speed can be greatly improved by continuously backing up sample experiences whilst waiting for real experience (as described in Chapter 9 of [Sutton and Barto, 1998] and used in [Konidaris and Hayes, 2005]). Whereas rtNEAT quickens adaptation by evolving from a limited evaluation period, reinforcement learning with planning (as defined by Sutton and Barto [Sutton and Barto, 1998]) speeds learning from little real feedback, both speed learning or evolution at the possible expense of accuracy. Slow adaptation will be a problem for any on-line search based optimiser but, would not be if the agents are trained prior to being used in a game. In a game like NERO where the agents learn in view of human players and the learning is integral to the game, steps will have to be taken as in rtNEAT to work around slow adaptation. It helps to have a fast adapting algorithm in the first place, like rtNEAT, but there exist algorithms for reinforcement learning that adapt faster than sarsa which could be used forsaking simplicity of implementation.

- *Memory of past states.* The violation of Markovian properties does not lead to a breakdown of reinforcement learning methods. Reinforcement learning environments should be as "Markovian as possible." The more Markovian they are the greater chance of eventual and timely convergence and optimal asymptotic performance. In other words:

  > Even when the state signal is non-Markov, it is still appropriate to think of the state in reinforcement learning as an approximation to a Markov state. In particular, we always want the state to be a good basis for predicting future rewards. ... Markov states provide an unsurpassed basis for doing all of these things. To the extent that the state approaches the ability of Markov states in these ways, one will obtain better performance from reinforcement learning systems. For all these reasons, it is useful to think of the state at each time step as an approximation to a Markov state, although one should remember that it may not fully satisfy the Markov property. [Sutton and Barto, 1998].

rtNEAT has succeeded where many academic AI methods have failed in getting into a commercial standard computer game. It may not have been done yet, but the properties of reinforcement learning do not preclude it from being able to work well in a game such as NERO.

One interesting development of rtNEAT is KB-NEAT [Cornelius et al., 2006] in which a character's initial neural network can be initialised from a finite state machine. This allows the characters to start evolving from a sane, rather than random, point. The characters can then start with a reasonable behaviour that is as good as that specified by a game designer as a finite state machine which means evolution starts from an already advanced stage. There may be some slowness in evolution or drop in asymptotic performance with a population all starting like this as the randomness inherent in the population accounts for the exploration that reinforcement learnings' random actions serve.

2.3.3 COIN

As mentioned in Section 2.2.4.6, the team utility function suffers from a poor signal-to-noise ratio. That is, it is possible and likely that an agent will be rewarded wrongly because of the way the reward signal is constructed. This signal to noise ratio will affect group utility functions in the same detrimental manner that it affects the team utility function. This is a well understood problem and so a method of overcoming it

to some extent would be welcome. For this purpose COIN (*CO*llective *IN*telligence) was investigated as it promises to improve the signal-to-noise ratio.

COIN uses substantially more computationally expensive utility functions to get a reward signal with a better signal-to-noise ratio when multiple agents are learning in the same environment [Wolpert and Tumer, 2000, Wolpert and Tumer, 2002a][8]. COIN's The wonderful life utility function is considered here. The main thrust of COIN is that the team utility function encourages competition rather than cooperation whereas the wonderful life utility function actually does encourage cooperation. When using the team utility function each agent is given the team's overall reward and so the agents learn that what is good for the team is good for them. However, each agent is acting individually and behaves selfishly. Thus we may get the situation where goal state A is simple for agents One and Two to achieve, though slightly quicker for agent Two to reach, whereas goal state B is only achievable by agent Two if agent Two has not achieved goal state A first. In this situation it is very probable that agent Two will always move towards goal state A beating agent One in the process. This behaviour is reinforced and goal state B is never achieved as it is unachievable by agent One at all and agent Two after reaching goal state A. This is an example of *the tragedy of the commons*, where individual greed undermines team achievement. The wonderful life utility function uses a history of the agents states and can replay the history without an agent's presence to more accurately calculate the "good" that that agent is doing at each time step. It does this by keeping a table of the agents' states at each step. Then one can see for example, that although agent One has already visited (and removed) a reward, if agent One had not existed then agent Two would have achieved that reward anyway. Thus it may make sense for agent One to attempt to achieve another reward, possibly that agent Two could not achieve. In this way the wonderful life utility function detects situations where competition arises. In these states the agent that should have avoided the competition to do better actions elsewhere is penalised through the reward signal. In the example given above the wonderful life utility function would detect that if agent Two had not existed then agent One would have reached goal state A leaving agent Two free to reach goal state B to achieve a higher overall reward. The wonderful life utility function for each agent is calculated (as in Equation 2.7) by taking the world utility (the summed reward of all the agents) at the current time step

[8]There is a substantial body of research around COIN. [Wolpert and Tumer, 2000, Wolpert and Tumer, 2002b, Wolpert et al., 2000b, Wolpert et al., 2001, Wolpert and Tumer, 2001, Tumer and Wolpert, 2000, Wolpert et al., 2000a, Wolpert and Lawson, 2002, Wolpert et al., 1999, Hoen and Bohte, 2003, Healing, 2005]

minus the world utility of all the agents except the agent in question. In Equation 2.7 WLU_η is the wonderful life utility for agent η; G is the function for calculating the total world utility; ζ is the set of joint actions for all the agents in the system; $\zeta_{\widehat{\eta}}$ represents the function for calculating the total world utility for the joint actions of all the agents except agent $\widehat{\eta}$; and CL is a clamping function that sets agent η's actions to nil (or null or zero). The function is named after Frank Capra's *It's a Wonderful Life* in which the main character is shown what good he has done in his life by viewing his town's history as it would have been without his existence.

$$WLU_\eta \equiv G(\zeta) - G(\zeta_{\widehat{\eta}}, CL_\eta) \tag{2.7}$$

Investigating the use of the wonderful life utility function in computer games would be a useful, interesting and novel contribution. Improving the signal-to-noise ratio for these simple utility functions (such as the team utility function and the group utility functions introduced here) would make reinforcement learning more suitable for use in computer games. It was initially planned that this thesis would be an investigation into the use of COIN for multiple computer game characters. However, COIN is not so suitable for use in computer games without considerable modification. The history that COIN utility functions rely on to calculate the wonderful life utility requires an absolute state representation and absolute action descriptions. This is so that the history makes sense when replayed. Wolpert and Tumer point out that the difference part of Equation 2.7 ($G(\zeta_{\widehat{\eta}}, CL_\eta)$) is fictional, and does not take into account the environment's dynamics [Wolpert and Tumer, 2000]. That is, the history does not need to be an accurate representation of the exact system without the presence of an agent. However, there are two constraints not mentioned elsewhere (to the author's knowledge). Without significant modifications, the COIN agents cannot use an agent centred state representation or deictic action descriptions. These would be required for a continuous task in a highly dynamic environment such as computer games. Section 2.2.4.2 explains why an agent centred state representation is suitable for computer games and Section 2.2.4.3 gives reasons why deictic actions should be used in games also. However, COIN's history relies on the fact that the state is absolute not relative. If one agent does not exist then other agents must be able to reach rewards that the missing agent would have achieved. Representing this with absolute coordinates for example is simple. However, when relative coordinates are used it becomes unclear how to represent rewards across agents without resorting to absolute coordinates. The same problem applies to deictic rather than absolute actions. It is a difficult or impossible

problem to solve without suffering excessively from state-space explosion and unnatural looking behaviour that the agent centred state representation and deictic actions were introduced to combat.

2.3.4 Programming by Reward and SHARSHA

This thesis adheres to the programming by reward concept that holds *separation of skills from preferences as its key tenet*. The separation hypothesis states that agents (be they computer game characters or people) have similar sets of skills but have varying and distinct preferences for executing those skills and, that it is these preferences that produce individual behaviour. In this way a great number of observably distinct individual behaviours can be learned by agents with the same underlying skill set, thus saving the human design time involved in skill development.

When using programming by reward in a reinforcement learning framework, the distinct preferences are ultimately determined (through learning) by more than one human end-user designed reward functions. Learning agents pick from a similar pool of available actions to learn to achieve a quantitative amount of reward as specified by some of their reward functions, qualified by some other of their reward functions. In other words, some of an agent's reward functions specify what task the agents should do and some of its other reward functions specify qualities the agents should show whilst doing the task. By separating the quantitative and qualitative requirements of agent behaviour we can hope to gain more control over their behaviour whilst retaining the benefits of reinforcement learning. However, whenever a human has any "extra" say in how reinforcement agents learn (as with for example progress estimators) there is a chance that the designer may be setting up an implicit game for the agents to learn. An implicit game is a task set by mistake and is most obviously detected when seeing that the learned agents' behaviours are not what they should be despite seemingly good convergence. One should be aware that extra control in this way just as with group utility function can come at the expense of the asymptotic performance because one is taking away the ability for the algorithm to optimise over the whole problem domain as quantitatively expressed.

Programming by reward has been used before with reinforcement learning to create a reactive agent behaviour specification language called Icarus [Shapiro and Langley, 2002, Shapiro et al., 2001, Shapiro and Langley, 1999, Langley et al., 2004, Choi et al., 2004]. Icarus combines SHARSHA which is a

reinforcement learning method that is a slightly altered version of the previously introduced sarsa method of on-line reinforcement learning with a simple reactive planning language to support programming by reward. The alterations allow for a hybrid use of a simple reactive planning language including *options* (i.e. meta-actions containing a sequence of atomic actions) along with sarsa. Sarsa normally maps state-action pairs to utility values but the altered method SHARSHA (state, hierarchy, action, state, hierarchy, action) maps the state and an action hierarchy to utility values. Most existing reinforcement learning methods act on flat state-action maps whereas SHARSHA includes some hierarchical information in its state (but is not hierarchical reinforcement learning). The action hierarchy partitions the action space so that backups, for example, are performed from [Collect gold - move west] to [Avoid pirates - move west]. The two move west actions are distinct. The planning language is simple and contains *goal structures* with *pre-conditions* and *means* for attaining goals. The means field consists of either *actions* or *meta-actions (or options)* (which may contain many actions and meta-actions). SHARSHA is used to numerically evaluate each of the *options*.

Icarus combines the best of hand coded rules and automated value based behaviour specification by merging explicit behaviour specification with policy learning. It allows human designers to specify behaviour that is within the realms of possibility or common sense for possibly complex tasks (such as computer games). Thus, potentially huge reductions in the size of the combined state-action space can be made and learning can be sped up or simplified [Shapiro et al., 2001]. Of course this comes at the expense of losing much of the exploratory benefits of reinforcement learning. By specifying some behaviour whole sections of the state-action space are cut off from exploration. Thus one has to assume that the correct rules have been written. This is pointed out when Shapiro et al. say, “reliance on background knowledge can be a two-edged sword because it eliminates feasible options.” [Shapiro et al., 2001] Icarus is entirely single agent based and its reward functions are based on the state of the agent in the environment with no reflection of any other learning agent (because only one agent learns at a time in an environment). The example reward functions shown in existing SHARSHA research are all based on a single agent.

Group utility functions can be combined (see Section 3.5) in just the same way that more than one preference can be combined in Icarus. Icarus uses a “piecewise linear function that mediates between the desires” [Shapiro et al., 2001] as per the example given in Equation 2.8. In Equation 2.8, features are scaled and bound to shape the

reward signal (R) according to how the designer envisages the agent's behaviour. In Section 3.5.2 the term *group affiliation* is used to refer to the scalar value that accentuates or diminishes a group's role in an agent's reward signal.

$$\begin{aligned} R &= min(0, 10 \times TimeToImpactAhead - 1000) \\ &+ min(0, 10 \times TimeToImpactBehind - 1000) \\ &+ 10 \times |TargetSpeed - Velocity| \end{aligned} \tag{2.8}$$

Icarus was used to to implement simulated drivers learning using the SHARSHA algorithm. All the drivers trained have the same set of actions at their disposal but have different preferences. For example, one driver wishes to deviate from his target speed as little as possible whereas another wishes to maximise the distance between his car and the car in front. Shapiro and Langley show that each of the six different drivers learn to drive with distinctly different behaviours thus showing that programming by reward is successful in simple one agent experiments. Successful, that is, in achieving the stated aim of learning different behaviours when using the same set of actions.

The progenitors of Icarus cite computer games as a possible use for their technology but, as yet, to the best of the author's knowledge, it has not been used for computer game characters. In fact Shapiro et al. ask for, "empirical studies to establish the system's behaviour in practice." [Shapiro et al., 2001]

2.4 Summary

It can be seen in Section 2.3 that:

- evolutionary methods have been successfully applied to a commercial grade computer game - NERO;
- that evolutionary methods (a search based optimisation technique) has been applied to computer games at the pre-release stage for testing purposes only (to find sweet spots);
- that COIN (COllective INtelligence) is representative of some academic reinforcement learning in that its implementation is ill suited to typical computer games;
- and finally that programming by reward should be an appropriate method to apply reinforcement learning to computer games but that existing work in this area has only been applied to single agents learning in isolation.

From Section 2.1 one can see that reinforcement learning for computer games is at an embryonic stage and game characters are still reliant upon conventional techniques such as finite state machines. There is potential for reinforcement learning to be applied to computer game characters. In particular, reinforcement learning would be most appropriate for large number of characters to learn relatively basic behaviours, freeing up human design time to work on characters requiring more complex behaviour whilst also providing a mass of characters showing somewhat individual behaviour.

The next chapter shows the approach taken to address the issue of how to train characters in a computer game environment to learn to perform a task whilst being constrained by a qualitative behaviour.

Chapter 3

Approach

This chapter shows the theory on which the experiments in Chapters 5 and 6 are based. By taking an approach similar to programming by reward (covered in Section 2.3.4) but with multiple agents one can apply reinforcement learning to computer game characters with a degree of control over the qualitative relationships between the characters. As stated in Chapter 2, the belief is that behaviour in computer games is analogous to the groups of characters in the game. Thereby, replicating the system of relationships between the groups of computer games characters is one appropriate way to set the reward function for reinforcement learning.

In Chapter 5, the tabular form of TD(λ) is applied to several game characters learning at the same time. These agents are affected by each other's presence in the environment but do not explicitly learn with a joint state as in some forms of multi-agent learning. Group utility functions are used to shape the reward signal in a similar way to programming by reward. The curse of dimensionality severely reduces the performance of tabular value functions at a certain level of complexity (shown in Section 5.2) and so a non-linear method of function approximation, artificial neural networks, is used to learn the same tasks (shown in Chapter 6). Using neural networks as a method of function approximation is powerful enough to represent complex unforeseen relationships between features of the state, action selection and expected return. They also generalise experience across the value function thereby covering more of the state-action space with reasonable estimated utility values and speeding learning. This chapter discusses group utility functions and how they modify the reward signal between the environment and the agents. In the next chapter the problem area is explored with agents learning singularly without any group utility functions (i.e. plain vanilla reinforcement learning in a computer game environment). Then in Chapter 5 the first

results of using group utility functions are shown.

3.1 Balance

Throughout this thesis it is assumed that *balance* is a useful quality for computer game characters to learn. In many games there are cases where balance is needed. For example, sports games often have several levels of difficulty so that on some level of difficulty they will be an equal match for many human players. In massively multi-player games often there are several competing computer controlled teams or groupings that should co-exist only to provide a setting for the human players. It would be useful to automate the process of creating a number of different behaviours for agents that individually, or as a group, perform equally as well as other agents or other groups of agents.

The *balancing* group utility function introduced in [Bradley and Hayes, 2005a], [Bradley and Hayes, 2005b] and this thesis, is based on achieving balanced performance. But, what is meant by "balanced performance?" As a definition, a *balanced performance* means that each agent, or group of agents, will achieve the same, or similar, amount of reward over some time period, such as an episode. For instance, if two agents both pick up seven items of reward each then they are perfectly balanced. If they pick up eight and seven rewards respectively then they are somewhat balanced and if they pick up zero and eight rewards they are unbalanced.

Also of interest when considering balanced performance between agents is work on *homo egualis* societies [Nowé et al., 2001]. The homo egualis approach works on the same problem as is addressed by the balanced group utility function and is described as applying to "multi-agent problems where local competition between agents arises, but where there is also the necessity of global cooperation." [Nowé et al., 2001] The homo egualis society method affects the action sets of agents so that if they are doing comparatively well then they are denied the use of certain actions until balance is attained. By affecting the action sets for periods of time, agents of a homo egualis society are said to be using a *periodic policy*. In this thesis agents select actions deterministically and therefore learn a *deterministic policy*. That is, they learn a fixed policy which when followed will lead to a balanced performance against another agent or group. Alternatively, the agents could use a *probabilistic policy* as this allows more detailed expression of the relative merits of actions. For example, fifty percent of the time choose action A and fifty percent of the time choose action B. However, a deter-

ministic policy is used for simplicity's sake as discussed in Sections 2.2.4.4 and 4.2.7.

Methods of achieving an equilibrium between groups of agents can be classed as *naive* and *sophisticated* (by [Tesauro, 2002]). Naive methods usually ignore the fact that there are other agents in the system, whereas sophisticated methods often model the other agents explicitly. Using group utility functions is best described as a naive method. Other agents are included in an agent's state but agents are aware of other agents actions in the world only through the modified (by group utility functions) reinforcement signal and the state space itself. This fits with this thesis's *simplicity* agenda.

3.2 Utility Functions

Section 2.2.4.5 explained that an agent's value function is learned from experience of the environment and the reward signal and should, over time, come to represent a version of the the environment's reward function but with reward spread into a gradient so that goal states are at the uppermost contour lines (as shown in Figure 2.3). Reward functions and value functions measure the utility of performing an action in a state. Hence, reward and value functions are often both called utility functions. A reward function is useful for setting the utility of states as a human, and a value function is useful for allowing agents to exploit its information about utility to perform a learned task. The terms *reward function* and *value function* have precise meanings in reinforcement learning but the term *utility function* can be useful for talking about intermediary steps between the reward and value functions and more general concepts of utility functions.

Although the reward function is nominally placed in the environment (which is possibly logical if one considers reinforcement learning to be biologically inspired) it is perfectly reasonable to think of each agent having a separate reward function (which is also possibly logical if one considers reinforcement learning to be biologically inspired). And, further so, that agents' reward functions can be set arbitrarily. The group utility function introduced here can be thought of as positioned between a reward function which is situated in the environment and the agent. The reward signal originates in the environment and is filtered through any group utility functions before arriving at the agent. The agents use this filtered, or modified, reward for learning their value functions. This is how the existing *team utility function* works.

3.3 The Team Utility Function

The team utility function sums all rewards gathered by all agents at the end of each time-step in an experiment. It is this sum that each agent uses to update their value function. This summation of the agents' reward is supposed to replicate the summation of their efforts just like in a team. For example, if there are three agents in an environment and at some time step, two of the agents were to pick up rewards (of unit-less value *one* for each reward), then each agent would receive a reward signal of two. Thus each agent would then take it that whatever action it had just performed from whatever state had a utility value of *two*. Obviously, the agent who did not pick up a reward may or may not *deserve* that reward. The agent may have facilitated the other agents and so the credit is assigned properly. Alternatively, it may have contributed nothing to the team effort and thus the reward would be assigned to that agent wrongly. This is indicative of the team utility function suffering from a poor signal-to-noise ratio as explained in Section 2.2.4.6. It is possible that some agents will receive positive reward for poor actions. This noise in the reward signal just adds to stochasticity and will most likely be overwhelmed by *correct* feedback. This is the type of problem that COIN (Section 2.3.3) aims to correct. It should be noted that group utility functions will suffer in exactly the same way from a poor signal-to-noise ratio. This is a point worth repeating, that because group utility functions are essentially similar to the team utility function they will too, potentially suffer from poor signal-to-noise issues.

There is a correspondence between the summation function and the relationship learned between the agents. By summing all the agents reward at each step the team utility function is bringing about a team-mate relationship amongst the agents. What other interesting functions could we replace summation with and what relationships would the functions encourage?

3.4 Combining Functions

The team utility function works by summing all the agents' rewards for the previous time step and then giving each agent the total group reward. In this thesis the summation function is called a combining function, as it combines the individual agents' rewards. The reward the agent receives after the being filtered through the combining functions is called the modified reward (notation: r'_i is the modified reward for agent i). The summation function lets us express a team-mate relationship. Here we consider

a few other possibly interesting combining functions and the relationships they should produce:

- Negative standard deviation - Equation 3.1. One can take the standard deviation of each of the agents' individually collected rewards at one time step to get a measure of how well balanced the performance of the agents is at that time step. Taking the negative result means the less deviation, the higher the modified reward will be. The equation used is actually the *population standard deviation*. This is the combining function that is investigated in this thesis to bring about balanced performances.

$$r'_i = -\sqrt{\frac{\sum(r_i - avg(r_1 ... r_n))^2}{n}} \tag{3.1}$$

- Negative range - Equation 3.2. This again measures how well balanced the agents' performances are. One could further investigate by looking at the inter-quartile range. This combining function has not been used in experimentation yet but is here to signify that there are several suitable functions to measure balance.

$$r'_i = -(max(r_1 ... r_n) - min(r_1 ... r_n)) \tag{3.2}$$

- Greater than - Equation 3.3. This simply states that one group should be performing better than another. That is, the better group's agents should achieve a higher reward over time. A magnitude can also be specified to define exactly how much greater than an agent's, or group's, reward should be compared to another's. For example, one agent's reward should be three times higher than another's.

$$r'_i = (r_a - r_b) \tag{3.3}$$

- Less than - Equation 3.4. The opposite of the greater than combining function (Equation 3.3). This function and a greater than function could be combined to keep performance within a range of another agent or group of agents.

$$r'_i = (r_b - r_a) \tag{3.4}$$

- Null - Equation 3.5. The agent is left unaffected by this utility function. This is just nomenclature to notate that agents or groups are uninfluenced by any group utility function. It is used in Chapters 5 and 6 to indicate that for some experiments the agents or groups are unconstrained.

$$r'_i = r_i \tag{3.5}$$

- Sum - Equation 3.6. This team utility function, included for completeness.

$$r'_i = \sum_{i=1}^{n} r_i \tag{3.6}$$

- Average - Equation 3.7. Similar to sum but not often considered. Is it different from the team utility function in any situation and, if so, how could it be used?

$$r'_i = \frac{\sum_{i=1}^{n} r_i}{n} \tag{3.7}$$

3.4.1 Negative Standard Deviation Measures Inequality

As stated, it is an assumption throughout this thesis that balanced gameplay is desirable for some games; to be able to create agents and groups of agents that can learn to compete equally well at tasks even though the agents or groups may differ in their actual capabilities or resources. As a practical example, the desire may be to use group utility functions to train two armies to fight each other so that the battle is very close and either army could win. To achieve this, one could use three group utility functions, two with sum combining functions and one with the balancing combining function. All of the agents (soldiers) of one army would have their reward modified by one of the summative utility functions. The agents in the other army would *belong to* the other summative utility function. The two summative combining functions should be *placed under* the balancing group utility function. This set-up with only four agents for soldiers can be seen in Figure 5.12. These group utility functions modify an agent's reward signal and should induce both team behaviour within each group and balanced group behaviour. How an agent's modified reward is calculated is explained in Section 3.5.2. This balancing group utility function uses the negative standard deviation as its combining function. Thus, using a negative standard deviation function as a combining function creates a balance or equilibrium inducing utility function amongst a group of agents.

To allow an element of balance, an agent's state representation must contain information indicating its performance, or its group's performance, relative to other agents or groups. An agent must know if they are currently performing relatively too well or not well enough. It is not possible to learn an effective equilibrium without this information. If not able to distinguish between winning or losing and, who they are winning or losing against, the agent cannot make sense of the reward signal changes that occur during imbalance. Other equilibrium learners use relative performance indicators in the state representation also [Bowling and Veloso, 2002, Bowling and Veloso, 2003, Nowé et al., 2001]. Of course, more information in the state representation increases the search space thus making learning more difficult. This again, is the curse of dimensionality as explained in Section 2.2.4.7.

3.5 Group Utility Functions

It is easiest to think of group utility functions as an extension of the well known team utility function. It is implicit when using a team utility function that all agents share the same goal. However, this is not the case with computer games that have many groups of agents that, whilst sharing the overall goal of providing the game playing experience, have differing goals at differing levels of abstraction.

Group utility functions should allow reinforcement learning to take account of these differing goals, or *preferences* as programming by reward denotes them. It is possible to take account of many preferences in one complex reward function. However, splitting the reward function into several utility functions makes a good clean abstraction and using the utility functions to denote, not preferences, but relationships makes intuitive sense.

3.5.1 Hierarchies of Group Utility Functions

Group utility functions can be arranged in a hierarchy to give the "society" of agents a structure. This is unlike the team utility function where only one relationship can be specified. The inputs to group utility functions are the groups or agents immediately below the group in the hierarchy. The reward signal originates in the environment and is then redirected through the hierarchy of group utility functions and then recombined before reaching the agents themselves. It is this modified reward (r') that the agents use in their value function update equation. The normal update function shown in Equation

2.2 becomes that shown in Equation 3.8 where $Q(s_t, a_t)$ is the expected reward when taking action a from state s at time t; α is the learning rate; r'_{t+1} is the modified reward at time step $t+1$; γ is the discount rate and $Q(s_{t+1}, a_{t+1})$ is the expected reward at when taking action a_{t+1} from state s_{t+1}.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r'_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{3.8}$$

3.5.2 Calculating Agents' Modified Rewards

To calculate an agent's modified reward, sum:

- its privately collected reward and;
- its parent group utility function result multiplied by the group affiliation (a scalar value) of the agent to the group and;
- its grandparent's group utility function result multiplied by the group affiliation of the parent group to the grandparent group and;
- so on until the top of the hierarchy is reached.

More formally, ${r'}^t_a$, agent a's modified reward at time t is given in Equation 3.9.

$${r'}^t_a = {r_a}^t + f(g_1)(a \leftrightarrow g_1) + \sum_{i=2}^{h} f(g_i)(g_{i-1} \leftrightarrow g_i) \tag{3.9}$$

where: r^t_a represents an agent's privately collected reward at time t, $f(g_i)$ is group i's utility function, $x \leftrightarrow y$ is the group affiliation between x and y and h is the height of the group hierarchy above an agent. As input, a group utility function takes the outputs of its immediate (one level deep) sub-groups and its immediate agents' (one level deep) privately collected rewards.

Figure 3.1 shows the agents' privately collected rewards, their affiliations to the groups and those groups' affiliations to other groups. It also shows the modified rewards returned to the agents for learning. Agents one and two have an affiliation of *one* to the sum utility function. That sum function has an affiliation of value *two* to the negative standard deviation utility function. Agent three has a group affiliation value of *one* for the negative standard deviation group utility function.

Scaling utility function output with group affiliations is a necessity for experiments with uneven groupings such as those in Sections 5.2.2 and 6.3.3. When under a balancing group utility function for instance, if an agent picks up a reward, the combined

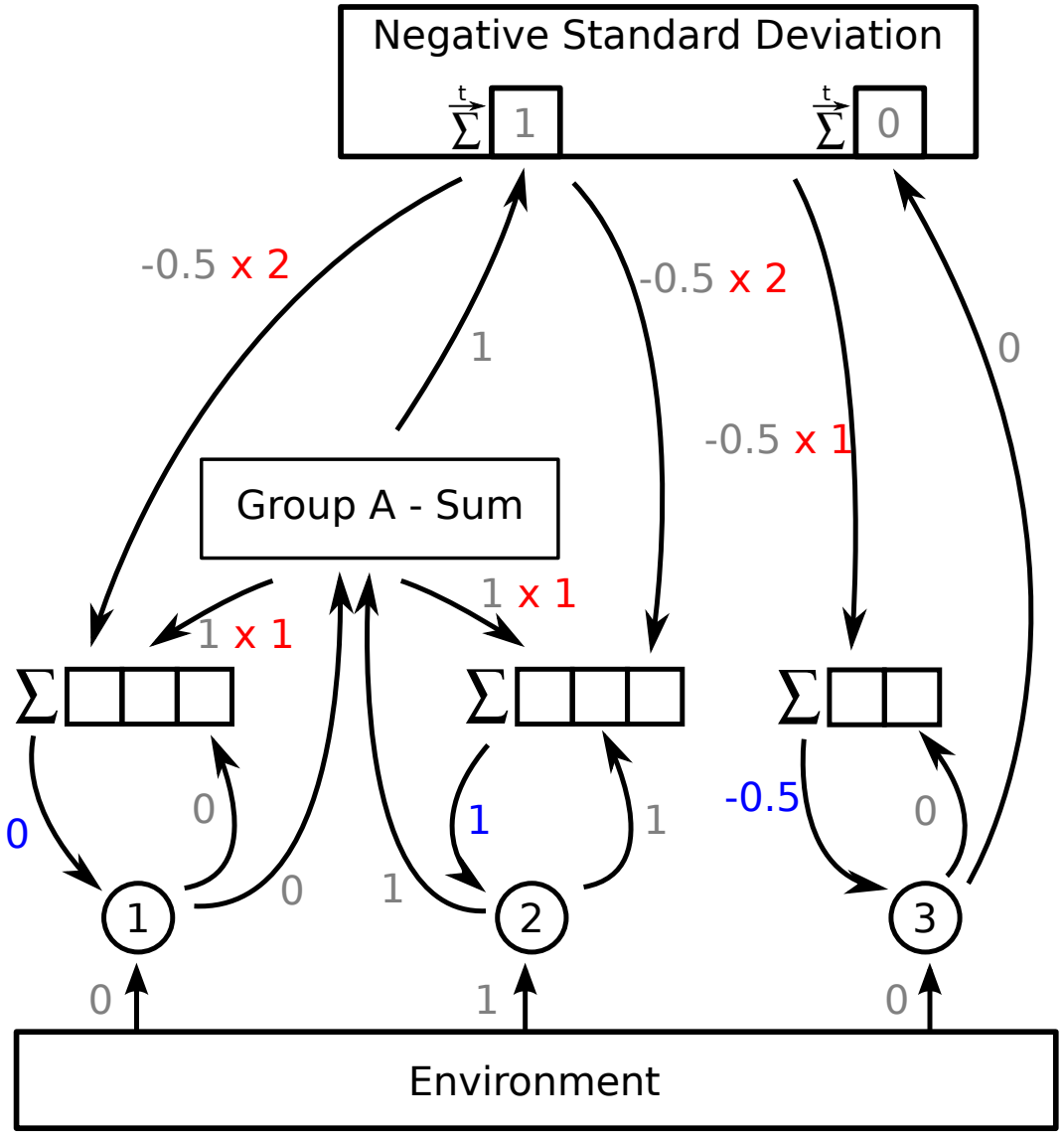


Figure 3.1: Showing private rewards from the environment to each agent (1, 2 and 3); reward as it is passed up the group utility function hierarchy; the group utility function output values combined with group affiliation values (in red preceded by '×'); and the agents' modified rewards in blue. The summation functions is shown with an entry for the agent's privately collected reward and an entry for every group above the agent. The small boxes on the negative standard deviation group utility function indicate that the running total of input values is being kept.

reward signal needs to reflect that it was both good to pick up a reward but, bad to unbalance the agents' or groups' performance. A larger group affiliation value gives a group utility function a larger presence in the agents' modified reward signal. If there is a group with a large number of agents under a sum group utility function then, if this group is placed under a negative standard deviation utility function, the sum signal sent back to the agent for learning could swamp the negative standard deviation signal. Thus one would need to set a large group affiliation between the sum group and the negative standard deviation group. Setting the group affiliations is quite intuitive but sometimes requires experimentation. Icarus uses a similar method as shown in Section 2.3.4 where individual components of the reward function are scaled individually before being added together.

The results in Sections 5.2.1 and 6.3.2 show experiments with two sum group utility functions with an even number of agents in each group, and each of these groups placed under a balancing group utility function. In these experiments the effect of the group affiliation is not interesting as groups of agents are naturally balanced. Recall that an agent's modified reward is the sum of its individually received reward and its group utility functions' values each scaled by group affiliation values. For all the experiments in Chapters 5 and 6, agents receive a reward of zero for each action not leading to a reward or, *one* when a reward is picked up. Sometimes a group utility's output might need to be scaled to interact appropriately with the agents typical rewards. If agents *one* and *two* in Figure 3.1 have both failed to pick up any reward in their last action steps they will get a modified reward of 0 or, $0+(-\sqrt{\frac{\sum(r_i-avg(r_1...r_n))^2}{n}})$. As mentioned, if one of the agents picks up a reward we need to reflect that it was good to pick up a reward but bad to unbalance the agents' performance. This is done using the group affiliation values to make sure that all the agent receive appropriate rewards and punishments. The negative standard deviation of the number of rewards picked up by each agent is calculated after each action step. After one reward is picked up (by agent *two* as shown in Figure 3.1), the negative standard deviation of the running total for the group and agent *three* will be -0.5. In Figure 3.1, when agent *two* picks up one reward it will receive a modified reward of 1 (the agent's individual reward of one, the sum group utility's output of *one* plus the negative standard deviation of -0.5 scaled by the group affiliation of 2). After picking up another reward, assuming the other agents have not yet picked up any rewards themselves, the agent will receive a modified reward of 0. That is the individually collected value of 1, plus the sum group utility function output of 1, scaled the the group affiliation value of 1, and the negative standard deviation of

0 and 2 scaled by the group affiliation value of 2 which is -2. This neither encourages the agent to pick up a reward leading to such imbalance again, nor does it discourage the picking up of the reward. After picking up a third reward the modified reward would be -1. Thus by setting the group affiliations as they have been set in Figure 3.1 the agents are encouraged to pick up reward if it will lead to an imbalance of one reward and discouraged from picking up a reward if it would lead to an imbalance of three or more rewards. The group affiliation could be set anywhere between 0.1 or 1.9 and this would give the balancing group utility function less or more presence in the agents' modified reward. The higher the group affiliation value the more discouraged an agent will be from picking up another reward when unbalanced. If in this situation the group affiliation value was set to *two* or higher then the agent would never learn to pick up rewards at all as the modified reward after picking up one reward (again assuming the other agent had not picked up any rewards) would be negative thus discouraging the agents from picking up any reward at all.

The balancing group utility function needs to keep a running total of its inputs for calculating the negative standard deviation of its agents or groups. The negative standard deviation needs to be taken of the number of rewards picked up by the agents or groups since the start of each episode. This way the agents can be allowed to pick up some reward and then only when the imbalance reaches a certain level will an agent be negatively rewarded for picking up another reward. The point at which the reward signal switches from positive, through zero to negative is set via the balancing function's group affiliation with its groups or agents as in the example just given.

To test the concept of group utility functions many qualitative experiments have been run for several arrangements of agents and groups and for several variations in parameters. These are shown in Chapters 5 and 6. For the experiments in Chapters 5 and 6, simple hierarchies of agents and groups are used. These are shown for balanced agents/ groups of agents in Figures 5.1, 5.2, 5.11, 5.12, 5.15 and 5.16 and for unbalanced groups of agents in Figures 5.19, 5.20, 5.23 and 5.24. The experiments in Chapter 5 use tabular value functions. Then the value function is approximated by neural networks in Chapter 6. For each hierarchy of agents and type of value function (tabular in Chapter 5 or approximated in Chapter 6) several experimental variations were run:

- Two agents acting individually;

- Two agents constrained by the balancing group utility function;
- Two groups of two agents with each set of agents acting under a sum group utility function with each group acting individually. i.e. under the *null* group utility function;
- Two groups of two agents. Each set of agents is under the sum group utility function and the groups are constrained by the balancing group utility function;
- Two groups of three agents with each set of agents acting under a sum group utility function with each group acting individually. i.e. under the *null* group utility function;
- Two groups of three agents. Each set of agents is under the sum group utility function and the groups are constrained by the balancing group utility function;
- One group of two agents and one single agent. The group of agents are under the sum group utility function and the group and single agent are unconstrained. i.e. under the *null* group utility function;
- One group of two agents and one single agent. The group of agents are under the sum group utility function and the group and single agent are constrained by the balancing group utility function;
- One group of three agents and one single agent. The group of agents are under the sum group utility function and the group and single agent are unconstrained. i.e. under the *null* group utility function;
- One group of three agents and one single agent. The group of agents are under the sum group utility function and the group and single agent are constrained by the balancing group utility function;

Several experiments for each hierarchy of agents, groups and group utility functions are run with varying parameters. These are:

- the agents' action selection policies:
  - ε-greedy action selection policy with a constant ε set at 0, 0.2, 0.4, 0.6, 0.8 and 1;

  - ε-greedy action selection policy with a degrading ε set initially at 0, 0.2, 0.4, 0.6, 0.8 and 1 and degraded linearly from its initial value to zero by the end of the experiment.

- the amount known about other agents in the environment:
  - knowing whether the agent is winning, drawing or losing and whether the agent is on the same team or not[1] to none or one of the other agents nearest to any reward and that agent's distance to that reward;
  - knowing whether the agent is winning, drawing or losing to none or one of the other agents nearest to a reward in the agent's state and the distance from that agent to that reward;

- the number of rewards in the environment known about:
  - knowing the distance to the nearest reward in the environment and being able to choose between moving towards or away from that reward;
  - knowing the distance to the nearest two rewards in the environment and being able to choose between moving towards or away from either reward;

For each hierarchy of agents, several experiments were run for each combination of the settings listed above. A graph for all the different values of ε for each experimental set-up is shown in the appendices to this thesis. To illustrate the interesting results from this large number of experiments, selected plots are taken from these graphs and are shown in Chapters 5 and 6. A discussion of each hierarchy of agents with each experimental set-up using tabular value functions is given in Chapter 5 and, similarly in Chapter 6 but with the value functions approximated by neural networks. The next chapter shows several initial experiments with varying parameters in the computer game environment. The single agent in these initial experiments perform the same foraging task used throughout this thesis.

[1] On the same team means that the two agents are directly under a sum group utility function.

Chapter 4

Exploration of Learning in the Problem Domain

This chapter describes the experimental set-up used to produce the results seen in Chapters 5 and 6. First a brief description of the task is given in Section 4.1. The computer game environment is described in Section 4.2.1. A practical solution for how to use group utility functions with asynchronous action steps is given in Section 4.2.2. This is followed by a discussion of whether the task presented is best modelled as a continuous or episodic task and how considering the signal-to-noise ratio can help in this decision. Next the learning algorithm is presented and explained. This is followed by a description of the state representation used, the actions available to the agents and the action selection policies used. Finally the chapter contains descriptions of several exploratory experiments, the graphs produced and an explanation of what they mean. These experiments were run to better understand how agents learn the task before introducing the group utility functions in Chapter 5.

4.1 The Task

In this chapter a single agent is given a simple task to learn in a computer game engine. The task is to pick up as many items as possible that are placed randomly about the game environment. The environment is free from obstacles, containing only the game character itself and the items to be picked up. This task is the base task used in all the experiments in this thesis. In Chapters 5 and 6 this task is the quantitatively measured part of the agents' behaviour, with the qualitative part brought about by group utility functions. Chapter 5 contains greater detail about the set-up of experiments. It is

expected that a single agent will, over the course of an experiment, learn to pick up some number of rewards. This number of rewards will be small at first and grow until asymptotic performance is reached. That is, until the agent has learned to pick up rewards in the most efficient way and could not pick up rewards any quicker.

4.2 Learning

As stated, it is assumed that balanced gameplay is desirable for some computer games. The experiments shown in Chapters 5 and 6 aim to prove that, using group utility functions, reinforcement learning agents can optimise their behaviour for a reward function that encourages some quantitative behaviour, in this case foraging for as many items as possible, and encouraging a qualitative behaviour, a well balanced amount of reward collected by each agent or groups of agents. Of course all agent behaviour must ultimately be measurable quantitatively using reward functions but there is a distinction between these behaviours. Foraging is the primary task, based on the quantity of items gathered whereas, trying not to unduly unbalance the performance of the agents and teams is a secondary task and is a quality of the primary task.

The experiments in this chapter show how single agents perform in the environment whilst learning to forage. The experiments give a base line for comparison with those experiments that show group utility functions and hierarchical group utility functions working in Chapters 5 and 6. They also form part of the learning curve for any real world application of reinforcement learning. Reinforcement learning used in a system of any reasonable size, even those that are quite simple, is not a transparent method. That is, once a solution has been found it is difficult to analyse the value function or agent behaviour to see how or why that behaviour came about. So it is important to start any investigation with simple experiments such as these presented in this chapter. This is especially true before going on to alter the reward function as in Chapters 5 and 6.

4.2.1 The Environment

Torque [Garage Games, 2001] was used for all experiments in this thesis (shown in Figure 1.1). Torque is a licensable, modern, 3D game engine. It is important to show implementations of reinforcement learning in computer game engines to serve as working examples for computer game developers instead of “grid world” style simulations

(grid-worlds are discussed in Section 2.2.4.3). This is the intent behind the decision to use a a computer game engine for the experiments. This gives some proof that the algorithm presented is likely to work in a "real" computer game. This is unlike COIN (Section 2.3.3) where it was discovered that the algorithm was actually very difficult to implement for typical computer games.

Using a computer game engine for any search based optimisation techniques brings with it some practical running time issues. This is because the experiments must be run for long enough to reach asymptotic performance, potentially taking many thousands of time steps. For the experiments, the Torque game engine was run as a dedicated server on linux machines. When run as a server, the Torque engine, does not display graphics and can be sped up so that experiments can be run more quickly. Each graphed curve in this thesis is the result of averaging four runs of the same experiment which should account for some experimental variation.

In all of the experiments in this thesis the agents and rewards are placed randomly within the environment at the start of the experiment (and replaced randomly at the start of each episode). Twenty rewards are placed randomly in the game environment at the start of each episode. When a reward is picked up by an agent, another reward is placed twenty metres away from any other reward and ten metres away from any character. An agent receives a reward of zero for each action that does not end in it picking up a reward and a reward of *one* if a reward is picked up. This is before modification by the group utility functions. Unlike the experiments shown here, in previous work [Bradley and Hayes, 2005a, Bradley and Hayes, 2005b] the agents received a reward of negative one for each reward picked up. Giving a negative reward for each time-step that did not lead to a terminal goal state (i.e. a state where a reward is situated) means that the agent should be encouraged to find a direct path to goal states. However, after initial investigative work [Bradley and Hayes, 2005a, Bradley and Hayes, 2005b] it became apparent that this encouragement was not needed in these experiments[1] and that a reward of zero at non-goal states simplifies comprehension. With the group utility function shown in the next chapter the agents are rewarded with the negative standard deviation of more than one agent. If the agents act asynchronously, as they are allowed to do in this thesis (as discussed in Section 4.2.2), then they will be punished purely for acting asynchronously. The agents still learned the task in the previously published work but did so under a noisy reward signal as the punishment for acting

[1] With a long enough amount of time to learn and an appropriately set discount factor, γ, the agent will learn the quickest path anyway.

asynchronously is all noise. In addition, not using punishment at every time-step not leading to a goal state should not lead to longer state-action space trajectories to reach positive reward. Given enough exploration time, reinforcement learning will learn the quickest route to positive reward without punishment. The agents have one hundred action steps each which are shared between them to complete the task. An episode ends when the allotted number of action steps is reached. The agents are given no order in which to act and this is left to the computer game engine. The game environment consists of one hundred and sixty square metres of undulating ground with rewards placed on the ground. The undulations make little difference to the agents actions although they add to the variance in the time taken per action, which can affect the order in which the agents take their turns. The agents are portrayed as ogre like creatures brandishing crossbows which is one of the example characters provided by the makers of Torque (although this detail is superfluous with respect to the experiments). The agents move in three metre steps and they can pick up a reward when it is two metres or less away from them. The Torque engine does not guarantee any level of accuracy when characters move. This leads to noise as agents are often positioned incorrectly after moving. The agents' positions are rounded to three metre steps before being used to build the state vector. This rounding is essential for the tabular value functions used in Chapter 5 for two reasons: to stop state-space explosion when using table based value functions as otherwise, a distance of three metres shows up in the state space as 3.01, 3.02, 2.99, 2.98, etc. and; rounding to three metres reduces the size of the state space without compromising the agents' ability to learn as the agents move in (approximately) three metre steps.

4.2.2 Using Group Utility Functions with Asynchronous Time-steps

Team utility assigns the same reward to each agent at each time step. It does this by summing each agents' privately collected reward for the current time step and passing the total to each agent as its modified reward, with which the agent learns. There is a small problem if the agents' actions are asynchronous as it is possible for other agents to complete more than one, or even less than one (zero), time steps during only one time step of any other agent. For the purposes of group utility, an agent's last reported privately collected reward is taken as its last time step reward. This is shown in Figure 4.1. This is not a large problem but one can see that it could add noise to the reward

signal.

Figure 4.1: Shows the relationship between privately collected reward (what the agent actually collects in the environment) and modified reward (what the agent receives from its group utility functions) for the sum combining function (essentially the team utility. Remember that the team utility is the sum of all the agents' rewards). Here all agents start with an assumed reward value of zero as in all the experiments. Each row represents an action step taken by one of the agents. As one can see, when an agent takes more than one action between another agent's actions the collective reward is calculated on the last reported reward of the other agents in the environment. This will introduce some noise in to the reward signal is one to to handle a reward signal which is a function of privately collected rewards with asynchronous action steps.

4.2.3 Continuous or Episodic Task

The foraging task used in all the experiments in this thesis does not end when an agent picks up a reward. Another reward is placed randomly and all the agents continue to forage. This raises the question, is this a continuous or episodic task (as described in Section 2.2.4)? It is informative when making this decision to look at the task in terms of reward signal noise. After picking up a reward, an agent then plays no part in determining its next state. Rather, its next state is randomly chosen by the algorithm that places a new reward. It is possible for the agent to learn the probability distribution

over the next states but, if the placement is truly random then there is little of interest to learn and so this state transition is treated as the end of an episode for all the agents in the experiment. Hence, the task is treated as episodic. Some information has been removed from the agents by making this decision. This trade off between noise and information is a familiar compromise in reinforcement learning. For example, when learning in the presence of other agents, each agent experiences noise from the other agents' actions, but this noise is tolerated to allow the agents to learn with a smaller state representation (i.e. without all the other agents being represented). One can imagine a sliding scale between episodic and continuous tasks from intolerable noise which should cause the end of an episode, to acceptable noise which still informs the agent in some small way and does not make learning impossible.

4.2.4 The Learning Algorithm

The learning algorithm shown below is standard sarsa(λ), as taken from [Sutton and Barto, 1998].

- $\forall$ agents
  - Place agent randomly
  - $\forall\, s \in S, a \in A, \quad Q(s,a) = 0$
  - Set $\alpha = 0.1$, $\gamma = 0.9$, $\lambda = 0.9$ and initial $\varepsilon = [0, 0.2, 0.4, 0.6, 0.8, 1]$
- Place rewards randomly at least twenty metres distance from other rewards and ten metres away from any agent.

- $\forall$ time-steps $t_0^{\text{number of episodes} \times (100 \times \text{number of agents})}$ (where there are 100 action steps for each agent in each experiment)
  - For each agent (Chosen in turn by the computer game engine. Essentially, whichever agent is ready. This is normally in the same repeating order e.g. Agent 1, Agent 2, Agent 3, Agent 1, Agent 2, ...)
    - ${r_i}^t \leftarrow R_i$ (Remove any rewards if picked up and replace with another reward)
    - ${r'}_i^t = {r_a}^t + f(g_1)(a \leftrightarrow g_1) + \sum_{i=2}^{h} f(g_i)(g_{i-1} \leftrightarrow g_i)$ (from Equation 3.9)

*

$$\delta = \begin{cases} {r'}_i^t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) & \text{if } s_t \text{ is not a reward state} \\ {r'}_i^t - Q(s_t, a_t) & \text{if } s_t \text{ is a reward state} \end{cases}$$

* $e(s_t, a_t) = 1$ (Replacing traces. Alternatively, accumulating traces $e(s_t, a_t) = e(s_t, a_t) + 1$ could be used. [Sutton and Barto, 1998].)
* $\forall\, e(s, a)$
  - $Q(s_t, a_t) = Q(s_t, a_t) + \alpha\delta e(s_t, a_t)$
  - $e(s_t, a_t) = \gamma\lambda e(s_t, a_t)$
* ${a_i}^{t+1} \leftarrow {\pi_i}^t$ using ε-greedy selection
* if degrading ε then $\varepsilon =$ initial $\varepsilon \times \left[\frac{\text{number of episodes} - t}{\text{number of episodes}}\right]$

Each agent is placed randomly, along with twenty rewards that are placed randomly but twenty metres from other rewards and ten metres from any agent. Each agent's value function table is initialised to zero for every possible entry. Each agent has their α value set to 0.1, γ set to 0.9, λ set at 0.9 and its initial ε value set to one of either 0, 0.2, 0.4, 0.6, 0.8 or 1 depending on the experiment. The experiment is then set-up and the agents are ready to start learning. The agents and rewards are reset to random starting positions at the start of every episode. Each agent takes turns to perform an action until $100 \times$ number of agents in the environment actions have been taken. If the agent is within two metres of a reward then that reward is removed and another placed, again twenty metres from other rewards and ten metres from any agent. For every reward collected the agent receives a reward of value, one for that time step. The agent's modified reward is then calculated as in Equation 3.9. δ, the error between the expected return in reward and the utility value currently in the value function is then calculated. δ is set to the difference between the expected reward return including the newly sampled reward function (${r'}_i^t + \gamma Q(s_{t+1}, a_{t+1})$) and the existing stored estimate ($Q(s_t, a_t)$) if the state does not contain a reward. If state, s_t, is a reward state and the problem is classed as episodic as this problem is (see Section 4.2.3) then $Q(s_{t+1}, a_{t+1})$ is defined as zero and so δ is set to the difference between the actual sampled reward ${r'}_i^t$ and the existing value held in the value function, $Q(s_t, a_t)$. The eligibility (e) of the current state-action pair is then set to one. Eligibility traces speed learning by keeping a short history[2] of visited state-action pairs and propagating the error (δ) back

[2]This thesis keeps an eligibility trace of length, twenty. When the eligibility trace is full the earliest entry is swapped for the current state-action pair. Eligibility traces can greatly improve the efficiency of reinforcement learning. [Sutton and Barto, 1998]

to each of these eligible entries in the value function. Using replacing traces each current state-action pair has its eligibility set to one. Then for every state-action entry in the eligibility trace the value function is updated by the difference, δ, scaled by the eligibility, e, and the learning rate, α. The learning rate stops the estimated utility value swinging back and forth across actual sampled reward and instead slowly moves to a value representing an average of sampled reward values for each state-action pair. The eligibility of each state-action pair is then scaled down as the samples move back in time and relevance. Each eligibility is scaled by the discount rate, γ, and λ which scales the strength of the eligibility trace back in time. A λ value of 0.9 is used in this thesis. If $\lambda = 0$ then the eligibility trace has no effect on learning and if $\lambda = 1$ then all states visited up until a reward is picked up are considered equally important in achieving that reward and the value function is set as so. Setting $\lambda = 1$ should greatly increase the speed of learning. The agent then chooses its next action. Finally the agent updates its ε value if the experiment is using degrading ε values; otherwise ε is left at its initial value.

For all of the experiments using tabular value functions the agents learned using sarsa(λ) [Sutton and Barto, 1998] as shown above with the following learning parameters: $\alpha = 0.1$, $\gamma = 0.9$, $\lambda = 0.9$ and group affiliations of 1 between each agent and their group (group affiliations only affect those experiments with more than one agent unlike the experiments in this chapter). The learning was sped up by using eligibility traces of ten steps long. ε-greedy action selection was used with initial ε values of 0 (fully exploitative), 0.2, 0.4, 0.6, 0.8 or 1 (i.e fully random). In half the experiments ε was degraded linearly from the initial value at the first episode to zero at the end of the experiment (as shown in the learning algorithm above. In the other half of the experiments ε was kept constant at its initial value.

4.2.5 State Representation

The state representation for each agent contains the distance to the nearest n rewards rounded to the nearest three metres. This is inspired by an approach from robotic football as described earlier in Section 2.2.4.2 and is a good way of representing potentially complex and especially dynamic computer game worlds for agents in a concise way. That is, if the rewards are somewhat homogeneous then they do not need to be recognised as *reward A* and *reward B* but can be recognised as *nearest reward* and *second nearest reward*. This way what was learned about *reward A* when it is the *nearest*

reward can be applied to *reward B* when it is the *nearest reward* too. This helps the agents' learning scale well when there are twenty rewards in the state and they can all be treated as similar with respect to how close they are to an agent. Rounding the distance to the nearest three metres cuts down the size of the state space whilst maintaining enough information to distinguish between near rewards. The state also contains the distances of the n nearest agents to any reward or to the rewards in the agent's state representation. Variables stating whether the agent is winning, losing or drawing with the other agent (in terms of reward collected) and if the other agent is on the same team or not are included. An agent is on the same team if they are under a *sum* group utility function. By organising the rewards by distance and taking only the nearest n the agents are learning a simpler task than they are actually achieving. The agents are able to forage across all the rewards even when they are dynamically placed but are in fact only ever aware of the nearest rewards. That is under the assumption that when considering which reward to pick up an agent only needs to consider the n nearest rewards. If this were not the case and an agent could not perform optimally with only n choices of reward then the designer could add more rewards into the state representation. The method works by cutting out what is deemed to be the least useful information, parametrised in this case by distance, to keep the state-action space relatively small. One could imagine though that any parametrisation could be used based on such variables as danger or familiarity of objects in the environment. In the graphs shown later in this chapter one can see that just two nearest rewards in the state space has an adverse affect on the performance of a single agent and three rewards results in far worse performance. Consequently, in the experiments in Chapters 5 and 6 one or two rewards are included in the state space.

The size of the state space is calculated by multiplying the number of possible values for each variable in the state representation together. For example, let us say that the state representation contains: a distance value rounded to the nearest three metres with a maximum of two-hundred metres and a minimum of zero metres; a value representing the comparative performance of an agent or group to another agent or group, i.e. winning, losing or drawing; a binary value representing the relationship of an agent or group to another agent or group, i.e. same group or other group. There are sixty-seven possible values that the distance variable can take, three that the comparative performance indicator can take and two that the group relationship variable can take. So the size of the state space in this case would be $67 \times 3 \times 2 = 402$. That is four-hundred and two possible states that the agent could be in. The actual size of the

state-action space for each experiment is shown in Section 5.1.1.

4.2.6 Actions

Actions available to the agents are: move towards any reward and, move away from any reward in its state representation. Thus the real task is to learn which reward to move towards and when not to move towards a reward. The move away actions are only available when the respective reward is less than two hundred metres away from the agent to contain the agents within a reasonably sized area.

Computer games allow us to use these high level actions and make the agents' actions appear much more realistic. These actions are called deictic actions as they are defined with reference to objects in the environment. Originally it was thought for this work that the traditional north, south, east and west actions would be used in the experiments but aesthetically this was very poor as the game characters' movements looked aesthetically poor and unrealistic. By using deictic actions we can allow game designers to still control the behaviour of the characters to a certain degree. Designers could also leverage control by setting permissible actions from certain states although this is not done here. As discussed in Section 2.3.3, it is these deictic actions which made it too difficult to use COIN for computer game characters as originally intended.

The size of the action space is calculated in the same way as the state space and multiplied together they give the combined state-action space which determines the size of the search space and hence, partly determines the quality and speed of learning.

4.2.7 Action Selection

The agents use ε-greedy action selection throughout the experiments here. ε-greedy selection simply chooses a random action with probability ε and chooses the action leading to highest return according to the value function with probability $1-\varepsilon$. ε is set to a value between one and zero. Certain theoretical guarantees state that in order for reinforcement learning to converge to an optimal policy, one of the conditions that needs to be met is that ε should be slowly decreased to zero during the experiment [Sutton and Barto, 1998]. In previous published work using group utility functions [Bradley and Hayes, 2005a, Bradley and Hayes, 2005b] the agents used a softmax action selection policy using the Boltzmann equation (Equation 4.1) to set the probability of each action being selected. Softmax action selection chooses actions probabilistically. The higher a state-action pair's utility value the more likely the action is to be

chosen. In the existing work [Bradley and Hayes, 2005a, Bradley and Hayes, 2005b] the Boltzmann distribution was used to weight each action's likelihood of being chosen. Action a was chosen at step t with probability given by Equation 4.1.

$$\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}} \tag{4.1}$$

Softmax action selection relies upon τ, the temperature, and setting it appropriately requires knowledge of the final utility values in the value function, which for some of the experiments in this thesis was difficult to foresee. Setting τ and its rate of decay is another complexity that is not needed for the concept of group utility functions to be demonstrated although the subtlety of probabilistic action selection would suit the cheap equilibrium point sought. In some ways it is simpler to learn a probabilistic policy for equilibria as the policy could learn to balance the utility of actions more easily. For example, when in already equilibrium choose to pick up more reward 50% of the time and choose not to pick up reward 50% of the time.

4.3 Exploration and Understanding of the Problem Domain

Results follow from a series of experiments that are designed to explore and show the features of the problem task, before any group utility functions are investigated. For example, the effect of state-action space size is shown along with the effect of varying the amount of time available to learn. First a base line performance measure is shown in Figure 4.2. This graph shows the performance of a single agent learning to pick up items of reward. One line shows the effect of using an initial ε value of one, degraded linearly to zero at the end of the experiment. The other line uses a constant value of $\varepsilon = 1$. The agent starts out performing random actions but by the end of the first few episodes it is already doing almost as well as it ever will. The point where the agent can do no better is called *asymptotic performance*. As the reader can see, the choice of constant or degrading ε has no visible effect on this simple task. The graph points are calculated by testing the agents at regular intervals throughout the experiments. The agents are tested for an entire extra episode whilst being greedy in action selection for each point on the graphs. The agent takes one hundred action steps each episode. Each experiment in this thesis was run four times and all graphs shown are the result of averaging these four runs. The line in Figure 4.2 is *bumpy* because agents and rewards

are placed randomly in the environment and thus a certain amount of natural variation in performance occurs throughout the experiments. Every plotted line in this thesis is an average (the mean value) taken from four runs of the same experiment. Figures 4.3 and 4.4 show the *standard error* at each sample episode for the experiments shown in Figure 4.2. The error bars show generally smaller errors across the four runs of the same experiment than across episodes shown on the mean plot. The natural variation in number of rewards picked up occurring because of random agent and reward placement is usually as large as or larger than the standard error measured over the four runs. The repeated experiments have similar performances to each other and this indicates that the experimental results are repeatable. The error bars for plots on other graphs are left off to aid clarity. Each line, other than the lines in Figure 4.2, is drawn as a Bézier curve over each point to simplify the presentation of the learning trend. This is useful when there are a number of lines on the graph and the natural variance in performance is hiding the significant differences. Figure 4.5 shows the same experiments as Figure 4.2 but is drawn with a smoothed line.

The combined state-action space of the agent in the experiment shown in Figure 4.5 has one hundred and thirty-four possible entries. In a typical run, after one thousand episodes of one hundred action steps, the agent visited seventy-two of those state-action pairs. Upon investigation, the unvisited states are those far away from the reward that due to random positioning of the agents and rewards, the agents never experience. The state representation contains the distance from the agent to the nearest reward rounded to the nearest three metres (as the agent moves in steps of three metres). The agent's actions are represented by two binary action bits - move towards (1, -1) and move away (-1, 1). This is a simple state and action representation which reflects the complexity of the task being learned. The state-action space grows as the complexity of the task increases in Chapters 5 and 6.

4.3.1 Exploration Versus Exploitation

Figures 4.6(a) and 4.6(b) show that the agent's learning performance is unaffected by the agent's action selection policy. The graphs show that, for constant values of ε (Figure 4.6(a)) and when ε is degraded linearly to zero at the end of the experiment (Figure 4.6(b)) for ε values of 0, 0.2, 0.4, 0.6, 0.8 and 1, performance remains almost identical. For such a simple task it is not that surprising that the agent is able to learn well using any of these policies. Even with no random exploration the initial zeroed

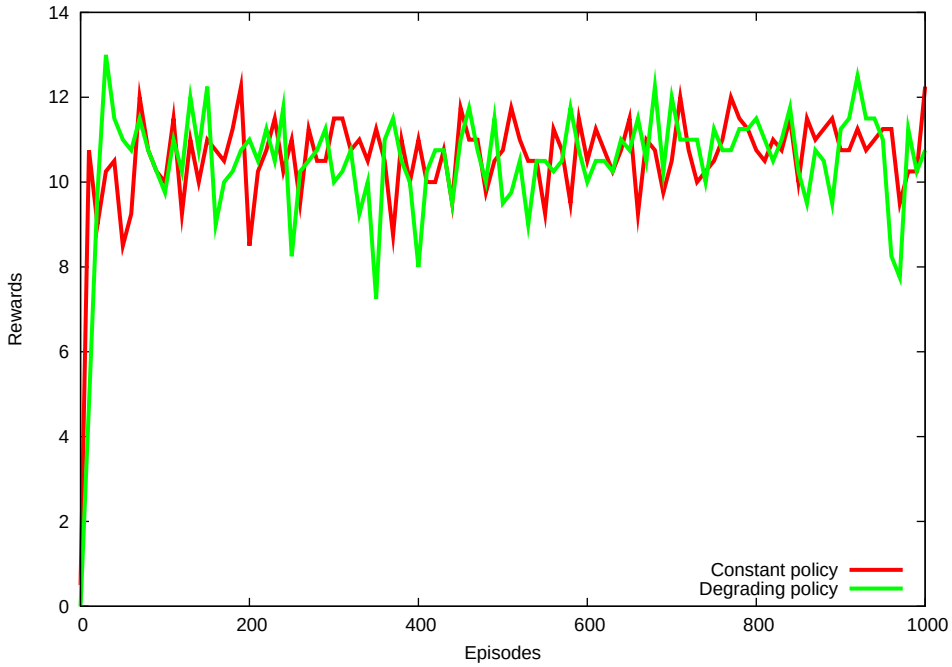


Figure 4.2: Both degrading and constant ε-greedy policies for $\varepsilon = 1$ for tabular value functions without smoothing. Both of the experiments featured one single agent.

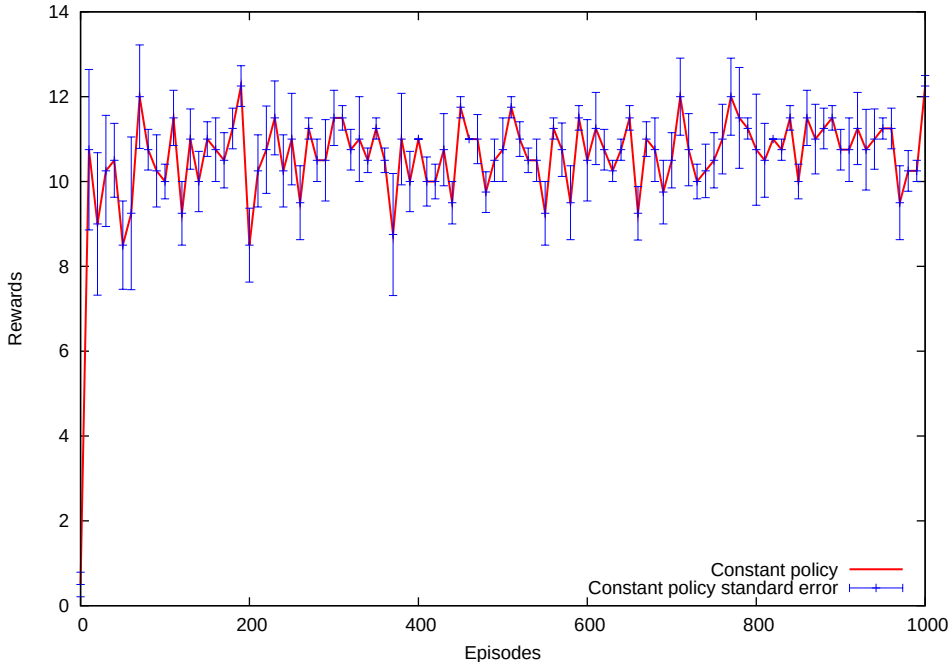


Figure 4.3: Constant ε-greedy policy with $\varepsilon = 1$ for tabular value functions without smoothing. Standard errors for each sample point are shown.

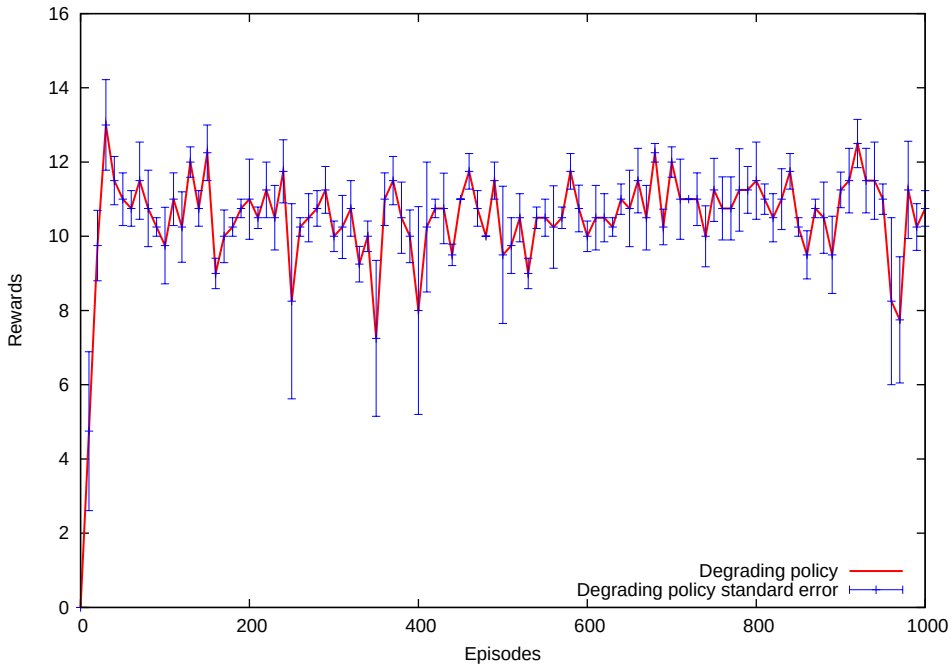


Figure 4.4: Degrading ε-greedy policy with $\varepsilon = 1$ for tabular value functions without smoothing. Standard errors for each sample point are shown.

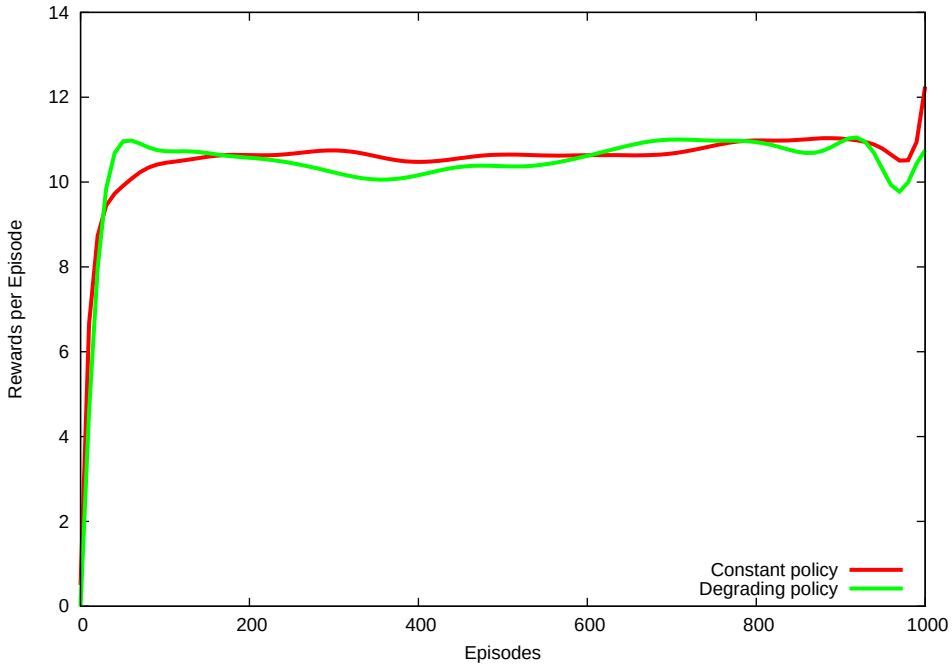
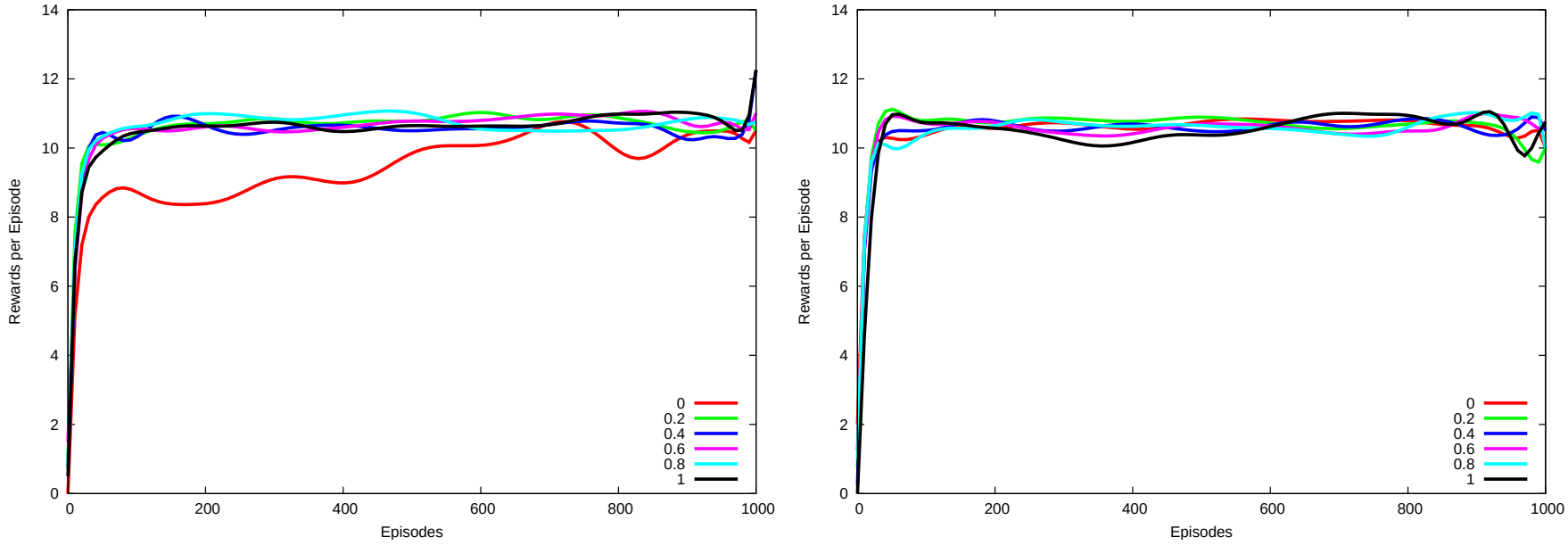


Figure 4.5: Both degrading and constant ε-greedy policies for $\varepsilon = 1$ for tabular value functions. Both of the experiments featured one single agent. The points are plotted as a Bézier curve.

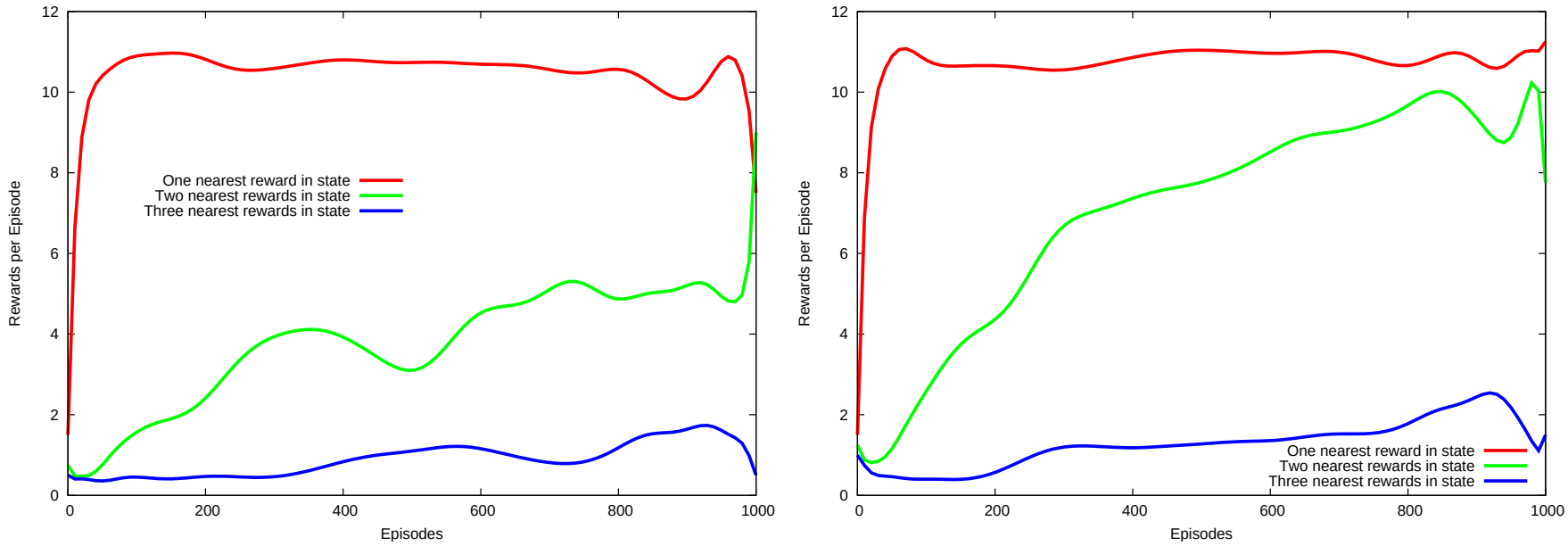
starting values are enough to encourage sufficient exploration of the small state-action space. This is not always the case as for more difficult tasks the policy must allow for a large amount of exploration to learn satisfactorily. When approximating the value function, as in Chapter 6, the use of non-stationary policies could be troublesome. With a non-stationary policy an approximation of a value function is not guaranteed to converge and could often fail to converge in theory and in practice.



(a) Constant ε

(b) Degrading, initial ε degrading linearly to zero at the end of the experiment.

Figure 4.6: Shows the performance of a single agent for various values of ε.



(a) Constant $\varepsilon = 1$

(b) Degrading, initial $\varepsilon = 1$ degrading linearly to zero at the end of the experiment.

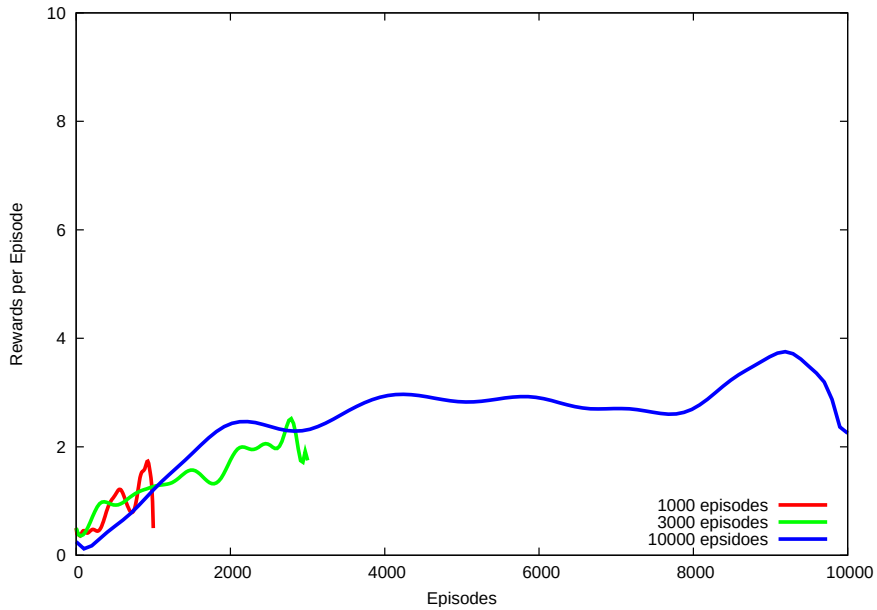
Figure 4.7: Shows the difference in performance when more rewards are represented in the state (this also gives the agent more action choices thereby also increasing the action space as well as the state space) with degrading and constant ε. Each experiment featured a single agent.

Figures 4.7(a) and 4.7(b) show the effect larger state-action spaces have on the ability of agents to learn the same simple task shown above. The lines in Figures 4.7(a) and 4.7(b) show an agent with one, two and three rewards represented in its state. The state representation contains the distance to the n nearest rewards and the agent can move towards and away from each of these rewards. With a larger combined state-action space an agent has to visit more state-action pairs to learn the task and this is shown by the poorer performance as the state-action space is increased. This is the dreaded *curse of dimensionality*.

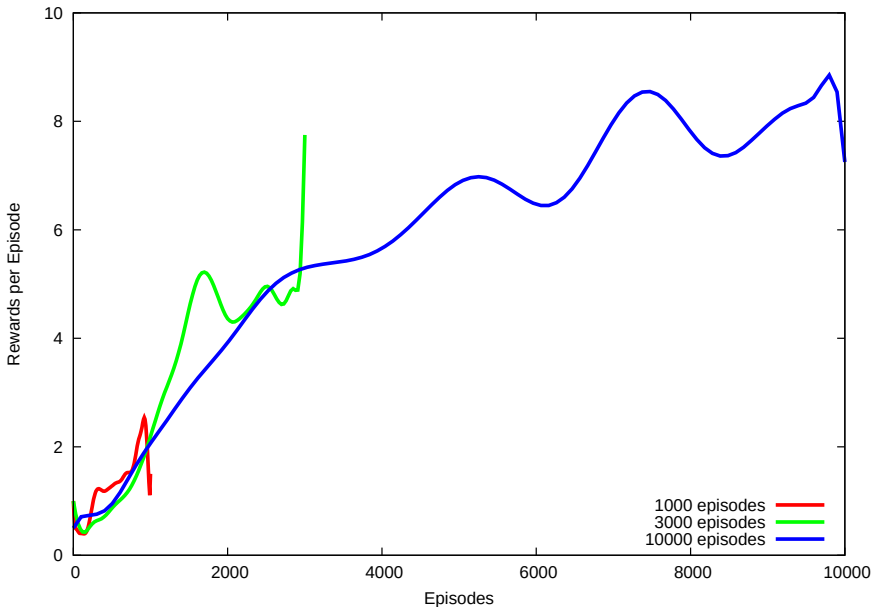
Figures 4.8(a) and 4.8(b) show the effect of allowing the agent varying periods of time in which to learn. The graph shows three lines: one for one thousand episodes, one for three thousand episodes and one for ten thousand episodes. The case with one thousand episodes is the same as the lower line shown in Figure 4.7(a) with three rewards in the state. When the agent is given longer to explore the environment and learn the value function, its performance improves. Even the modestly long runs of three thousand and ten thousand epsilons with large value function tables take a long time to run and use a substantial amount of memory on the machines used to run them. The solution is not to allow more time for the agent to learn. Consider that when the state contains distances to the *three* nearest rewards, and the agent can choose to move towards or away from each of the rewards, the combined state-action space increases to 1,804,578 entries. However, in a typical experiment, over one thousand episodes the agent only visits 19,674 of these.[3] That's just 1.1% of the state-action space covered. It will not be important for the agent to cover much of the state-action space in any significant detail or at all to still perform very well at the task set but, judging by Figures 4.7(a) and 4.7(b), 1.1% is obviously not enough. TD(λ) relies on the agent revisiting state-action pairs a large number of times, so even the states that have been visited will most likely not have been visited often enough. To illustrate this point the value function for the experiments shown in the bottom line of Figure 4.7(a) and the shortest line of Figure 4.8(a) is shown in Figure 4.9 for a constant policy and the bottom line in Figure 4.7(b) and the shortest line in Figure 4.8(b) is shown in Figure 4.10 for a degrading policy, where the basic shape of the true value functions can be seen (only just taking shape in Figure 4.9) as well as a lot of noise from states that have not been visited often enough. An example of what the value function should ideally look like can be seen in Figure 4.12. The solution is not to increase the time

[3] When given three thousand episodes to learn the agent in a typical experiment visits 25,903 state-action pairs. This increases to 37,179 when given ten thousand episodes.

spent learning but to approximate the value function as is done in Chapter 6. This is confirmed with the experiments shown in Chapter 5 that are more complicated and/ or have a larger state-action space.



(a) Constant $\varepsilon = 1$



(b) Degrading, initial $\varepsilon = 1$ degrading linearly to zero at the end of the experiment.

Figure 4.8: The experiment with 3 rewards in the state shown in the lower lines of Figure 4.7 repeated but with 3000 and 10000 instead of 1000 episodes.

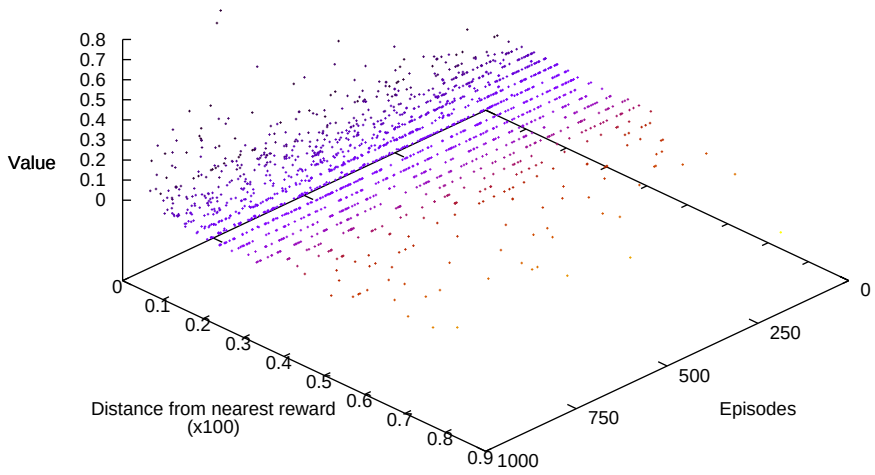


Figure 4.9: 1 agent's value function as it is updated during the experiment over 1000 episodes with 3 rewards in its state representation (constant $\varepsilon = 1$).

4.3.2 Varying α, the Learning Rate

For this simple task, the reinforcement learning algorithm is resilient to changes in the learning rate (alpha - α) when measuring the number of rewards picked up. i.e. the agent's behaviour does not change. The value function learned does change however and this is shown in Figures 4.11 through to 4.17. All the experiments used to generate these value function plots showed a performance similar to that shown in Figure 4.5. They all performed as well as each other. The value functions shown all exhibit the same general trend as the experiment progresses, to form a wave shape curve of expected return as the reward is approached toward the end of the period of learning. The shape of the wave's curve is mostly determined by the gamma value (discount factor) which was set at 0.9. That means that for each step away from a reward (in this case, of value one), the expected return falls off by 0.9 - hence the smooth shape of the curve. The graphs with smaller α values (learning rates) clearly do not reach their target value function in magnitude. They do so in shape which is sometimes all that the agent needs to perform tasks well. With the experiment for $\alpha = 1$, the value function shows a lot of noise. This can be seen as plotted values "behind" the wave

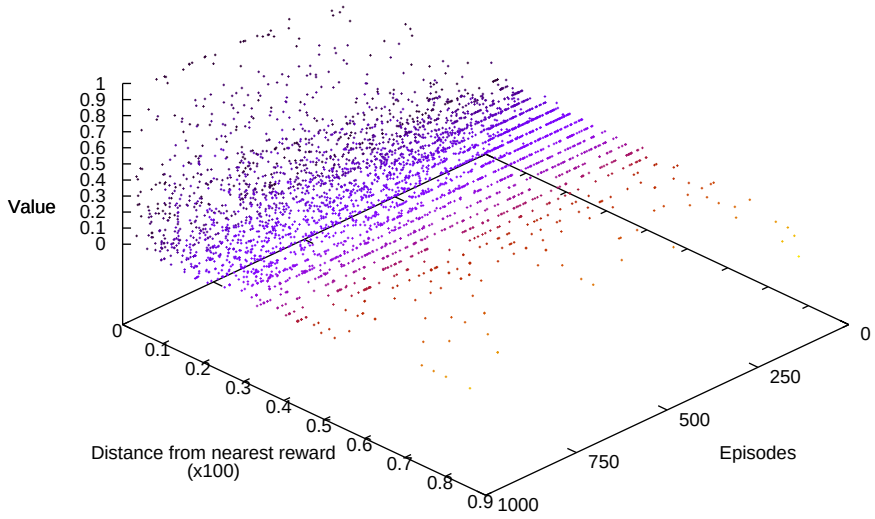


Figure 4.10: 1 agent's value function as it is updated during the experiment over 1000 episodes with 3 rewards in its state representation (degrading $\varepsilon = 1$).

shape in Figure 4.11 and is even seen a little in some of the other figures with smaller α values. Figure 4.12 gives the cleanest value function approximation. It appears to have come to this shape quickly without too much noise being generated by an overly large α value, as in Figure 4.11. Figure 4.13 also shows a good final value function shape but, only just manages to attain the shape and magnitude by the end of the experiment. An α value of 0.1 is used for the rest of the experiments in this thesis as suggested by these figures. 0.1 is a commonly used α value in existing literature and research. One can see that reinforcement learning is fairly resistant to changes to the learning rate and will produce the shape, if not the magnitude, if the learning rate is too low. If the learning rate was set too high it produced the shape and magnitude of the true value function but with some imperfections as utility values are adjusted too quickly during learning.

4.4 Summary

This chapter introduced the foraging task and gave examples of the performance of one agent under various experimental conditions such as different learning rates, state-

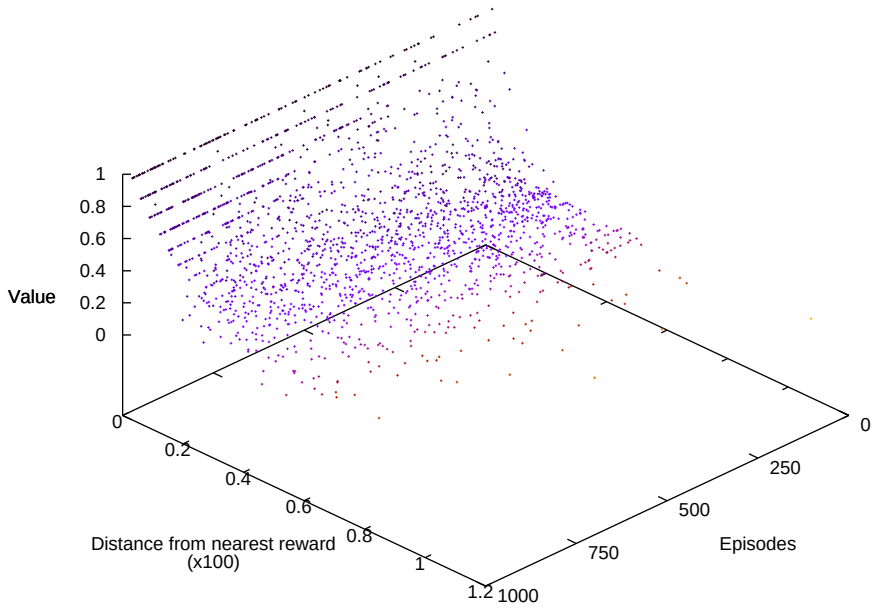


Figure 4.11: One agent's value function as it is updated during the experiment with $\alpha = 1$.

