TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Computer Science
Knowledge Engineering Group
Supervisor: Prof. Dr. Johannes Fürnkranz

Diploma Thesis

Creating Adaptive Game AI in a Real Time Continuous Environment using Neural Networks

Author: Andreas Pfeifer
March 2009

Abstract

This thesis gives some background on the use of Artificial Intelligence techniques in game development, notably reinforcement learning techniques implemented to allow games to learn to play and improve via selfplay, or playing against a human opponent and artificial neural networks used in the decision making process of ingame agents.

Some of these ideas and techniques, originally developed for symbolic games are then applied to a real time, continuous game akin to modern, commercial video games.

A modular artificial neural network architecture is used to create AI agents, capable of not only showing visibly intelligent behaviour, but also of adapting to changing game parameters in the game via online learning algorithms.

The concept of controlled learning is introduced to make the use of learning AI agents more attractive to game developers.

Motivation

Commercial computer games today are vastly complex programs. They are developed by teams of up to several hundreds of programmers, designers and artists of all kinds. They have near photorealistic graphics. They use advanced physics simulations. They cost millions of dollars. They produce revenues of billions of dollars. They create complex, virtual worlds.

Yet they generally fail at any kind of higher intelligence.

Most of today's games fail to provide an adequate challenge to human players unless they present the player with vastly unfair situations or use cheating measures. This has a definite negative effect on the immersion human players feel, despite all the impressive work done in many other areas. Smart and adaptive artificial intelligence would go along way in taking computer games to another level. Ironically scientists have been making huge advances in programming games that are even able to beat human expert players. But this work has been done on a different kind of computer game. The symbolic or human games, such as Chess, Checkers and Backgammon.

This thesis will try to understand the reasons behind this development and bridge the gap between these two types of games. We will use techniques used for human games, such as reinforcement learning and artificial neural networks, in an environment that resembles commercial games to show that these techniques can be used to implement smart AI that adapts to the player's actions online.

Declaration of authenticity

This thesis is my original work and has not been submitted for a degree at this or any other university in part or wholly, nor does it contain, to the best of my knowledge, any material published or written by another person, except where noted.

Darmstadt, Germany, May 1th 2009 Andreas Pfeifer

.

.

Outline of this thesis

Chapter 1 presents the motivation for using machine learning techniques in a commercial type computer game. It summarizes the state of artificial intelligence in videogames and gives reasons why both groups of game designers, the developers of commercial games and the scientists working on symbolic games, can profit from better cooperation.

Chapter 2 gives a short summary of the machine learning techniques used during the course of this thesis. The emphasis lies on artificial neural networks capable of controlling game agents and evolving to better play a game.

Chapter 3 defines a small task that will be used to show the viability in using neural nets in a real time, continuous environment typical to modern games. The goal of the task is to produce agents that exhibit seemingly intelligent and believable behaviour, controlled by an artificial neural network.

Chapter 4 summarizes the first stage of the implementation of the task defined in chapter 3. A small game engine will be created to serve as a testing ground for the AI techniques explored in the following chapters.

Chapter 5 A first version of the Agent AI is developed using an artificial neural network. Several techniques such as supervised and controlled learning are introduced to add adaptive qualities to the AI.

Chapter 6 A detailed analysis of the learning algorithms used to create the AI agents is given and several improvements are implemented and tested.

Chapter 7 Introduction of reinforcement learning techniques as an alternative to the supervised learning techniques used in chapters 5 and 6.

Chapter 8 Variations on the structure of the neural network used are examined. Implementation and testing of some of the learning techniques from the former chapters on these changed structures.

Chapter 9 A concluding discussion of the thesis.

Table of Contents

1 Background

1.1 Computer games and Artificial Intelligence.

The scientific field called “Artificial Intelligence”, short AI, has its origins in the 1940s and 50s. The original dream was to build an artificial brain capable of human logic and deduction. In 1965, American scientist H. A. Simon stated: “Machines will be capable, within twenty years, of doing any work a man can do.”

This optimistic outlook proved to be wrong and the goals of AI research have since been redefined.

Today the field of AI in scientific studies is divided in many smaller and specialized fields such as, knowledge based systems, computational intelligence or cognitive systems. The results are applied to varied fields of applications like, data mining, industrial robotics, speech recognition and medical diagnosis.

One very unique type of application that was used very early by researchers to test and improve their work were games.

Even back in 1950 the first checkers and chess programs were written, which let the computer play against a human player. Eventually AI players were able to beat the human world champions in both these games by the mid 1990s. This interest in games though was concentrated on so called “human” or “symbolic” games.

The science community has been focused on analytical games, often also called symbolic games because they can be described using symbolic representations, such as board and card games. Normally the AI takes on the role of a human player in these games, thereby adhering to the original goal of creating an artificial brain. The so-called “good old-fashioned artificial intelligence” (GOFAI) techniques **[Haugeland85]** work well with them, and to a large extent, such techniques were developed for these games. Computer programs have beaten the world champions in chess ,Deep Blue in 1997 **[Campbell02]** and checkers **[Schaeffer et al. 96].**

This is the result of a long time of learning in games going back to 1959 (Samuel's checkers program). Some more games explored in this field are: Tic-Tac-Toe **[Michie61]**, **[Sutton88]**, Backgammon **[Pollack98]**; **[Tesauro94]**, Go **[Richards et al. 1997]**; **[StanleyMiikkulainen04]** and Othello **[MoriartyMiikkulainen95]**; **[Yoshioka et al. 98]**.

Parallel to these board game adaptations another kind of game emerged in the 1960s and 70s, video games sometimes also refered to as pc games or commercial games. Pong probably being the most famous one of the pioneers, which was released 1971 by Atari and was one of the first commercially released games. The first computer controlled opponents were used in games such as Space Invaders and Pac Man. These early games used simple deterministic logic and pattern movement to model opponent behaviour, therefore they were of no real interest to the scientific AI community. Only recently have there been more attempts to use AI techniques in commercial video games.

Many of the learning methods developed for analytical games can be applied to commercial video games as well, as has been shown by several scientitific publications: **[Fogel et al. 04]**, **[LairdLent00]**, **[Spronck05]**, **[Geisler02]**, **[Lucas05], [BryantMiikkulainen03]**; **[RevelloMcCartney02]** and **[Yannakakis et al. 04]**. What these works have in common is the fact that scientists applied some of their work to

commercial games by modding them (modifying some of the original game files) or creating small games of their own, essentially copies, based on older commercial games.

The video game industry itself has not made any of the machine learning techniques their own and incorporated them into their games, essentially using techinques that were developed 30 years ago **[Bauckhage et al. 03]**. The only notable exceptions are the ubiquitous A* algorithm, which is the basis for most pathfinding solutions in commercial games, and a handful of games using AI that adapts to the player's actions: Creatures (Mindscape 1996) , Black & White (Lionhead Studios 2001) and Battlecruiser 3000AD (Take Two Interactive 1997) are probably the most famous ones.

This is especially surprising since the game industry has been a driving force in the development of high end software and hardware, especially on the field of computer graphics.

1.2 Why now is there such little use of modern AI techniques in commercial games?

To answer this question we have to look at the different expectations and tasks a game AI has to fulfil and to take a closer look at games in general.

The foremost goal of game AI is not to beat the human player or play as best as possible, its primary goal is to provide a fun experience to the human player.This is the one cardinal difference between Artificial Intelligence in commercial and symbolic games. In many cases this means the AI player has to have deliberate weaknesses. It would be easy to program an AI agent capable of beating a human player with a 100% chance in many games. In shooter games for example the AI agent has to be programmed to not have perfect aim or reaction time, to provide a fun challenge for the player. This is probably one of the main reasons why scientists hesitate to work on commercial games and why game developers steer away from algorithms that let the AI player evolve and improve, potentially breaking the game for the human player. The goal of gaming AI is to imitate intelligent and above all humanlike behaviour in order to create a fun experience for the player.

The goal of AI researchers on the contrary has always been to develop algorithms that solve a task in the most efficient way possible. Therefore not all techniques developed in research are directly applicable to games. Many of them have been developed for very well defined often quite simple domains, whereas modern games simulate unpredictable virtual worlds.

Another reason that AI has been neglected is the fact that computer games have always pushed the current hardware to its limits, leaving few power for the AI. This process is getting slower though and especially the advent of multi core processors allows for more system power dedicated to AI routines.

1.3 Why should scientists take interest in commercial games?

Video games have always needed some kind of artificial intelligence, to be able to provide a challenging and exciting experience **[Chan et al. 04].** Modern commercial computer games simulate complex worlds that often have qualities close to the real one. Games are played by millions of people worldwide and the market is constantly growing. Because there are so many players and because video games carry perhaps the least risk to human life of any real-world application, they make an excellent testbed for techniques in artificial intelligence and machine learning **[LairdLent00].**

The reason for this is the difference between symbolic games and video games. Video games often simulate a physical environment where NPCs interact with the player and each other. They use data obtained through sensors which return numerical rather than symbolic values. Many kinds of noisy input from different sources has to be observed and the behaviour has to be changed under real time conditions. This provides a new challenge as the old techniques, developed for symbolic games, are often not well suited to this kind of conditions.

Many human-level control tasks, such as navigation, combat, team and individual tactics and strategy have to be solved in games **[Miikkulainen et al. 06]**.

These tasks are well suited for soft computational intelligence techniques such as neural networks, evolutionary computing, and fuzzy logic. They are able to deal with the fast, noisy, numerical, statistical, and changing domains provided by modern video games.

Therefore, video games constitute an opportunity similar to that of the symbolic games for GOFAI in 1980s and 1990s: an opportunity to develop and test techniques, and an opportunity to transfer the technology to industry **[Miikkulainen et al. 06]**.

1.4 Why game developers should use modern machine learning techniques.

The game industry has largely ignored scientific advances in the last 30 years.The AI used to control the behavior of the non-playercharacters (NPCs) in modern games often uses labor-intensive finite state machines, scripting and authoring methods **[Miikkulainen et al. 06]**. Even though it is possible to create interesting behaviour like that, the results are often repetitive and inflexible. Agents just cycle through a fixed repertoire of possible actions and reactions. Beating a game when playing against AI controlled agents often only consists of finding out how the AI works and then finding a way to beat this. Once such a solution is found the game instantly becomes boring **[Stanley et al. 05].** If a human player acts in unforeseen ways the AI will not be able to find an appropriate response. Once found such a weakness can be exploited as often as the player wants to because the AI is not able to correct its response.

Modern machine learning techniques have the potential to produce agents with the capabilities to adapt and learn, thus keeping video games interesting for a longer time.This of course also bears the risk of the agent learning unwanted behaviour, making the gaming experience unsatisfying. To minimize this risk precautions have to be taken. For example extensive offline learning or some kind of control algorithm has to be used.

Machine learning techniques could also be used to create entirely new genres, e.g. teaching a group of agents to play a game **[Miikkulainen et al. 06]**.

Most modern machine learning techniques such as neural networks are also more flexible and could be reused easier than complex finite state machines or scripts that have to built from scratch for a different game. Machine learning can make video games more interesting and decrease their production costs **[Fogel et al. 04]**.

Another use of machine learning besides implementing the actual AI could also be to test games. For example by learning human player action sequences that lead to either unwanted game behavior or to game states that allow to test certain ideas and expectations of the designer. In the long term, the game industry increasingly will have to rely on tools that help in finding such game states, preferrably on tools that do so automatically **[Chan et al. 04]**.

2 Machine Learning

The process or technique by which a device modifies its own behavior as the result of its past experience and performance.

Machine learning is a branch of artificial intelligence research. It covers a range of techniques used to allow computer programs to learn and thereby enhance their problem solving capabilities becoming more efficient at what they were programmed to do.

The field of application for machine learning procedures is quite varied. Many real world problems have been successfully approached with machine learning solutions. Examples are medical diagnostics, meteorologic prognosis, natural language processing, handwriting recognition, robot locomotion, game playing and many more.

Machine learning algorithms can be classified into distinct categories, depending on the available input and desired output. It should be mentioned that algorithms in the same category may reach their goal in quite different ways.

In the following chapter a description of machine learning terms and techniques which are of relevance for this thesis is given.

2.1 Offline learning

Offline learning refers to learning that is done without a human player playing the game. It can take many different forms. Often sample data is used to train ingame agents. The learning method to initialize the neural networks used in chapters 4, 5 and 6 falls under this category. It can also be implemented by letting two AI players play against each other.

2.2 Online learning

Online learning takes place while a game is being played. It can be supervised or unsupervised. The main reason to use online learning is to adapt to the way the player plays the game and keep it challenging. Game developers have been loath to use any kind of online learning as it adds a high level of unpredictability to published games. They fear that the AI could get worse and have a negative impact on gameplay.

2.3 Supervised learning

Supervised learning is a term that is used in the literature to describe two slightly different things:

i) Training that takes place while a human player is playing the game. Some kind of feedback is given directly by the player, which is used to make adjustments to the game AI during playtime. One popular commercial game which used this as an integral part of its design was Black & White (Lionhead 1996). In this game the player could indirectly control a creature by rewarding or punishing its

actions. Although the game got good press and sold well it did not have any far reaching influence on the game development scene. Despite offering interesting possibilities for unusual gamedesign supervised learning is very uncommon in commercial games.

ii) Training that uses input data which is associated with output data representing correct results. The trainining algorithm used in chapter 4 to initialize the neural network falls under this category because it uses predefined state/action pairs.

2.4 Unsupervised learning

Unsupervised learning is learning without guidance from an outside source. There are no labeled examples. Agents using unsupervised learning have to rely on their own information and knowledge.

2.5 Reinforcement learning

Reinforcement learning became popular in the 1990s within machine learning and artificial intelligence, but also within operations research and with offshoots in psychology and neuroscience. In reinforcement learning algorithms the program does not receive training information from outside (e.g. a predefined training set) as in supervised learning. Instead, it will receive feedback from the environment, the so-called reinforcement or reward, on its own actions ingame. Agents can be programmed without explicitly telling them how to achieve their goals, instead they are taught by reward and punishment.

This kind of learning is well suited to online learning as long as a good feedback function can be designed. In games this is normally quite trivial because of well defined victory conditions.

Reinforcement learning is a class of problems rather than a set of techniques **[Kaelbling et al. 96]**.

Two main approaches to these problems can be distinguished.

Searching in the space of behaviours until one has found a successful strategy or finding ways of estimating states and actions in the world and assigning them a value.

The former is often done via genetic algorithms and programming, the latter is based on dynamic programming and statistical techniques.

The standard reinforcement learning model

In the standard reinforcement learning model an agent is connected to its environment which is directly influenced by the agent's actions. State and action value functions are used to evaluate the agents progress and the state of the environment. The environment communicates with the agent via the reward signal, telling it if it is doing well or not.

The agent
The Reinforcement learning agent has to be able to sense the state of the environment and its actions have to be able to influence the environment. A Policy π is defined which maps states of the environment to actions. In most basic Reinforcement Learning algorithms a policy will be implemented as a simple function or lookup table. The agent's goal is to optimize its policy so that it generates actions which lead to the greatest possible reward.

The environment
The environment is typically modelled as a Markov Decision Process.
The MDP is defined by: A set of states S, an action set for each state A(s), the probability of a transition between states s and s' given an action a P(s, s', a), the expected reward R(s, s', a) and the discount rate for delayed rewards γ.
When it is not possible to model the environment as fulfilling the Markov Property it is nonetheless possible to use reinforcement learning algorithms and ideas as long as a state represantation can be found that is a close approximation of a Markov state **[Sutton98]**.

The reward and value function
The goal of RL algorithms is the maximisation of the long-term discounted reward. This means a policy π has to be found which maps actions to states in such a way that the value (expected future reward) of each state and each state, action pair is maximized. Value function are used to judge the long term reward, reward functions define the immediate reward given to the agent and are therefore highly problem specific.
Value functions are normally defined in the follwing way, they can be used to reward states or state/action pairs:
The state-value function $V^{\pi}(s) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3}... | s_t = s, \pi\}$.
The action-value function $Q^{\pi}(s,a) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3}... | s_t = s, a_t = a, \pi\}$.
Values have to be consistently reevaluated for a Reinforcement Learning algorithm to improve. One of the biggest tasks in designing such an algorithm is to improve the value function in fact this is the basis to most Reinforcement Learning algorithms.

Optimal value functions and policies
Reinforcement Learning theory proved that there are optimal value and policy function. The solution to finding those and thereby solving the reinforcement learning problem relies on solving the Bellman optimality equation, which entails exhaustive search, looking ahead at all possibilities, computing their probabilities of occurrence and their desirabilities in terms of expected rewards. In practice this is only possible under specific circumstances**[Sutton98]**.

- Perfect knowledge of the dynamics of the environment.
- Enough computational resources to complete the computation of the solution
- The task suffices the Markov property

Exploration and exploitation
One of the characteristic challenges in developing Reinforcement Learning algorithms is to find a tradeoff between exploration, trying out new actions, and exploitation, taking actions that are already known to be good and which therefore will produce a high reward. Always choosing the highest valued action is referred to as the greedy policy.

Delayed rewards

When taking an action an agent often cannot tell what consequences it will have in the future. To be able to correctly value such an action an agent has to be able to learn from delayed rewards. A reward given by a future action, made possible by past actions.