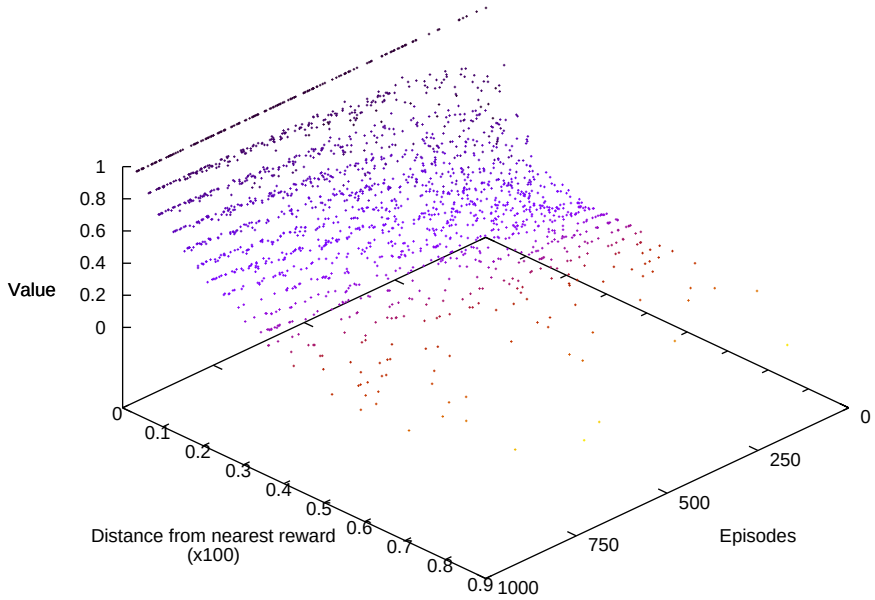


Figure 4.12: One agent's value function as it is updated during the experiment with $\alpha = 0.1$.

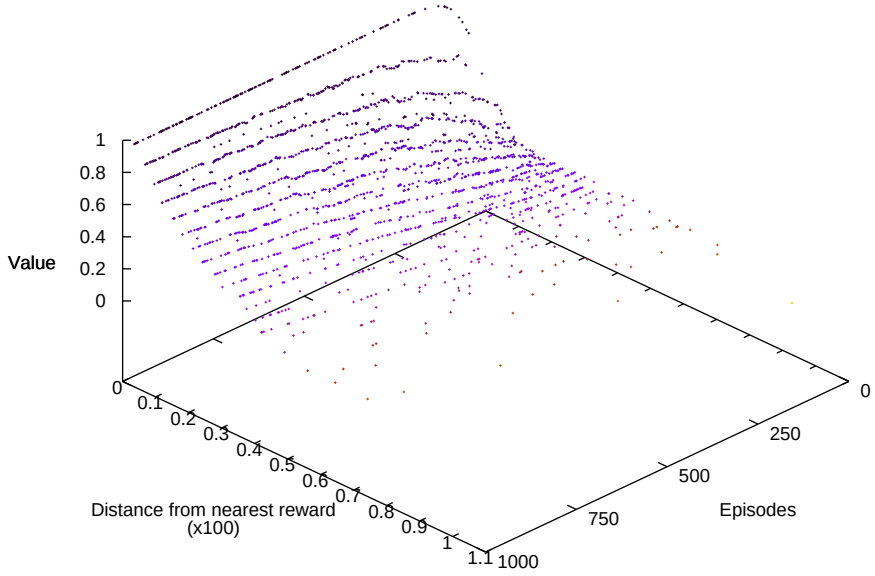


Figure 4.13: One agent's value function as it is updated during the experiment with $\alpha = 0.01$.

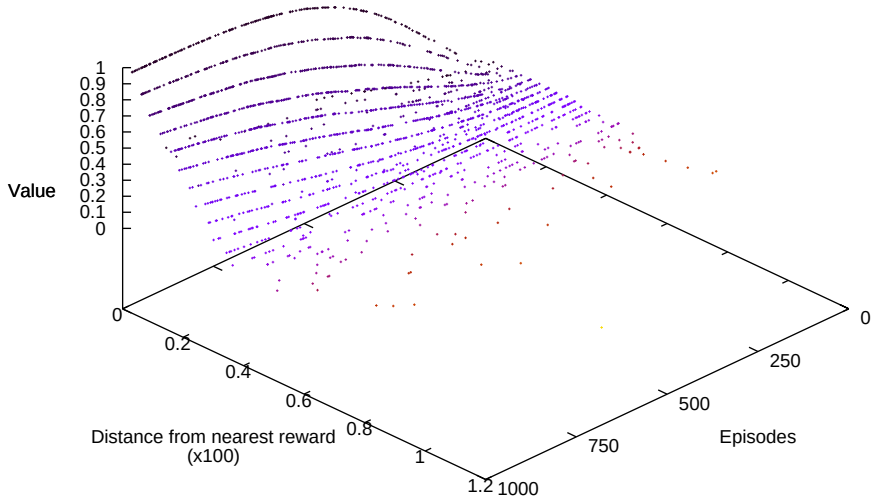


Figure 4.14: One agent's value function as it is updated during the experiment with $\alpha = 0.001$.

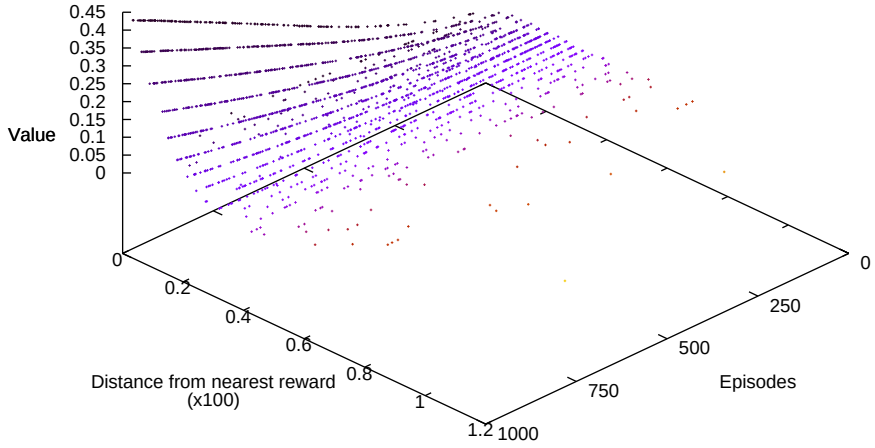


Figure 4.15: One agent's value function as it is updated during the experiment with $\alpha = 0.0001$.

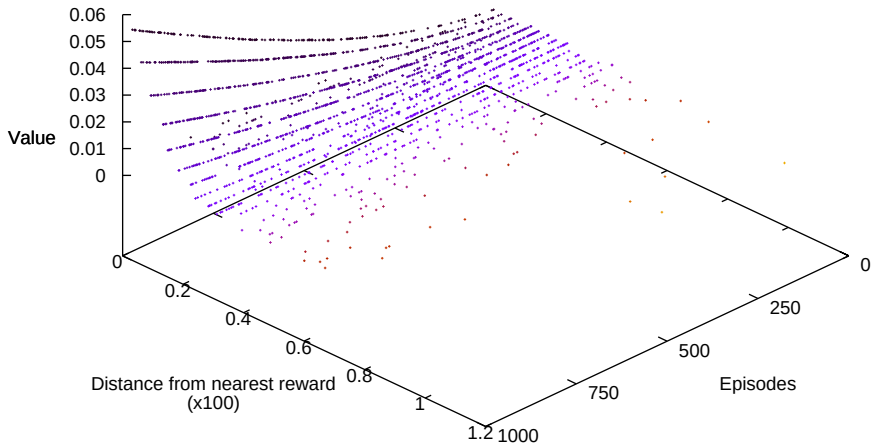


Figure 4.16: One agent's value function as it is updated during the experiment with $\alpha = 0.00001$.

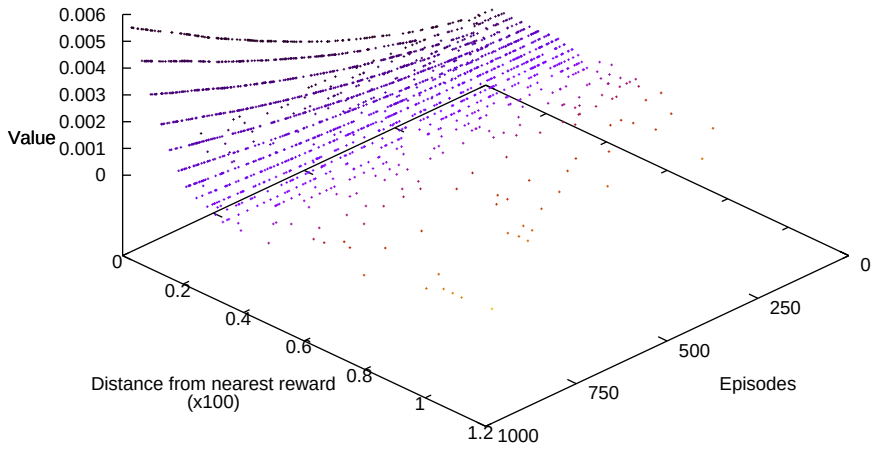


Figure 4.17: One agent's value function as it is updated during the experiment with $\alpha = 0.000001$.

action space sizes, number of episodes to learn in and action selection policies. It has been shown that the state-action space increases very quickly as it succumbs to *curse of dimensionality*, making it difficult for the agents to learn even simple tasks. This can be somewhat alleviated by running the experiments for longer but this significantly increases both the time needed to learn and the amount of memory needed to hold the value function. It was shown that a learning rate (α) of 0.1 is likely to be a good value for learning the correct behaviour and this was shown by looking at learned value functions. These value functions are shown again when using function approximation in Chapter 6 and it is interesting to see the differences.

No conclusive difference was shown between keeping a constant policy and using a non-stationary policy by degrading ε or for the different initial values of ε. Thus all the experiments to come are shown using a constant and a linearly degrading ε value for a range of ε values.

The next chapter introduces the group utility functions to this basic task. First, in Section 5.1 two characters are put into the environment performing the same foraging task together. In one experiment the agents are unconstrained aside from their goal of picking up rewards. In another experiment, the agents are affiliated with a group util-

ity function that uses the negative standard deviation to combine the agents' privately collected rewards, which encourages an equal number of rewards to be picked up each episode. Next, the number of agents per group is increased for several experiments as shown in Section 5.2. Experiments with uneven groupings of agents are then shown where one agent along with a group of two or three agents learn together. The experiments with a larger number of agents forseeably run into problems learning with a large state-action space using a tabular value function over a reasonably short training period so, in Chapter 6, the experiments are repeated using function approximation methods to represent the value function. Function approximation should not suffer from relatively short periods of learning as it generalises feedback thus, making more efficient use of time spent learning. It also should not suffer from high memory use due to the compact nature of the neural networks used for approximation. As an illustrative example for the equivalent tabular and approximated value functions experiments from Figures 6.27 and 6.38, the approximated value function experiments took exactly the same time to run and used 38% less memory (at peak memory use). In a similar comparison but with larger state-action spaces (in experiments from Figures 6.32 and 6.43) the approximated value function experiments took the same time to run and used 50% less memory (at peak memory use). The fact that the tabular value functions did not use vastly greater amounts of memory in practice was due to storing the sparsely populated tabular value function as a hash-table.

Chapter 5

Using Group Utility Functions

This chapter presents results from a number of experiments showing that by using balancing and team group utility functions a quantitative goal can be learned (foraging for rewards) whilst learning a qualitative aspect of behaviour (a simple equilibrium between two agents, groups or an agent and group).

The first section (5.1) of this chapter describes the set-up and shows the results for an experiment similar to that seen in Section 4.1 but with *two* agents that learn to forage whilst *also* learning to perform equally as well as each other. This is the first experiment that shows applied group utility functions in a computer game environment. The work is based on a previously published "proof of concept" paper [Bradley and Hayes, 2005a]. The second section (5.2) builds on the first with more complex experiments which show the limitations of the learning algorithm as the number of agents involved increases (this work is based on another previously published paper [Bradley and Hayes, 2005b].

The group utility used to bring about an equal level of performance between two agents or groups is the negative standard deviation of both agents or groups as was shown in Equation 3.1 of Section 3.4. The graphs show that using this group utility function, a simple equilibrium can be learned at the same time as optimising the quantitative goal of accruing rewards. That is, when learning to perform in equilibrium the difference in rewards picked up each episode can be kept lower than if the agents or groups are unconstrained. The graphs also show that this equality of performance comes at a price in quantitative maximisation as the agents picked up less reward in total due to efforts spent on actions to bring them or their groups into balance. Note that the equality of performance is learned between the two single agents or the two groups and not within the groups of agents. Each group of agents acts as a team under a sum

group utility function and no balancing takes place between the agents in a group.

Finally, the graphs show that tabular value functions perform well when used with simpler tasks and with smaller state-actions spaces but fail to support learning for larger state-action spaces and more complex tasks.

5.1 Initial Experiments Using Group Utility Functions

There are two experiments in this section. The first experiment shows the base case of two agents learning to forage for rewards in the same environment but paying no attention to each other. The experiment is set up the same as the experiments in the Chapter 4. The only difference is that the state representation for each agent contains information about the other agent (actually, the nearest agent to any reward in the environment). The state representation holds the distance of that other agent to its nearest reward, whether that other agent has collected more, less or the same amount of reward (i.e. winning, losing or drawing) as the agent and if the other agent is in the same team or not.[1] As stated in Chapter 4, each experiment in this thesis was run four times and all graphs shown are the result of averaging these four runs. The second experiment in this section again shows the two agents learning to forage for rewards but, this time, trying to maintain equal amounts of items collected at any one time. The experiments use the same parameters as detailed in Section 4.2.4 (surmised in Table 5.1) and have a constant ε value of 1 for ε-greedy action selection. The layout of the agents and group utility functions for each experiment is shown in Figures 5.1 for the agents acting individually and 5.2 for the learning to be balanced agents.[2]

It is expected that the two unconstrained agents will learn to pick up approximately twice as many rewards as the single agent in Chapter 4. Some inefficiency may be seen because the two agents may compete for the same reward leading to wasted actions. Performance for the two constrained agents should be lower than for the unconstrained agents. Due to the extra actions taken in order to maintain balance the agents will not have as many actions available to pick up rewards. This should be seen in Figure 5.3. Figure 5.4 shows the difference in number of rewards picked up by both the uncon-

[1] Being in the same team or not means little for this experiment but is kept in the state to provide better comparison with the later experiments.

[2] This chapter and Chapter 6 contain several figures similar to 5.1 and 5.2. Each of these figures is a representation of a hierarchy of the agents and group utility functions used in a set of experiments. The diagrams show a simplified flow of reward *upwards* through the group utility functions and *back down* as *modified reward* to be used by the agent to learn. A full example of this process is given in Figure 3.1.

strained agents and the agents constrained by the balancing group utility function. It is expected that this graph will show that there is a natural difference in performance between the two unconstrained agents and that this difference is smaller for the agents learning to maintain balance. In fact this is the expected pattern for each set of future experiments.

The performance of the agents in both experiments is shown in Figure 5.3 whose curves show the number of rewards picked up each episode by both agents together throughout the learning experiment. Time is along the x-axis and so at the left hand edge of the graph no learning has taken place and at the right hand edge the agents should have reached asymptotic performance. The agents are trying to increase the number of rewards picked up in both experiments so the higher these curves the better. The curves in Figure 5.4 show the absolute difference in the number of rewards picked up by each agent for each episode throughout learning. These curves represent the difference in performance between the two agents. The upper curve shows the difference in performance when there is no encouragement to perform equally (agents arranged as in Figure 5.1) and the lower curve shows the difference when learning to perform equally as well as each other (agents arranged as in Figure 5.2). These two experiments show that:

- the agents can still perform their quantitative goal of maximising the number of rewards picked up when under the balancing group utility function;
- the agents can learn to perform equally as well as each other when under the balancing group utility function.

The experiment with the two **individual** agents is run to show the normal performance of the two agents under the same conditions. For every experiment in this thesis a corresponding experiment was run without the agents and/ or group under the null group utility function so that comparisons can be made clearly.

5.1.1 Results for Initial Experiments with Group Utility Functions

Figures 5.3 shows the performance of the two agents when under the *null* and *balancing*. The curves show the combined number of rewards picked up each episode by both agents. As before and for all graphs herein, the graph points are calculated by testing

α (Learning rate)	0.1
γ (Discount rate)	0.9
λ (Eligibility trace discount factor)	0.9
Maximum length of eligibility trace	20
Rewards present in environment	20
Agents move in steps of	3m
Agents pick up rewards within radius of	3m
Rewards replaced	20m from any existing reward
Rewards replaced	10m from any agent
Environment size	$160m^2$

Table 5.1: Common parameters used in all experiments in Chapters 5 and 6

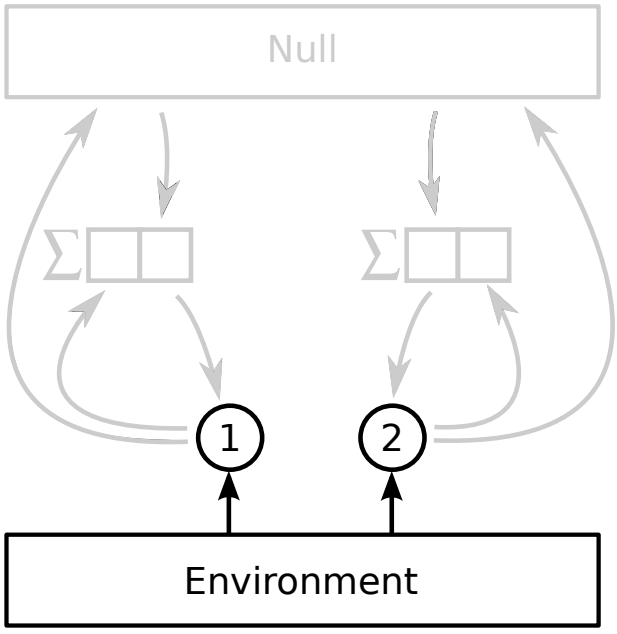


Figure 5.1: The layout of the unconstrained agents (1 and 2) and group utility functions for the upper curve shown in both Figure 5.3 and 5.4. The null utility function returns zero. i.e it has no effect on the reward signal and so is greyed out in the diagram. It is included in this diagram for comparison with Figure 5.2. See Footnote 2 of Section 5.1 for explanation of diagram notation.

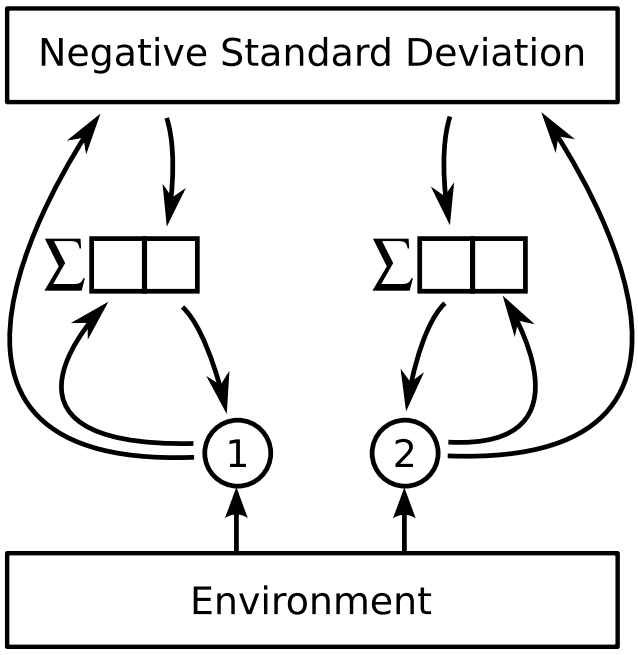


Figure 5.2: The layout of the balanced agents (1 and 2) and group utility functions for the lower curve shown in Figure 5.3 and 5.4.

the agents at regular intervals throughout the experiments. The agents are tested for an entire extra episode whilst being greedy in action selection for each point on the graphs. The reader can see that the agents unhindered by learning to perform equally learned to pick up more rewards per episode than the balanced agents by approximately two to three rewards. Figure 5.4 shows the absolute difference in the number of rewards picked up by each agent again under either the *null* or *balancing* group utility functions. From Figure 5.4 one can see that when acting individually the agents differ in the number of rewards picked up each episode by between two and two and a half rewards whereas the agents under the balancing group utility function differ by far less, a half to one reward. It can be said that the balanced agents have an inequality aversion (Section 3.1). Compared to the two agents acting individually the agents are able to learn and maintain a balance between the performance of two agents whilst learning a main quantitative task and the final policies maintain that balance. The results show that what is gained in balance is lost in performance. This makes intuitive sense. With randomly placed agents and rewards, a certain amount of extra actions will be needed by the agents to maintain an equal number of collected rewards.

The curve that shows the absolute difference between the number of rewards picked up by each agent in Figure 5.4 never reaches zero. One may think that the difference

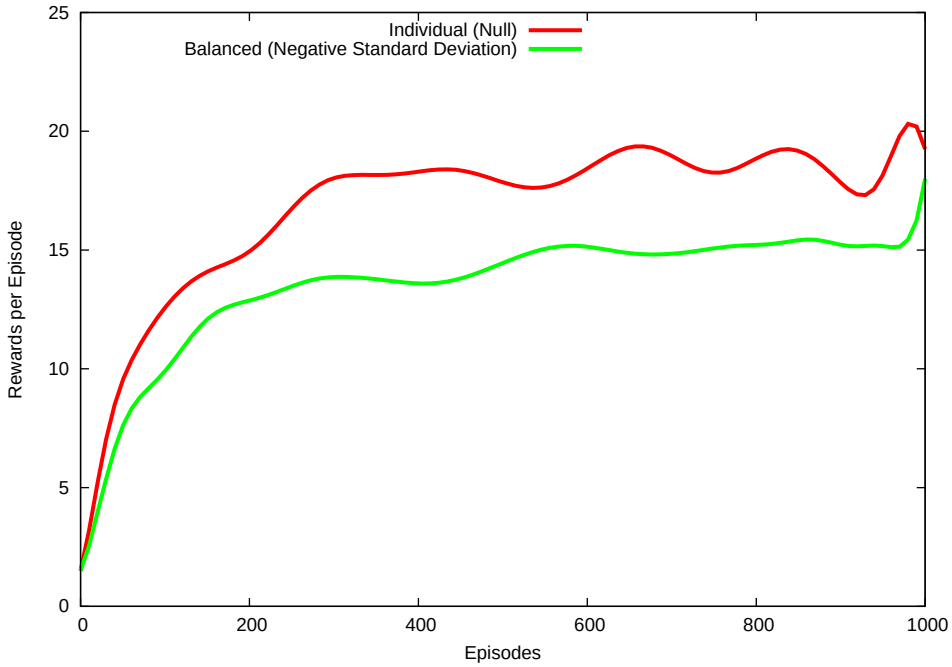


Figure 5.3: Compares the performance of two agents under a null group utility function and under a negative standard deviation group utility function. One can see the drop in performance moving from the null function to the negative standard deviation function.

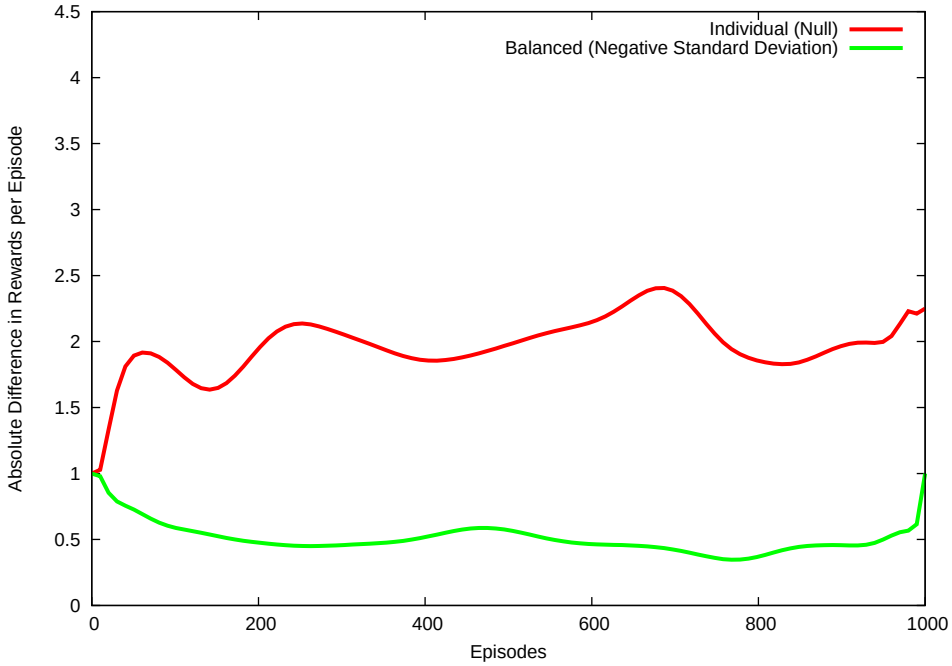


Figure 5.4: Compares the difference in performance of two agents under a null group utility function and under a negative standard deviation group utility function. The curves show the absolute difference between the number of rewards each agent has picked up. The top curve shows that there is a natural difference in how well the agents perform and the bottom curve shows that the agents have learned to balance their performance reasonably well. That is, to within one reward or less on average each episode.

should reach zero as the agents are learning to pick up an equal amount of reward. However, punishing the agents for differing by one reward, discourages the agents from picking up any rewards at all. The negative standard deviation equation used, as shown in Equation 3.1, only discourages imbalance for a difference of two or more rewards. For a difference of one reward, the positive reward for picking up that reward outweighs the negative reward caused by imbalance. For a difference of two rewards, the positive reward of picking up another reward is the same as the negative reward for causing such imbalance. For a difference of three rewards, the positive reward for picking up yet another reward is outweighed by the negative reward for causing a large imbalance. As the difference in number of rewards grows so does the negative standard deviation part of the reward signal. The different parts of the reward signal can be scaled using *group affiliation* values to alter the interaction of different group utility functions that make up the modified reward signal. Remember, that it is the modified reward signal that the agents use to update their value functions.

To keep results comparable, the state representations for the graphs in this chapter and those comparable graphs in Chapter 6 are built similarly and parameters varied in the same standard way for each set of experiments. This sometimes means not using the most efficient state representation for learning as in the first set of experiments here. For each layout of agents, many experiments were run and this has generated many graphs. To keep explanation simple these graphs have been moved to the appendices and a set of *interesting* curves from those graphs are included in the main body of the thesis. Experiments were run for each set-up of the agents and group utility hierarchy:

- Two **individual** agents (as in Figure 5.1);
- Two **balanced** agents (as in Figure 5.2);
- Two **individual** groups of **two agents each**, with each group acting as a **team** (i.e. the four agents are split into two groups. Each group of two agents is under a sum group utility function and the two groups themselves, are under a null group utility function. A *team* here, refers to a group with a group utility function that uses the sum combining function - the same as the team utility function.);
- Two **balanced** groups of **two agents each**, with each group acting as a **team**;
- Two **individual** groups of **three agents each**, with each group acting as a **team**;
- Two **balanced** groups of **three agents each**, with each group acting as a **team**;

- One group of **two agents** acting as a **team** and, **one agent** acting **individually**;
- One group of **two agents** acting as a **team**, **balanced** with **one agent**;
- One group of **three agents** acting as a **team** and, **one agent** acting **individually**;
- One group of **three agents** acting as a **team**, **balanced** with **one agent**.

Several different state representations were tried for each hierarchy of agents and group utility functions[3] For each different state representation the size of the combined state-action space is given as described in Sections 4.2.5 and 4.2.6. The first state representation shows more details of calculating the state-action space size:

- The distance to the nearest reward in the state (67 possibilities); the distance of the nearest agent to **any** reward (67 possibilities), whether that agent is in the same team or not[4] (2 possibilities) and if that agent or agent's group (if in a group other than that of the agent whose state is being described) has collected more reward, less reward or the same amount of reward during the current episode (3 possibilities). [State-action space size: (The agent can move towards or away from each reward in its state representation: 2 possibilities) $67 \times 67 \times 2 \times 3 \times 2 = 53,868$];
- The distance to the nearest reward in the state; the **distance of the nearest other agent to that nearest reward**, whether that agent is in the same team or not and, if that agent or agent's group has collected more reward, less reward or the same amount of reward during the current episode. [State-action space size: $67 \times 67 \times 2 \times 3 \times 2 = 53,868$];
- The distance to the nearest reward in the state; the distance of the **nearest other agent** to that nearest reward **and** the distance of the nearest agent to **any** reward, whether **those agents** are in the same team or not and if those agents or agents' group has collected more reward, less reward or the same amount of reward during the current episode. [State-action space size: $67 \times 67 \times 67 \times 2 \times 2 \times 3 \times 3 \times 2 = 21,654,936$];

[3] Measures in these states are taken relative to the agent that is having their state set unless otherwise stated.

[4] This is irrelevant for experiments with just two agents as there is only one other agent but is kept in the state representation to maintain comparable state-action spaces across experimental set-ups.

- The distance to the **two nearest rewards** in the state; the distance of the nearest agent to any reward, whether that agent is in the same team or not and if that agent or agent's group has collected more reward, less reward or the same amount of reward during the current episode. [State-action space size: $67 \times 67 \times 67 \times 2 \times 3 \times 4 = 7,218,312$];

- The distance to the two nearest rewards in the state; the distance of the **nearest other agents (one nearest agent to each nearest reward) to those nearest rewards**, whether **those** agents are in the same team or not and, if those agents or their groups have collected more reward, less reward or the same amount of reward during the current episode. [State-action space size: $67 \times 67 \times 67 \times 67 \times 2 \times 2 \times 3 \times 3 \times 4 = 2,901,761,424$];

- The distance to the two nearest rewards in the state; the distance of the nearest agents to the nearest reward (one nearest agent to each nearest reward) **and**, the distance of the nearest agent to any reward and, whether those agents are in the same team or not and if the agent or their group has collected more reward, less reward or the same amount of reward during the current episode. [State-action space size: $67 \times 67 \times 67 \times 67 \times 67 \times 2 \times 2 \times 2 \times 3 \times 3 \times 3 \times 4 = 1,166,508,092,448$];

Each experiment was repeated with agents using both a constant action selection policy (constant ε) and a non-stationary policy (ε linearly degraded to zero at the end of the experiment) for ε values of 0, 0.2, 0.4, 0.6, 0.8 and 1 as shown in Figure 4.6.

As an example of the types of graphs produced by these experiments, Figures 5.5 through to 5.8 show the graphs for the experiments with just two agents; acting individually and learning to be balanced; with the distance to one nearest reward and information about the nearest agent to any reward in the state representation for each agent and; when the agents are using a constant policy. For each experiment two graphs are shown, the performance of the agents in terms of overall number of rewards collected each episode and the absolute difference between the agents (or groups, or the agent and group in later experiments) at the end of each episode. One can see from Figures 5.5 through to 5.8 that almost all of the curves follow the same trend as previously seen in Figures 5.3 and 5.4. Only the curves for low values of ε such as zero or 0.2 perform badly in rewards picked up (Figures 5.5 and 5.7). It is difficult to see that the poor performance of these curves in Figures 5.6 and 5.8 because poor performance measured as rewards picked up more than likely means the agents also

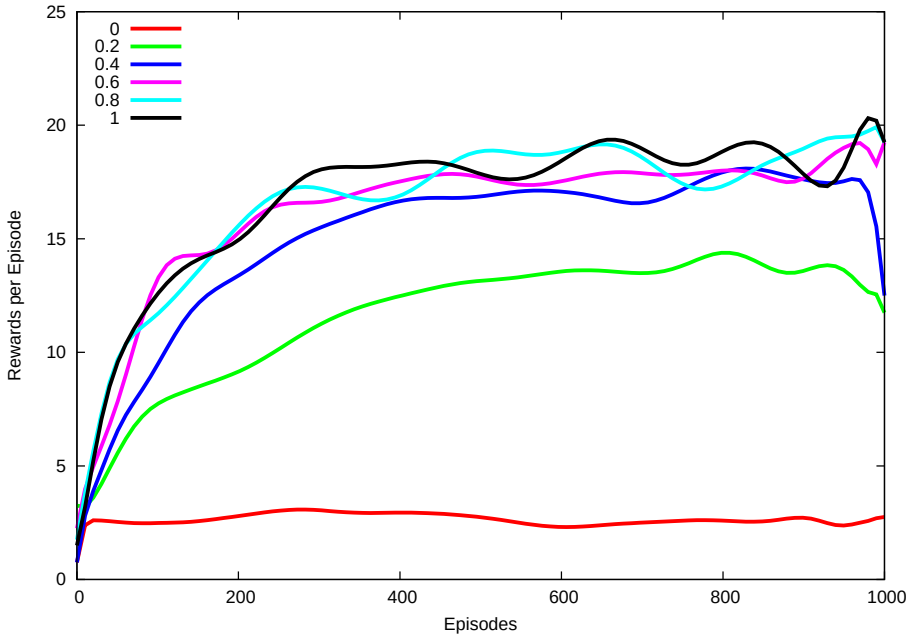


Figure 5.5: Various values of ε for two individual agents.

have a low absolute difference in rewards picked up each episode. All of the graphs that show the absolute difference between agents or group or agents and groups in this thesis should be read with the corresponding graph showing the number of rewards picked up. This way one can see that the balancing group utility function is working to produce balanced agents *without* unduly negatively affecting performance.

Graphs such as these are included in the appendices for each and every experiment run. It was necessary to run these experiments to find parameters (such as ε) that proved to work consistently. Many experiments were run up to the point of running those for inclusion in this thesis to determine good values for α, γ, λ, dimensions of environment, length of eligibility trace, etc.[5] Such a parameter search generates many graphs so from Figure 5.9 onwards a subset of curves generated are plotted. Figures 5.9 and 5.10 show a selection of interesting, representative experiments (performance and difference respectively) from the other combinations of state representation and action selection policies.[6] The graphs shows curves for the agents acting individually and balanced, for $\varepsilon = 1$ and for each of the six different state representations (one of which was already shown in Figures 5.5 through to 5.8). The full graphs for each of

[5]The chosen values are presented in Table 5.1.

[6]The selection of experiments plotted in Figures 5.9 and 5.10 are the same set chosen to show in the main body of the thesis for each set or experiments run.

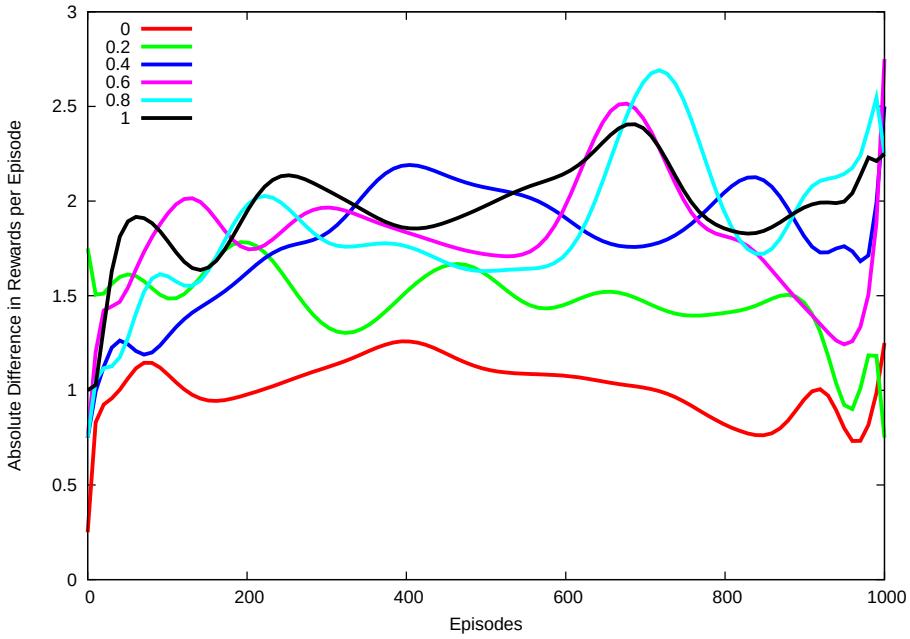


Figure 5.6: Various values of ε for two individual agents.

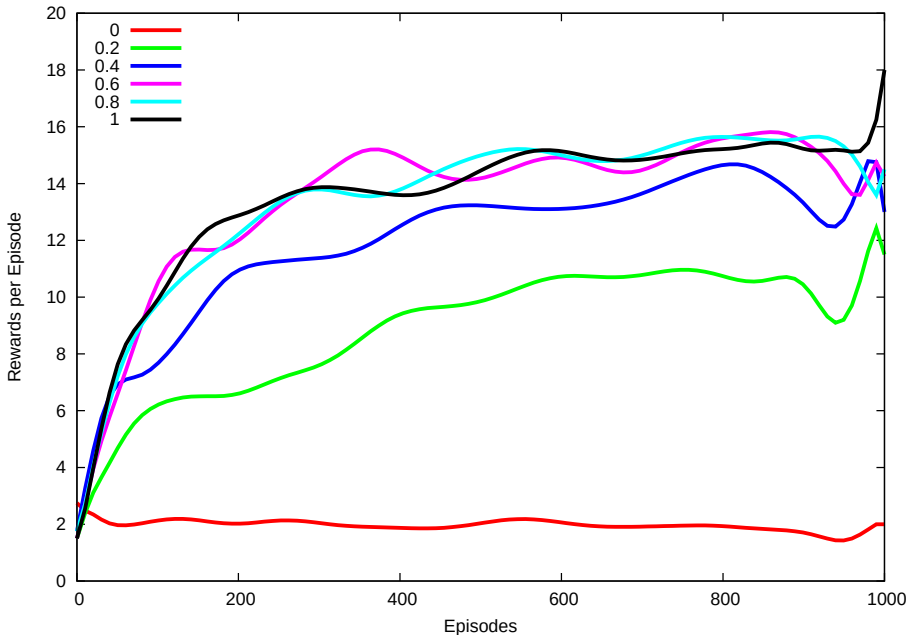


Figure 5.7: Various values of ε for two balanced agents.

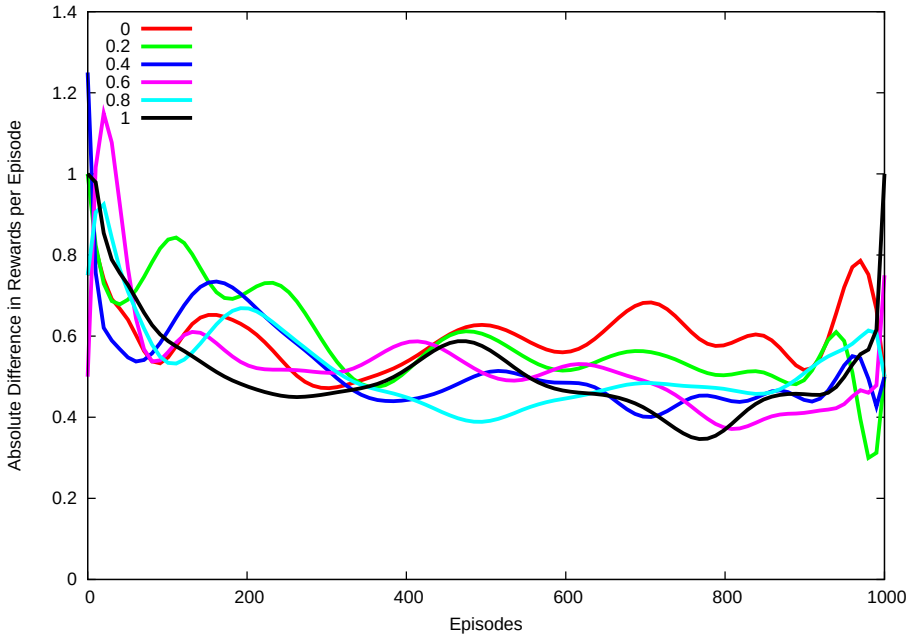
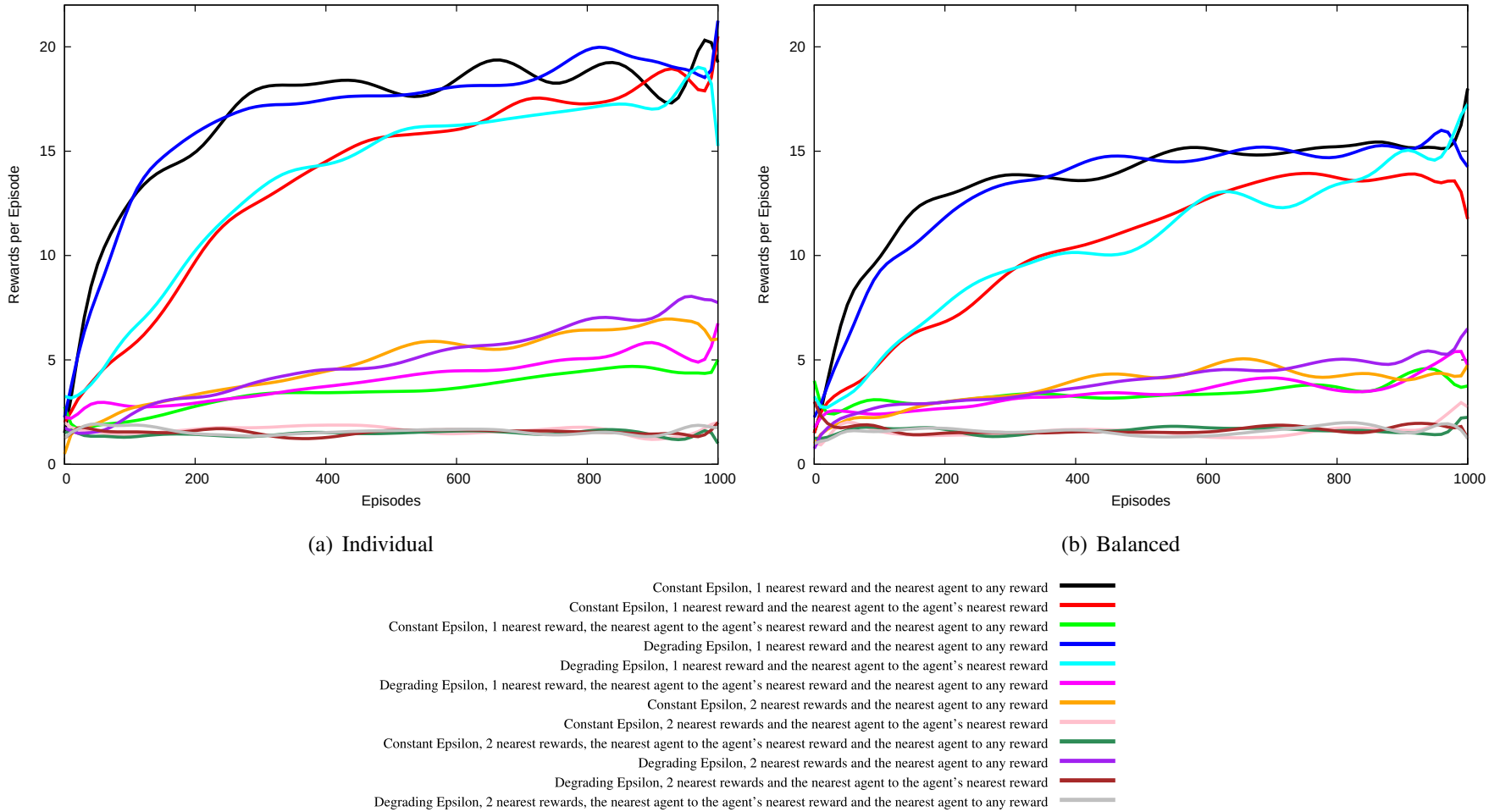


Figure 5.8: Various values of ε for two balanced agents.

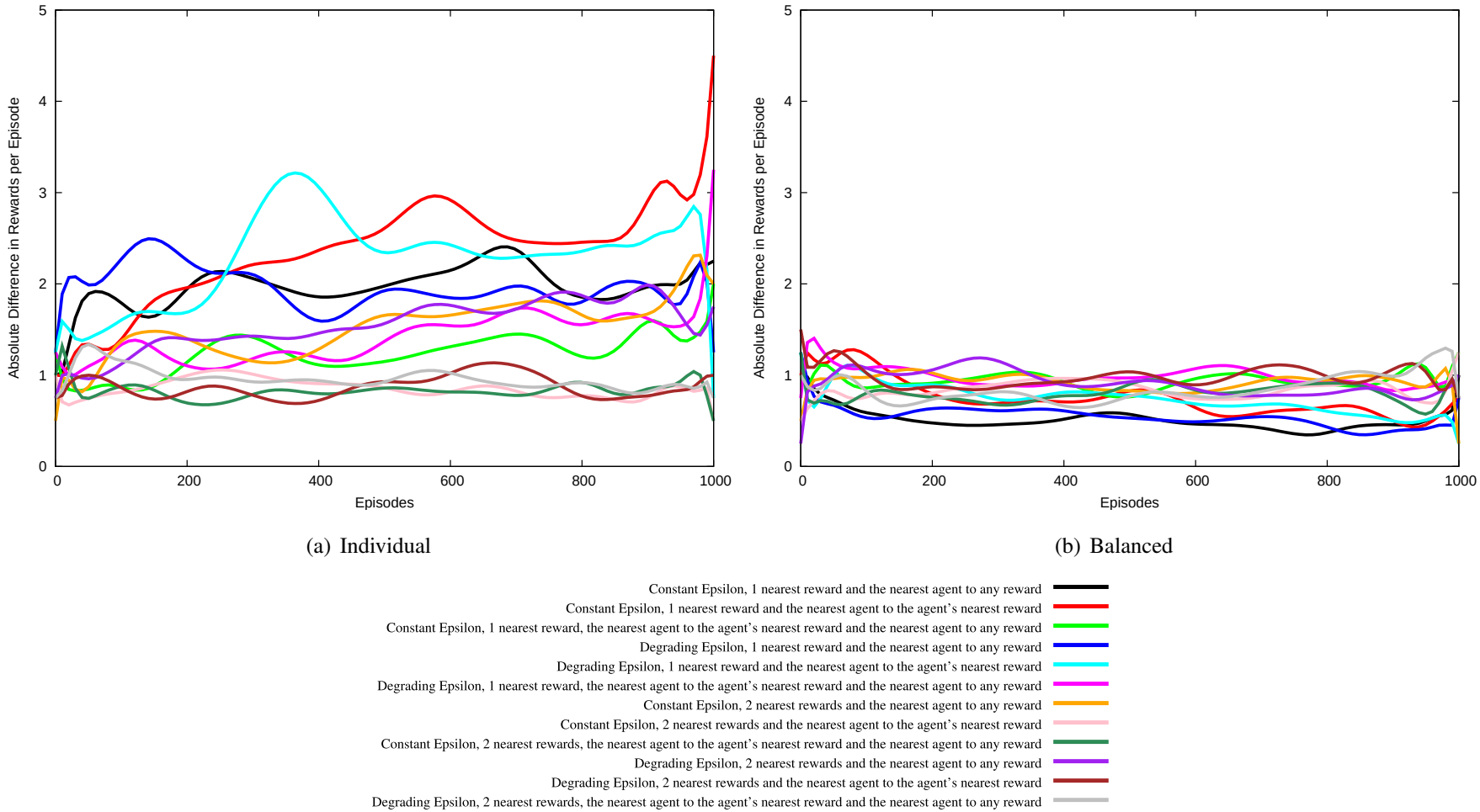
these set-ups can be seen in the appendices from Figure A.3(a) to A.10(f). $\varepsilon = 1$ was chosen as it gives a good performance for each experiment in this chapter and Chapters 4 and 6. $\varepsilon = 0.8$ could have been chosen as this often seems to give a good performance also but, is left out of the main body of the thesis for brevity's sake and shown, along with other values of ε, in the appendices instead.



(a) Individual

(b) Balanced

Figure 5.9: Performance of two agents when acting individually and when balanced with all curves with initial $\varepsilon = 1$.



(a) Individual

(b) Balanced

Figure 5.10: Absolute difference in number of rewards picked up by two agents when acting individually and when balanced with all curves with initial $\varepsilon = 1$.

Figure 5.9(a) shows two curves that perform well; two other curves that do reasonably well; four curves that perform to some degree and four curves that clearly show no improvement in performance due to learning. The curves that perform well are for constant and degrading $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward in the environment. This is the most compact state representation of the experiments. A state representation as compact is used for the next two curves that still perform reasonably well. These two curves are again for constant and degrading $\varepsilon = 1$ when there is one nearest reward represented in the state *but* information on one nearest agent to that reward. The fact that these curves indicate poorer performance than when the nearest agent to any reward is used is interesting. In both cases (nearest other agent to any reward or nearest other agent to the agent's nearest reward) the other agent is the same agent as there is only one other agent. The only difference is that the agent knows the distance of the other agent to the other agent's nearest reward or the agent's nearest reward. Theoretically, both of these options could have been useful to the agent. Empirically at least in this graph there is a difference though more so in the early stages of learning as the asymptotic performance is not that different and could be accounted for by random variation. The next best performing pair of curves is for constant and degrading $\varepsilon = 1$ when there are two nearest rewards represented in the state and information on one nearest agent to any reward in the environment. This is expected as although the agents know more about their environment and have have more choice over which reward to move to[7] the state-action space is larger and thus more difficult to cover in limited time. This result is predictable when looking at Figure 4.7. The next best performing pair of curves are for constant and degrading $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to that reward *and* information on one nearest agent to any reward in the environment. These two other agent's (the nearest to the agent's nearest reward and the nearest to any reward) are of course the same agent but it might follow that because the best two performing pairs of curves had information about the nearest other agent to the agent's nearest reward or the nearest other agent to any reward that both bits of information might produce better performance. However, the state space is larger than for the pair of curves above these two and that is most likely the determining factor. In the four experiments plotted on the lower curves the agents did not learn to pick up any more rewards than random action did at the start

[7]The agents can choose to move towards or away from each reward in the state and two rewards are in this state representation.

of the experiment. This is because the state is larger still. The only interesting fact is that two of those curves were for constant and degrading $\varepsilon = 1$ when there are two nearest rewards represented in the state and information on one nearest agent that reward. The corresponding experiments with the same state representation but with the nearest other agent to any reward rather than nearest other agent to the agent's nearest reward perform significantly better. This indicates that there may well be a significant difference between keeping information on the nearest other agent to any reward or the other agent nearest to the agent's nearest reward.

Figure 5.9(b) shows the same experiments but when the two agents are learning to perform equally as well as each other. The curves are grouped, with regard to performance, exactly the same as in Figure 5.9(a) when behaving individually. The interesting thing to note is that for all the experiments the agents have learned to pick up fewer rewards each episode by the end of the experiments. For the best performing experiments the drop in performance is approximately four or five rewards per episode. This is the loss of performance due to learning under the balancing group utility function.

Figure 5.10 shows the absolute difference in the number of rewards picked up by each agent for the same set of experiments as in Figure 5.9. Figure 5.10(a) shows the naturally occurring difference in performance of the two agents when learning individually and can be used to gauge how successfully the agents shown in Figure 5.10(b) have learned to perform equally as well as each other. One can see from Figure 5.10(a) that the natural difference in performance per episode for the best performing two experiments in Figure 5.9(a) is just over two rewards. The same experiments shown in Figure 5.10(b) should show less difference in rewards picked up per episode and they do. The balanced agents for constant and degrading $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward in the environment show a difference of half a reward. That is less than one reward's difference on average which successfully shows that those agents have learned to balance their performances. The same can be said for all of those experiments in Figure 5.10(b) that show significantly less inequality than their corresponding experiments in Figure 5.10(a). That is except for those experiments which failed to learn (shown in the lower four curves of Figures 5.9(a) and 5.9(b)).

This first set of experiments can be considered in so much as they show that using group utility functions, two agents have learned a quantitative goal of maximising the number of rewards picked up whilst at the same time learning to respect a qualitative goal - that of performing equally as well as one another. To acquire this secondary,

qualitative behaviour came at a loss of approximately 22.5% of the quantitative goal. Most of this loss is probably due to the agents having to take more action steps to avoid becoming out of balance but some of it may be due to the task being more difficult for reinforcement learning[8].

In the next section (5.2) the issues of scaling up the experiments are investigated. Group utility functions should scale to learning equilibriums between groups of agents as well as single agents. In the next chapter, group utility functions are used to learn an equilibrium between teams of two and three agents (Section 5.2.1) and also between teams of two or three agents and a single agent (Section 5.2.2). Learning an equilibrium between unbalanced groups is inherently more difficult as the balancing part of the reward signal is more evident. Even groups are somewhat naturally balanced and the effect of the negative standard deviation group utility function should be seen less often and at less magnitude by the agents.

5.2 Issues of Scale

Building on the previous section's (5.1) results and confident that using group utility functions can mix a main quantitative goal with a secondary qualitative goal, this section presents further experiments undertaken to see how the algorithm in the previous section scales when more agents are added. In this section there are six experiments split into two sections: even groups (Section 5.2.1) and uneven groups (Section 5.2.2) of agents. There are two experiments showing how the algorithm works when teams of even numbers are asked to learn to forage whilst attempting to perform equally as well as each other. The other two experiments show how the algorithm copes with uneven groups of agents. Both sets of experiments are run for constant and non-stationary policies with varying epsilon values and are preceded by their unconstrained equivalent experiments. That is, the same experiments except under the null group utility function instead of the balancing group utility function as shown before. The graphs shown are similar to those in Figures 5.9 and 5.10 with all the experiments graphed in full in the appendices. Trying to keep equilibrium when the groups are inherently unbalanced (uneven numbers of agents) is a harder task so it is interesting to see if the qualitative equilibrium part of the reward function can still be optimised or if the

[8]There may be ways to calculate what the best possible balanced performance could be by using handwritten behaviour rules to see if 22.5% is around the least loss of performance possible to keep balanced.

equality relationship will be lost.

The group affiliation values (as in Section 3.5.2) are fixed, at *one*, for the experiments with evenly matched groups as they were in the last experiments. The group affiliation values exist to scale the reward signal from group utility functions to lower or raise their magnitude to an appropriate level when considering the other parts of the reward signal and the task the designer has in mind. For the preceding experiments and the experiments with even groups (Section 5.2.1) all group affiliations were and are set at *one*. That is, the parts of the reward signal coming from each group utility function were left unaffected. For the experiments with uneven numbers of agents in each group in Section 5.2.2 (i.e. two agents in a group alongside a single agent and three agents in a group alongside a single agent) the magnitude of the output of the balancing group utility function compared to the sum group utility function becomes important. Because there is more than one agent under a sum function the balancing part of the reward signal can be easily swamped. At the same time the single agent is left to the "full force" of the balancing part of the reward signal. This was not an issue in the evenly matched groups experiments as both groups were affected in the same way (i.e. they were both under a sum group utility function). Whereas, in the case where a group is learning alongside a single agent the single agent is affected far more by the balancing signal than the group is. Thus the balancing component of the reward signal needs to be scaled up to affect the agents in the group approximately as much as the single agent is affected.[9] Otherwise the single agent is "punished" to a greater extent than the grouped agents are for imbalance. For the experiments with uneven groups the group affiliation values were set intuitively. For example, in the case of a single agent learning to perform equally as well as a group of three agents (Figure 5.24) the group affiliation value between the sum group and the balancing group utility function was *three* and between the single agent and balancing function, *one*. The assumption is that three agents should be "restrained" by the balancing signal by three times the amount that one agent should be.

As before the foraging task has been used for all the experiments here and the set-up is kept entirely the same except for the hierarchy of the agents and groups (and group affiliation values for those unevenly distributed groups of agents). As with all the experiments, the graph points are calculated by testing the agents at regular intervals throughout the experiments. The agents are tested for an entire extra episode whilst

[9]Conversely, the output of the sum group utility function could have been scaled down so the protective effect of being in a group would be lessened.

being greedy in action selection for each point on the graphs. Firstly, several experiments are shown with two evenly matched groups of agents. Later in Section 5.2.2 the experiments are repeated with uneven groups of agents. Again, as before, each curve plotted is the average of four runs and each agent and reward is placed randomly at the start of each episode.

5.2.1 Even Groups

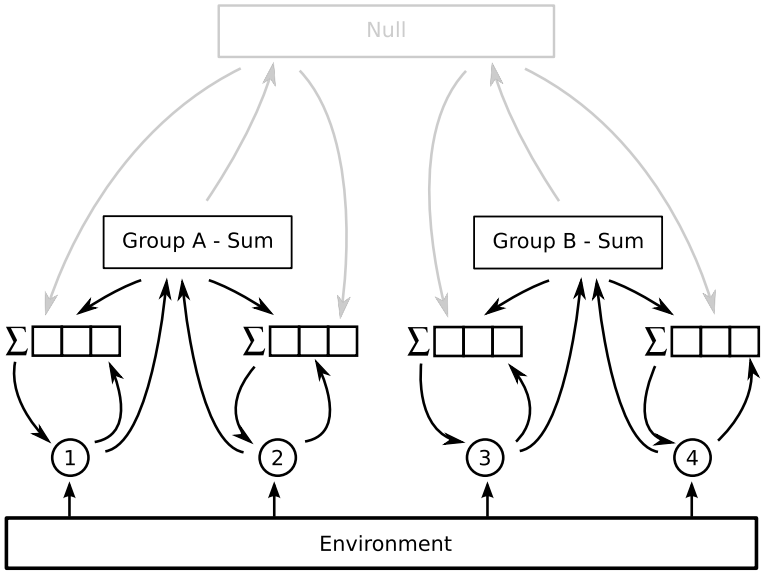


Figure 5.11: The layout of the unconstrained groups (A and B) and group utility functions for each of the curves shown in Figures 5.13(a) and 5.14(a). The null utility function returns zero. i.e it has no effect on the reward signal. It is included in this diagram for comparison with Figure 5.12.

Figures 5.11 and 5.12 show four agents split into two sum group utility functions groups. These two sum group utility functions are under either a further *null* or *balancing* group utility function. Under this set-up the agents should learn to act as teams of two and when under the balancing group utility function these teams should learn to perform as well as each other. These are the first experiments showing hierarchical group utility functions as introduced in Section 3.5.1. Whereas the results above in Section 5.1.1 showed two agents learning under a balancing group utility function these experiments shows two groups learning under a balancing group utility function. The groups themselves have a group utility function and those agents in the groups are now affected by their group's utility function and the group utility function above that.

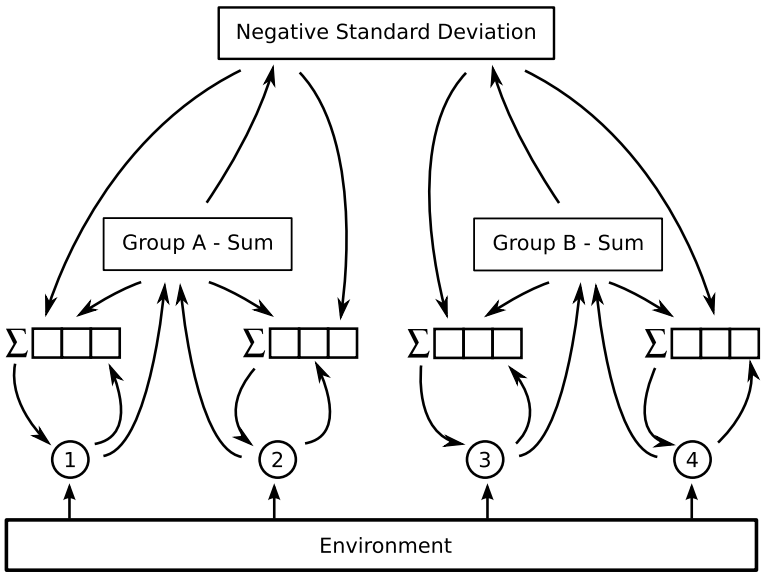
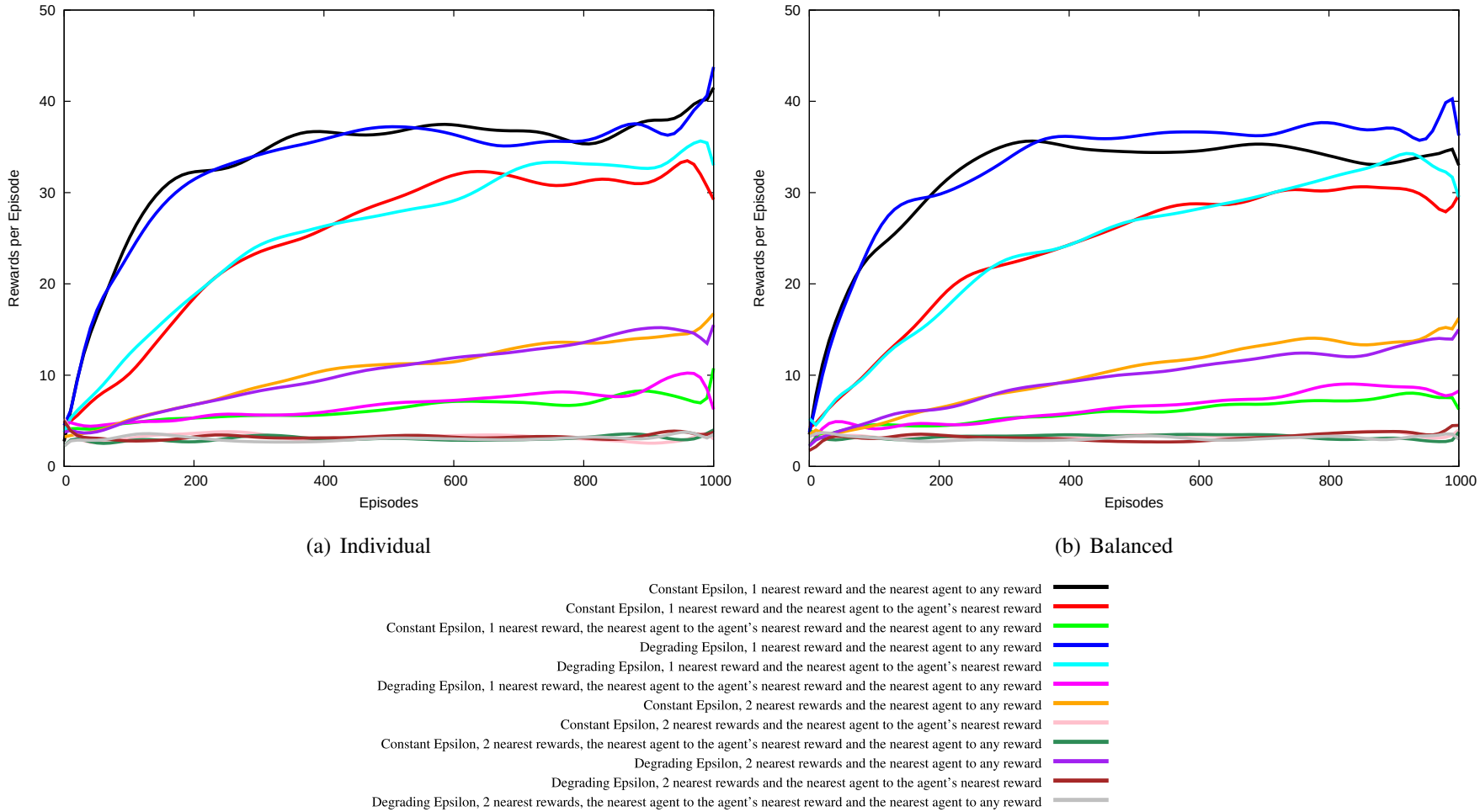


Figure 5.12: The layout of the balanced groups (A and B) and group utility functions for each of the curves shown in Figures 5.13(b) and 5.14(b).

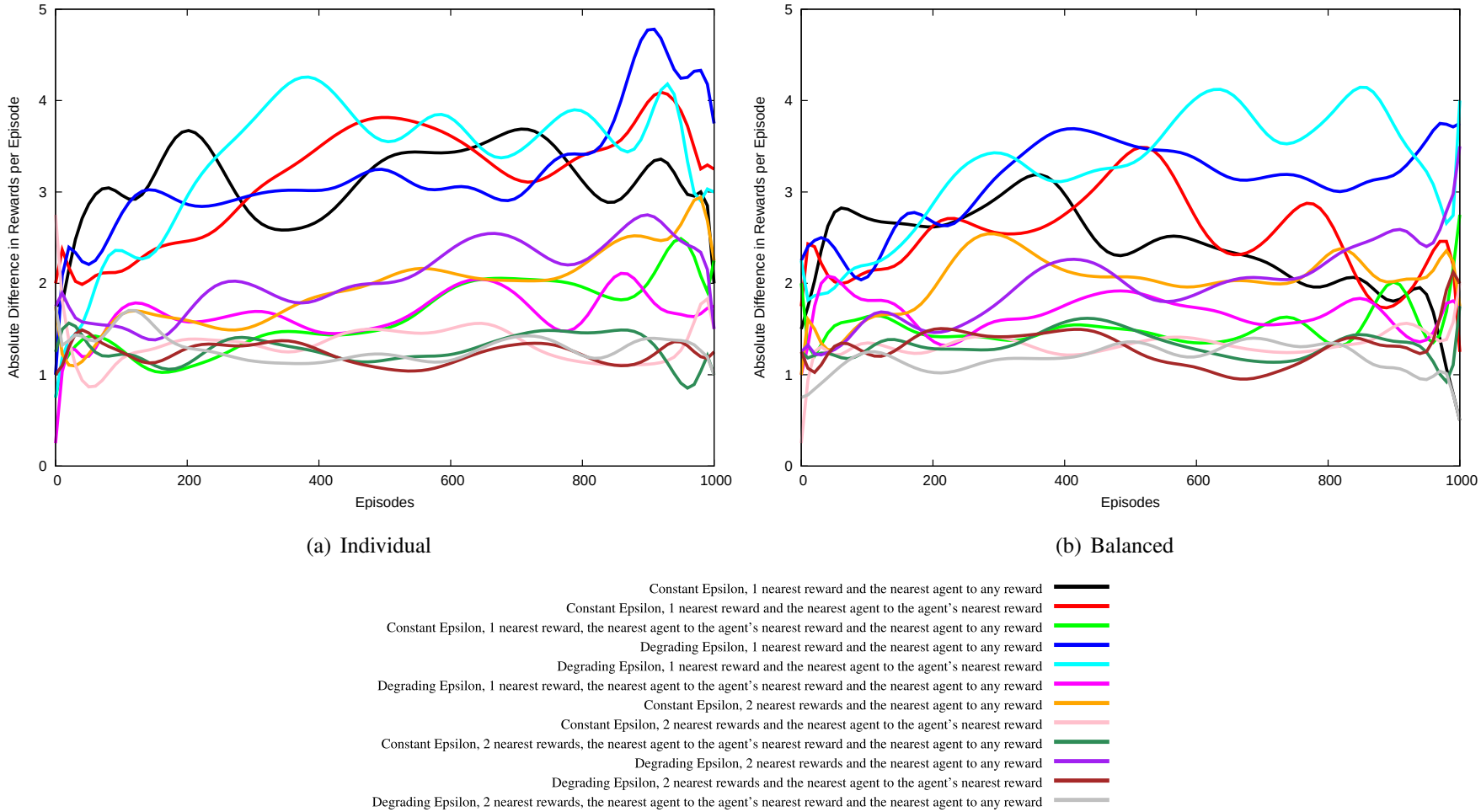
The graphs in Figures 5.13 and 5.14 should look similar to those in Figures 5.9 and 5.10 respectively but with two groups rather than two agents. The full graphs for each of these set-ups can be seen in the appendices from Figure A.11(a) to A.18(f).



(a) Individual

(b) Balanced

Figure 5.13: Performance of two groups of two agents when acting individually and when balanced with all curves with initial $\varepsilon = 1$.



(a) Individual

(b) Balanced

Figure 5.14: Absolute difference in number of rewards picked up by two groups of two agents when acting individually and when balanced with all curves with initial $\varepsilon = 1$.

The curves in Figure 5.13(a) show the joint performance of all four agents as laid out in Figure 5.11. Agents *1* and *2* should learn to behave as a team of agents and agents *3* and *4* should learn to act as another team. These teams are unconstrained under the *null* group utility function. The legend shows that the curves represent the experimental conditions exactly the same as the previous legend in Figure 5.9. The legends are kept the same in all experiments from here on. The curves in Figure 5.13(a) show exactly the same arrangement as in Figure 5.9(a). This reinforces that already said about the effect of the different experimental conditions above. The best pair of curves is as in Section 5.1.1 for constant and degrading $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward in the environment. The two experiments represented by these two curves show asymptotic performance of about forty-three rewards collected in total by all four agents combined. Compared to the equivalent experiments from Figure 5.13(a) that achieved asymptotic performance of approximately twenty-one rewards, the agents in these experiments collected about twice as many rewards. This makes sense as there are twice as many agents collecting rewards. Now to look at the performance of the equivalent experiments but with the two groups of two agents under the balancing group utility function as shown in Figure 5.13(b). These experiments again show the same order of asymptotic performance as those in Figures 5.13(a) and 5.9. The best performing experiments are those for constant and degrading $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward in the environment. The performance between these experiments (a total of thirty-five rewards collectively) and their equivalent experiments in Figure 5.13(a) differs by eight rewards or, 18.6%. This is less performance loss than the performance loss with two agent as above which was 22.5%. One would hypothesise that with four agents instead of two there would be fewer extra steps to maintain balance. That is, more generally, with more resources it is easier to cope with constraints.

Figure 5.14 shows the absolute difference in rewards collected each episode between the two groups of agents. This should be similar to Figure 5.10 but for two groups instead of two agents. The curves in Figure 5.14 again share the same legend as in Figure 5.13. The graphs in Figure 5.14 are difficult to read and one must look closely to see the interesting results. Taking the two best performing experiments from Figure 5.13(a) and looking at Figure 5.14(a) one can surmise that even when unconstrained the groups show little imbalance probably due to the fact that there is natural balance. Especially because difference in performance is now averaged over the two

agents in each team. In Figure 5.14(b) the experiments should produce less difference and therefore the curves should be lower to indicate this. Taking the top performances from Figure 5.13 in descending order:

1. Degrading ε, one nearest reward and information about the nearest other agent to any reward;

2. Constant ε, one nearest reward and information about the nearest other agent to any reward;

3. Degrading ε, one nearest reward and information about the nearest other agent to the agent's nearest reward;

4. Constant ε, one nearest reward and information about the nearest other agent to the agent's nearest reward;

one can see for experiments 1 and 3 that in Figure 5.14(b) the difference in performance is relatively similar to that in the equivalent unconstrained experiments in Figure 5.14(a). i.e. the groups have *not* learned to perform only as well as each other. However, for experiments 2 and 4 the difference in Figure 5.14(a) is 2 and 3.3 respectively and in Figure 5.14(b) their groups differ by only 0.5 and 1.3. This shows that for the experimental conditions marked as 2 and 4 above the groups have learned to perform only as well as each other. So for these set of experiments with two groups of two agents one can see a drop in performance of 18.6% along with the ability to learn the secondary qualitative goal of balance. The experimental set-up, constant ε, one nearest reward and information about the nearest other agent to any reward, shows the least degradation in number of rewards collected and the least absolute difference between the groups.

Next, similar experiments with two groups of three agents are shown. These should show similar performances to those shown in Figures 5.13 and 5.14.

Figures 5.15 and 5.16 show the hierarchy of agents and group utility functions for the next set of experiments where two groups of three agents again learn to forage for rewards when unconstrained and when constrained by a balancing group utility function.

Figures 5.17 and 5.18 show the number of rewards picked up by the six agents agents collectively and the absolute difference in number of rewards picked up by each group for the various experimental conditions already described and shown above in Figures 5.9, 5.10, 5.13 and 5.14. Again, the plotted curves in Figure 5.17 should show

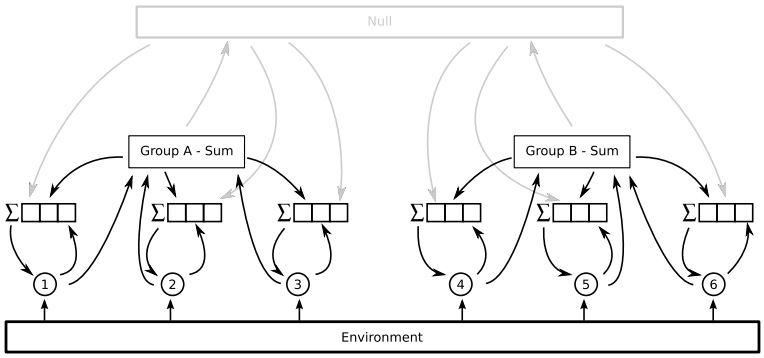


Figure 5.15: The layout of the unconstrained groups (A and B) and group utility functions for each of the curves shown in Figures 5.17(a) and 5.18(a). The null utility function returns zero. i.e it has no effect on the reward signal. It is included in this diagram for comparison with Figure 5.16.

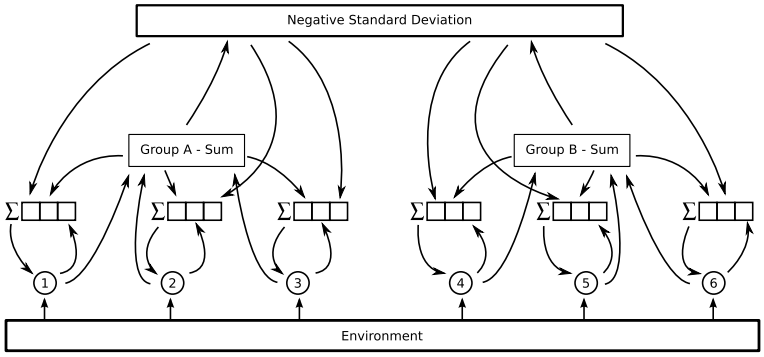
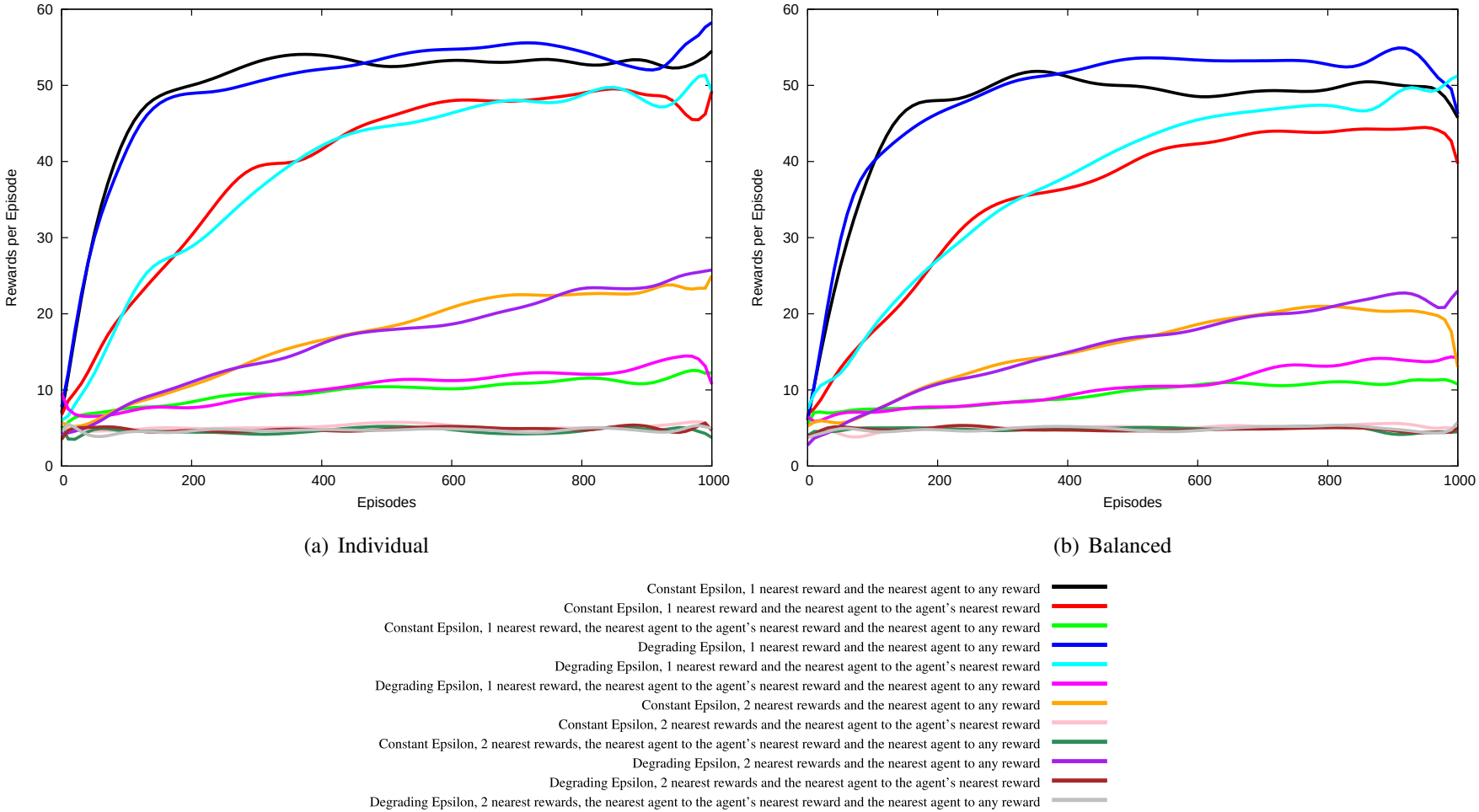


Figure 5.16: The layout of the balanced groups (A and B) and group utility functions for each of the curves shown in Figures 5.17(b) and 5.18(b).

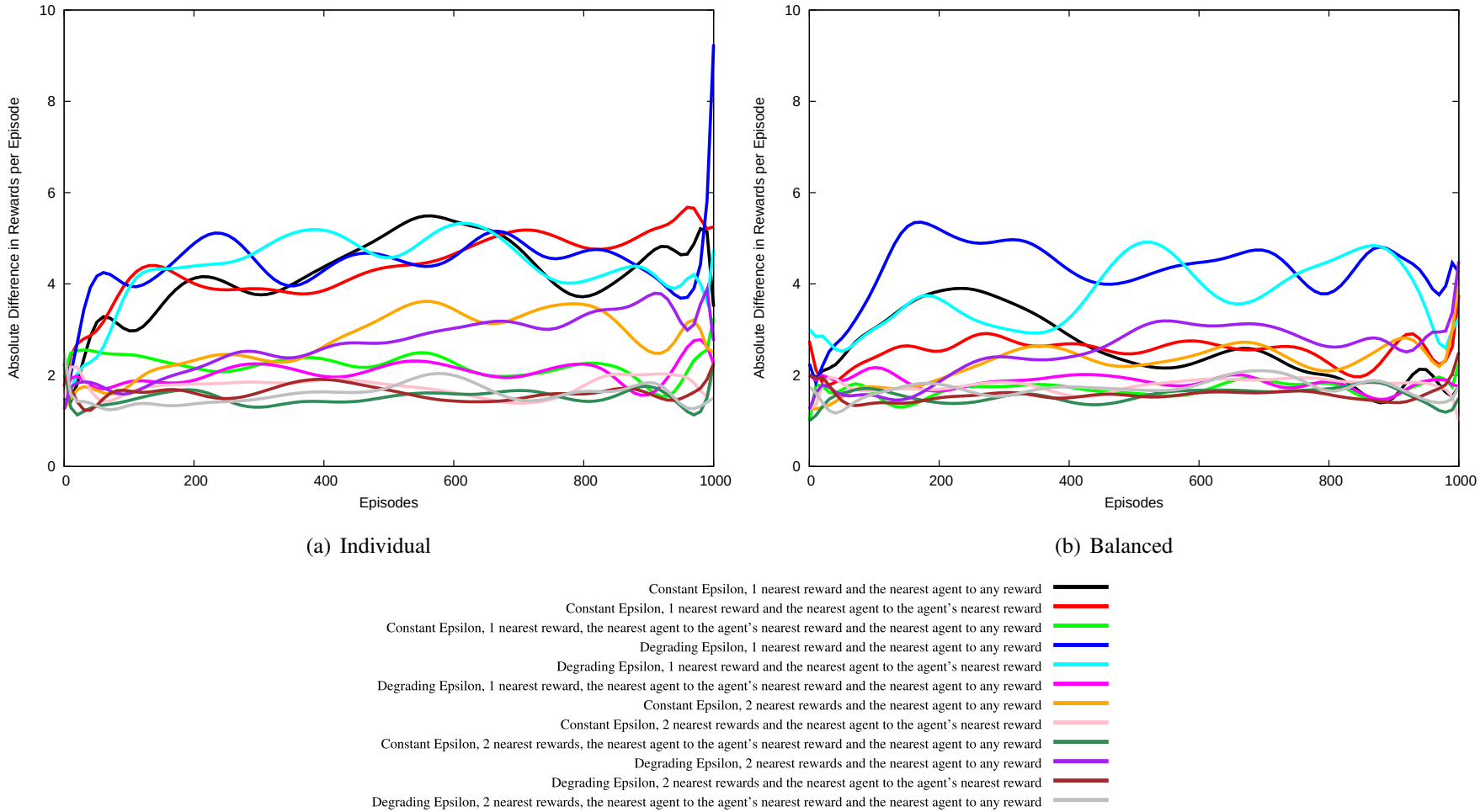
high values of rewards collected and their equivalent curves in Figure 5.18 should show the low values of absolute difference in rewards collected. The full graphs for each of these set-ups can be seen in the appendices from Figure A.19(a) to A.26(f).



(a) Individual

(b) Balanced

Figure 5.17: Performance of two groups of three agents when acting individually and when balanced with all curves with initial $\varepsilon = 1$.



(a) Individual

(b) Balanced

Figure 5.18: Absolute difference in number of rewards picked up by two groups of three agents when acting individually and when balanced with all curves with initial $\varepsilon = 1$.

Figures 5.17(a) and 5.17(b) show the similar pattern seen before in previous experiments. The only slight difference is that degrading $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest other agent to that reward is the best performing experimental set-up which it has not been in the previous experiments. Taking the curve representing the experimental set-up for constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward in the environment as a reference because it has performed well previous, here it sows asymptotic performance of fifty-five rewards per episode. Compared with the asymptotic performance of this curve in Figure 5.13(a) (two groups of two agents) the two groups of three agents each pick up 34% more rewards (fifty-five rewards compared to forty-two rewards). This seems appropriate as there are a third more agents in this experiment showing that the results when unconstrained are scaling in-line with resources. The plotted curve for degrading $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest other agent that reward in Figure 5.17(b) shows a decrease in rewards collected per episode of nine (fifty-five rewards down to forty-six). That is a 16.4% drop in collective rewards per episode compared to 18.6% in Figure 5.13 (for two groups of two agents each) and 22.5% in Figure 5.9 (for two agents). This shows a trend for increased efficiency for picking up rewards when under the balancing group utility function with more agents. In other words it is easier to maintain performance measured against the main quantitative goal whilst learning to satisfy the secondary qualitative goal with more agents than with less.

Figures 5.18(a) and 5.18(b) allow us to compare the absolute difference in rewards picked by the two groups shown in Figures 5.15 and 5.16. For each of the four top performing experimental set-ups shown in Figure 5.17 the absolute difference's in rewards picked up in Figure 5.18(b) when constrained by the balancing group utility function are all significantly lower than in Figure 5.18(a) when unconstrained. For example for the oft well performing experimental set-up of constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward; when unconstrained the absolute difference at asymptotic performance is 3.5 rewards and, when constrained by the balancing group utility function is 1.7 rewards.

As usual for this chapter, for all other experimental set-ups the performance in Figure 5.17 is indicative of little or no learning at all having happened. This is because of the curse-of-dimensionality and has been consistent throughout the experiments shown here with even numbers of agents and/ or groups.

The two sets of experiments shown above in Figures 5.13, 5.14, 5.17 and 5.18 with even numbers of agents in two groups have shown empirically that the idea of group utility functions works when applied to groups of agents, albeit small groups of agents. The reader has seen that when unconstrained the total number of rewards has increased in-line with the number of agents in the environment and furthermore the several experimental set-ups have displayed the same levels of performance as shown in Figures 5.9 and 5.10 for just two agents indicating a sound experimental set-up with little variance in expected results. Other than the fact that hierarchical group utility functions have been shown to produce useful results, it is interesting to see that the "burden" of the required qualitative behaviour (balance) is less noticeable when there are more agents to shoulder the task. Next, similar experiments are shown but looking at uneven numbers of agents within hierarchical group utility functions.

5.2.2 Uneven Groups

The first of the two experiments in this section show the same selection of experimental conditions when the same foraging experiments are run but with a group of two agents under sum group utility function and a single agent both under the null group utility function (i.e. unconstrained, Figure 5.19) or under the balancing group utility function (i.e. the negative standard deviation of the rewards collected by the group and the single agent, Figure 5.20). The experiments in this section use differing group affiliation values to compensate for the mismatch in group size. The group affiliation for the single agent to the top group utility function remains as *one* and the group affiliation values between agents *1* and *2* and the sum group utility function remain at *one* also. The group affiliation value between the sum group utility function and the top group utility function is set at *two* for this upcoming set of experiments and *three* for the experiments shown later with a group of three agents.

As in the previous section, Figure 5.21 shows the total number of rewards collected by all three agents for each episode. The experimental conditions were varied in the same way as before and the legend maintains the same colours so as to ease comparison. The plotted curves in Figure 5.21 should be high to indicate that the agents have learned to pick up a large number of rewards. Figure 5.21(a) shows asymptotic performance of thirty rewards per episode for constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward. This is appropriately, approximately half-way between the total number of rewards collected

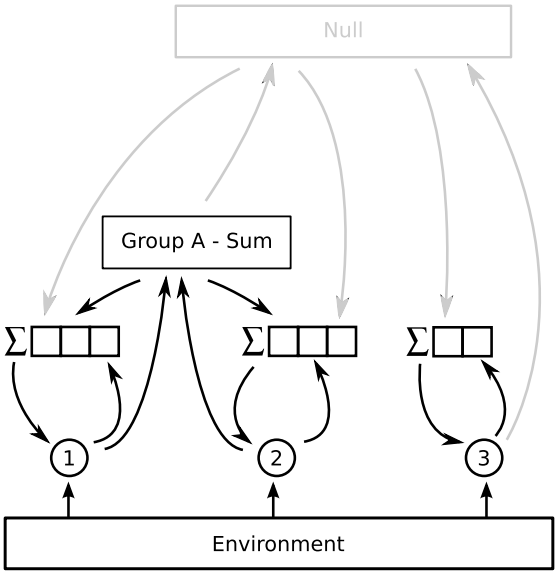


Figure 5.19: The layout of the group of two agents (A), individual agent and group utility functions for each of the curves shown in Figures 5.21(a) and 5.22(a). The null utility function returns zero. i.e it has no effect on the reward signal. It is included in this diagram for comparison with Figure 5.20.

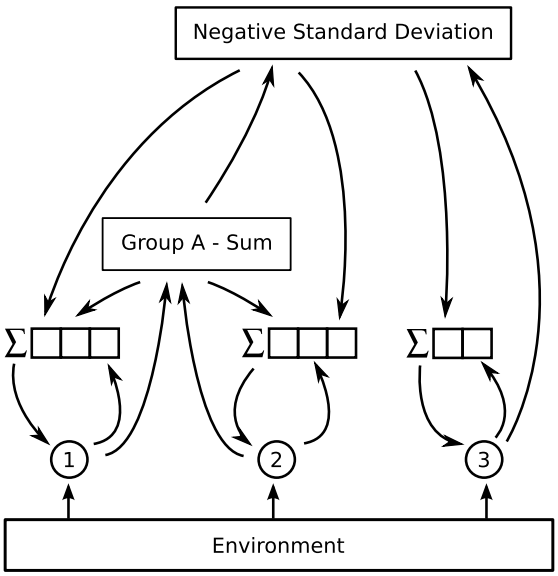
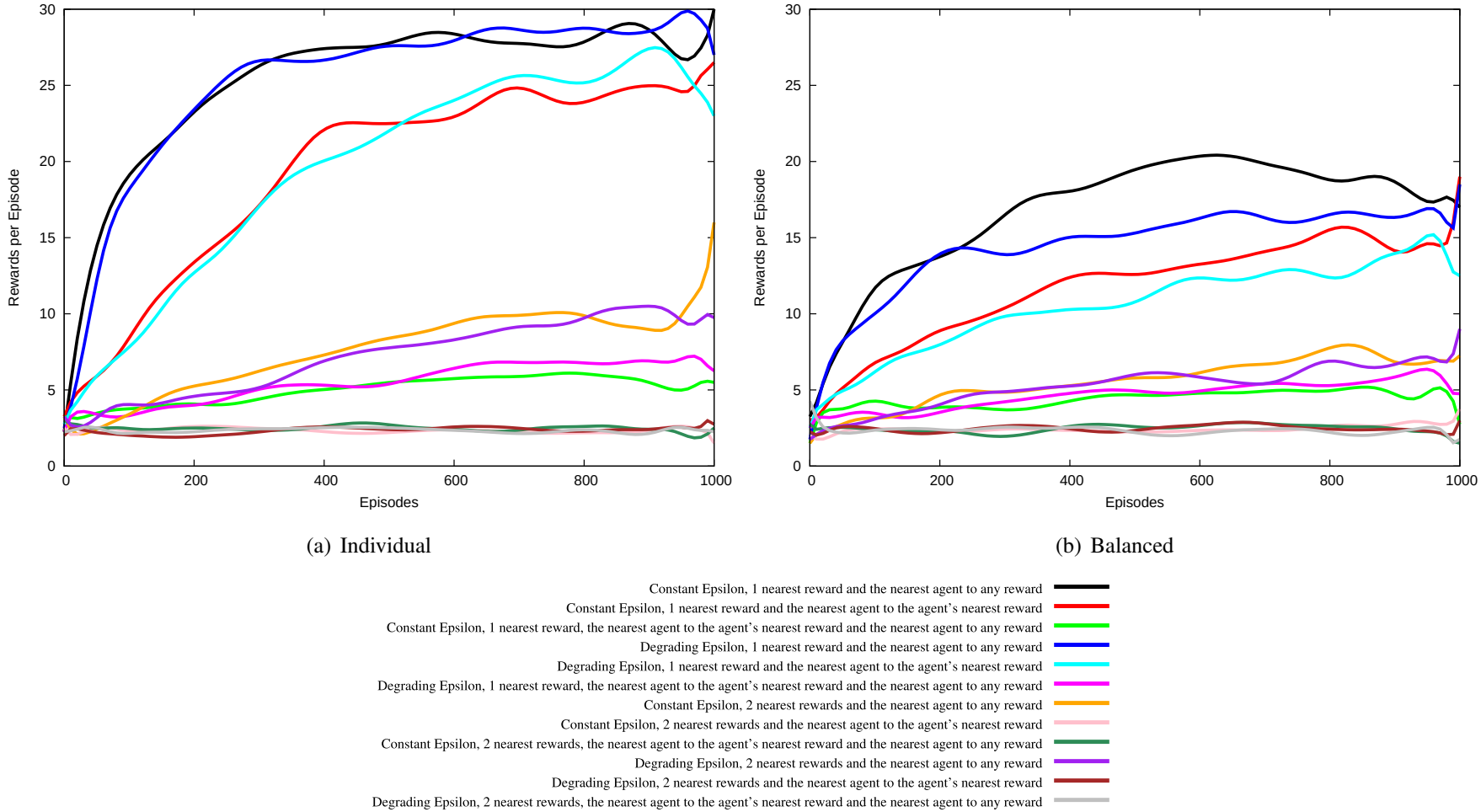


Figure 5.20: The layout of the group of two agents (A), individual agent and group utility functions for each of the curves shown in Figures 5.21(b) and 5.22(b).

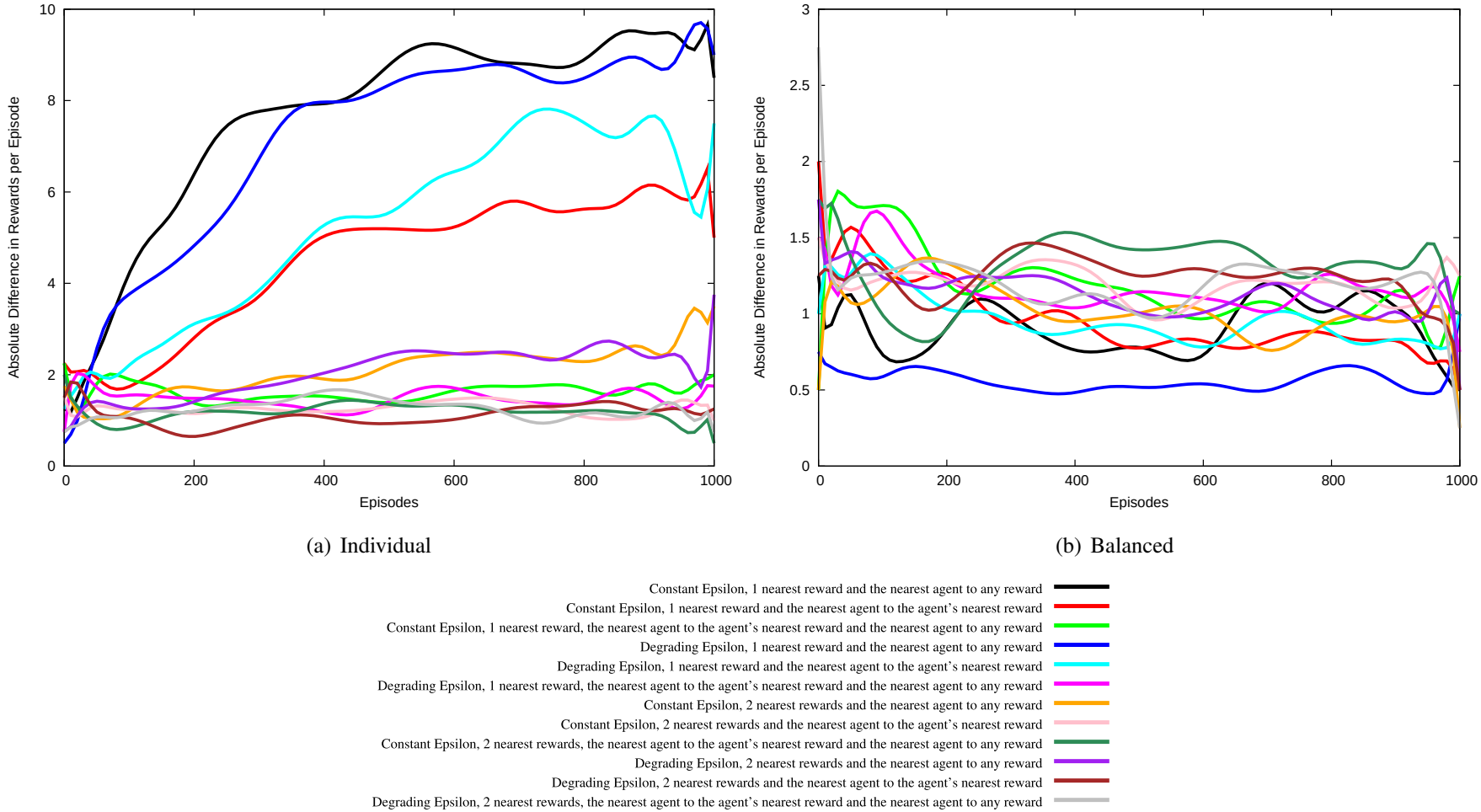
for the two agents in Figure 5.9(a) (twenty-one rewards) and for the four agents in Figure 5.13(a) (forty-three rewards). Otherwise, the plotted curves for each experimental variation show the same order of performance as before in the previous experiments of this chapter. The full graphs for each of these set-ups can be seen in the appendices from Figure A.27(a) to A.34(f).



(a) Individual

(b) Balanced

Figure 5.21: Performance of a group of two agents and an individual agent when both acting individually and when balanced with all curves with initial $\varepsilon = 1$.



(a) Individual

(b) Balanced

Figure 5.22: Absolute difference in number of rewards picked up by a group of two agents and an individual agent when both acting individually and when balanced with all curves with initial $\varepsilon = 1$.

Figure 5.21(b) shows a significant drop in collective rewards picked up each episode when compared to the equivalent curves of Figure 5.21(a). This makes intuitive sense. Unlike previous experiments where the number of agents on each side of the balancing function are equal here the group of two agents must perform only as well as the single agent. Thus one should expect the number of rewards being picked up per episode when learning under the balancing group utility function at asymptotic performance to be near eighteen rewards - that shown in Figure 5.9(b) for two balanced agents. In fact for constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward the number of rewards picked up collectively per episode in Figure 5.21(b) is 17.5. That is a drop in performance of 12.5 rewards, or 41.7% compared to the equivalent experiment shown in Figure 5.21(a).

For Figure 5.22 the scale is not kept the same as the two graphs vary so greatly in the absolute difference in rewards collected by the group and single agent. Figure 5.22(a) shows the now expected plot of the various experimental conditions. When looking at the top four best performing experiments in Figure 5.21 one can see that for each of these experiments the imbalance is significantly lower when constrained by the balancing group utility function in Figure 5.22(b) than when left unconstrained as shown in Figure 5.22(a). Taking the experiments for constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward as a reference, the unconstrained, naturally occurring absolute difference in rewards picked up between the group of two agents and the single agent is 8.5 rewards and when learning a balanced performance is 0.5 rewards.

The final set of experiments in this chapter shows three agents in a group set against one single agent. Along with the above experiments this should show the effects of inherently unbalanced resources with hierarchical group utility functions. For these last experiments the hierarchy of agents, groups and group utility functions is shown in Figures 5.23 (unconstrained) and 5.24 (balanced).

The full graphs for each of experimental conditions can be seen in the appendices from Figure A.35(a) to A.42(f). Figure 5.25 shows each interesting set of experimental conditions as before and, as before, the same ordering of experiments is seen with the same pair clustering. The best performing experiment, when the group and agent were unconstrained, as shown in Figure 5.25(a), is that for constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward, which achieves asymptotic performance of thirty-seven collective rewards.

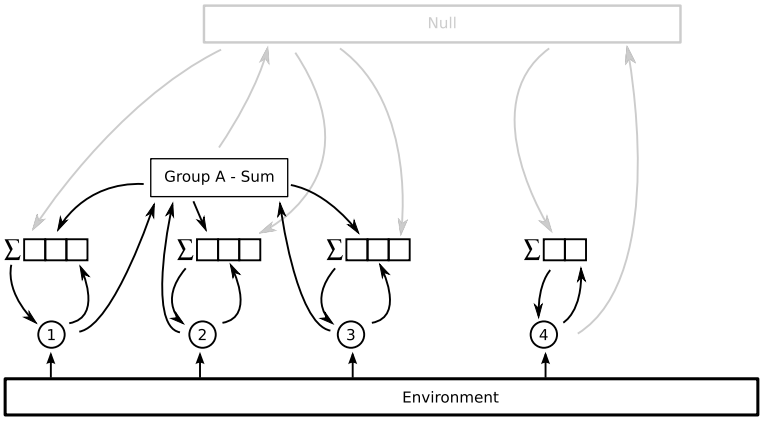


Figure 5.23: The layout of the group of three agents (A), individual agent and group utility functions for each of the curves shown in Figures 5.25(a) and 5.26(a). The null utility function returns zero. i.e it has no effect on the reward signal. It is included in this diagram for comparison with Figure 5.24.

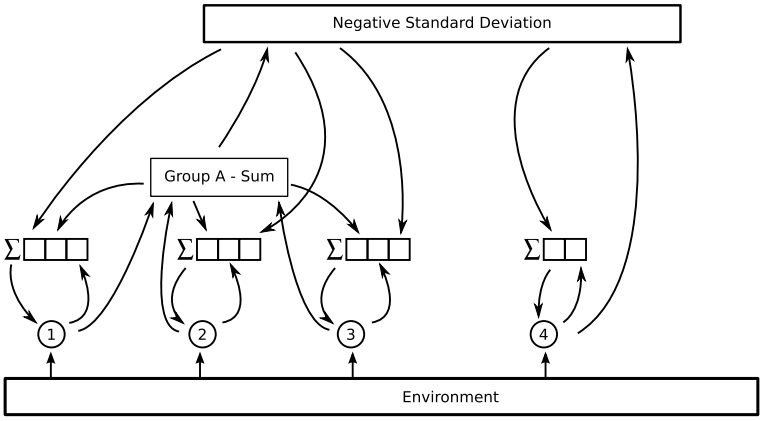
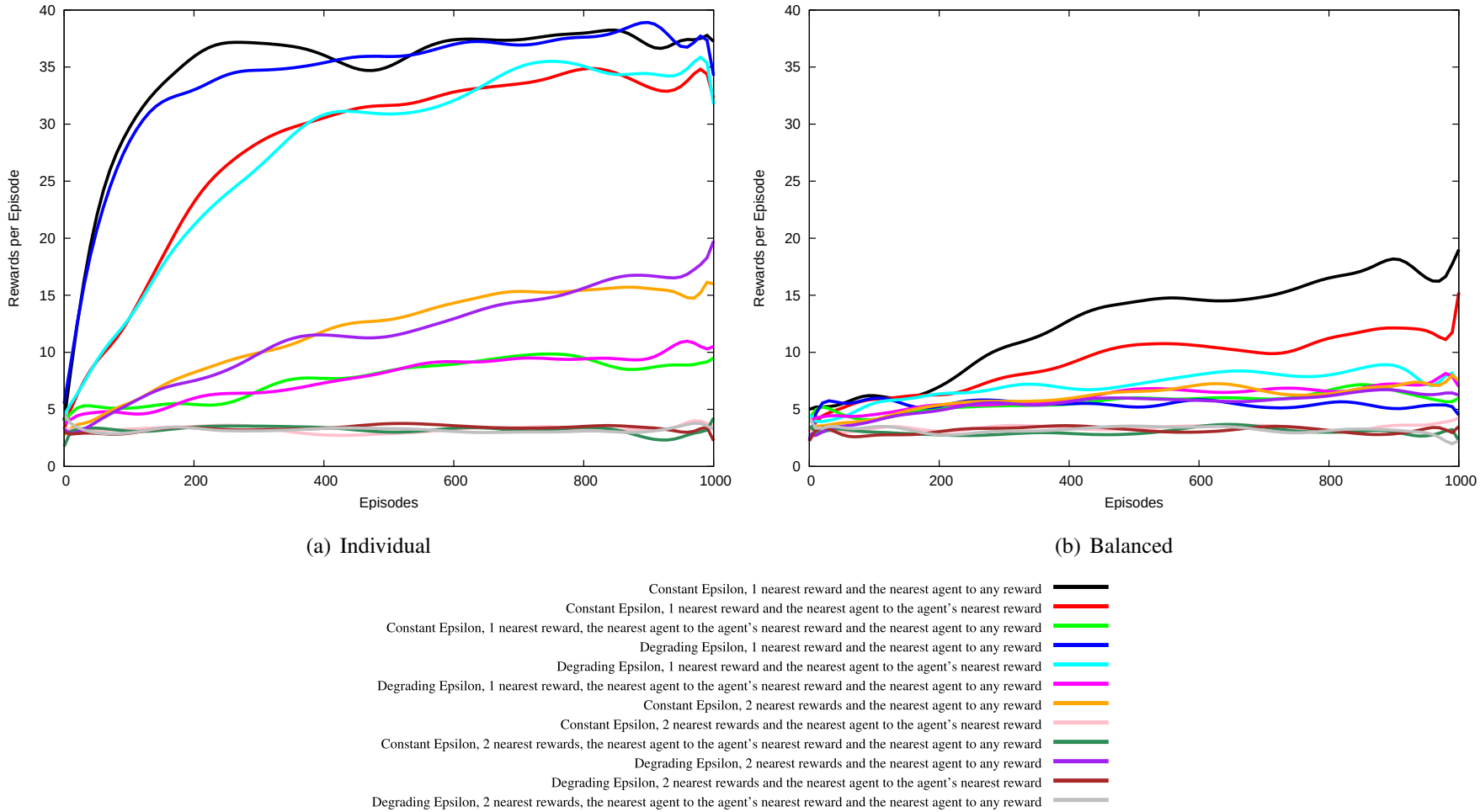


Figure 5.24: The layout of the group of three agents (A), individual agent and group utility functions for each of the curves shown in Figures 5.25(b) and 5.26(b).

This puts its performance in-line with the equivalent experiment with four agents of Figure 5.14(a) that collected forty-three rewards. All of the experiments plotted in Figure 5.25(b) show greatly reduced performance when compared to Figure 5.25(a). Appropriately so, as the three agents in the group are now learning to pick up only as many rewards as the single agent and so the reduction in that total number of rewards collected should be sizeable. Again as in the previous set of experiments the balanced group and agent should pick up around eighteen rewards (the best asymptotic performance in Figure 5.9(b) when experimenting with just two agents). In this case, for constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward the group and single agent picked up a combined nineteen rewards per episode, a reduction in performance of 51.4%.

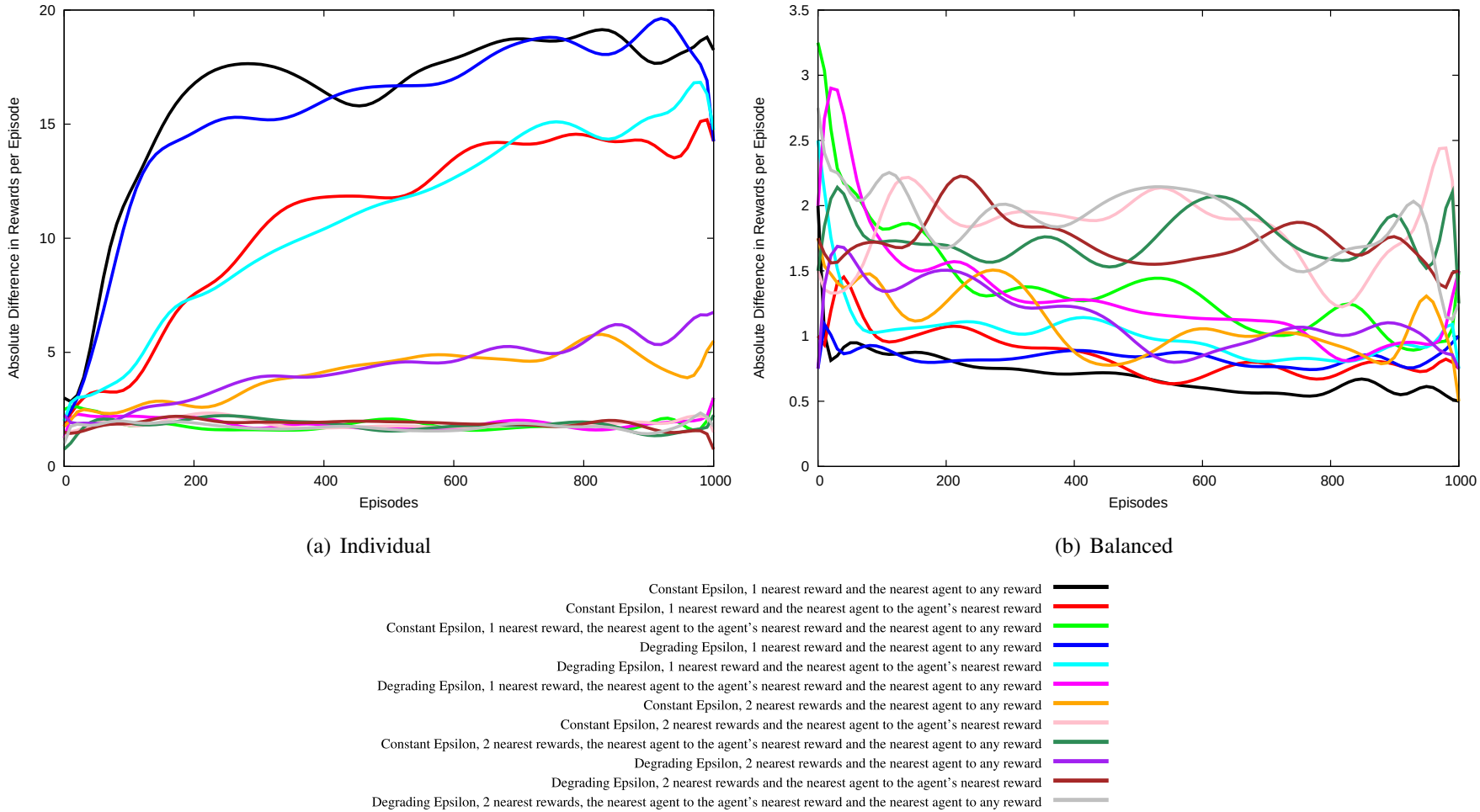
Alongside this when looking at the absolute difference between the rewards collected per episode by the group of three agents and the single agent one as shown in Figure 5.26 (note the vertical scale is different for each graph) one can see that when under the balancing group utility function the agents and group of agents have learned successfully to perform as well as each other. To take the case for constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one nearest agent to any reward, when unconstrained the imbalance is eighteen rewards whereas when learning to balance their performance the imbalance between the group and agent is just half a reward.



(a) Individual

(b) Balanced

Figure 5.25: Performance of a group of three agents and an individual agent when acting individually and when balanced with all curves with initial $\varepsilon = 1$.



(a) Individual

(b) Balanced

Figure 5.26: Absolute difference in number of rewards picked up by a group of three agents and an individual agent when acting individually and when balanced with all curves with initial $\varepsilon = 1$.

5.3 Summary

This chapter presented graphs and discussion of a number of experiments for two agents, two groups of two agents, two groups of three agents, a group of two agents along with a single agent and a group of three agents alongside a single agent under several experimental conditions. The first section (Section 5.1.1) showed that group utility functions can be applied to computer game characters to induce a simple equilibrium between two learning agents. Several interesting experiments were shown plotted and the complete set of graphs are included in the appendices. The effect of larger state spaces and choice of state representation was shown and discussed. The curves were all taken for $\varepsilon = 1$ whether with a constant or non-constant (degrading ε) policy. The effect of lowering the value of ε can be seen by looking at the appropriate graphs in the appendices but there was little of interest to be seen in those other plotted curves. In these foraging experiments the sequence of actions needed to reach a goal is short and the main benefit for the agents is in exploring the state space as they improve through learning. Thus the best policy is to explore as much as possible. Without this exploration throughout the experiments the agents do not get to experience drawing, losing and winning against the other group or agent whilst covering various other parts of the state space. It is later in the experiments when the agents have learned their basic quantitative goal of learning to pick up rewards that they still need to take enough exploratory actions to determine the best way to maximise the qualitative part of their reward function.

In Section 5.2 the results of using hierarchies of group utility functions with groups of agents and single agents were shown. First with evenly matched groups of agents in Section 5.2.1 and then with inherently imbalanced hierarchies with a single agent and a group or either two or three agents in Section 5.2.2. The performance of the agents largely matched those in the first set of experiments which showed that the experiments were soundly composed, scaled as expected and gave a good background for comparing the effect of hierarchical group utility functions. There were no surprises in the results of Sections 5.2.1 and 5.2.2 with the groups of agents and single agents performing much the same as the first experiments with two single agents. The experiments combined offer a body of empirical work showing reinforcement learning used for computer game characters in a simple but real commercial computer game environment. It was interesting to see that it was more efficient to learn the qualitative balancing part of the reward signal when there were more agents with which to

distribute the extra load over. It would be interesting to see the results for large scale experiments of this sort. Could a qualitative behaviour such as balancing be learned and maintained over hundreds of agents with little effect at the individual agent behaviour? This would be useful for the war game scenario used in Chapter 3. How would the balancing group utility function affect the behaviours of the twenty-two football players in computer football games where the ability to learn a relative performance would be desirable? It was also interesting to see in Section 5.2.2 that the group utility functions could still learn and maintain the qualitative goal of balanced performances when the group of agents had a clear advantage over the single agent. This would be ideal for training "cannon fodder" type non-player characters that only perform to a certain level set by existing characters or possibly even recorded human play.

The idea of group utility functions is useful to induce a relationship between agents whilst they optimise a quantitative goal and the algorithm described in Chapter 3, implemented and whose results are shown here works well. The algorithm allows a certain qualitative relationship to be learned at the same time as learning to maximise a quantitative goal and there is a trade off associated with the qualitative part of the reward function. Throughout all the experiments in this chapter there were several experimental conditions that always failed to perform. This was due to the large combined state-action space. This severely limits the algorithm to simple tasks as shown and is a well understood problem with reinforcement learning and other search based optimisation methods as said in Chapter 2. The obvious way to combat a large state-action space is to use some form of function approximation (Section 2.2.4.7) instead of the table based approach used here in this chapter. Using function approximation to represent the agents' value functions it should be possible to alleviate the problems of state-space explosion and potentially use reinforcement learning and specifically hierarchical group utility functions in real computer games. To see how hierarchical group utility functions would respond to approximating the value function the experiments run for this chapter (shown here and in the appendices) were repeated with a powerful form of non-linear function approximation in Chapter 6.

Chapter 6

Scaling Towards Real-World Games

Approximating the value function is the obvious way to alleviate the scaling issues seen in Chapter 5. All of the experiments seen so far in Chapters 4 and 5 are repeated here using non-linear neural networks to approximate the value function of each agent.

6.1 Approximating the Value Function

Using a table to represent a value function means storing a potentially massive number of values. For the experiments in this thesis some of the value function tables have had, on inspection at the end of the experiment, up to thirty thousand entries, with each row containing twenty-five values each. That is about 750,000 values to represent an agent's value function. For function approximation, as used below around 525 values are needed to represent an agent's value function for a similar experiment. That is a saving of 99.07% for this case[1].

The space savings are not the primary gains of approximating the value function though. Function approximation can also generalise experience over neighbouring states making more efficient use of the reward signal. Updating the approximated value function through back-propagation of value errors for one sample state-action pair alters the utility values for nearby state-action pairs also. This is how approximation speeds learning and often provides good approximated utility values even for never before visited state-action pairs. Of course, the approximation method needs to

[1]These values are theoretically derived for typical runs of the experiments in this thesis that used the largest state-action representation for the longest number of episodes. These experiments are shown in Figure 4.8(a) for the tabular value function case and Figure 6.7(a) for when using value function approximation. The curves for three rewards in the state representation when the experiment lasted ten thousand episodes were used. Actual space savings are shown in Section 4.4. The use of hash tables significantly reduced the storage required for tabular value functions in practice.

be capable of representing the true value function to an acceptable level of accuracy too. The method should produce neither an under or over-fitted function as in Figure 6.1. The function approximation itself can be difficult to get right and a simpler solution should be sought before experimenting with approximation [Sutton, 2001]. The idea that group utility functions can be used with the negative standard deviation combining function to produce evenly balanced performance was shown to work in Chapter 5. Approximating the value function allows each agent to learn a satisfactory policy (even in those experiments that failed due to too large a state-action space in Chapter 5) as shown in this chapter.

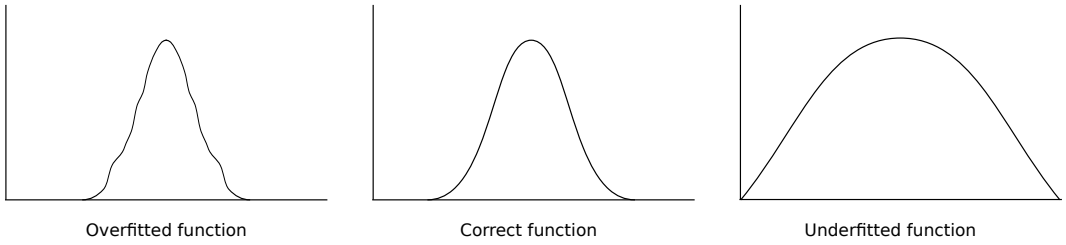


Figure 6.1: A function and over and under fitted approximations of it.

There are several common methods of approximating value functions for reinforcement learning. They can be broadly categorised as linear or non-linear methods, that is, linear (or not) in respect to the input values from the agent's state-action vector. Non-linear neural networks are used in this thesis as they can approximate complex underlying functions, in particular, those parts of a value function that rely on interactions between features [Sutton and Barto, 1998]. For example, if the presence of a feature is good only when another feature is absent. When using linear approximation methods these interactions cannot be represented in the function and so they must be predicted and features made from their combinations must be put into their state-action representation.

Theoretical guarantees of convergence exist for linear methods with gradient-descent TD algorithms[2] that do not exist for non-linear methods. However, in practice the experiments using non-linear function approximation in this section mostly converge to a good policy which performs similarly or better to those experiments in Section 5 using tabular value functions. The main benefit of using non-linear approximation is not having to foresee complex interactions between the state/ action features

[2]Temporal difference algorithms such as the sarsa learning algorithm used in this thesis. See Section 2.2.4 for an explanation of TD methods.

as may arise when using a modified reward signal based on more than one learning objective as is the case in this thesis. Due to the nature of group utility functions it is difficult to know *a priori* the feature dependencies. The risk is that agents' policies do not converge to be near optimal but as shown in this chapter, for most action selection policies the value function does converge to give good observed performance. These experiments provide empirical evidence of non-linear function approximation used successfully with TD methods of reinforcement learning.

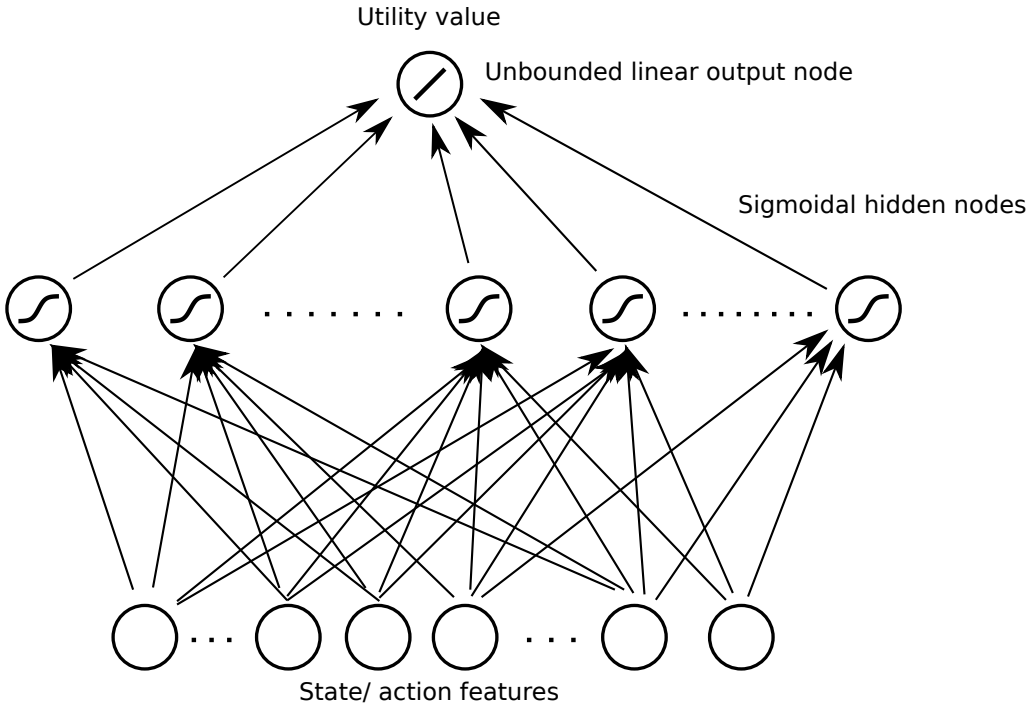


Figure 6.2: The topology of the neural networks used to approximate value functions in this chapter.

The neural networks used in this chapter look like that shown in Figure 6.2. Each feature of an agent's state together with the binary action encoding makes up the input nodes. These values are the same as those that make up the columns in the tabular value functions used in Chapter 5. Each arrow in Figure 6.2 represents a real valued scalar and it is these *weights* that are adjusted by back-propagation of error values to affect the given utility values of the function, just as the entries in the value function table were updated in the algorithm used in Chapters 4 and 5 described in Chapter 4. The values presented by the hidden nodes are calculated as in Equation 6.1, where ς is a sigmoidal function shown in Equation 6.2; h_j is the output value of the j'th hidden node; w_{ij} is the weight for hidden node j and input feature i; w_j^{bias} is the *bias* weight

for hidden node j; n is the number of input values; and f_i is the value for feature i in the state-action representation. The sigmoidal function shown in Equation 6.2 is used as the activation function for each hidden node and limits the hidden node output values to between zero and one. The sigmoid function provides a good differential for back-propagation of errors. The *bias* node always has a value of *one* and so is independent of any input feature. The bias allows a node's output to be scaled independent of the input values. The output node is calculated in a similar way to Equation 6.1 as shown in Equation 6.3. In Equation 6.3 v is the output value; m is the number of hidden nodes; $w_{j,output}$ is the weight between hidden node j and the single output node; and w^{bias}_{output} is the *bias* weight for output node. The difference is that the output is unbounded by any activation function so that the function can represent any real value.

$$h_j = \varsigma\left[\left(\sum_{i=0}^{n} w_{ij} \times f_i\right) + w_j^{bias}\right] \tag{6.1}$$

$$\varsigma(a) = \frac{1}{1+e^{-a}} \tag{6.2}$$

$$v = \left(\sum_{j=0}^{m} w_{j,output} \times h_j\right) + w^{bias}_{output} \tag{6.3}$$

Too many hidden nodes may lead to over-fitting and too few may lead to over-generalisation but these rules-of-thumb depend greatly on many other factors including the complexity of the function being approximated and the number of training samples [Sarle, 2009]. In these experiments $m = 20$ hidden nodes were used and, in testing, the results appeared quite independent of the number of hidden nodes.

6.1.1 The Learning Algorithm

The learning algorithm is sarsa(λ) (gradient descent is performed by back-propagating the error through the neural network nodes) used with a non-linear neural network for value function approximation (adapted from [Sutton and Barto, 1998]).

- $\forall$ agents
  - Place agent randomly
  - Set α, γ, *lambda* and initial ε
  - $\forall\, i, j \quad w_{ij} = random[-0.005...0.005]$

  - $\forall\ j \quad w_{j,output} = random[-0.005...0.005]$

- Place rewards randomly.

- $\forall$ time-steps $t_0^{\text{number of episodes} \times (100 \times \text{number of agents})}$ (where there are 100 action steps for each agent in each experiment)

  - For each agent (Chosen in turn by the computer game engine. Essentially, whichever agent is ready. This is normally in the same repeating order e.g. 1, 2, 3, 1, 2, ...). All the variables below are with respect to each separate agent.

    * Take action a_t from state s_t, observe modified reward r'_t (from Equation 3.9) and state s_{t+1} (Replace any rewards picked up at a new random position)
    * $a_{t+1} \leftarrow \pi_t$ using ε-greedy selection
    * $\delta = \begin{cases} r'_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) & \text{when } s_{t+1} \text{ is not a reward state} \\ r'_t - Q(s_t, a_t) & \text{when } s_{t+1} \text{ is a reward state} \end{cases}$
    * $\forall\ j$ (including bias) $\begin{aligned} e_j &= (\gamma \times \lambda \times e_j) + (o_j \times \delta) \\ \delta_j &= o_j\prime \times w_{j,output} \times \delta \end{aligned}$
    * $\forall\ i, j$ (including bias) $e_i = (\gamma \times \lambda \times e_i) + (f_i \times \delta_j)$
    * $\forall\ j$ (including bias) $w_{j,output} = w_{j,output} + (\alpha \times e_j)$
    * $\forall\ i, j$ (including bias) $w_{ij} = w_{ij} + (\alpha \times e_i)$
    * if degrading ε = initial $\varepsilon \times \left[\frac{\text{number of episodes to go}}{\text{number of episodes}}\right]$

where: the learning rate, $\alpha = 0.1$; the discount rate, $\gamma = 0.9$; r'_t modified reward at time t; o_j represents the output of node j; δ represents the TD error at the output node; δ_j represents the TD error at hidden node j; i and j represent nodes in the input and hidden layers (respectively); f_i represents the value of feature i; w_{ij} is the weight between node i and j; e_i is the eligibility of node i and $o_j\prime$ is the derivative of the output of node j with respect to its activation function (shown in Equation 6.2). ε-greedy action selection was used with an initial ε of 1 (i.e fully random), 0.8, 0.6, 0.4, 0.2 and 0 (i.e. fully exploitative). As in Chapter 5 in half the experiments ε was degraded linearly from its initial value at the first episode to zero at the end of the experiment (as shown in the learning algorithm above). A description of this algorithm using tabular

value functions was written in Section 4.2.4. The use of approximated value functions is the only difference between the two learning algorithms. The common experimental parameters are summarised in Table 5.1.

6.2 Experimentation

The experiments in Chapter 5 are repeated here in this Chapter with value function approximation. The experiments used the same hierarchies of agents, groups and group utility functions:

- Two **individual** agents (as in Figure 5.1);
- Two **balanced** agents (as in Figure 5.2);
- Two **individual** groups of **two agents each**, with each group acting as a **team** (i.e. the four agents are split into two groups. Each group of two agents is under a sum group utility function and the two groups themselves, are under a null group utility function. A *team* here, refers to a group with a group utility function that uses the sum combining function - the same as the team utility function.);
- Two **balanced** groups of **two agents each**, with each group acting as a **team**;
- Two **individual** groups of **three agents each**, with each group acting as a **team**;
- Two **balanced** groups of **three agents each**, with each group acting as a **team**;
- One group of **two agents** acting as a **team** and, **one agent** acting **individually**;
- One group of **two agents** acting as a **team**, **balanced** with **one agent**;
- One group of **three agents** acting as a **team** and, **one agent** acting **individually**;
- One group of **three agents** acting as a **team**, **balanced** with **one agent**.

The experimental conditions used were the same as in the last chapter with several different state representations used for each hierarchy of agents, groups and group utility functions:

- The distance to the nearest reward in the state; the distance of the nearest agent to **any** reward, whether that agent is in the same team or not[3] and if that agent

[3]This is irrelevant for experiments with just two agents as there is only one other agent but is kept in the state representation to maintain comparable state-action spaces across experimental set-ups.

or agent's group (if in a group other than that of the agent whose state is being described) has collected more reward, less reward or the same amount of reward during the current episode;

- The distance to the nearest reward in the state; the **distance of the nearest other agent to that nearest reward**, whether that agent is in the same team or not and, if that agent or agent's group has collected more reward, less reward or the same amount of reward during the current episode;
- The distance to the nearest reward in the state; the distance of the **nearest other agent** to that nearest reward **and** the distance of the nearest agent to **any** reward, whether **those agents** are in the same team or not and if that agent or agent's group has collected more reward, less reward or the same amount of reward during the current episode;
- The distance to the **two nearest rewards** in the state; the distance of the nearest agent to any reward, whether that agent is in the same team or not and if that agent or agent's group has collected more reward, less reward or the same amount of reward during the current episode;
- The distance to the two nearest rewards in the state; the distance of the **nearest other agents (one nearest agent to each nearest reward) to those nearest rewards**, whether **those** agents are in the same team or not and, if those agents or their groups have collected more reward, less reward or the same amount of reward during the current episode;
- The distance to the two nearest rewards in the state; the distance of the nearest agents to the nearest reward (one nearest agent to each nearest reward) **and**, the distance of the nearest agent to any reward and, whether those agents are in the same team or not and if the agent or their group has collected more reward, less reward or the same amount of reward during the current episode;

In the following experiments, training the function approximator and reinforcement learning happens at the same time. There are now two optimisation processes running simultaneously and both have the potential to fail to converge in time or to diverge. The agent's action-selection policy influences the rate at which the agent exploits the knowledge about the task that is stored in the value function - in this case the neural network. Thus there is the potential for the choice of stationary or non-stationary policy to make a more significant difference than in Chapter 5.

6.2.1 State Representation

The state and action features are used as inputs to the neural network in the same format that they were entered in the columns of the tabular value function in Chapters 4 and 5. The format of the state features and actions was chosen so that both the tabular and approximated value functions would have the same inputs. Although it would have been better to choose different formats in terms of performance[4], the same format allows a direct comparison and highlights the difference between tabular and approximated value functions. In previous published work on group utility functions work [Bradley and Hayes, 2005a], [Bradley and Hayes, 2005b] for the value function table the distance was split into concentric circles starting with small distances and increasing, i.e. 1m, 2m, 4m, 8m, 20m, etc. to allow much of the state space to be covered whilst keeping its representation small. Tile coding [Sutton and Barto, 1998] could have been used for the same reason with probably greater success but, the experiments in Chapters 4 and 5 show that group utility functions work to mix a qualitative behaviour with a quantitative goal and give a good comparison point for the upcoming experiments in this chapter.

Distance is represented as a real valued number for the neural network input. The distance values are scaled down (divided by one hundred) to make the values closer to the other input values (of zero and one). Having some input values orders of magnitude larger than others can make it difficult for the neural network to converge as it has to account for the large differences in input values on top of learning. To represent whether, with respect to another agent or group, an agent is winning, drawing or losing and whether an agent is in the same team or not, 1-of-C coding is used [Sarle, 2009]. For 1-of-C coding each possible category is given its own variable set to positive or negative *one*. For example the winning, losing or drawing feature is represented by three bits and to indicate winning the pattern is [1, -1, -1], losing is [-1, 1, -1] and drawing is [-1, -1, 1]. This should make it easier for the neural network to adapt appropriately in response to the coded qualitative features. One should certainly not make the mistake of representing three categories with one variable with three settings i.e. one variable set to one or two or three, as the qualitative categories are likely not linearly dependent yet this coding implies that dependency. Using 1-of-(C-1) coding will suffice but care must be taken to ensure other properties of the neural network

[4]The tabular case was disadvantaged by having a larger state-action space than could have been devised. The function approximation method is capable of dealing with many inputs so may have benefited from a more detailed description of the environment and even the actions of the other agents.

hold [Sarle, 2009]. No such care must be taken for 1-of-C coding making it suitable and simple for experimentation. It would be equally valid and more descriptive to use a single scalar value to represent the difference between agents and/ or groups but this would lead to a vastly larger state space which would have a massive impact on the ability of the tabular value functions used in Chapters 4 and 5.

6.2.2 Exploration

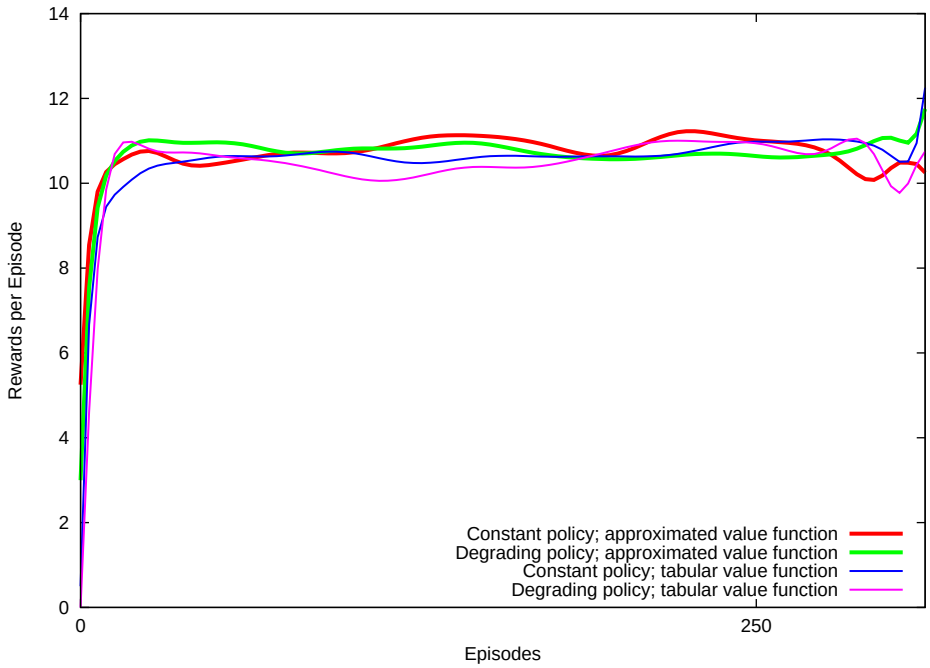
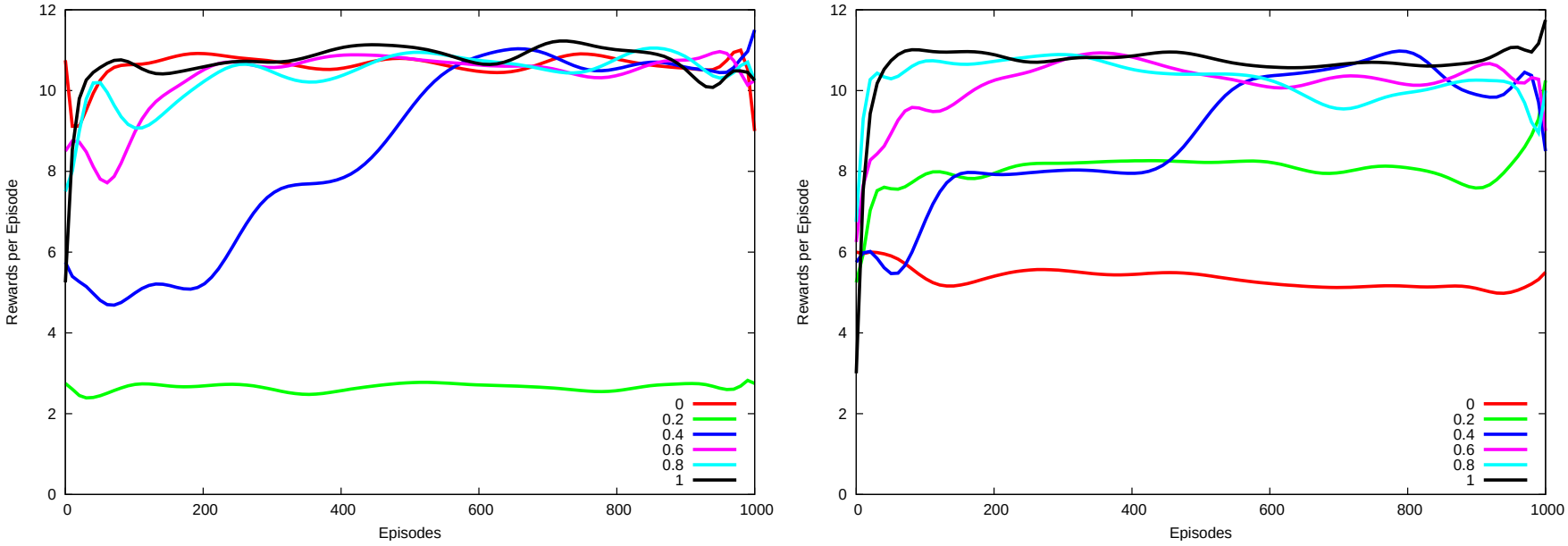


Figure 6.3: Comparison between tabular value function and an approximated value function for both degrading and constant policies (i.e. degrading and constant ε). Each of the experiments featured just one single agent. The tabular value function lines were shown before in Figure 4.5.

Figure 6.3 shows the performance of an individual agent performing the foraging task (with a similar set-up to the experiments in the previous chapters) when that agent is using function approximation as described above for its value function. The points on the lines are calculated by regularly testing the agents' performance using an extra episode. As before, for these test episodes the agents act entirely greedy with respect to their value functions. That is, they are entirely exploitative and perform no random exploration during these test episodes. The graph shows two lines for this single agent: degrading ε and stationary ε. The graph also shows the same lines for when using

tabular value functions as in Chapters 4 and 5. These lines were previously shown in Figure 4.5. The lines representing the use of approximated value functions show the same level of performance as in the tabular cases for constant and degrading ε values. Various other values of ε when kept constant and non-constant are shown in Figure 6.4. As the reader can see, some of the lines in these graphs for low values of ε, i.e. little exploration, show no signs of learning or only learn late in the experiments. This is particularly true for the experiments where ε is reduced to zero over the length of the experiment. This is a sign of poorer performance when using function approximation compared to using tabular value functions as can be seen by looking at the equivalent graphs for tabular value functions in Figure 4.6. To understand the reason why this might be the case one can look at the value functions after learning as shown later in Figures 6.8 through to 6.16. The difference in performance is explained later in this section.

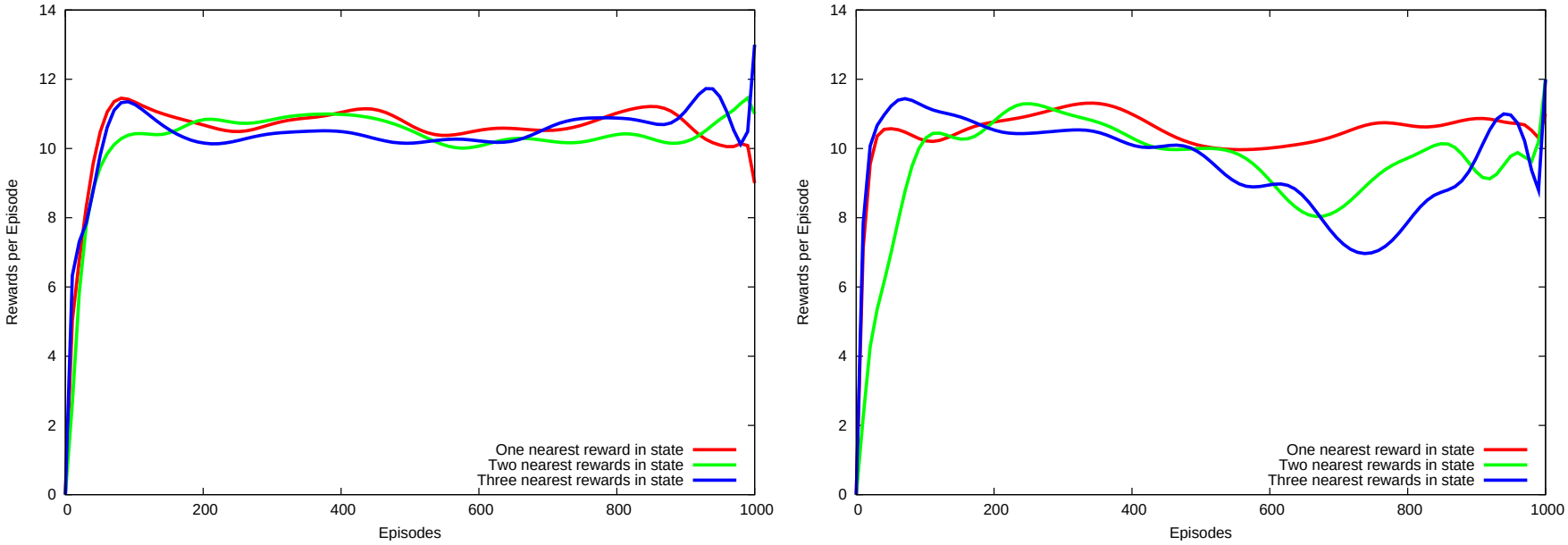


(a) Constant ε

(b) Degrading, initial ε degrading linearly to zero at the end of the experiment.

Figure 6.4: Shows the performance of a single agent with an approximated value function for various values of ε.

Comparing Figure 6.5 to its tabular value function counterpart in Figure 4.7 one can immediately see the benefits of approximating the value function using a neural network. The effect of more rewards represented in the agent's state has no discernible effect on the final performance of the agent. Also, Figure 6.5 shows us that for this task, knowing the distances to the two or three nearest rewards is only as useful as knowing the distance to the nearest reward only as the asymptotic performance is unchanged. This makes sense considering the simplicity of the foraging task. Although on close inspection, in both Figure 6.5(a) and Figure 6.5(b) those experiments with the distance to more rewards kept in the state representation do collect more rewards than those experiments using smaller state representations. This is not significant enough to draw any conclusions from though. In Figure 6.5(b) there is some difference in the learning of the agent over the course of the experiments. Each plot of rewards collected per episode "dips" approximately two thirds of the way through the experiment. Close inspection revealed that at this point in the experiment the value function was higher for short *and* long distances to the nearest reward. The value function differed from the ideal as shown in Figure 6.6. This is probably due to a combination of the generalisation that function approximation provides and degrading ε. As the value function takes the true shape so the agent also starts to exploit the knowledge that it holds. For these experiments the agent used an initial ε value of *one*. As ε comes down so the agent chooses to exploit its value function more. At a certain point in the experiment the agent will have learned to move towards the nearest reward successfully and will do so more and more exclusively. Because function approximation updates the utility value for not just the current state-action pair but, neighbouring pairs, the value function between the average starting distance away from a reward and zero metres steepens. The tail of the value function, out beyond the average distance from a reward, is left high. The approximated function is then (mistakenly) *u*-shaped until the agent moves in that direction, as it will do because its value function says it is good to do so. The agent's visits to the far distances from the nearest reward and subsequent learning correct the shape of the value function and the performance recovers. This shows one of the problems with approximating the value function. The function approximation acts like a rubber band if viewed as two-dimensional, or rubber sheet if viewed as three-dimensional. Whenever an error is back-propagated at one state-action pairing the function is affected throughout the length of all of its dimensions.



(a) Constant $\varepsilon = 1$

(b) Degrading, initial ε degrading linearly to zero at the end of the experiment.

Figure 6.5: Shows the difference in performance when more rewards are represented in the state (this also gives the agent more action choices) when using function approximation. Each experiment featured just one single agent. The equivalent graph for tabular value functions is shown in Figure 4.7. Note the prominent dip in performance for the degrading ε experiments.

Whereas Figure 6.5 highlighted the benefits of approximating the value functions (i.e. almost no difference in end performance despite an increased state-action space), Figure 6.7 highlights the frustrations of function approximation when used with reinforcement learning. In the equivalent experiments using tabular value functions shown in Figure 4.8 the number of rewards collected per episode increased the longer the agent learned for. However, in Figure 6.7(a) the agents perform worse as the experiments lengthen. In Figure 6.7(b) the three thousand episode long experiment does outperform the shorter experiment, but the ten thousand episode long experiment has the worst asymptotic performance. These set of experiments are discouraging after the encouraging results shown in Figure 6.5. The results shown in Figure 6.7 are not poor in comparison to their tabular equivalents shown in Figure 4.8 as they have have all increased significantly to the point where their asymptotic performance is only just under the best seen for a single agent. The disappointment is in seeing a slight decrease in performance over time in Figure 6.7(a) and no discernible pattern or relationship in Figure 6.7(b).

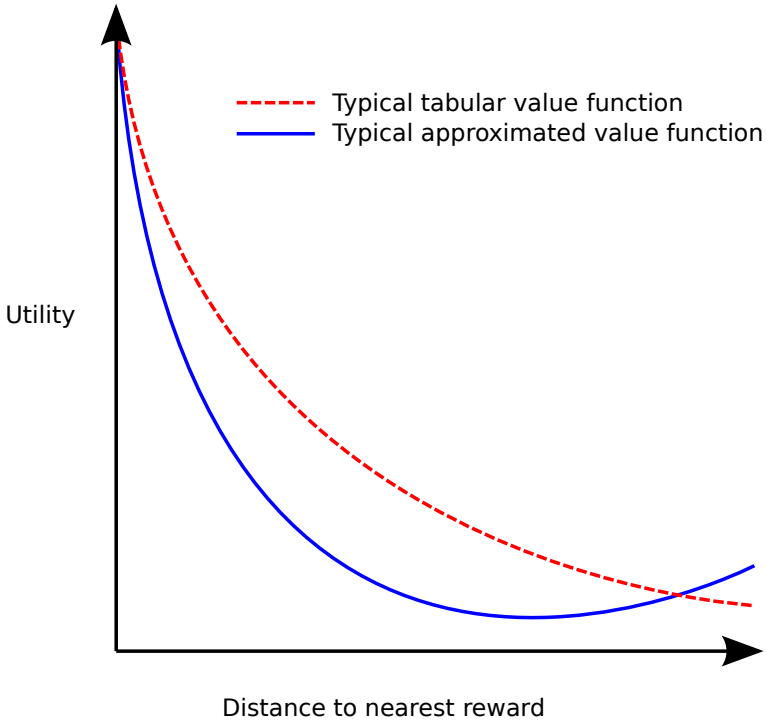
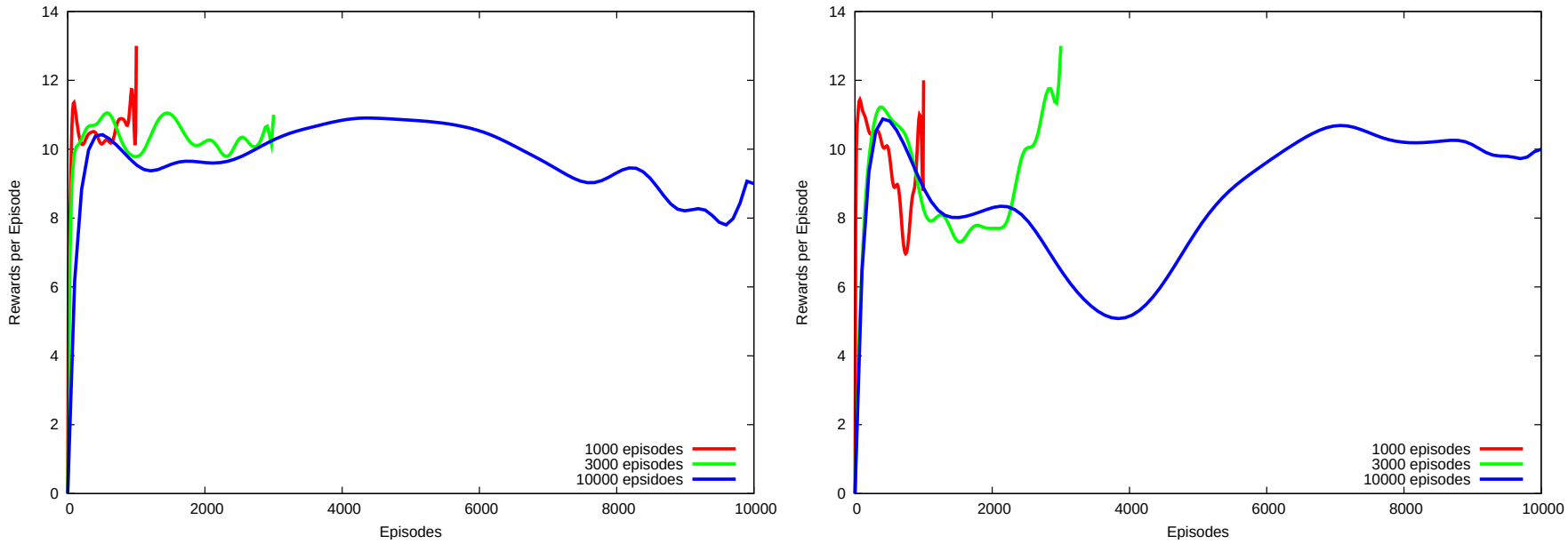


Figure 6.6: Shows the dip in the approximated value function which produces a dip in performance. This simplified graph shows a typical approximated function when about two-thirds of the experiment has run. The red line shows the shape of a typical tabular value function and the blue line shows the shape of a typical approximated value function.



(a) Constant $\varepsilon = 1$

(b) Degrading, initial ε degrading linearly to zero at the end of the experiment.

Figure 6.7: The experiment with three rewards in the state as shown in the blue lines of Figure 6.5 repeated but with three thousand and ten thousand instead of one thousand (red lines here) episodes. The tabular value function version of this experiment is shown in Figure 4.8.

To see why these few experiments using function approximation are unstable and give mixed results one can look at the learned value functions. The value functions for the case with a single agent learning over one thousand episodes with the distances to the nearest three rewards kept in its state representation are shown in Figures 6.8 (constant ε) and 6.9 (degrading ε). Figure 6.9 clearly shows a good slope upwards as the agent moves closer to the reward and is a reasonable approximation of the equivalent tabular value function previously seen in Figure 4.10. In the tabular case there was a lot of noise but a good curve was still visible whereas in this case, the approximated value function, there is little noise due to generalisation but the curve is very shallow if it exists at all. In Figure 6.8 much as in its tabular equivalent value function in Figure 4.9 there is little to be seen of any function, let along a good curve like one would expect or hope for. The function in Figure 6.8 does show sporadic signs of rising when the agent is close to the reward and that is clearly enough to produce reasonable performance as seen above in Figures 6.7(a) (shortest line) and 6.5(a) (blue line). As with all value functions and reinforcement learning, the magnitude of the value function does not necessarily have to match that of the true reward function but the shape does need to.

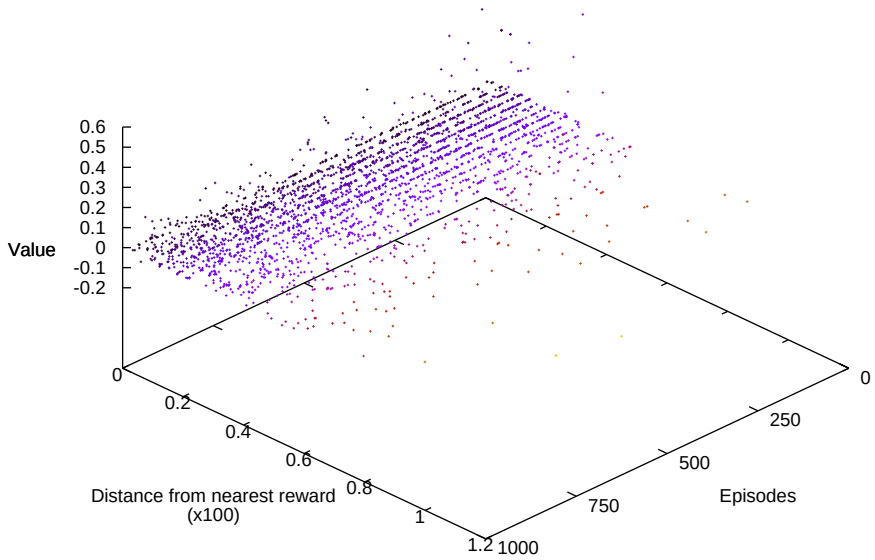


Figure 6.8: One agent's value function as it is updated during the experiment over one thousand episodes with three rewards in its state representation. The tabular value function equivalent of this graph is shown in Figure 4.9.

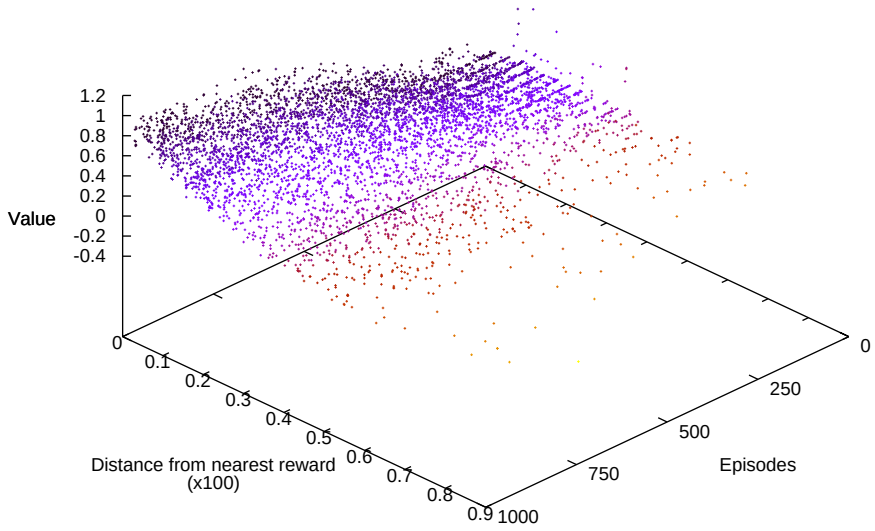


Figure 6.9: One agent's value function as it is updated during the experiment over one thousand episodes with three rewards in its state representation. The tabular value function equivalent of this graph is shown in Figure 4.10.

The effect of varying the alpha values is shown in Figures 6.10 through to 6.16 where α is set at 1, 0.1, 0.01, 0.001, 0.0001, 0.00001 and 0.000001 respectfully. The agents choose their actions according to the usual ε-greedy policy with ε degrading linearly from *one* to zero. These graphs can be compared to those from 4.11 to 4.17 which are the same experiments except for the use of tabular value functions.

Figure 6.11 is by far the best approximation of the expected curve from zero to one. It looks most like the curve of its tabular variant in Figure 4.12 for $\alpha = 0.1$ because it appeared to be the most accurate value function. That for the approximated value function, $\alpha = 0.1$ learns the best value function provides further indication that, empirically at least, this value of α works well for these experiments. Otherwise, Figures 4.13, 4.14, 4.15 and 4.16 for $\alpha = 0.01, 0.001, 0.0001$ and 0.00001 approximate their tabular value function equivalents with some small degree of accuracy. They are in reality ever so slightly sloped upwards in the direction of the rewards meaning that just this small gradient slope will actually allow the agent to perform just as well as that agent in whose value function is shown in Figure 6.11 for $\alpha = 0.1$. Figures 6.10 and 6.16 mostly fail to approximate the true reward function other than in the slight

trend towards the rewards. An α value of 0.000001 is clearly too small as the value function is still moving from its initial random readings towards zero. This is shown in Figure 6.16 as a slope running the opposite direction to the other similar graphs and is the result of such a small α value nullifying the positive reward received when picking up a reward. Without that positive reward the value function is tending towards zero (from a random starting point) for the entire state-action space. Figure 6.10 for $\alpha = 1$ fails to approximate the reward function probably due to the effects of too much noise as shown in its equivalent tabular value function as shown in Figure 4.11.

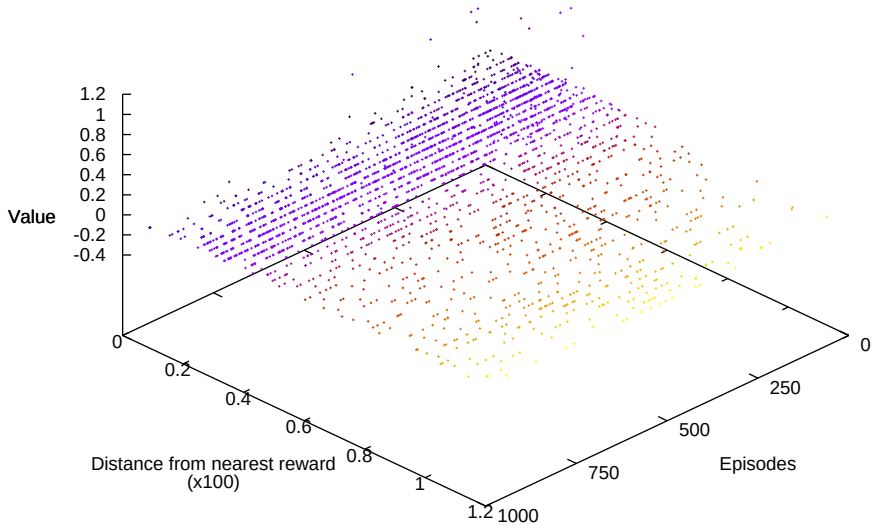


Figure 6.10: One agent's approximated value function as it is updated during the experiment with $\alpha = 1$.

The value functions plotted from Figure 6.8 to Figure 6.16 remind one of the point that function approximation really is only an approximation and that the significant gains in storage space and associated look-up speed improvement and the benefits of generalisation come at the price of poorly learned value functions. The low quality of the approximated value functions can be seen by comparing Figures 6.8 through to 6.16 with their tabular value function counterparts in Figures 4.9 through to 4.17.

Next is the series of experiments previously run for and shown in Chapter 5 repeated with the value function approximated by neural networks instead of stored as a table. In practical terms, these experiments ran about twice as fast and took up sub-

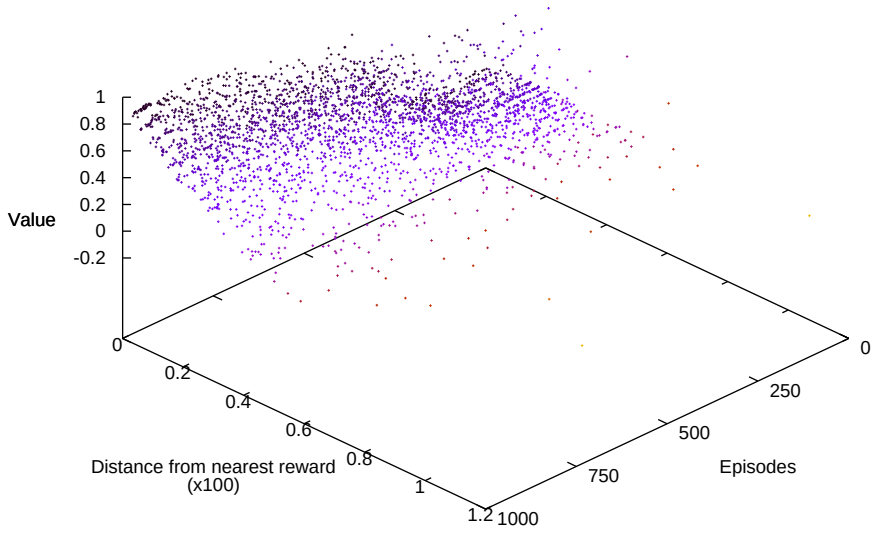


Figure 6.11: One agent's approximated value function as it is updated during the experiment with $\alpha = 0.1$.

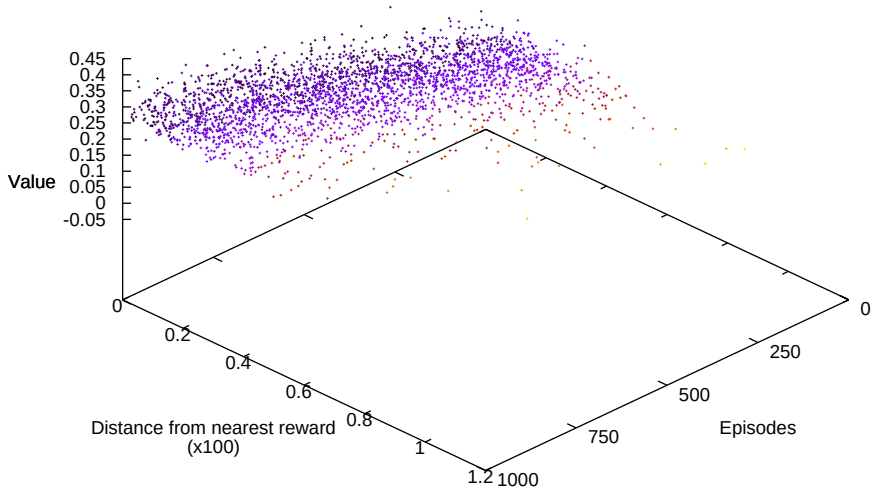


Figure 6.12: One agent's approximated value function as it is updated during the experiment with $\alpha = 0.01$.

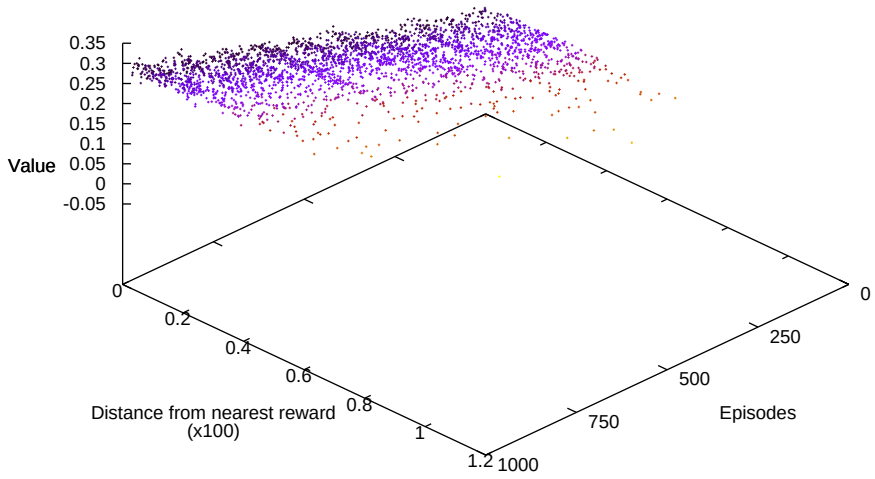


Figure 6.13: One agent's approximated value function as it is updated during the experiment with $\alpha = 0.001$.

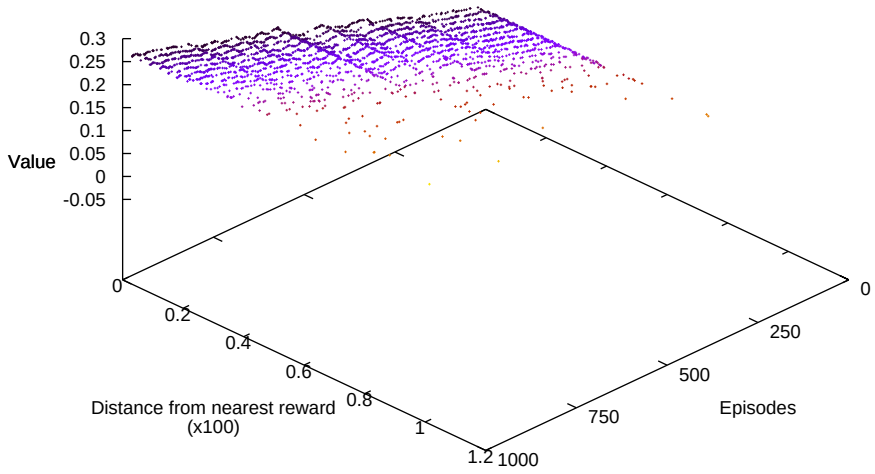


Figure 6.14: One agent's approximated value function as it is updated during the experiment with $\alpha = 0.0001$.

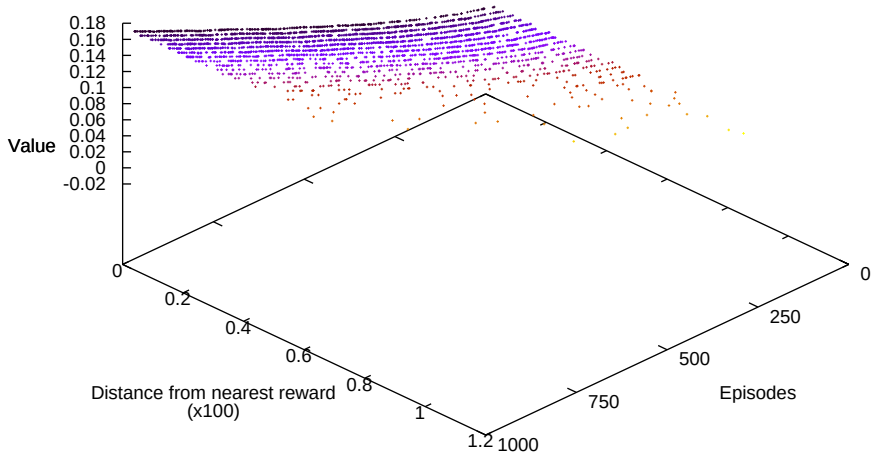


Figure 6.15: One agent's approximated value function as it is updated during the experiment with $\alpha = 0.00001$.

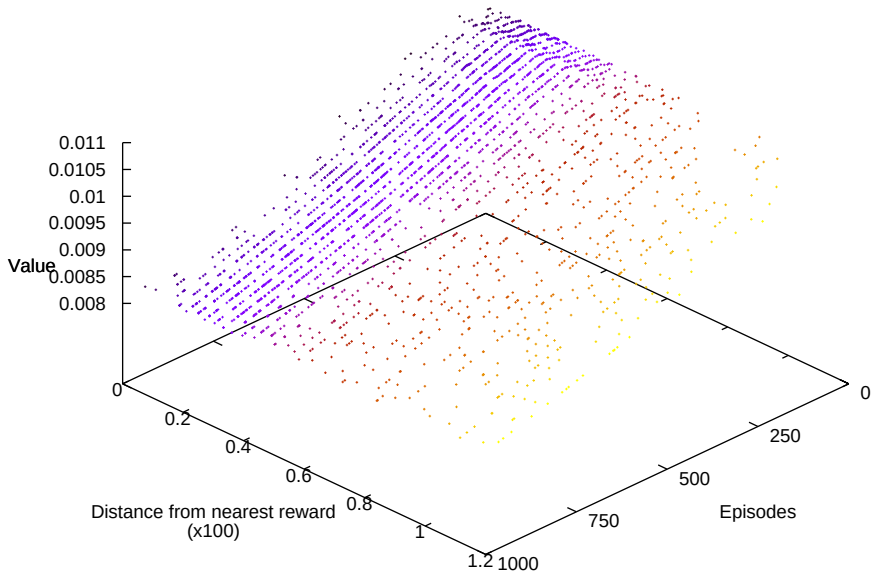


Figure 6.16: One agent's approximated value function as it is updated during the experiment with $\alpha = 0.000001$.

stantially less run-time memory. In fact the neural networks take up hardly any storage space at all especially when compared to the larger memory resources required by the tabular value functions.

6.3 Results of Approximating the Agents' Value Functions

For this section each experiment run in Chapter 5 was repeated with exactly the same set-up except for the use of neural networks to approximate the value function. Each of these experiments as listed in Section 6.2 is graphed in the appendices from Figures B.2(a) to B.42(f) just as they were for those tabular value function experiments. Again as in Chapter 6, the experiments for $\varepsilon = 1$ were taken and plotted on the same graph for each tested hierarchy of agents, group and group utility functions. These graphs now follow with commentary on how they differ from their tabular value function counter experiments.

6.3.1 Two Agents

Firstly two agents acting both individually and in balance as arranged in Figures 5.1 and 5.2 were tested with value function approximation. This experiment in Chapter 5 was the first to show that the group utility function using the negative standard deviation of the agents collected rewards worked to produce a balanced performance compared to that of the agents acting without regard for each other.

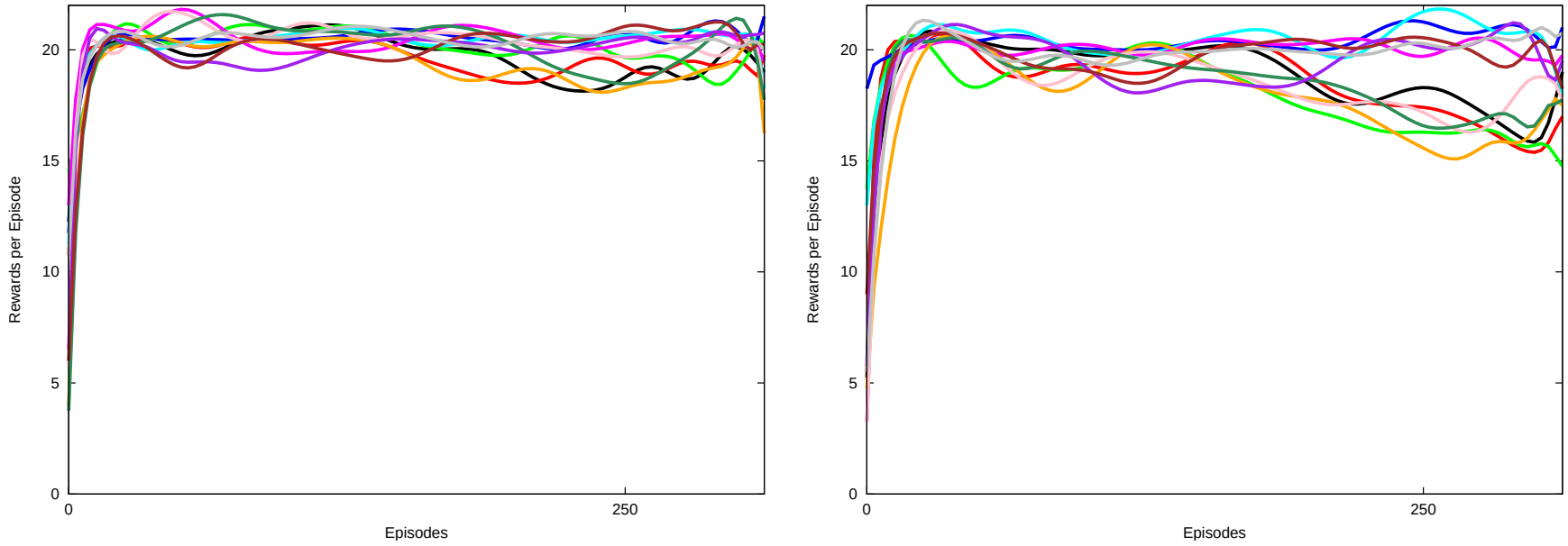
Figure 6.17(a) shows the number of rewards collected by the two single agents each episode whilst learning. The graph is in stark contrast to the tabular value function version of this graph shown in Figure 5.9(a). All of the experiments performed well at learning to pick up rewards. Perhaps two experimental conditions:

- Using a constant ε-greedy action selection policy with the distance to the two nearest rewards in the state; the distance of the nearest agent to any reward, whether that agent is in the same team or not and if that agent or agent's group has collected more reward, less reward or the same amount of reward during the current episode;

- Using a constant ε-greedy action selection policy with the distance to the two nearest rewards in the state; the distance of the nearest agent to the nearest reward

and the distance of the nearest agent to any reward, whether that agent is in the same team or not and if that agent or agent's group has collected more reward, less reward or the same amount of reward during the current episode.

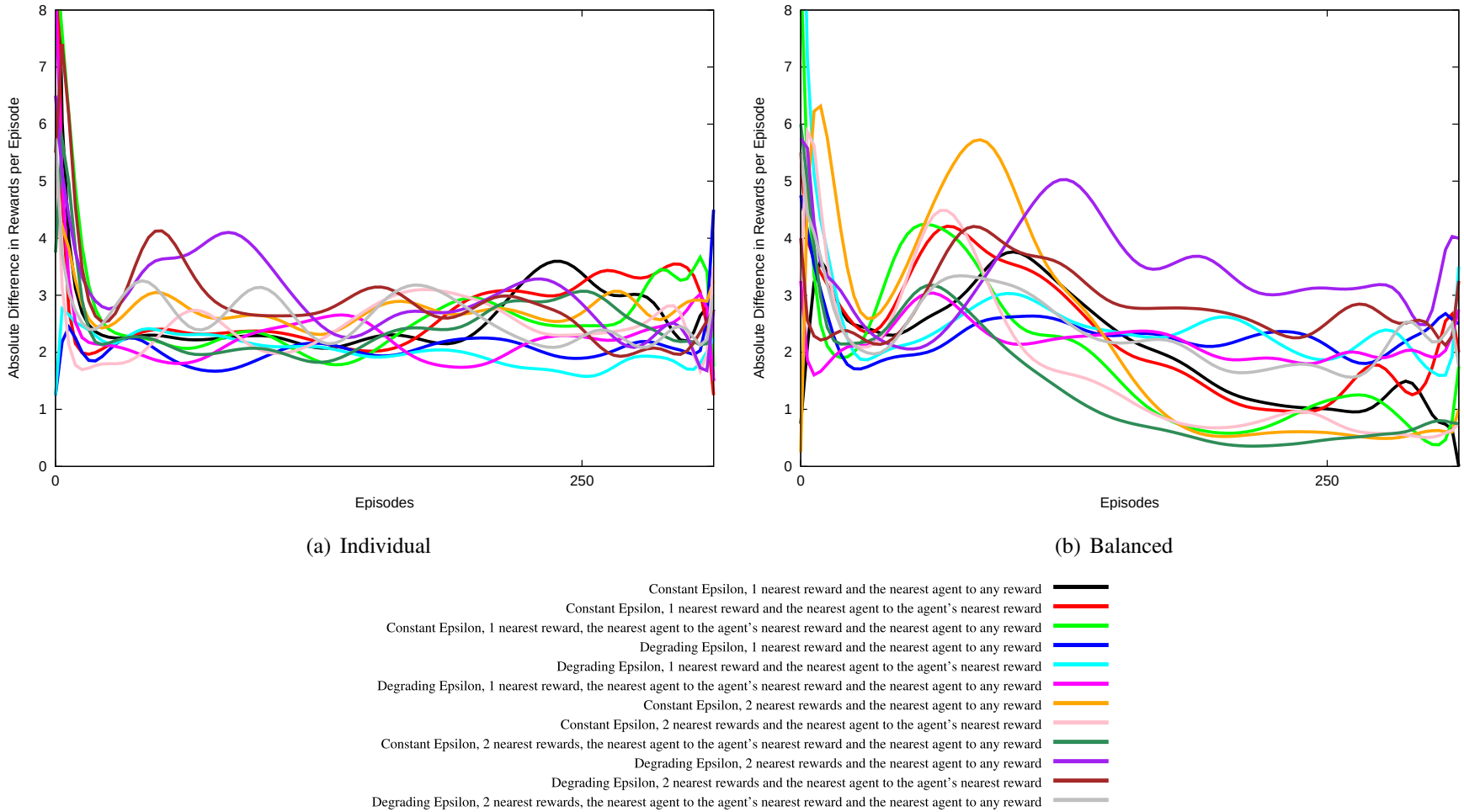
performed worse than the other experiments but it is unlikely to be significant but rather just experimental variation. These results show the benefit of function approximation. Despite being given a larger state space with the same amount of time to learn, the agents under all experimental conditions reached a good asymptotic level of performance. When the agents are under the balancing group utility function as shown in Figure 6.17(b) some conditions show a slight drop in performance. From the equivalent tabular value function graph in Figure 5.9 one can see that in the case for the best performing agents lost 22.5% of their quantitative goal in order to achieve a comparatively more balanced performance. So we must look at Figure 6.18 and in particular Figure 6.18(b) first before judging the success of the agents.