There exist many different types of reinforcement learning algorithms based on the above model that are able to solve these problems. Monte Carlo methods and Temporal difference learning to name two.

The disadvantage of these techniques in the scope of this thesis is their assumption that the state and action space or some kind of generalized represantation can be ennumerated and somehow stored in tables. This is much more complicated for the state/action space of a commercial computer game . In a continuous, constantly changing environment with many independent agents as it is typical for these games this becomes impracticable. Another key problem is the fact that the state of the environment is not only dependent on the player's actions, one of the assumptions of the MDP. The environment is highly dynamical and influenced by other AI agents and the human player(s).

This doesn't mean that reinforcement is useless for learning in video games. Many of the ideas behind the reinforcement learning algorithms can be used to help in creating adaptive game AI.

Value functions can be generalized by neural networks. An approach which will be used by one of the learning architectures presented in chapter 5.

The idea of immediate reward [Kaelbling et al. 96] is used as basis for the first learning algorithm used in chapter4, which will later be refined with delayed rewards.

2.6 Artificial neural networks

The human brain is composed of billions of neurons. Connections betweens these neurons, called synapses form a complex network that is used to process and store information.

Artificial neural networks, attempt to imitate this model on a smaller scale.

The artificial networks used in science are quite simple by comparison only containing comparatively few (a dozen or so) neurons. A few specific applications can use networks of up to thousands of neurons, still much smaller than our brain. Therefore the artificial networks are not as all powerful as the human brain but they can be used for very specific tasks.

Artificial neural networks can also be understood in a less biological sense. They are often understood as mathematical function approximators. Input to the network represents independent variables, while output represents the results and dependant variables. The network itself is a mathematical function giving one unique set of output for one set of input.

A big advantage of neural nets is that they can represent highly nonlinear functions very well.

The AI community uses many different kinds of neural networks to solve all sorts of vastly different problems, from pattern recognition to data processing and many in between. Neural networks often are combined with other techniques such as fuzzy systems, genetic algorithms, and probabilistic methods.

There are many different kinds of networks to today. Some of the most important are: Feedforward neural networks (layered and general), time delay networks, perceptrons, recurrent networks or cascading neural networks.

In this thesis multilayer feed-forward networks will be used with backpropagation for training purposes.

Evolutionary Artificial Neural Networks

Instead of reinforcement learning to teach and evolve a neural net evolutionary algorithms can be used. These are genetic algorithms that create permutations of the network and judge the results by a fitness function. The best networks are then used to create a new generation on which the process is repeated. This has been shown to be a working method for evolving neural nets offline but is considered to be too slow for online learning **[Spronck05]**. Evolutionary networks are therefore genreally not viable for the kind of online learning explored in this thesis. There is however research in the area of online evolutionary algorithms **[Stanley et al. 05]**.

Artifical neural networks in games.

In the scientific game community neural nets are mainly used as function approximators, often for value functions such as in TD-Gammon **[Tesauro95]**. This approach is not very useful for commercial games as modern games use continous, highly complex environments which are hard to evaluate. But there have also been first attempts at adapting neural networks for usage in commercial games **[Spronk05]** and **[Smith01]** in the last years.

There are many more different ways of using artificial neural networks in games. They can be used used to simplify the coding of complex state machines or rules-based systems and substitute some of the decision making parts of the game AI. One of their most attractive features is the ability to adapt and learn during gameplay, thereby potentially improving the gaming experience by tuning themselves to the playing style of the player. A neural net will also be able to deal with situations the programmer might have forgotten to code into a finite state machine.

However, game developers are still very hesitant to use neural nets in commercial games. This is probably due to several factors.

Firstly, although neural networks are great at handling nonlinear problems that cannot easily be solved using traditional methods their inner workings are hard to understand because they constantly change of themselves. It is therefore very hard for a human to comprehend how they produce their results. Secondly, it's difficult at times to predict what a neural network will generate as output, especially if the network is programmed to learn or adapt within a game. Compared to traditional AI techniques like finite state machines, which are prevalent in game programming, they are hard to test and debug. Thirdly, adaptive AI in general is seen with scepticism by game programmers because

it produces unpredictable AI. The danger that the AI will deteriorate after the game is distributed and thereby decrease the enjoyment a customer gets from playing the game **[Miikkulainen et al. 06]**.

Early attempts to use neural networks in games involved complete AI systems composed entirely out of neural nets to completely control a game agent. Such an approach amplifies the problems game programmers have with neural nets concerning predictability, testing, and debugging.

A different approach will be used in this thesis, namely the use of several small neural nets which are highly specialized and work in tandem with more traditional AI techniques. Thus we hope to lessen the unpredictability and show a more attractive use of artificial neural networks for game progamming.

This kind of a modular network hierarchy, composed of independent expert networks has already been investigated for other purposes such as implementing game evaluation functions for Backgammon **[Boyan92]** and Tic Tac Toe and been proven superior to the monolithic approach with one big network. It is easier to implement and faster **[Wiering95]**.

Some possible uses of neural networks in games

Control of vehicles

In Robotics neural networks often are used as controllers for robots. For example to control a robot's motor control system. Inputs from the sensory system are used to detect obstacles ahead and fed into the neural network. The net's outputs are then used to steer the robot.

In games a similar network could control some kind of vehicle, a robot even.

Assessment of game situations

In a strategy game in which the player can build different kinds of units to fight against the AI, a neural net could be used to predict the possible threat posed by the player's unit composition and choose the AI players reaction accordingly.

Behavioural control

In a RPG game a neural network could control how certain creatures in the game behave, for example whether a creature will attack the player, run away or flock with other creatures. In chapter 4 a scenario very similar to this will be implemented.

Generalization

Another big advantage of neural networks compared to more static control structures such as finite state machines is their ability to generalize. This means that a neural network will always be able to respond to input that the programmer might not have in mind while programming it. A finite state machine or a decision tree based control algorithm can normally not cope with such situations as well as a neural network.

3. The Chase/Flock/Evade Task

3.1 The task

The task consists of a group of AI agents that have to chase down and eliminate a single human controlled agent. The AI agents' decisions are controlled by an artificial neural net. To test the viability of using neural nets in a realistic setting, a simple real time engine similar to those in many games will be used. The neural net will be trained both offline, before the task is run, and online to allow the agents to adapt to the player while the simulation is running.

The purpose of implementing this task is to gain insights into implementing neural networks to control computer game agents. The first step is to create working agents that exhibit visibly intelligent and believable behaviour as well as the ability to adapt to changing game conditions. This will be described in detail in chapters 4 and 5. Chapter 6 will then expand on the basics learned in the previous chapter and improve the network implementation as well as devise methods to design and test a neural net more methodically. A second network will then be added to support the first one, creating a small hierarchy of interconnected nets.

3.2 The agents

There are two types of agents in the game. The player and the AI agents. Both have an equal amount of life (hitpoints), a 360° field of vision and unlimited sightrange, they move at a constant speed and their turning rate is capped at a maximum amount of 10° per gametick.

The AI agents can switch between three behaviours: Chasing, evading and flocking. These are implemented using deterministic algorithms describe in detail in chapter 4. The decision process is controlled by a neural net. All AI agents use the same net, they thereby share their experiences.

To test the adaptive qualities of the neural net the attributes of the player controlled agent will be changed online, while the game is running.

Flocking

While flocking the agents try to stay close to other agents in their immediate vicinity. While in this state they exhibit a group movement. The algorithm is based on 3 principles **[Reynolds87]**:

- Cohesion: Have each unit steer toward the average position of its neighbors.
- Alignment: Have each unit steer so as to align itself to the average heading of its neighbors.
- Separation: Have each unit steer to avoid hitting its neighbors.

Evading

When an agent is low on health it shall avoid the player in order to survive. In order to do this it simply steers away from the player and tries to avoid his course.

Chasing

When the agent is in a good position to attack the player it shall intercept and chase him, until the player dies or the agent has to retreat. To do this the agent plots a course over the last 2 known positions of the player and tries to get in front of him.

3.3 The environment

The playing field is a 2 dimensional plane without obstacles, it uses floating point coordinates to simulate a continuous environment. The playing field is restricted by boundaries. If agents leave the restricted area they reenter the playing field on the opposite side, continuing to move in their current direction. There is no information hidden from the agents, they can access the exact position and state (health, direction etc.) of all other agents at any time.

3.4 Combat

The game uses a highly abstracted form of combat. Every AI agent close to the player loses a set amount of hitpoints per gametick, the corresponding distance will be refered to as combat distance. The player loses a set amount of hitpoints for each AI agent close to him. Once there are no enemy agents close every agent slowly regains hitpoints up to its starting amount of health.

3.5 The artificial neural network

The AI agents use a 3-layer-feed-forward neural net to control their behavioural state. This net is accessed by all agents collectively. Before the start of the game it is trained using a supervised learning method. A training set with given state/action pairs is used for this. An artificial neural network was chosen instead of some other techniques such as a finite state machine or a decision tree because of its ability to generalize over the different states. This allows it to produce sensible outputs even for inputs not originally considered while training the network. In addition to this a tried and tested supervised learning algorithm could be used to train the network; the backpropagation learning algorithm.

The network is composed of 4 input nodes, 3 hidden nodes and 3 output nodes.

Inputs: The number of friendly agents in close proximity.

The health of the current agent.

If the player is engaged with AI agents.

The distance to the player.

Outputs: The behavioural strategy the agent shall use, namely evading, chasing or flocking.

Evaluation criteria

To evaluate the artificial neural net, several criteria have to be taken into account.

The following computational and functional requirements towards online learning as defined in **[Spronck05]** will be used to discuss the results.

Computational requirements:

- Speed: Online learning must be computationally fast as it takes place during runtime.
- Effectiveness: Online learning must create effective game AI, at least as good as manually created code.
- Robustness: It has to be able to deal with randomness.
- Efficiency: The wanted behaviour has to emerge swiftly enough (after a low number of trials) to be useful ingame.

Functional requirements:

- Clarity: The emergent behaviour has to be easy to understand.
- Variety: The learning process has to yield a variety of behaviour so that the game does not get boring.
- Consistency: The average number of trials before the wanted behaviour emerges has to be consistent, there should not be a huge variety in the amount of time it takes to learn.
- Scalability: The learning process must be able to scale to the difficulty level of the game.

In addition to these requirements a simple fitness function will be used to compare different neural nets to each other and to a hand coded deterministic AI using a finite state machine.

4. Implementation of the Chase/Flock/Evade task

4.1 Goals of the implementation

The goal of the implementation of the Chase/Flock/Evade task is to show several ways of using a neural network in a game like environment. The emphasis in this first implementation lies on testing different methods to train the neural net and finding rules to build a workable game AI using a net for decision making. To implement the task defined in chapter 3 a small game engine capable of controlling agents in a real time, continous environment was created. It will be refined and augmented in the following chapters.

4.2 Tools

The program is written in C++ using DirectX to manage the graphical and input components of the game as well as all the vector arithmetics needed. It is therefore dependant on Windows as operating system and uses the .net framework.

4.3 Architecture

It is divided into the following components:

Entry function

Creates and initializes the Windows classes and engine components. All game variables including the agents are initialized and the neural net's basic training is run. After this the main game loop is run. A simplified pseudo code version of the game loop looks like this:

```
while(game_is_running)
{
        read_input();           //user_input is read and processed
        run_simulation()        //the simulation class runs the game logic, including the AI
        run_3d_engine()         //the current gamestate is put on screen
}
```

Figure 4.1 Game entry function

The game class

A helper class (Game.cpp) that contains pointers to all the data that needs to be shared by the different components such as the agents and several global variables such as the mapsize. A 256 units long playing field is used, the combat distance is set to 16 units, coordinates are double values.

The 3d-engine

The engine class (Engine.cpp) contains all methods needed to draw the map, the user interface and the agents onto the screen. The class uses 3 dimensional vectors enabling it to draw 2 and 3 dimensional scenes.

The agents

The agents are implemented using an abstract base class (Entity.cpp) from which they are inherited. The base class provides basic methods for moving, turning, checking distance to other agents etc. Ai agents include methods for the deterministic, simple part of their AI, namely their intercept, flock and evasion routines. The number of AI agents is set to 20 for all the tests in this chapter. The maximum hitpoints of player and AI agents are set to 20, their speed to 0.5 units per gameround and the damage done to the player by each AI agent in combat distance is set to 0,3 hp per gameround. The player damage is variable.

The chase algorithm

The chase algorithm implementation in pseudo code can be written as follows:

```
if (player in front of agent) // player in 180° front arc, plot intercept course
{
        rangeToClose = prey.getCoordinates() - coordinates;
        predatorVelocity = speed * direction;
        preyDirection = newPosition - oldPosition;
        preyVelocity = (prey.getSpeed()) * preyDirection;
        closingVelocity = preyVelocity - predatorVelocity;

        timeToClose = Length(rangeToClose) / Length(closingVelocity);
        futurePositionOfPrey =
                prey.getCoordinates() + prey.getSpeed() * preyDirection * timeToClose;

        newDirection = futurePositionOfPrey - coordinates;
}
```

```
else  //player in 180° back arc, stop and turn until player in front arc
{
        stop();
        turnAround();
}
```

Figure 4.2 Chase algorithm

The flocking algorithm

While flocking an agent checks all friendly agents in flocking distance. If they are close enough and in a 270° angle to his front they are considered neighbours that are taken into account while flocking. The agent now calculates their average position and heading. The average of these two is used as the new heading for the agent. If another agent is too close separation is factored in too, to avoid collisions. The separation is mixed into the new heading as a corrective force.

The flocking algorithm implementation in pseudo code can be written as follows:

```
for (all agents in neighbourDistance)
{
        distance = testedEntity.getCoordinates() - activeEntity.getCoordinates();
        directionToTestedEntity = testedEntity.getCoordinates() -  activeEntity.getCoordinates();
        angle = activeEntity.getAngleTo(testedEntity)
        if (angle < 270°)
        {
                neighbourCount++;
                averagePosition += testedEntity.getCoordinates();
                averageDirection += testedEntity.getDirection();
                if (distance < closestDistance) //separation
                {
                        closestDistance = distance;
                        closestNeighbourPosition =testedEntity.getCoordinates();
                }
        }
}

if (neighbourCount > 0)
{
        averagePosition = averagePosition / neighbourCount;
        directionToAveragePosition = averagePosition - activeEntity.getCoordinates();
```

```
        newGoal = averagePosition + averageDirection;
        if (closestDistance < closestWantedFlockingDistance)
        newGoal = newGoal + activeEntity.getCoordinates() - closestNeighbourPosition;
        activeEntity.setNewDirection(newGoal);
}
```

Figure 4.3 Flocking algorithm

The evade algorithm

The evade algorithm implementation in pseudo code can be written as follows:

```
if (player is left of agent)
{
        turnRight10();
}
else //player is right of agent
{
        turnLeft10();
}
```

Figure 4.4 Evade algorithm

The simulation

The simulation class (Simulation.cpp) controls the game logic and the higher level AI including the online learning algorithms and the neural network.

First the combat is resolved, after this the decision making process for the agents is run, consisting of the AI routines for the computer controlled agents and the input processing for the player controlled agent. After all agents have decided on their goals the desired movement of all agents is checked by the game logic and is carried out if possible, killed agents are respawned close to the center of the map.

The artificial neural network

The neural net is implemented in two classes: NeuralNetwork.cpp and NeuralNetworkLayer.cpp. The number of layers is fixed to 3, all other network variables are flexible (number of nodes, inputs, outputs, learning rate etc.). A backpropagation training method and a feedforward algorithm are implemented and can be used for online and offline learning. The sigmoid function was chosen as activation function. Learning rate (0,9) and momentum (0,2) are set to one value for both off- and online learning.

Inputs

The inputs are as follows, they are scaled to lie between 0 and 1.

- Input 0: Number of friendly agents in combat distance / total number of AI agents
- Input 1: Health of current agent / maximum amount of hitpoints
- Input 2: 1 if player in combat with at least on AI agent , 0 if player not engaged
- Input 3: Distance to player / length of map

The scaling of all inputs to a common range prevents big numbers, such as the distance (up to around 362), to drown out the lower numbers, such as the player engaged variable (0 or 1).

Outputs

The following actions are associated with the nodes in the output layer.

- Output 0: Chase
- Output 1: Flock
- Output 2 Evade

The action identified by the node with the highest value is chosen as the final output of the network.

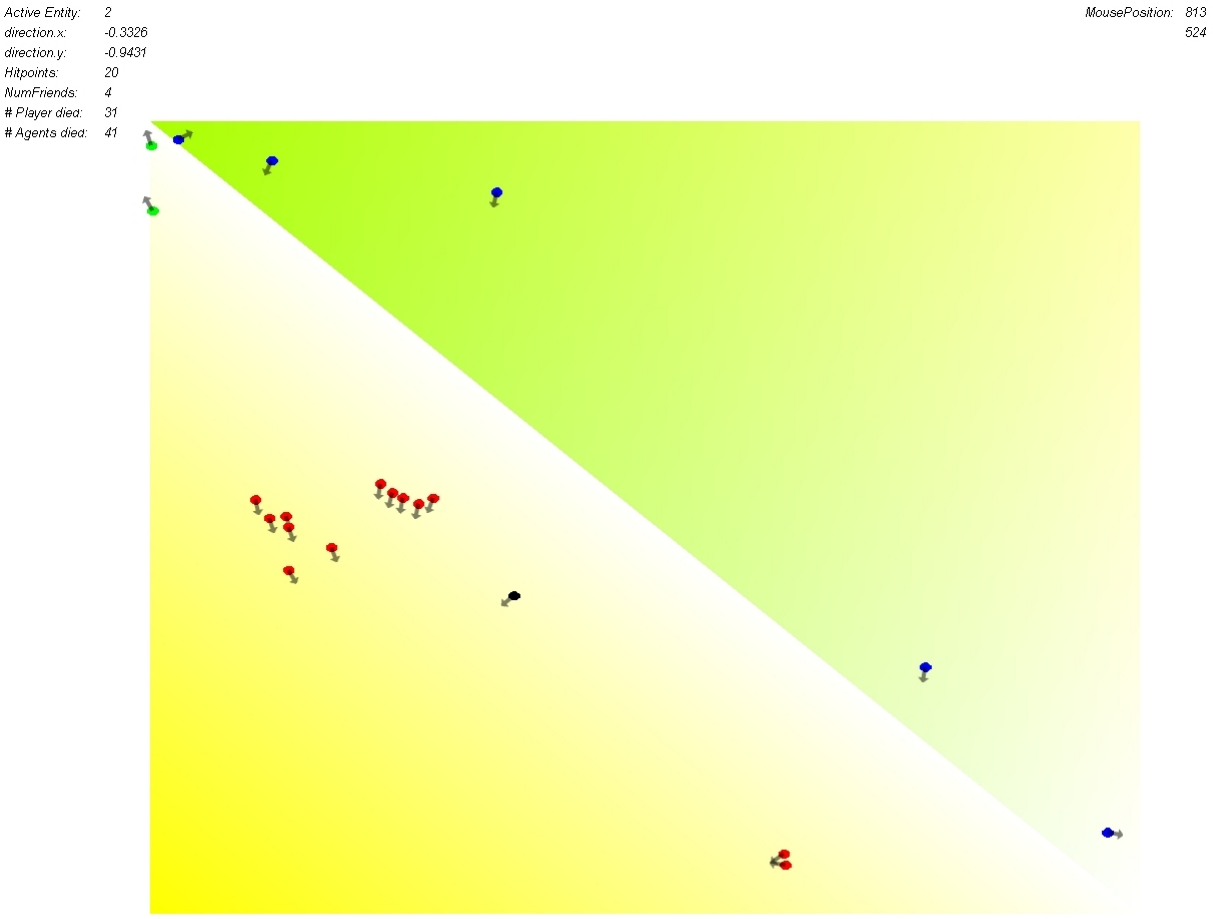

Figure 4.5 Screenshot of the game

Figure 4.5 shows a game in progress: The agents are depicted as circles with an arrow that is pointing in their current direction. Their state is colour coded: Black = Player, Red = Chase, Green = Evade, Blue = Flock

5. Creating the agent AI

The first subgoal in developing a working agent AI was to create a visibly intelligent behaviour of the computer controlled agents in the context of the Chase/Flock/Evade task. To achieve this several methods were tested, namely off- and online supervised learning and online reinforcement learning. The first version of the supervised learning was taken as the basis for all these methods and improved several times as there was no suitable algorithm found in the literature. Chapter 5 describes the process of improving the basic supervised learning through the addition of several rules that counter some of the unwanted side effects of an online learning algorithm. This technique was then named controlled learning.

The following steps are described in this chapter:

- Offline training of the neural network to achieve basic behaviour.
- Implementing a first online supervised training algorithm driven by training events.
- Implementing additional rules to steer the behaviour.
- Introducing the concept of controlled learning.
- Melding the techniques together to form a better learning algorithm.

The original tests were done with a human player which of course added a lot of randomness. As a result of this and the experimental nature of the development process in this early stages the focus during this stage was simply the creation of believable behaviour. To allow for a more thorough analysis of the AI, the human player was later exchanged for a computer controlled one, which was used to run more detailed tests. This will be described in detail in chapter 6. Chapter 7 will then introduce a reinforcement learning algorithm that will supplant the supervised learning.

Wanted behaviour

To be better able to judge the agents' AI a definition of the wanted behaviour has to be given. Three characteristics are important:

- Aggressiveness: An agent that is close to the player shall try to chase and kill the player unless it has no chance to achieve its goal because it is too damaged or alone.
- Agents which are alone and far away from the player are supposed to flock in order to form bigger groups until they are in a strong enough position to attack again.
- Agents which are hurt shall flee and evade the player until they are healed again.

5.1 Basic training

A supervised training method was used to create the initial agent behaviour.

The first step in initializing the neural net is to find a good set of state/action pairs for the offline training. This training set could then be used to set the weigths to values that help the agents to start the game already exhibiting a smart behaviour. The training itself is run via the backpropagation algorithm.

Once a good training set is found it can be used as a basis for further on- and offline learning.

Finding a good training set is highly dependable on the task of the neural net, therefore no good algorithms for defining useful state/action pairs can be utilized. Simple experimentation was used to find a set that led to a satisfying starting behaviour.

The training set consists of 16 state/action pairs. A state is define by 4 values which are used as inputs for the neural net, they are scaled exactly like the inputs in chapter 4.3. The corresponding action is defined by the desired values of the output nodes. For the outputs values of either 0.9 or 0.1 were chosen, to reflect activated or deactivated nodes.

The following values were used:

State / Inputs				Action / Outputs		
Number of friends	Hitpoints	Player engaged	Distance to player	Chase	Flock	Evade
0	1.0	0	0.2	0.1	0.9	0.1
0.0	1.0	1	0.2	0.1	0.9	0.1
0.2	1.0	0	0.2	0.9	0.1	0.1
0.1	0.8	1	0.2	0.9	0.1	0.1
0.0	1.0	0	0.8	0.1	0.9	0.1
0.1	0.5	0	0.2	0.9	0.1	0.1
0.0	0.25	1	0.5	0.1	0.9	0.1
0.0	0.2	1	0.2	0.1	0.1	0.9
0.1	0.2	1	0.2	0.9	0.1	0.1
0.0	0.2	0	0.3	0.1	0.9	0.1
0.0	1.0	0	0.2	0.1	0.9	0.1
0.0	1.0	1	0.6	0.1	0.1	0.9
0.0	1.0	0	0.8	0.1	0.9	0.1
0.1	0.2	0	0.2	0.1	0.1	0.9
0.0	0.25	1	0.5	0.1	0.1	0.9
0.0	0.6	0	0.2	0.1	0.1	0.9

Figure 5.1 The initial training set for the neural network

Creation of state/action pairs for the training set

To create these values the following procedure was used

1. Choose a possible state of the controlled agent.
2. Translate the state into inputs for the neural network.
3. Choose a desired action for the state.
4. Translate the desired actions into output values for the neural network.
5. Map the translated state to the translated outputs.
6. Repeat as often as deemed necessary

The interesting part is step 1. Examples of states and actions which are conclusive enough to lead to intelligent behaviour have to be chosen. In the described case 16 examples were sufficient to create the wanted basic AI. Alternatively a list with all possible states could have been constructed and then outputs for all of them would have to be defined. The sheer number of similar or unimportant states makes this behaviour prohibitively slow. The learning also takes longer the more examples are used for the training. One of the advantages of the neural network is that it can generalize output values for many more different inputs than the 16 it was explicitly trained for. The refinement of the behaviour was planned to be achieved mainly through online learning in later stages of the program. Therefore the decision for a relatively low number of state/action pairs was made. This should simulate a situation similar to the one found in commercial games which all have a working AI at release, the online training's task is to change and improve this basic behaviour, not to create new behaviour from scratch every time the game is run.

5.2 Experiments

Setup for the experiments

For testing the neural network a number of experiments based on the following setup were run.

The AI agents start in close formation around the center of the map. The player starts in the top right corner. The player tries to eliminate as many AI agents as possible while not getting eliminated in turn. Once an AI agent is eliminated it is respawned close to the center of the map. The player respawns in a random spot when he is killed.

To test the adaptiveness of the net the damage rate of the player is changed online, making him more or less lethal to the AI agents. The agents were expected to become more aggressive and attack in smaller numbers when the player is weaker and prefer to evade him and only attack in greater numbers when he is stronger.

The first test is a simple observational test. The game is played for 6000 rounds with its initial training and learning settings to test the normal behaviour. Then the game is run 2 more times for 6000 rounds each with a considerably higher and a lower damage rate. This is repeated several times to account for the inherent randomness in the player's movement. The damagerate describes the amount of health the player takes off each AI agent in combat distance during one gameround. The damage dealt to the player is set to 0.3 hitpoints per AI agent in range.

The only statistical data collected from test 1 is the average ratio of AI agents deaths to player deaths, further refered to as death ratio.

5.3 Basic artificial neural network

Observations about the emergent behaviour from the basic neural network trained with the training set from chapter 5.1. No online learning implemented yet.

Testrun	Player damagerate	Death ratio
1	*0.5 hitpoints damage per gameround*	0.67
2	1.0 hitpoints damage per gameround	5
3	*0.4 hitpoints damage per gameround*	0,5

Figure 5.2 Basic network tests

Testrun 1: original damage rate (0.5 hp / gameround)

The AI agents start to chase the player from the beginning and exhibit quite aggressive behaviour. When they are in a distance of roughly a quarter of the length of the map they chase the player unless they are alone. If they are in a group of 4 or more agents they tend to switch to the chase state at a slightly higher distance than in smaller groups. When they are alone they only chase the player if he is very close (about 16 units). Agents that are farther away choose to flock. The evasion state is nearly not used at all, only when an agent is attacking the player alone and close to death. In this case the evasion state is entered too late for the agent to get away and it dies. At the end of the test the AI agents are scattered into small groups of 2 to 5 units due to flocks getting separated and agents getting killed.

	State of agent	Frequency of action
Chase	If close and friends > 1	Medium
Evade	Only when alone and close to death	Very low
Flock	When not close	High

Figure 5.3 State / action summary basic network

Testrun 2: higher damage rate (1.0 hp / gameround)

Because of the missing ability to adapt the player has a much easier time to chase the AI agents. The big difference in ratios compared to run 1 confirms this.

Testrun 3: lower damage rate(0.4 hp / gameround)

As could be expected the opposite effect to run 2 can be observed.

Summary of the tests:

The agents exhibit intelligent behaviour. The chase and flock actions dominate.

5.4 Online supervised learning

Online supervised learning triggered by player and agent kills

The next step was to implement some kind of online learning. It was decided to begin with a set of simple supervised learning rules based on certain events happening to the agents. This approach was inspired by the reward in reinforcement learning theory. Instead of directly rewarding the current action taken by the agent as in true reinforcement leearning a preferred action is proposed which gives better control over the resulting behaviour.

- *Event 1:* If agent was close to player while player died, reinforce chase behavior under the current circumstances.
- *Event 2:* If agent died, encourage evasion behaviour in the current circumstances.

The retraining of the neural net works as follows:

Every time an agent is in one of the two aforementioned situations the training method of the neural net is run. The same kind of feed forward + backpropagation training method is used as for the offline learning. The only difference is that only one state/action pair is used instead of a whole training set.

This simple algorithm works directly on the neural network which can be seen as the action selection policy in reinforcement learning terms. No value function is defined to help in steering the learning process into the right direction. The reward in shape of the player or agent death is directly translated into a policy change.

The tests for the neural network using this simple supervised learning method were then repeated using different damage rates for the player making him deal more or less damage to nearby AI agents.

Observations about the emergent behaviour from the neural network incorporating supervised learning.

Testrun	Player damagerate	Death ratio)
1	*0.5 hitpoints damage per gameround*	0.86
2	1.0 hitpoints damage per gameround	4.84
3	*0.4 hitpoints damage per gameround*	0.73

Figure 5.4 Supervised learning tests

Testrun 1: original damage rate (0.5 hp / gameround)

The AI agents exhibit very aggressive behaviour at the start, as long as a big group of about 6 or more stay close together, they will chase the player even over long

distances (the whole length of the map). Once the game has run for a longer time and the agents split into smaller groups the behaviour starts to change. The smaller groups, about 3 or less agents tend to evade the player at long enough ranges to leave him no chance to get to them. Small groups at longer distances tend to flock. After about 3000 to 4000 gamerounds only greater packs chase the player anymore. In a few test runs the behaviour of the AI agents tended to sway strongly to either chasing or evading. In the case of heavy chasing it led to a big group of agents following the player making it hard for him to kill any agents without dieing himself. In the case of heavy evading it lead to a big number of agents running around on their own avoiding the player and only if bigger groups of about 6 to 10 found together via flocking they changed their behaviour back to chasing the player. There was definitely a bigger variety in the behaviour between the iterations of the testrun compared to the previous iteration of the neural network.

	State of agent	Frequency of action
Chase	If close and friends > 6	High
Evade	When alone or in small group and player comes closer, / when low on hitpoints	Low
Flock	When alone or in small groups	Medium

Figure 5.5 State / action summary supervised learning

Testrun 2: higher damage rate (1.0 hp / gameround)

The behaviour changes more drastically in run 2. About round 4000 the number of flocking AI agents has sunk very low and most agents use the chase or evade behaviour. Interestingly the chase behaviour is used more than the evade behaviour in most cases although the AI agents have a very hard time against the player because of the high damage rate as can seen by the ratio. In one extreme case where the player was able to eliminate more than 10 AI agents without dieing himself the AI nearly exclusively switched to the evade behaviour. Overall the extent of the behavioural changes is much higher in run 2 compared to run 1, this most likely stems from the higher number of deaths, be it player or AI agents deaths, which in turn leads to more occurances of training the neural network.

Testrun 3: lower damage rate(0.4 hp / gameround)

The changes to the behaviour are slightly stronger than run 1, as the evade behaviour becomes very rare after about 2000 to 3000 rounds. Apart from this the behaviour is very similar to run 1.

Summary of the tests

The behaviour seems more intelligent as before as there is a visible adaptation process going on. One rare case of undesired behaviour could emerge though: Suddenly all agents switch from one state to another, sometimes several times shortly in a row. After a short analysis this behaviour could be traced to the quite simple output function used in these experiments. This behaviour can occur when, due to some weight changes, the probability for two output nodes to produce near identical values is very high. In such a case the output function simply chooses the higher one, even if the

difference is very small. Very small state changes can then lead to a fluctuation in the output function. Another unwanted behaviour was the near elimination of the flocking action in run 2. This stems from the simple fact that none of the reinforcment rules boost the flocking action.

The behaviour exhibited by the neural network so far meets several of the criteria formulated in chapter 2.

Computational requirements:

- Speed: The tests so far have concentrated on the quality of the resulting behaviour, less on the computational burden of the learning algorithm. At this point the online learning algorithm is fast enough to not slow down the program. To reliably test the computational component of online learning with neural networks another set of tests will have to be devised.
- Effectiveness: There is no hand written conventional AI yet to test the neural network against.
- Robustness: The agents adapt to the changes in the damage rate. Most notably resulting in a stronger use of the evasion behaviour when the damage rate is raised.
- Efficiency: The wanted behavioural changes occur reasonably quickly, in most tests slight variations of the agents' behaviour could be observed after about 4 to 5 training events.

Functional requirements:

- Clarity: The changes in behaviour are easy to understand in most cases. The only notable exception being the sudden switch of all agents to chasing from flocking as described in the summary of test 2.
- Variety: The agents adapt their behaviour and become slightly more cowardly or aggressive during our testruns dependant on the kill ratio.
- Consistency: In all test runs the speed in which the behaviour changed was quite close to each other. It normally took about 2000 to 4000 gamerounds for the agents' behaviour to change notably.
- Scalability: The big change in ratios between the test runs 1 and 3 compared to 2 hints at the inability of our supervised learning to cope with the big change (doubling) of the damage ratio. The evasion behaviour was not preferred strongly enough to keep a similar kill ratio to the other test runs.

5.5 Rules

Adding rules to address the weaknesses of the basic supervised learning

To improve on the above mentioned weaknesses it was decided to add several rules that are not triggered by an event happening to the agent, but that are called when an agent is in a specific state.

- *Rule 1:* If not in combat distance to player and less than 3 friends are close -> Flock.

This rule's intention is to avoid the flocking behaviour getting suppressed by the learning of the other 2 event driven learning rules.

A few quick tests with this added rule showed a big problem. The behaviour for small groups was changed to flocking exclusively. After a very small amount of gamerounds no evasion or chase behaviour was exhibited by any agents. The reason for this lies in the much higher number of training events for this rule. Learning rules for events 1 and 2 are only called when an agent dies, be it the player or an AI agent, the new rule is called by every AI agent in every gameround in which the conditions are met (small group, not in combat). To counter this effect we added a time based condition to the rule, so that it only gets called every 100 gamerounds. After this was added , the flocking behaviour did not drown out the other behaviours anymore but it was still exhibited stronger than before. This could become a problem in the later stages of a game when the agents are more fragmented and rarely find together to bigger flocks.

To counter this effect one more rule was added.

- *Rule 2*: If not in combat distance to player and more than 3 friends are close -> Chase.

The tests were then repeated with the new ruleset.

Observations about the emergent behaviour from the neural network incorporating supervised learning with additional rules.