(a) Individual

(b) Balanced

Figure 6.17: Performance of two agents when acting individually and when balanced with all lines with initial $\varepsilon = 1$.



(a) Individual

(b) Balanced

Figure 6.18: Absolute difference in number of rewards picked up by two agents when acting individually and when balanced with all lines with initial $\varepsilon = 1$. Some of the graph points for the early episodes have been cut off to view the interesting results better.

Figure 6.18(b) shows in comparison to Figure 6.18(a) that four experimental conditions produced less imbalance between the agents number of collected rewards for each episode. They were:

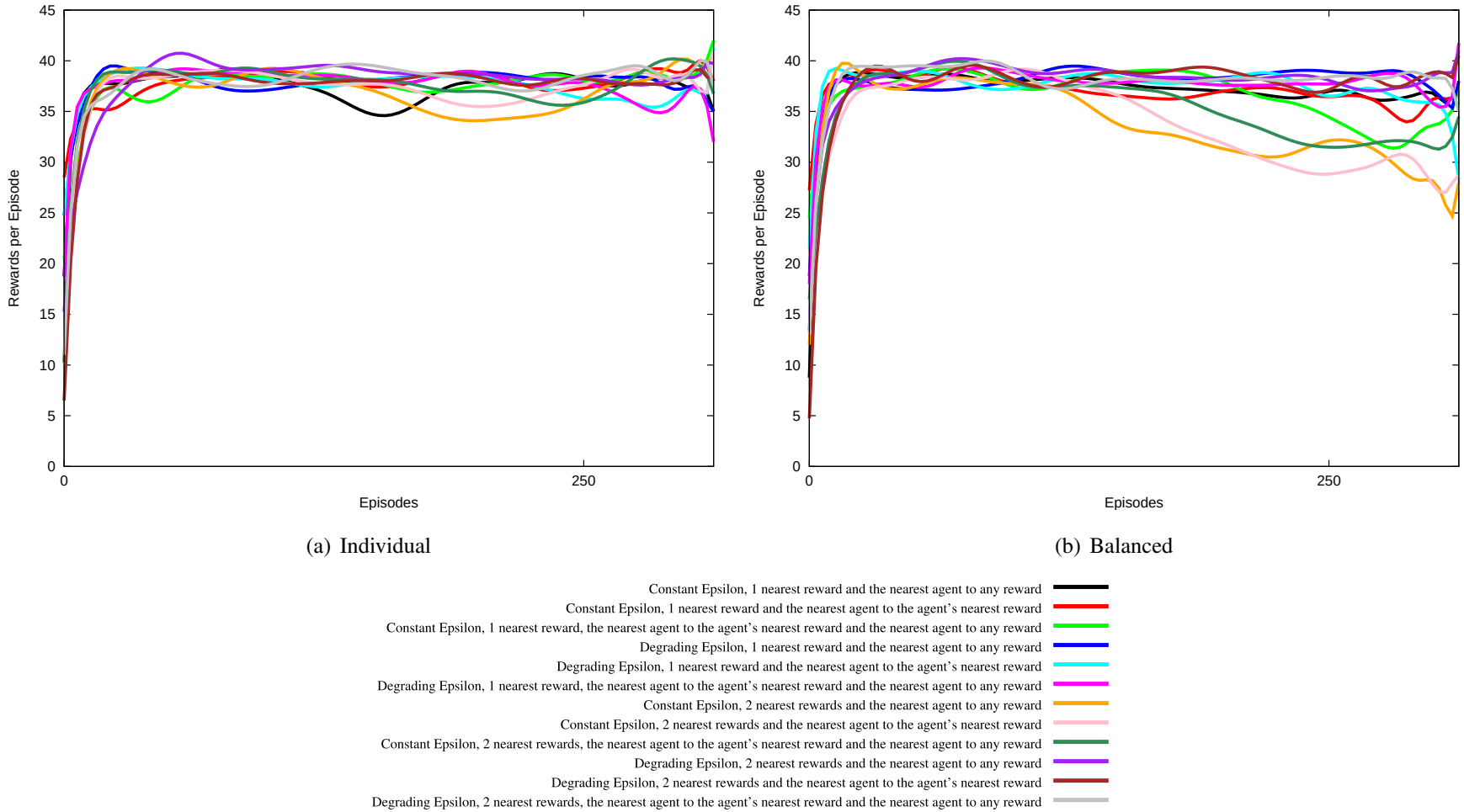
- constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one other agent nearest to any reward in the environment;
- constant $\varepsilon = 1$ when there are two nearest rewards represented in the state and information on one other agent nearest to any reward in the environment;
- constant $\varepsilon = 1$ when there are two nearest rewards represented in the state and information on one other agent nearest to the agent's nearest reward;
- constant $\varepsilon = 1$ when there are two nearest rewards represented in the state and information on one other agent nearest to any reward in the environment and one other agent nearest to the agent's nearest reward.

There seems to be little of significance to these set of experiments other than that their agents all used constant $\varepsilon = 1$ action selection policies. The best performing line when considering balance was for constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one other agent nearest to any reward in the environment which was often the case for the experiments in Chapter 5. Here this set of experimental conditions produced an average difference in the number of rewards collected by the two agents when under the balancing group utility function of zero. This can be seen at episode one thousand for the black curve in Figure 6.18(b). It could not be any more balanced which is a good result. This perfect balance came at a loss of no quantitative performance at all. This is compared to a loss of 22.5% of the total rewards collected in the tabular case of this experiment. This is the perfect result for group utility functions, no loss of quantitative performance and the agents have taken on the qualitative relationship specified by the group utility function. It is probable that these perfect results are unlikely to happen all the time though the experiments were averaged over four runs of the same experiment. It is enough that the quantitative performance is kept relatively high and the qualitative relationship is maintained to a degree, much as in the experiments shown in Chapter 5. The full graphs for each of experimental conditions can be seen in the appendices from Figure B.3(a) to B.10(f). In summary, this set of experiments has shown that the idea of group utility functions still works (indeed works very well) when using neural networks to approximate value functions.

Next as in Chapter 5 the experiments run with evenly resourced groups of agents are repeated with approximated value functions.

6.3.2 Even Groups

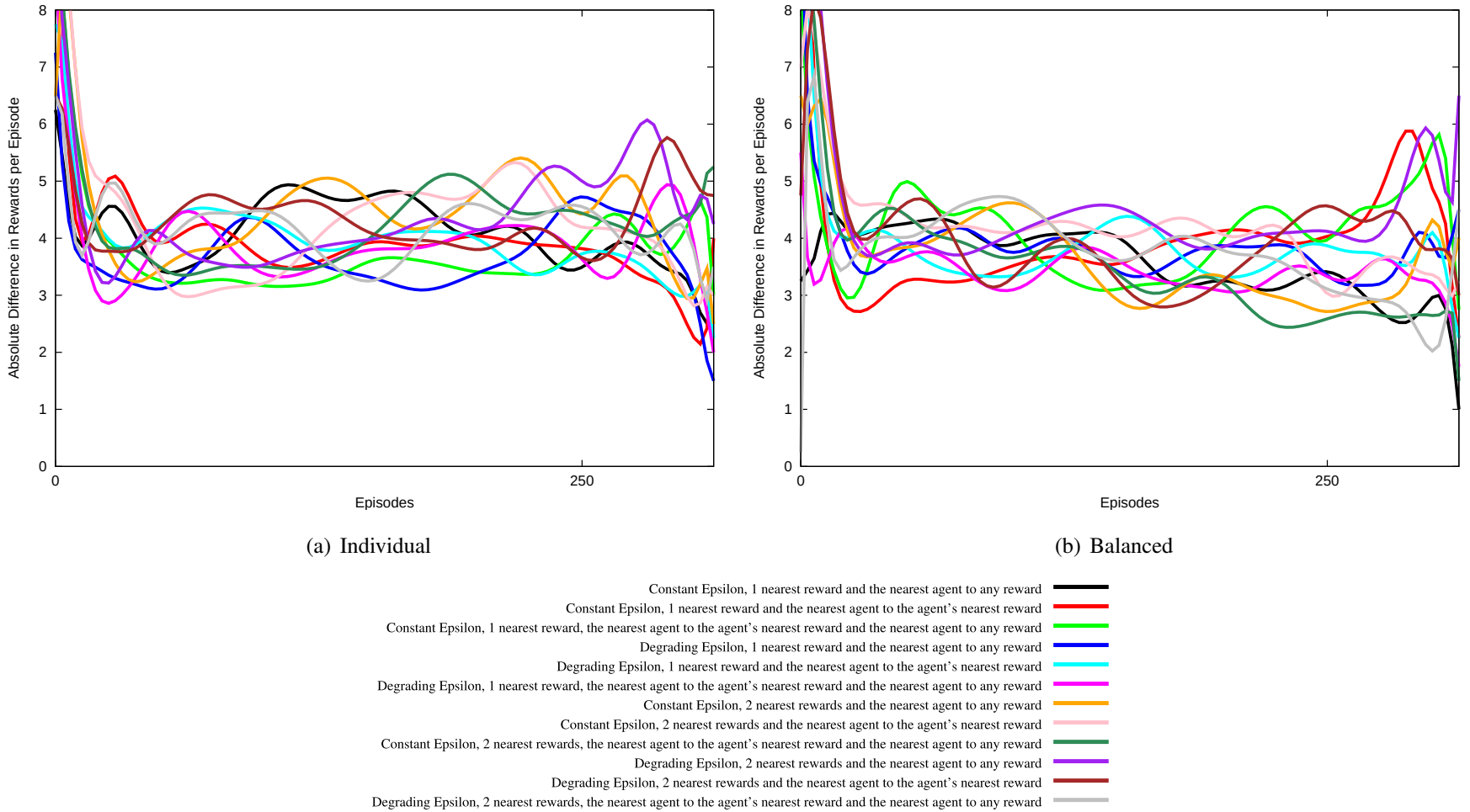
Compared to the perfect balance attained in the last set of experiments for the case for constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one other agent nearest to any reward in the environment, the same experiment for the agents laid out as in Figures 5.11 and 5.12 are good too. The two groups of two agents when acting individually all perform well due to the benefits of function approximation. To take the case for constant $\varepsilon = 1$ when there is one nearest reward represented in the state and information on one other agent nearest to any reward in the environment (black curve) as a reference again, the asymptotic performance is thirty-five rewards per episode as shown in Figure 6.19(a). When under the balancing group utility function arranged as in Figure 5.12 the equivalent asymptotic performance is better, at thirty-seven rewards per episode. This goes against what was seen throughout Chapter 5, that performance when learning to perform only as well as another agent or group would detract from the quantitative goal of maximising number of rewards collected. Of course, the good results shown in Figure 6.19 are have to be considered along with how well the groups have learned to match each other's performance which is shown in Figure 6.20. Figure 6.20(a) shows a natural imbalance between the groups of 2.5 rewards per episode at the end of the experiment whereas for the balanced groups Figure 6.20(b) shows an absolute difference of one reward. That is for the experiment where the agents use constant $\varepsilon = 1$ and when there is one nearest reward represented in the state and information on one other agent nearest to any reward in the environment (black curve again). Some of the other experiments plotted do perform well too. Those that show reduced performance in Figure 6.19(b) do show a corresponding improvement in balance in Figure 6.20(b). It is more difficult to see the failing experiments when using approximated value functions than tabular value functions as with the latter the number of rewards collected was poor whereas with function approximation it is good but the balancing part of the signal fails to be learned. The full set of graphs for each of experimental conditions for two groups of two agents can be seen in the appendices from Figure B.11(a) to B.18(f).



(a) Individual

(b) Balanced

Figure 6.19: Performance of two groups of two agents when acting individually and when balanced with all lines with initial $\varepsilon = 1$.



(a) Individual

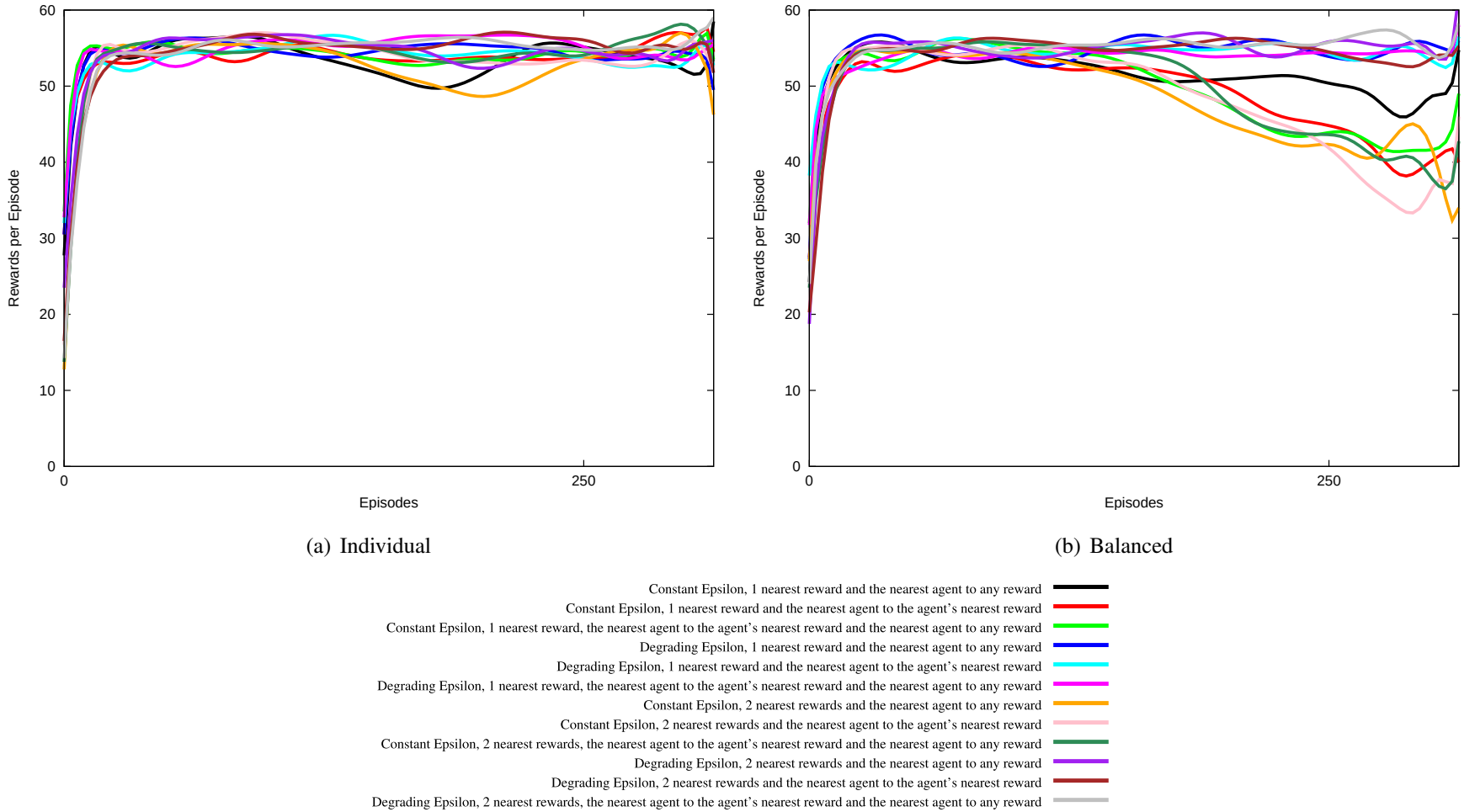
(b) Balanced

Figure 6.20: Absolute difference in number of rewards picked up by two groups of two agents when acting individually and when balanced with all lines with initial $\varepsilon = 1$.

Figure 6.21 shows the number of rewards collected per episode for two groups of three agents arranged as in Figures 5.15 and 5.16. The tabular value function equivalent of these two graph is shown in Figure 5.17. Figure 6.21(a) shows the usual several experimental conditions and shows asymptotic performances all around fifty-five rewards an episode. All of these experiment results are comparable with the best results when using tabular value functions. Figure 6.21(b) shows that for several of the experimental conditions no performance is lost at all and for others, some performance has been lost. Figure 6.21(b) means little of course without looking at the graphs showing naturally occurring and learned balance in Figures 6.22(a) and 6.22(b) respectively. When taking the most often successful case when the agents choose actions with a constant ε equal to *one* and when there is one nearest reward represented in the state and information on one other agent nearest to any reward in the environment one can see that although the number of rewards collected has not fallen in Figure 6.21(b) neither has the imbalance narrowed in Figure 6.22(b) compared to Figure 6.22(a). That is for the most dependable set-up over the series of experiments so far for tabular and approximated value function for this hierarchy of agents, groups and group utility functions, the groups have failed to learn to perform equally as well as each other. This is a disappointing result and the only result so far to not result in the expected agent and group behaviour. The only set of conditions to significantly lessen the imbalance observed in Figure 6.22(a) is for constant $\varepsilon = 1$ when there are two nearest rewards represented in the state and information on one other agent nearest to any reward in the environment (yellow curve) which reduces the absolute difference in rewards collect by the two groups from about four rewards per episode to 1.7 rewards as shown by the lower line in Figure 6.22(b). This reduction in imbalance comes alongside a reduction in total number of rewards collected of twelve rewards, or 26% (again for constant $\varepsilon = 1$ when there are two nearest rewards represented in the state and information on one other agent nearest to any reward in the environment - the yellow curve). That is a 26% drop in quantitative maximisation of the number of rewards picked up in total each episode compared to a 16.4% reduction as seen by the equivalent experiments for tabular value functions. One can draw little from these results other than whereas some of the results for these experiments when using tabular value functions were successful, this set of experiments has failed to learn and maintain both the quantitative and qualitative parts of the reward function. The full set of graphs for each of experimental conditions for two groups of three agents can be seen in the appendices from Figure B.19(a) to B.26(f). Looking at these graphs one can see that there are experimental

conditions, particularly for $\varepsilon = 0.8$ that do maintain a reasonably high level of rewards collected each episode *and* learn a balanced relationship. However, although these results were produced showing that it is possible, they are possibly not as reliable as the often seen best performing experimental conditions usually used as a reference in this chapter and in Chapters 4 and 5 such as the case when the agents choose actions with a constant ε equal to *one* and when there is one nearest reward represented in the state and information on one other agent nearest to any reward in the environment.

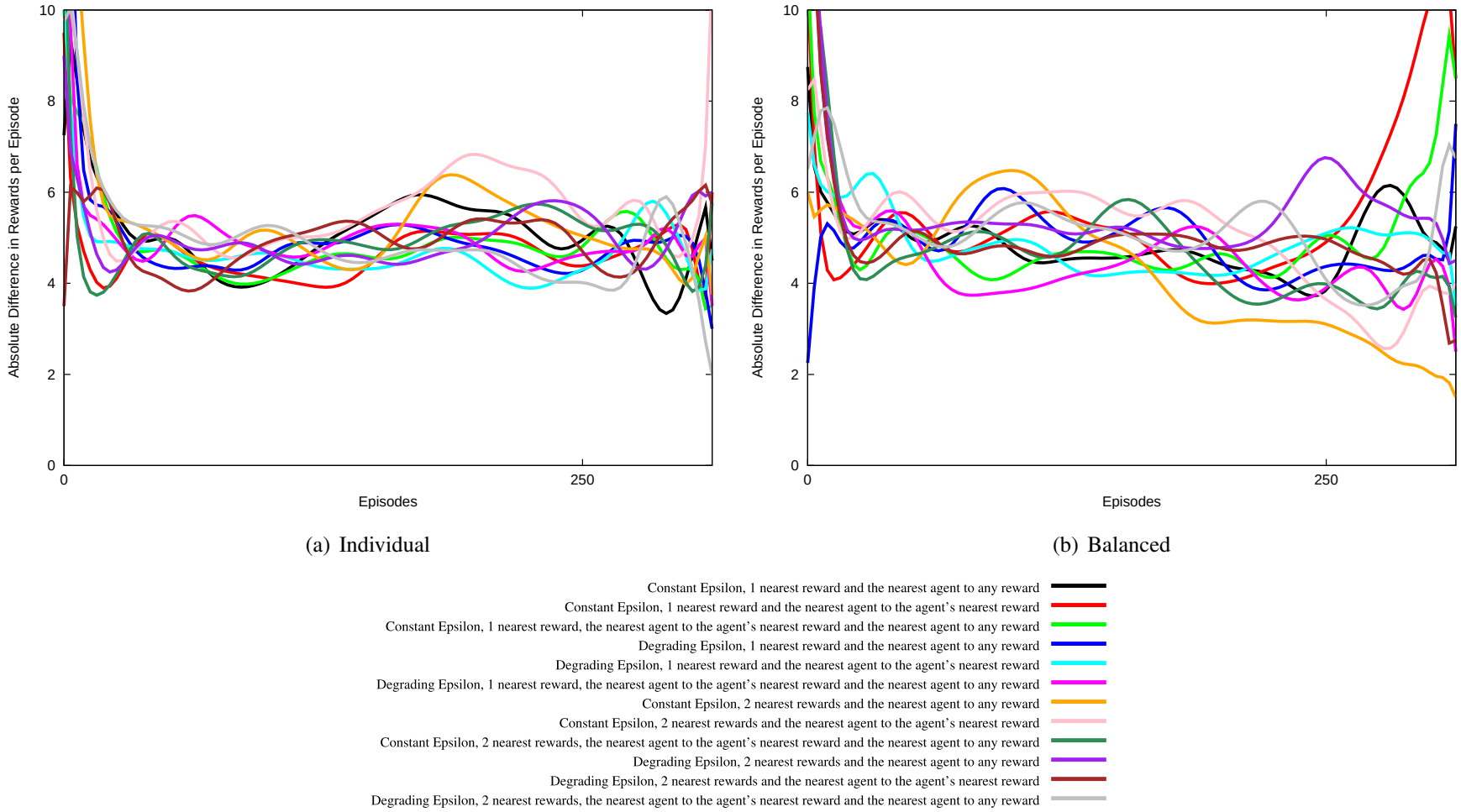
Next the naturally unbalanced groups of agents, a single agent and a group of two agents and a single agent and group of three agents are revisited using function approximation.



(a) Individual

(b) Balanced

Figure 6.21: Performance of two groups of three agents when acting individually and when balanced with all lines with initial $\varepsilon = 1$.



(a) Individual

(b) Balanced

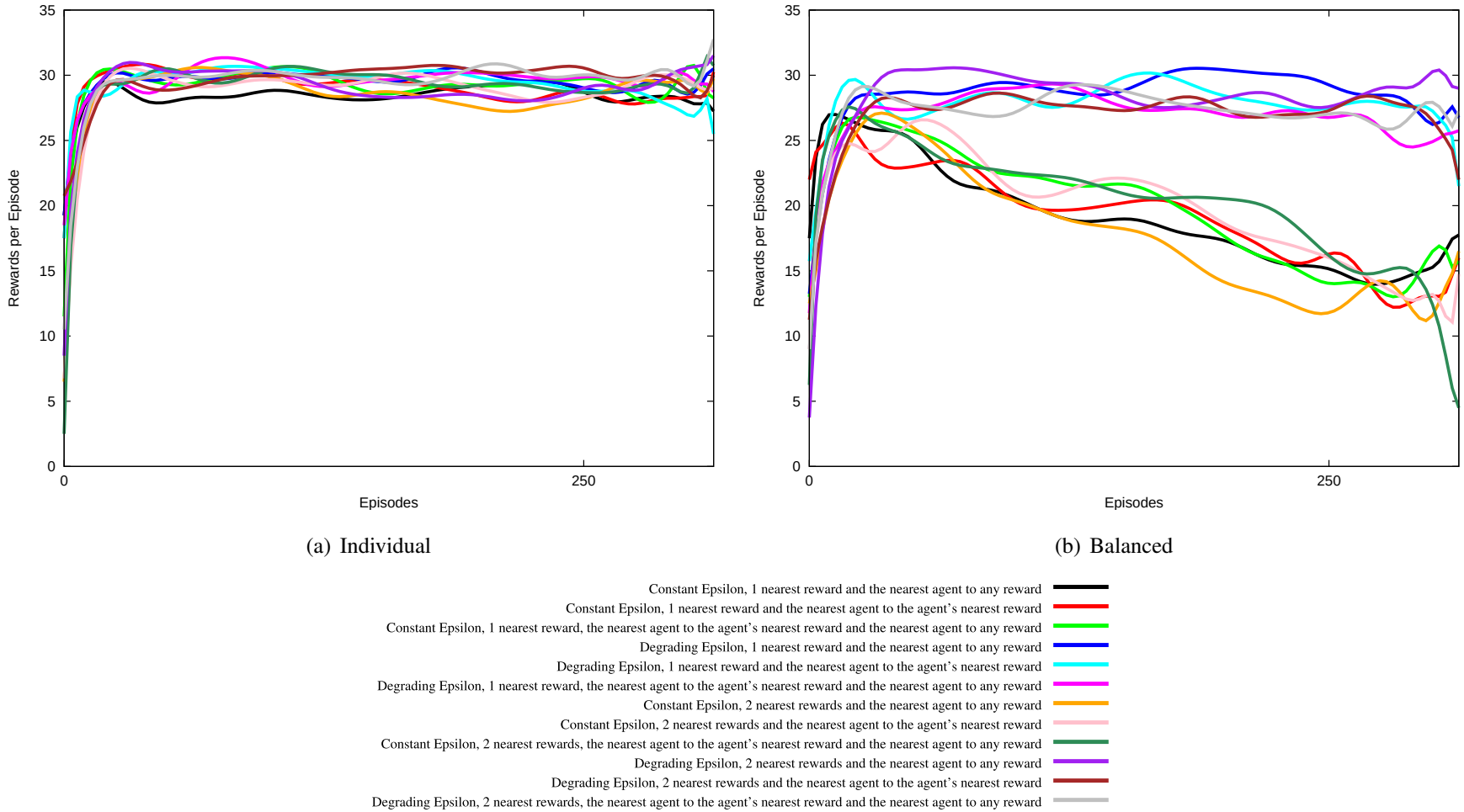
Figure 6.22: Absolute difference in number of rewards picked up by two groups of three agents when acting individually and when balanced with all lines with initial $\varepsilon = 1$. Some points have been deliberately left off the graph to better show the finishing values.

6.3.3 Uneven Groups

Those results in Sections 6.3.1 and 6.3.2 for evenly matched agents or groups performed extremely well in two cases but poorly in one case producing the only failure to learn both the quantitative goal and qualitative relationship. In Chapter 5 it was shown that learning with an uneven number of agents on each side of the relationship was more difficult as the behaviour of the larger group of agents had to be changed a lot because of the increased presence of the qualitative relationship inducing part of the hierarchical of group utility functions. First, experiments were run with the hierarchy of agents, groups and group utility functions shown in Figures 5.19 (unconstrained) and 5.20 (balanced).

Figures 6.23 and 6.24 show, respectively, the collective number of rewards picked up per episode and the absolute difference in the number of rewards picked up by a group of two agents and a single agent. Figure 6.23(a) shows that all the experiments learned to pick up a high number of rewards at asymptotic performance. About thirty rewards are picked up on average which is on a par with the equivalent best experiments using tabular value functions shown in Figure 5.21(a). Figure 6.23(b) shows that several of the experimental conditions produced a noticeable drop in total rewards collected per episode. Those same six experiments show significantly less imbalance in Figure 6.24(b) as compared to Figure 6.24(a). The set of six curves that show both good performance in Figure 6.23(b) and the least absolute difference in Figure 6.24(b) all have one parameter in common. Each of those experimental set-ups uses a constant value of ε rather than altering the action-selection policy over the course of an experiment. This is a significant finding as similar results are seen in the next set of experiments. When looking back at Figures 6.21(b) and 6.19(b) the same effect of using a constant policy can be seen, though to a much lesser extent and only in the curves showing the total number of rewards picked up each episode. In some cases for Figure 6.24(b) such as when the agents choose actions with a constant ε equal to *one* and when there is one nearest reward represented in the state and information on one other agent nearest to any reward in the environment the imbalance is still relatively high compared to previously learned levels of imbalance. The best performing experiment in terms of least rewards per episode lost due to the most balance gained is for constant $\varepsilon = 1$ when there are two nearest rewards represented in the state and information on one other agent nearest to any reward in the environment which was previously shown to be a good experimental set-up in Section 6.3.1. That experiment

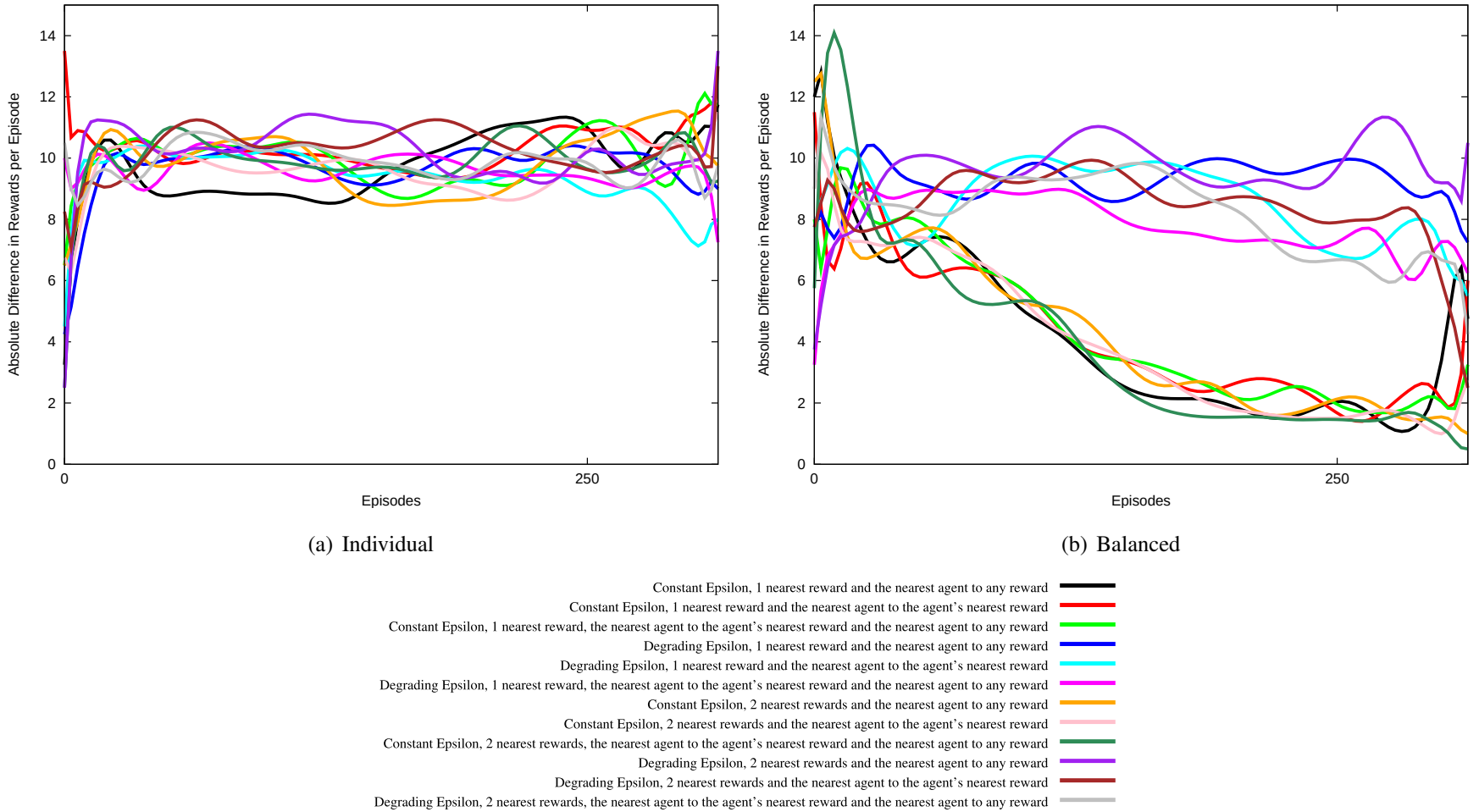
shows a drop in performance of about twelve and a half rewards or 43.1%. Although these values are difficult to conclude anything from as they are a result of the group of two agents learning to perform only as well as one agent. The best result in the tabular value function version of this set of experiments lost 41.7% due to the balanced relationship. The loss of 43.1% of the rewards collected per episode is relatively slightly high but the experiment managed to reduce imbalance from almost ten rewards to one reward per episode. These results indicate a reasonable level of performance but still show an unpredictability between results for the same experimental conditions unlike those experiments in Chapter 5 that gave more predictable outcomes. The full set of graphs for each of experimental conditions for one group of two agents and a single agent can be seen in the appendices from Figure B.27(a) to B.34(f).



(a) Individual

(b) Balanced

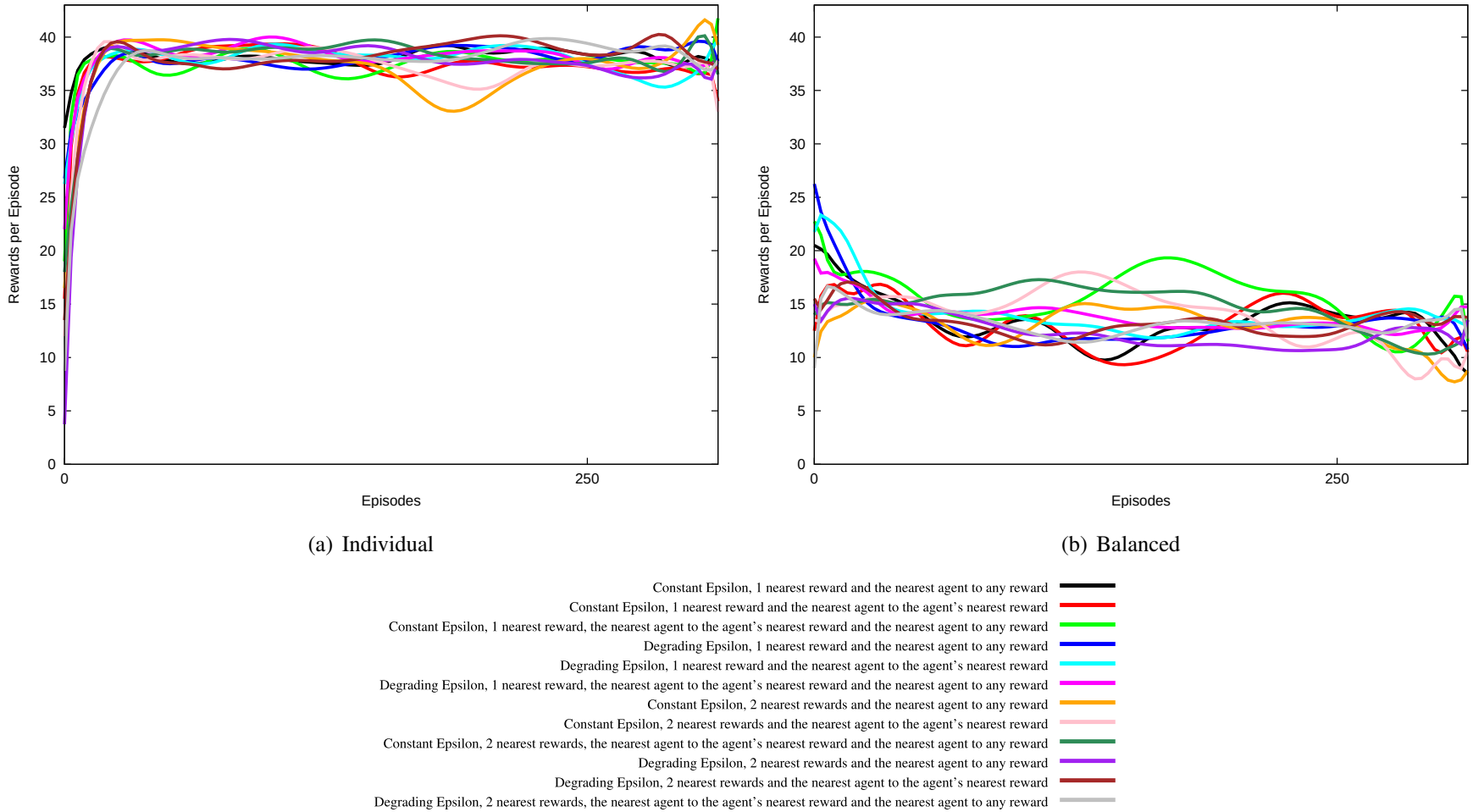
Figure 6.23: Performance of a group of two agents and an individual agent when both acting individually and when balanced with all lines with initial $\varepsilon = 1$.



(a) Individual

(b) Balanced

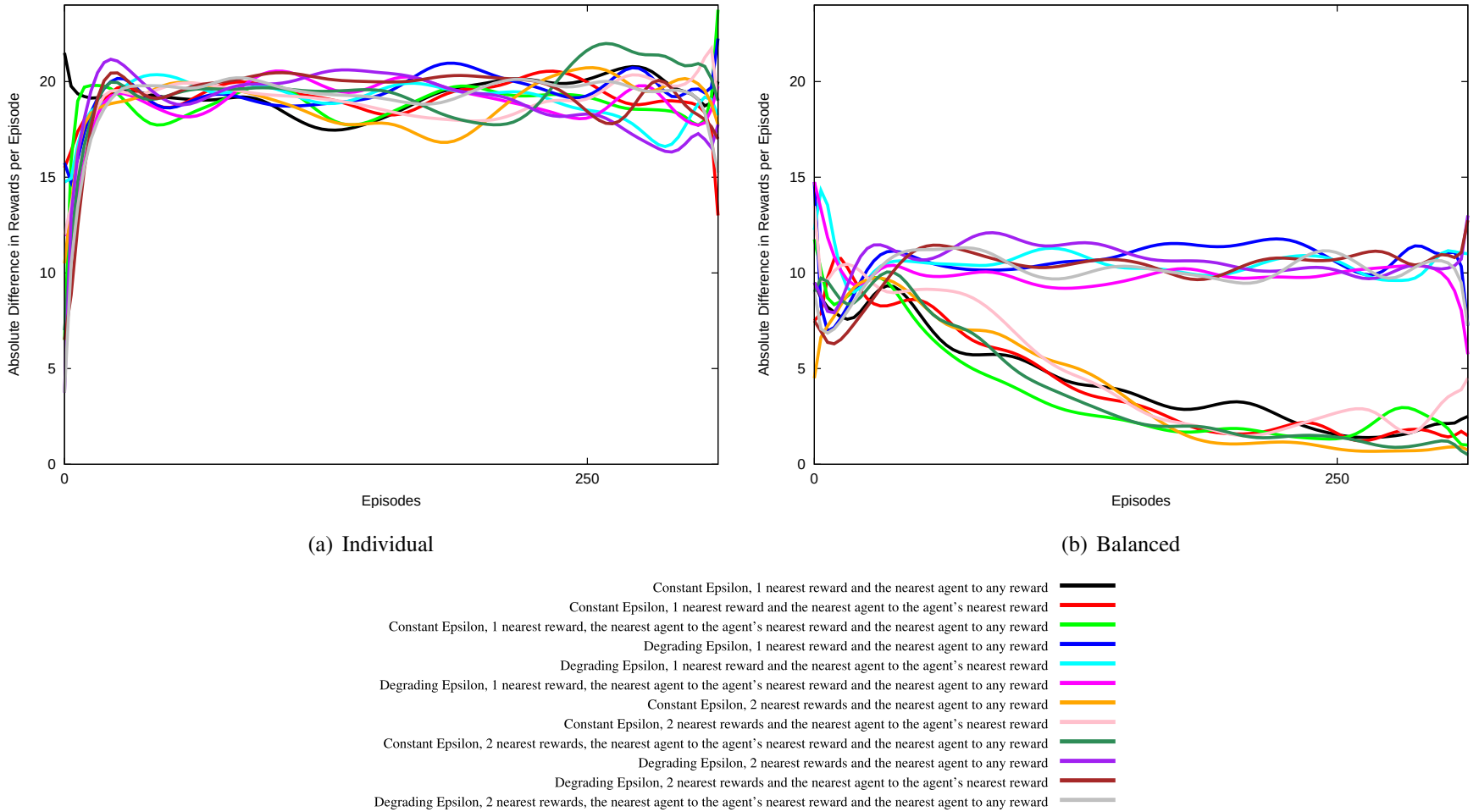
Figure 6.24: Absolute difference in number of rewards picked up by a group of two agents and an individual agent when both acting individually and when balanced with all lines with initial $\varepsilon = 1$.



(a) Individual

(b) Balanced

Figure 6.25: Performance of a group of three agents and an individual agent when acting individually and when balanced with all lines with initial $\varepsilon = 1$.



(a) Individual

(b) Balanced

Figure 6.26: Absolute difference in number of rewards picked up by a group of three agents and an individual agent when acting individually and when balanced with all lines with initial $\varepsilon = 1$.

The final experiments in this thesis, that for two groups of three agents arranged as in Figures 5.23 when unconstrained and Figure 5.24 when constrained by learning the balancing qualitative relationship are shown in Figures 6.25 and 6.26. The full set of experiments for this hierarchy of agents, groups and group utility functions are shown in Figures B.35(a) to B.42(f).

Figure 6.25 shows the total number of rewards collected by the four agents each episode while learning over the course of an experiment. Figure 6.25(a) shows asymptotic performance of thirty-five to forty rewards per episode. This compares well to the best of the equivalent tabular value function experiments of thirty-seven rewards shown in Figure 5.25(a). These agents are attempting to collect as many rewards as possible but the group of three agents must learn to perform only as well as the single agent when under the balancing group utility function. So it is appropriate that the asymptotic performance of the four agents arranged in two groups shown in Figure 6.19(a) is the same.

The same set of six experimental conditions that have learned to significantly reduce the imbalance as shown in Figure 6.26(b) compared to Figure 6.26(a) show a significant decrease in performance in Figure 6.25(b) as compared to Figure 6.25(a). This is mostly because the three agents in the group must curtail their foraging enough to match that of the single agent. The single agent has one third of the resources available and for the case when the agents select actions with constant $\varepsilon = 1$ and when there are two nearest rewards represented in the state and information on one other agent nearest to any reward in the environment and one other agent nearest to the agent's nearest reward, the rewards collected each episode total 14.5 (down from thirty-seven, or a 60.8% drop compared to the same experiment in Figure 6.25(a)). For the same experiment, the absolute difference in rewards collected between the single agent and the group of three agents is half a reward. This experiment can be considered a success and there is little variability in the other experiments shown in Figures 6.25 and 6.26 other than those that fail to learn the balancing part of the reward signal shown as the clearly separate group of upper lines in Figure 6.26(b). Of most interest is the fact that the same result as observed for the previous set of experiments can clearly be seen again in Figure 6.26(b). Each curve that shows decreased imbalance is the average of four experiments using a constant action-selection policy. This is as opposed to linearly degrading ε from *one* at the start of each experiment to zero at the end. The potential for reinforcement learning using function approximation to fail to converge to a satisfactory value function was raised in Sections 4.3.1 and 6.2. There was no

evidence that this would be the case in the simple experimentation shown in Section 6.2.2 at the start of this chapter. Other than, that is, in Figures 6.5 and 6.7 where a dip in performance was seen only in the degrading ε case. This dip in performance was partially explained using Figure 6.6 but the results seen in Figures 6.23, 6.24, 6.25 and 6.26 point clearly to the failure to learn when using approximated value functions.

6.4 Summary

The experiments repeated in this chapter show that hierarchies of group utility functions can be used with function approximation which is in reality a necessity if they are, or reinforcement learning is, to be used in typical large scale computer games containing many non-player characters. Aside from the decrease in storage space over tabular value functions, the benefits of approximating the value function using neural networks that have been shown empirically by the experiments in this chapter are:

- Generalisation means that larger state-action spaces are handled just as well as smaller state-action spaces. This is something that the experiments in Chapter 5 showed were the main problem with tabular value functions. This can be seen most clearly in Figure 6.5 but is evident in all of the experiments shown in this chapter;
- Some of the experiments proved to learn better than their equivalent tabular value function experiments.

In hand with these benefits, the experiments in this chapter highlighted the potential instability of function approximation when compared to the results in Chapter 5. In particular, Figures 6.21 and 6.22 showed no clear results that both the qualitative relationship and the quantitative goal had been learned. Although there were always experimental set-ups that did perform well when using function approximation for some set-ups the varied performances show a lack of stability. This was maybe indicated by the poor quality of the value functions learned as shown in Section 6.2.2.

Most significantly, for the naturally unbalanced hierarchy of groups and agents (those experiments shown in Figures 6.23, 6.24, 6.25 and 6.26), those experiments using a decreasing value of ε failed to learn the qualitative part of the reward signal. These experiments still showed that high numbers of rewards were picked up each episode but did not show that imbalance had been reduced significantly.

6.5 Significance of Results

The duration of the experiments run in the Torque computer game engine can be long and hence, the experiments reported so far have been repeated only four times to form an average curve to plot. The statistical significance of the results shown is therefore unclear. Figures 6.38 through to 6.48 show the six experiment configurations that summarise the findings of the experimentation done. For each experiment configuration, four graphs are shown - rewards collected and absolute difference in rewards collected for tabular and approximated value functions. On each graph there are four curves. These are the number of rewards collected or the absolute difference in rewards collected per episode when the agents/ groups are acting either: individually, balanced, randomly or according to a simple hand-coded policy to move towards the nearest reward at all times. The standard deviations across the repeated experiments are shown at each point also. The curves shown in Figures 6.38 through to 6.48 were calculated over separate runs of one-hundred experiments for each curve in order to investigate the statistical significance of the results. As it proved to work consistently well in this chapter and Chapter 5 the experimental set-up variation using *constant* $\varepsilon = 1$ *when there is one nearest reward represented in the state and information on one nearest agent to any reward in the environment* is used for each of the experiments plotted in these graphs.

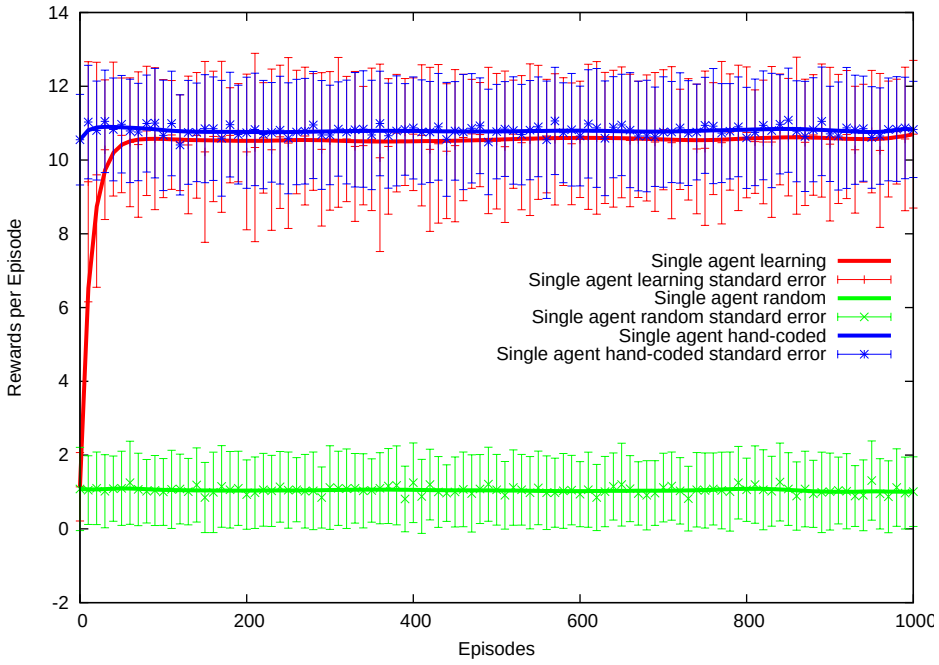


Figure 6.27: Significance of the performance of a single agent using a tabular value function. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

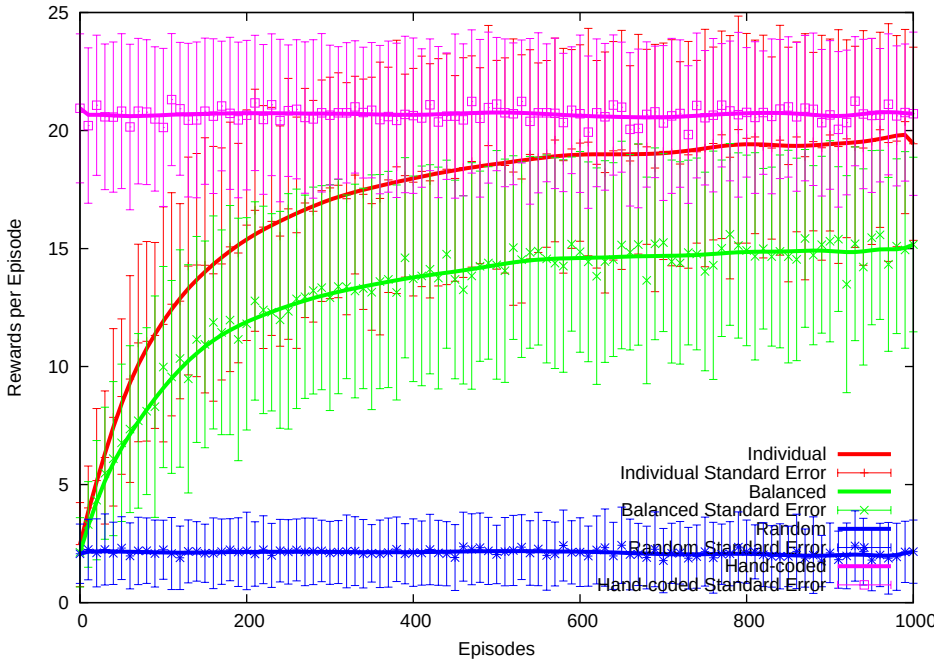


Figure 6.28: Significance of the performance of two agents using tabular value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

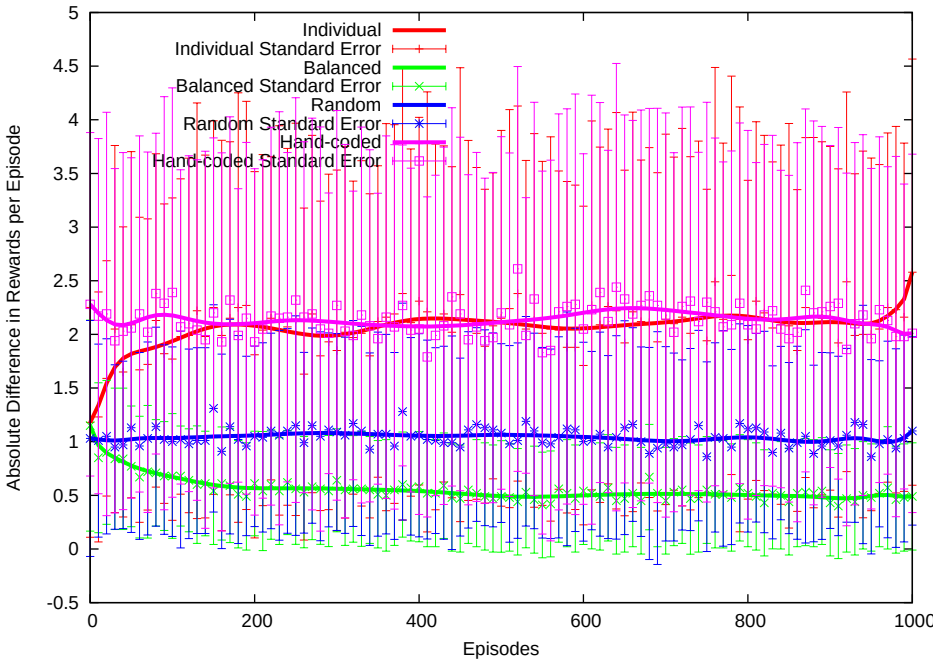


Figure 6.29: Significance of the absolute difference in the number of rewards picked up by two agents using tabular value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

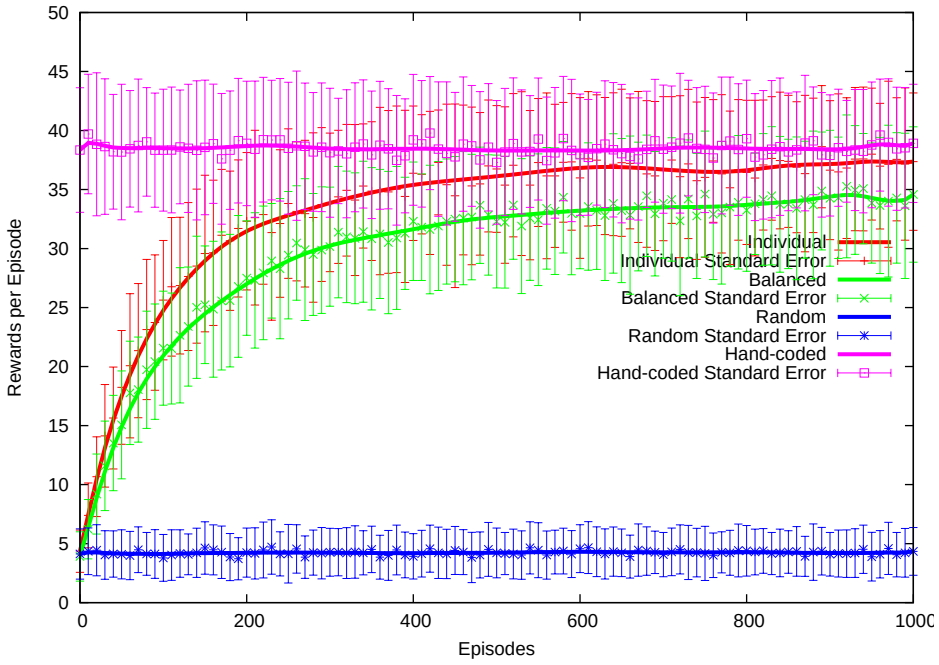


Figure 6.30: Significance of the performance of two groups of two agents each using tabular value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

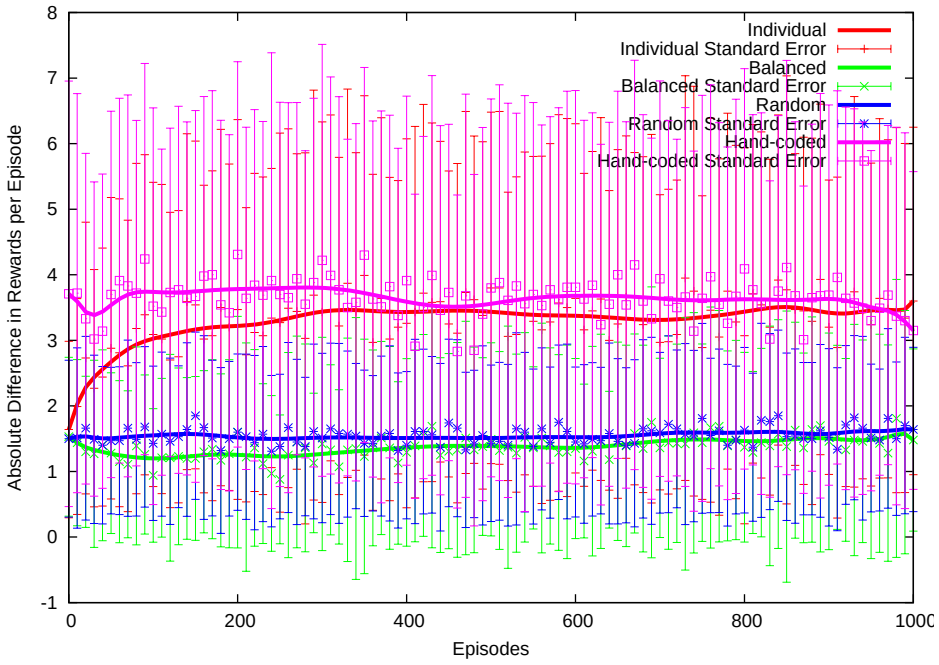


Figure 6.31: Significance of the absolute difference in the number of rewards picked up by two groups of two agents each using tabular value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

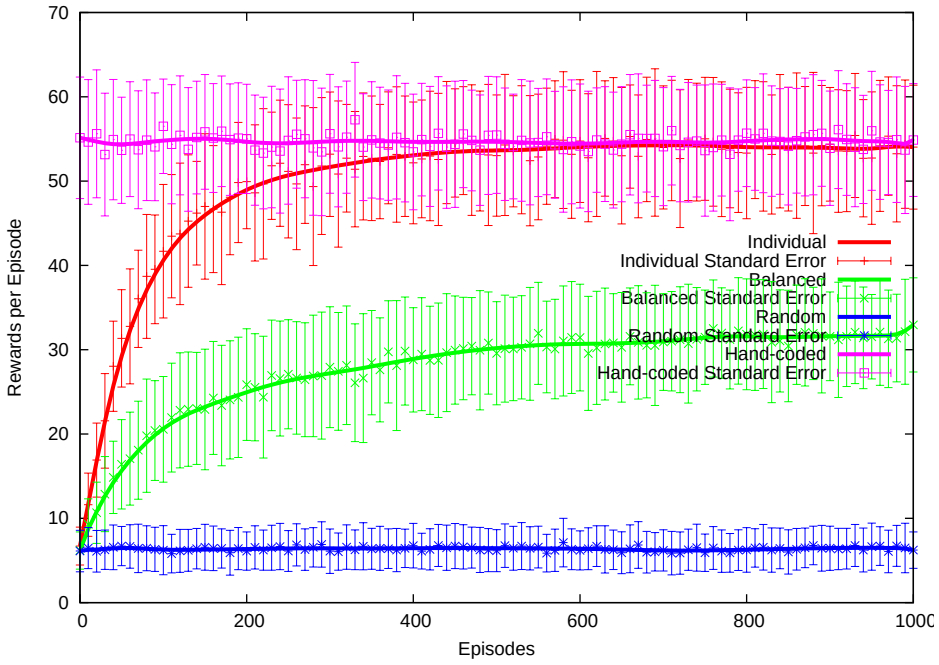


Figure 6.32: Significance of the performance of two groups of three agents each using tabular value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

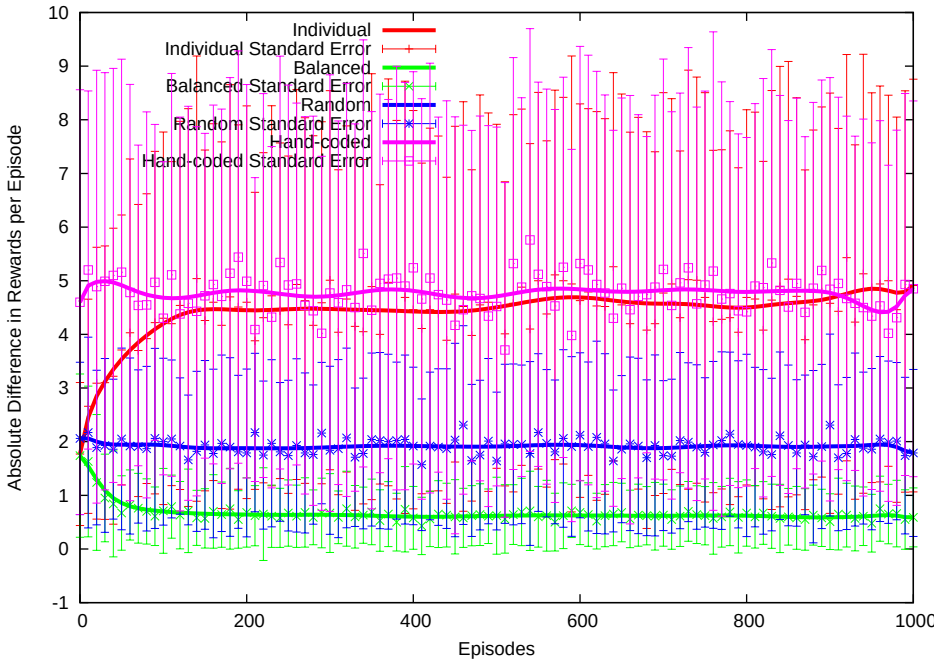


Figure 6.33: Significance of the absolute difference in the number of rewards picked up by two groups of three agents each using tabular value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

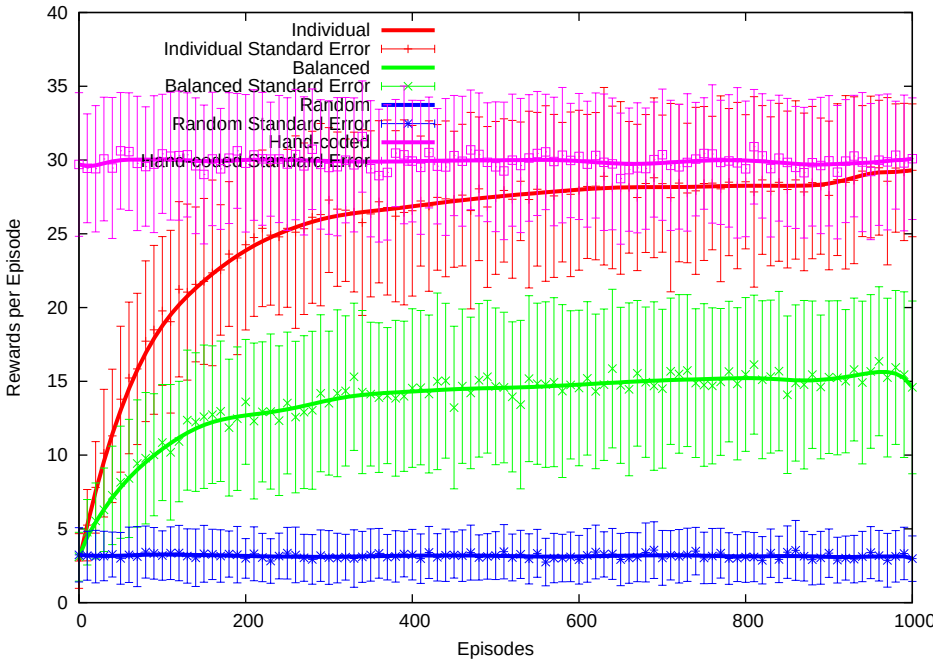


Figure 6.34: Significance of the performance of a group of two agents and a single agent each using tabular value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

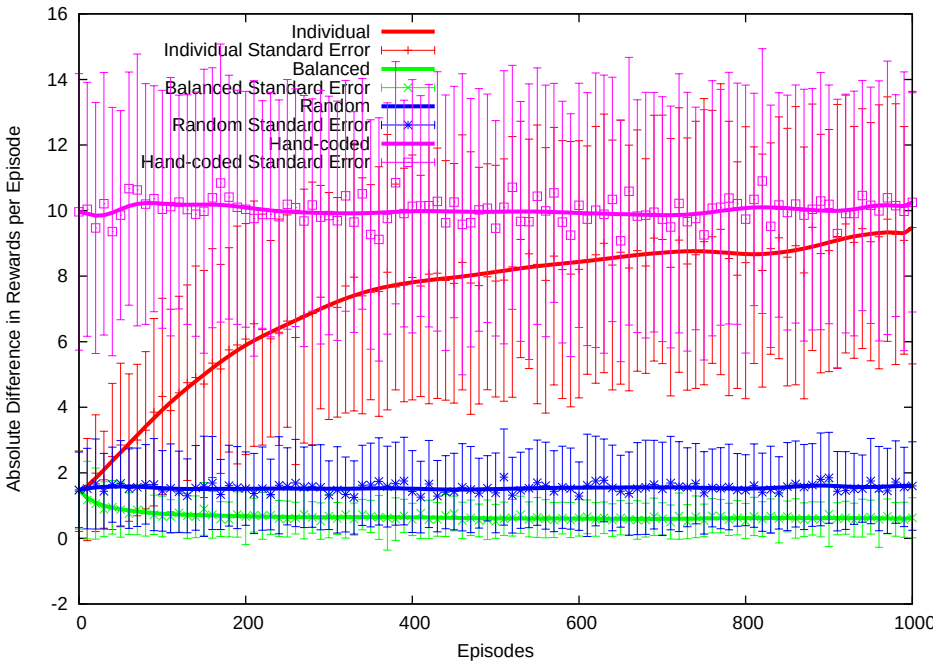


Figure 6.35: Significance of the absolute difference in the number of rewards picked up by a group of two agents and a single agent each using tabular value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

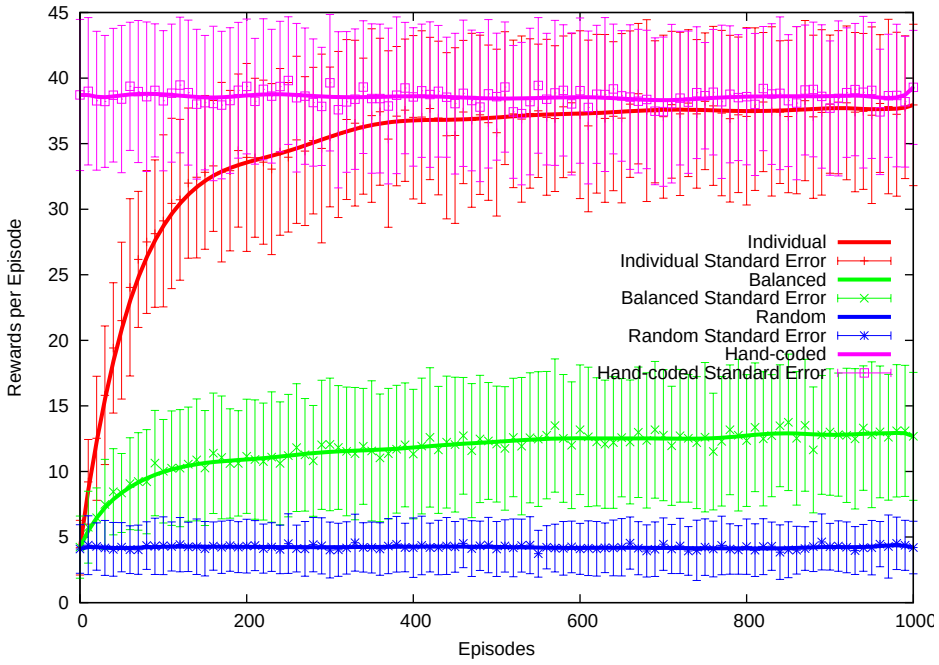


Figure 6.36: Significance of the performance of a group of three agents and a single agent each using tabular value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

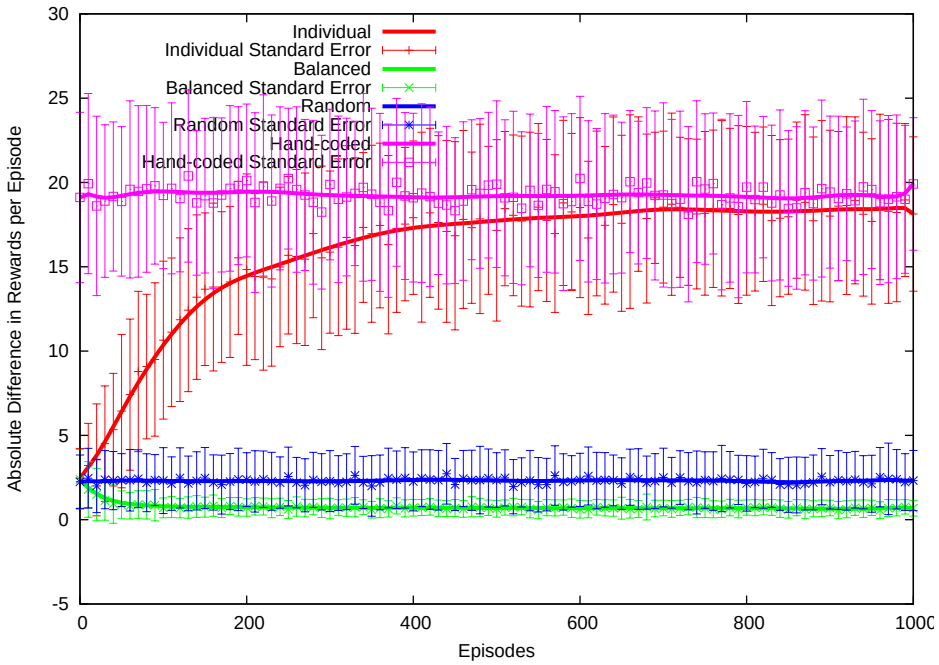


Figure 6.37: Significance of the absolute difference in the number of rewards picked up by a group of three agents and a single agent each using tabular value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

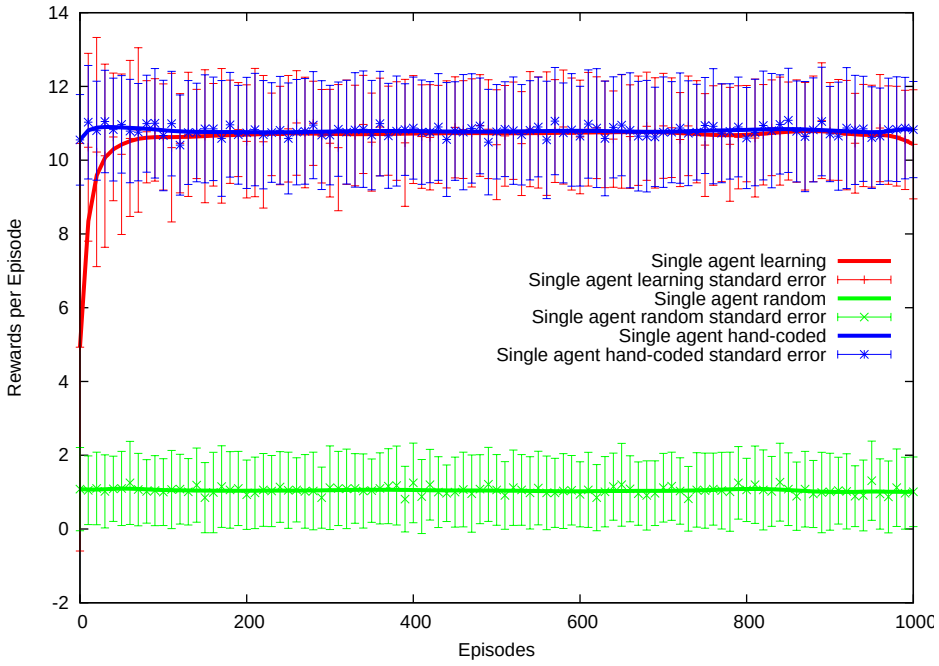


Figure 6.38: Significance of the performance of a single agent using an approximated value function. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

The graphs in Figures 6.27 through to 6.48 show the standard deviations at each sampled point for four different policies: random, hand-coded to move towards the nearest reward, learn individually and learn to have a balanced performance. The fixed random policy shows that the learning policies always do significantly better than taking random actions. The fixed hand-coded policy which always chooses to move towards the nearest reward shows two useful facts: the optimal or near-optimal performance for the number of rewards picked up per episode and, the baseline standard deviation across repeated experiments inherent in the task due to the random placement of the agents and rewards. When unconstrained by the balancing group utility function the agents and/ or groups learn to perform as well as the hand-coded policy showing that the learning algorithm and experiment environment is sound and that optimal or near optimal asymptotic performance is achieved. For the results to be considered convincing, the standard deviations plotted should be approximately the same height or smaller as those for the fixed, hand-coded policy. One can see that in all these graphs the learned policies (i.e. learning individually and learning a balanced behaviour) show similar standard deviations to those of the hand-coded (move towards nearest reward)

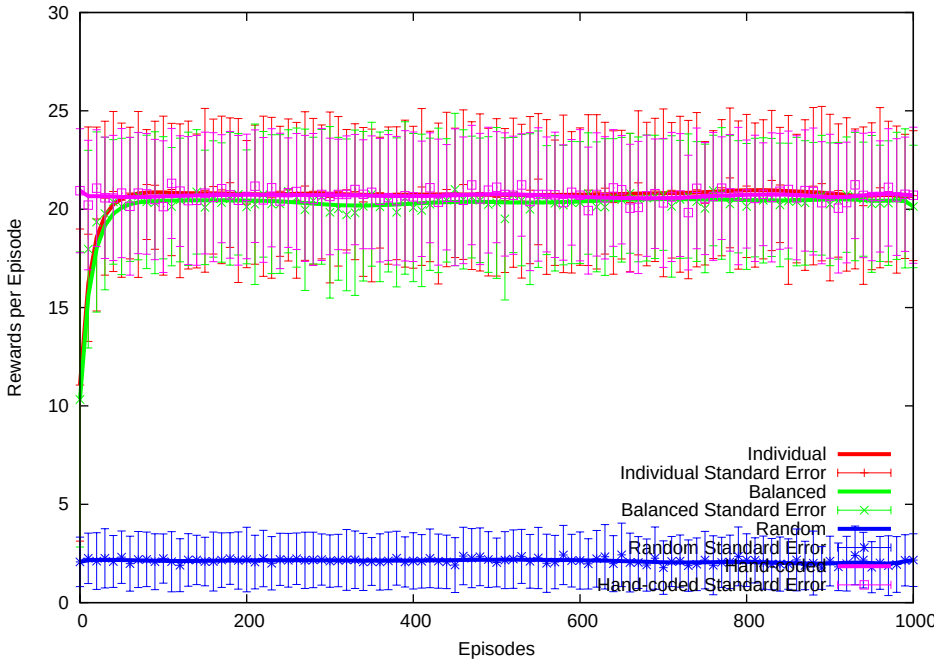


Figure 6.39: Significance of the performance of two agents using approximated value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

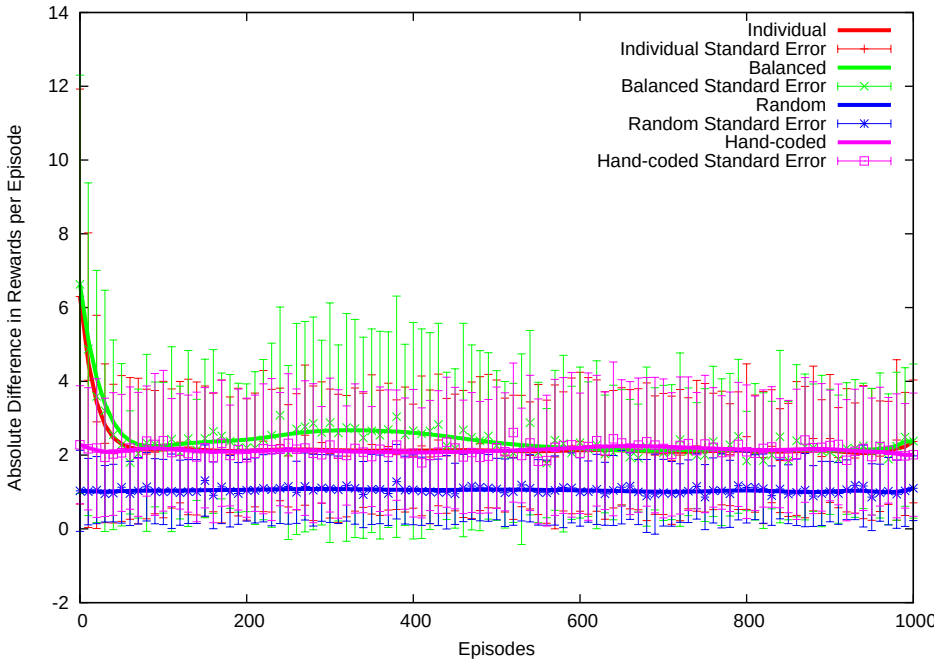


Figure 6.40: Significance of the absolute difference in the number of rewards picked up by two agents using approximated value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

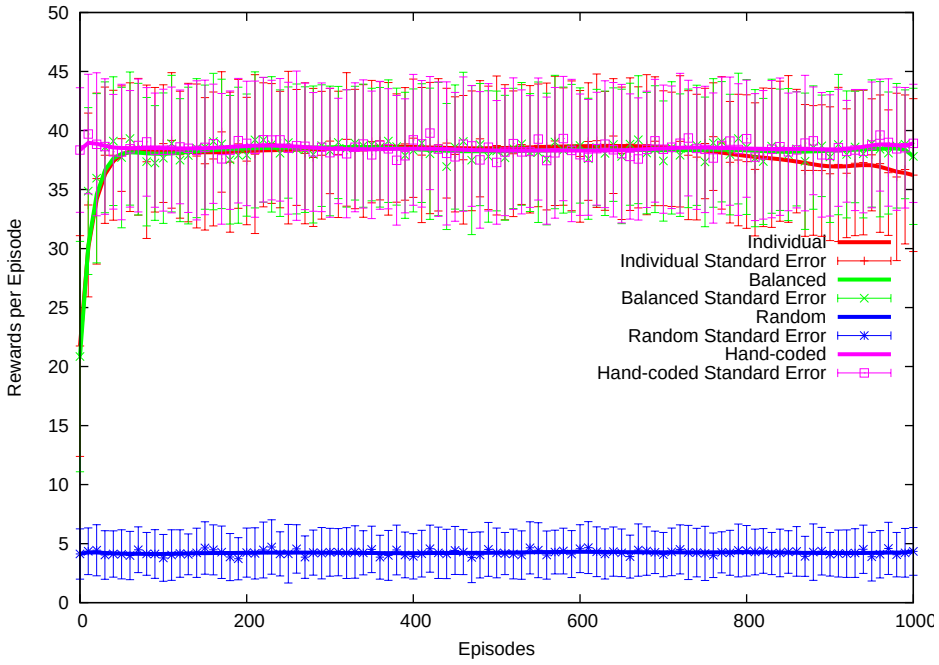


Figure 6.41: Significance of the performance of two groups of two agents each using approximated value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

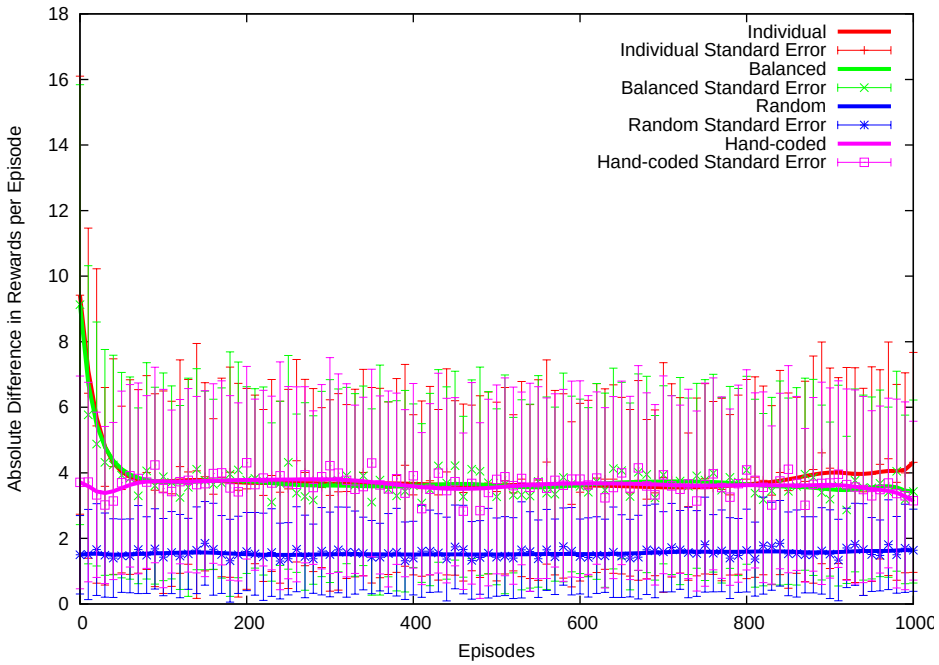


Figure 6.42: Significance of the absolute difference in the number of rewards picked up by two groups of two agents each using approximated value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

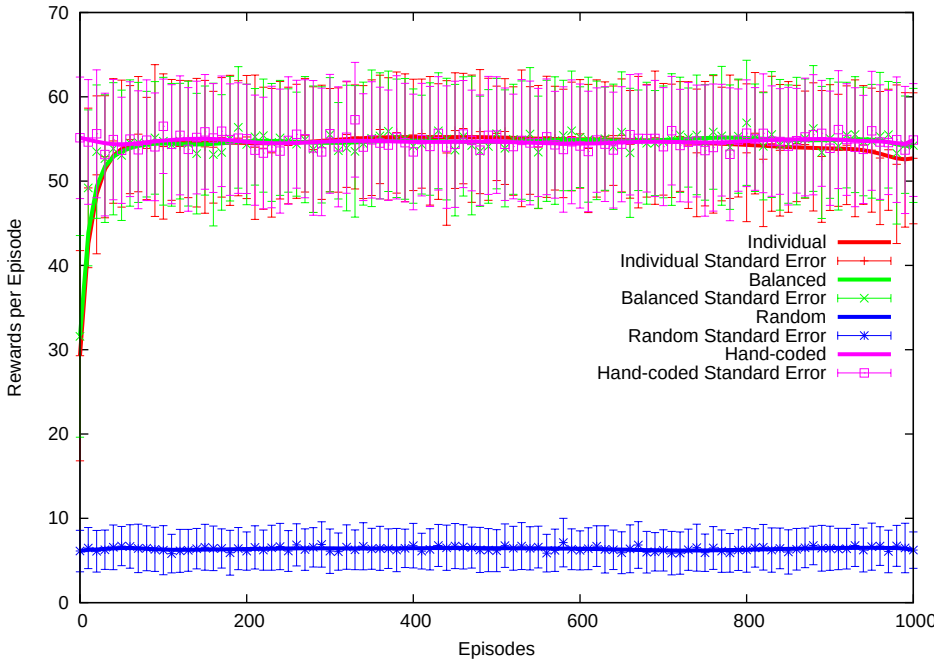


Figure 6.43: Significance of the performance of two groups of three agents each using approximated value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

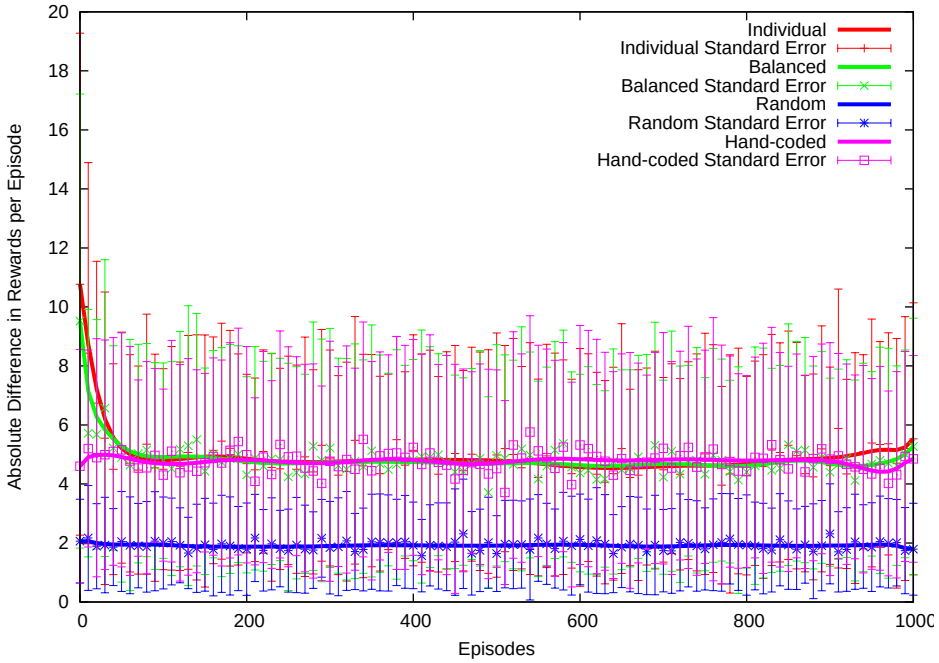


Figure 6.44: Significance of the absolute difference in the number of rewards picked up by two groups of three agents each using approximated value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

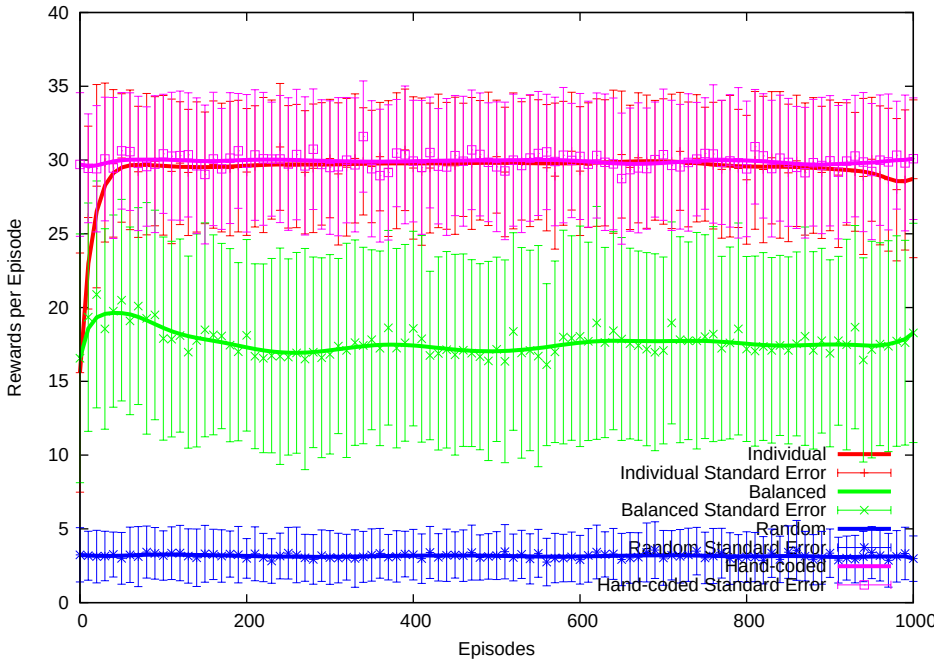


Figure 6.45: Significance of the performance of a group of two agents and a single agent each using approximated value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

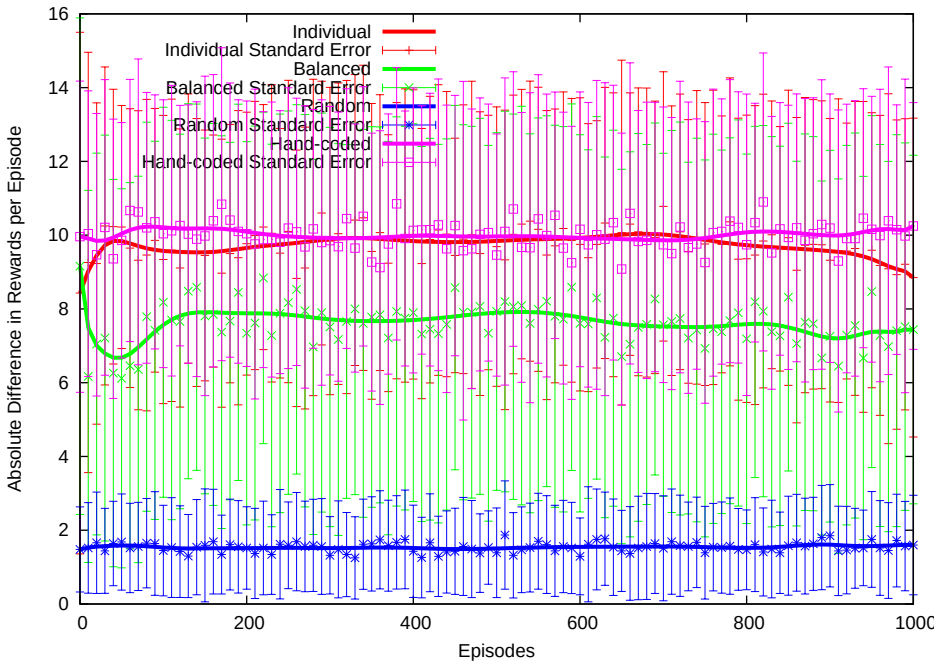


Figure 6.46: Significance of the absolute difference in the number of rewards picked up by a group of two agents and a single agent each using approximated value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

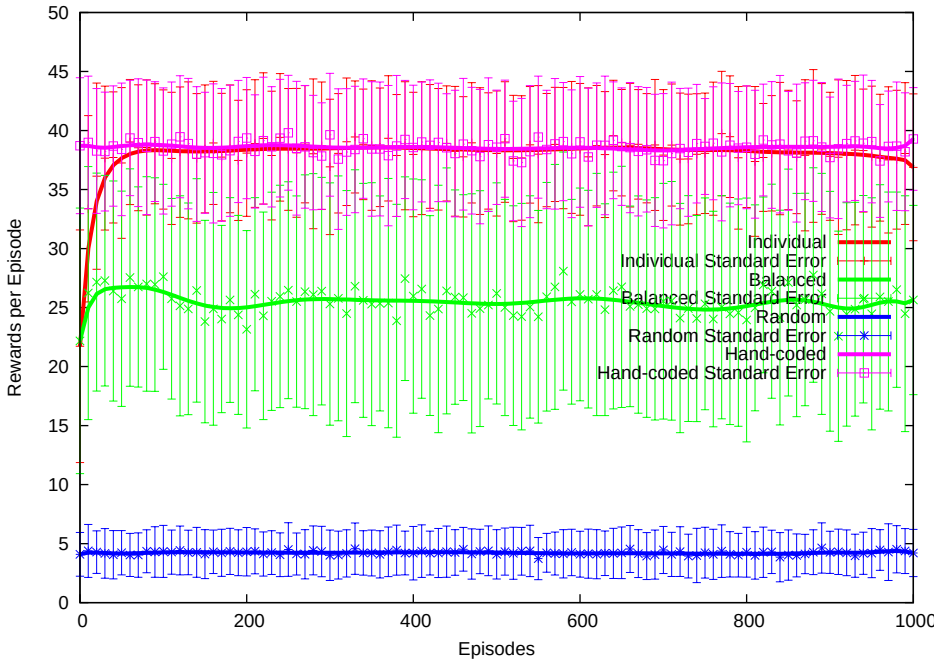


Figure 6.47: Significance of the performance of a group of three agents and a single agent each using approximated value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

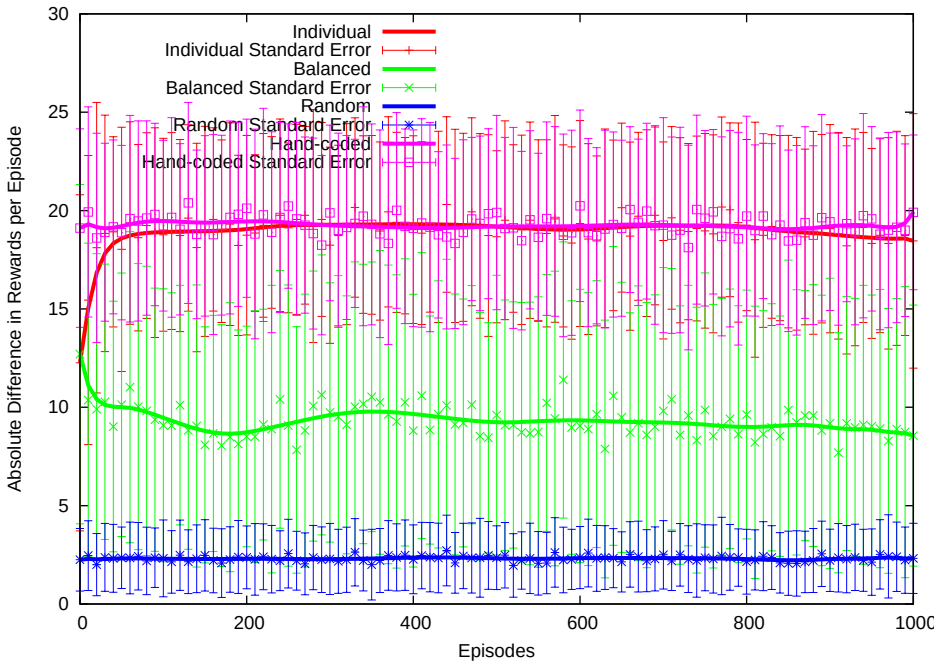


Figure 6.48: Significance of the absolute difference in the number of rewards picked up by a group of three agents and a single agent each using approximated value functions. Each curve is the average of one-hundred experiments. Standard deviations are shown for each curve.

policy.