Testrun	Player damagerate	Death ratio
1	*0.5 hitpoints damage per gameround*	0.98
2	1.0 hitpoints damage per gameround	6.1
3	*0.4 hitpoints damage per gameround*	0.8

Figure 5.6 Supervised learning with additional rules tests

Testrun 1:

The changes in agent behaviour are less noticable than before (test 2). Flocking and Chasing dominate, evading is only used when an agent is close to dying. If the evading agent is close to friends he is normally able to escape, if he is alone the player is able to catch him.

Testrun 2:

The AI agents use the evade routine much more and try to flee. Because of the high damage rate they have a hard time escaping as can be seen in the high ratio. Compared to run 1 a definite change in their behaviour can be observed, albeit smaller than in test 2.

Testrun 3:

The AI agents stay quite aggressive only very small groups flock. Evasive behaviour can only be seen from isolated agents.

Summary of supervised learning

Tests 2 and 3 showed several problems when implementing game AI with neural networks using supervised online learning for adaptivity. The following rules were concluded from these.

- *Number of training events important:* If some rules are triggered more frequently this can lead to a suppression of the lesser frequently used rules. Make sure that the most important rules do not get eliminated by some unimportant ones.
- *Rules needed to promote every possible action:* If an action is never reinforced it will be eliminated after a while. Make sure that every action has at least one rule that strengthens it, unless you expressly want it to disappear after some time.
- *Type of game important for evaluation of behaviour:* To be able to evaluate the emergant behaviour it is important to define a clear goal for it.

5.6 Controlled learning

Controlled learning without supervised learning

In the previous tests, one of the weaknesses found in the behaviour of the AI agents was the unpredictability. In a few cases the outcome of the test differed significantly from the others (chapter 5.4). This is a common problem for traditional, reward based online learning architectures using neural nets and one of the main reasons why they are scarcely used in commercial games. To alleviate this unwanted effect a rules based supervised learning component was added (5.5).

This segment will build up on the rules based learning approach and test a more complex set of rules to train the neural network. This learning process can run parallel to the normal event driven learning and help to control the behaviour. The rules are imprinted onto the same neural network that is used for the supervised learning. In chapter 5.6 only the rules based learning will be used, in chapter 5.7 both approaches will be combined.

The following rules were used.

- Rule 1: If agent is dying faster than player -> Evade
- Rule 2: If agent is healthy, alone and not in combat distance -> Flock
- Rule 3: If agent is far away from player (more than half the map) -> Flock
- Rule 4: If entity is not alone and close to player -> Chase

Observations about the emergent behaviour from the neural network incorporating pure controlled learning.

Testrun	Player damagerate	Death ratio)
1	*0.5 hitpoints damage per gameround*	0.85
2	1.0 hitpoints damage per gameround	4.0
3	*0.4 hitpoints damage per gameround*	0.84

Figure 5.7 Controlled learning tests

Testrun 1:

As could be expected from the rules the agents attack when they are close to the player and in small groups. When they are far away from the player they will always flock together. When they are in combat with the player they will start to evade if they are alone or in small groups, if they are in a bigger group they will not evade.

Testrun 2:

Although the ratio is comparable to the earlier tests a rise in lethality could be observed. More agents died but the player was also killed more often. This stems from the improved flocking behaviour, especially during the later rounds of the game. The bigger groups mean that the agents are more aggressive which leads to the higher number of deaths.

Testrun 3:

As could be expected no difference to run 1 was observed because there is no learning in place at this time.

Summary of controlled learning

The purely rules based approach to learning used to train the neural net in this test is able to generate clearly defined and intelligent behaviour, similar to the earlier tests. It has no adaptive qualities though.

5.7 Controlled supervised learning

Controlled learning rules together with supervised learning

In this test the supervised learning rules were combined with the controlled learning rules adding an adaptive component to the agents' behaviour again.

Observations about the emergent behaviour from the neural network incorporating controlled supervised learning.

Testrun	Player damagerate	Death ratio)
1	*0.5 hitpoints damage per gameround*	1
2	1.0 hitpoints damage per gameround	4.1
3	*0.4 hitpoints damage per gameround*	0.7

Figure 5.8 Controlled supervised learning tests

Testrun 1:

The variation in behaviour was obvious. The agents start to chase the player at longer ranges and they put a big emphasis on the player's state (already engaged or not). Both these behaviours could not be seen in the purely controlled network (chapter 5.6).

Testrun 2:

There were big changes to test 4. In the early phase of the game (about 1000 rounds), it could happen that all agents switched to evasive behaviour simultaneously. This effect got countered by the rules based part of the learning algorithm after a few rounds though.

Testrun 3:

Similar behaviour as in run 1 could be observed.

Summary of controlled supervised learning

The added adaptation through the reward based training is clearly visible in the isolated tests and also when the damage rate of the player is dynamically changed during the game. The rules based learning helps to control the behaviour, especially when the game runs for a longer time. In the early phase of the game there were a few disturbances between the two learning methods leading to unwanted, but shortlived behaviour. These could be avoided by using a better pre-trained neural network.

5.8 Comparison

Both types of AI, the learning adaptive AI and the hand coded controlled AI improved the behaviour of the basic neural network but they also proved to be less efficient as can be seen by the ratios. Their behaviour was less monotonous, as the agents switched more frequently between the different states and all the states were used while the basic network nearly completely ignored the evade state.

Figure 5.9 shows that supervised learning with rules creates the most inefficient behaviour so far while the basic network surprisingly is the most efficient at the low and standard damage segments while supervised and controlled learning improve the performance during the high damage segment.

Chapter 6 will show if this will last or if the supervised learning simply needs a longer time to take effect and influence the agents' behaviour more positively.

Overall the controlled and controlled supervised learning showed more stable and clearly defined behaviour than the purely supervised learning AI which in rare cases created very monotonous behaviour, e.g all agents evading and not switching back to any other state.

	Normal	High	Low
Basic network	0.67	5	0.5
Supervised learning	0.86	4.84	0.73
Supervised learning with rules	0.98	6.1	0.8
Controlled learning	0.85	4	0.84
Controlled supervised learning	1	4.1	0.7

Figure 5.9 Comparison of death ratios

5.9 Summary of chapter 5

In this chapter the agent AI for the task defined in chapter 3 was implemented, showing that a neural network can be used to control agents in a real time continous environment. Several ways were then devised to generate intelligent behaviour and adaptive qualities of the agents.

The methods used were:

- Immediate event triggered supervised learning: Producing adaptive but unpredictable behaviour.
- Rules based learning: Producing clearly defined but non-adaptive behaviour.
- Controlled supervised learning: Producing an adpative but easier to control behaviour.

Some of the weaknesses the neural network showed:

- Ineffective evasion behaviour: Single agents were never able to escape the player if they were in combat because they started to evade too late. This is a result of the basic supervised learning used which only takes the state of an agent into account at the moment of the event triggering the training, not his past actions leading to this state.
- Unwanted behaviour in early stages of the game: This problem can be simply avoided by using a longer time to train the neural net offline, before running the game.
- The neural network can only be used to control single agents at a relatively low level. So far there is only rudimentary cooperation between the agents. This is more of a design weakness than a functional one. The inputs and outputs chosen simply define only low level behaviour.

Based on these observations the following areas will be researched in the next chapters:

- Modifications to the learning algorithm.
- Adding another neural net to allow for improved cooperation between the AI agents.
- Improving methods of regulating the adaptiveness and the learning speed of the neural network.
- Implementing a reinforcement learning algorithm to replace the supervised learning.
- Developing better formal measures of fitness for the neural network, to allow for more in depth comparison of the algorithms.

6. Supervised learning

The goal of this chapter is to refine the methodology used in chapter 5 to create the AI for the AI agents. This will be done by further improving the supervised learning algorithm and running more detailed tests. In addition to the controlled and supervised learning introduced in chapter 5 a short term memory will be implemented and two different action selection strategies will be used: Greedy and softmax selection.

The following learning strategies will be examined in this chapter:

- Supervised offline learning
- Supervised online learning
- Supervised online learning with rules
- Controlled online learning
- Supervised online learning with memory

6.1 New parameters

To be able to better compare the behaviour of different neural networks in the Chase/Flock/Evade task the following parameters are defined. Measurements of them are collected while online learning takes place to better judge the progress of the training process.

- Death ratio: Player deaths / AI agent deaths
- Number of player deaths: Sum of the times the player agent died during on game.
- Number of agent deaths: Sum of all AI agent deaths during one game.
- Sum of turns per state: The total sum of gamerounds agents spend in each one of the three possible states.
- Training events: Number of training events which occurred during a game.
- Sum of training events per state: Total reinforcements applied to each state

6.2 Implementing a computer controlled player

As a requirement for a good comparison of the different neural networks the randomness in the tests has to be lowered. Therefore a computer controlled player will be used instead of a human player. It will use a fixed, rulesbased AI to provide reliant behaviour. The player behaviour is roughly defined as follows:

```
if (player.health < closestAiAgent.health) player.evade(closestAiAgent)
else player.chase(closestAiAgent)
```

6.3 Selection strategies

The neural network used for the agents' decision making is tested with two different selection strategies. The selection strategy takes the outputs of the neural network into account and then chooses the agent's action based on the following rules:

- Greedy selection: The output node with the highest output is directly translated into the corresponding action.
- Softmax selection: If the differences between the output nodes are bigger than 0.2 use the greedy strategy, otherwise take the highest and the next closest output and randomize between the actions corresponding to the outputs with a 1 to 1 ratio.

The reason for using a softmax selection lies in the possibilty of a training algorithm leading to a suboptimal policy because the initial behaviour does not allow the agent to choose actions that steer the process into the right direction. Instead such an algorithm might only converge on a local maximum as opposed to the global one.

6.4 Test setup

To thoroughly test the changes a longer test than in chapter 5 is run for all the different variants of the Chase/Flock/Evade task. It consists of several repetitions of a game of 300000 gamerounds length. During the first 100000 rounds the damagerate of the player is set to 0.6 hitpoints per gameround, during the second 100000 rounds it is set to 0.4 and finally to 0.8 until the game ends. 10000 gamerounds are equivalent to one timestep in the following diagrams. Thus the standard damage rate lasts from step 0-10, the low damage rate segment from 11 to 20 and the high damage segment from 21 to 30.

6.5 Basic network

Basic neural net

To be able to judge the adaptivity and the performance of the different learning algorithms and network structures a test with the basic net is run first as a benchmark.

Figure 6.1 shows the average state distribution of the agents controlled by the basic net.

It can be clearly seen that the evade state is by far the least used one. This is not surprising as it is only used when an agent is close to death.

The flock and chase behaviour instead are both used while the player is further away and therefore they dominate the agent behaviour. On average the chase behaviour is about twice as common as the flocking.

The variance between the highest and lowest flocking (after the initial steps) lies about 60000 turns, as does the variance in the flocking. The difference between the highest chase and the lowest evade behaviour is 120000 turns per state.

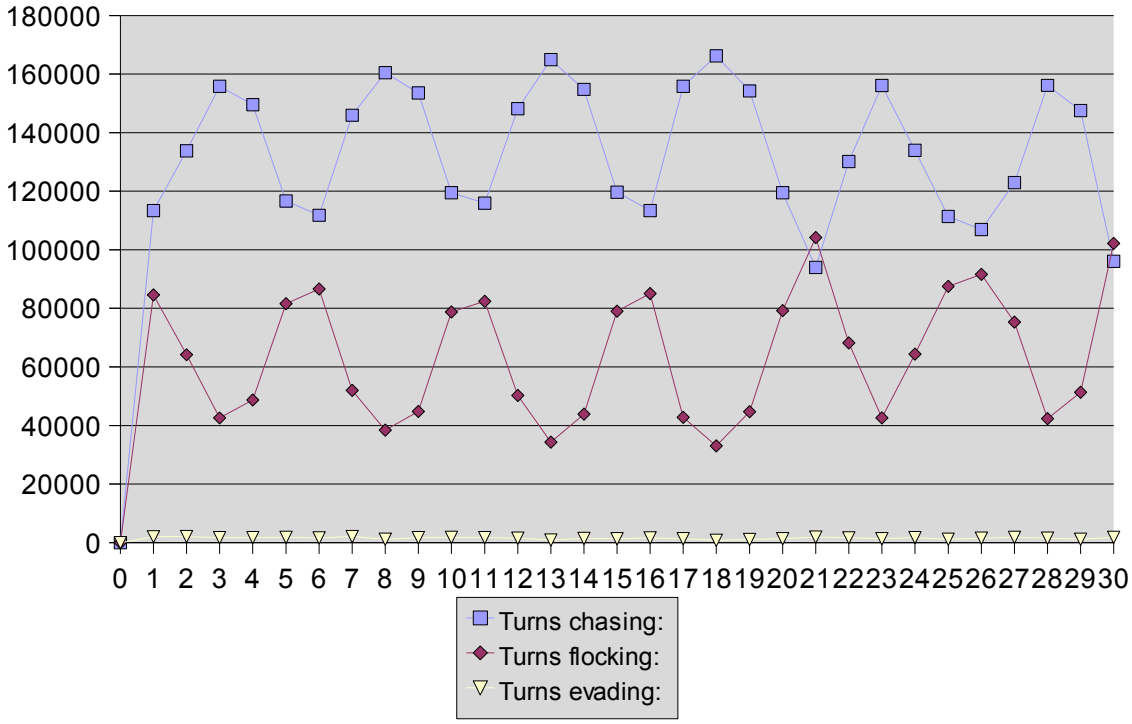

Figure 6.1 Turns per state distribution, basic network

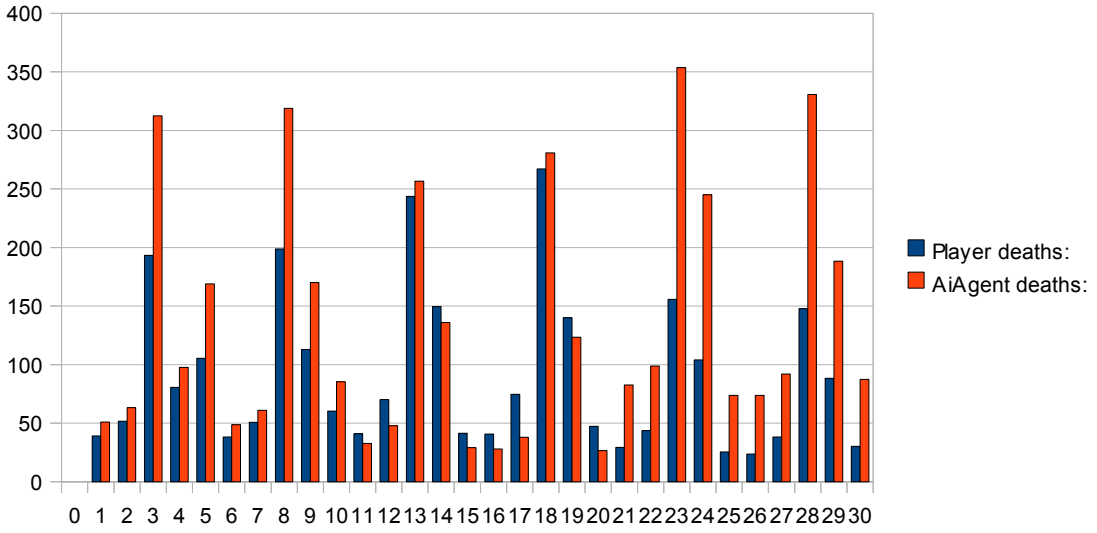

Figure 6.2 Death ratio over time, basic network

Death ratio

The average death ratio of the basic net is 0.68. A distribution over time clearly shows the higher ratio (more player deaths) in the middle segment of the game when the player damage is set lower and contrastingly the lower ratio during the last 10 steps (100000 rounds) of the game.

The average death ratios for the standard settings, lower damage and higher damage are as follows:

- Standard: 0.68
- Low: 1.12
- High: 0.42

6.6 Supervised learning

Supervised learning, greedy selection

Figure 6.3 shows the influence of the supervised learning algorithm. After about 6 steps the flocking behaviour becomes suppressed by the chase and evade behaviour. This confirms the observations made in chapter 5. Chasing dominates heavily, while evading is more noticeable than during the test of the basic network. The overall behaviour stabilizes during the late phase of the standard damage segment. After this the variance of the chase and evade behaviour amounts to only about 10000 to 20000 turns per state, while the absolute numbers are around 190000 turns spent chasing and 10000 evading per timestep.