Six t-tests were calculated to show the level of statistical significance for a sample of these experiments. These were:

- Two agents for tabular and approximated value functions with the null hypothesis that:
  - the learning agents pick up more rewards than the random policy:
    Tabular: $t = 39.78$ i.e. significant at the 0.001 confidence level.
    Approximated: $t = 51.27$ i.e. significant at the 0.001 confidence level.
  - when the agents were constrained by the balancing group utility function the absolute difference in number of rewards picked up was smaller than when unconstrained:
    Tabular: $t = 10.15$ i.e. significant at the 0.001 confidence level.
    Approximated: $t = 0.04$ i.e. *not* significant at the 0.05 confidence level (this is discussed below).
- One agent versus a group of three agents for tabular and approximated value functions with the null hypothesis that when the agent and group were constrained by the balancing group utility function the absolute difference in number of rewards picked up was smaller than when unconstrained:
  Tabular: $t = 37.72$ i.e. significant at the 0.001 confidence level.
  Approximated: $t = 10.63$ i.e. significant at the 0.001 confidence level.

Thus we can say with some confidence that these results are statistically significant. Three of the experimental setups when using function approximation (those for two agents, two groups of two agents and two groups of three agents) show differing behaviour to those experiments already shown earlier in this chapter. This is highlighted by the t-test that shows non-significant difference between the absolute difference in rewards picked up per episode. These experiments are discussed in Section 6.7.

6.6 Visualisation of Behaviour

The following section describes the behaviour of the agents during two of the learning experiments. The simple experiment graphed in Figure 4.5 (the red line for constant ε) was done to explore the problem domain and has one agent learning to pick up rewards with only the distance to the reward in its state representation. The trace on the left of

Table 6.1 show the simple random actions as the agent moves backwards and forwards. The trace on the right of the same table shows that the agent has learned to always move towards the reward. This sample traces of the agent's state and choice of action taken from the first (left) and last (right) measured episodes of a typical experiment run. The distance unit shown is 100 metres i.e. 0.03 is three metres from the nearest reward.

State	Action	State	Action
0.12	Move towards	0.18	Move towards
0.09	Move towards	0.15	Move towards
0.06	Move away	0.12	Move towards
0.09	Move away	0.09	Move towards
0.12	Move away	0.06	Move towards
0.15	Move towards	0.03	Move towards
0.12	Move towards	0.15	Move towards
0.09	Move away	0.12	Move towards
0.12	Move towards	0.09	Move towards
0.09	Move towards	0.06	Move towards
0.06	Move towards	0.03	Move towards

Table 6.1: Two short traces of an agent's state and actions taken from an experiment in Figure 4.5. The left side of the table is taken from the first assessed episode at the start of the experiment and the trace on the right side is taken from the last assessed episode.

A trace from a typical experiment of the type in Figure 5.20 is shown in Table 6.2. This is taken from the first assessed episode and so shows essentially random actions. Agent One and Two are in a group and agent Three is the individual agent. The state representation shows the distance to the nearest reward, the distance of the nearest agent to any reward, whether or not the agent is winning, losing or drawing against that agent and whether that agent is in the same team or not. Figure 6.49 shows a graphical representation of a snapshot of the three agents' policies during the final assessed episode. The trace for this figure is shown in Table 6.3 also. Agents One and Two do not move towards any rewards in particular. Agent Three is seen moving towards its nearest reward at each step as it is losing compared to the group of agents One and Two. This shows the expected, learned behaviour given such a situation.

Agent	State	Action
One	0.06, 0.09, Drawing, Other	Move towards
Two	0.33, 0.06, Drawing, Team-mate	Move away
Three	0.09, 0.06, Drawing, Other	Move away
Two	0.33, 0.06, Drawing, Team-mate	Move towards
Three	0.09, 0.06, Drawing, Other	Move towards
One	0.03, 0.12, Drawing, Other	Move away
Two	0.3, 0, Drawing, Team-mate	Move away
Three	0.06, 0.06, Drawing, Other	Move away
One	0.06, 0.06, Drawing, Other	Move towards
Two	0.33, 0.09, Drawing, Other	Move away
Three	0.09, 0.12, Drawing, Other	Move towards
Two	0.36, 0.12, Drawing, Team-mate	Move towards
One	0.03, 0.15, Drawing, Other	Move towards
One	0, 0.15, Drawing, Other	Move away
Two	0.33, 0.09, Winning, Other	Move towards

Table 6.2: A short trace of three agent's states and actions taken from an experiment as shown in Figure 5.20. The trace is taken from the first assessed episode at the start of an experiment.

Figure 6.49: Showing the learned policies of three agents arranged as in Figure 5.20. Rewards are shown as stars. Agents One (blue) and Two (green) are on the same team which has collectively picked up more reward than the Agent Three (red), the individual agent. The agents behaviours reflect the situation well. Agents One and Two are not making positive moves towards rewards as they are winning and have learned to maintain balance. Agent Three is clearing moving towards a reward. This trace is taken from the final assessed episode of an experiment. The trace for this visualisation is shown in Table 6.3.

Agent	State	Action
Two	0.27, 0.15, Winning, Other	Move away
One	0.48, 0.15, Winning, Other	Move away
Two	0.3, 0.12, Winning, Other	Move away
Three	0.12, 0.33, Losing, Other	Move towards
Two	0.33, 0.12, Winning, Other	Move away
One	0.54, 0.09, Winning, Other	Move towards
Three	0.09, 0.3, Losing, Other	Move towards
Two	0.3, 0.12, Winning, Other	Move away
Three	0.06, 0.27, Losing, Other	Move towards
One	0.51, 0.06, Winning, Other	Move towards
Three	0.03, 0.33, Losing, Other	Move towards
Two	0.33, 0.03, Winning, Other	Move away
One	0.48, 0.03, Winning, Other	Move away
One	0.51, 0.03, Winning, Other	Move away
Three	0, 0.3, Losing, Other	Move towards

Table 6.3: A short trace of three agent's states and actions taken from an experiment as shown in Figure 5.20. The trace is taken from the final assessed episode at the end of an experiment. Agent Three picks up a reward in the final row of the table. This trace is visualised in Figure 6.49.

6.7 Summary of Results

This section gives a concise overview of the results of the experimentation in Chapters 4, 5 and 6. Each sub-section here briefly describes an experimental configuration along with the results and comments about the significance of these results. All the comments relate to the performance of the balancing group utility function.

6.7.1 One agent

This was a simple experiment done to show that the learning algorithm and environment was sound and to determine expected baseline level of performance. The agent learns to perform optimally for tabular and approximated value functions (Figure 6.3). Figures 6.27 and 6.38 show that these results are statistically significant over one-hundred repeated experiments for tabular and approximated value functions respectively.

6.7.2 Two agents

The was an experiment (the first in this thesis) done to show how a group utility function, the balancing group utility function, performs. Figures 5.9 and 5.10 (tabular value functions) and Section 6.3.1 (approximated value functions) show that the agents lose some performance (in terms of number of rewards picked up per episode) and learn to reduce the imbalance in the number of rewards picked up as expected. Results showing the statistical significance for the tabular case are shown in Figures 6.28 and 6.29. The results showing statistical significance for approximated value functions (Figures 6.39 and 6.40) show poor performance (i.e. the balanced behaviour is not in evidence). This is true of the experiments run to test statistical significance for approximated value functions for three experimental configurations with one common factor: an even number of agents in each group. The failing configurations are for two single agents (this configuration), two groups of two agents (below) and two groups of three agents (further below). For these experimental configurations the approximated value functions fail to learn to pick up a near equal number of rewards. For these experiments the number of rewards picked up is also unaffected by the balancing group utility function. However, for these configurations, their equivalent tabular cases show good performance as seen in Chapter 5 so the issue lies in the way the value functions are approximated. Still for a single agent, a group of two agents and a single agent

and for a group of three agents and a single agent, the approximated value function experiments run over one-hundred iteration do show that the balancing group utility function works as shown earlier in this chapter. This failure in some of the approximated value function experiments is most likely due to the fact that the experimental configurations concerned are already naturally balanced and the effect of the balancing group utility function is small and is "lost" in the approximation method due to over-generalisation. For those experiments where the effect of the balancing group utility function is large (where the groups are inherently unbalanced) the repeated runs of one-hundred experiments are shown to work well.

6.7.3 Two groups of two agents each

This was an experiment done to show how well hierarchical group utility functions perform. Figures 5.13 and 5.14 (tabular value functions) and Figures 6.19 and 6.20 (approximated value functions) showed that the agents learned to reduce the imbalance by a small amount with a small reduction in the number of rewards picked up per episode. In terms of statistical significance, Figures 6.30 and 6.31 (tabular value functions) show the expected performance with good significance. Figures 6.41 and 6.42 (approximated value functions) show with significance that the balancing function has no discernible effect. The possible reason for this failure to learn the balancing part of the reward function has been discussed already above.

6.7.4 Two groups of three agents each

This was an experiment done to show how well hierarchical group utility functions perform with a larger number of agents. Figures 5.17 and 5.18 (tabular value functions) showed that the agents learned to reduce the imbalance by a small amount with a small reduction in the number of rewards picked up per episode. Figures 6.21 and 6.22 (approximated value functions) showed that the agents failed to learn the balancing part of the reward function as with two groups of two agents and two single agents above. In terms of statistical significance, Figures 6.32 and 6.33 (tabular value functions) show the expected performance with good significance again. Figures 6.43 and 6.44 (approximated value functions) show with significance that the balancing function has no discernible effect. The possible reason for this failure to learn the balancing part of the reward function has been discussed already above.

6.7.5 A group of two agents and a single agent

This was an experiment done to show how well hierarchical group utility functions perform with unequal groups. Figures 5.21 and 5.22 (tabular value functions) showed that the agents learned to reduce the imbalance with a reduction in the number of rewards picked up per episode as expected. Figures 6.23 and 6.24 (approximated value functions) showed that the agents learned to behave more equally than when unconstrained and that they lost some performance as expected. In terms of statistical significance, Figures 6.34 and 6.35 (tabular value functions) show the expected performance with good significance again. Figures 6.45 and 6.46 (approximated value functions) show, with significance, similar performance to that expected.

6.7.6 A group of three agents and a single agent

This was an experiment done to show how well hierarchical group utility functions perform with unequal groups again but this time with a larger inequality. Figures 5.25 and 5.26 (tabular value functions) showed that the agents learned to reduce the imbalance with a reduction in the number of rewards picked up per episode as expected. Figures 6.25 and 6.26 (approximated value functions) showed that the agents learned to behave more equally than when unconstrained and that they lost a lot of performance (number of rewards picked up per episode) as expected. In terms of statistical significance, Figures 6.36 and 6.37 (tabular value functions) and 6.47 and 6.48 (approximated value functions) show, with significance, similar performance to that expected.

The final chapter critiques the research presented in this thesis and addresses the main goals as listed in Section 1.4. Chapter 7 also places the research herein in the context of existing research and suggests some interesting research possibilities for further work.

Chapter 7

Conclusions and Further Work

This chapter summarises the contributions made by this thesis and places the research herein within the research areas of reinforcement learning and computational intelligence in computer games. There is a critique of the research within and then further interesting work is noted.

There is a desire for automated or semi-automated character behaviour specification and in reinforcement learning there is a possibility to meet the desire. It was achieved with rtNEAT (Section 2.3.2) with some significant changes to the way evolutionary methods are normally applied. Using hierarchies of group utility functions seems to be a good way to adapt reinforcement learning to better suit its application in computer games. There has not been a lot of published research on using reinforcement learning for computer games as noted in Chapter 2 and so this thesis can be considered a start, contributing empirical, practical experimentation in a real computer game engine. Many graphs are included in the appendices that show a search through the parameter space of both reinforcement learning and hierarchical group utility functions. This search needed to be done and there is probably much more useful empirical work to be done on applying reinforcement learning to computer games. Especially as the game featured in this thesis was nothing more than foraging albeit, implemented on top of a "real," commercial computer game engine. Using a real computer game engine was important from the outset of this thesis as very little academic research has been shown working in commercial computer game engines. Although using a computer game engine may not appear to have made a difference to the results, much time was spent overcoming the practical problems encountered in marrying the reinforcement algorithm to the game engine and in running experiments that may potentially have taken too long to be feasible. This was presented in Sections 4.2.2, 4.2.3, 4.2.5 and

4.2.6. Otherwise, using neural networks as the approximation method was a challenge that could have been avoided by using tile-based coding. Thinking ambitiously though, if these methods are to be used successfully in large scale computer worlds then such a powerful method as neural networks would need to be tried eventually. Some of the results in Chapter 6 were muddied because of the use of neural networks which introduced a secondary, parallel optimisation process. This could have been overcome by allowing more time for each experiment though the experiments included took a long time to run, or running experiments to search through several neural network topologies and parameters. Topologies were tested and re-tested several times prior to running the experiments to support this thesis but enough experiments to perform a reasonably thorough search through various topologies and parameters of suitable neural networks would have taken a substantially longer period of time to both run and analyse.

7.1 Contributions

- The main contribution of this thesis is the introduction of hierarchical group utility functions, a novel algorithm for reinforcement learning for learning qualitative relationships between agents and groups of agents whilst learning a quantitative main goal. The concept of group utility functions was derived from *programming by reward*, the team utility function and was implemented using sarsa, an existing reinforcement learning algorithm.

- To support this novel algorithm, many experiments were run in Torque, a commercial computer game engine, providing numerous empirical results with discourse showing and analysing group utility functions with tabular and approximated value functions.

- This thesis also provides a theoretical basis and some practical examples of using reinforcement learning for modern computer games.

- Finally this thesis provides empirical evidence that neural networks can be successfully used to approximate agents' value functions. This is despite being told that, "all bets are off" with respect to convergence when doing so (in Section 2.2.4.7). However, using a non-constant action-selection policy was shown not to work for some of the more complex experiments in Chapter 6.

7.2 Critique of Thesis

This thesis applied a novel learning algorithm, to a still reasonably novel domain for reinforcement learning using a difficult method of function approximation. This was a difficult and timely task and as such there were many ideas that were not followed through due to time constraints (Section 7.3) and there were many areas that could have been improved to make the research more valid.

Despite the use of a commercial computer game engine, the foraging task set to the agents was simplistic and not a realistic representation of the possible complexities involved in modern computer games. The foraging task was chosen to show clearly the effect of hierarchies of group utility functions. It would have been more convincing if the algorithm was shown learning a typical computer game tasks such as a sports game simulation or a tactical fighting game where the simple equilibrium might prove most useful. Reinforcement learning and neural networks are "black box" methods. One cannot see the effects they have by inspecting the values that are being optimised. Agent behaviour shows the effect of these optimisation methods and so it is difficult to start with a complex task considering the novelty and/ or difficulty elsewhere in the algorithm. It is probably best to leave the use of such algorithms to commercial computer game developers who may convinced of the need for them in future games production.

Group utility functions suffer from a poor signal-to-noise ratio as mentioned several times in this thesis. This is true because of the simplistic way the utility function is calculated just as in the team utility function. COIN (Section 2.3.3) offers a more complex way of calculating an agent's reward value which reduces the signal-to-noise ratio problem but does so at the cost of constraints on the way actions and states must be represented. More generally, there are methods used to speed or improve reinforcement learning such as hierarchical learning that could happily co-exist with group utility functions but were not used for the sake of simplicity, transparency and time.

Reinforcement learning would really only show its benefit for computer games when used with fiendishly difficult tasks where its search based optimisation approach could find unthought-of or difficult to concisely specify behaviour or when used to train a great number of characters. The experiments in Chapters 4, 5 and 6 fit neither of these cases. It remains to be seen how well hierarchical group utility functions would scale to large numbers of agents and groups. Presumably the signal-to-noise ratio would become more troublesome but Chapter 6 showed that the curse of dimensionality can

be overcome by using function approximation. The use of neural networks may be sufficient to both generalise experience to speed and broaden learning and to represent the true value function with enough clarity. This is the most difficult critique of this thesis to defend as it is hard to know how the methods involved would perform when scaled up. At least a start has been shown by using neural networks to successfully approximate value functions in Chapter 6 while mostly preserving the effect of group utility function hierarchies.

It is intriguing to see that the experiments using approximated value functions with an equal number of agents failed to learn the appropriate balanced behaviour (when tested for statistical significance in Section 6.7). Possible reasons have been given for this but it remains an open issue though the issue of instability when using non-linear methods of function approximation is known (Section 2.2.4.7).

7.3 Further Work

So little has been published on using reinforcement learning for game character behaviour specification that there are many avenues for future research.

Group utility functions represent arbitrary relationships between agents and/ or groups using arbitrary functions of the agents or groups collected reward. To illustrate this, in Section 3.4 a few such functions were shown alongside the balancing function used in the experiments of Chapters 5 and 6. Further work could investigate several functions other than using the negative standard deviation of the agents or groups collected reward.

Similarly, it would be enlightening to see how group utility functions work under more complex hierarchies. Simple extensions such as using three, four or many groups of agents under a balancing group utility function could be used rather than just two as in the experiments in Chapters 5 and 6. More ambitiously it would be interesting to investigate hierarchies of agents, groups and group utility functions where an agent belonged to more than one group at the same time or, where an agent switched groups under certain conditions. This could be used for characters with mixed allegiances which could be interesting for human game players.

In all of the experiments in this thesis each agent has had the same ability to affect the environment. Using agents with heterogeneous abilities brings even more benefits to using reinforcement learning for computer game characters. Using homogeneous agents is likely to lead to homogeneous characters although the different learning his-

tories will affect their behaviours making them somewhat individual. Using reinforcement learning with heterogeneous agents removes the need to alter the behaviour specification for each different type of agent. Reinforcement learning makes sure that each different type of agent behaves at least rationally given the agent's abilities and environment. It would be further interesting to investigate using reinforcement learning to train a *game conductor*.

A game conductor is a game agent that controls parts of the game other than the characters themselves. For example, in a sports game the game conductor would control the weather conditions. A game conductor is a novel idea but has some parallels with the idea of a real-time director for interactive drama [Magerko, 2002]. A game conductor's behaviour could be hand scripted just as with character specification but a game conductor could be included in the reinforcement learning process under a suitable hierarchy of group utility functions. The game conductor would then be a regular reinforcement learner but with greater powers to affect the game. Using these greater powers the game conductor could make it easier for the characters to achieve their goals. As an illustrative example, in the foraging task used in this thesis the rewards are placed randomly. The majority of the drop in performance when under the balancing group utility function was because the agents had to travel further or waste actions not picking up rewards to maintain balance. If the placement of rewards was the task of a game conductor then the game conductor could place the rewards in a pattern to help the agents maintain balance. This method of keeping balance might appear more natural to the observing human game player. Certainly if done with some subtlety the combination of character agents and a game conductor agent learning their behaviours from a hierarchy of group utility functions could be far more affective than just the agents without the help of a game conductor.

As with Icarus (Section 2.3.4), the use of a planner for computer game characters [Hawes, 2000] can simply allow designers to constrain the choices of characters for large sections of a game and still allow the agents to use reinforcement learning at certain choice points in the game. The use of a combined planner/ learning would be ideal for many modern computer games. The planner removes whole chunks of the state-action search space whilst still displaying some intelligent behaviour and the reinforcement learning algorithm can optimise the characters' decisions when it is deemed beneficial to the game.

7.3.1 Last Words

Currently and in the past, academic artificial intelligence has rarely crossed over successfully to game AI. Whilst "good old fashioned AI" such as the BDI structure in Black & White (Section 2.1.8) and the STRIPS planner in F.E.A.R. (Section 2.1.7) is being used it will be some time before evolutionary or learning methods become commonplace. Theses such as this one, growing numbers of senior academics running game research centres and projects such as NERO (Section 2.3.2) are showing the way. Reinforcement learning algorithms should be useful for automating game character behaviour specification for complex tasks and environments. However, practical work in reinforcement learning needs to be presented showing its use in computer games. This thesis goes some way to adapting reinforcement learning methods to, and showing their successful use in, computer games.

Bibliography

[ai-blog.net, 2007] ai-blog.net (2007). Thoughts on industry / academic collaboration. Date accessed: March 2008. http://www.ai-blog.net/archives/000135.html.

[Baekkelund, 2006] Baekkelund, C. (2006). A brief comparison of machine learning methods. In Rabin, S., editor, *AI Game Programming Wisdom 3*. Charles River Media.

[Barnes and Hutchens, 2002] Barnes, J. and Hutchens, J. (2002). Scripting for undefined circumstances. In *AI Game Programming Wisdom*. Charles River Media.

[Blizzard Entertainment, 2001] Blizzard Entertainment (2001). World of warcraft. Computer game.

[Blizzard Entertainment, 2009] Blizzard Entertainment (2009). World of warcraft. http://www.worldofwarcraft.com. Date accessed: March 2009.

[Bowling and Veloso, 2002] Bowling, M. and Veloso, M. (2002). Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136:215–250.

[Bowling and Veloso, 2003] Bowling, M. and Veloso, M. (2003). Simultaneous adversarial multi-robot learning. In *Proceedings of IJCAI'03*, pages 699–704.

[Bradley and Hayes, 2005a] Bradley, J. and Hayes, G. (2005a). Adapting reinforcement learning for computer games: Using group utility functions. In *IEEE Symposium on Computational Intelligence and Games*, pages 133–140. Available at http://homepages.inf.ed.ac.uk/s0128829/CIG2005.ps (Date accessed: September 2009).

[Bradley and Hayes, 2005b] Bradley, J. and Hayes, G. (2005b). Group utility functions: learning equilibria between groups of agents in computer games by modifying the reinforcement signal. In *IEEE Congress on Evolutionary Computation*.

Available at http://homepages.inf.ed.ac.uk/s0128829/CEC2005.ps (Date accessed: September 2009).

[Bragge and Storgrds, 2007] Bragge, J. and Storgrds, J. (2007). Profiling academic research on digital games using text mining tools. In *CD-proceedings of Digital Games Research Association (DiGRA) Conference*.

[Burrow and Lucas, 2009] Burrow, P. and Lucas, S. M. (2009). Evolution versus temporal difference learning for learning to play ms. pac-man. In *IEEE Symposium on Computational Intelligence and Games*. To appear.

[Campbell et al., 2001] Campbell, M., A. Joseph Hoane Jr., and Hsu, F. (2001). Deep blue. http://sjeng.org/ftp/deepblue.pdf.

[Champandard, 2007a] Champandard, A. (2007a). 10 Reasons the Age of Finite State Machines is Over. http://aigamedev.com/questions/fsm-age-is-over#more-230. Date accessed: July 2008.

[Champandard, 2007b] Champandard, A. (2007b). AI Game Dev. http://aigamedev.com/. Date accessed: August 2007.

[Champandard, 2007c] Champandard, A. (2007c). Assaulting F.E.A.R.'s AI: 29 Tricks to Arm Your Game. http://aigamedev.com/reviews/fear-ai. Date accessed: February 2009.

[Champandard, 2007d] Champandard, A. (2007d). A tale of two tunnels: Industry and academia trying to connect. http://aigamedev.com/discussion/academia-industry-collaboration. Date accessed: January 2008.

[Champandard, 2007e] Champandard, A. (2007e). Top 10 most influential AI games. http://aigamedev.com/reviews/top-ai-games. Date accessed: April 2008.

[Chan et al., 2004] Chan, B., Denzinger, J., Gates, D., Loose, K., and Buchanan, J. (2004). Evolutionary behavior testing of commercial computer games. In *Proceedings CEC 2004, Portland, 2004, pp. 125-132.*

[Choi et al., 2004] Choi, D., Kaufman, M., Langley, P., Nejati, N., and Shapiro, D. (2004). An architecture for persistent reactive behavior. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 988–995, Washington, DC, USA. IEEE Computer Society.

[Cornelius et al., 2006] Cornelius, R., Stanley, K. O., and Miikkulainen, R. (2006). Constructing adaptive ai using knowledge-based neuroevolution. In Rabin, S., editor, *AI Game Programming Wisdom 3*. Charles River Media.

[Denzinger and Winder, 2005] Denzinger, J. and Winder, C. (2005). Combining coaching and learning to create cooperative character behavior. In *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05), Essex University, Colchester, Essex, UK, 4-6 April, 2005*, pages 78–85.

[Durnan, 2004] Durnan, T. (2004). Reinforcement learning. Teaching material. CSE571. Department of Computer Science and Engineering. University of Notre Dame. Available at: http://www.cse.nd.edu/courses/cse571/www/presentations/chap21.pdf.

[Dybsand, 2001] Dybsand, E. (2001). Game developers conference 2001: An AI perspective. Date accessed: September 2003. http://www.gamasutra.com/features/20010423/dybsand_01.htm.

[Dybsand, 2004] Dybsand, E. (2004). AI roundtable moderator's report. Date accessed: November 2004. http://www.gameai.com/cgdc04notes.dybsand.html.

[Evans, 2001] Evans, R. (2001). AI in games: A personal view. http://www.gameai.com/blackandwhite.html. No longer available. Assumed 2001 publish date. Retrieved from http://web.archive.org/web/20070827114228/http://www.gameai.com/blackandwhite.html. Date accessed: July 2007.

[Evans and Lamb, 2002] Evans, R. and Lamb, T. B. (2002). Gdc 2002: Social activities: Implementing wittgenstein. Date accessed: December 2004. http://www.gamasutra.com/features/20020424/ evans_01.ht. Originally published in the proceeding of Game Developers Conference 2002.

[Fairclough et al., 2001] Fairclough, C., Fagan, M., Namee, B. M., and Cunningham, P. (2001). Research directions for AI in computer games. In *Proceedings of the Twelfth Irish Conference on Artificial Intelligence and Cognitive Science*. Date accessed: September 2006. http://www.comp.dit.ie/bmacnamee/papers/ResearchDirectionsForGameAI.pdf.

[Ferguson et al., 2006] Ferguson, W., Diller, D., Leung, A., Benyo, B., and Foley, D. (2006). Behavior modeling in commercial games. http://seriousgames.bbn.com/behaviorauthoring/Behavior_Authoring_Games_Final_Report.pdf. Date accessed: August 2006.

[Garage Games, 2001] Garage Games (2001). Torque game engine. http://www.garagegames.com. Date accessed: November 2001.

[Gold, 2005] Gold, A. (2005). Academic AI and video games: A case study of incorporating innovative academic research into a video game prototype. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG'05)*, pages 141–148.

[Hawes, 2000] Hawes, N. (2000). Real-time goal-orientated behaviour for computer game agents. In *Game-On 2000, 1st International Conference on Intelligent Games and Simulation*, pages 71–75.

[Healing, 2005] Healing, A. (2005). Collective intelligence with agents which have differing preferences. 4th Year Undergraduate Project Report for Artificial Intelligence and Computer Science. Edinburgh University.

[Hoen and Bohte, 2003] Hoen, P. J. and Bohte, S. M. (2003). Collective intelligence with sequences of actions - coordinating actions in multi-agent systems. In *ECML*, pages 181–192.

[Houlette and Fu, 2003] Houlette, R. and Fu, D. (2003). The ultimate guide to FSMs in games. In *AI Game Programming Wisdom 2*. Charles River Media.

[Howland, 1999] Howland, G. (1999). Practical guide to building a complete game AI: Volume I. http://www.lupinegames.com/articles/prac_ai.html. Date accessed: February 2004.

[Isla et al., 2001] Isla, D., Burke, R. C., Downie, M., and Blumberg, B. (2001). A layered brain architecture for synthetic creatures. In *IJCAI*, pages 1051–1058.

[Kirby, 2004] Kirby, N. (2004). AI roundtable moderator's report 2004. http://www.gameai.com/cgdc04notes.kirby.html. Date accessed: January 2004.

[Konidaris and Hayes, 2005] Konidaris, G. D. and Hayes, G. M. (2005). An architecture for behavior-based reinforcement learning. *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems*, 13(1):5–32.

[Kosak, 2002] Kosak, D. (2002). What's this world coming to? The future of massively multiplayer games. http://archive.gamespy.com/gdc2002/mmog/. Date accessed: August 2004.

[Laird and van Lent, 2000] Laird, J. E. and van Lent, M. (2000). Human-level AI's killer application: Interactive computer games. http://ai.eecs.umich.edu/people/laird/papers/AAAI-00.pdf. Date accessed: February 2002.

[Laird and van Lent, 2005] Laird, J. E. and van Lent, M. (2005). Machine learning for computer games. Presentation at Game Developers Conference 2005. http://research.microsoft.com/en-us/collaboration/papers/machinelearningforcomputergames-gdc2005.pdf. Date accessed: January 2006.

[Langley et al., 2004] Langley, P., Cummings, K., and Shapiro, D. (2004). Hierarchical skills and cognitive architectures. In *Proceedings of the Twenty-Sixth Annual Conference of the Cognitive Science Society*, pages 779–784.

[Laramée, 2002] Laramée, F. D. (2002). Genetic algorithms: Evolving the perfect troll. In *AI Game Programming Wisdom*. Charles River Media.

[Lee-Urban et al., 2008] Lee-Urban, S., Smith, M., and noz Avila, H. M. (2008). Learning winning policies in team-based first-person shooter games. In *AI Game Programming Wisdom 4*. Charles River Media.

[Lionhead Studios, 2001] Lionhead Studios (2001). Black and White. http://blackandwhite.ea.com. Date accessed: November 2002.

[Lucas, 2009] Lucas, S. M. (2009). Computational intelligence and AI in games: a new IEEE transactions. Editorial, IEEE Transactions on Computational Intelligence and AI in Games. Available from: http://csee.essex.ac.uk/staff/lucas/tciaig/editorial.pdf.

[Magerko, 2002] Magerko, B. (2002). A proposal for an interactive drama architecture. In *In Artificial Intelligence and Interactive Entertainment: Papers from the 2002 AAAI Spring Symposium*, Menlo Park, CA, USA.

[Manslow, 2002] Manslow, J. (2002). Learning and adaptation in games. In *AI Game Programming Wisdom*. Charles River Media.

[Manslow, 2003] Manslow, J. (2003). Using reinforcement learning to solve ai control problems. In *AI Game Programming Wisdom 2*. Charles River Media.

[Manslow, 2006] Manslow, J. (2006). Practical algorithms for in-game learning. In Rabin, S., editor, *AI Game Programming Wisdom 3*. Charles River Media.

[McIlroy, 2008] McIlroy, T. (2008). The future of computer gaming. http://www.thefutureofpublishing.com/industries/the_future_of_computer_gaming.html. Date accessed: October 2008.

[McPartland, 2008] McPartland, M. (2008). A practical guide to reinforcement learning in first-person shooters. In *AI Game Programming Wisdom 4*. Charles River Media.

[Midway Games, 1992] Midway Games (1992). Mortal kombat. Computer game.

[Molyneux, 2001] Molyneux, P. (2001). Postmortem: Lionhead Studios' Black & White. http://www.gamasutra.com /features/20010613/molyneux_01.htm. Date accessed: November 2002.

[Monolith Productions, 2002] Monolith Productions (2002). No One Lives Forever 2: A Spy in H.A.R.M.'s Way. http://nolf2.sierra.com. Date accessed: January 2005.

[Niederberger and Gross, 2002] Niederberger, C. and Gross, M. H. (2002). Towards a game agent. Technical Report 377, Institute of Scientific Computing, ETH Zürich.

[Nowé et al., 2001] Nowé, A., Verbeeck, K., and Lenaerts, T. (2001). Learning agents in a homo egualis society. In *Proceedings of Learning Agents Workshop at Agents 2001 Conference*.

[Orkin, 2003] Orkin, J. (2003). Applying goal-oriented action planning to games. In *AI Game Programming Wisdom 2*.

[Orkin, 2004] Orkin, J. (2004). Symbolic representation of game world state: Toward real-time planning in games. Technical report, Proceedings of the AAAI Workshop on Challenges in Game AI.

[Orkin, 2006] Orkin, J. (2006). 3 States and a Plan: The AI of F.E.A.R. In *In the proceedings of the Game Developer's Conference*. CMP Game Group. Available at http://web.media.mit.edu/ jorkin/gdc2006_orkin_jeff_fear.doc. Date accessed: October 2006.

[Pottinger and Laird, 2000] Pottinger, D. C. and Laird, J. E. (2000). Game AI: The state of the industry, part two. http://www.gamasutra.com/features/20001108/laird_01.htm. Date accessed: December 2002.

[Rabin, 2002] Rabin, S., editor (2002). *AI Game Programming Wisdom*. Charles River Media.

[Rabin, 2003a] Rabin, S., editor (2003a). *AI Game Programming Wisdom 2*. Charles River Media.

[Rabin, 2003b] Rabin, S. (2003b). Common game AI techniques. In *AI Game Programming Wisdom 2*.

[Rabin, 2003c] Rabin, S. (2003c). Promising game AI techniques. In *AI Game Programming Wisdom 2*.

[Rabin, 2006] Rabin, S., editor (2006). *AI Game Programming Wisdom 3*. Charles River Media.

[Rabin, 2008] Rabin, S., editor (2008). *AI Game Programming Wisdom 4*. Charles River Media.

[Reynolds, 2002] Reynolds, J. (2002). Tactical team AI using a command hierarchy. In *AI Game Programming Wisdom*.

[Reynolds, 2003] Reynolds, J. (2003). Team member AI in an FPS. In *AI Game Programming Wisdom 2*.

[Rushe, 2007] Rushe, D. (2007). Video games grow up. The Sunday Times newspaper, Business section. December 9. http://business.timesonline.co.uk/tol/business/industry_sectors/technology/article3021342.ece.

[Sarle, 2009] Sarle, W. S. (2009). AI FAQ/Neural Nets. http://www.faqs.org/faqs/ai-faq/neural-nets/. Date accessed: July 2005.

[Shapiro and Langley, 1999] Shapiro, D. and Langley, P. (1999). Controlling physical agents through reactive logic programming. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 386–387, New York, NY, USA. ACM.

[Shapiro et al., 2001] Shapiro, D., Langley, P., and Shachter, R. (2001). Using background knowledge to speed reinforcement learning in physical agents. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 254–261, New York, NY, USA. ACM.

[Shapiro and Langley, 2002] Shapiro, D. G. and Langley, P. (2002). Separating skills from preference: Using learning to program by reward. In *ICML '02: Proceedings of the Nineteenth International Conference on Machine Learning*, pages 570–577, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Stanley et al., 2005] Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2005). Evolving neural network agents in the nero video game. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG'05)*, pages 182–189.

[SteveMcLean, 2003] SteveMcLean (2003). Multi-agent cooperation using trickle down utility. Master's thesis, Edinburgh University.

[Stone and Sutton, 2001] Stone, P. and Sutton, R. S. (2001). Scaling reinforcement learning toward robocup soccer. In *ICML '01: Proceedings of the Eighteenth International Conference on Machine Learning*, pages 537–544, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Studios, 2002] Studios, L. (2002). http://www.lionhead.com. Date accessed: November 2002.

[Sutton, 2001] Sutton, R. (2001). Reinforcement learning FAQ: Frequently asked questions about reinforcement learning. http://www.cs.ualberta.ca/ sutton/RL-FAQ.html#backpropagation. Date accessed: June 2003.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

[Tesauro, 2002] Tesauro, G. (2002). Multi-agent learning mini-tutorial. Multi-Agent Learning: Theory and Practice Workshop at NIPS2002. http://www.cs.rutgers.edu/%7Emlittman/topics/nips02/nips02/tesauro.ppt. Date accessed: December 2004.

[Togelius, 2007] Togelius, J. (2007). On industrial-academic collaboration in game ai. http://togelius.blogspot.com/2007/12/on-industrial-acadmic-collaboration-in.html. Date accessed: January 2008.

[Tumer and Wolpert, 2000] Tumer, K. and Wolpert, D. (2000). Collective intelligence and braess' paradox. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 104–109. AAAI Press / The MIT Press.

[United Business Media, 2001] United Business Media (2001). Gamasutra. Date accessed: March 2001 (Formerly published by CMP Game Group).

[University of Texas at Austin, 2009] University of Texas at Austin (2009). Neuroevolving robotic operatives. nerogame.org. Date accessed: 2009.

[van der Sterren, 2002a] van der Sterren, W. (2002a). Squad tactics: Planned maneuvers. In EDITOR, editor, *AI Game Programming Wisdom*. Charles River Media.

[van der Sterren, 2002b] van der Sterren, W. (2002b). Squad tactics: Team AI and emergent maneuvers. In *AI Game Programming Wisdom*. Charles River Media.

[Wallace, 2003] Wallace, N. (2003). Hierarchical planning in dynamic worlds. In *AI Game Programming Wisdom 2*. Charles River Media.

[Wardrip-Fruin, 2007] Wardrip-Fruin, N. (2007). On academic and game industry ai. http://grandtextauto.org/2007/12/17/on-academic-and-game-industry-ai/. Date accessed: January 2008.

[Wexler, 2002] Wexler, J. (2002). Artificial intelligence in games: A look at the smarts behind Lionhead Studios "Black and White" and where it can and will go in the future. Unpublished. Available at http://www.cs.rochester.edu/u/brown/242/assts/termprojs/games.pdf. Date accessed: September 2005.

[Wolpert et al., 1999] Wolpert, D., New, M. H., and Bell, A. M. (1999). Distorting reward functions to improve reinforcement learning. Technical report, NASA-ARC-IC-99-71.

[Wolpert and Tumer, 2000] Wolpert, D. and Tumer, K. (2000). An introduction to collective intelligence. Technical Report ARC-IC-99-63, NASA.

[Wolpert and Tumer, 2001] Wolpert, D. and Tumer, K. (2001). Optimal payoff functions for members of collectives. *Advances in Complex Systems*, 4(2–3):265–279.

[Wolpert and Tumer, 2002a] Wolpert, D. and Tumer, K. (2002a). Collective intelligence. http://ic.arc.nasa.gov/projects/COIN/.

[Wolpert et al., 2000a] Wolpert, D. H., Kirshner, S., Merz, C. J., and Tumer, K. (2000a). Adaptivity in agent-based routing for data networks. In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*. ACM.

[Wolpert and Lawson, 2002] Wolpert, D. H. and Lawson, J. W. (2002). Designing agent collectives for systems with markovian dynamics. In Castelfranchi, C. and Johnson, W. L., editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 3, pages 1066–1073.

[Wolpert et al., 2001] Wolpert, D. H., Sill, J., and Tumer, K. (2001). Reinforcement learning in distributed domains: Beyond team games. In *In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*.

[Wolpert and Tumer, 2002b] Wolpert, D. H. and Tumer, K. (2002b). Collective intelligence, data routing and braess' paradox. *Journal of Artificial Intelligence Research*, 16:708–714.

[Wolpert et al., 2000b] Wolpert, D. H., Wheeler, K. R., and Tumer, K. (2000b). Collective intelligence for control of distributed dynamical systems. *Europhysics Letters*, 49:708–714.

[Woodcock, 2000] Woodcock, S. (2000). Game AI: The state of the industry. http://www.gamasutra.com /features/20001101/woodcock.htm. Date accessed: December 2003.

[Woodcock, 2002] Woodcock, S. (2002). Recognizing strategic dispositions: Engaging the enemy. In *AI Game Programming Wisdom*. Charles River Media.

[Woodcock, 2003] Woodcock, S. (2003). AI roundtable moderator's report. http://www.gameai.com. Date accessed: January 2004.

[Yannakakis, 2005] Yannakakis, G. N. (2005). *AI in Computer Games: Generating Interesting Interactive Opponents by the use of Evolutionary Computation*. PhD thesis, University of Edinburgh.

[Zubek, 2006] Zubek, R. (2006). Introduction to hidden markov models. In Rabin, S., editor, *AI Game Programming Wisdom 3*. Charles River Media.

Appendices

Appendix A

All Experimental Graphs with Tabular Value Functions

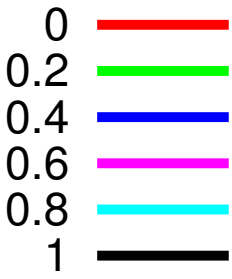
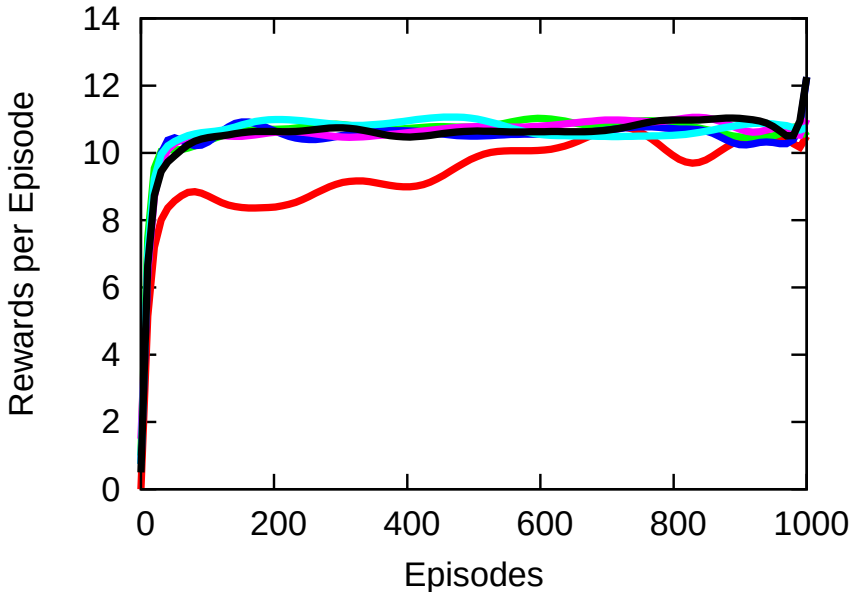
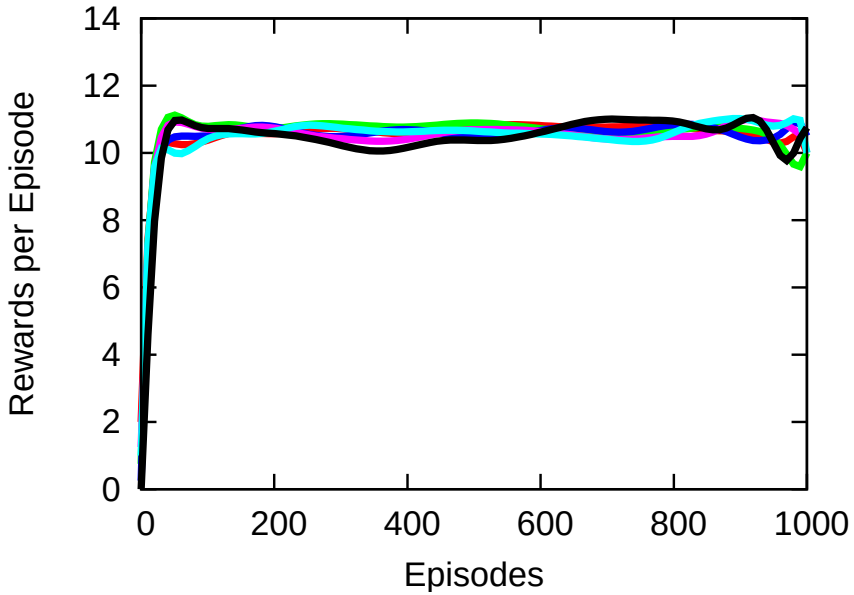


Figure A.1: The legend for each graph in the appendices. The colours represent different values of ε.

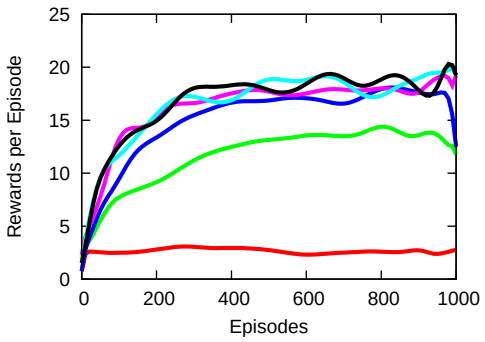


(a) Constant ε values.

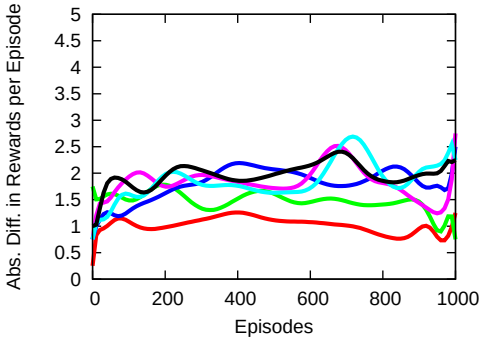


(b) Degrading ε values.

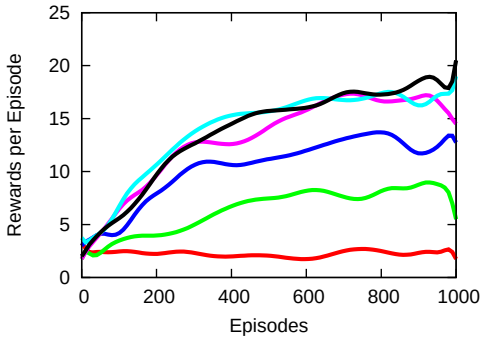
Figure A.2: Tabular value functions - An individual agent, one nearest reward in the state representation.



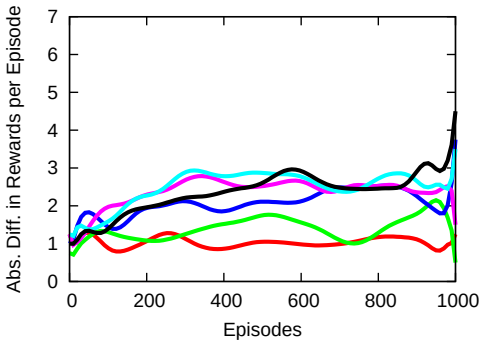
(a) Closest agent to any reward.



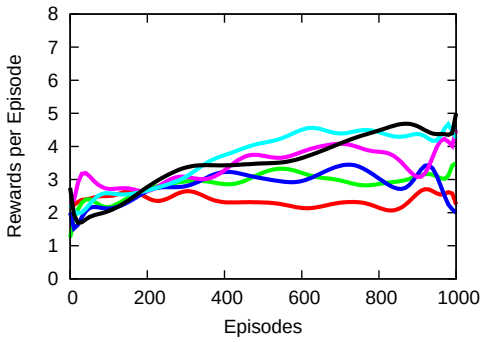
(b) Closest agent to any reward.



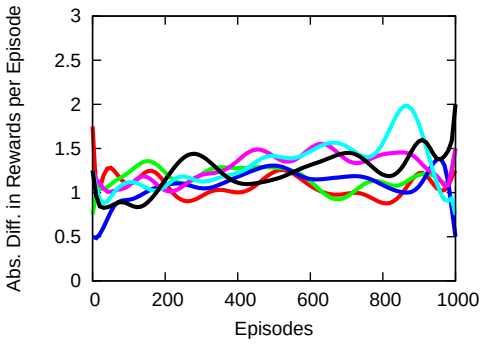
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

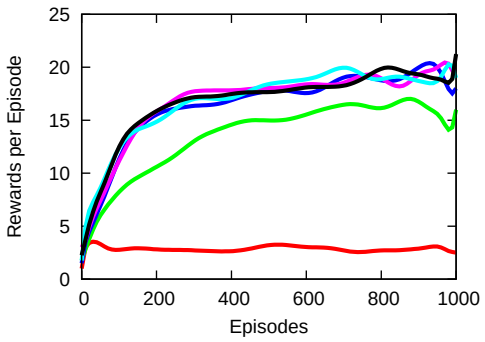


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

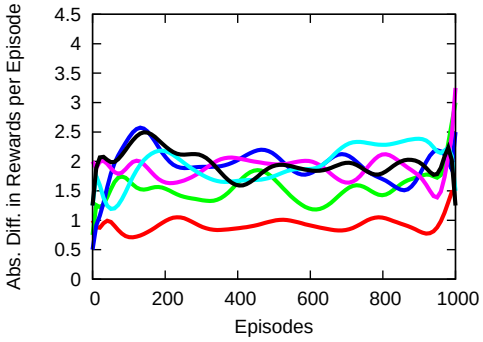


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

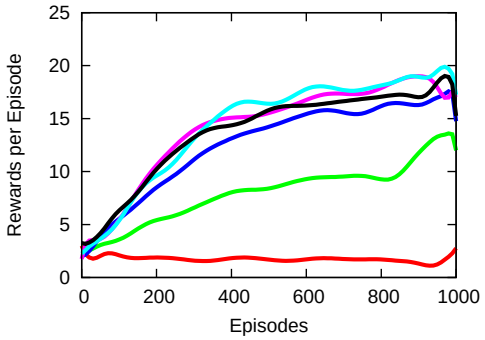
Figure A.3: Tabular value functions - Two individual agents, one nearest reward in the state representation with constant ε values.



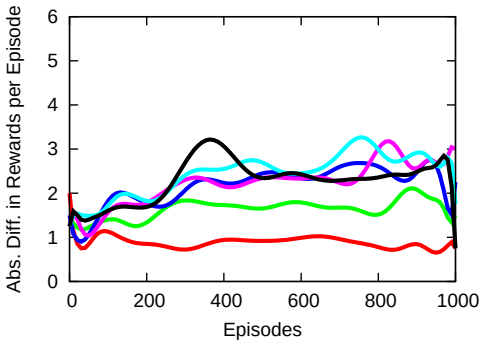
(a) Closest agent to any reward.



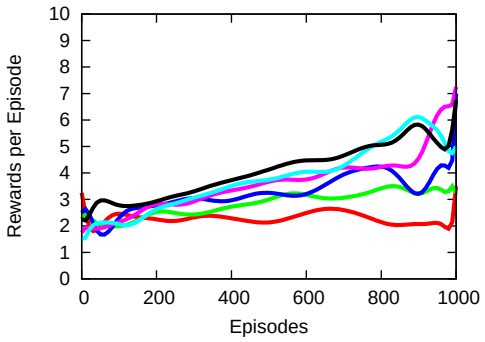
(b) Closest agent to any reward.



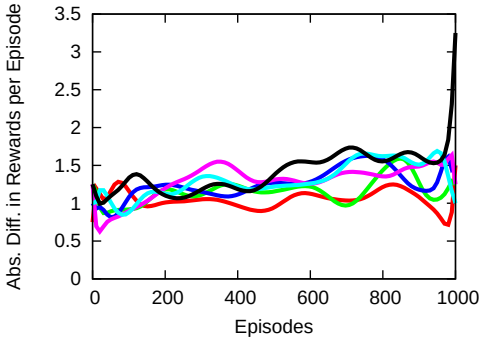
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

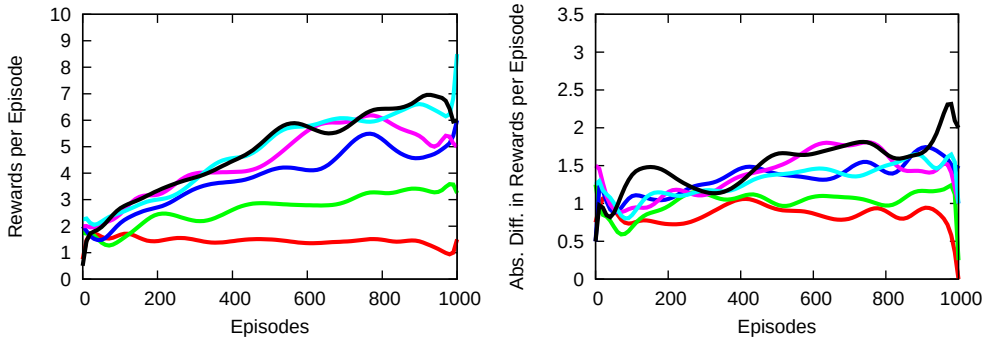


(e) Closest other agent to agent's nearest reward and closest agent to any reward.



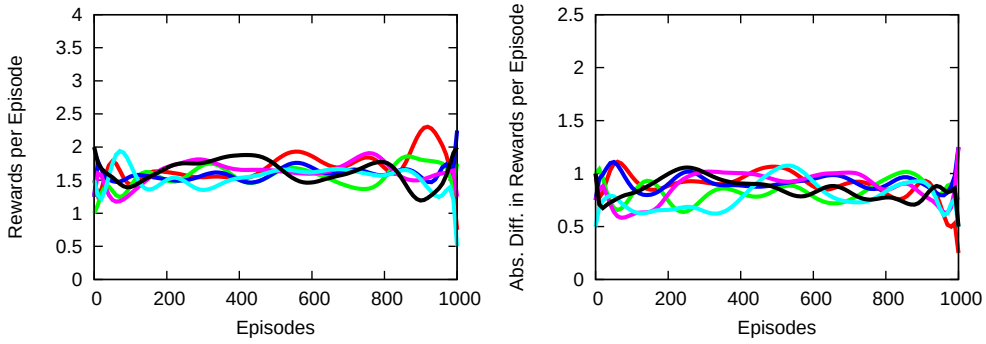
(f) Closest other agent to agent's nearest reward and closest agent to any reward.

Figure A.4: Tabular value functions - Two individual agents, one nearest reward in the state representation with degrading ε values.



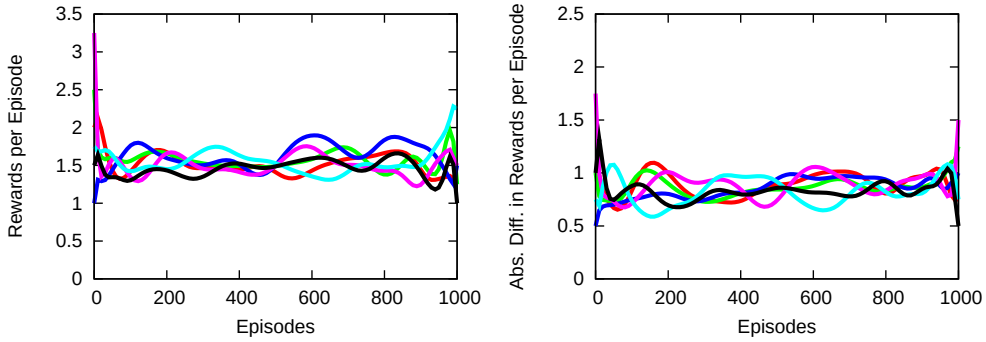
(a) Closest agent to any reward.

(b) Closest agent to any reward.



(c) Closest other agent to agent's nearest two rewards.

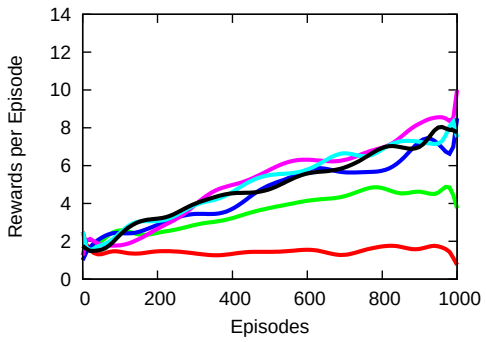
(d) Closest other agent to agent's nearest two rewards.



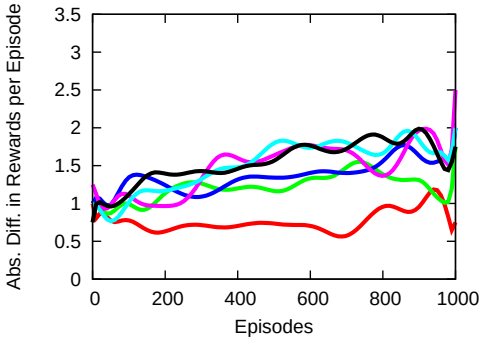
(e) Closest other agent to agent's nearest two rewards and closest agent to any reward.

(f) Closest other agent to agent's nearest two rewards and closest agent to any reward.

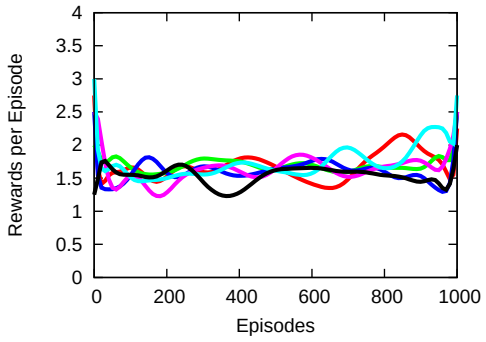
Figure A.5: Tabular value functions - Two individual agents, two nearest rewards in the state representation with constant ε values.



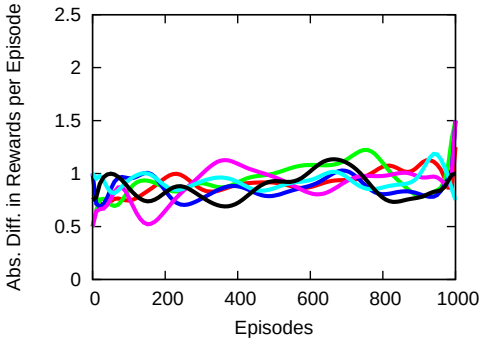
(a) Closest agent to any reward.



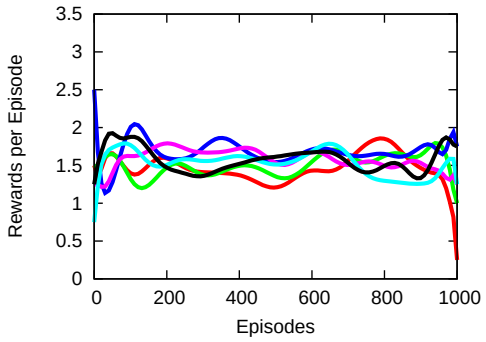
(b) Closest agent to any reward.



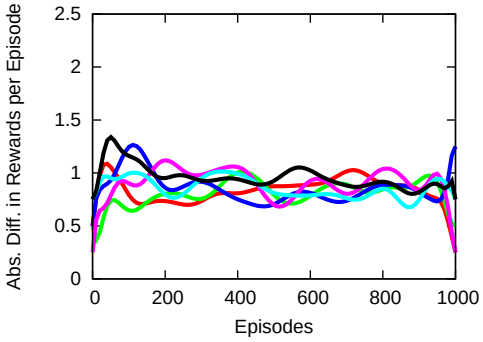
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

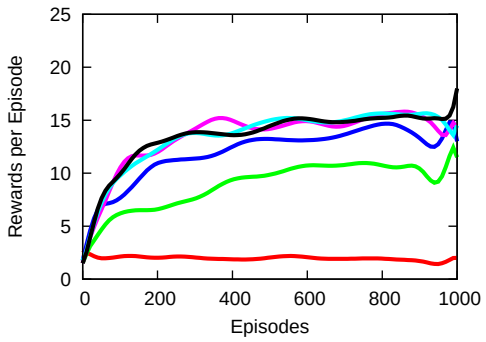


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

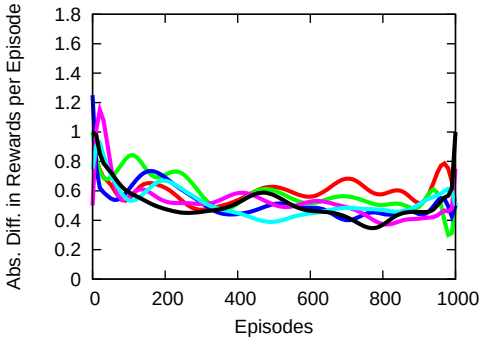


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

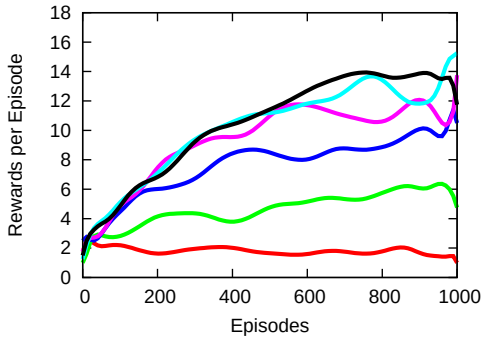
Figure A.6: Tabular value functions - Two individual agents, two nearest rewards in the state representation with degrading ε values.



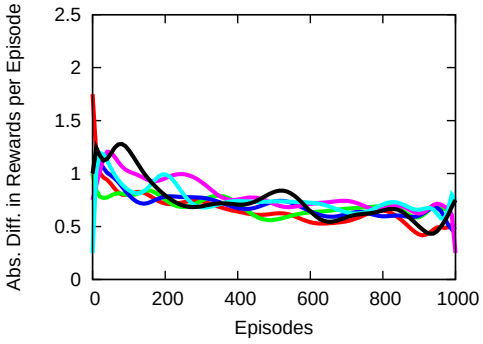
(a) Closest agent to any reward.



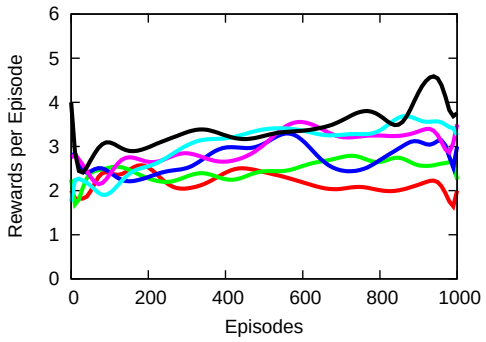
(b) Closest agent to any reward.



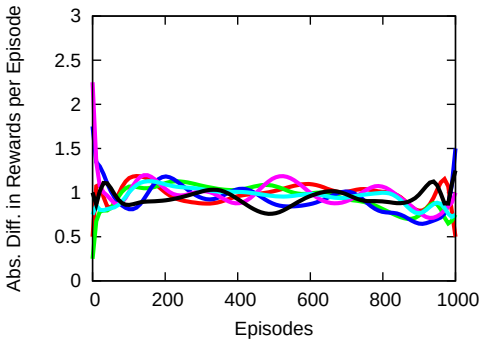
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

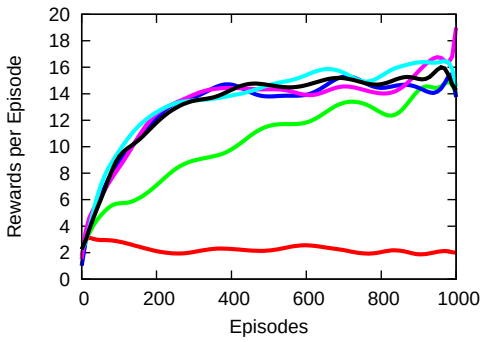


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

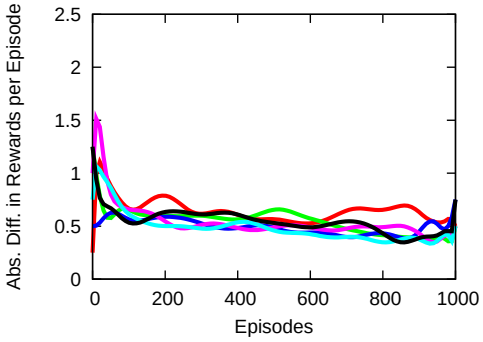


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

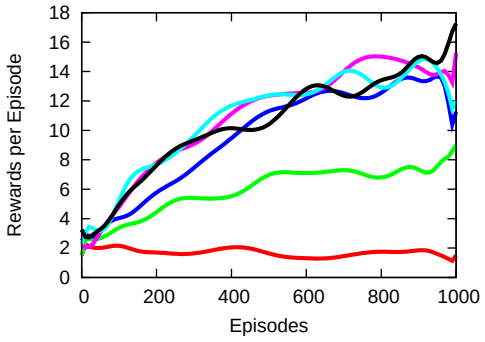
Figure A.7: Tabular value functions - Two balanced agents, one nearest reward in the state representation with constant ε values.



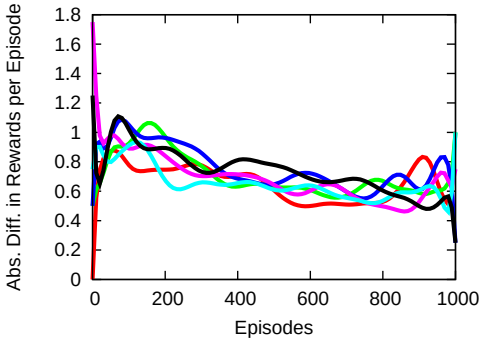
(a) Closest agent to any reward.



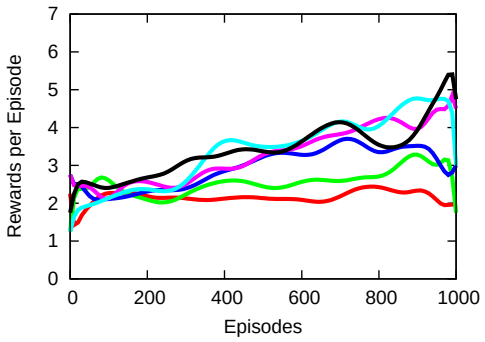
(b) Closest agent to any reward.



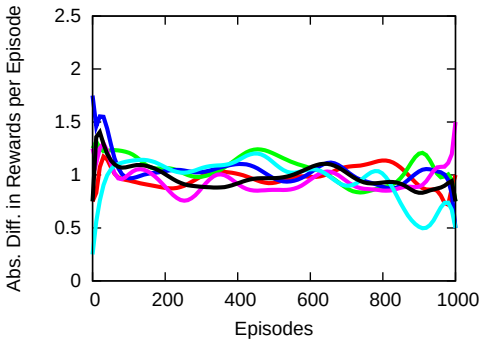
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

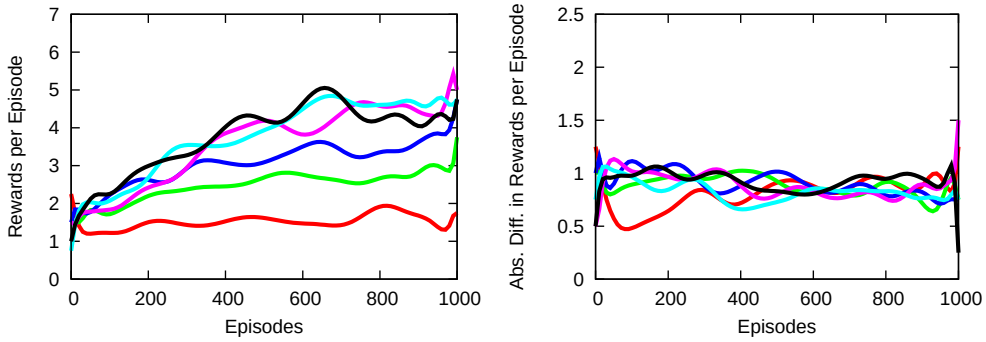


(e) Closest other agent to agent's nearest reward and closest agent to any reward.



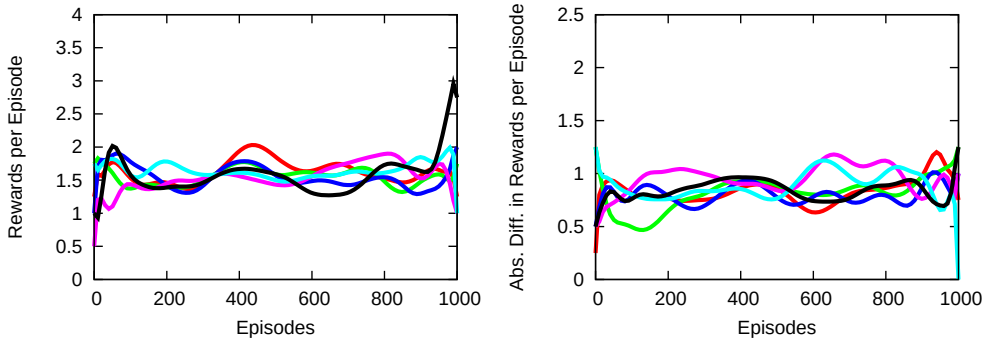
(f) Closest other agent to agent's nearest reward and closest agent to any reward.

Figure A.8: Tabular value functions - Two balanced agents, one nearest reward in the state representation with degrading ε values.



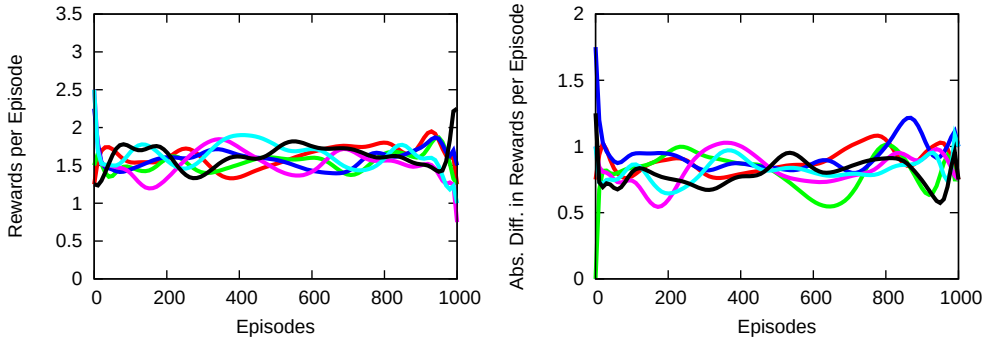
(a) Closest agent to any reward.

(b) Closest agent to any reward.



(c) Closest other agent to agent's two nearest rewards.

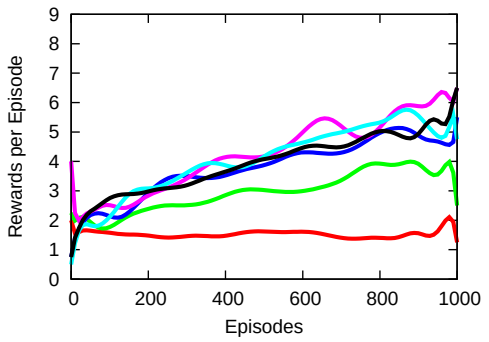
(d) Closest other agent to agent's two nearest rewards.



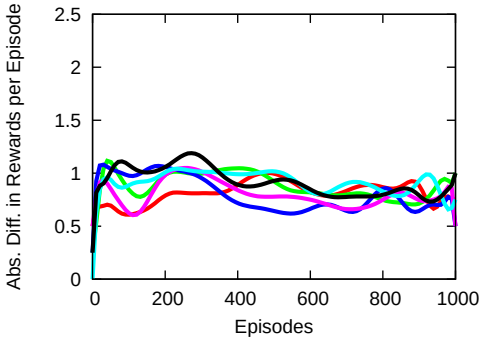
(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

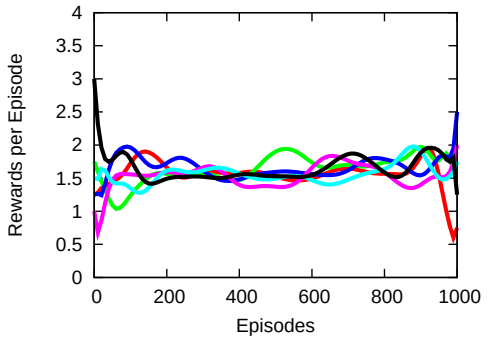
Figure A.9: Tabular value functions - Two balanced agents, two nearest rewards in the state representation with constant ε values.



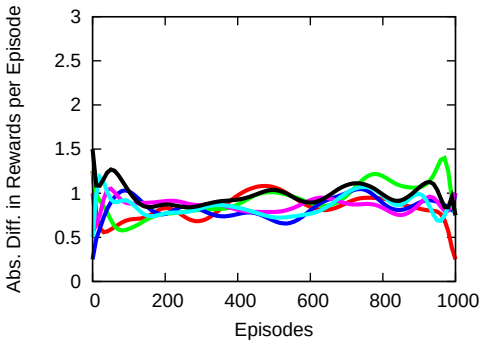
(a) Closest agent to any reward.



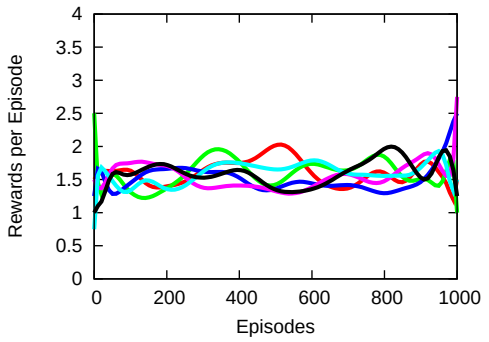
(b) Closest agent to any reward.



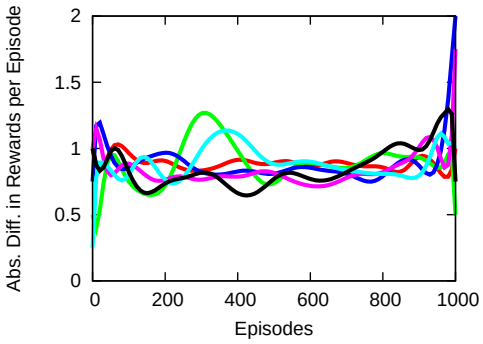
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

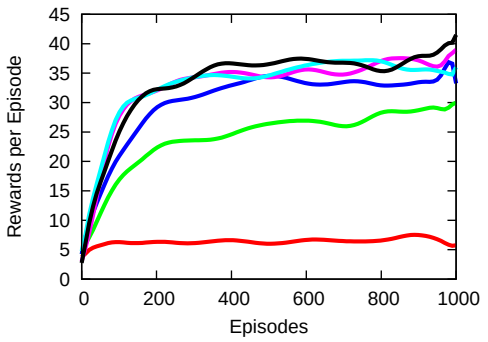


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

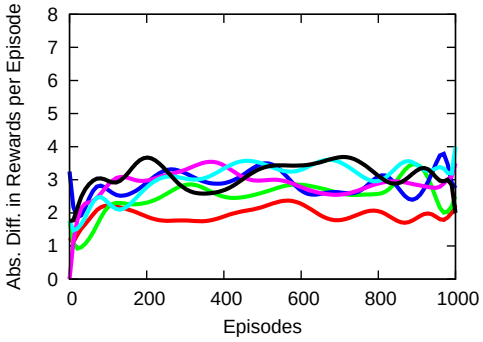


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

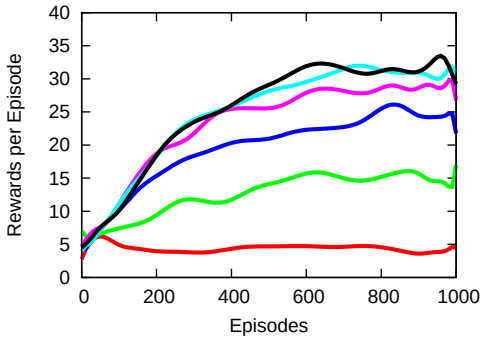
Figure A.10: Tabular value functions - Two balanced agents, two nearest rewards in the state representation with degrading ε values.



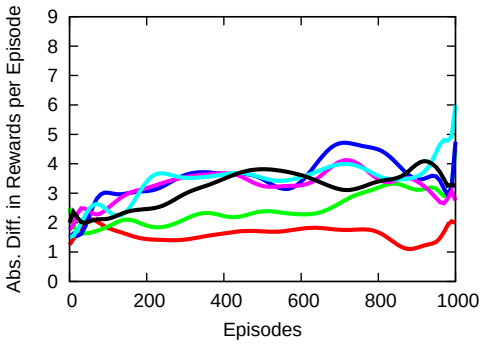
(a) Closest agent to any reward.



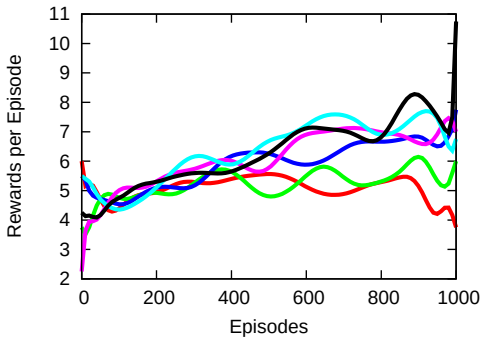
(b) Closest agent to any reward.



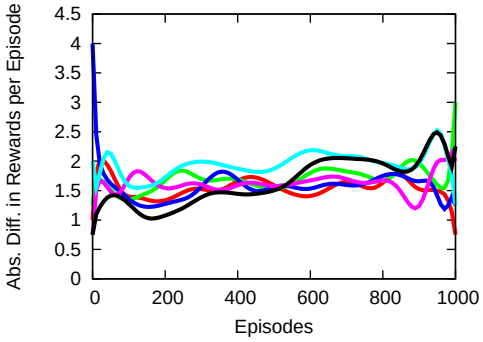
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

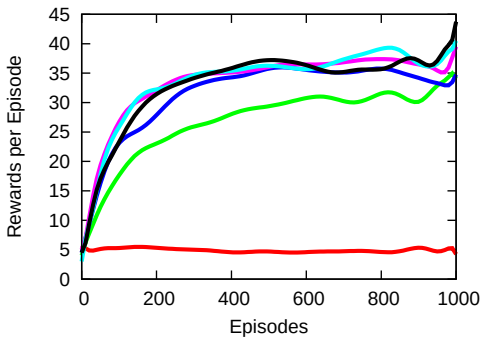


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

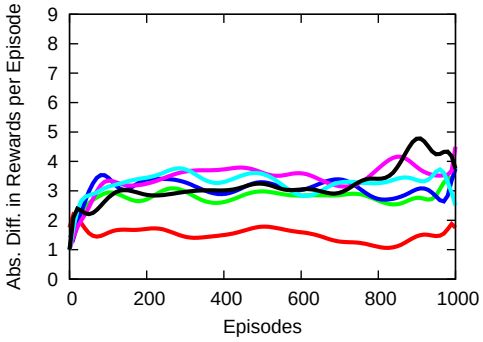


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

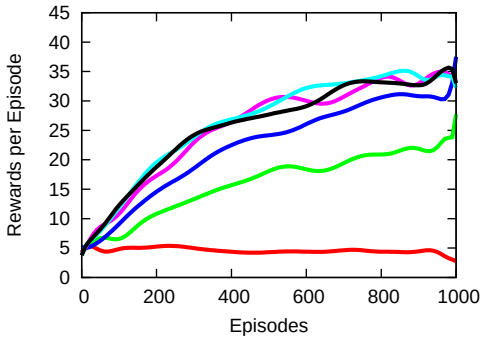
Figure A.11: Tabular value functions - Two individual teams of two agents each, one nearest reward in the state representation with constant ε values.



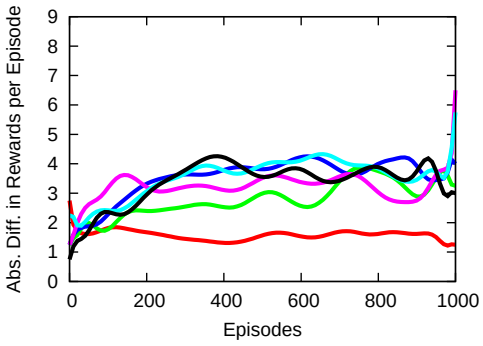
(a) Closest agent to any reward.



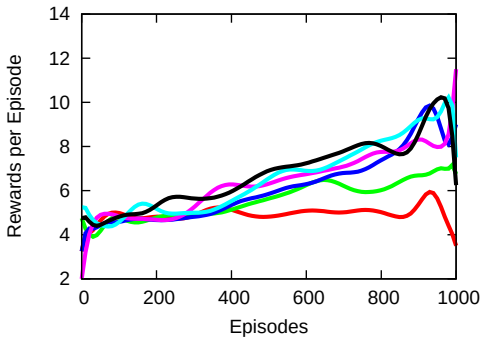
(b) Closest agent to any reward.



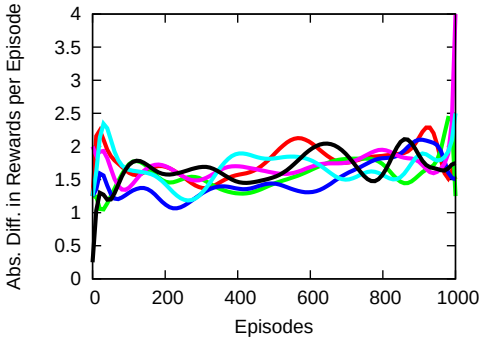
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

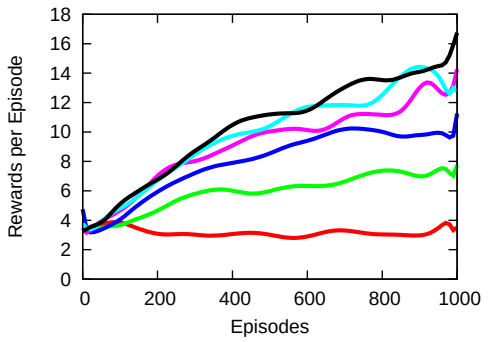


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

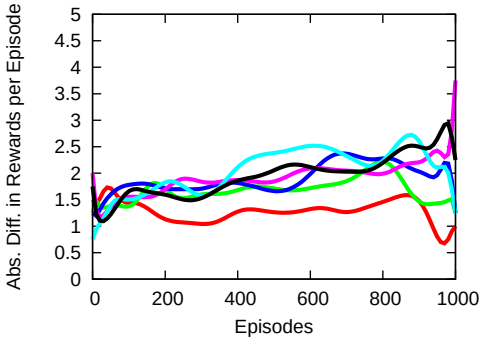


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

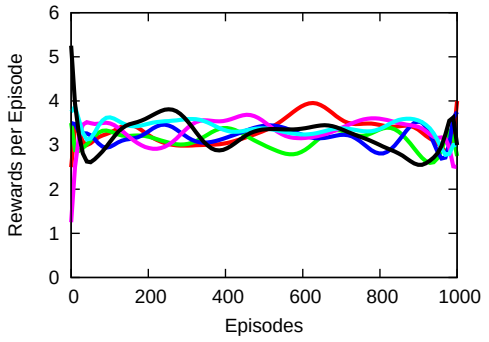
Figure A.12: Tabular value functions - Two individual teams of two agents each, one nearest reward in the state representation with degrading ε values.