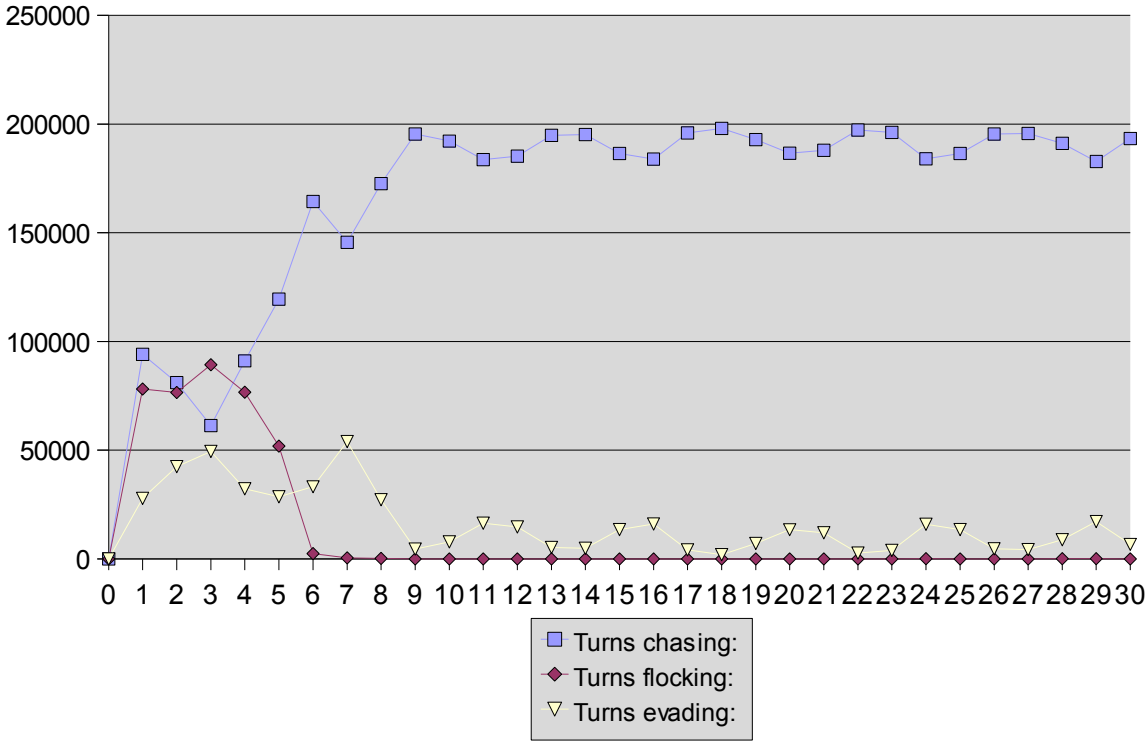

Figure 6.3 Turns per state distribution, supervised learning with greedy selection

Death ratio

The average death ratio of the network trained with the supervised learning is 0.73. The average death ratios for the standard settings, lower damage and higher damage are as follows:

- Standard: 0.67
- Low: 1.39
- High: 0.45

These clearly show that the learning algorithm is able to improve the efficiency of the AI agents during the low damage segment while it doesn't show significant changes in the standard or high damage segments.

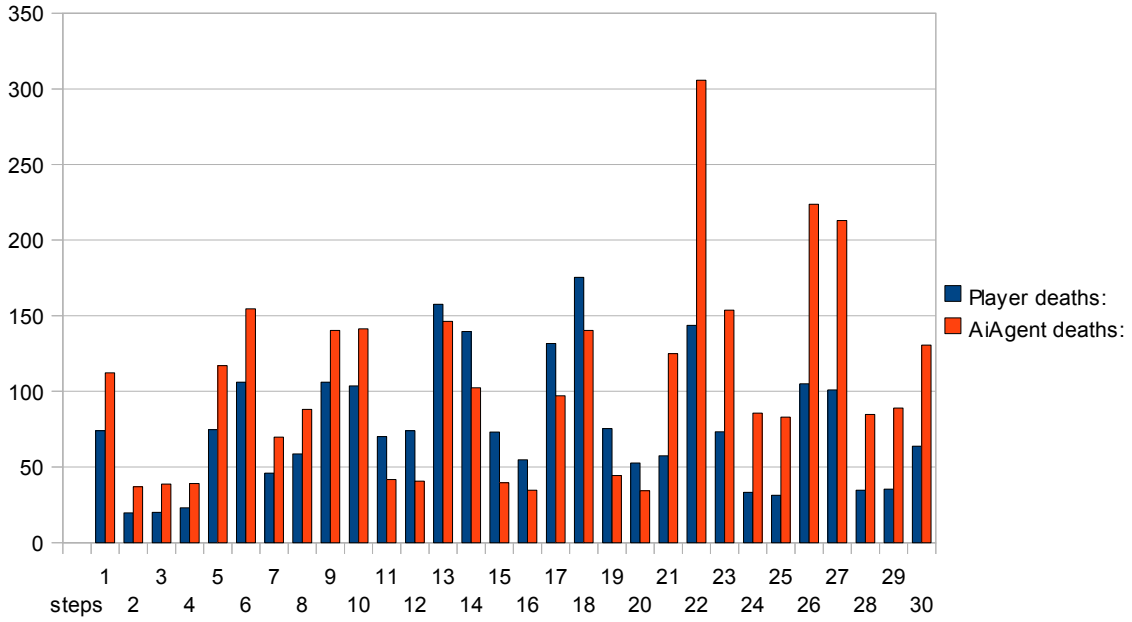

Figure 6.4 Death ratio over time, supervised learning with greedy policy

When comparing Figure 6.4 with 6.5 an interesting aspect of the supervised learning algorithm can be seen, when looking at step 13 a very high peak in training events can be observed, its height doesn't seem to be warranted by the amount of player deaths relative to the number of AI agent deaths. This stems from player deaths while many agents are close at the same time, all getting the reward, leading to a very high number of chase training events per player death. Such peaks influence the policy negatively as the chase behaviour is amplified stronger than wanted. This points to a possible weakness in the supervised learning function which doesn't take the number of AI agents close to the player into account.

The effect explained above leads to a noticeable difference to the expected behaviour which can be seen during the late stage of the test. Although the player damage is set very high and there is a high number of AI agent deaths, the chase behaviour is exhibited stronger than the evade behaviour. The reason for this can be clearly seen in Figure 6.5. at step 22 during the early phase of the low player damage segment, a higher number of chase reinforcing events occurred despite the relatively higher number of AI agent deaths.

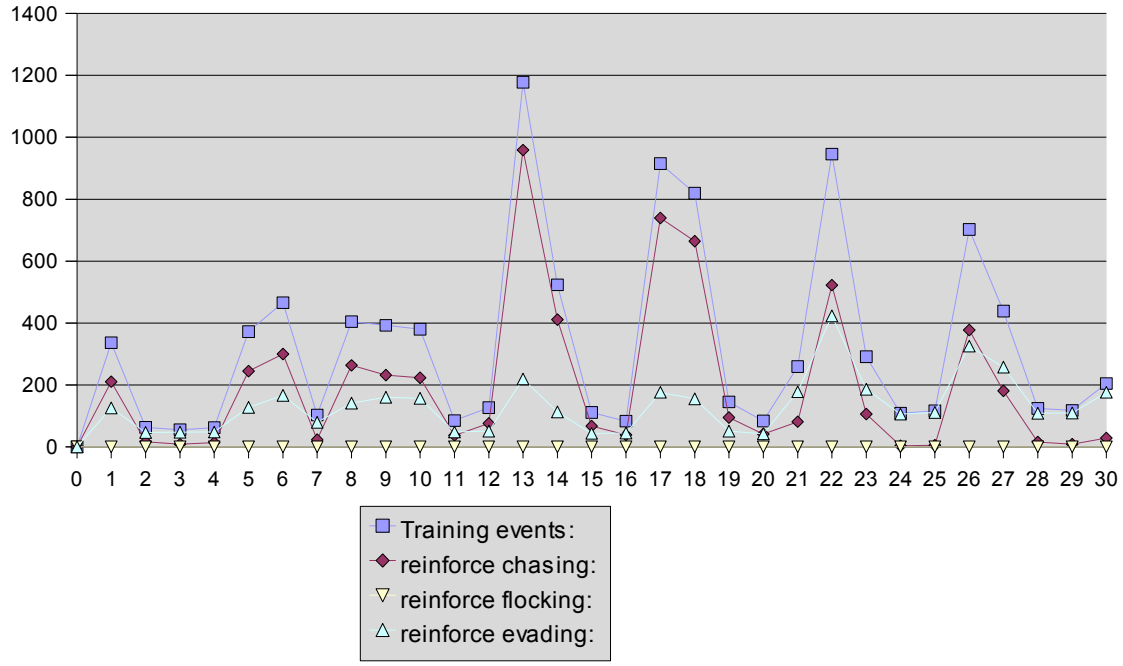

Figure 6.5 Training events, supervised learning with greedy policy

Supervised learning, softmax selection

The softmax selection strategy shows one major difference to the greedy strategy: The evasion and flocking behaviour is suppressed faster. Which leads to a slightly better efficiency overall which can be seen in the slightly better death ratio during the early phase of the testruns. After about 6 timesteps the variance in the chase and evade behaviour goes down to nearly zero.

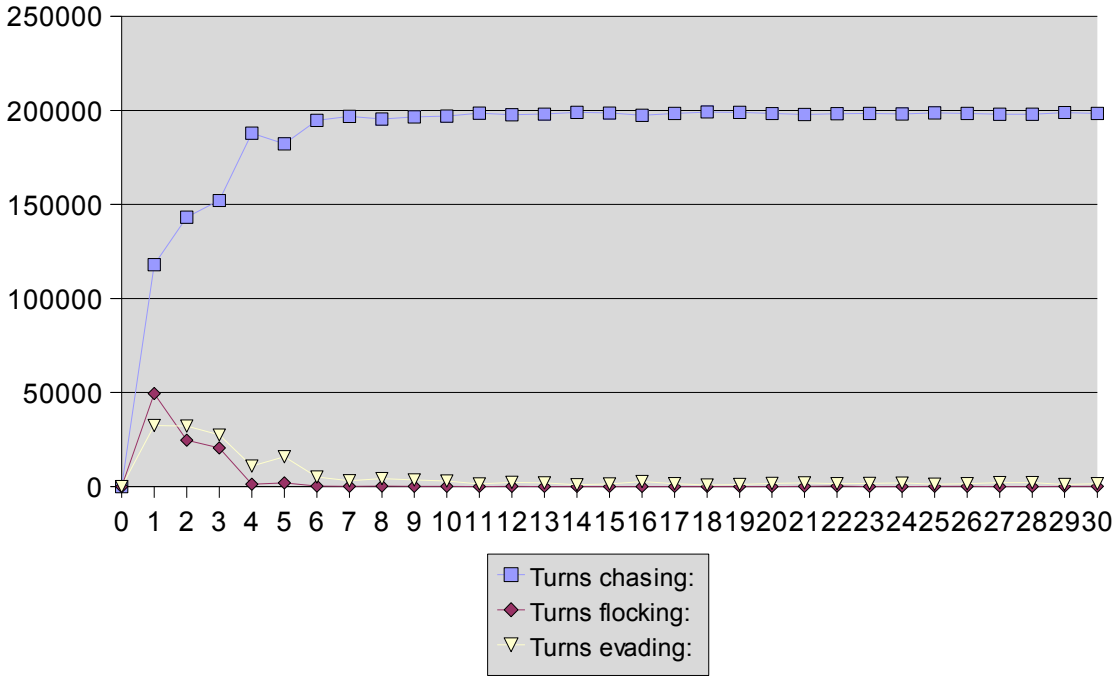

Figure 6.6 Turns per state distribution, supervised learning with softmax selection

Death ratio

The average death ratio of the network trained with the supervised learning and softmax selection is 0.76. The average death ratios for the standard settings, lower damage and higher damage are as follows:

- Standard: 0.74
- Low: 1.4
- High: 0.45

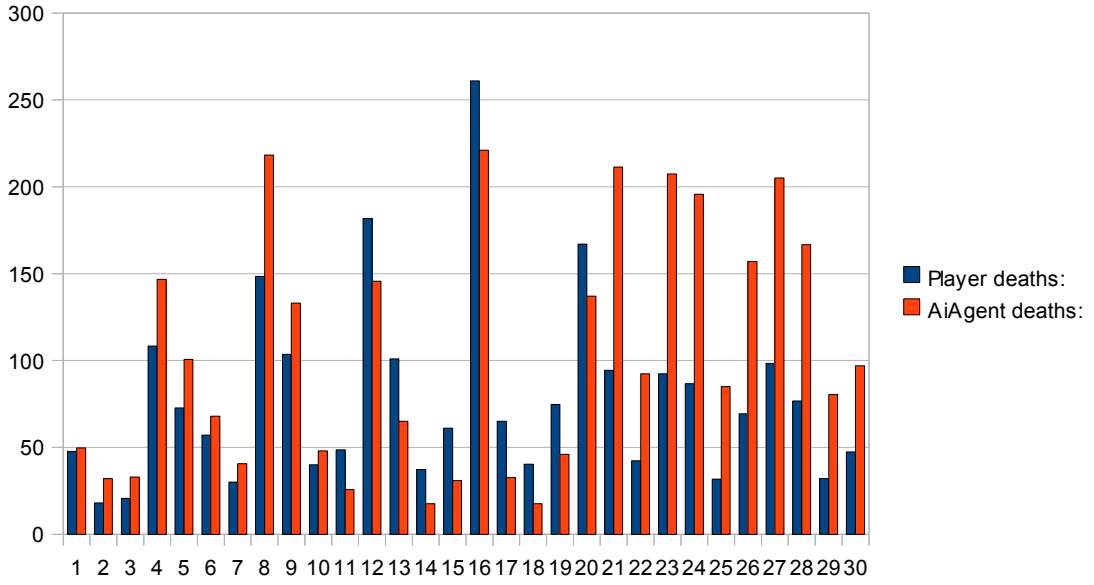

Figure 6.7 Death ratio, supervised learning with softmax policy

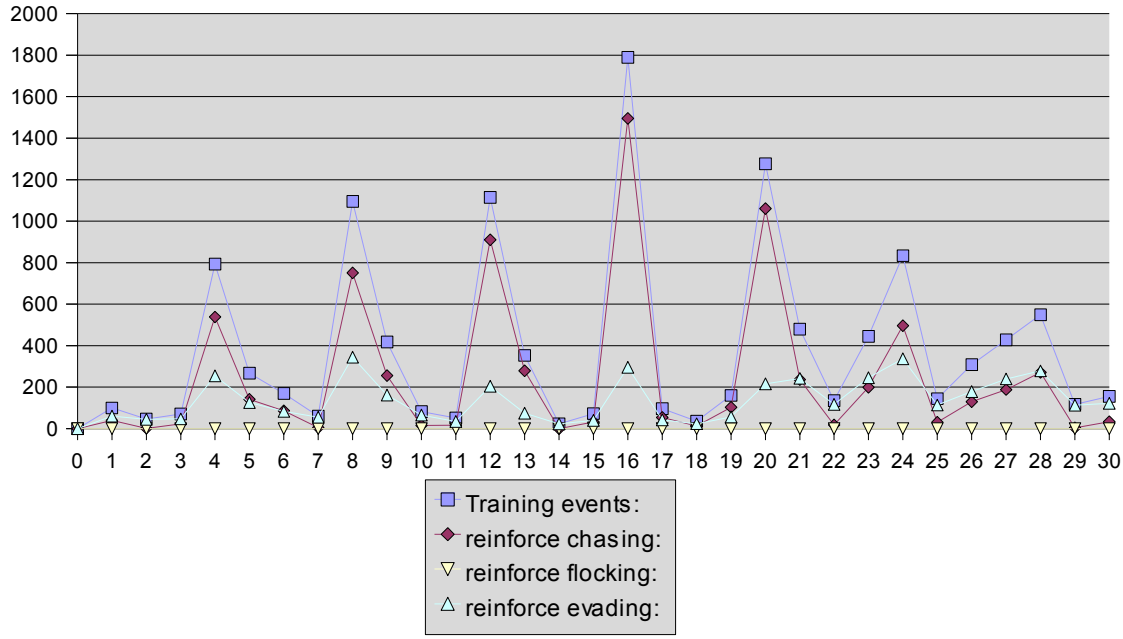

Figure 6.8 Training events, supervised learning with softmax policy

6.7 Supervised learning with rules

The rules from chapter 5.4 are added to the supervised learning algorithm and tested with a greedy and softmax selection strategy.

Supervised learning and rules, greedy selection

Figure 6.9 shows the huge change in behaviour introduced with the rules. The flocking behaviour is not suppressed anymore, instead it is as dominating as the chase behaviour while evading is hardly used at all. Both behaviours swing between 70000 and 120000 turns spent per state taking turns being the most popular strategy for the agents.