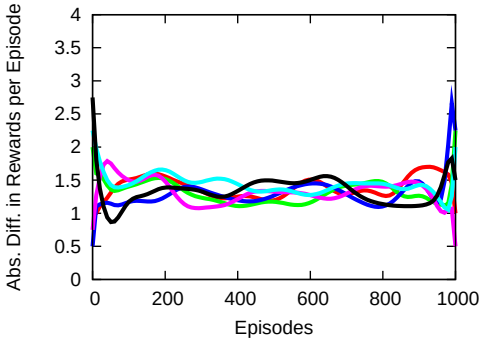
(a) Closest agent to any reward.



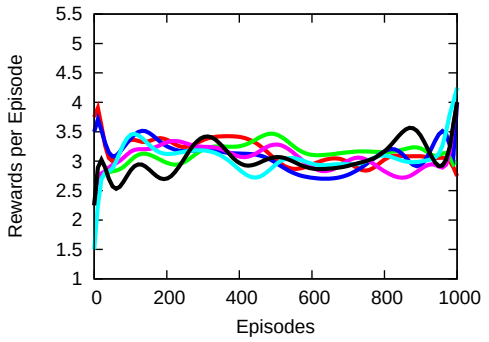
(b) Closest agent to any reward.



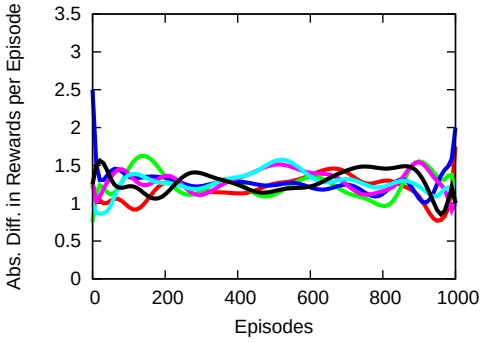
(c) Closest other agent to agent's nearest two rewards.



(d) Closest other agent to agent's nearest two rewards.

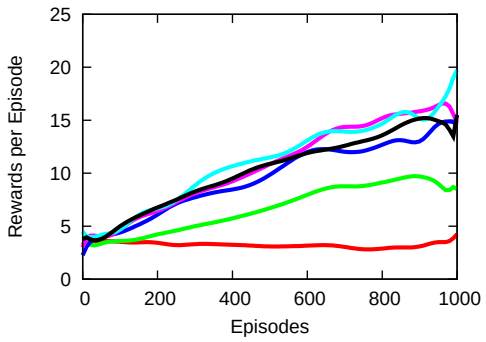


(e) Closest other agent to agent's nearest two rewards and closest agent to any reward.

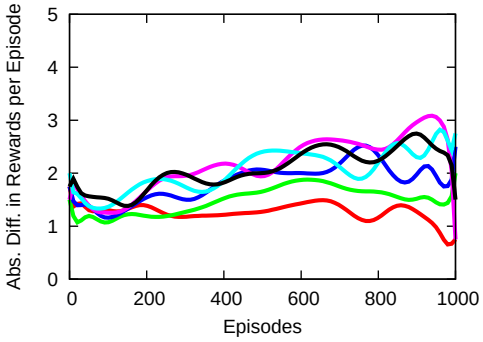


(f) Closest other agent to agent's nearest two rewards and closest agent to any reward.

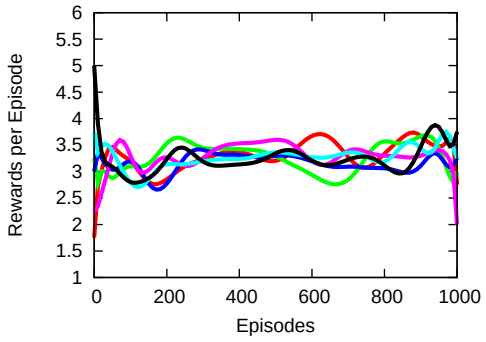
Figure A.13: Tabular value functions - Two individual teams of two agents each, two nearest rewards in the state representation with constant ε values.



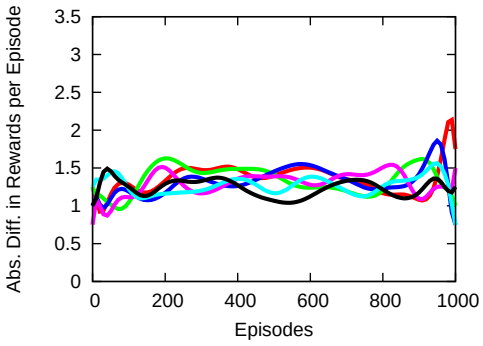
(a) Closest agent to any reward.



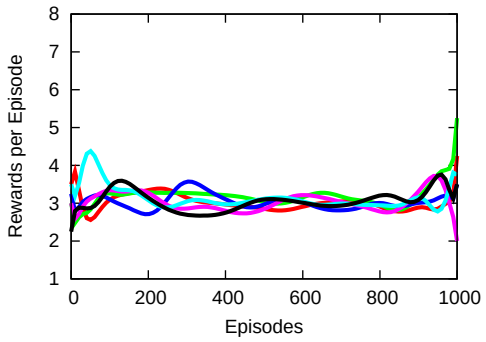
(b) Closest agent to any reward.



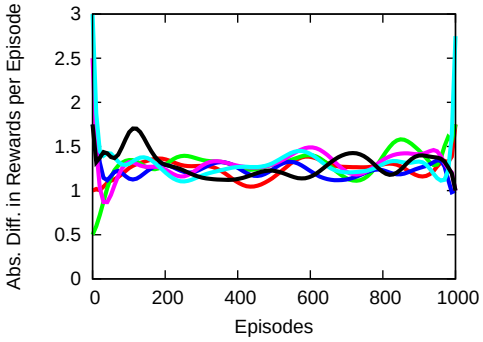
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

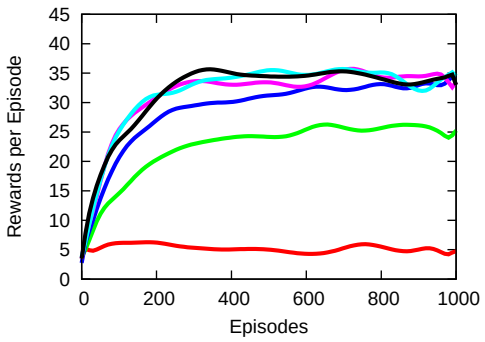


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

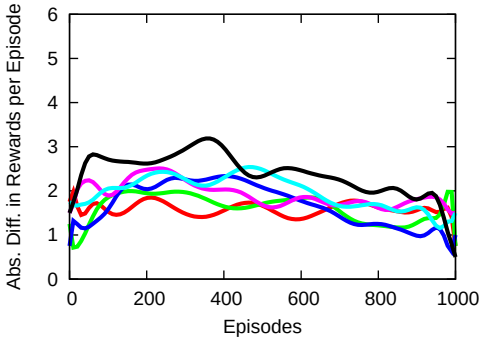


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

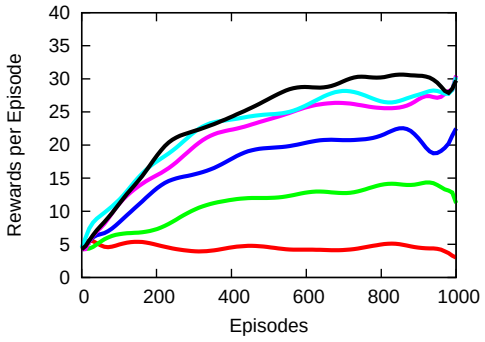
Figure A.14: Tabular value functions - Two individual teams of two agents each, two nearest rewards in the state representation with degrading ε values.



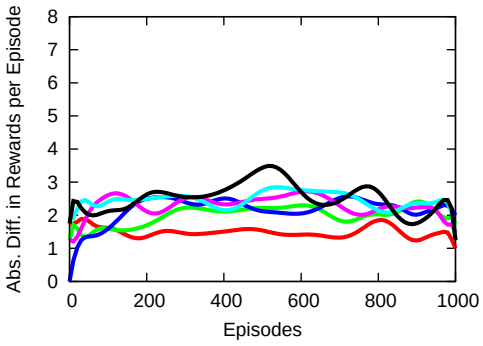
(a) Closest agent to any reward.



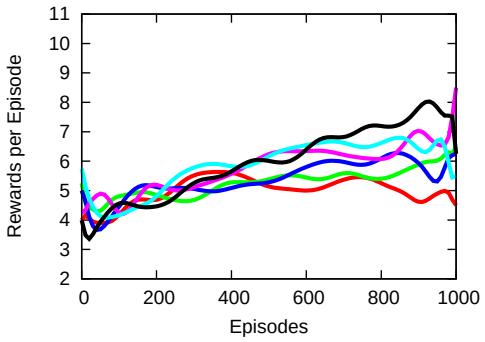
(b) Closest agent to any reward.



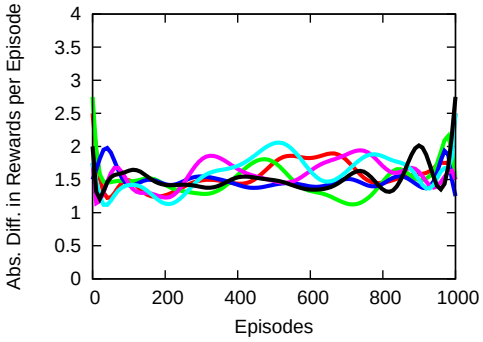
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

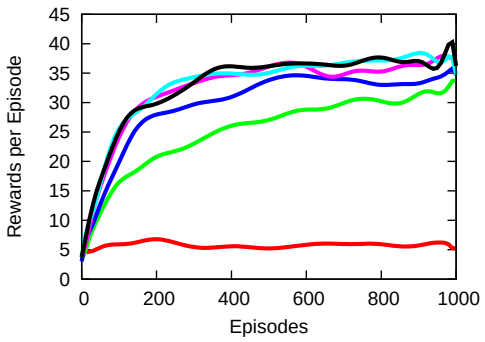


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

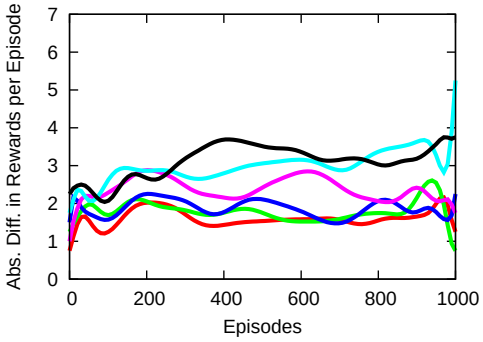


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

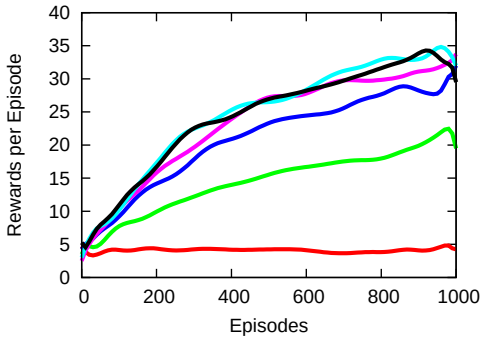
Figure A.15: Tabular value functions - Two balanced teams of two agents each, one nearest reward in the state representation with constant ε values.



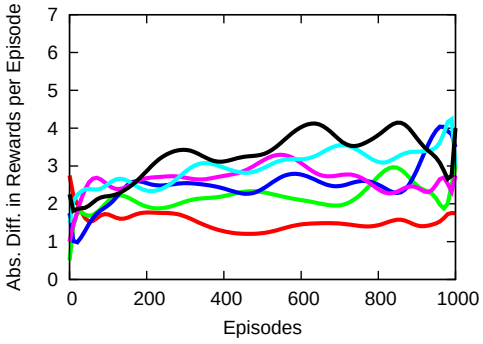
(a) Closest agent to any reward.



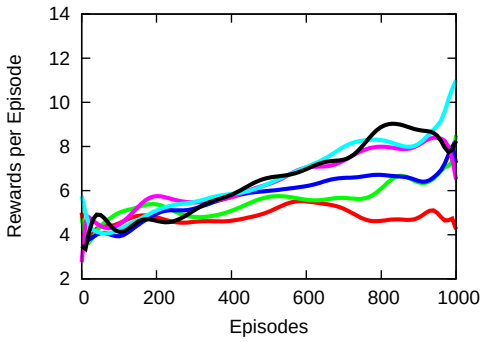
(b) Closest agent to any reward.



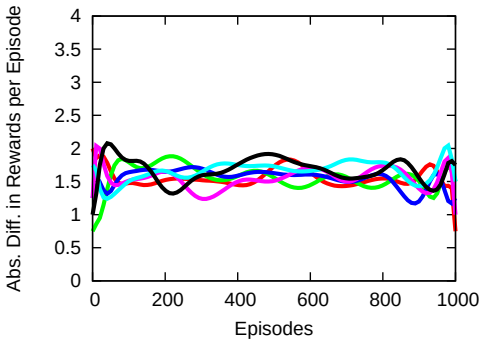
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

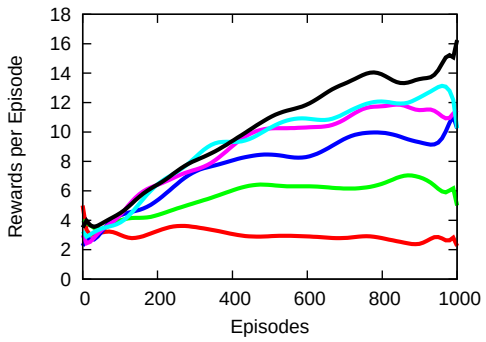


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

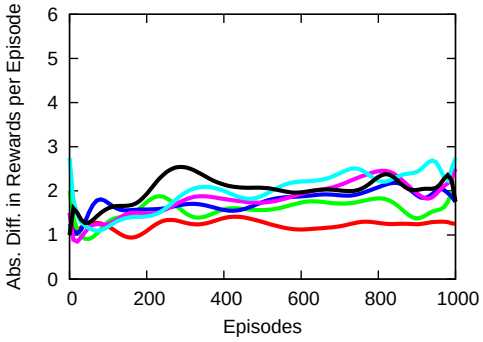


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

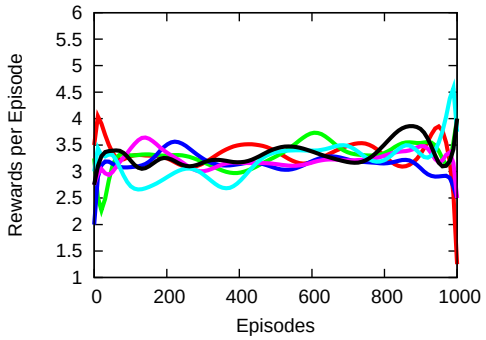
Figure A.16: Tabular value functions - Two balanced teams of two agents each, one nearest reward in the state representation with degrading ε values.



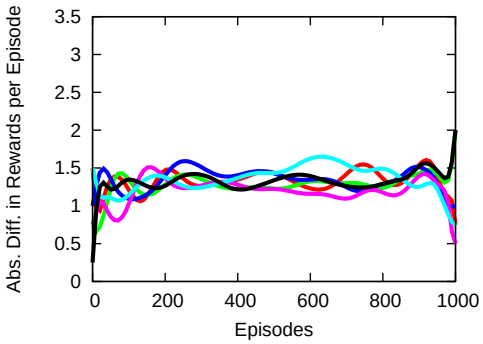
(a) Closest agent to any reward.



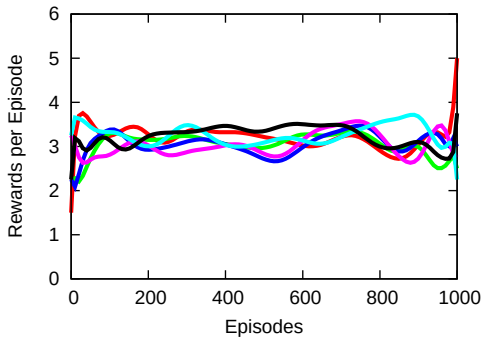
(b) Closest agent to any reward.



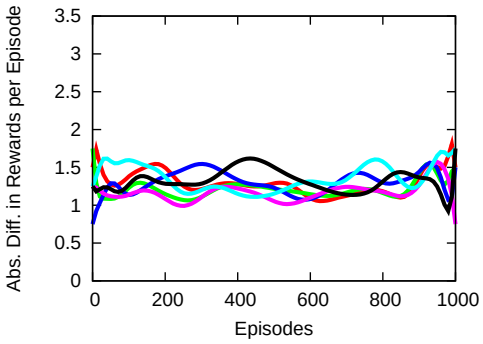
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

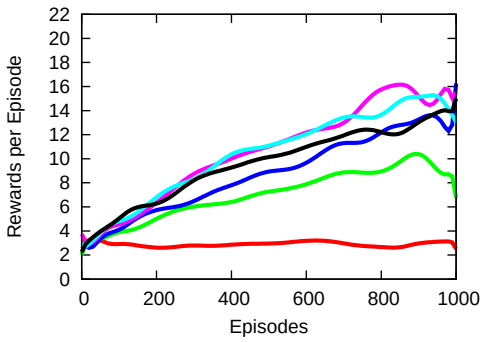


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

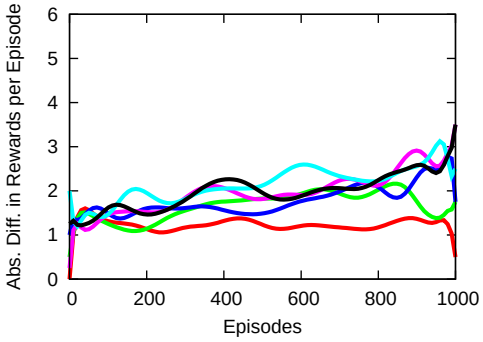


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

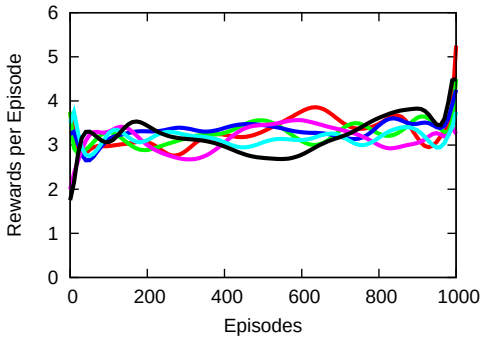
Figure A.17: Tabular value functions - Two balanced teams of two agents each, two nearest rewards in the state representation with constant ε values.



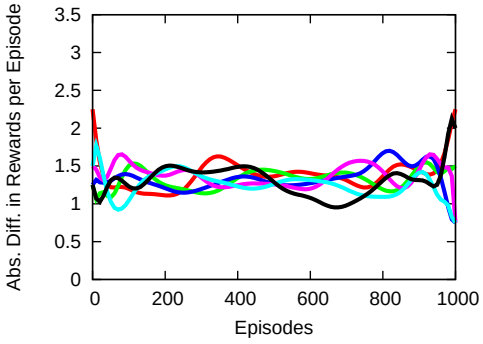
(a) Closest agent to any reward.



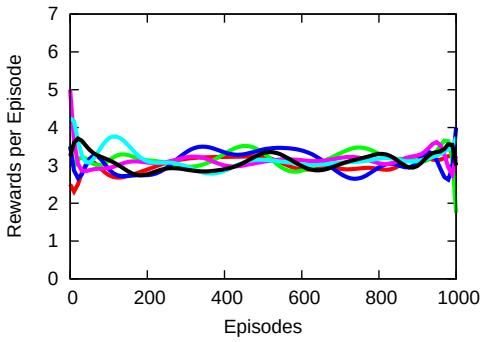
(b) Closest agent to any reward.



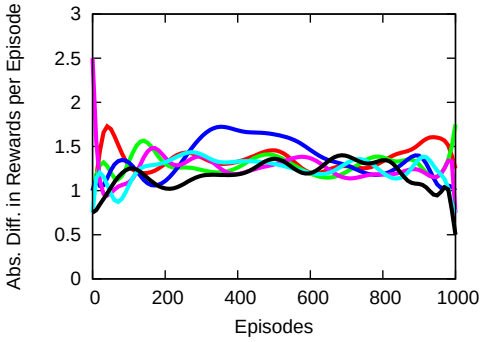
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

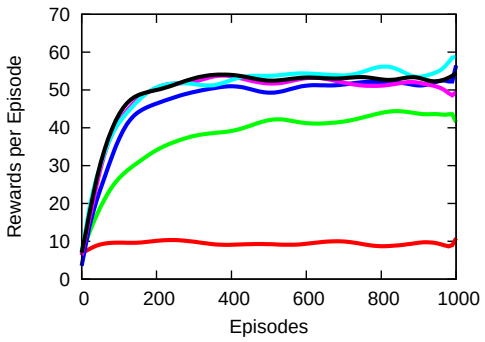


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

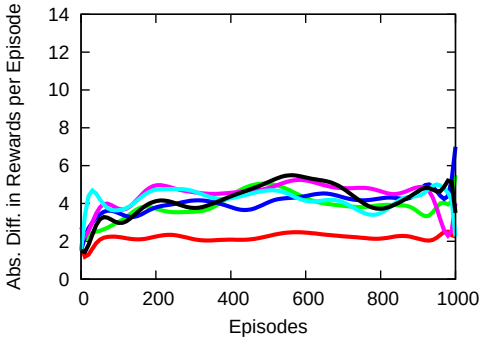


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

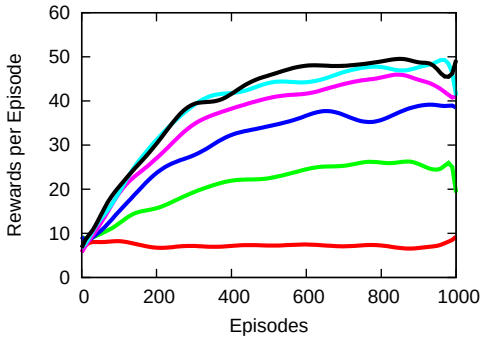
Figure A.18: Tabular value functions - Two balanced teams of two agents each, two nearest rewards in the state representation with degrading ε values.



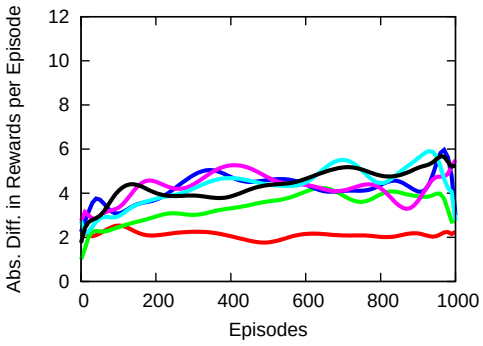
(a) Closest agent to any reward.



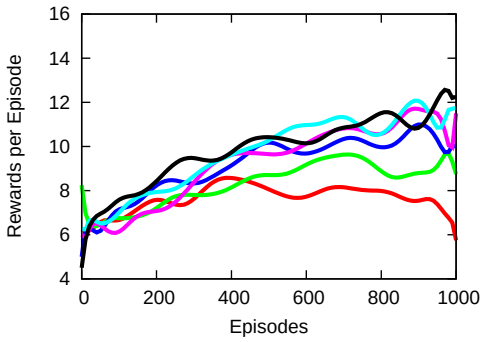
(b) Closest agent to any reward.



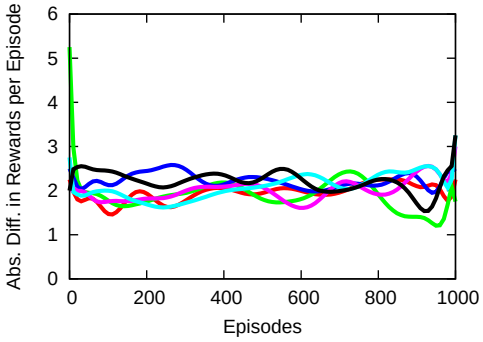
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

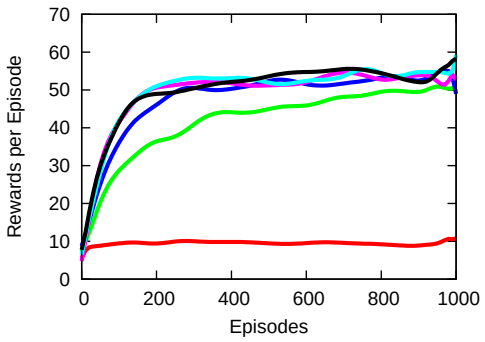


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

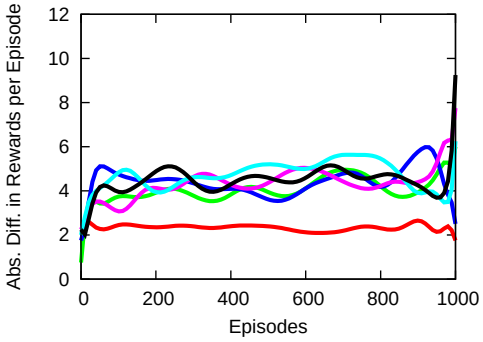


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

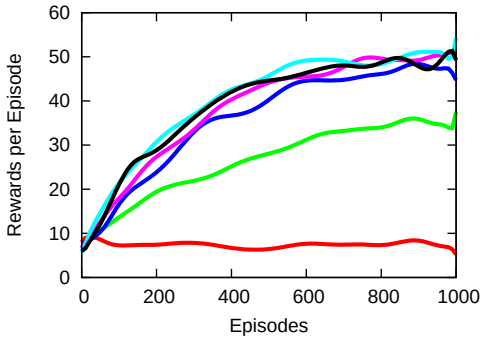
Figure A.19: Tabular value functions - Two individual teams of three agents each, one nearest reward in the state representation with constant ε values.



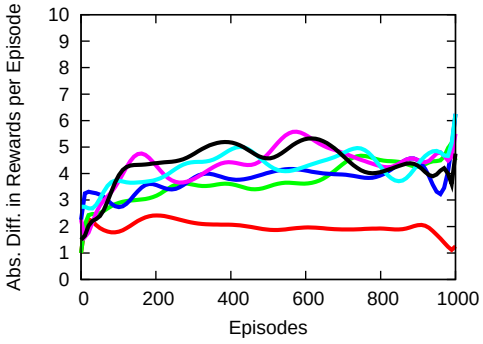
(a) Closest agent to any reward.



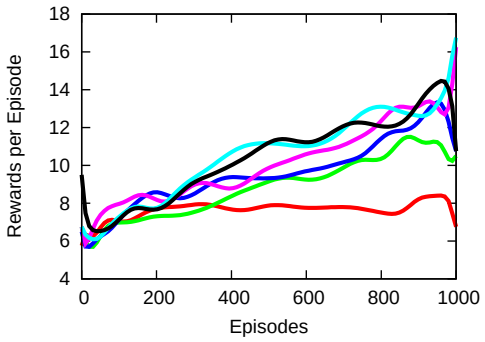
(b) Closest agent to any reward.



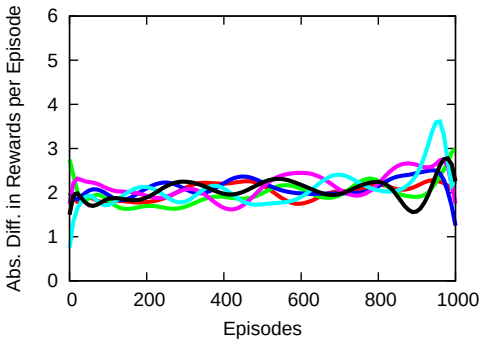
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

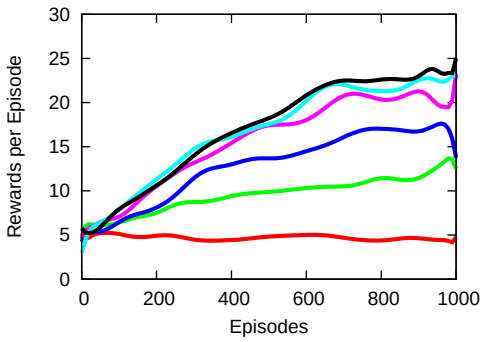


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

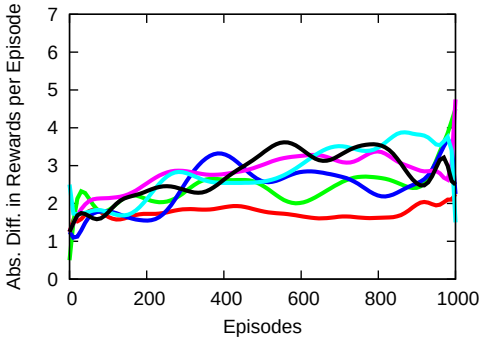


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

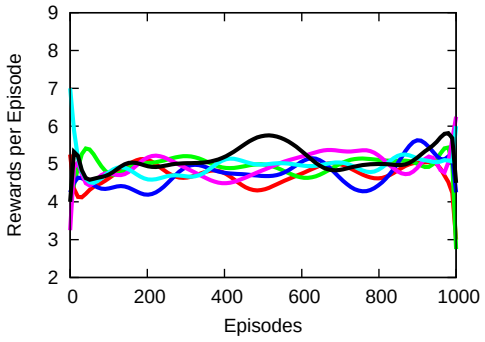
Figure A.20: Tabular value functions - Two individual teams of three agents each, one nearest reward in the state representation with degrading ε values.



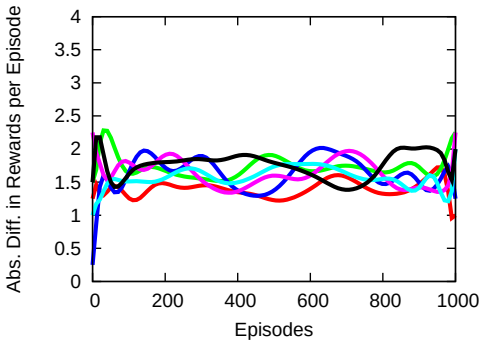
(a) Closest agent to any reward.



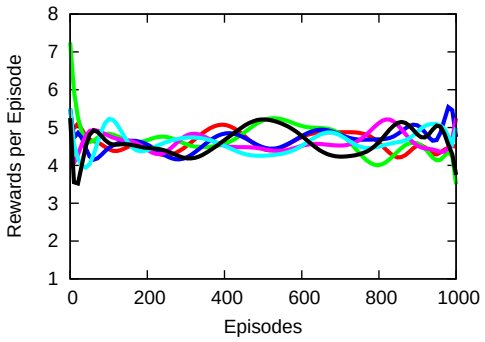
(b) Closest agent to any reward.



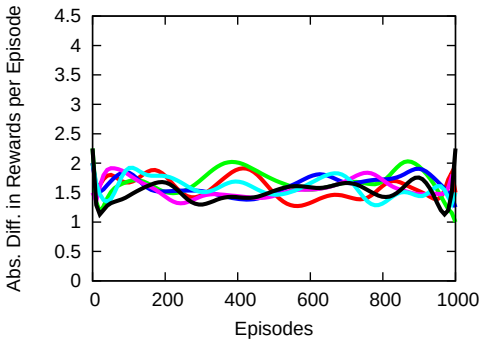
(c) Closest other agent to agent's nearest two rewards.



(d) Closest other agent to agent's nearest two rewards.

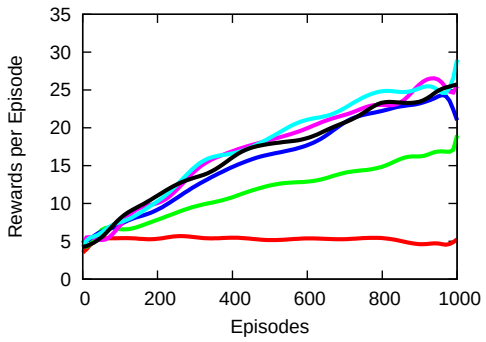


(e) Closest other agent to agent's nearest two rewards and closest agent to any reward.

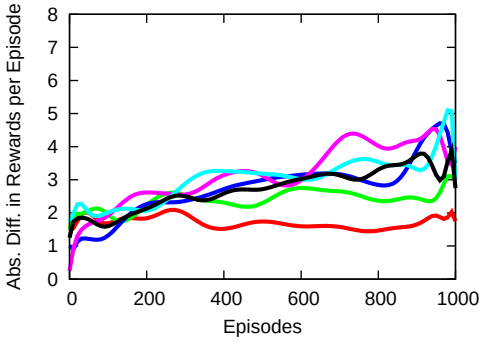


(f) Closest other agent to agent's nearest two rewards and closest agent to any reward.

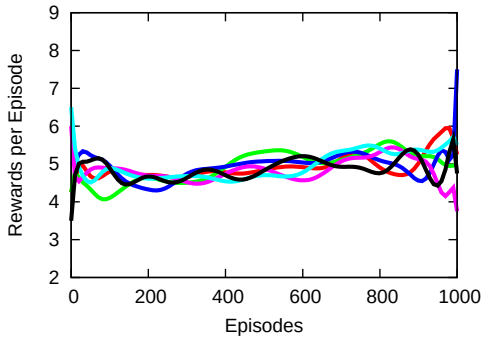
Figure A.21: Tabular value functions - Two individual teams of three agents each, two nearest rewards in the state representation with constant ε values.



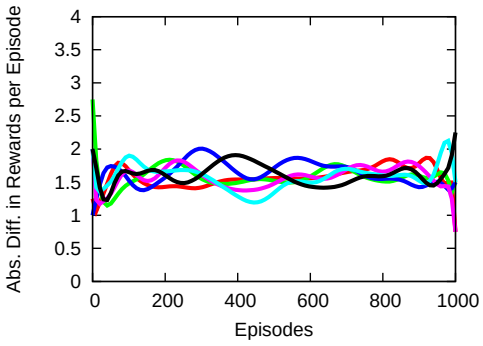
(a) Closest agent to any reward.



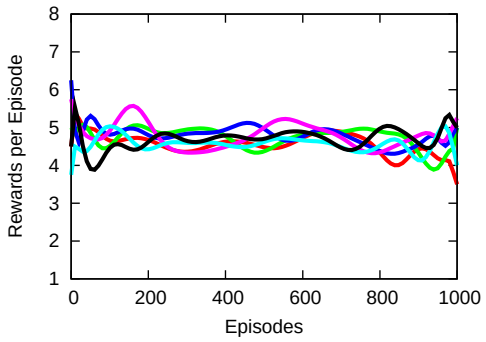
(b) Closest agent to any reward.



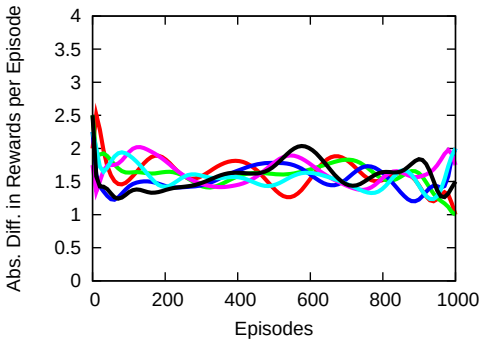
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

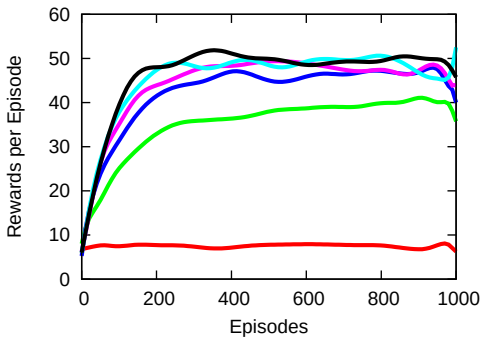


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

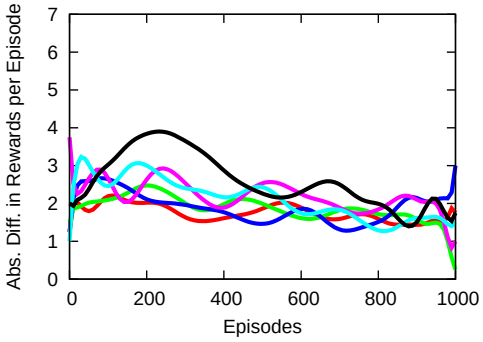


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

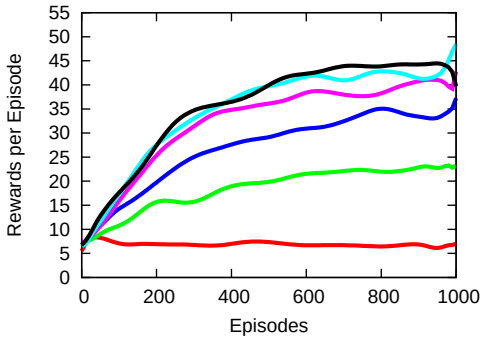
Figure A.22: Tabular value functions - Two individual teams of three agents each, two nearest rewards in the state representation with degrading ε values.



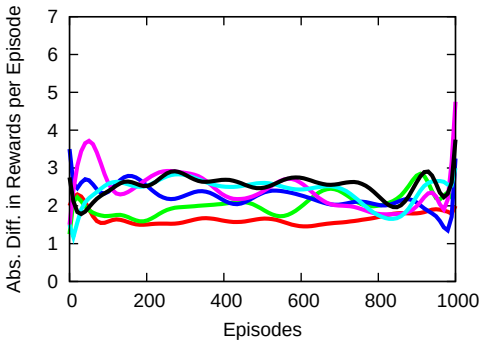
(a) Closest agent to any reward.



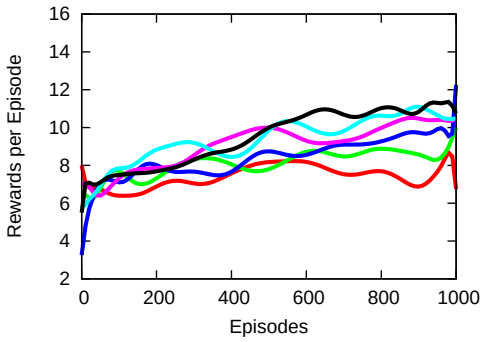
(b) Closest agent to any reward.



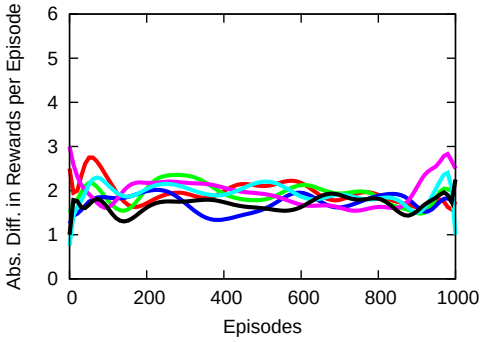
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

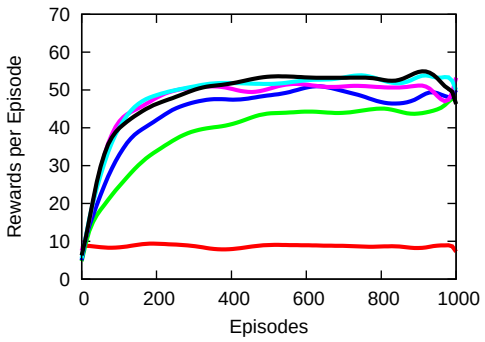


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

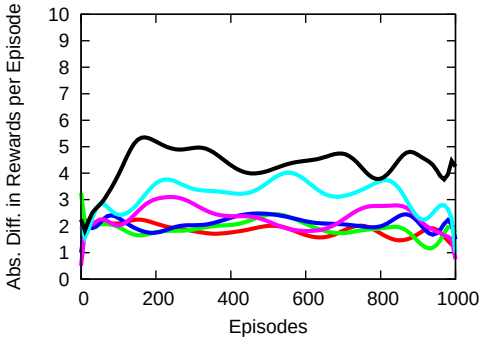


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

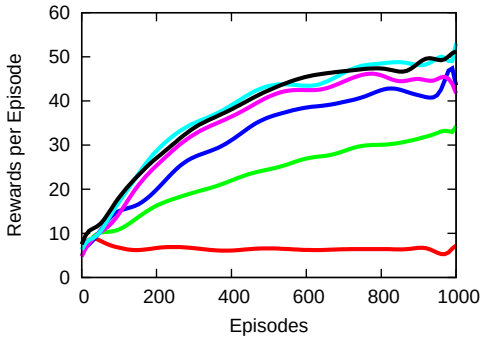
Figure A.23: Tabular value functions - Two balanced teams of three agents each, one nearest reward in the state representation with constant ε values.



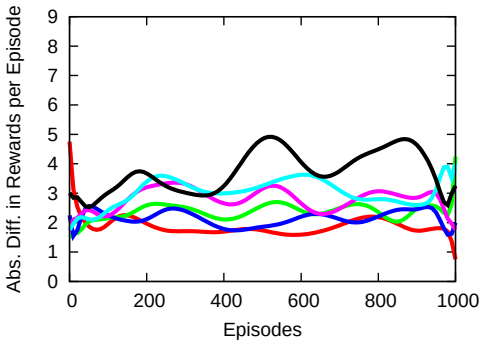
(a) Closest agent to any reward.



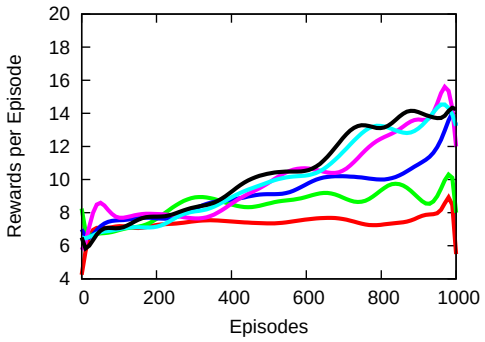
(b) Closest agent to any reward.



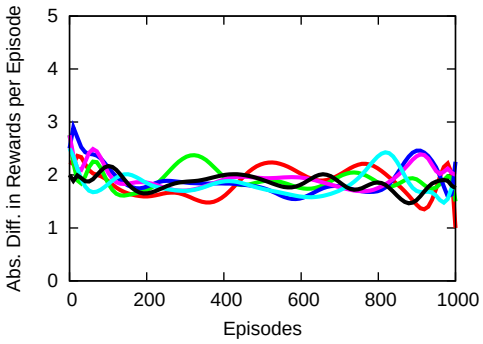
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

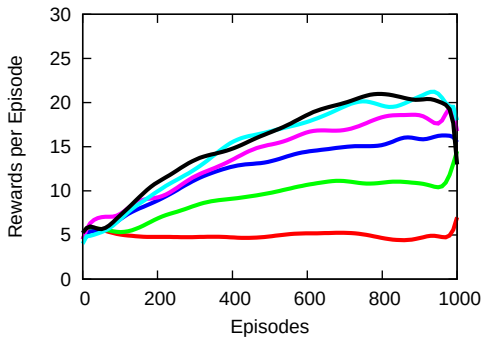


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

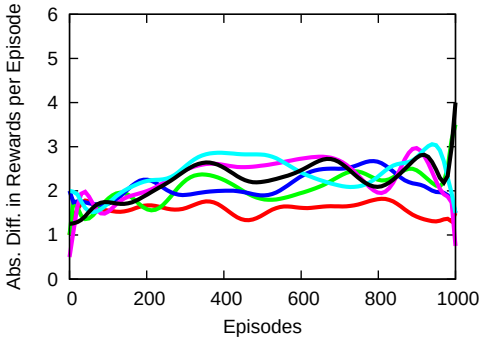


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

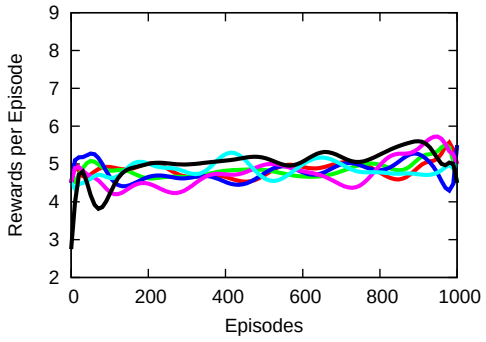
Figure A.24: Tabular value functions - Two balanced teams of three agents each, one nearest reward in the state representation with degrading ε values.



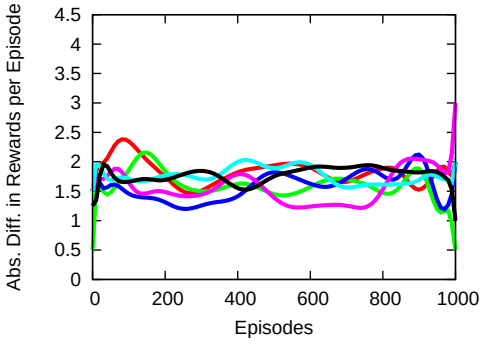
(a) Closest agent to any reward.



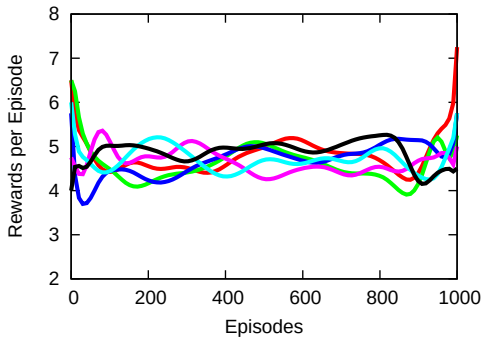
(b) Closest agent to any reward.



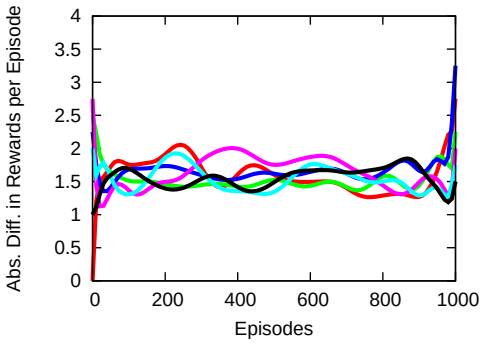
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

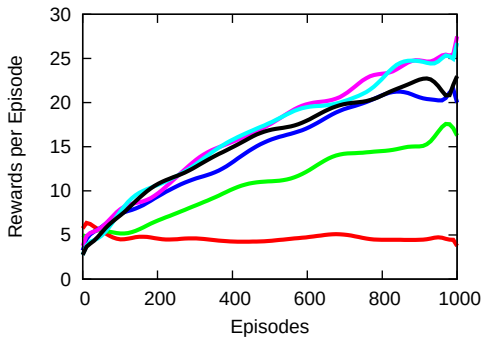


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

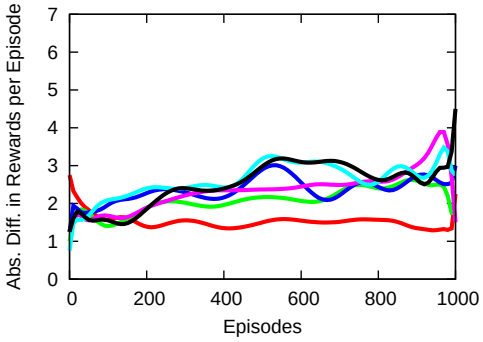


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

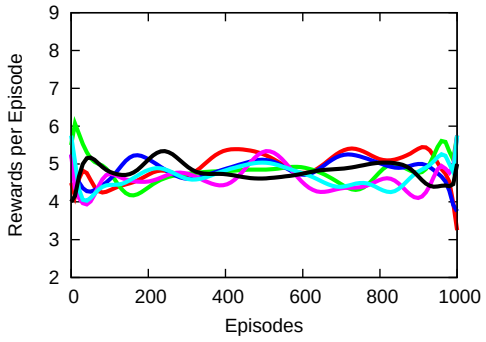
Figure A.25: Tabular value functions - Two balanced teams of three agents each, two nearest rewards in the state representation with constant ε values.



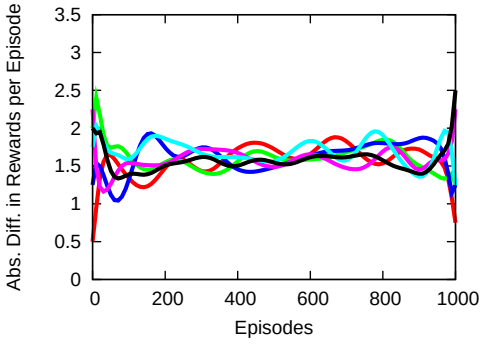
(a) Closest agent to any reward.



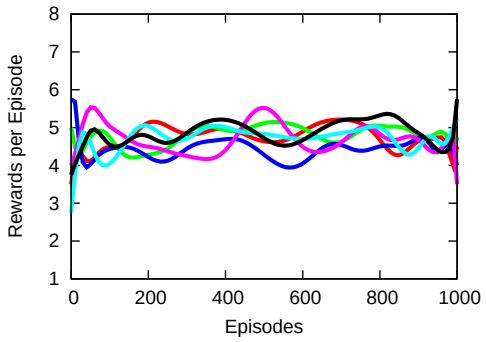
(b) Closest agent to any reward.



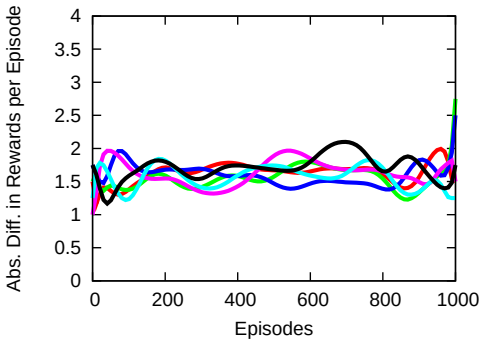
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

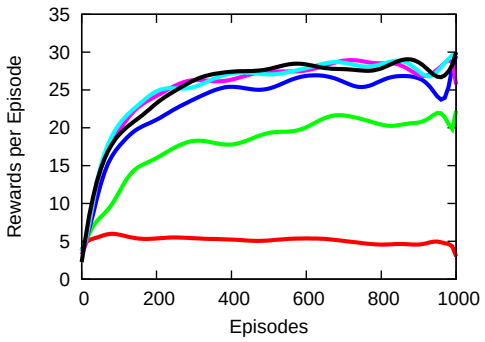


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

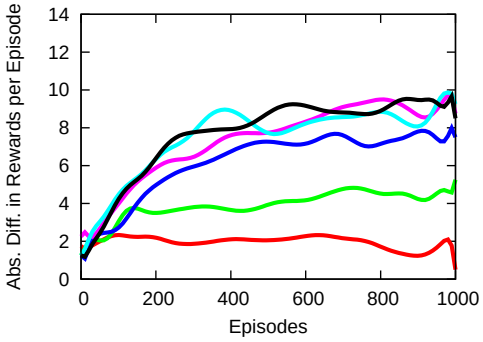


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

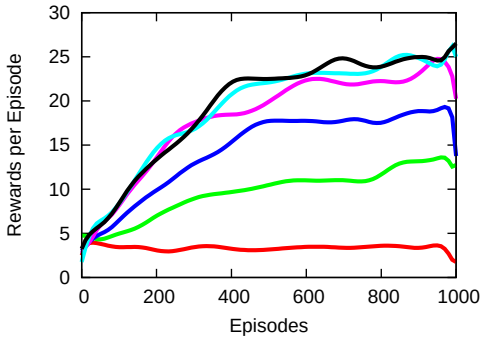
Figure A.26: Tabular value functions - Two balanced teams of three agents each, two nearest rewards in the state representation with degrading ε values.



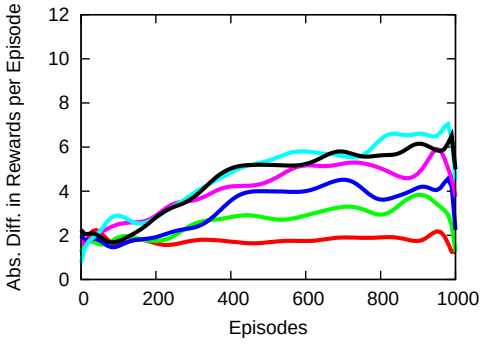
(a) Closest agent to any reward.



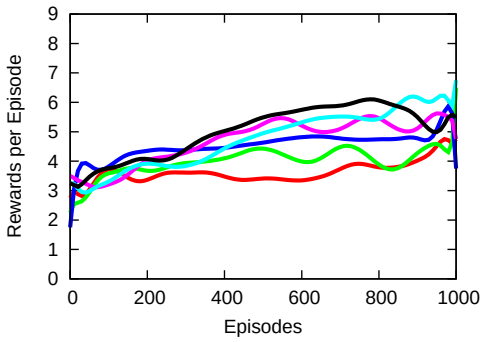
(b) Closest agent to any reward.



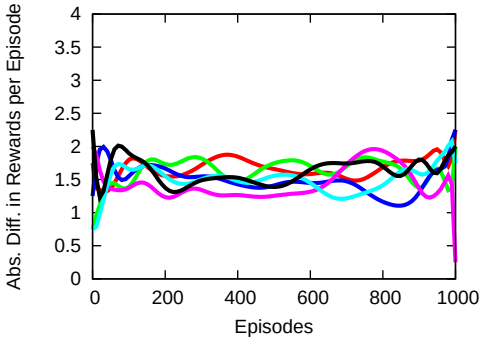
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

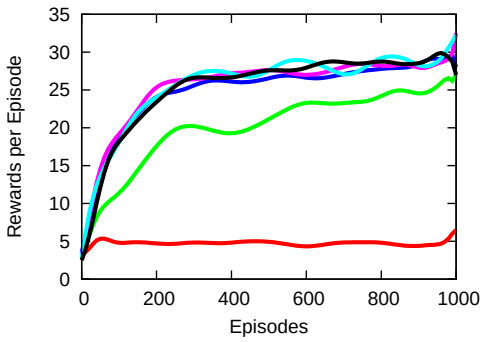


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

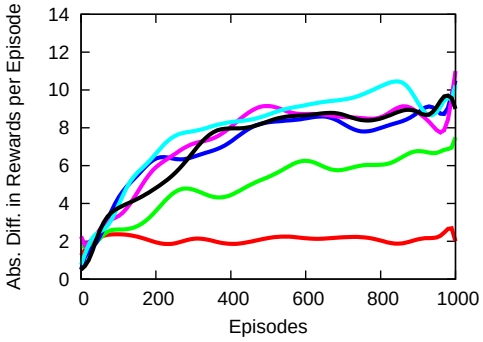


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

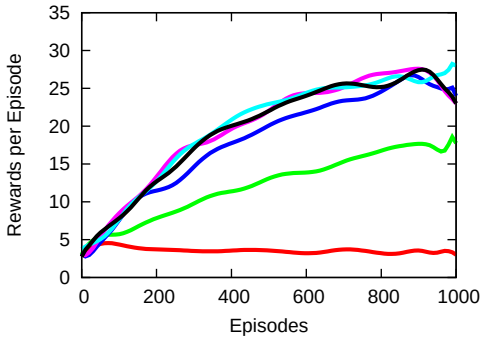
Figure A.27: Tabular value functions - A team of two agents and a single agent acting individually, one nearest reward in the state representation with constant ε values.



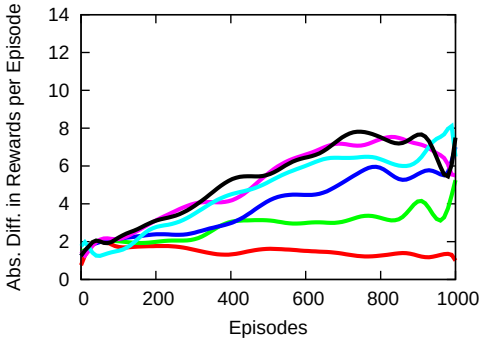
(a) Closest agent to any reward.



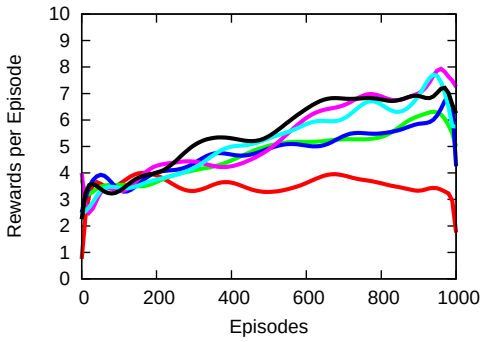
(b) Closest agent to any reward.



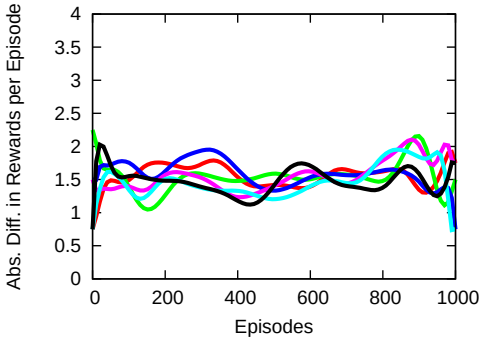
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

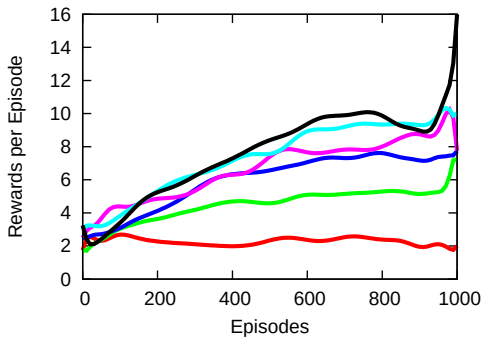


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

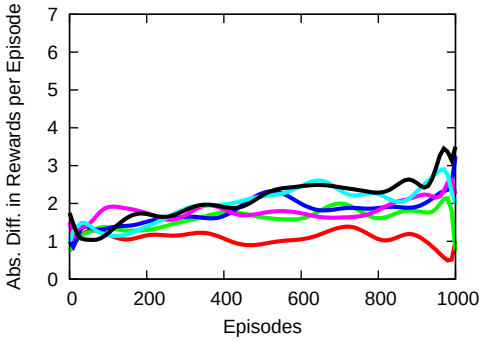


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

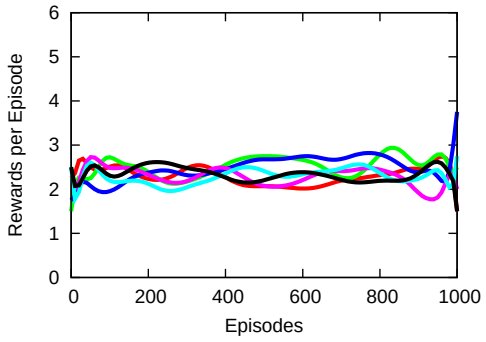
Figure A.28: Tabular value functions - A team of two agents and a single agent acting individually, one nearest reward in the state representation with degrading ε values.



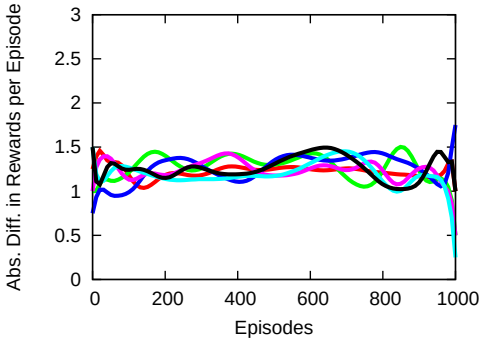
(a) Closest agent to any reward.



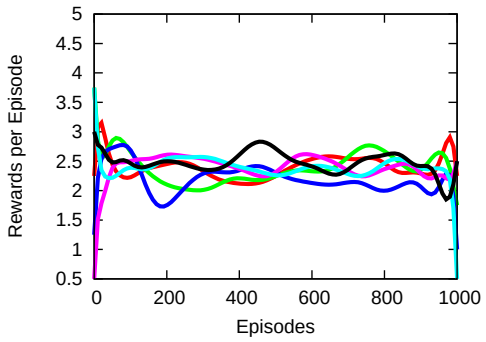
(b) Closest agent to any reward.



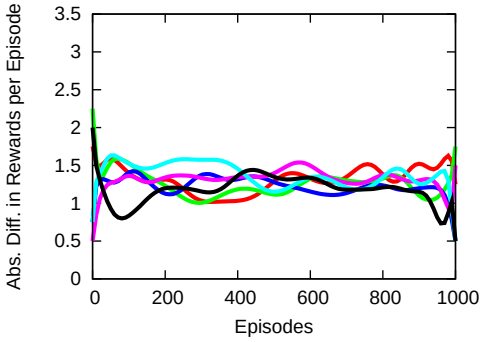
(c) Closest other agent to agent's nearest two rewards.



(d) Closest other agent to agent's nearest two rewards.

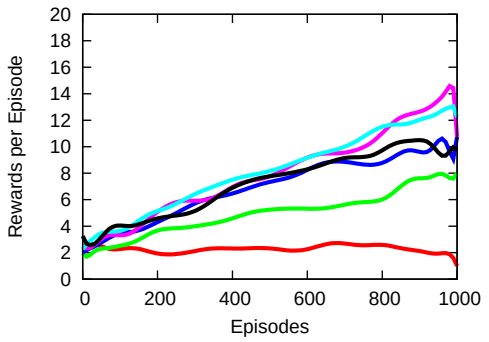


(e) Closest other agent to agent's nearest two rewards and closest agent to any reward.

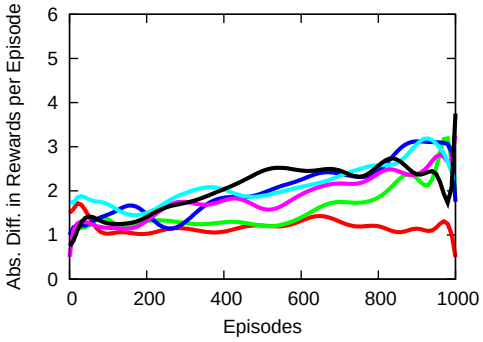


(f) Closest other agent to agent's nearest two rewards and closest agent to any reward.

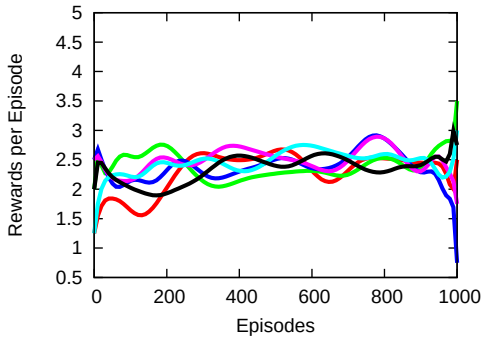
Figure A.29: Tabular value functions - A team of two agents and a single agent acting individually, two nearest rewards in the state representation with constant ε values.



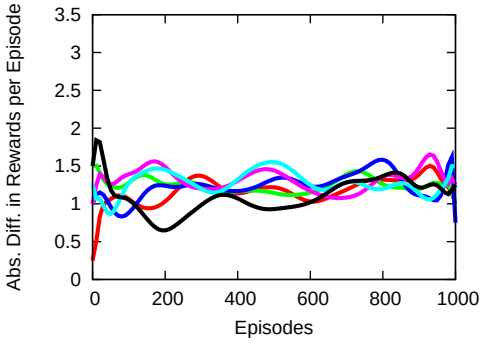
(a) Closest agent to any reward.



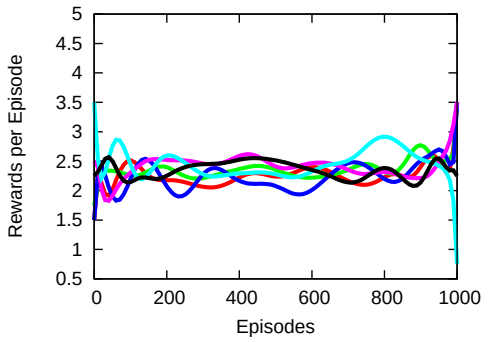
(b) Closest agent to any reward.



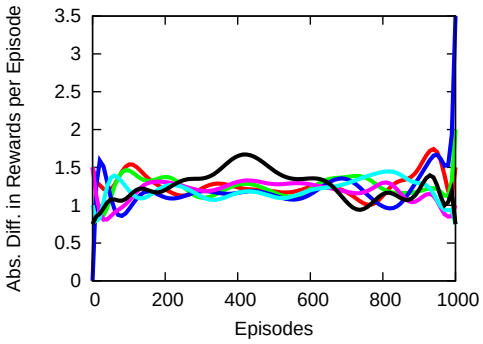
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

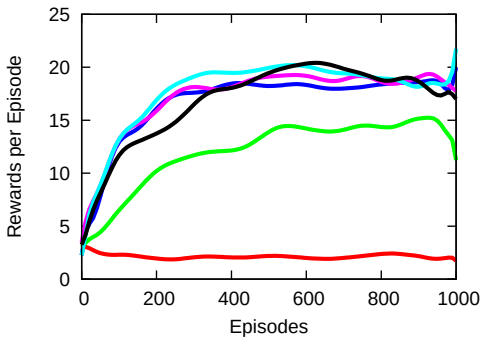


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

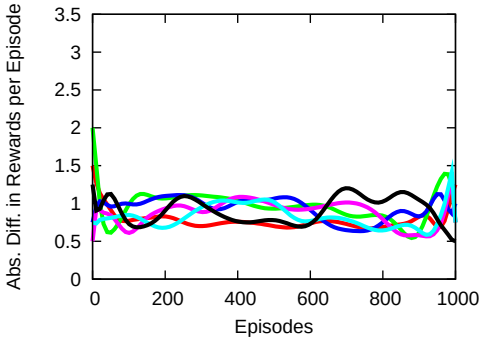


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

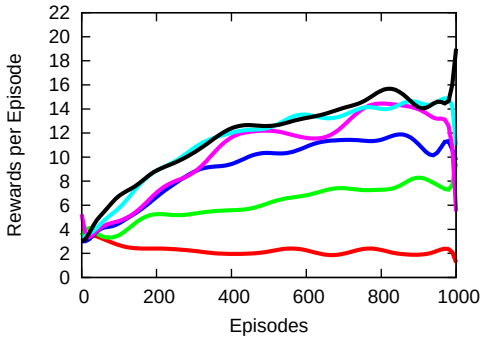
Figure A.30: Tabular value functions - A team of two agents and a single agent acting individually, two nearest rewards in the state representation with degrading ε values.



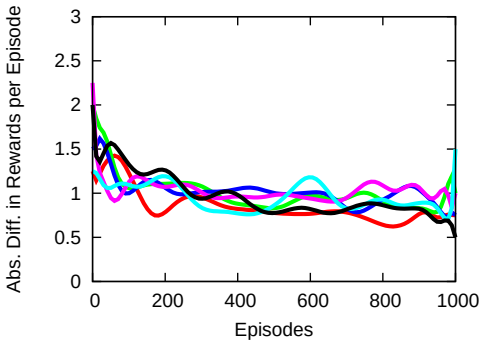
(a) Closest agent to any reward.



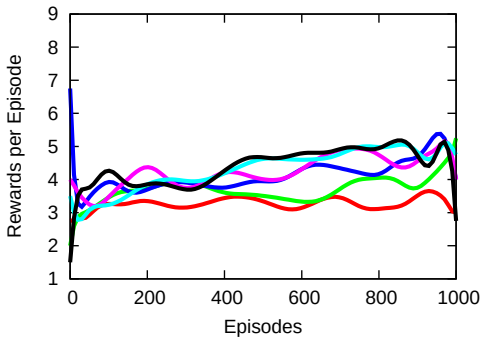
(b) Closest agent to any reward.



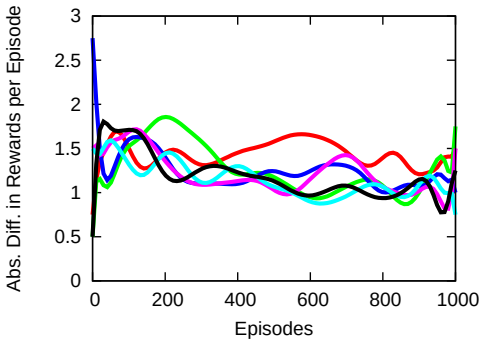
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

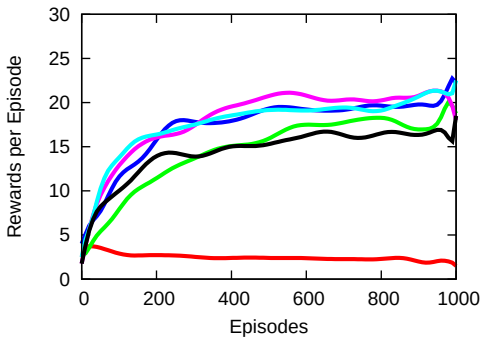


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

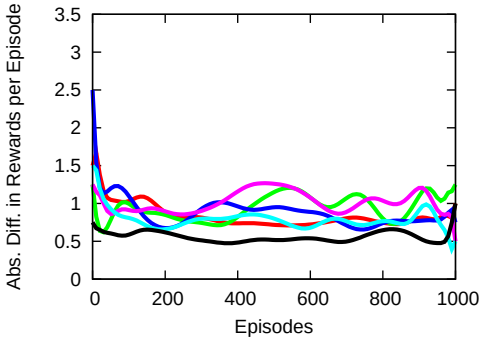


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

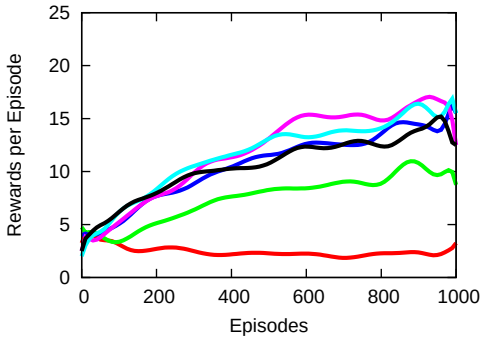
Figure A.31: Tabular value functions - A team of two agents and a single agent in balance, one nearest reward in the state representation with constant ε values.



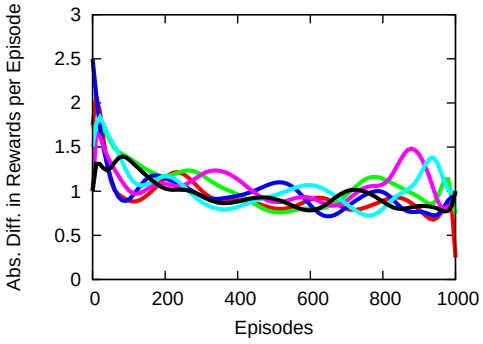
(a) Closest agent to any reward.



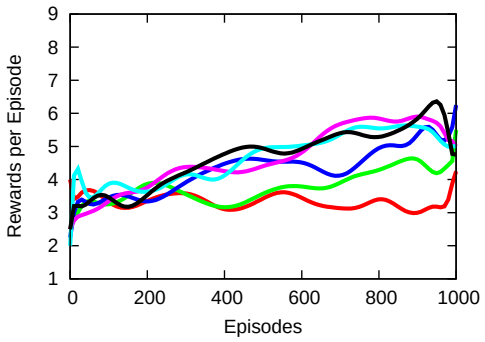
(b) Closest agent to any reward.



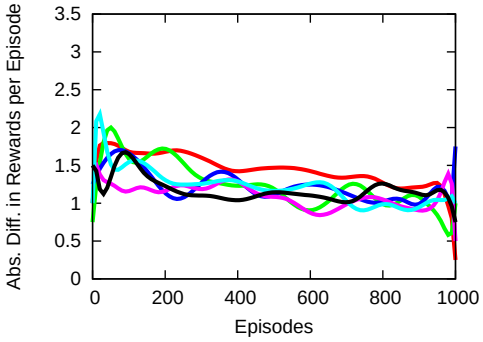
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

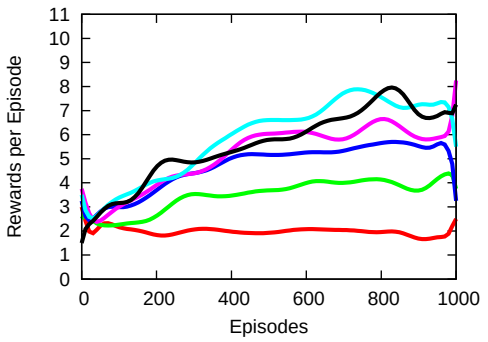


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

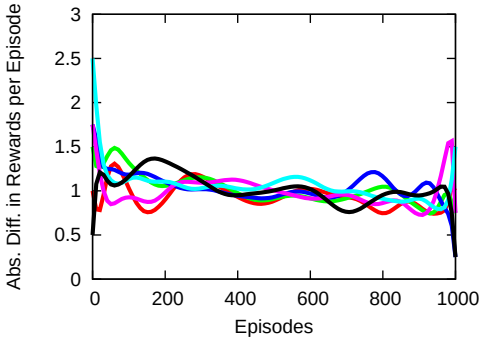


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

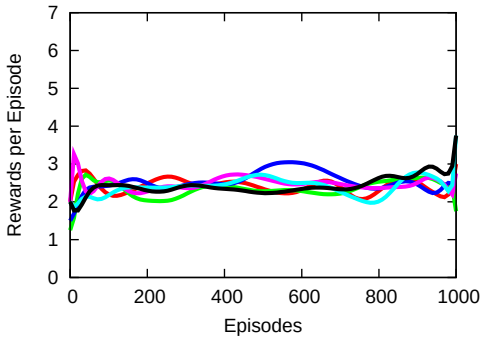
Figure A.32: Tabular value functions - A team of two agents and a single agent in balance, one nearest reward in the state representation with degrading ε values.



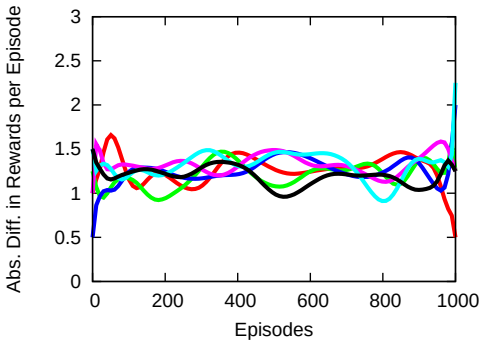
(a) Closest agent to any reward.



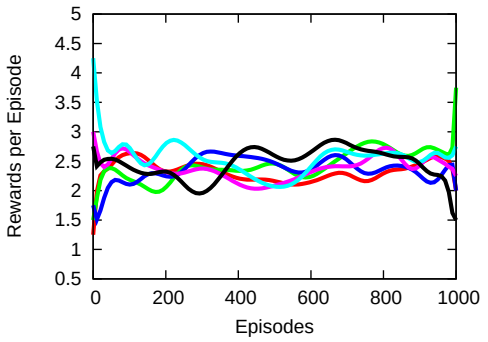
(b) Closest agent to any reward.



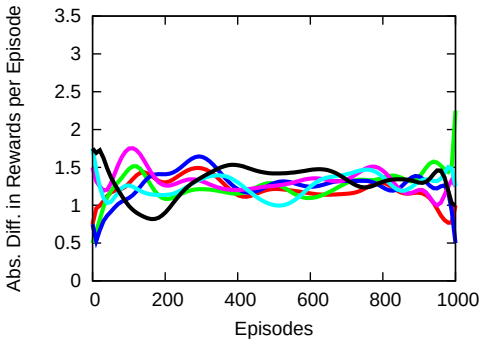
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

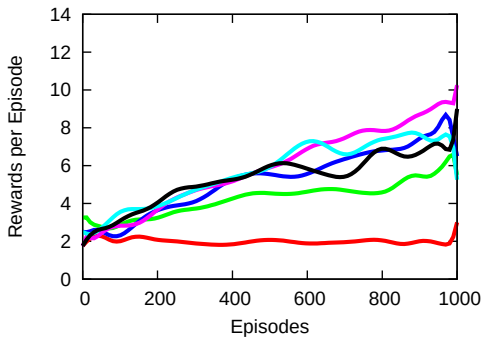


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

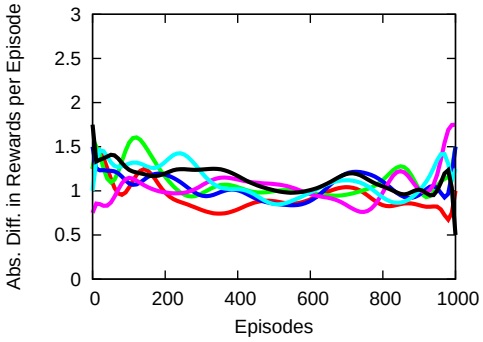


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

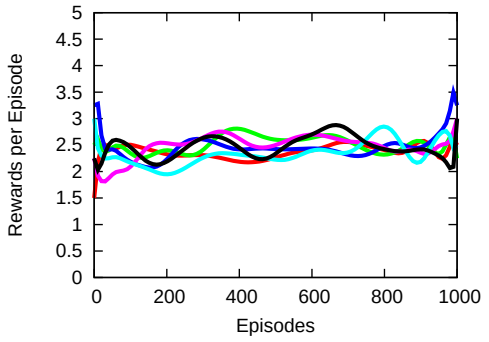
Figure A.33: Tabular value functions - A team of two agents and a single agent in balance, two nearest rewards in the state representation with constant ε values.



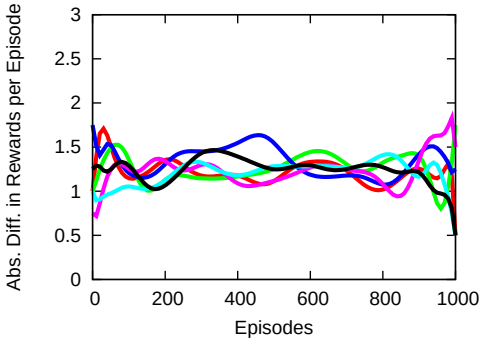
(a) Closest agent to any reward.



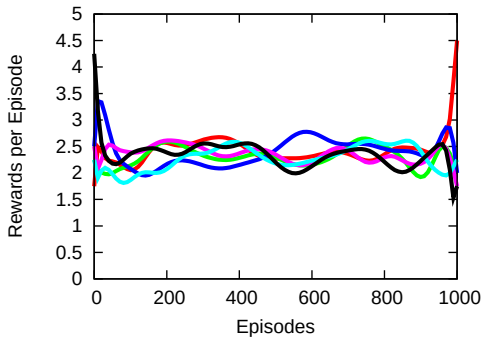
(b) Closest agent to any reward.



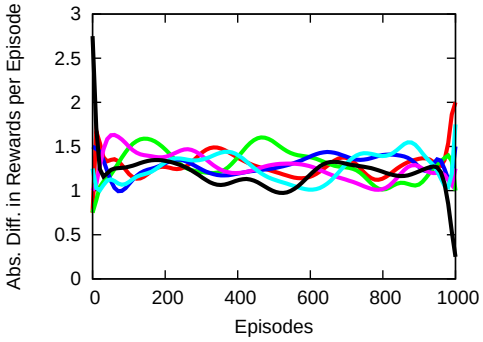
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

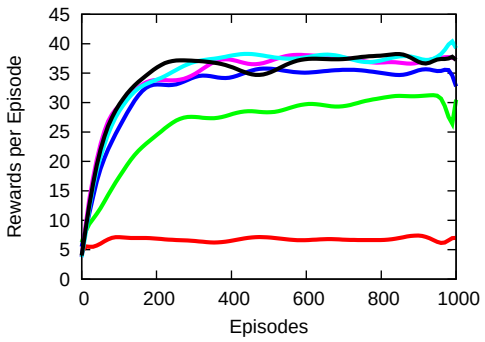


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

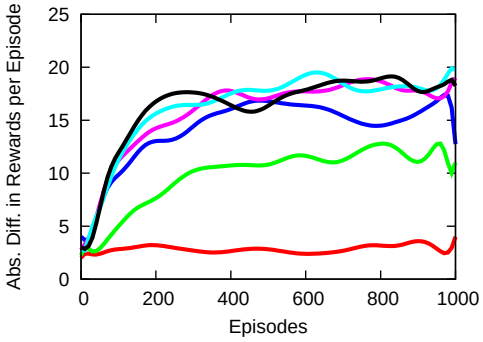


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

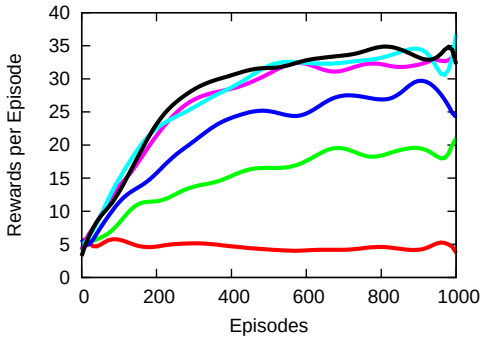
Figure A.34: Tabular value functions - A team of two agents and a single agent in balance, two nearest rewards in the state representation with degrading ε values.



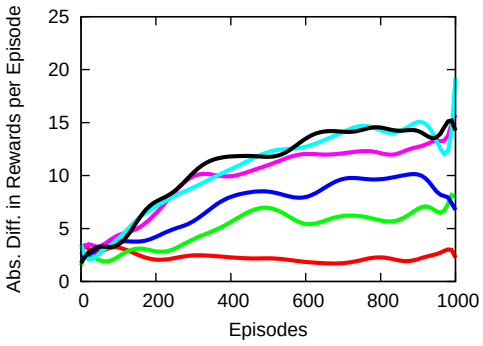
(a) Closest agent to any reward.



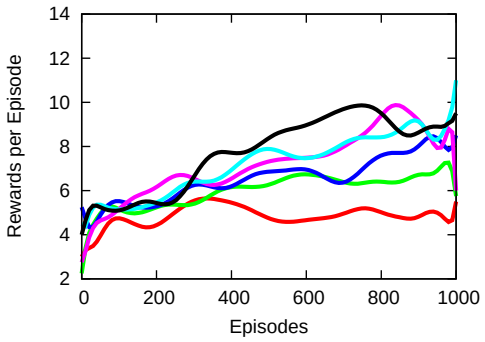
(b) Closest agent to any reward.



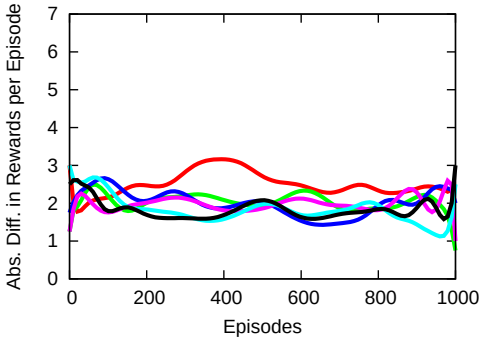
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

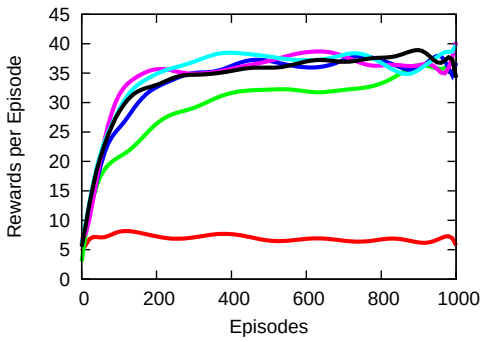


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

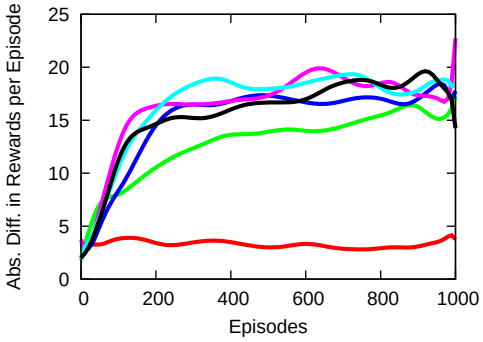


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

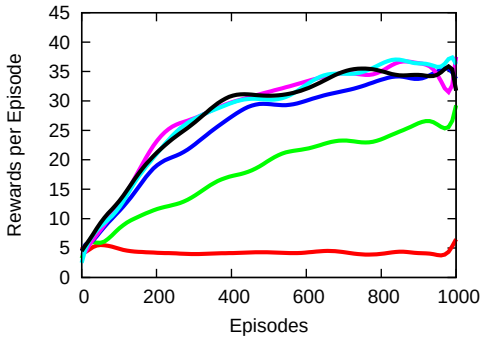
Figure A.35: Tabular value functions - A team of three agents and a single agent acting individually, one nearest reward in the state representation with constant ε values.



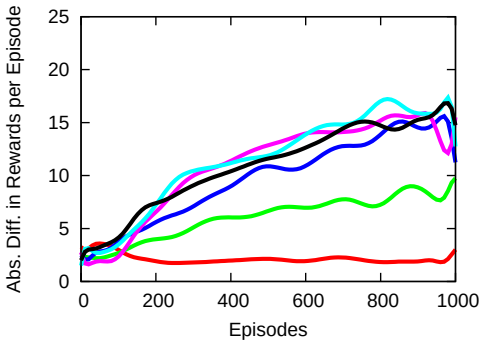
(a) Closest agent to any reward.



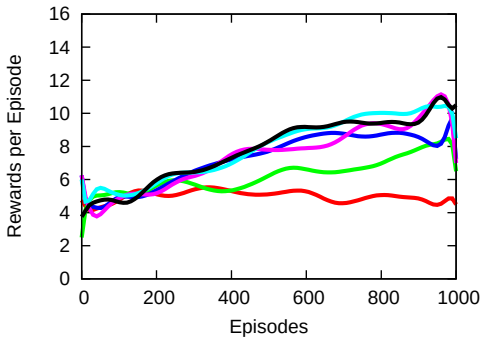
(b) Closest agent to any reward.



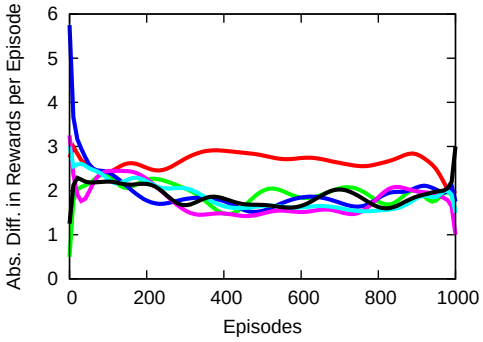
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

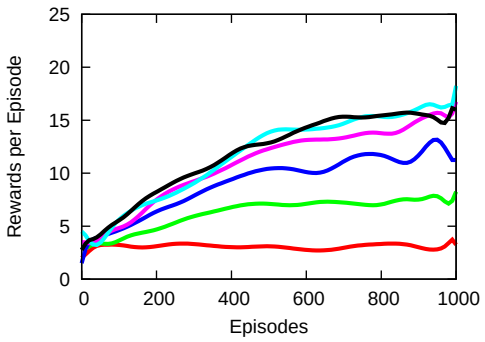


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

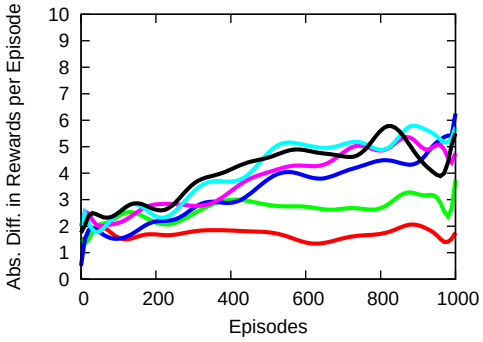


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

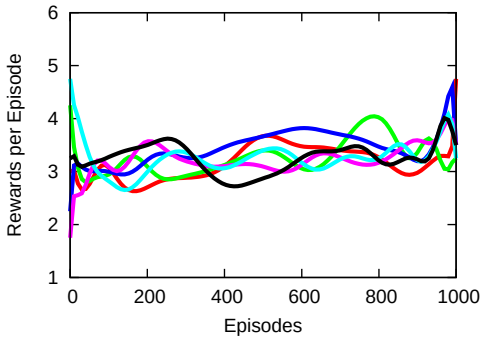
Figure A.36: Tabular value functions - A team of three agents and a single agent acting individually, one nearest reward in the state representation with degrading ε values.



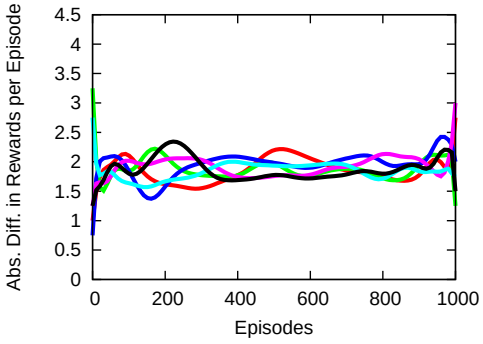
(a) Closest agent to any reward.



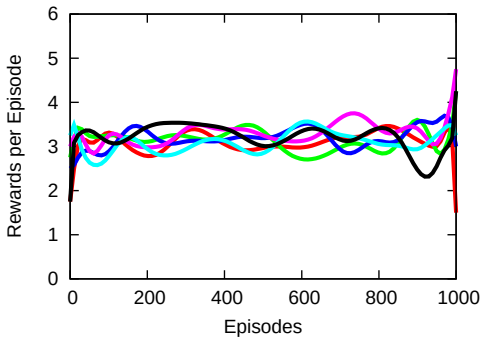
(b) Closest agent to any reward.



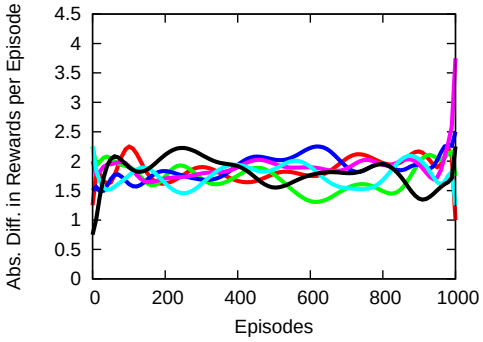
(c) Closest other agent to agent's nearest two rewards.



(d) Closest other agent to agent's nearest two rewards.

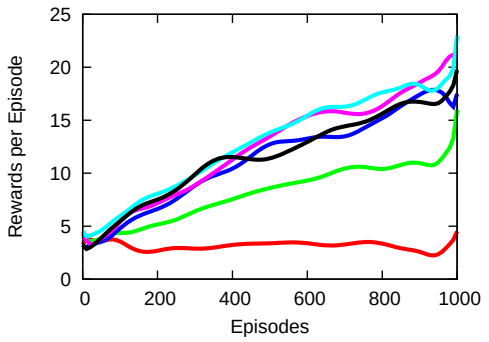


(e) Closest other agent to agent's nearest two rewards and closest agent to any reward.

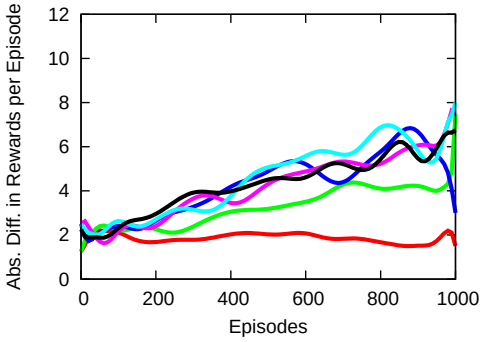


(f) Closest other agent to agent's nearest two rewards and closest agent to any reward.

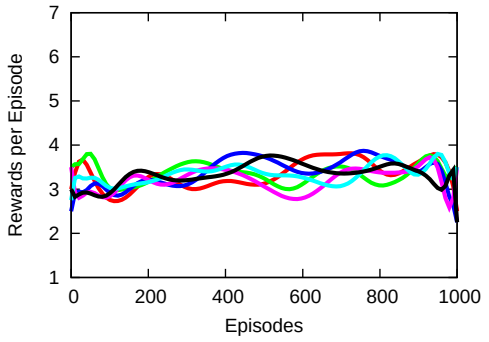
Figure A.37: Tabular value functions - A team of three agents and a single agent acting individually, two nearest rewards in the state representation with constant ε values.



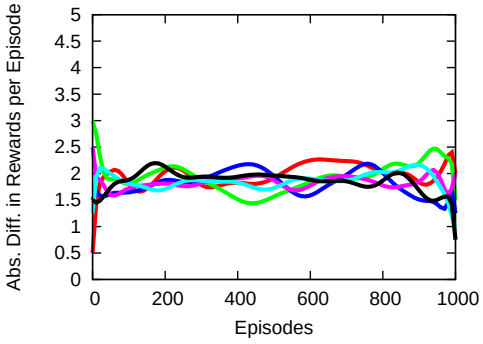
(a) Closest agent to any reward.



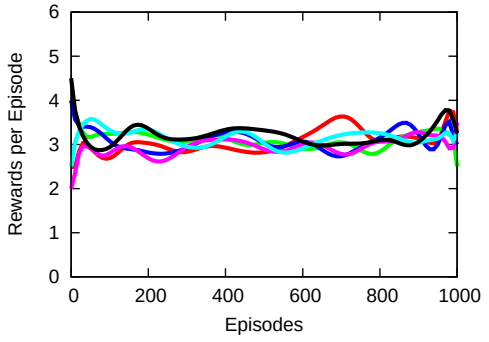
(b) Closest agent to any reward.



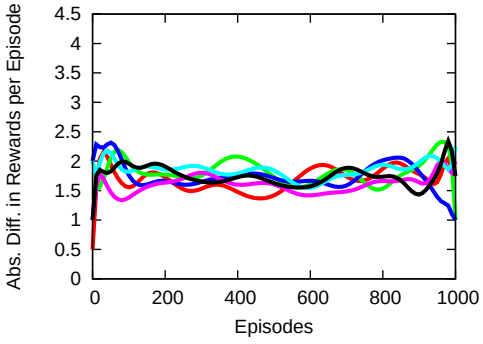
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

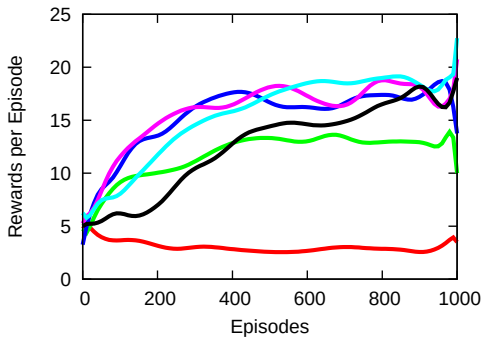


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

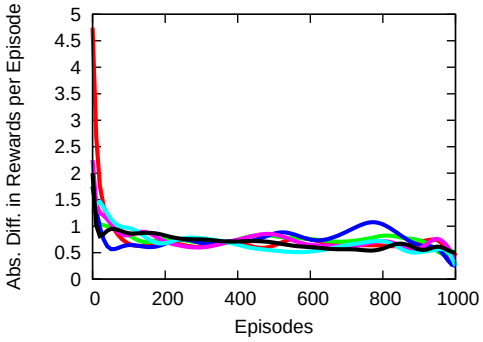


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

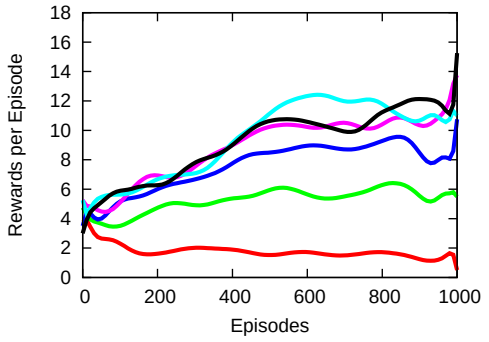
Figure A.38: Tabular value functions - A team of three agents and a single agent acting individually, two nearest rewards in the state representation with degrading ε values.



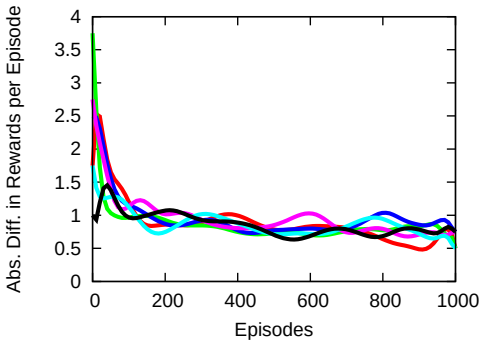
(a) Closest agent to any reward.



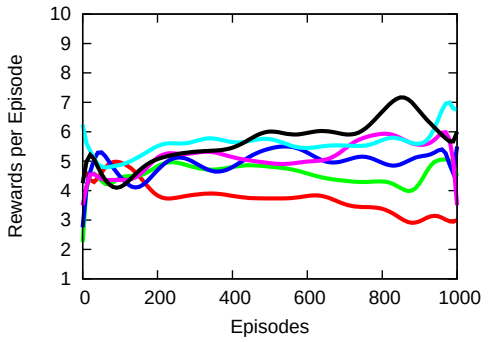
(b) Closest agent to any reward.



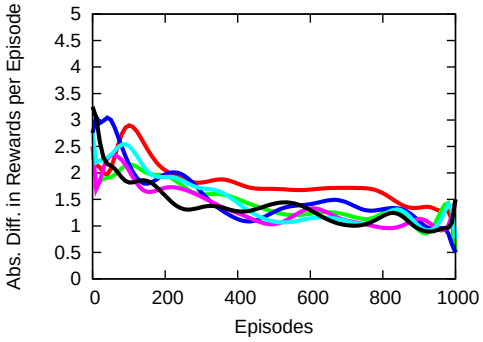
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

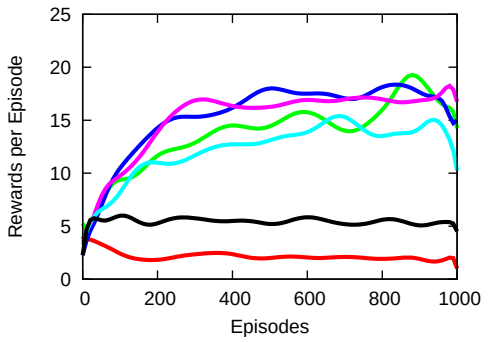


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

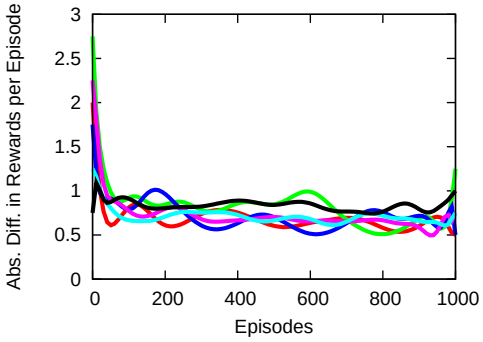


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

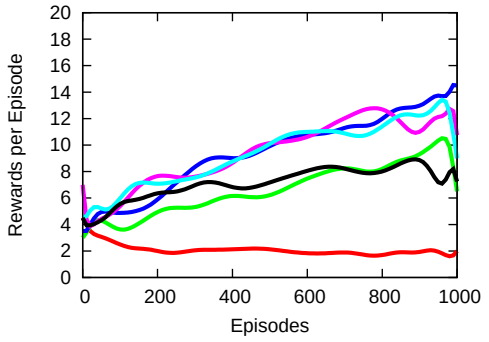
Figure A.39: Tabular value functions - A team of three agents and a single agent in balance, one nearest reward in the state representation with constant ε values.



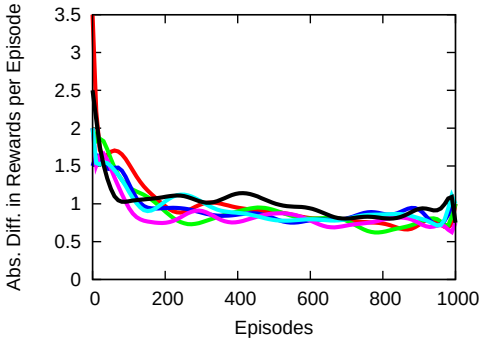
(a) Closest agent to any reward.



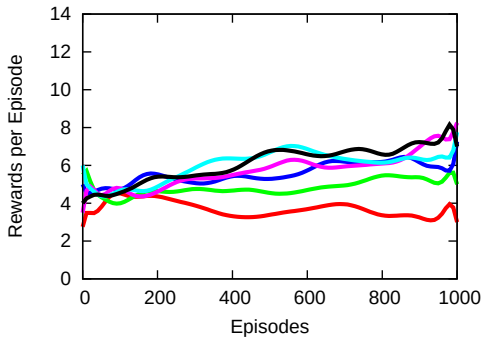
(b) Closest agent to any reward.



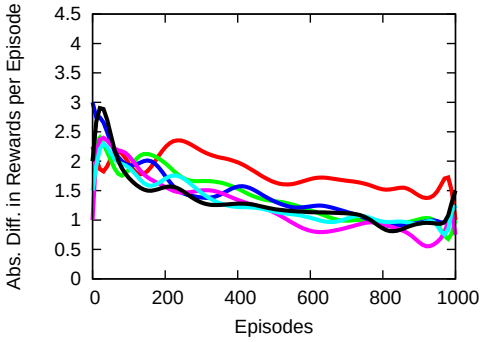
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

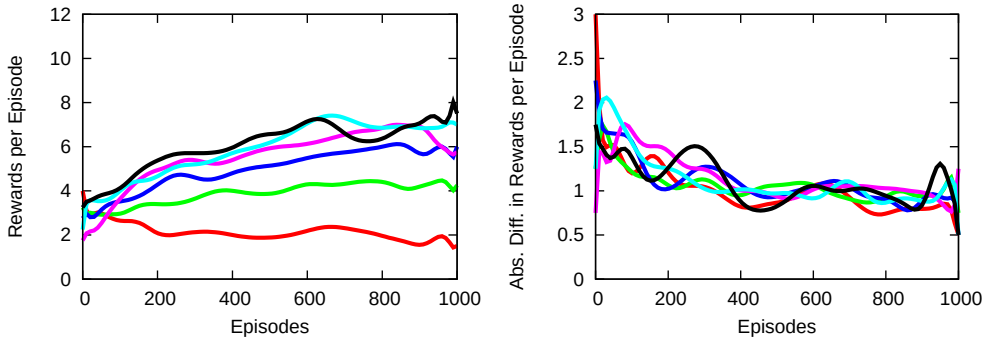


(e) Closest other agent to agent's nearest reward and closest agent to any reward.



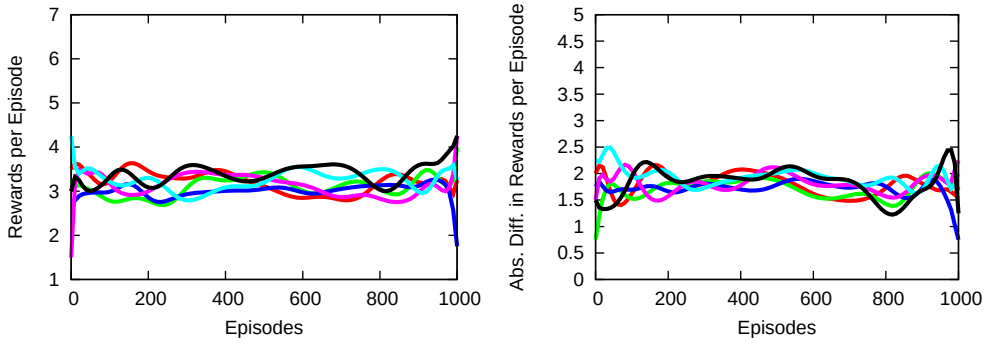
(f) Closest other agent to agent's nearest reward and closest agent to any reward.

Figure A.40: Tabular value functions - A team of three agents and a single agent in balance, one nearest reward in the state representation with degrading ε values.



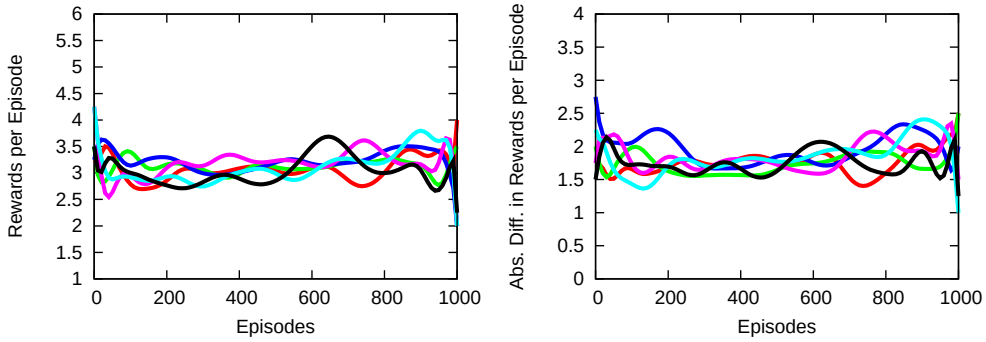
(a) Closest agent to any reward.

(b) Closest agent to any reward.



(c) Closest other agent to agent's two nearest rewards.

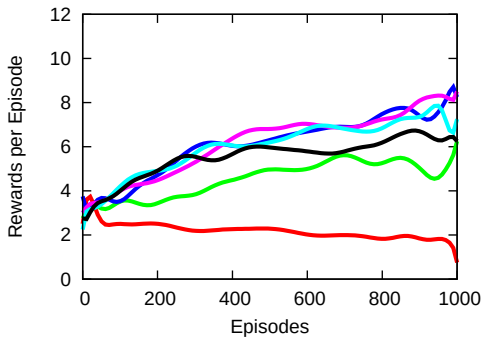
(d) Closest other agent to agent's two nearest rewards.



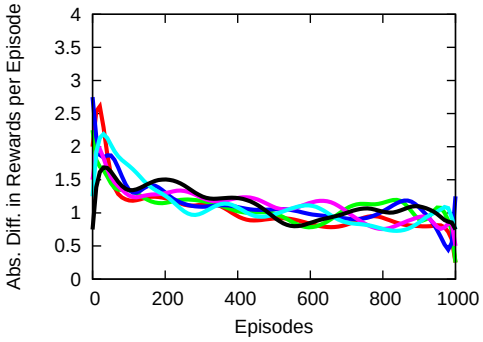
(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

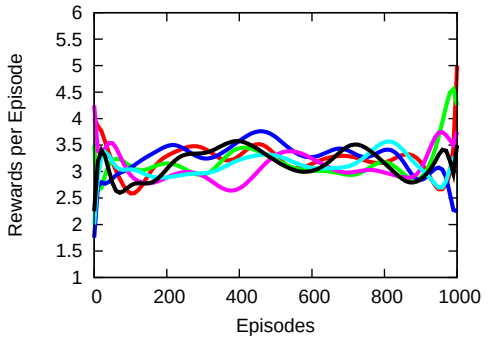
Figure A.41: Tabular value functions - A team of three agents and a single agent in balance, two nearest rewards in the state representation with constant ε values.



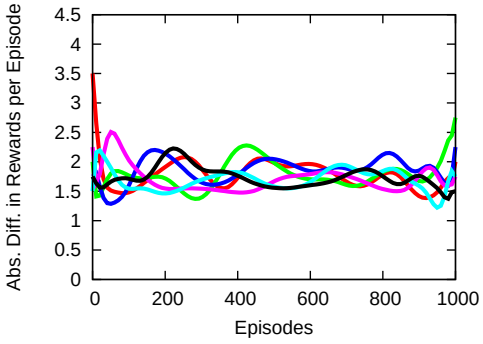
(a) Closest agent to any reward.



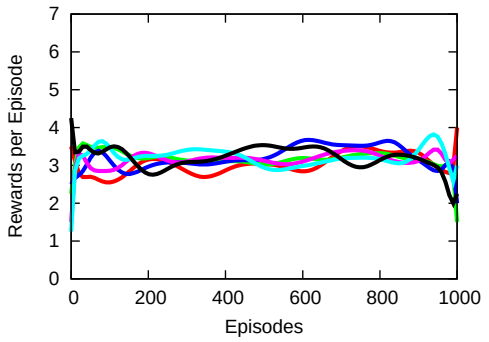
(b) Closest agent to any reward.



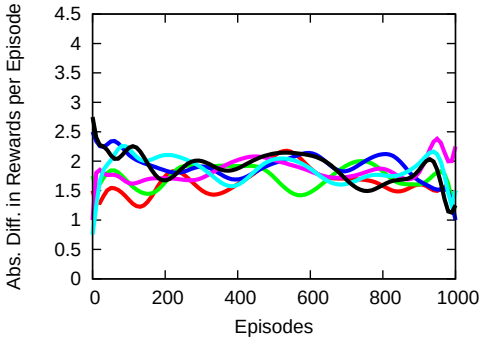
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.



(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.



(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

Figure A.42: Tabular value functions - A team of three agents and a single agent in balance, two nearest rewards in the state representation with degrading ε values.

Appendix B

All Experimental Graphs with Approximated Value Functions

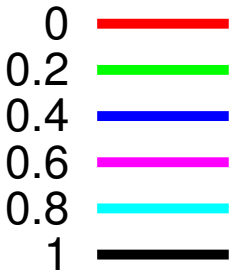
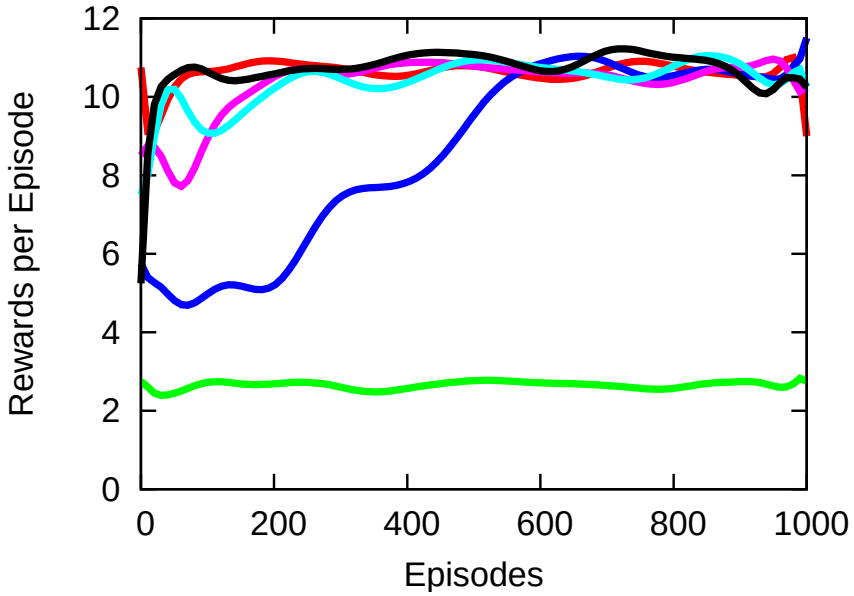
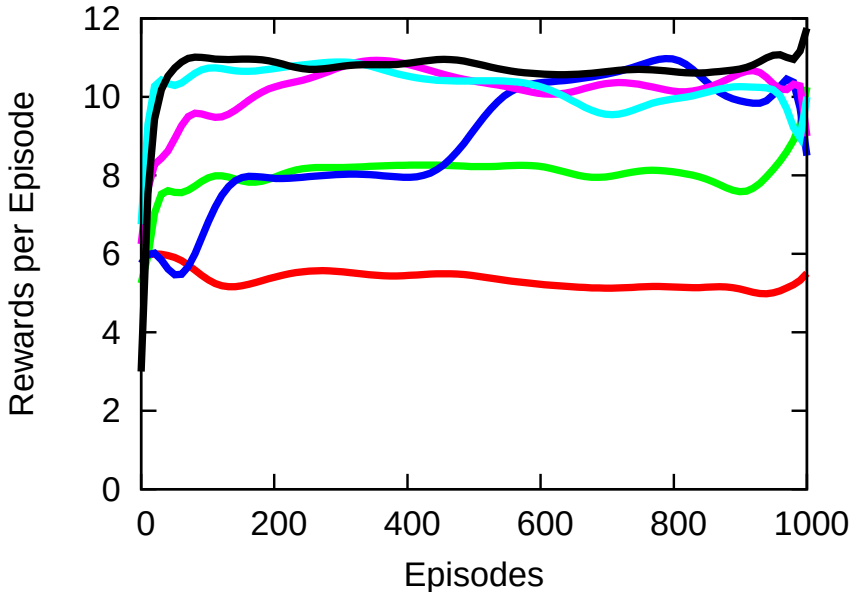


Figure B.1: The legend for each graph in the appendices. The colours represent different values of ε.

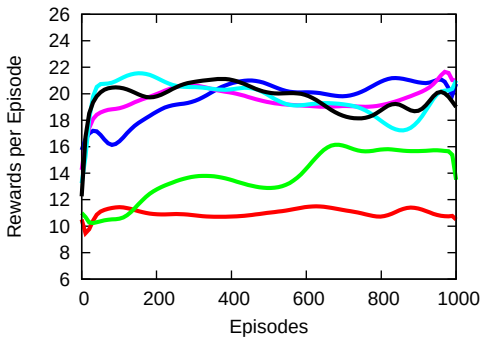


(a) Constant ε values.

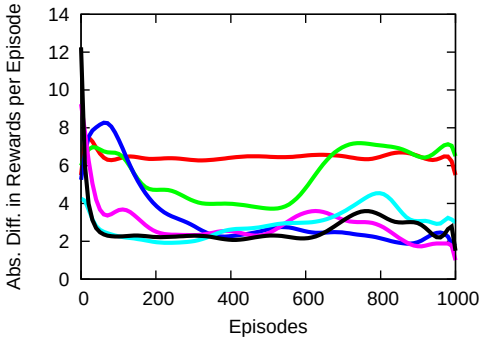


(b) Degrading ε values.

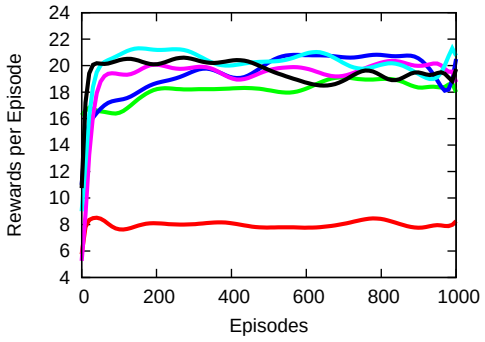
Figure B.2: Approximated value functions - An individual agent, one nearest reward in the state representation.



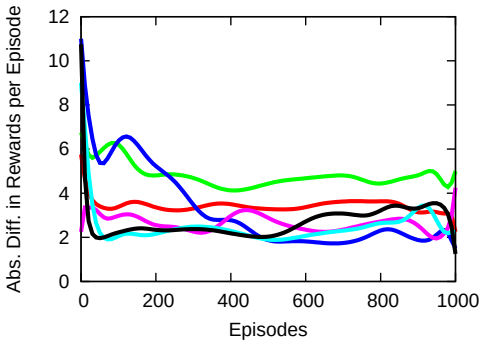
(a) Closest agent to any reward.



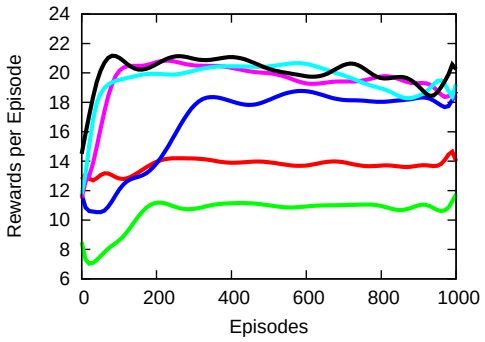
(b) Closest agent to any reward.



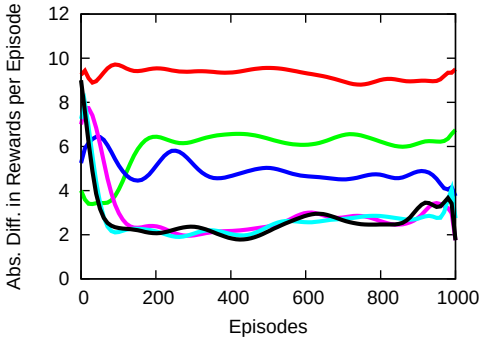
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

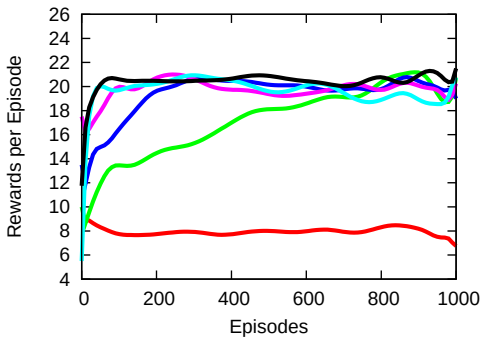


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

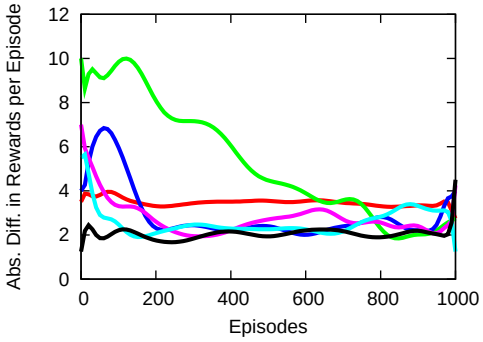


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

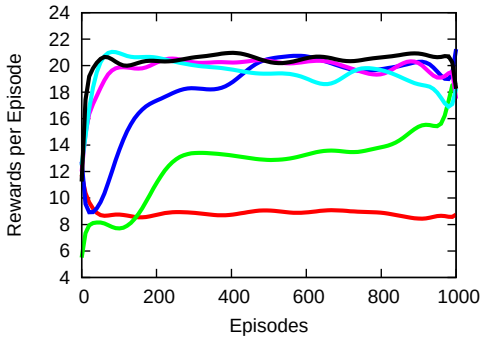
Figure B.3: Approximated value functions - Two individual agents, one nearest reward in the state representation with constant ε values.



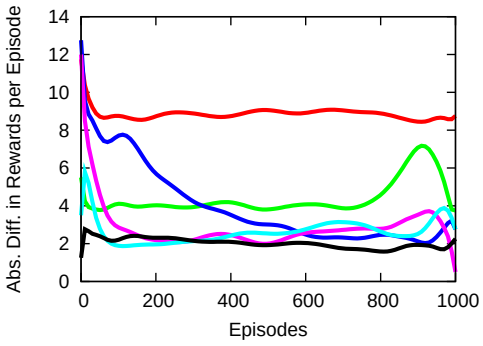
(a) Closest agent to any reward.



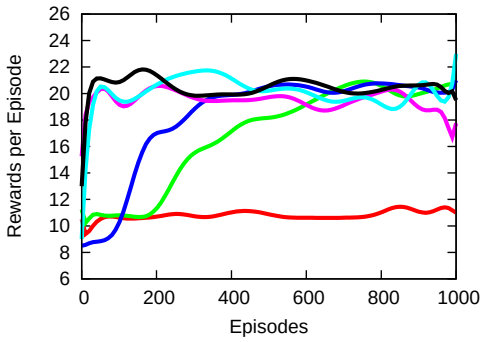
(b) Closest agent to any reward.



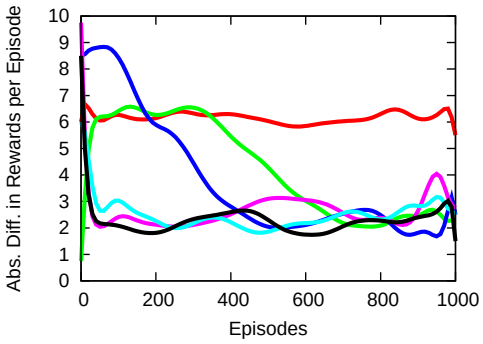
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

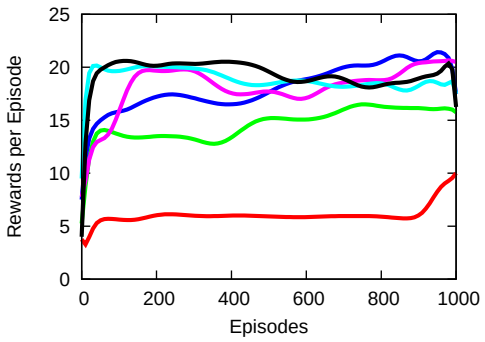


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

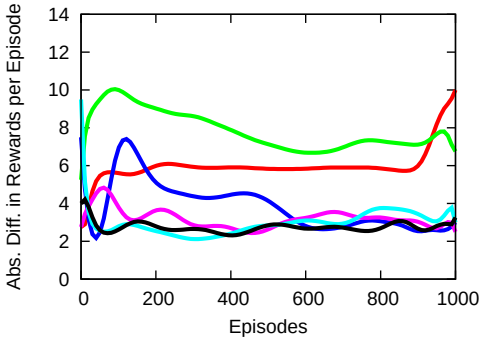


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

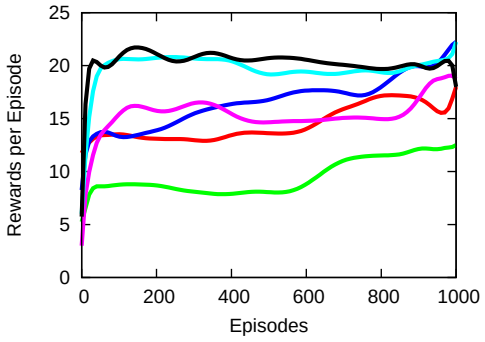
Figure B.4: Approximated value functions - Two individual agents, one nearest reward in the state representation with degrading ε values.



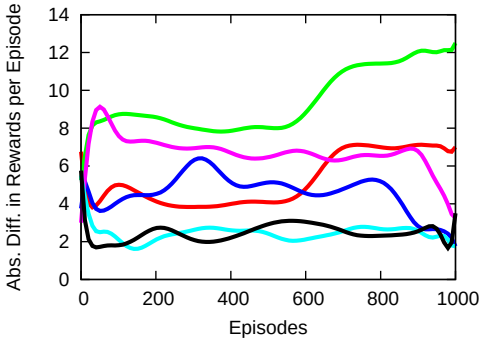
(a) Closest agent to any reward.



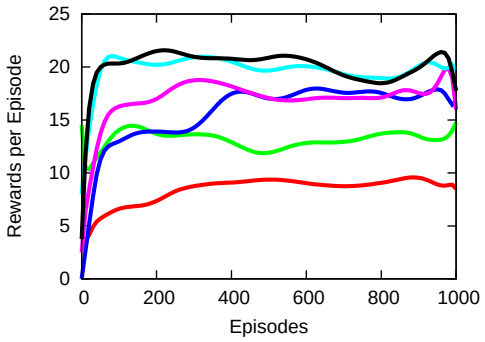
(b) Closest agent to any reward.



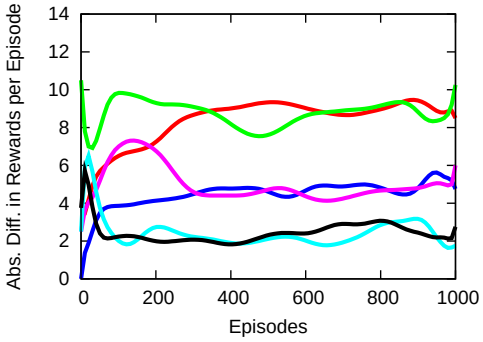
(c) Closest other agent to agent's nearest two rewards.



(d) Closest other agent to agent's nearest two rewards.

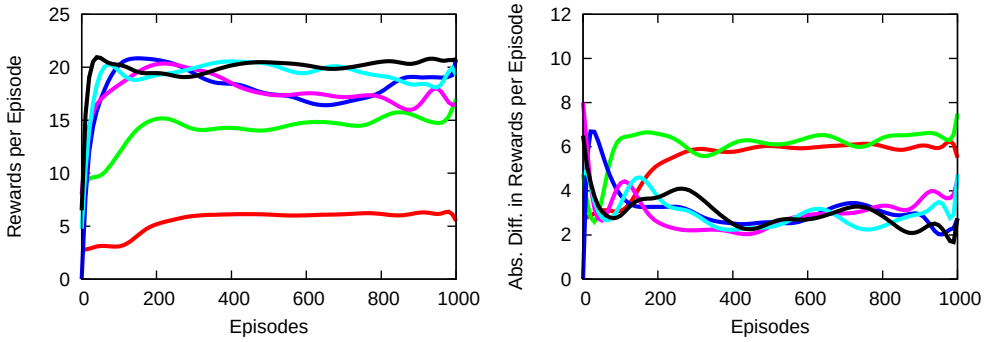


(e) Closest other agent to agent's nearest two rewards and closest agent to any reward.



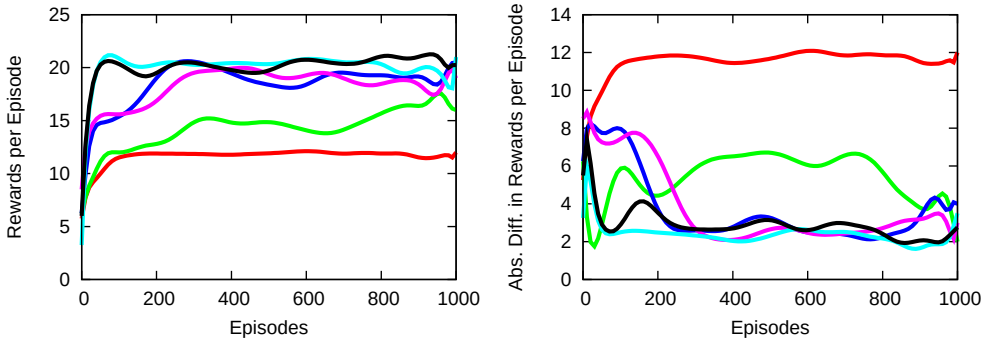
(f) Closest other agent to agent's nearest two rewards and closest agent to any reward.

Figure B.5: Approximated value functions - Two individual agents, two nearest rewards in the state representation with constant ε values.



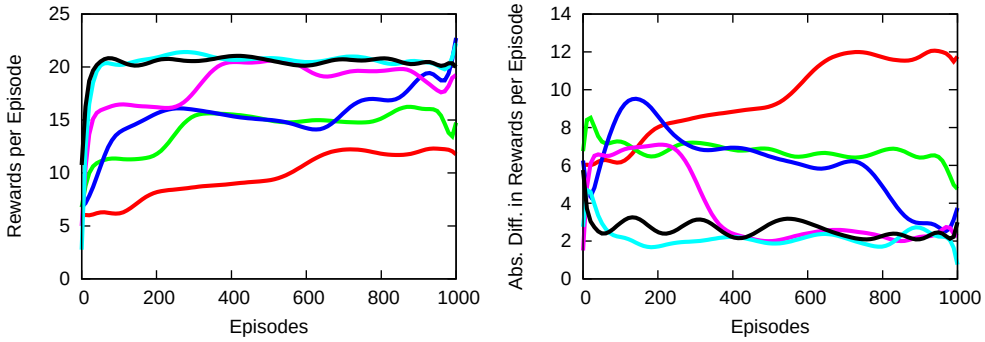
(a) Closest agent to any reward.

(b) Closest agent to any reward.



(c) Closest other agent to agent's two nearest rewards.

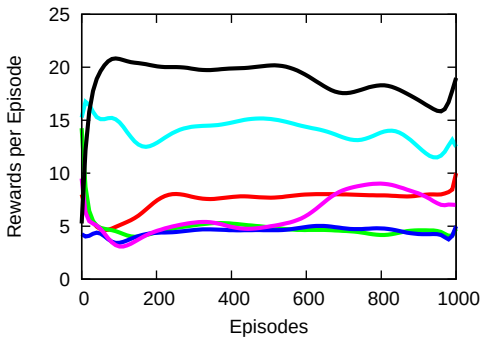
(d) Closest other agent to agent's two nearest rewards.



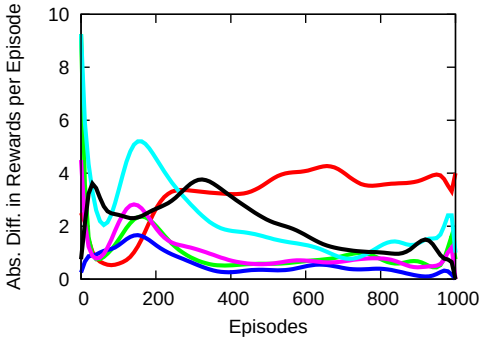
(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

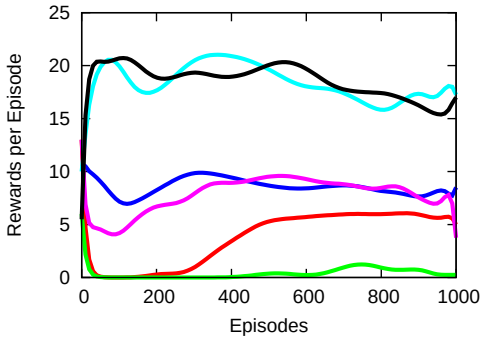
Figure B.6: Approximated value functions - Two individual agents, two nearest rewards in the state representation with degrading ε values.



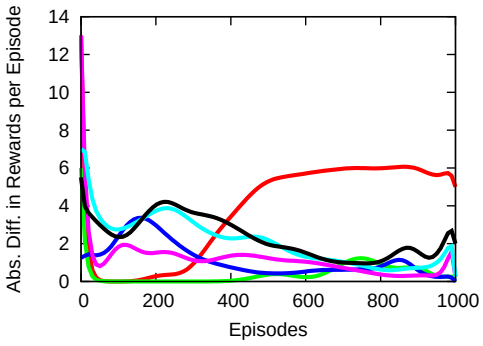
(a) Closest agent to any reward.



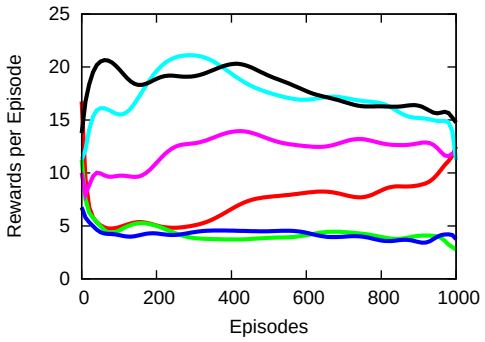
(b) Closest agent to any reward.



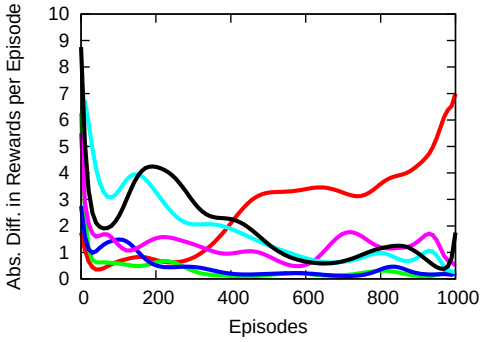
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

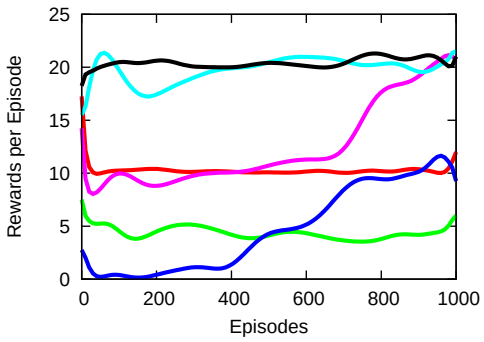


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

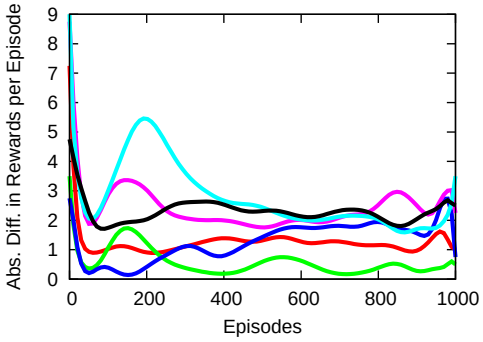


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

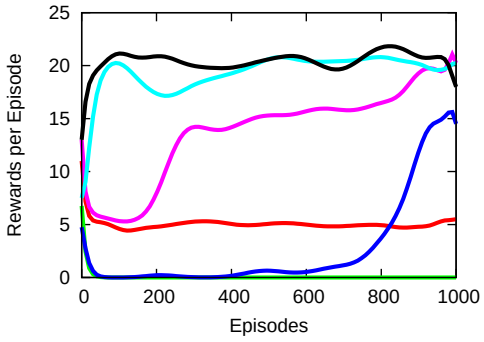
Figure B.7: Approximated value functions - Two balanced agents, one nearest reward in the state representation with constant ε values.



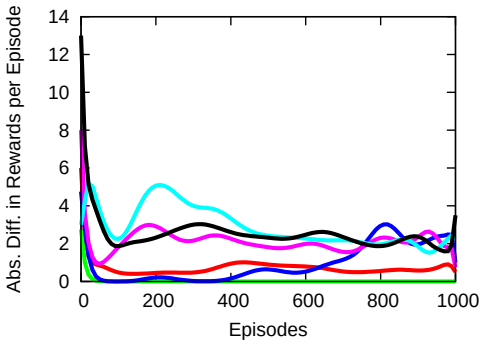
(a) Closest agent to any reward.



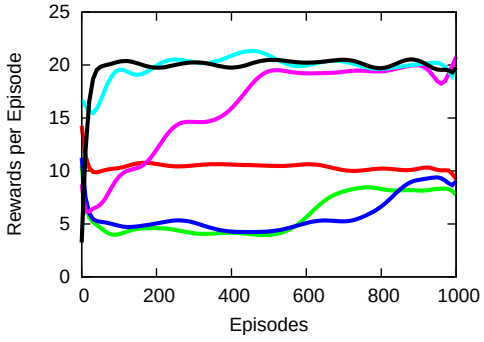
(b) Closest agent to any reward.



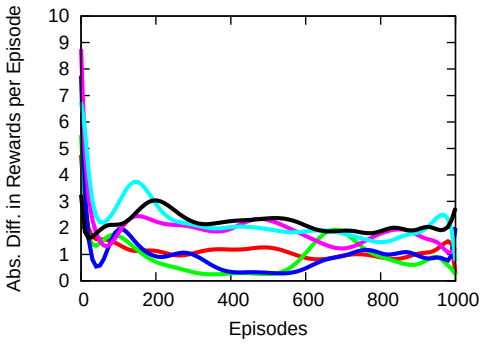
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

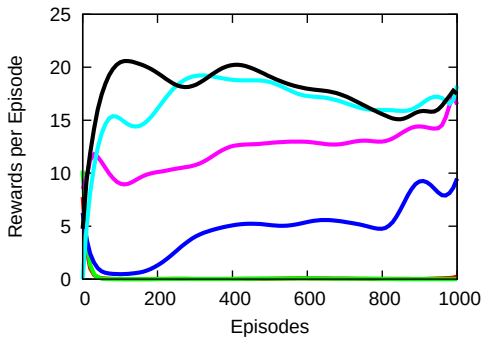


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

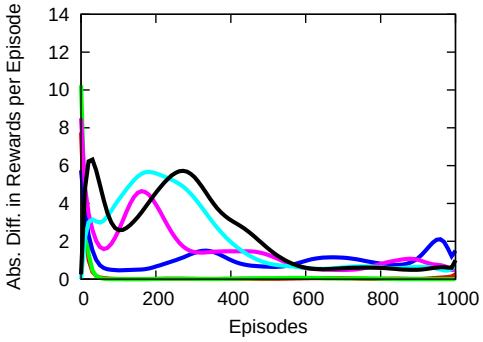


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

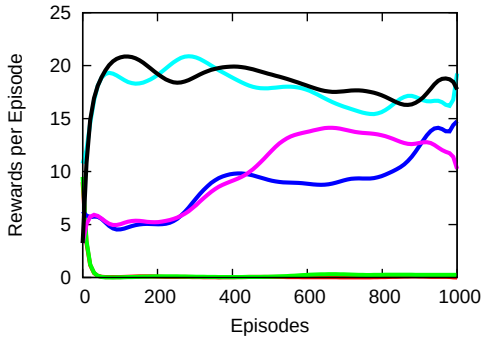
Figure B.8: Approximated value functions - Two balanced agents, one nearest reward in the state representation with degrading ε values.



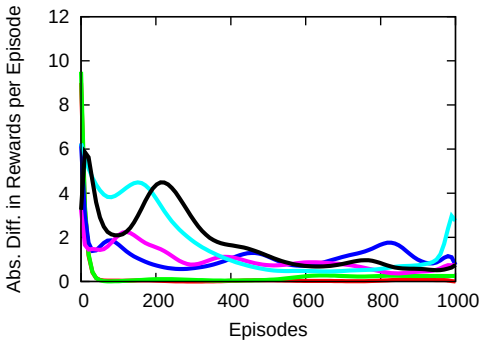
(a) Closest agent to any reward.



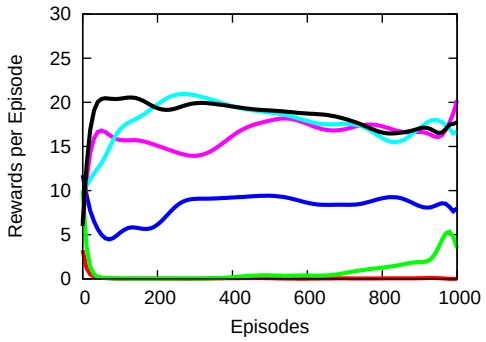
(b) Closest agent to any reward.



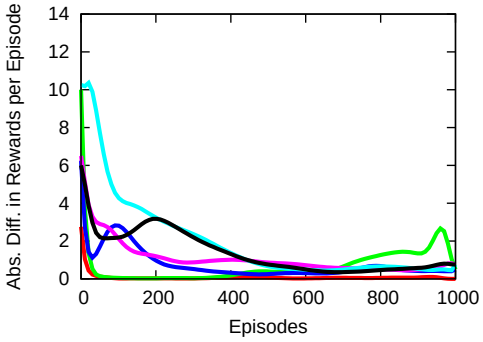
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

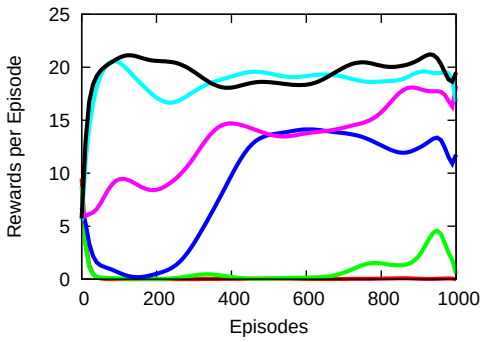


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

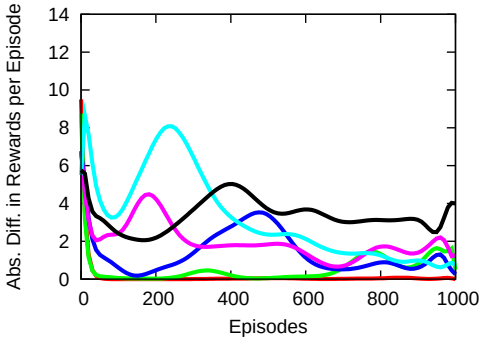


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

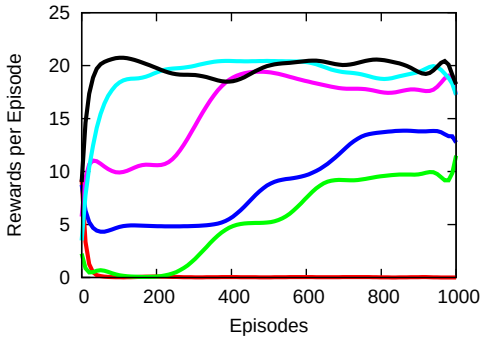
Figure B.9: Approximated value functions - Two balanced agents, two nearest rewards in the state representation with constant ε values.



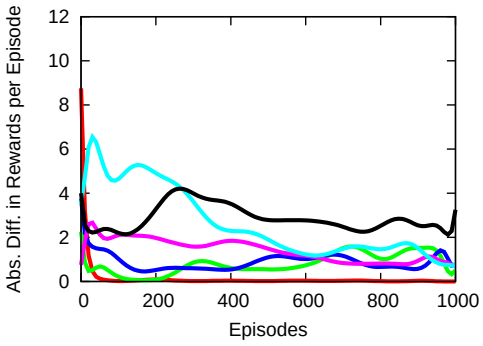
(a) Closest agent to any reward.



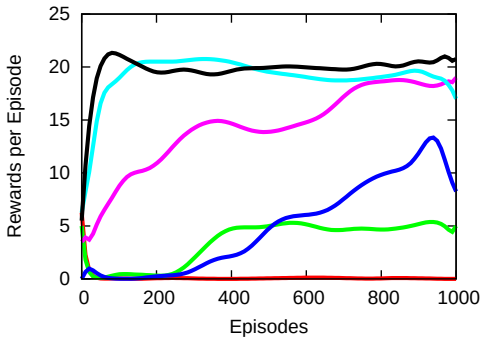
(b) Closest agent to any reward.



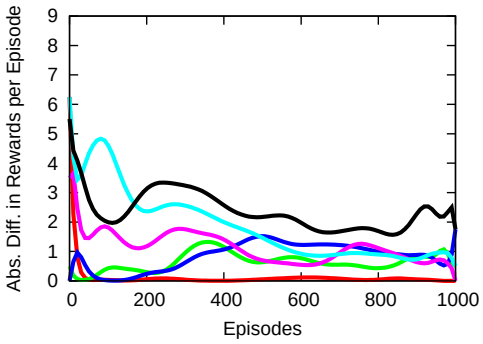
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

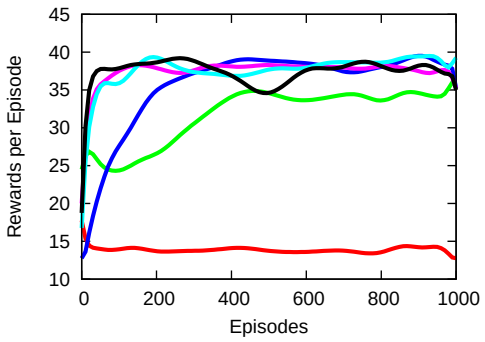


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

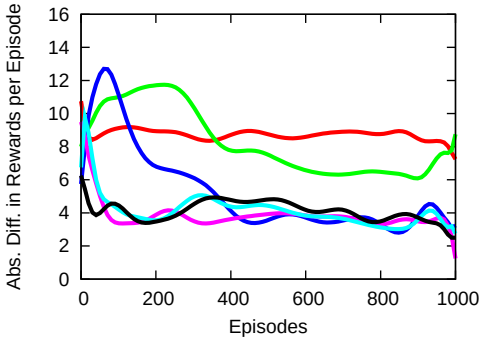


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

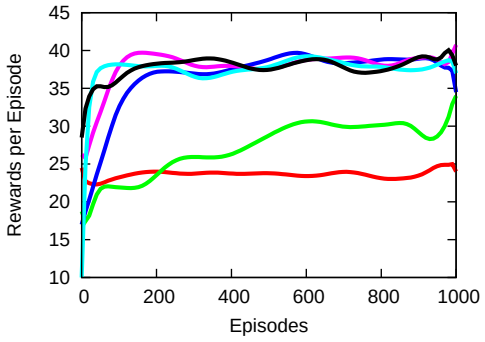
Figure B.10: Approximated value functions - Two balanced agents, two nearest rewards in the state representation with degrading ε values.



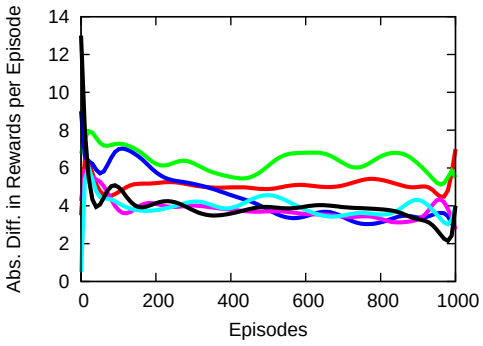
(a) Closest agent to any reward.



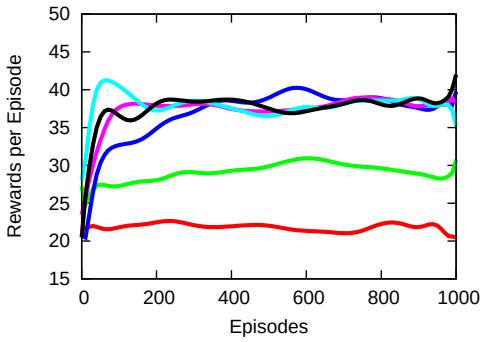
(b) Closest agent to any reward.



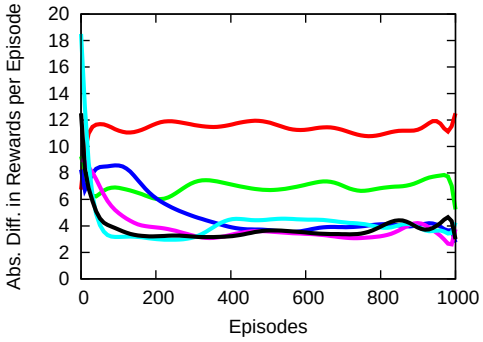
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

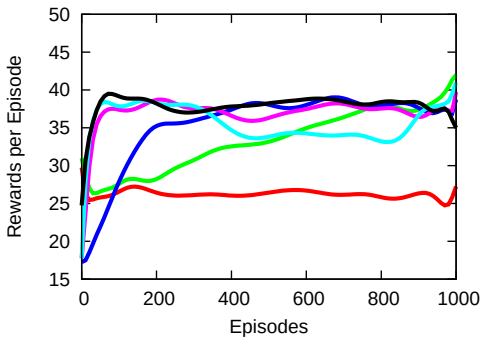


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

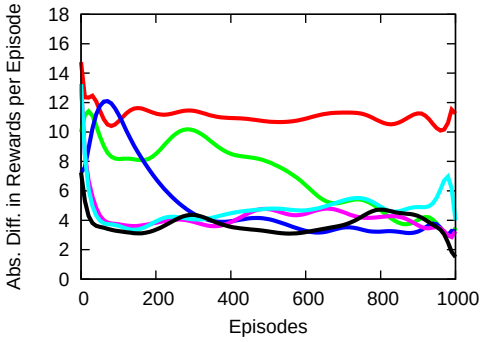


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

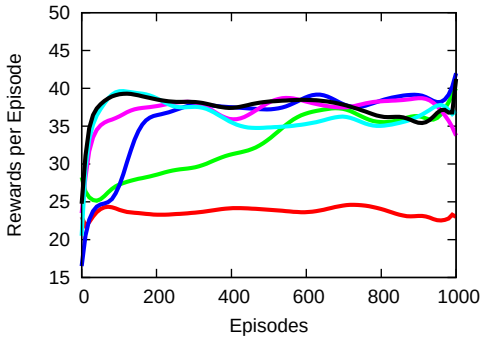
Figure B.11: Approximated value functions - Two individual teams of two agents each, one nearest reward in the state representation with constant ε values.



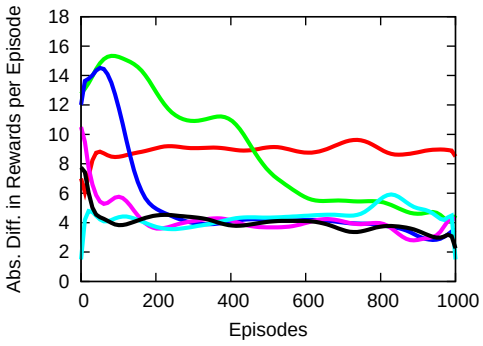
(a) Closest agent to any reward.



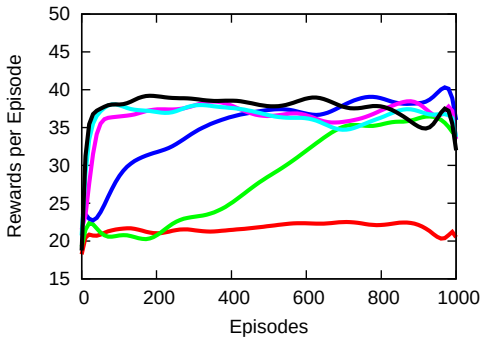
(b) Closest agent to any reward.



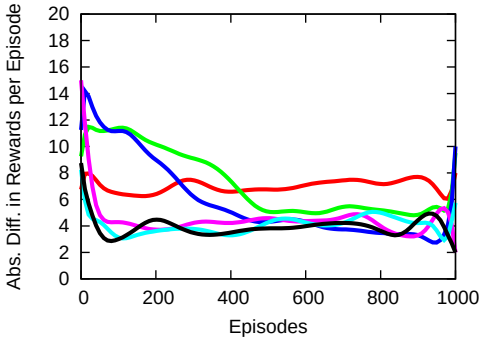
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

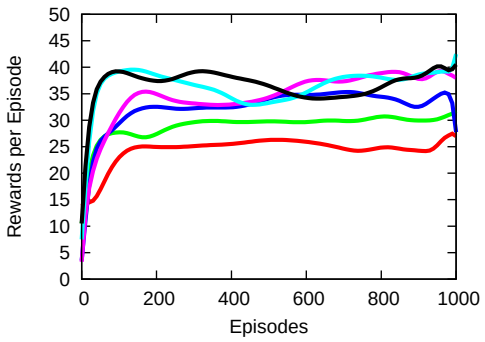


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

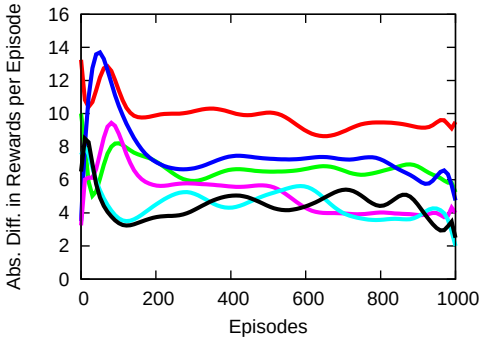


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

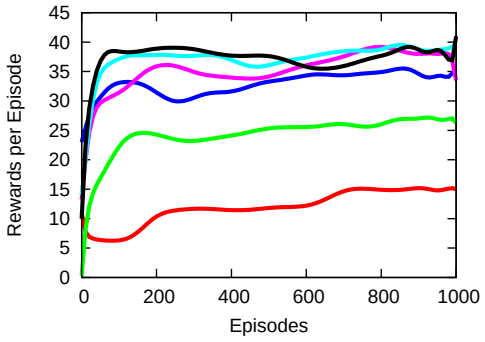
Figure B.12: Approximated value functions - Two individual teams of two agents each, one nearest reward in the state representation with degrading ε values.



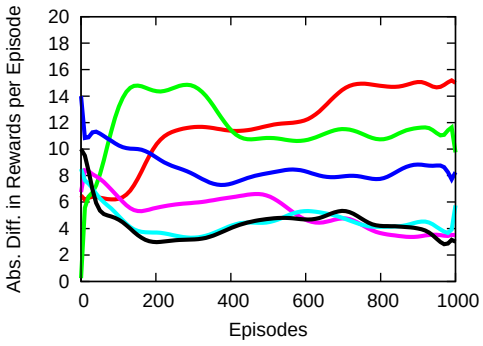
(a) Closest agent to any reward.



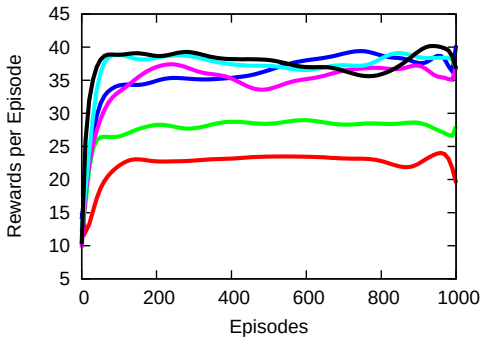
(b) Closest agent to any reward.



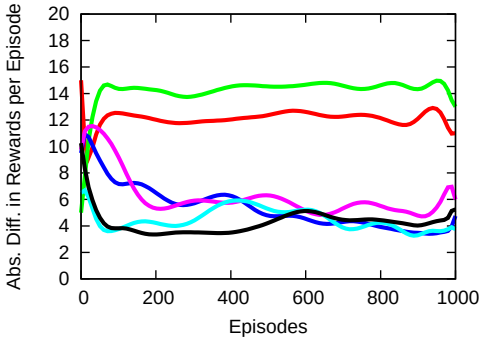
(c) Closest other agent to agent's nearest two rewards.



(d) Closest other agent to agent's nearest two rewards.

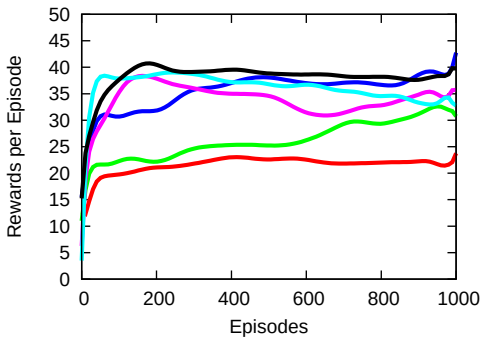


(e) Closest other agent to agent's nearest two rewards and closest agent to any reward.

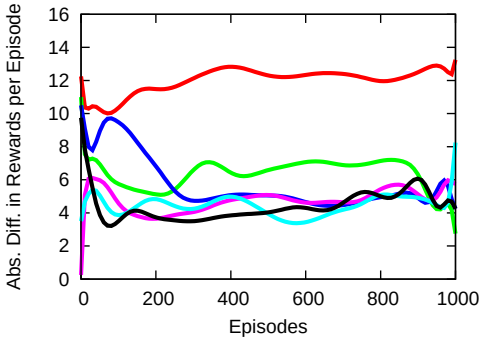


(f) Closest other agent to agent's nearest two rewards and closest agent to any reward.

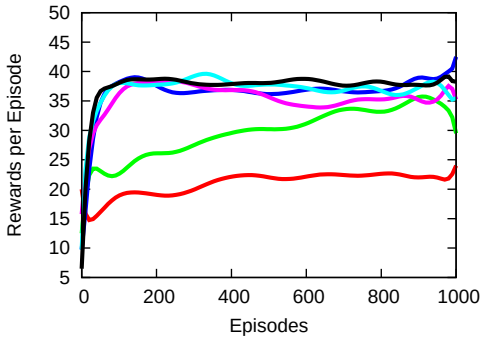
Figure B.13: Approximated value functions - Two individual teams of two agents each, two nearest rewards in the state representation with constant ε values.



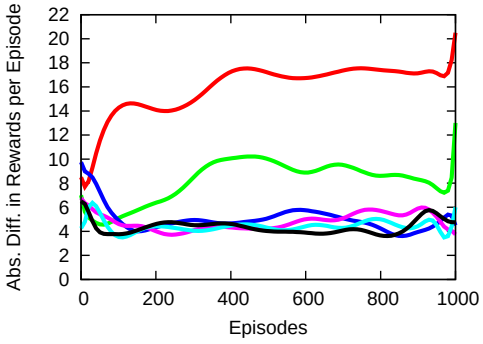
(a) Closest agent to any reward.



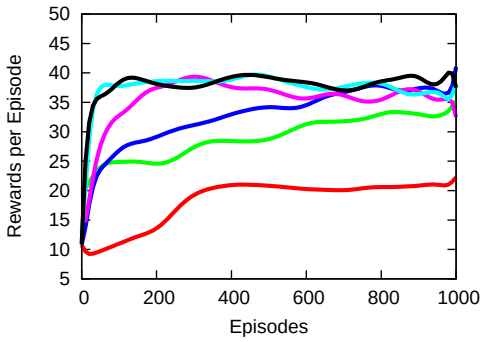
(b) Closest agent to any reward.



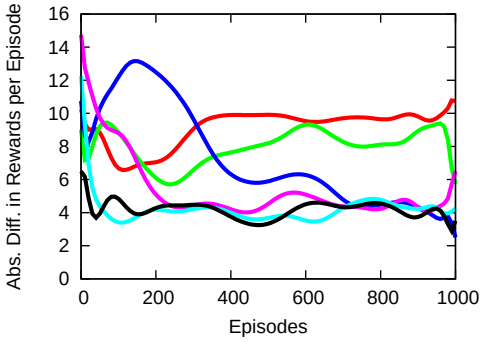
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

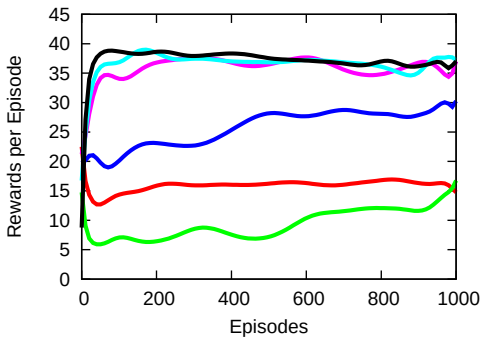


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

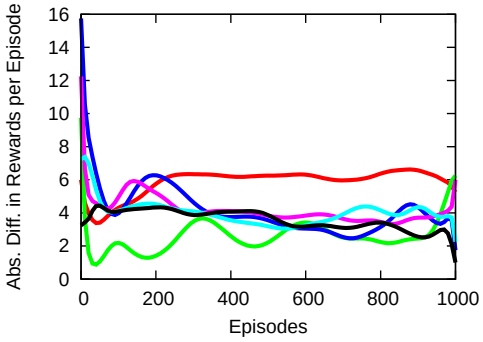


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

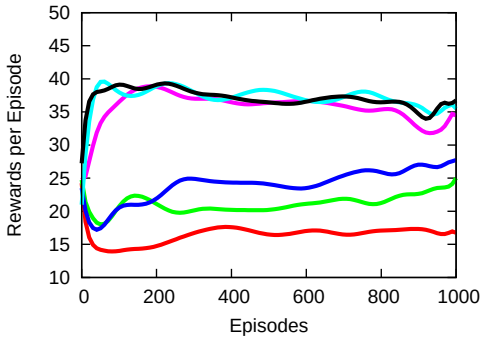
Figure B.14: Approximated value functions - Two individual teams of two agents each, two nearest rewards in the state representation with degrading ε values.



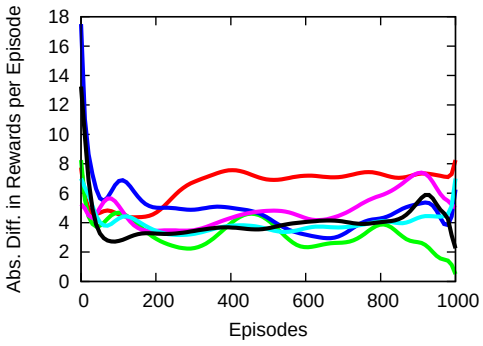
(a) Closest agent to any reward.



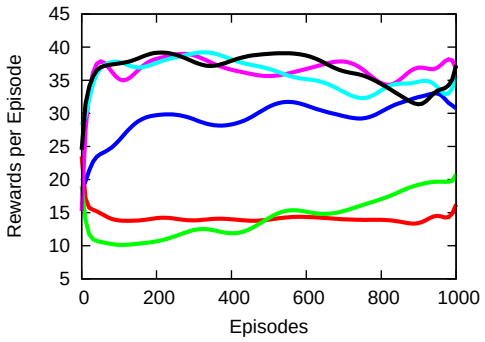
(b) Closest agent to any reward.



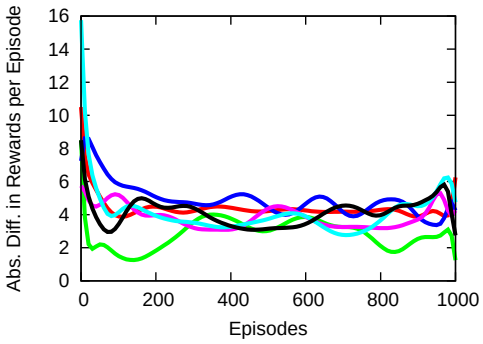
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

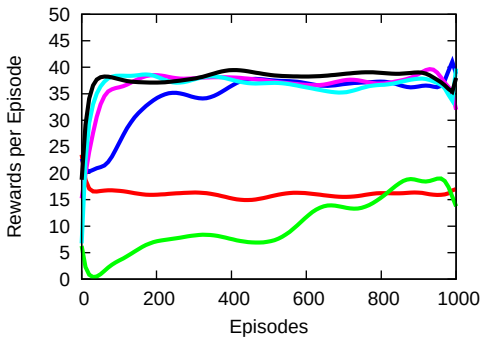


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

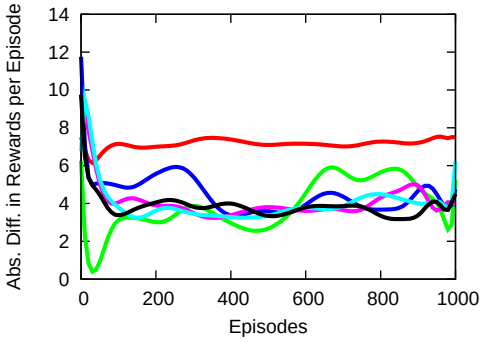


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

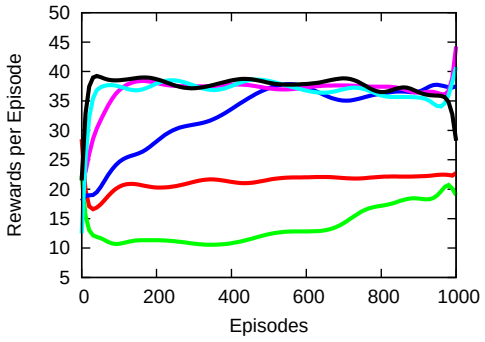
Figure B.15: Approximated value functions - Two balanced teams of two agents each, one nearest reward in the state representation with constant ε values.



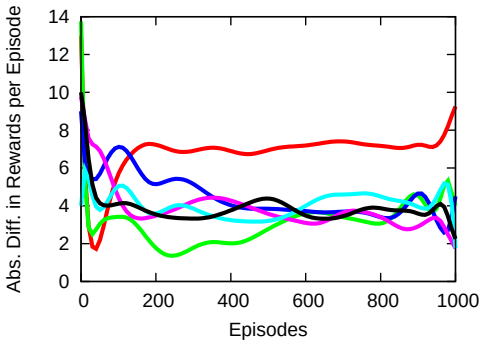
(a) Closest agent to any reward.



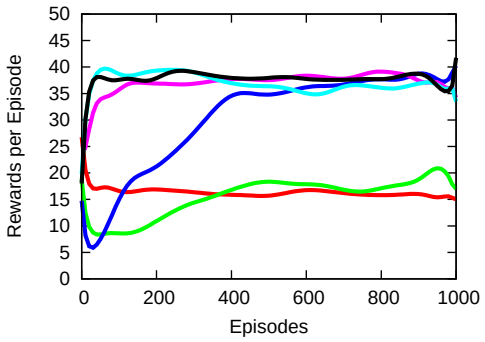
(b) Closest agent to any reward.



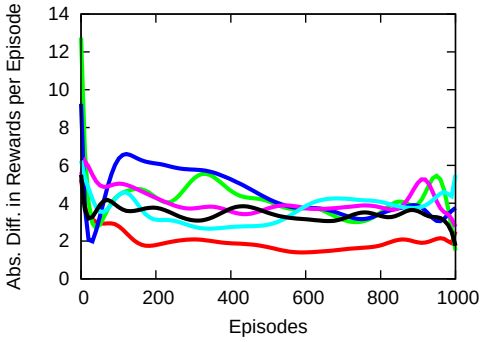
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

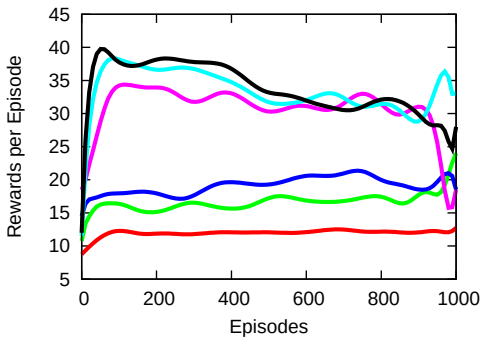


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

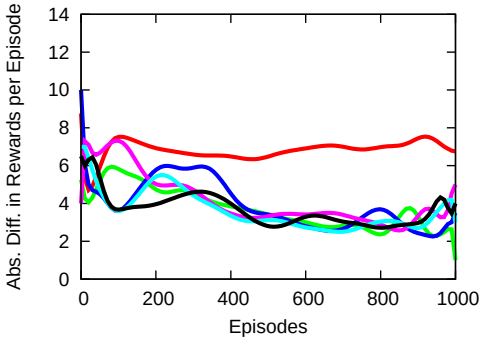


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

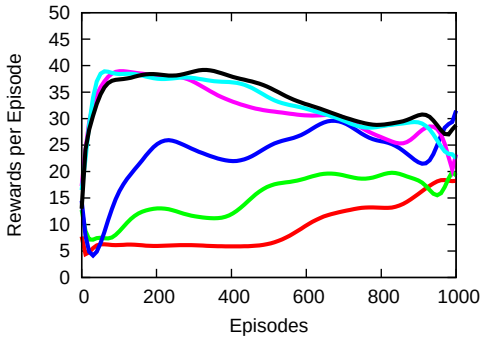
Figure B.16: Approximated value functions - Two balanced teams of two agents each, one nearest reward in the state representation with degrading ε values.



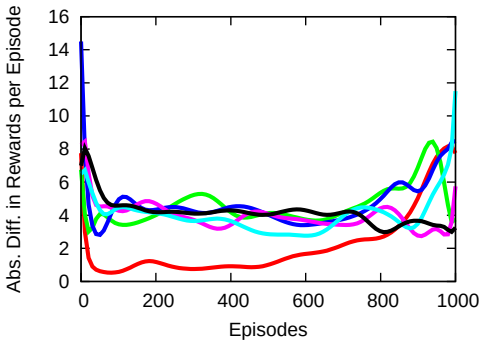
(a) Closest agent to any reward.



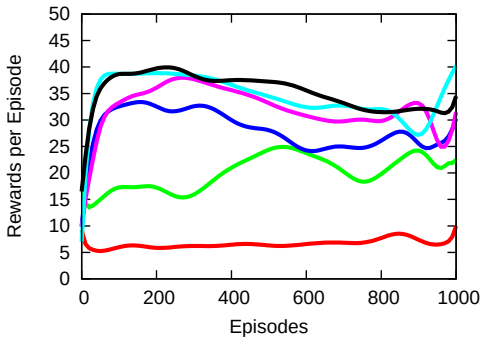
(b) Closest agent to any reward.



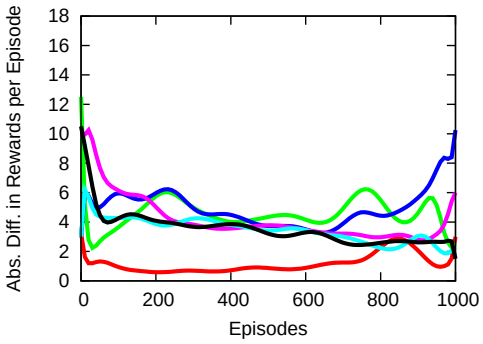
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

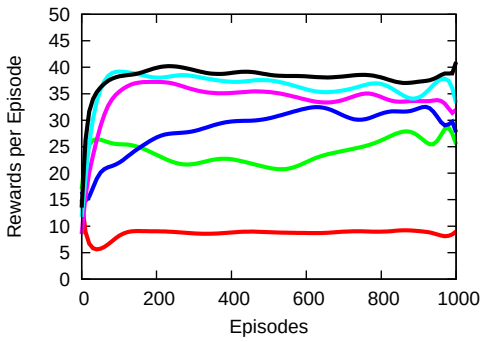


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

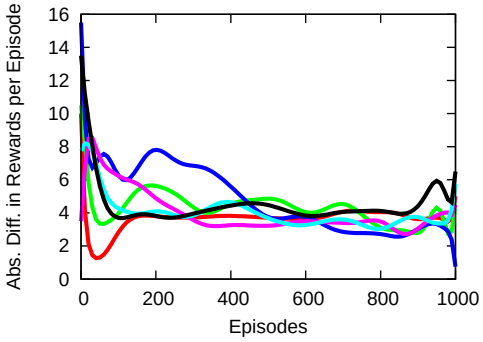


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

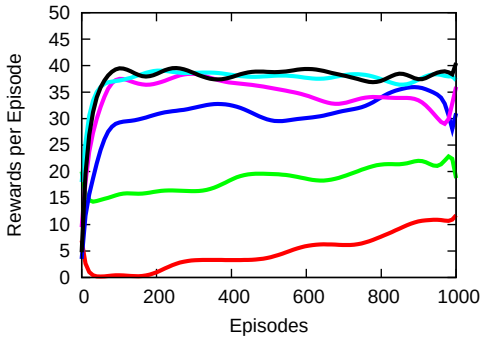
Figure B.17: Approximated value functions - Two balanced teams of two agents each, two nearest rewards in the state representation with constant ε values.



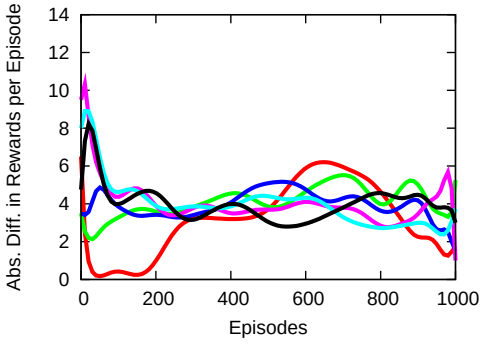
(a) Closest agent to any reward.



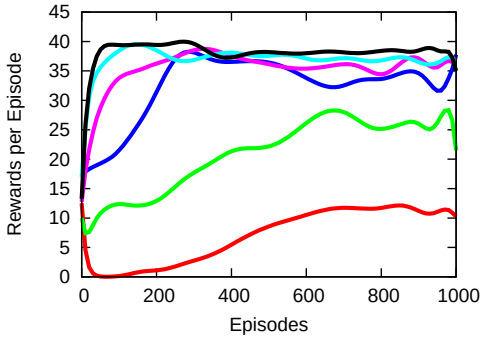
(b) Closest agent to any reward.



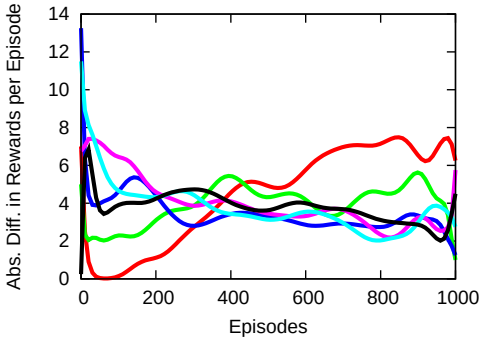
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

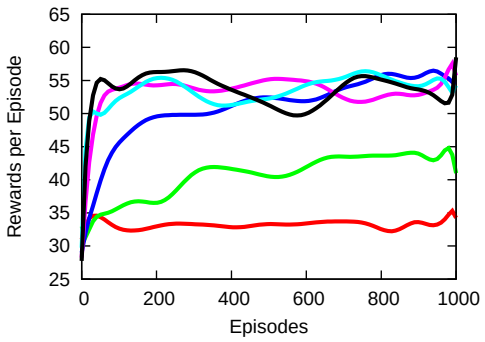


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

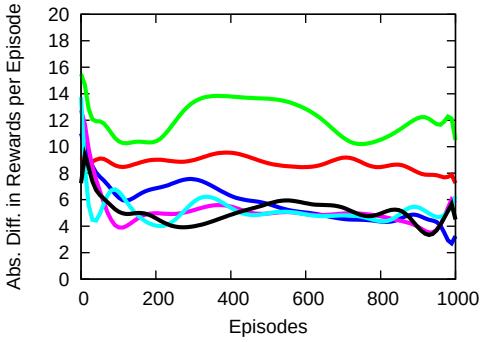


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

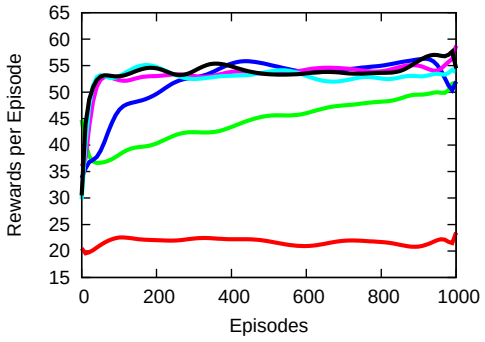
Figure B.18: Approximated value functions - Two balanced teams of two agents each, two nearest rewards in the state representation with degrading ε values.