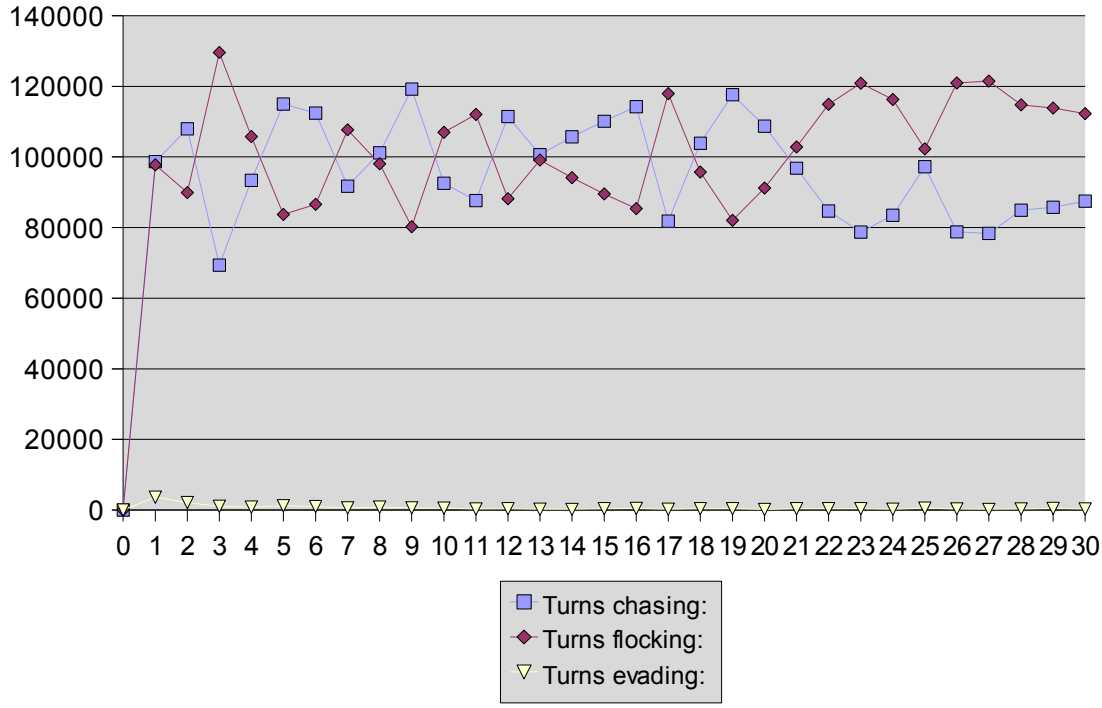

Figure 6.9 Turns per state distribution, supervised learning with rules and greedy selection

Death ratio

The average death ratio of the network trained with the supervised learning and rules is 0.69. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.71
- Low: 1.19
- High: 0.41

Figure 6.10 Shows a similar death ratio distribution over time as the supervised learning without rules despite the differences in behaviour. This can be seen in the absolute numbers for agent and player deaths. The agents with the added rules are less aggressive and therefore kill the player less compared to the pure learning agents but the strengthened flocking also makes it harder for the player to kill AI agents. As a result the average death ratio is nearly identical to the one of the basic network, lower than the one of the pure supervised learning network.

Figure 6.11 very nicely shows the near constant reinforce rate of the flocking behaviour caused by the rules. Interestingly enough the evade behaviour gets reinforced throughout the whole game but not strongly enough compared to flocking and chasing. This leads to the near absence of evading although it does not get weakened, it is just not trained to a similar extent as the others.

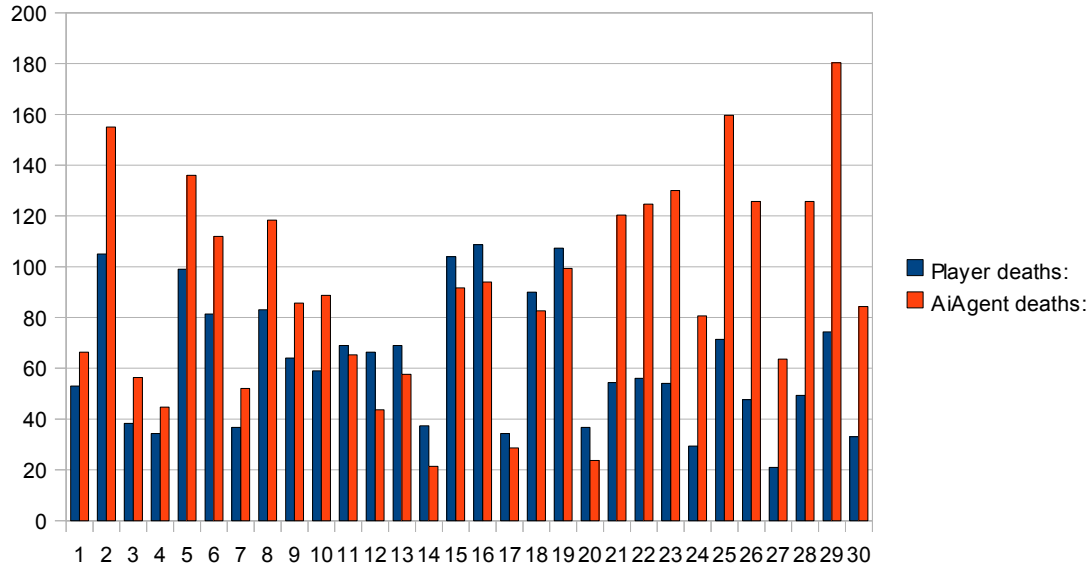

Figure 6.10 Death ratio, supervised learning with rules and greedy policy

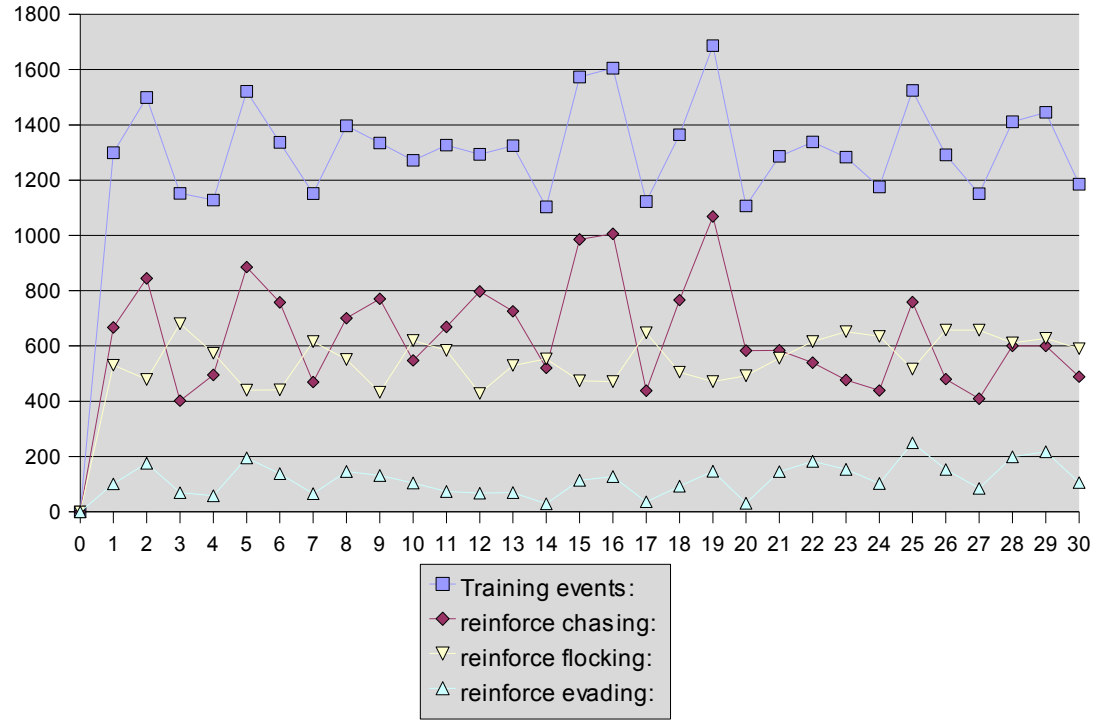

Figure 6.11 Training events, supervised learning with rules and greedy policy

Supervised learning and rules , softmax selection

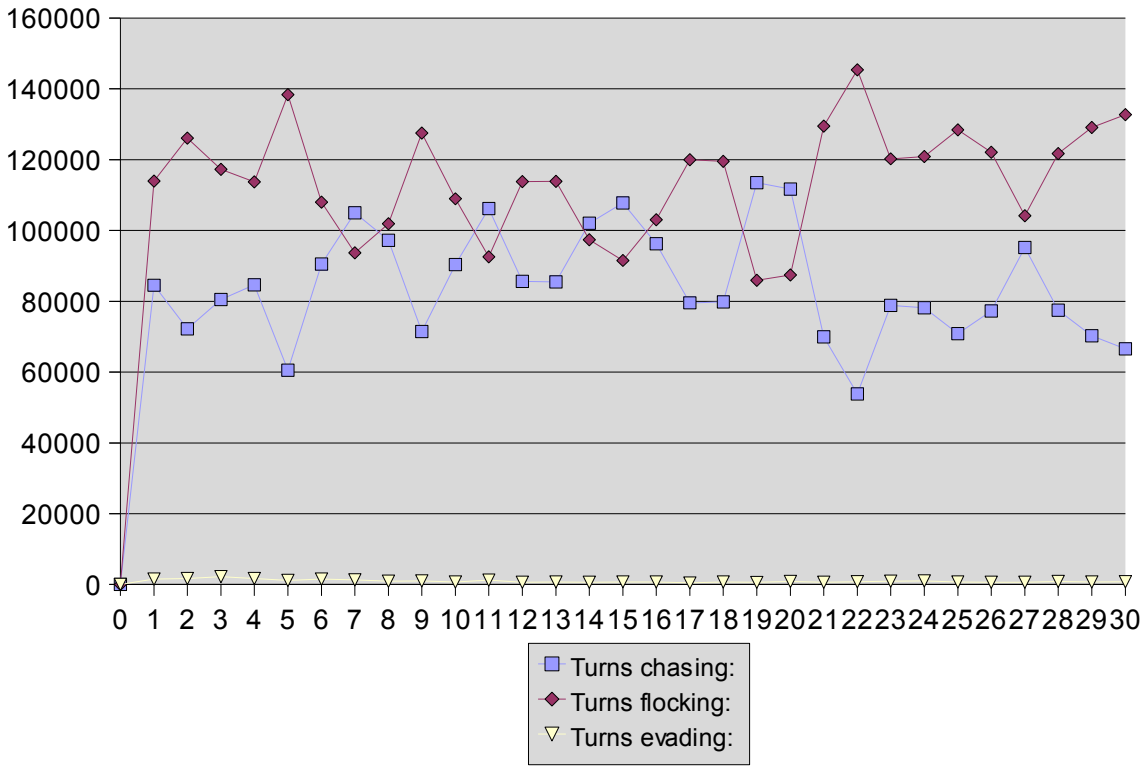

Figure 6.12 Turns per state distribution, supervised learning with rules and softmax selection

In distinction from the greedy selection strategy flocking is nearly constantly prefered to chasing. A similar effect could also be seen in Figure 6.6 where the softmax selection strategy led to a smoother turns per state distribution curve with less position switches between the dominating behaviours.

Death ratio

The average death ratio of the network trained with the supervised learning and rules with softmax selection is 0.68. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.65
- Low: 1.18
- High: 0.39

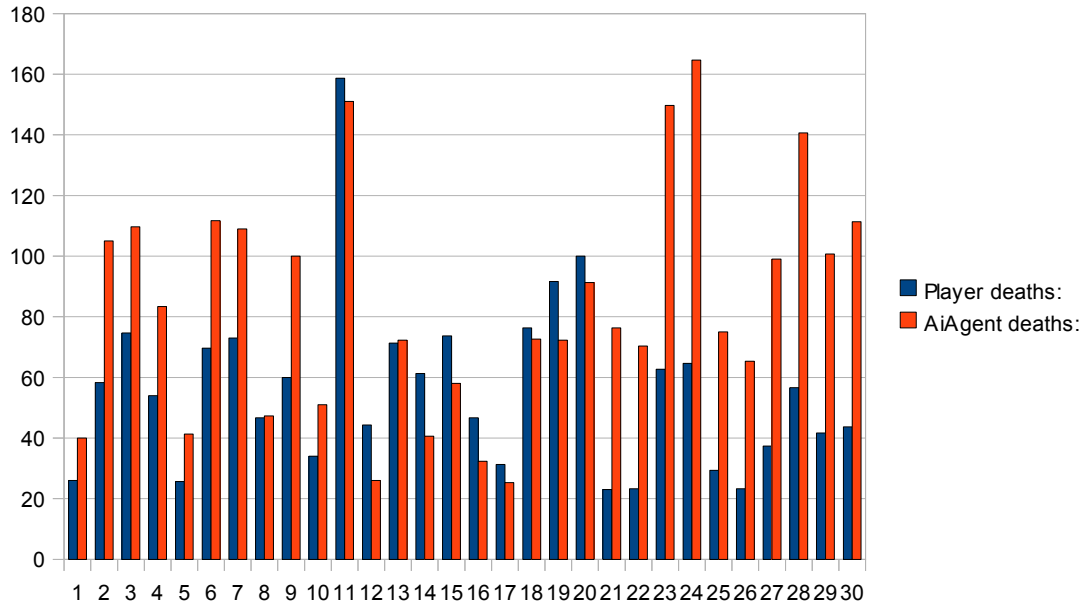

Figure 6.13 Death ratio, supervised learning with rules and softmax policy

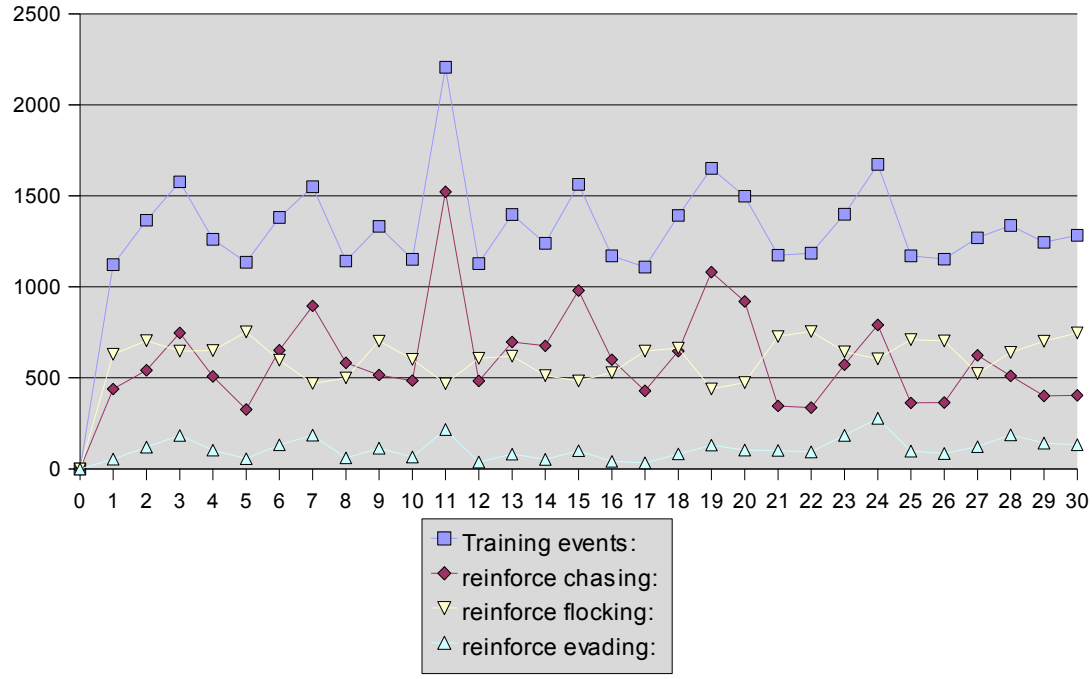

Figure 6.14 Training events, supervised learning with rules and softmax policy

6.8 Controlled supervised learning

The controlled learning rules from chapter 5.6 are tested with a greedy and softmax selection strategy. They are set to run parallel to the supervised learning algorithm.

Controlled supervised learning, greedy selection

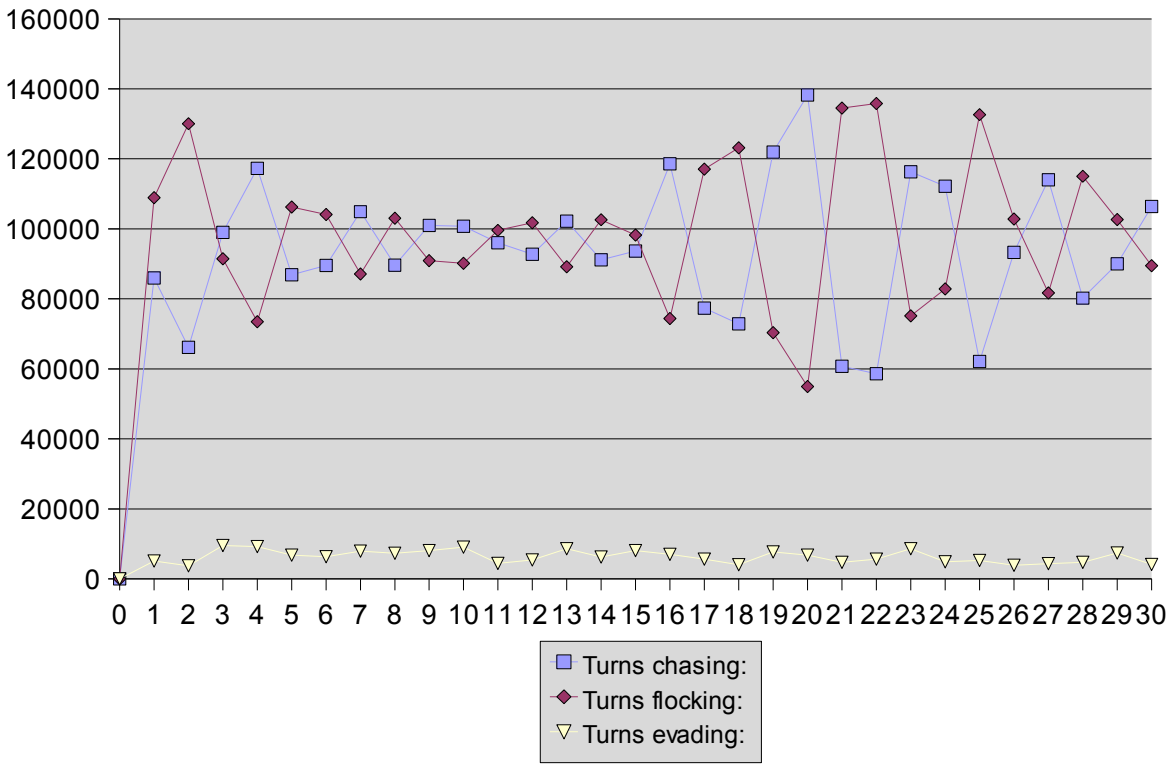

Figure 6.15 Turns per state distribution, controlled supervised learning with greedy selection

When looking at Figures 6.15 and 6.16 the behaviour seems to constantly switch between chasing and flocking with a low but noticeable and durable evade behaviour. The variance between the peaks and minima during the different damage segments is more varied than compared to the former algorithms. A higher number of deaths events and especially a high absolute difference between agent and player deaths leads to a higher variance in behaviour best seen during timesteps 17 to 27. During these periods the supervised learning part of the AI agents influences the neural net stronger (see Figure 6.17) while during periods of relative calm, timesteps 5 to 16, the controlled learning part steers them back to a more balanced behaviour.

Death ratio

The average death ratio of the network trained with controlled supervised learning with greedy selection is 0.73. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.73
- Low: 1.31
- High: 0.42

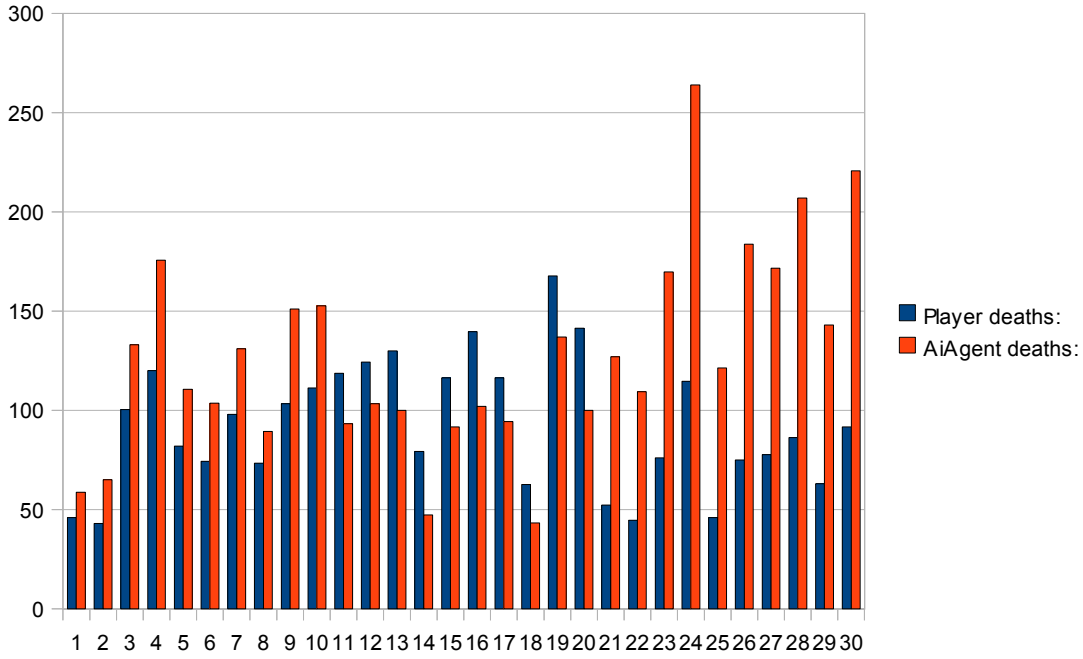

Figure 6.16 Death ratio, controlled supervised learning with greedy selection

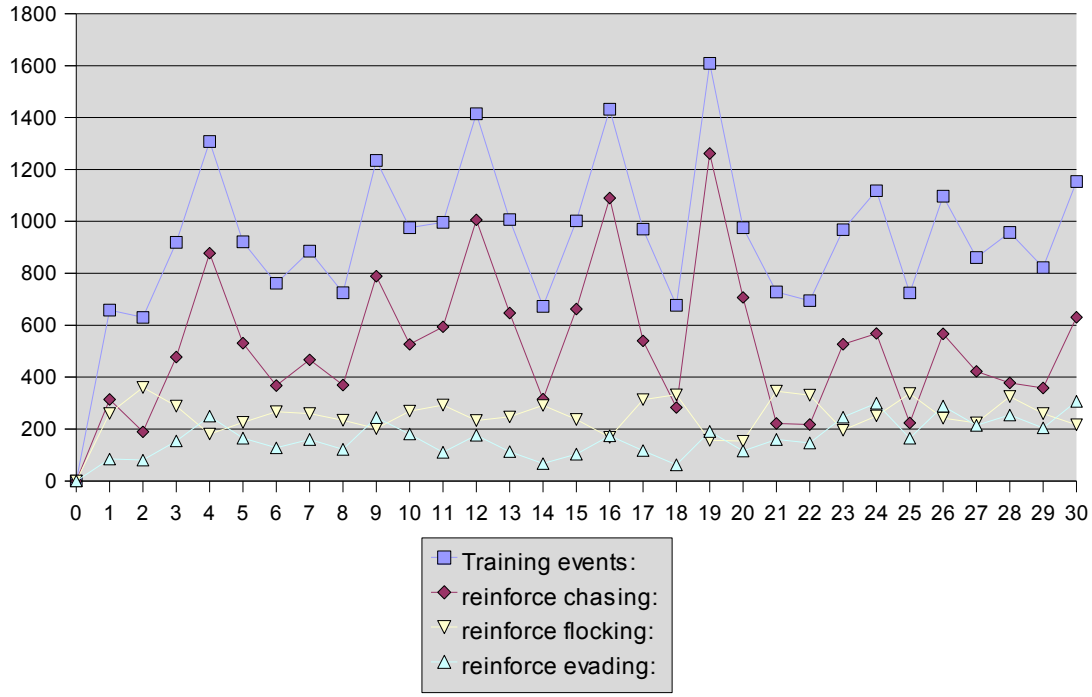

Figure 6.17 Training events, controlled supervised learning with greedy selection

Controlled learning, softmax selection

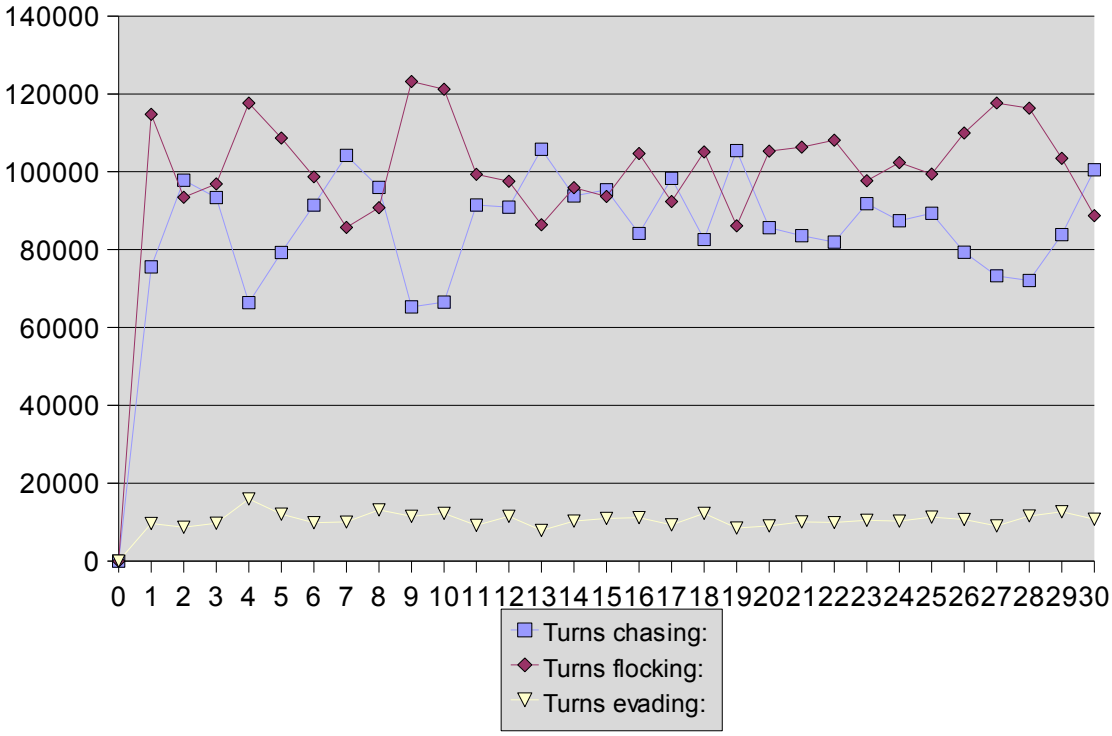

Figure 6.18 Turns per state distribution, controlled supervised learning softmax selection

Figure 6.18 again shows a smoother curve with less variance than its greedy selection counterpart. There are less switches between the dominating behaviours.

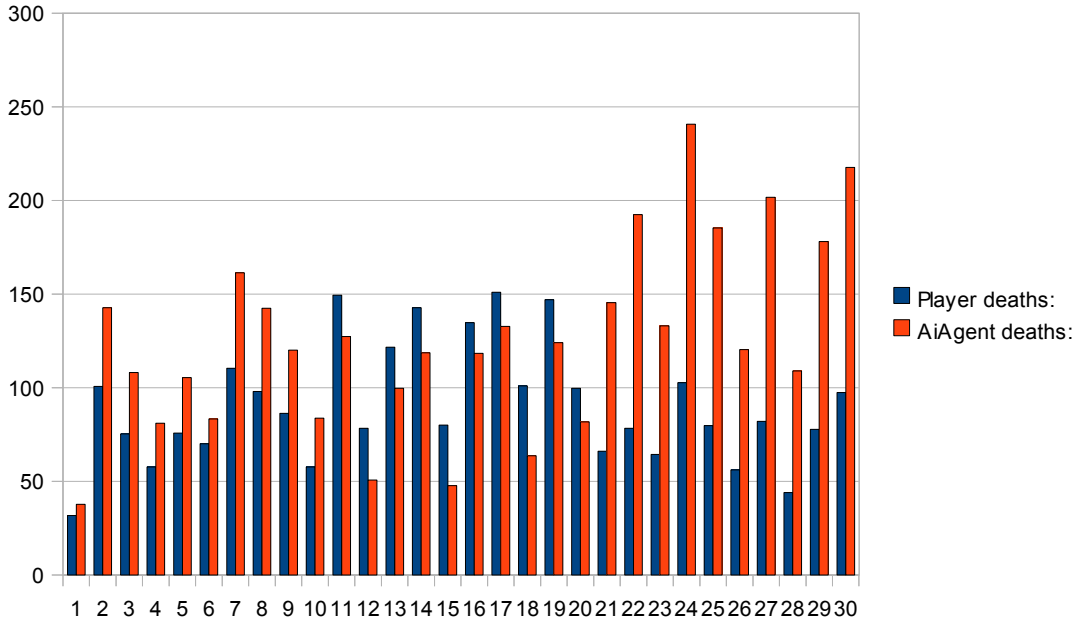

Figure 6.19 Death ratio, controlled supervised learning with softmax selection

Death ratio

The average death ratio of the network trained with controlled supervised learning with softmax selection is 0.72. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.72
- Low: 1.25
- High: 0.43

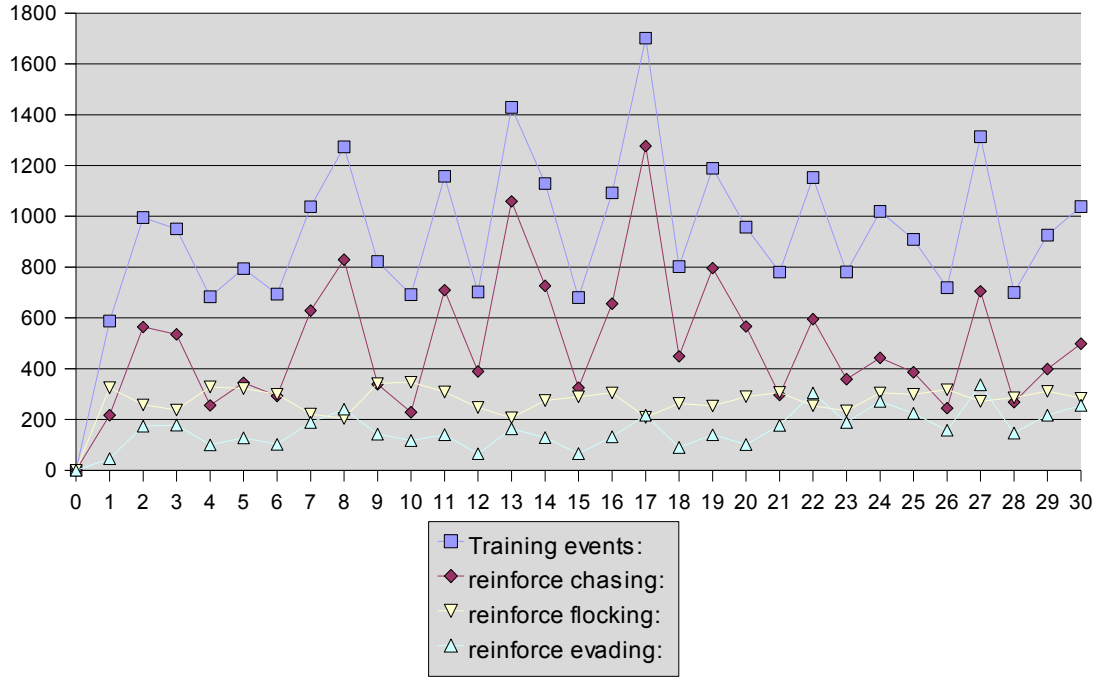

Figure 6.20 Training events, controlled supervised learning with softmax selection

6.9 Supervised learning with memory

So far the supervised learning only takes the last state of the AI agent into consideration when training the neural network. This leads to the following problem: The action taken at the current state might not be as decisive for the success of the agent as the actions taken some time before it. To avoid this a short term memory is implemented which allows the learning algorithm to train not only the current state but also its predecessors. In reinforcement learning this is also known as the delayed reward. A state in the past is rewarded based on the present time experience.

Implementing the memory

To implement this every agent stores a number of past states in the form of input variables to the neural network. When a training event is triggered by the death of the player or the agent, the neural network is not only retrained with the current inputs but also with the past inputs. This means that all of the stored states, defined by the inputs, are trained to result in the action prefered by the supervised learning.

Discounted memory

A modification to the first implementation of the memory based supervised learning is inspired by the concept of discounted reward in reinforcement learning. A past action is considered to be less important in achieving the result of the current action and therefore rewarded less. In the context of the supervised learning algorithm a similar effect can be achieved by lessening the training effect of the retraining of the neural network for the past inputs. This is simply achieved via weights which allow for a

greater error during the backpropagation training algorithm when it is run for the inputs associated with a state in the past. In practice this means less repetitions of the backpropagation training algorithm and therefore less weight changes.

Supervised learning with memory

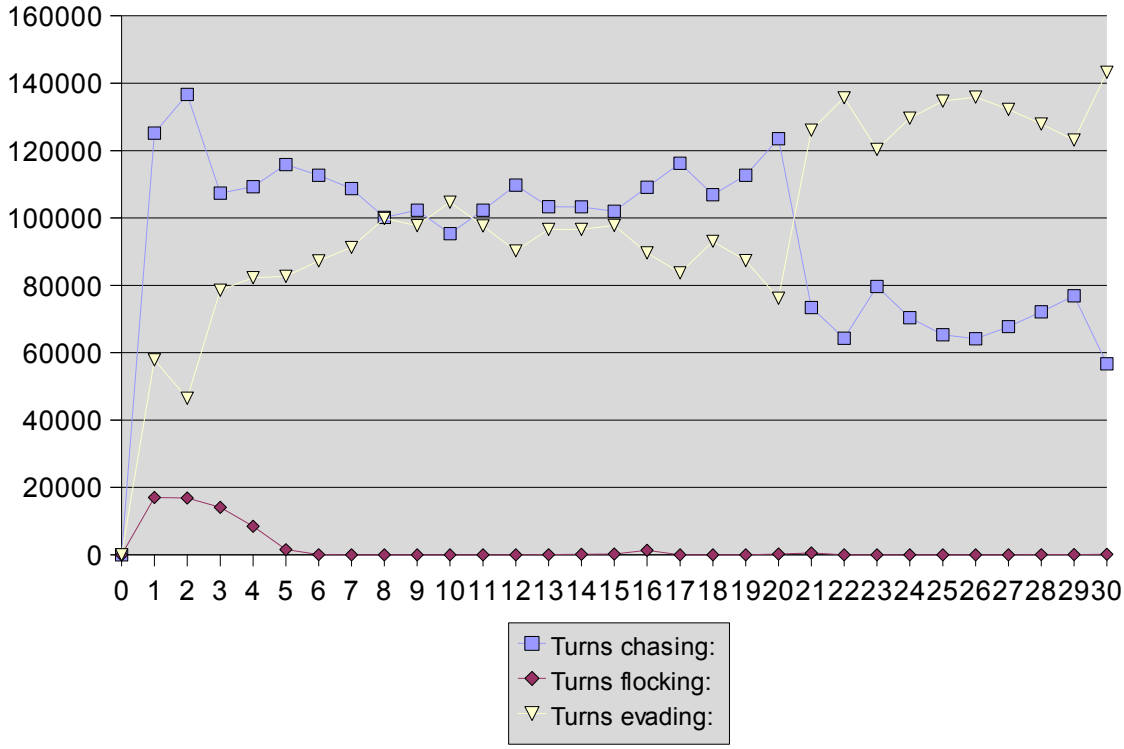

Figure 6.21 Turns per state distribution, supervised learning with memory

The supervised learning algorithm using a short term memory shows a very strong emphasis of the evade behaviour during all phases of the game. In the high damage segment it even becomes the single dominating behaviour, something it never came close to when any of the other learning algorithms were used. This is a result of the supervised learning imprinting evasion behaviour on agent states in the past when the normal behaviour would have been chasing. Agents are trained to evade too early when they are in a good position to attack and help nearby friendlies even if this could mean their own death. This overemphasis of their self preservation also leads to a very low death ratio. Similar to the basic supervised learning the flocking behaviour is completely suppressed.