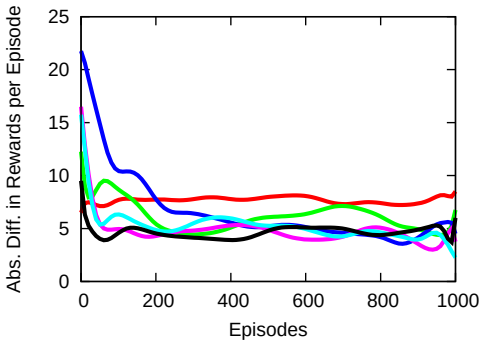
(a) Closest agent to any reward.



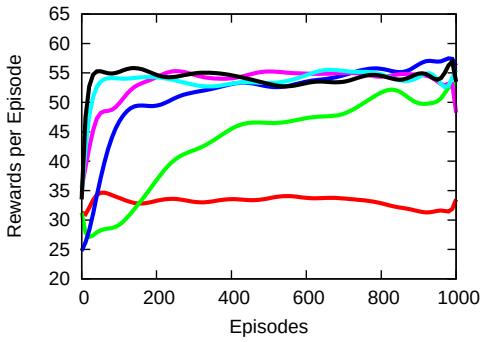
(b) Closest agent to any reward.



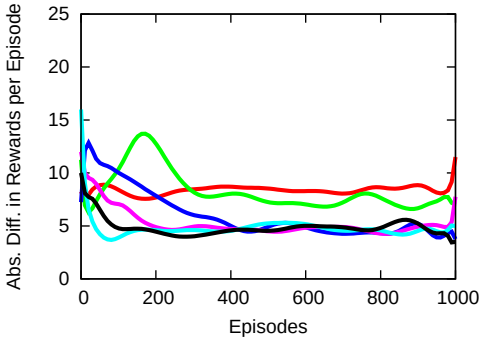
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

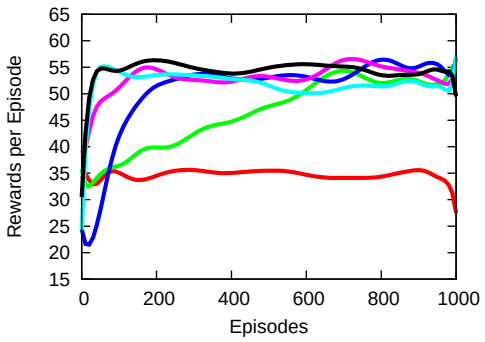


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

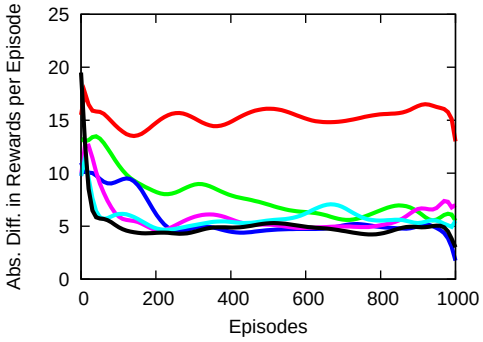


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

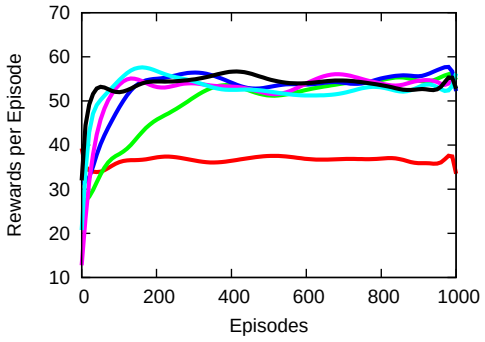
Figure B.19: Approximated value functions - Two individual teams of three agents each, one nearest reward in the state representation with constant ε values.



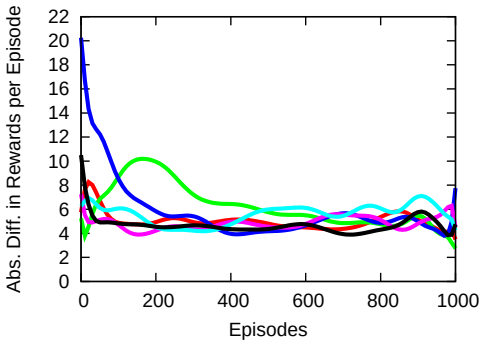
(a) Closest agent to any reward.



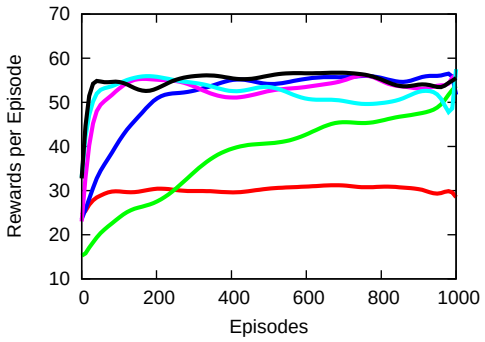
(b) Closest agent to any reward.



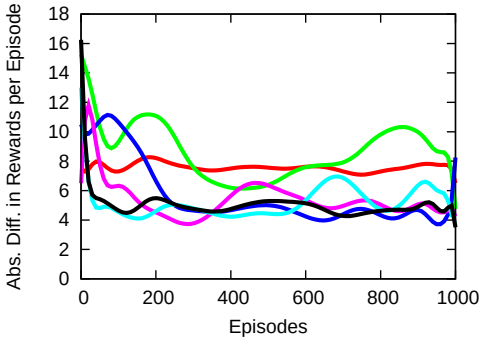
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

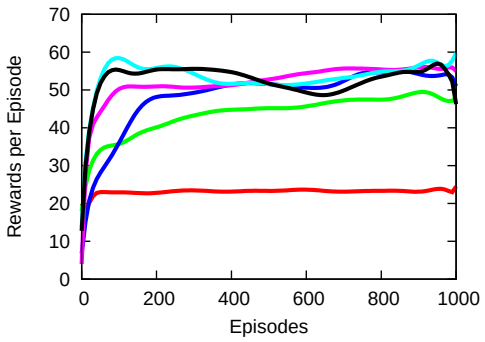


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

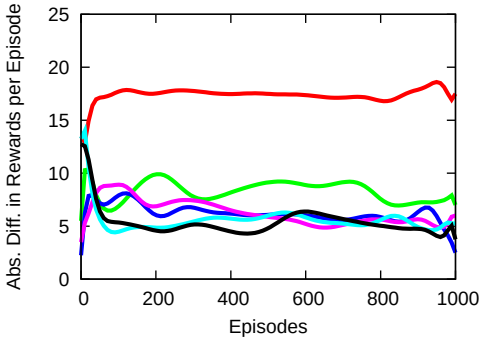


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

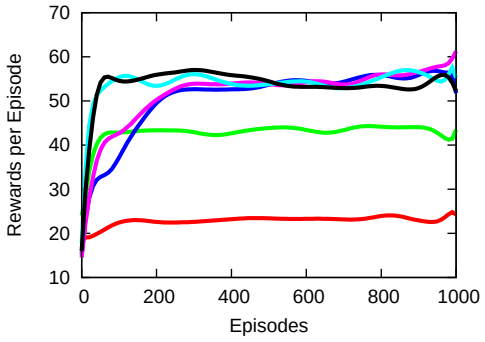
Figure B.20: Approximated value functions - Two individual teams of three agents each, one nearest reward in the state representation with degrading ε values.



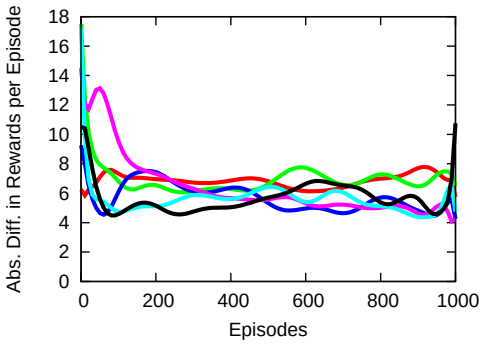
(a) Closest agent to any reward.



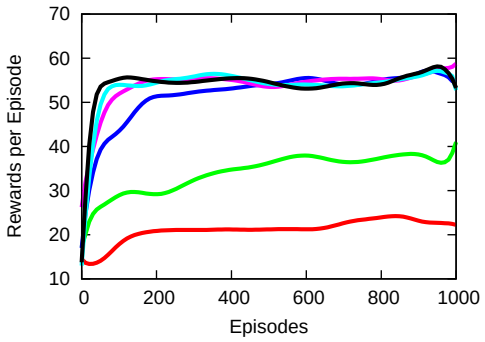
(b) Closest agent to any reward.



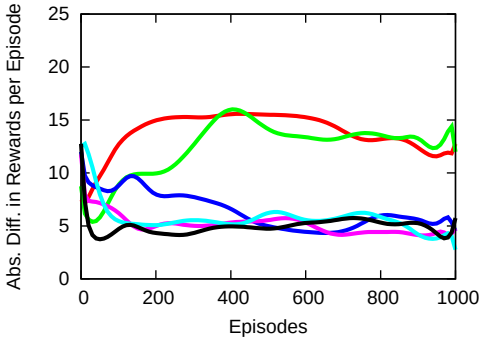
(c) Closest other agent to agent's nearest two rewards.



(d) Closest other agent to agent's nearest two rewards.

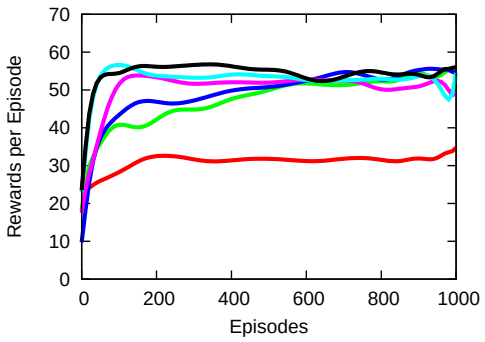


(e) Closest other agent to agent's nearest two rewards and closest agent to any reward.

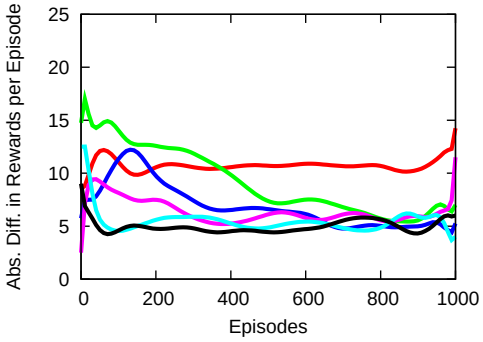


(f) Closest other agent to agent's nearest two rewards and closest agent to any reward.

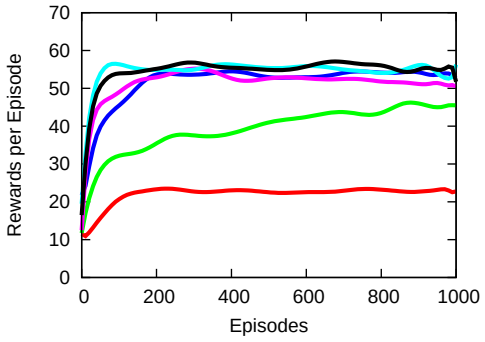
Figure B.21: Approximated value functions - Two individual teams of three agents each, two nearest rewards in the state representation with constant ε values.



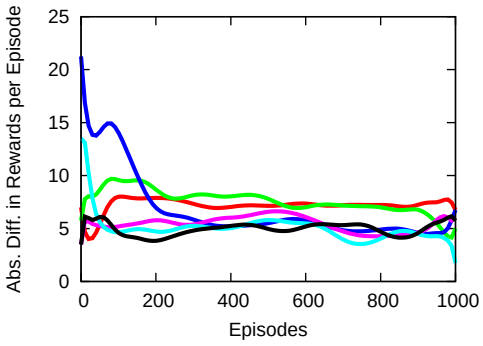
(a) Closest agent to any reward.



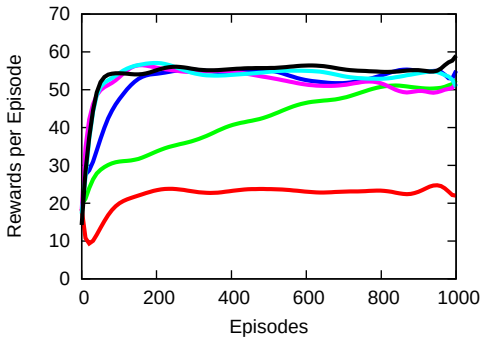
(b) Closest agent to any reward.



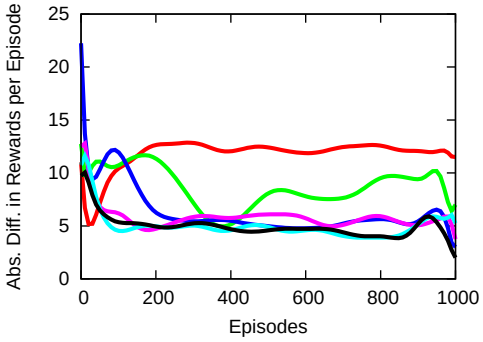
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

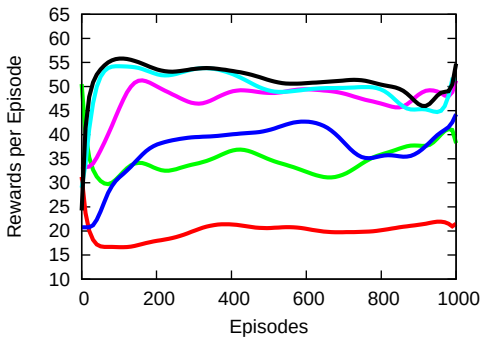


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

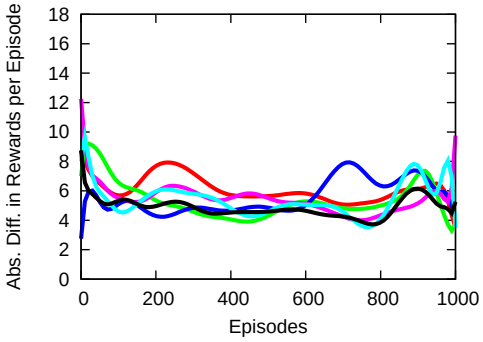


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

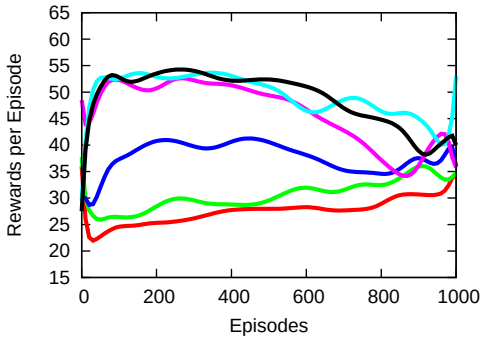
Figure B.22: Approximated value functions - Two individual teams of three agents each, two nearest rewards in the state representation with degrading ε values.



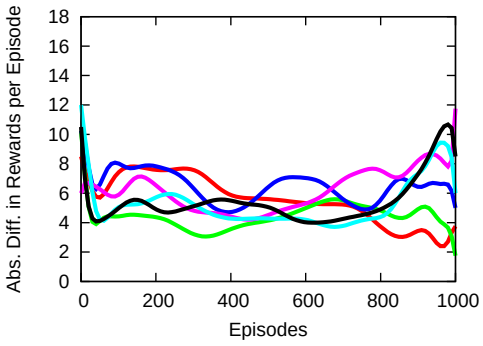
(a) Closest agent to any reward.



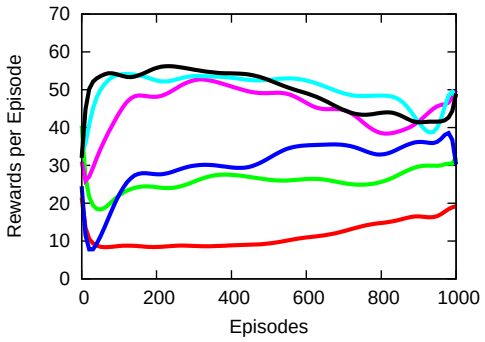
(b) Closest agent to any reward.



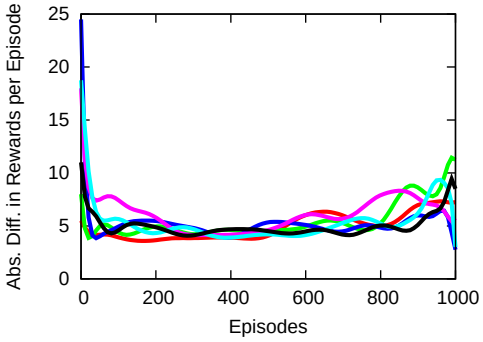
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

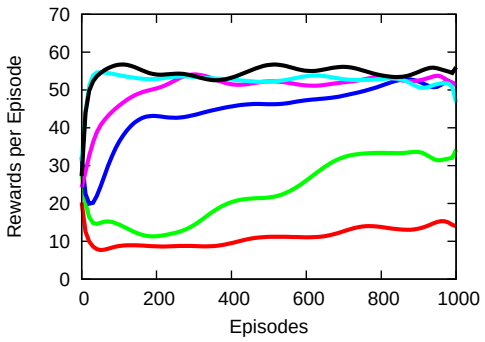


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

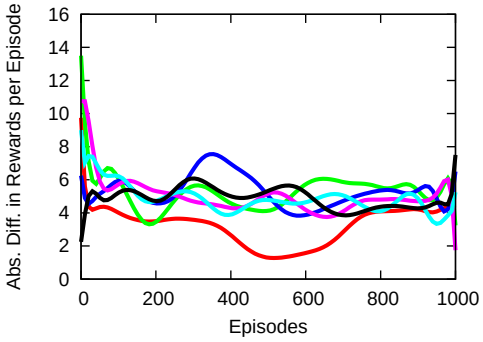


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

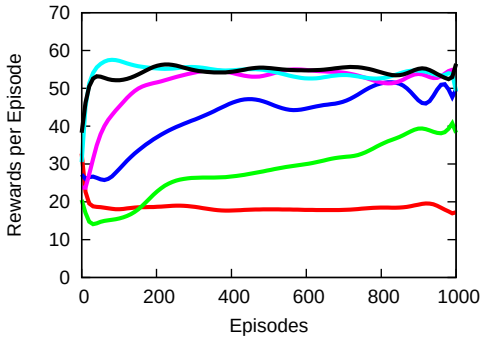
Figure B.23: Approximated value functions - Two balanced teams of three agents each, one nearest reward in the state representation with constant ε values.



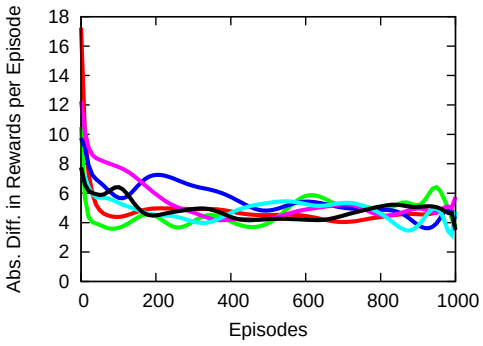
(a) Closest agent to any reward.



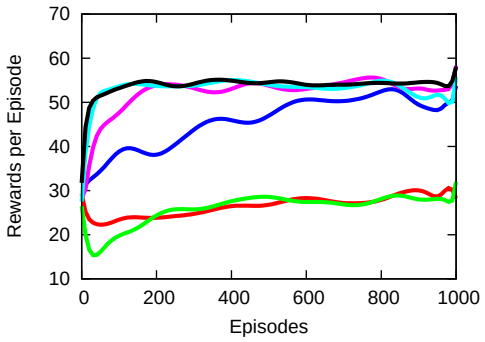
(b) Closest agent to any reward.



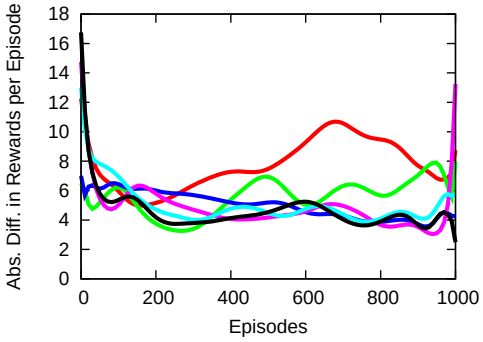
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

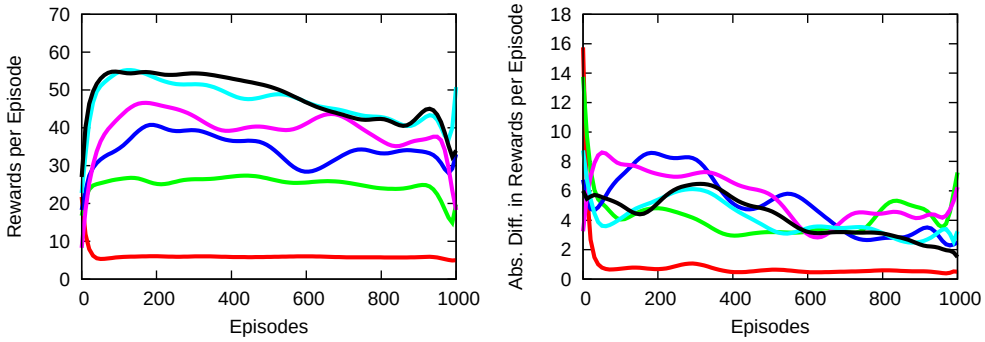


(e) Closest other agent to agent's nearest reward and closest agent to any reward.



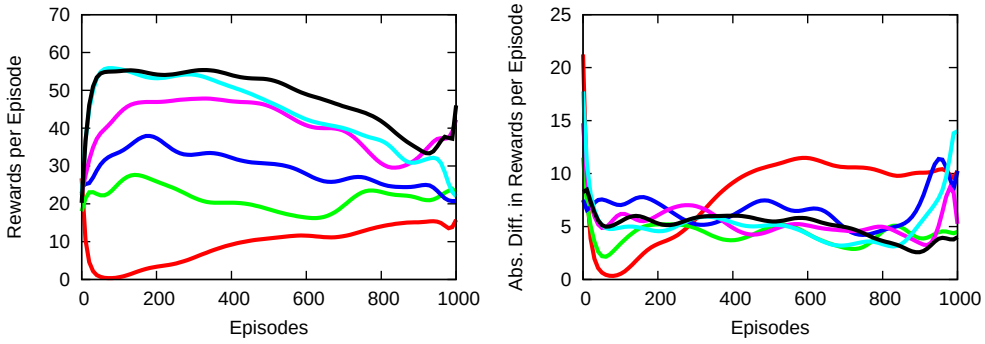
(f) Closest other agent to agent's nearest reward and closest agent to any reward.

Figure B.24: Approximated value functions - Two balanced teams of three agents each, one nearest reward in the state representation with degrading ε values.



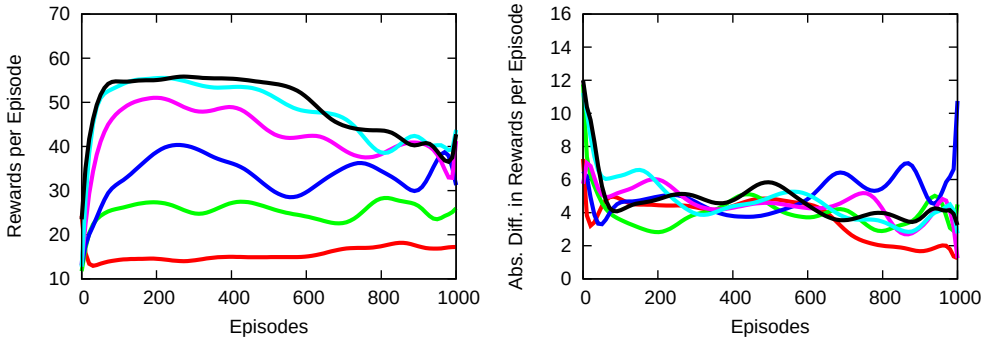
(a) Closest agent to any reward.

(b) Closest agent to any reward.



(c) Closest other agent to agent's two nearest rewards.

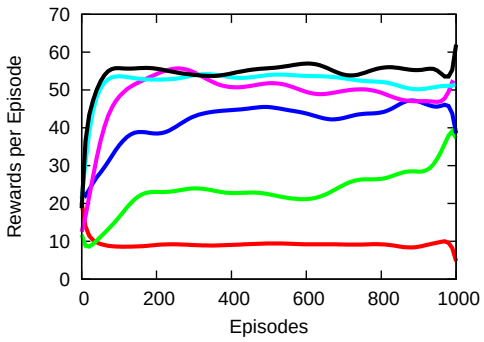
(d) Closest other agent to agent's two nearest rewards.



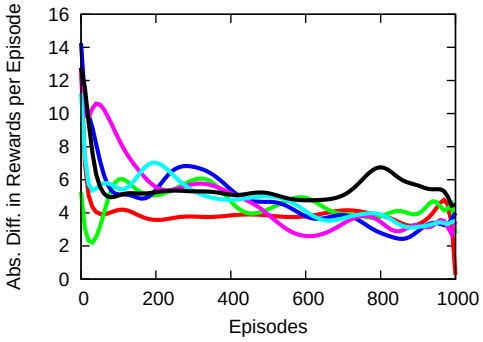
(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

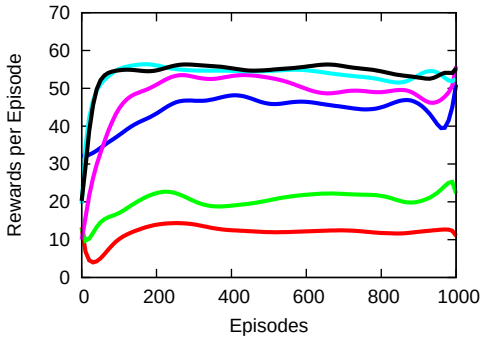
Figure B.25: Approximated value functions - Two balanced teams of three agents each, two nearest rewards in the state representation with constant ε values.



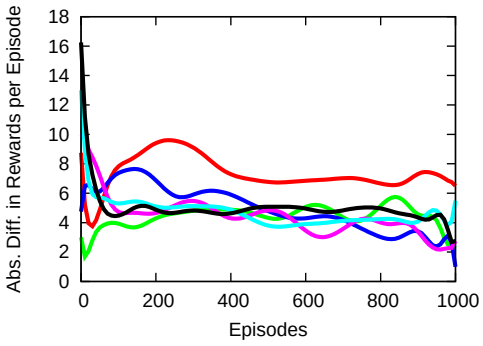
(a) Closest agent to any reward.



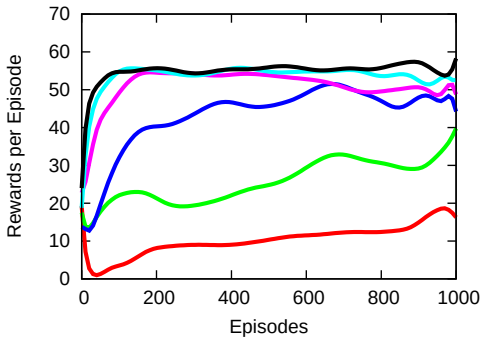
(b) Closest agent to any reward.



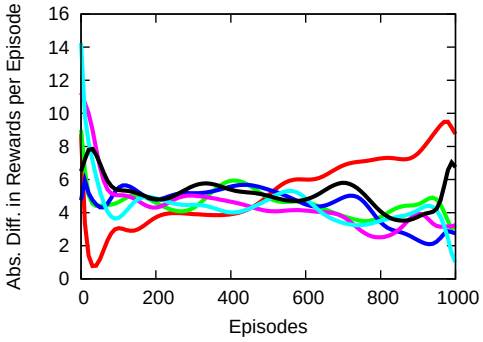
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

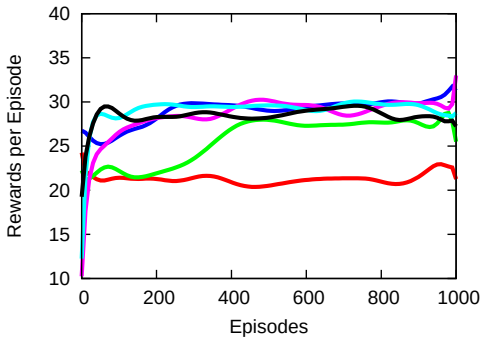


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

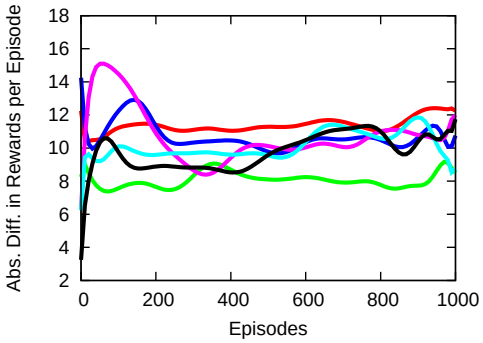


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

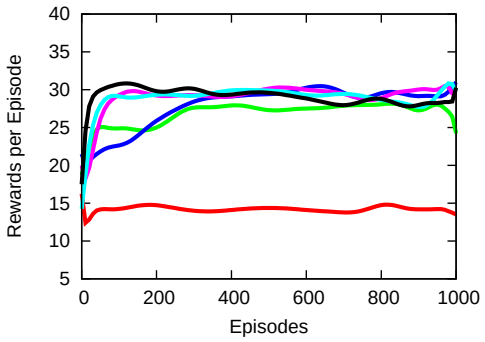
Figure B.26: Approximated value functions - Two balanced teams of three agents each, two nearest rewards in the state representation with degrading ε values.



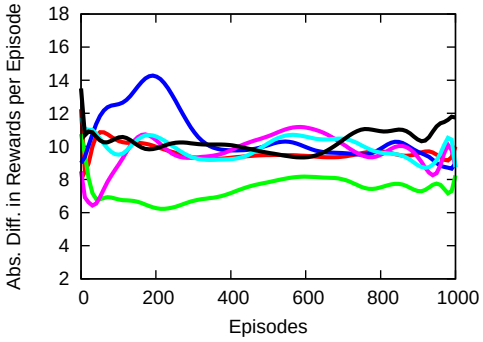
(a) Closest agent to any reward.



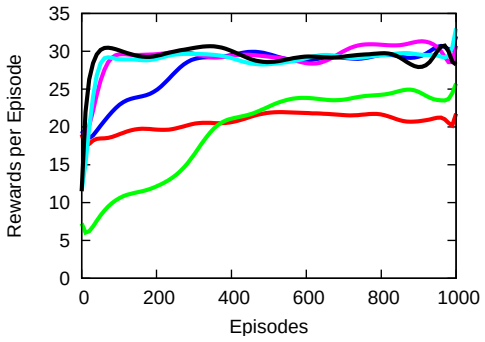
(b) Closest agent to any reward.



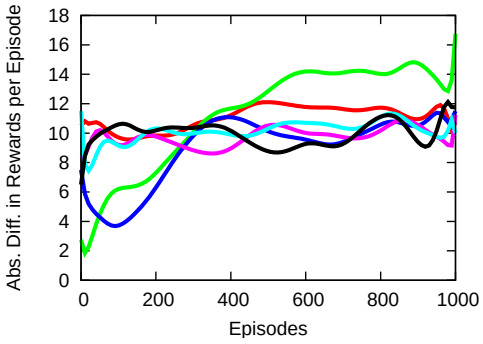
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

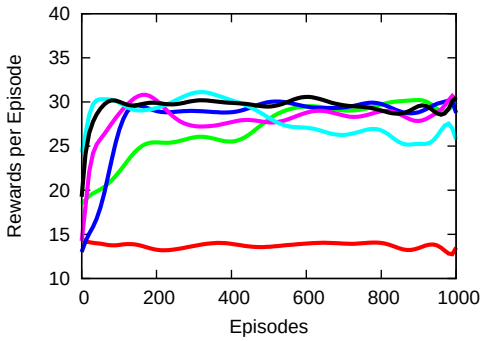


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

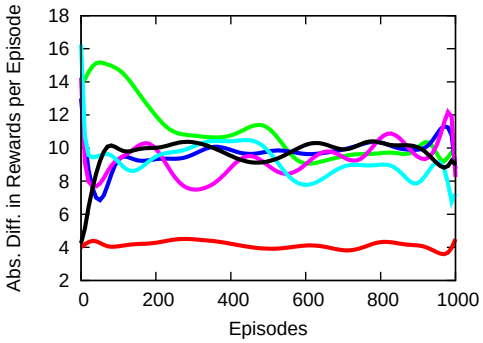


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

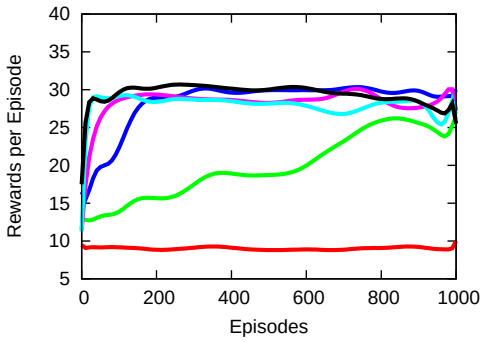
Figure B.27: Approximated value functions - A team of two agents and a single agent acting individually, one nearest reward in the state representation with constant ε values.



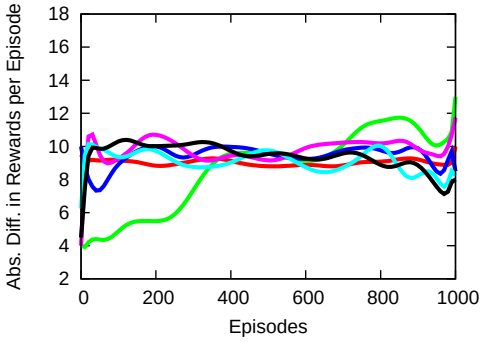
(a) Closest agent to any reward.



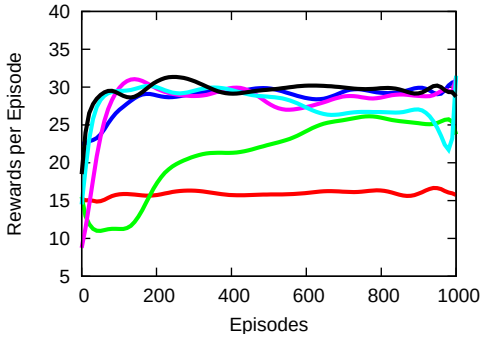
(b) Closest agent to any reward.



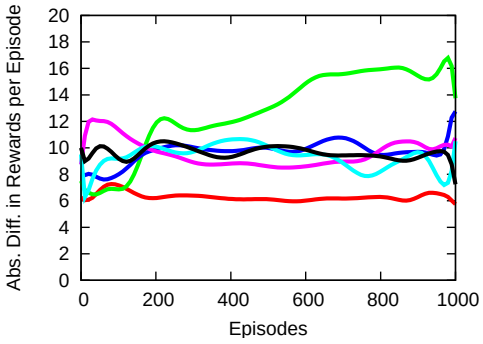
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

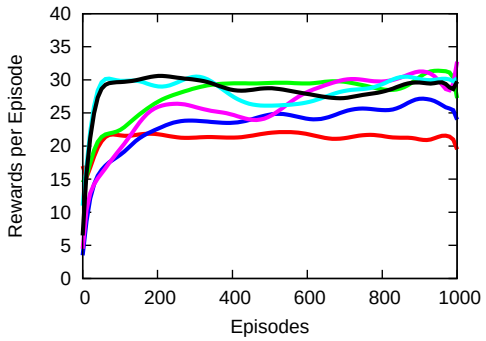


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

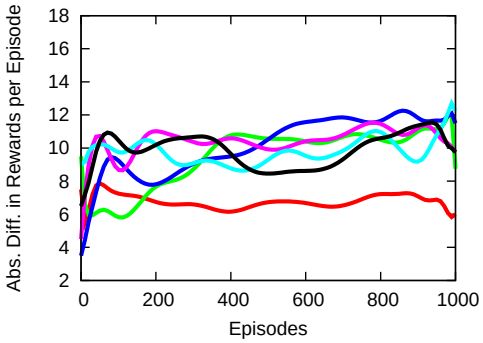


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

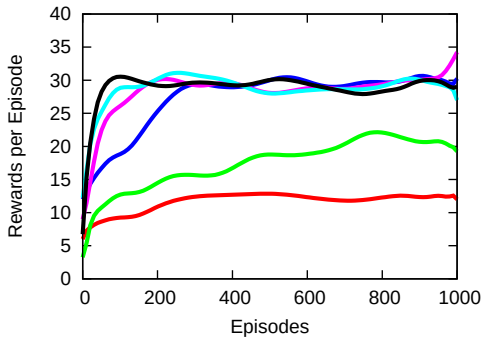
Figure B.28: Approximated value functions - A team of two agents and a single agent acting individually, one nearest reward in the state representation with degrading ε values.



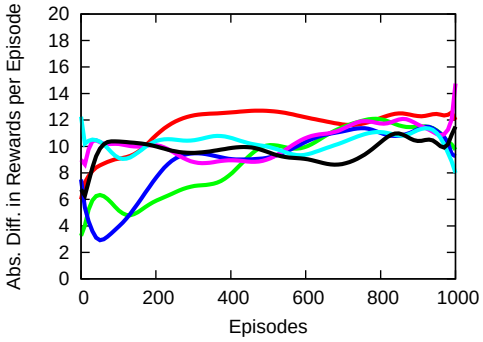
(a) Closest agent to any reward.



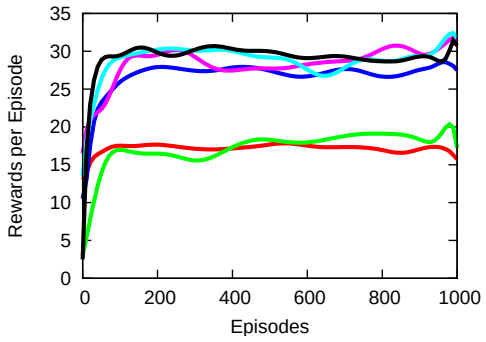
(b) Closest agent to any reward.



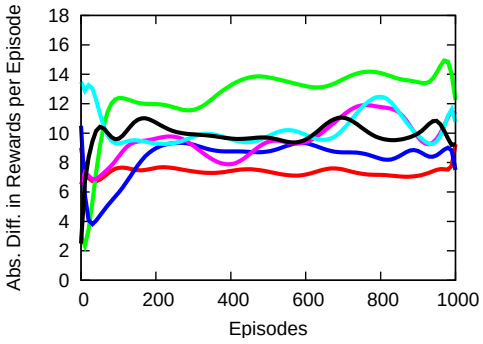
(c) Closest other agent to agent's nearest two rewards.



(d) Closest other agent to agent's nearest two rewards.

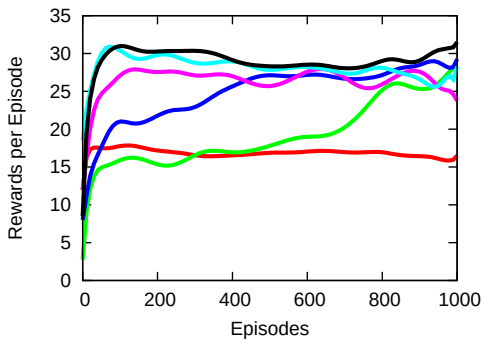


(e) Closest other agent to agent's nearest two rewards and closest agent to any reward.

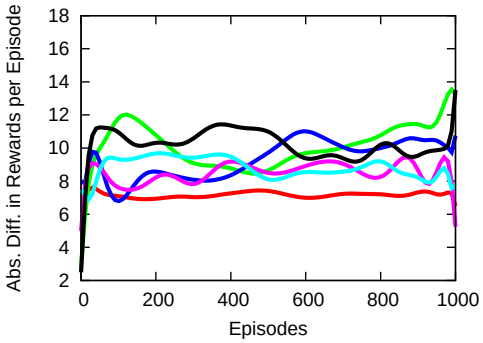


(f) Closest other agent to agent's nearest two rewards and closest agent to any reward.

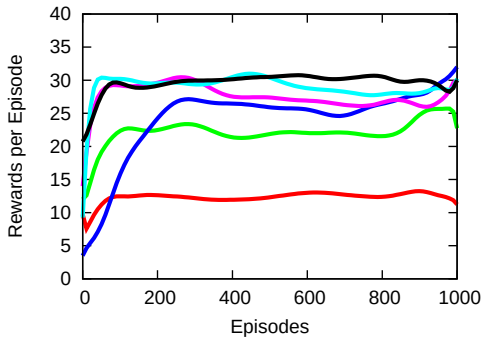
Figure B.29: Approximated value functions - A team of two agents and a single agent acting individually, two nearest rewards in the state representation with constant ε values.



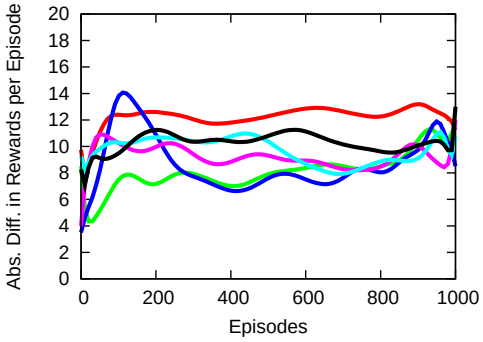
(a) Closest agent to any reward.



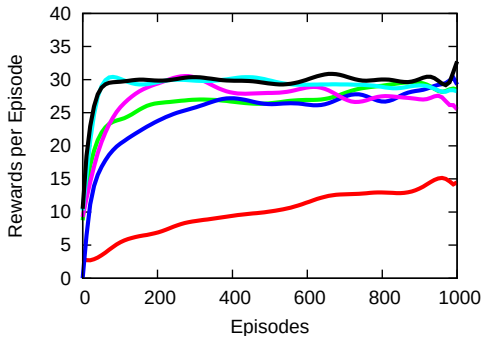
(b) Closest agent to any reward.



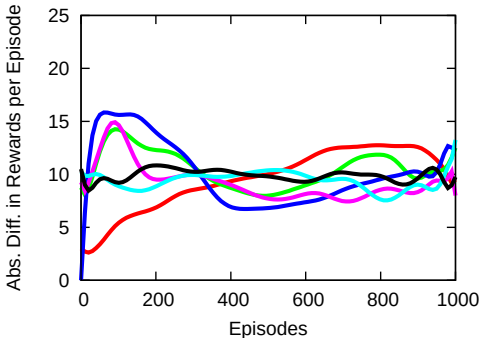
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

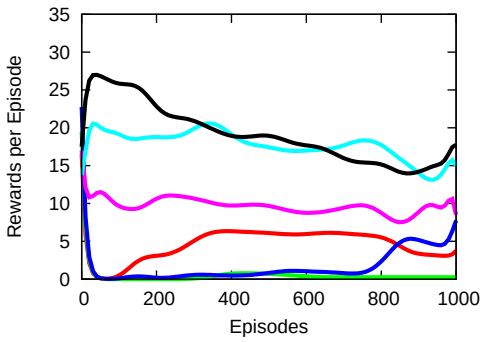


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

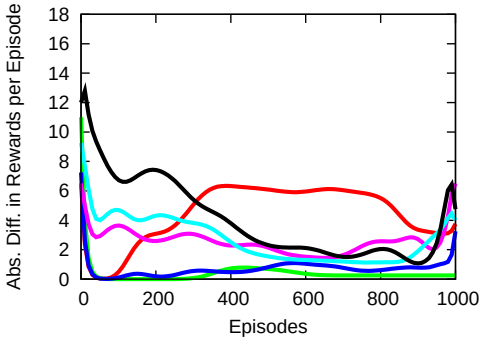


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

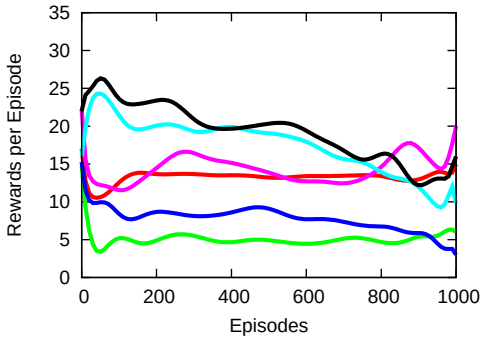
Figure B.30: Approximated value functions - A team of two agents and a single agent acting individually, two nearest rewards in the state representation with degrading ε values.



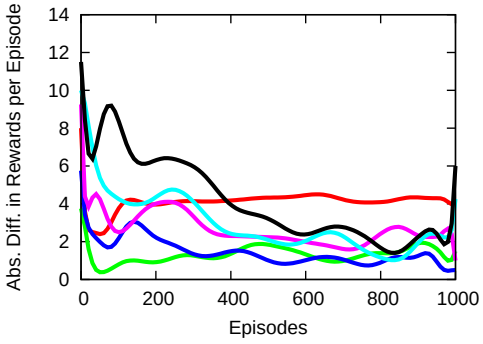
(a) Closest agent to any reward.



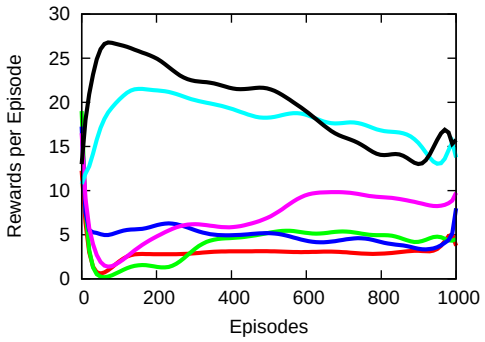
(b) Closest agent to any reward.



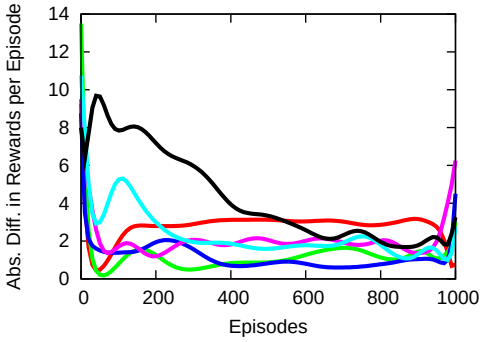
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

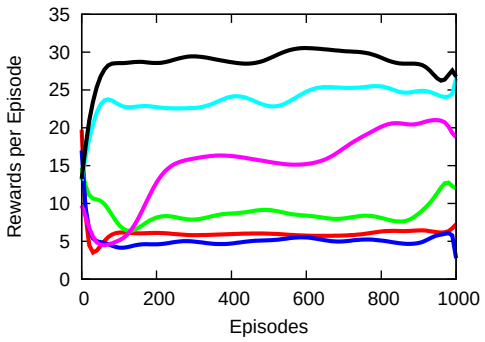


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

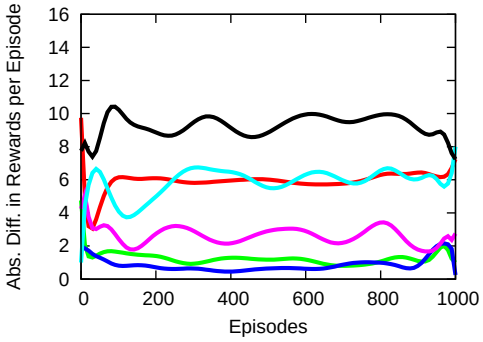


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

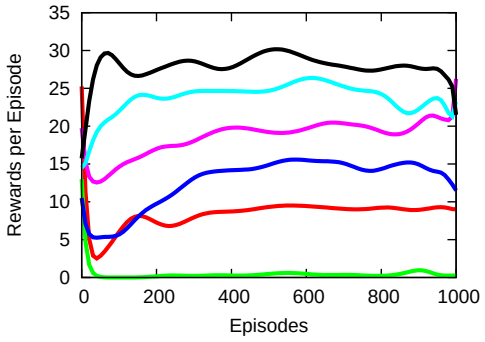
Figure B.31: Approximated value functions - A team of two agents and a single agent in balance, one nearest reward in the state representation with constant ε values.



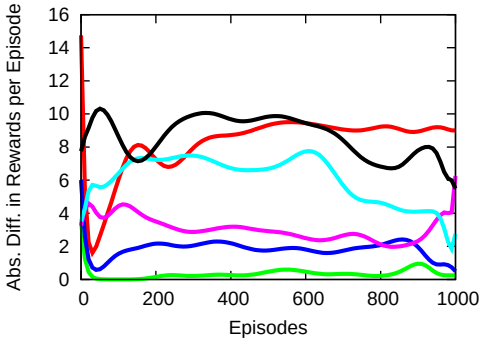
(a) Closest agent to any reward.



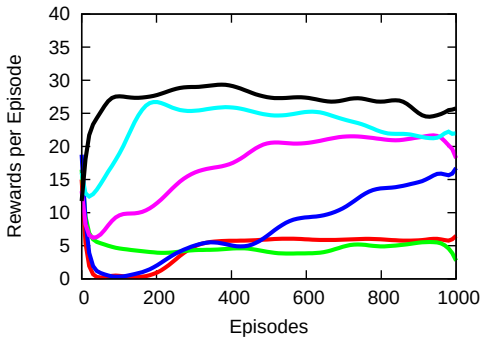
(b) Closest agent to any reward.



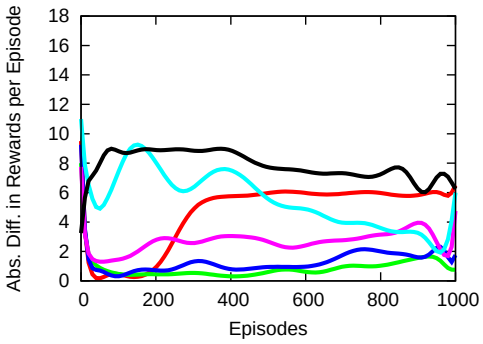
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

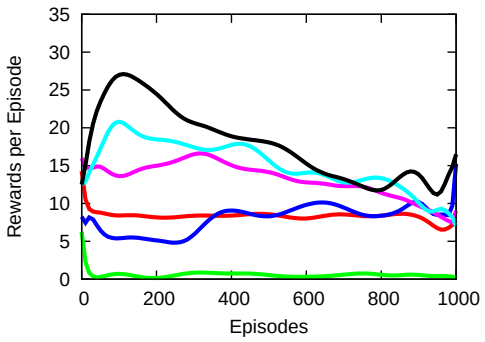


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

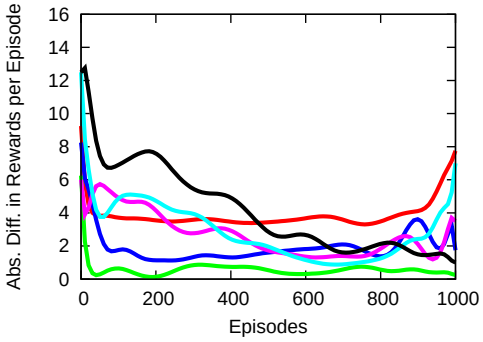


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

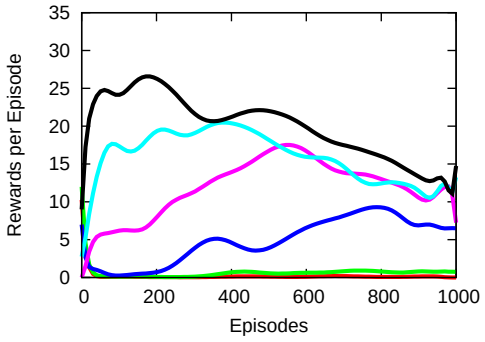
Figure B.32: Approximated value functions - A team of two agents and a single agent in balance, one nearest reward in the state representation with degrading ε values.



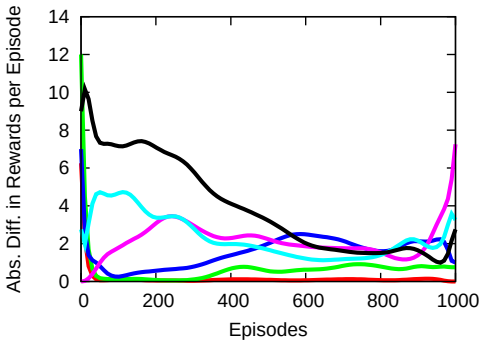
(a) Closest agent to any reward.



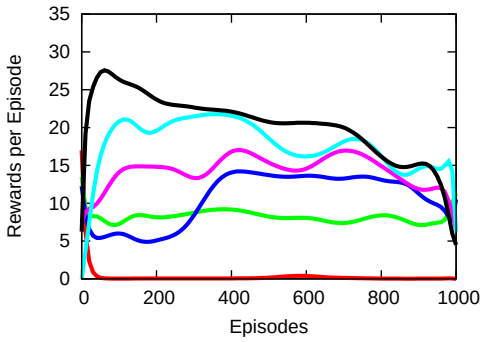
(b) Closest agent to any reward.



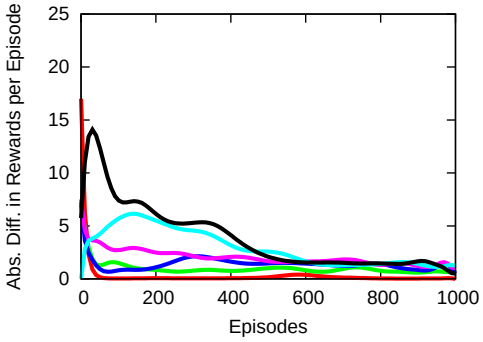
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

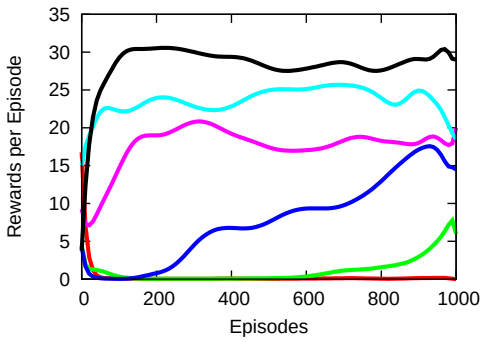


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

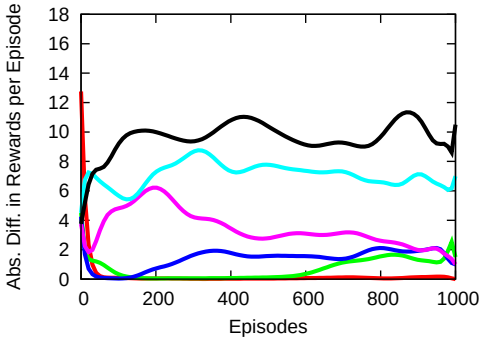


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

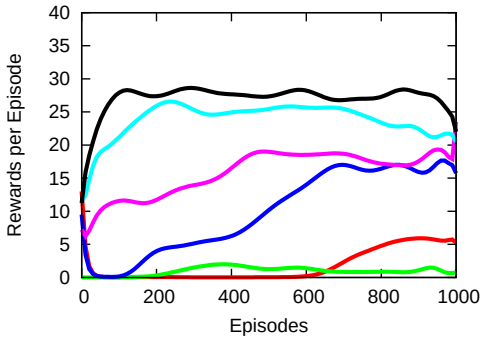
Figure B.33: Approximated value functions - A team of two agents and a single agent in balance, two nearest rewards in the state representation with constant ε values.



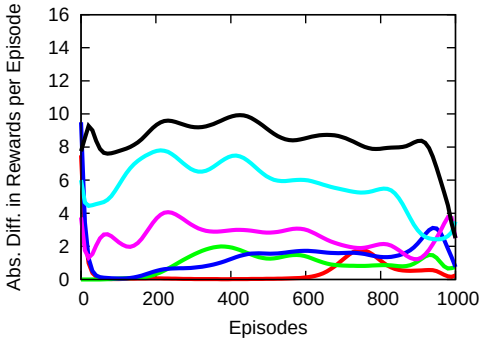
(a) Closest agent to any reward.



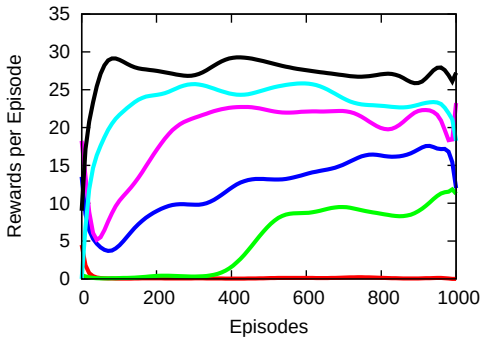
(b) Closest agent to any reward.



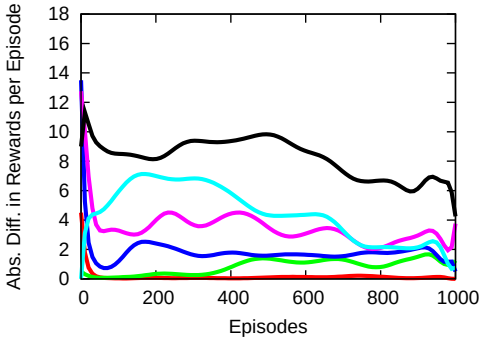
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

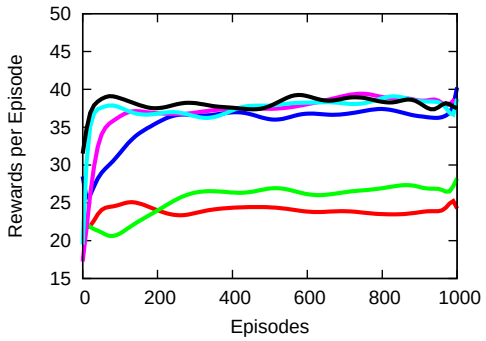


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

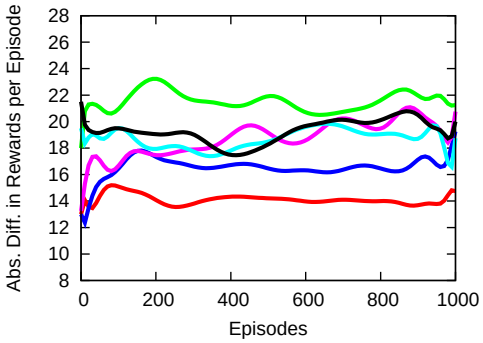


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

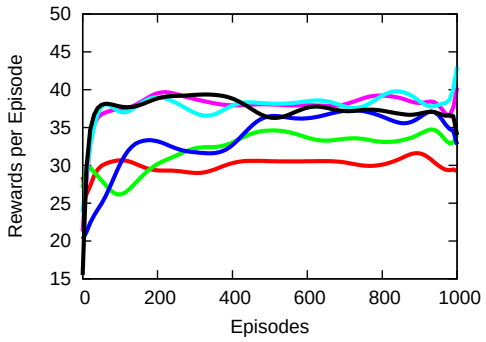
Figure B.34: Approximated value functions - A team of two agents and a single agent in balance, two nearest rewards in the state representation with degrading ε values.



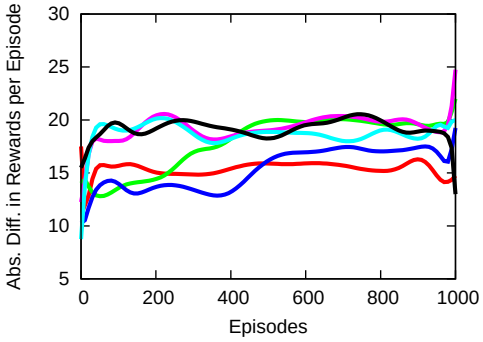
(a) Closest agent to any reward.



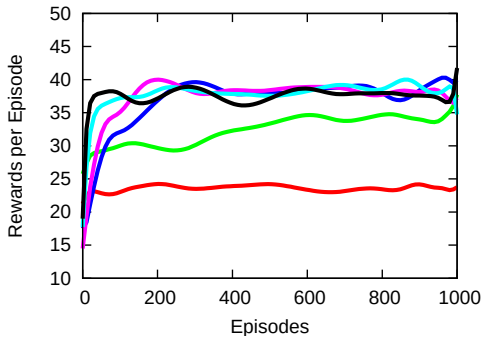
(b) Closest agent to any reward.



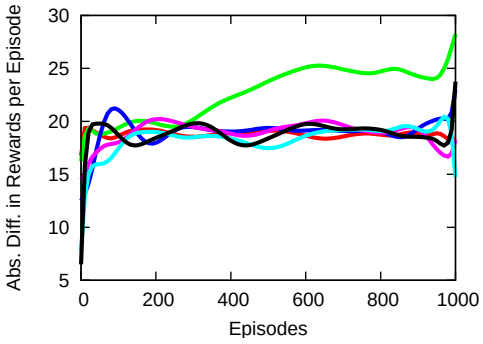
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

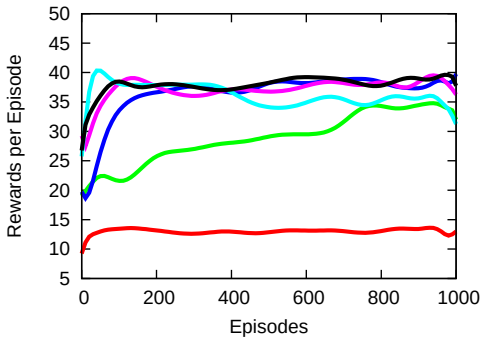


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

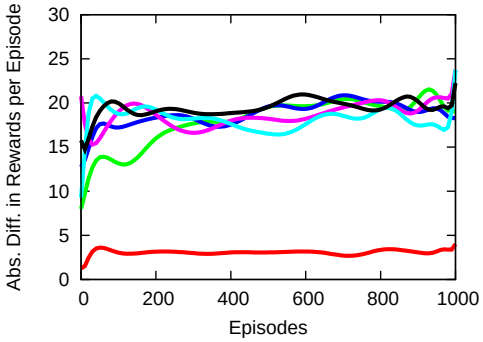


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

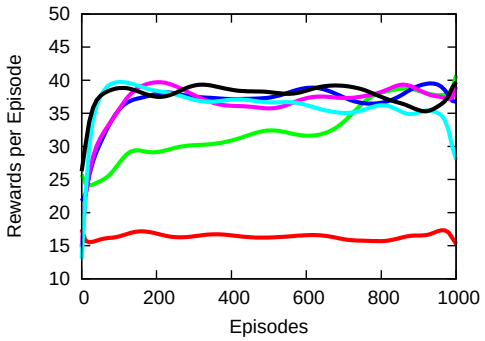
Figure B.35: Approximated value functions - A team of three agents and a single agent acting individually, one nearest reward in the state representation with constant ε values.



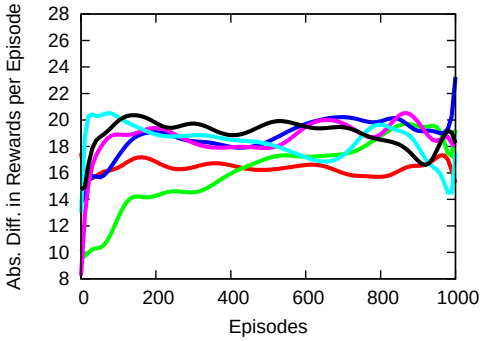
(a) Closest agent to any reward.



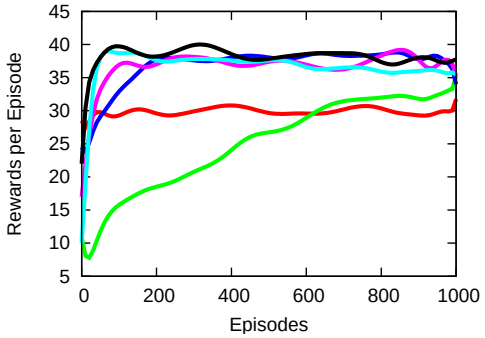
(b) Closest agent to any reward.



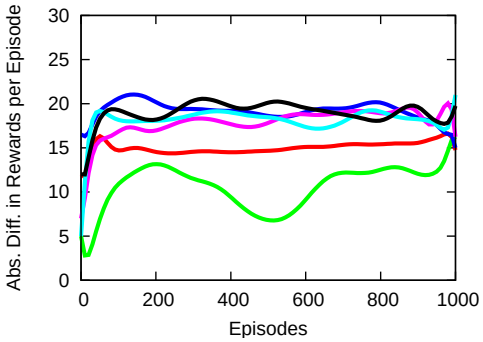
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

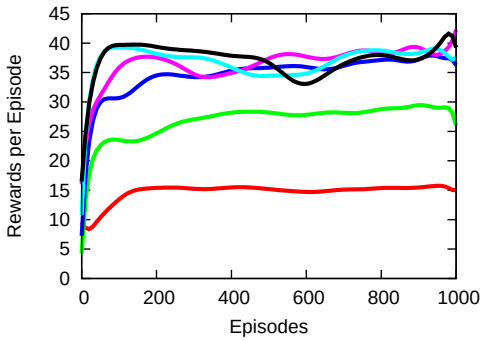


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

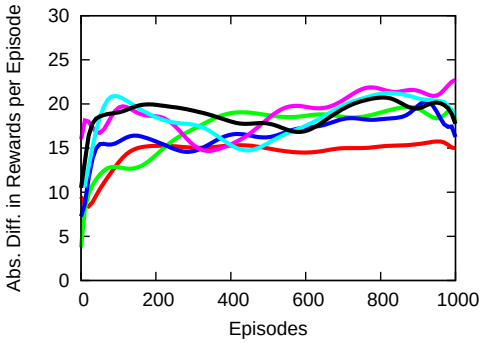


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

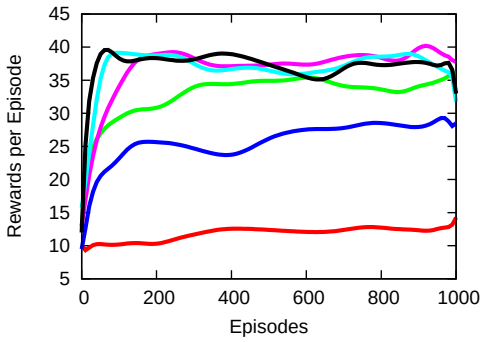
Figure B.36: Approximated value functions - A team of three agents and a single agent acting individually, one nearest reward in the state representation with degrading ε values.



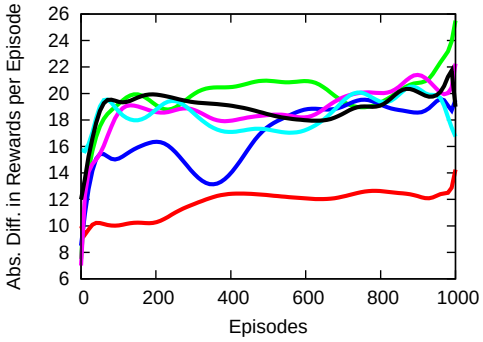
(a) Closest agent to any reward.



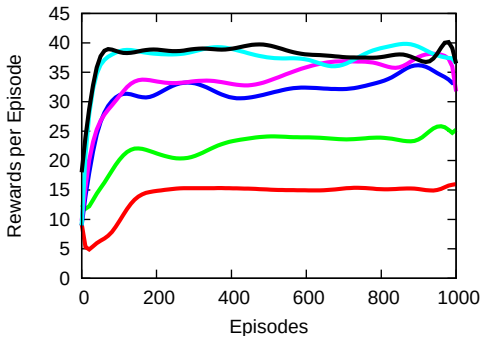
(b) Closest agent to any reward.



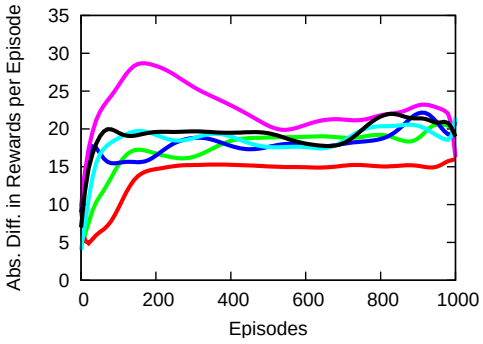
(c) Closest other agent to agent's nearest two rewards.



(d) Closest other agent to agent's nearest two rewards.

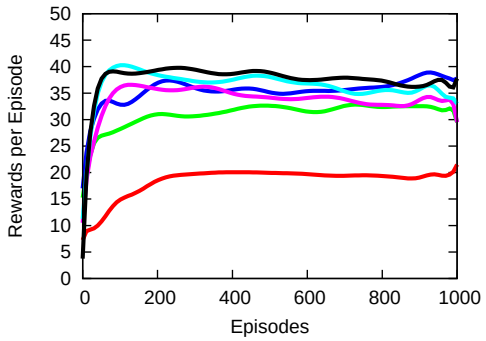


(e) Closest other agent to agent's nearest two rewards and closest agent to any reward.

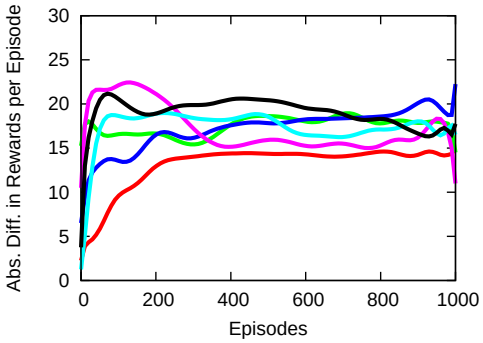


(f) Closest other agent to agent's nearest two rewards and closest agent to any reward.

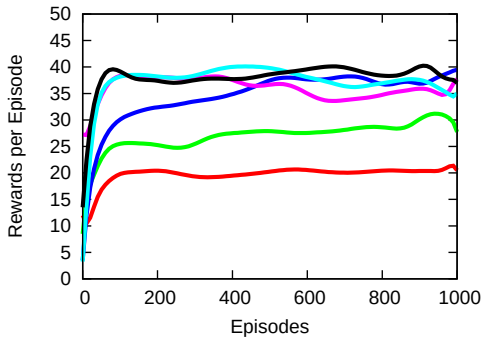
Figure B.37: Approximated value functions - A team of three agents and a single agent acting individually, two nearest rewards in the state representation with constant ε values.



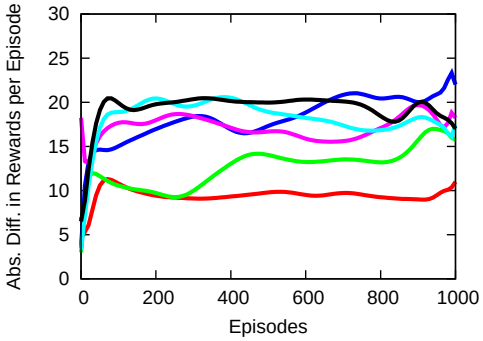
(a) Closest agent to any reward.



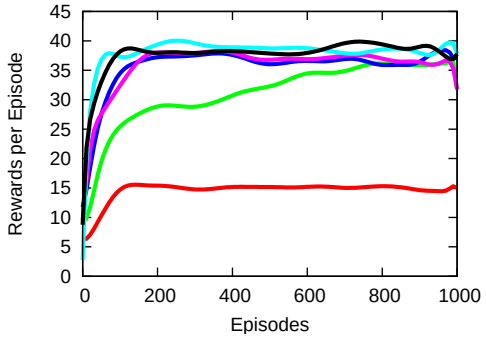
(b) Closest agent to any reward.



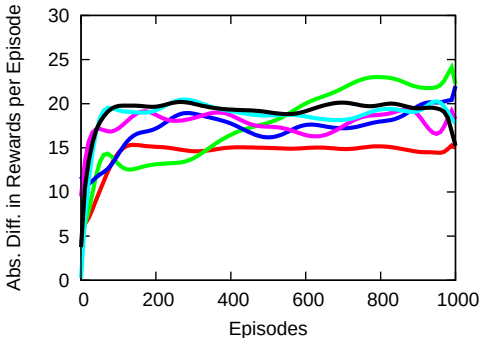
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

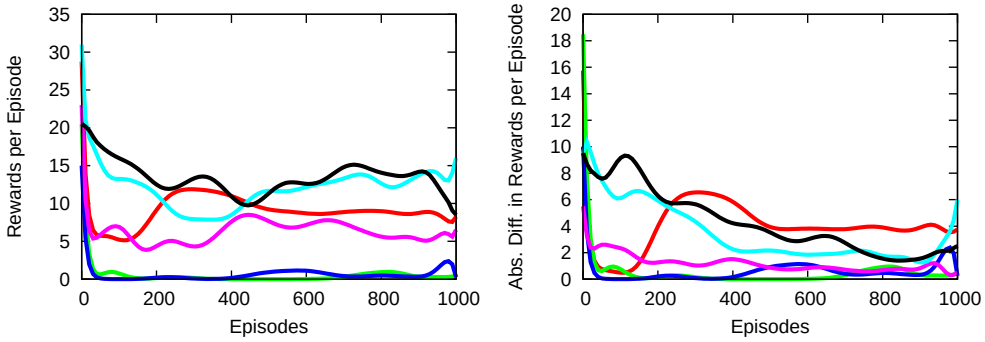


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.



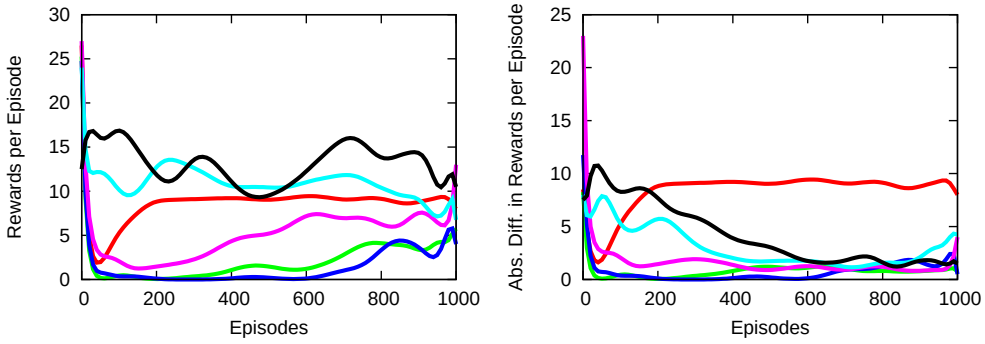
(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

Figure B.38: Approximated value functions - A team of three agents and a single agent acting individually, two nearest rewards in the state representation with degrading ε values.



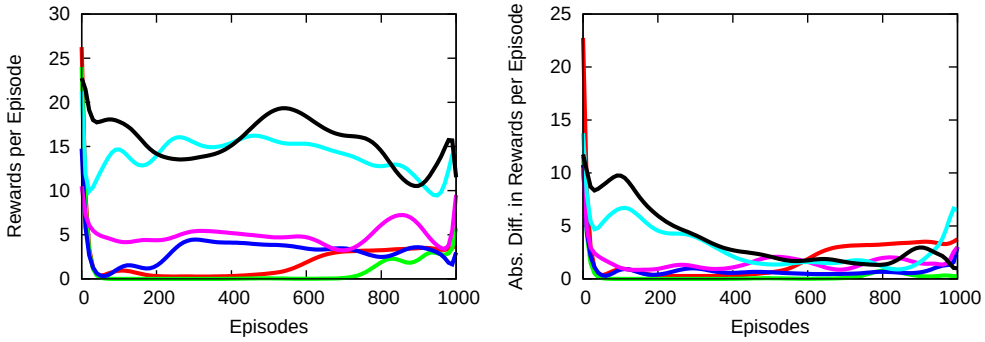
(a) Closest agent to any reward.

(b) Closest agent to any reward.



(c) Closest other agent to agent's nearest reward.

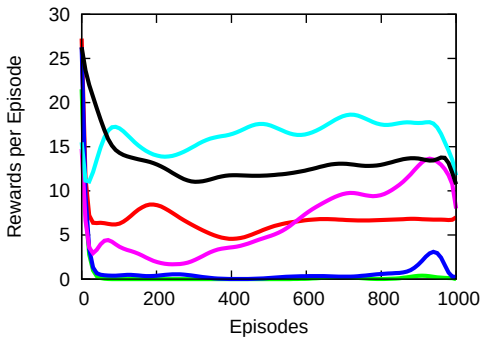
(d) Closest other agent to agent's nearest reward.



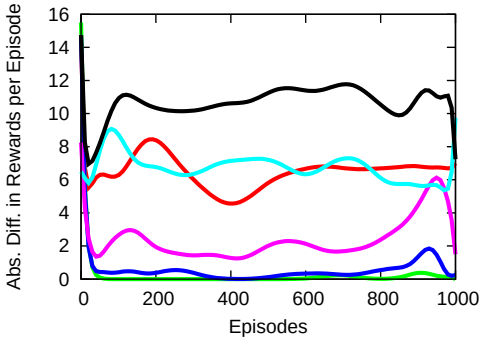
(e) Closest other agent to agent's nearest reward and closest agent to any reward.

(f) Closest other agent to agent's nearest reward and closest agent to any reward.

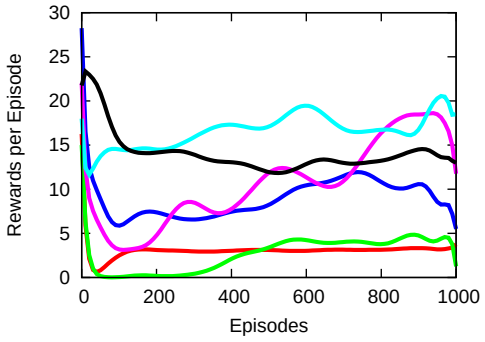
Figure B.39: Approximated value functions - A team of three agents and a single agent in balance, one nearest reward in the state representation with constant ε values.



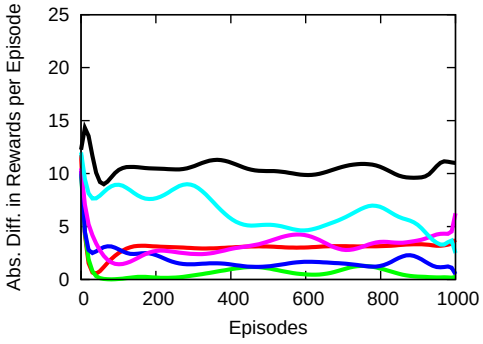
(a) Closest agent to any reward.



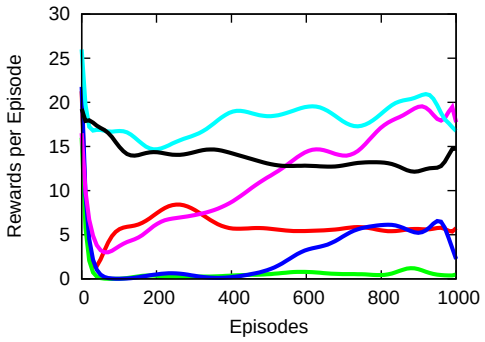
(b) Closest agent to any reward.



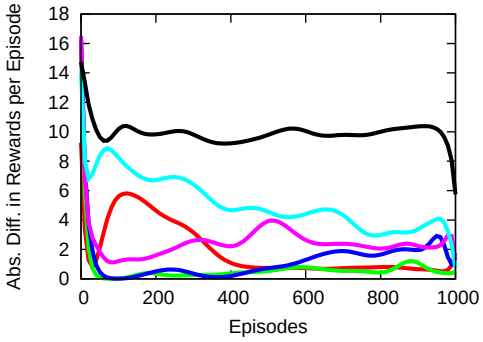
(c) Closest other agent to agent's nearest reward.



(d) Closest other agent to agent's nearest reward.

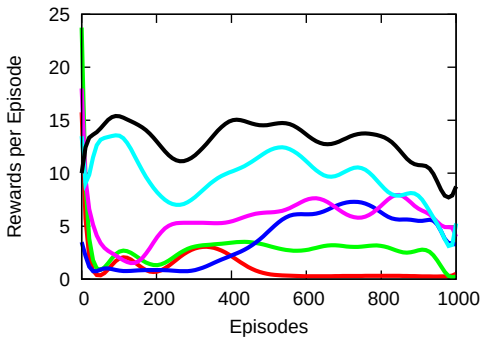


(e) Closest other agent to agent's nearest reward and closest agent to any reward.

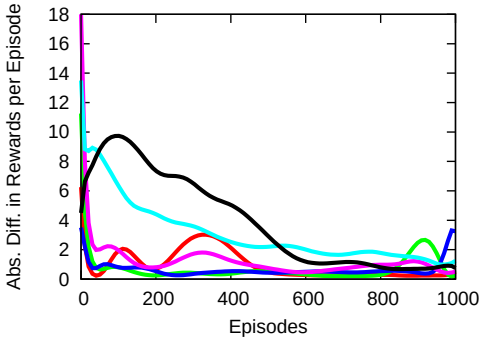


(f) Closest other agent to agent's nearest reward and closest agent to any reward.

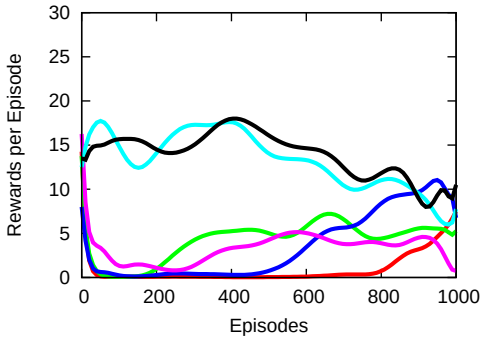
Figure B.40: Approximated value functions - A team of three agents and a single agent in balance, one nearest reward in the state representation with degrading ε values.



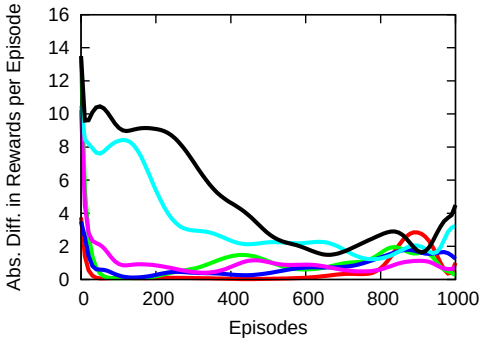
(a) Closest agent to any reward.



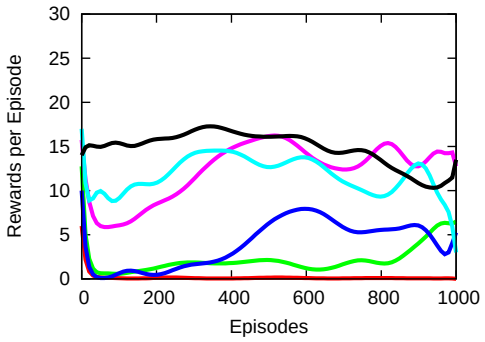
(b) Closest agent to any reward.



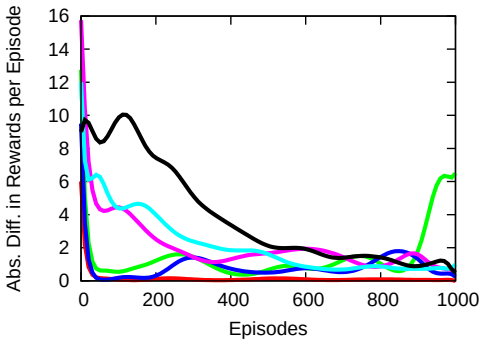
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.

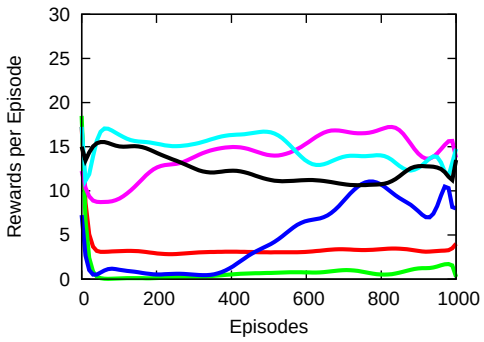


(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.

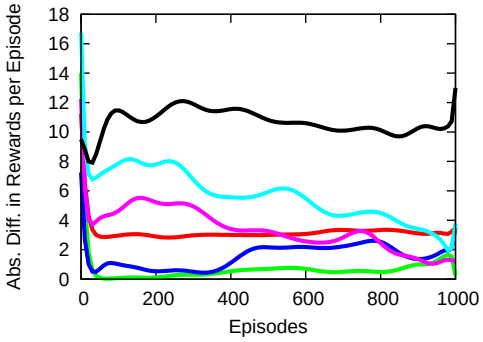


(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

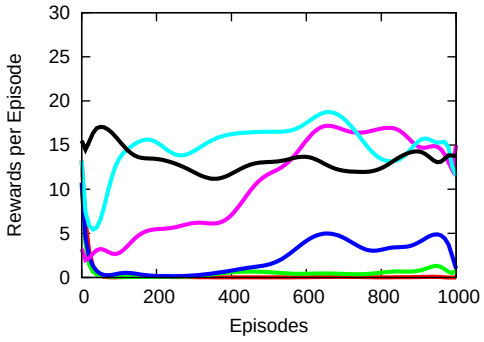
Figure B.41: Approximated value functions - A team of three agents and a single agent in balance, two nearest rewards in the state representation with constant ε values.



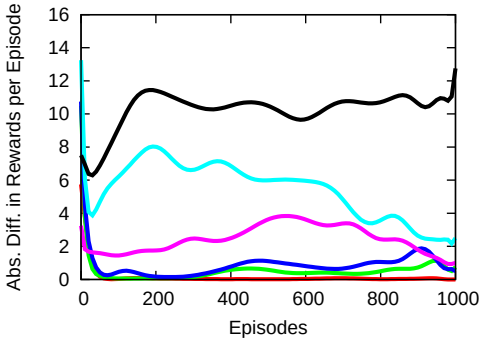
(a) Closest agent to any reward.



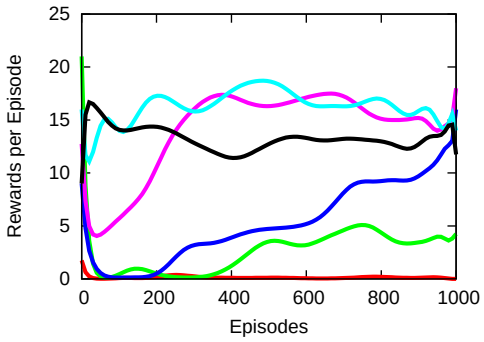
(b) Closest agent to any reward.



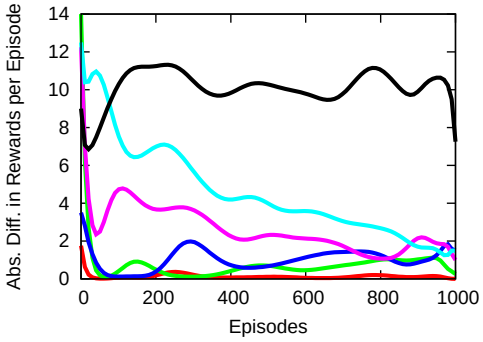
(c) Closest other agent to agent's two nearest rewards.



(d) Closest other agent to agent's two nearest rewards.



(e) Closest other agent to agent's two nearest rewards and closest agent to any reward.



(f) Closest other agent to agent's two nearest rewards and closest agent to any reward.

Figure B.42: Approximated value functions - A team of three agents and a single agent in balance, two nearest rewards in the state representation with degrading ε values.