Death ratio

The average death ratio of the network trained with supervised learning with memory is 0.45. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.53
- Low: 0.51
- High: 0.33

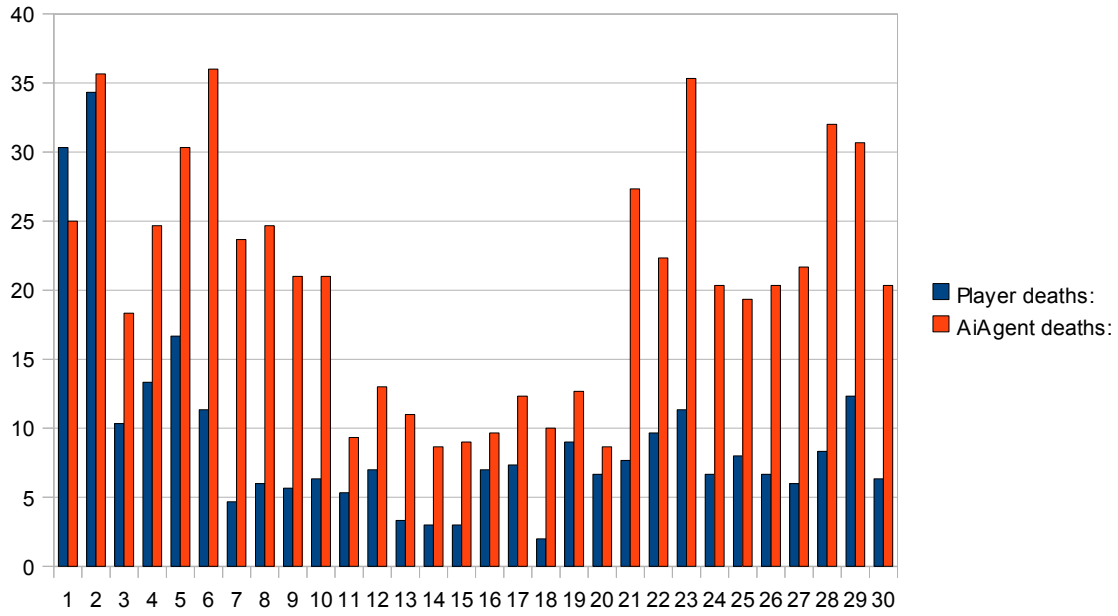

Figure 6.22 Death ratio, supervised learning with memory

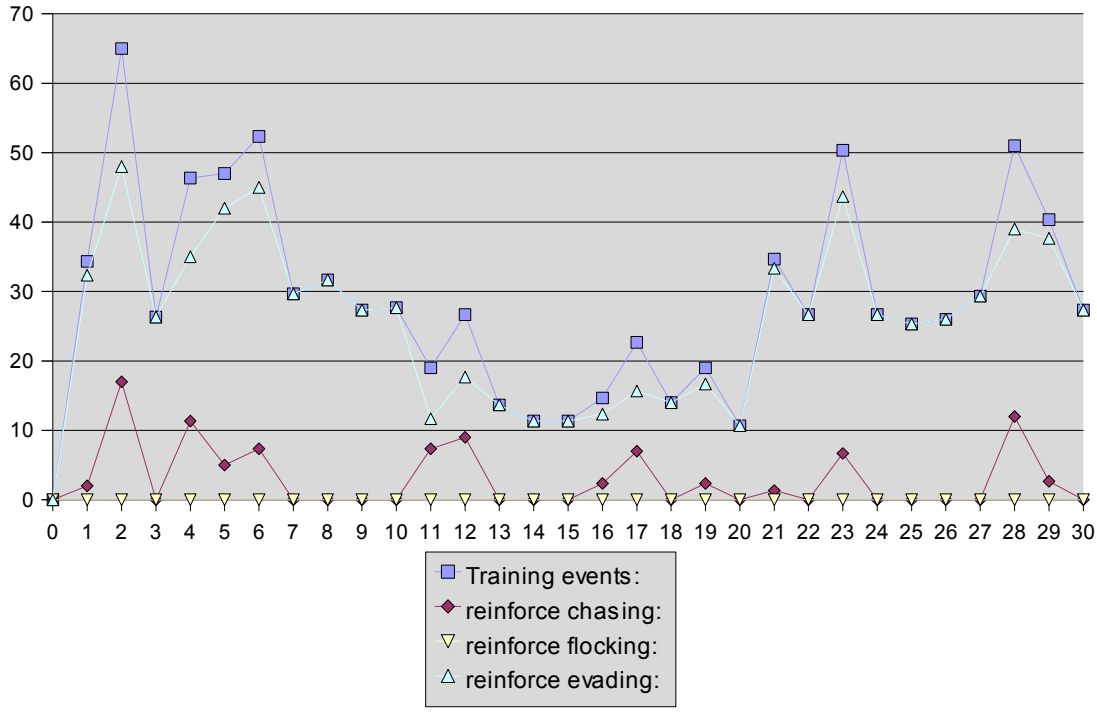

Figure 6.23 Training events, supervised learning with memory

Supervised learning with discounted memory

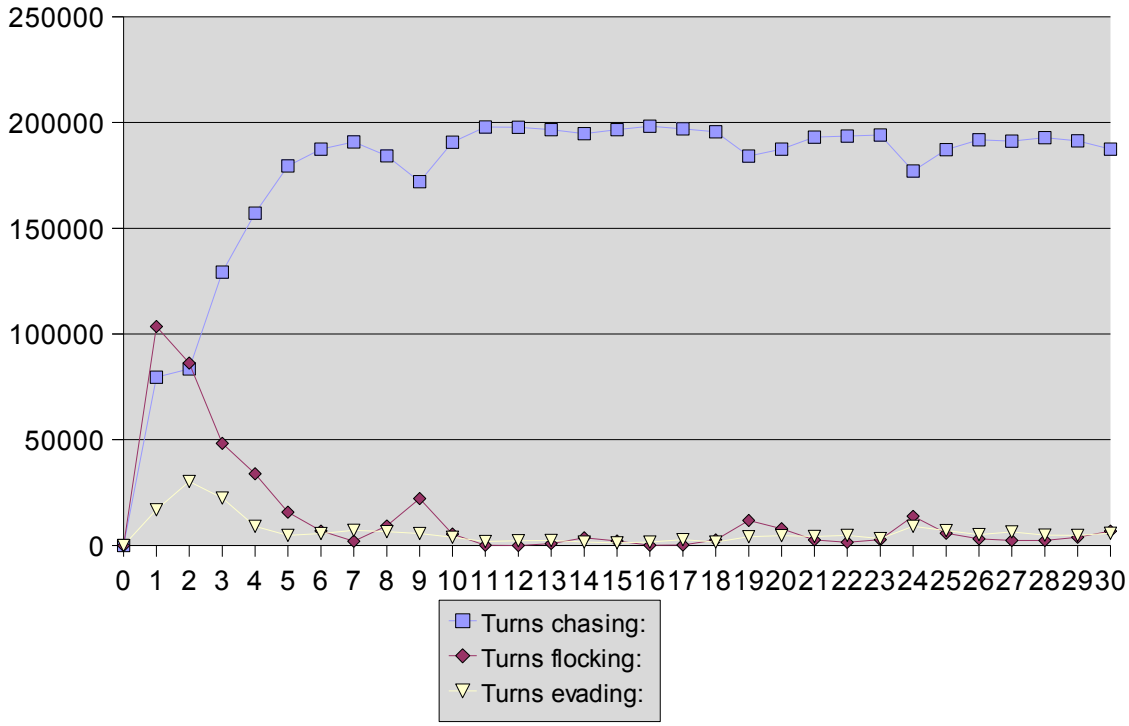

Figure 6.24 Turns per state distribution, supervised learning with discounted memory

The agent behaviour when using discounted behaviour is very similar to the normal supervised learning. Chasing dominates, flocking gets eliminated and evading is used very little. The discounted memory obviously has a much lower effect on the agents than the simple memory and the training events triggered by the player deaths are able to dominate the events triggered by the agent deaths.

Death ratio

The average death ratio of the network trained with supervised learning with discounted memory is 0.76. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.76
- Low: 1.32
- High: 0.46

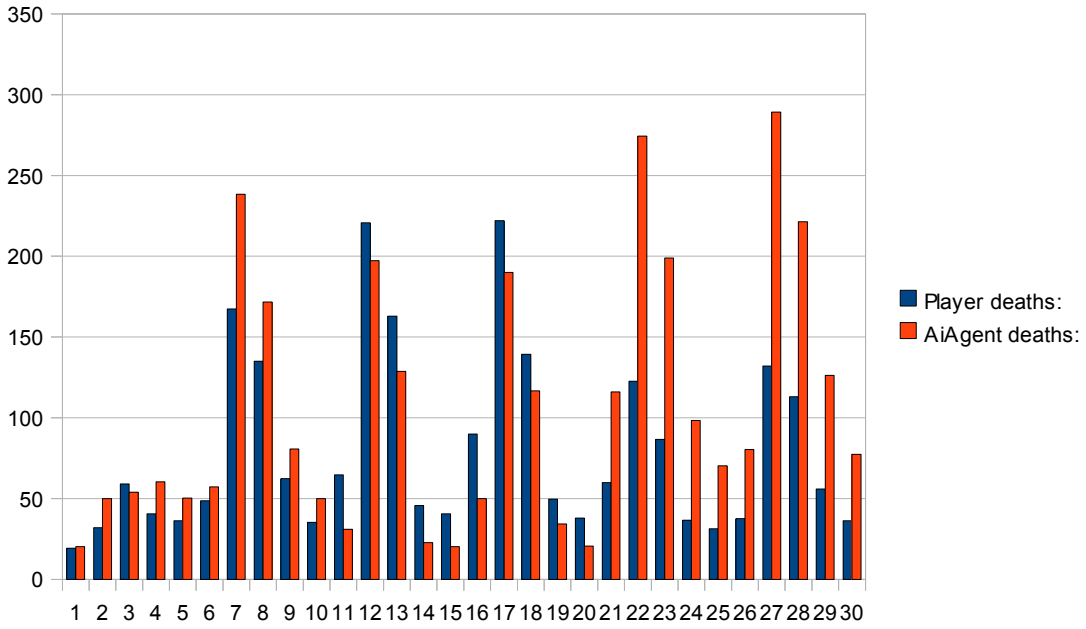

Figure 6.25 Death ratio, supervised learning with discounted memory

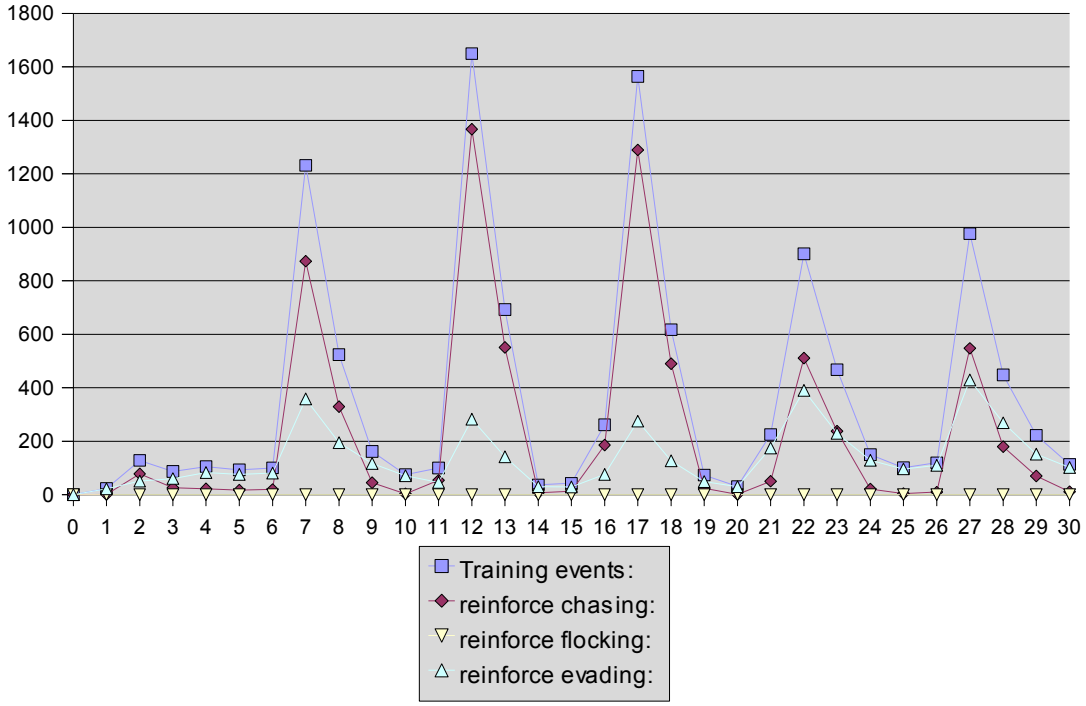

Figure 6.26 Training events, supervised learning with discounted memory

6.10 Comparison of the algorithms

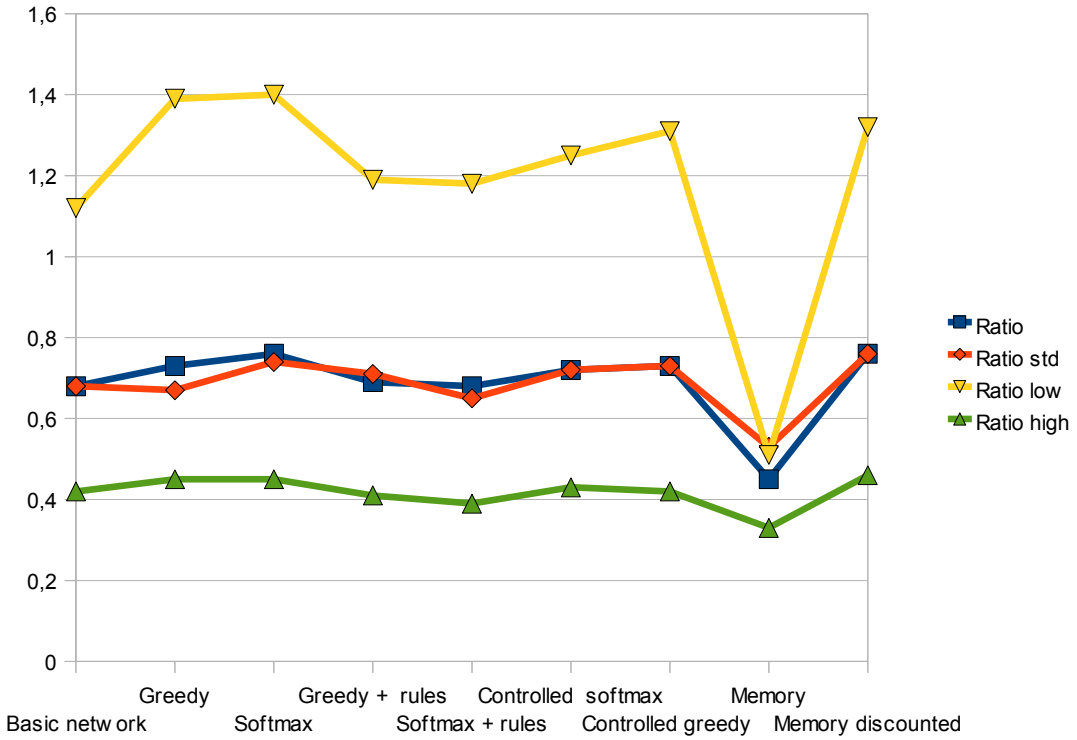

Figure 6.27 Death ratio comparison

Figure 6.27 shows the overall changes in death ratio and hence efficiency when compared to the basic neural network. It is remarkable that the highest absolute improvement is relatively low at 0.08 which corresponds to a 12% rise in efficiency for the average ratio. The biggest positive impact can be seen during the low damage segment. Here the top efficiency boost amounts to 25%. Both values are taken from the pure supervised learning algorithm which shows the best overall performance. The discounted memory algorithm comes very close while the undiscounted memory algorithm is obviously the worst one. It is the only one to worsen the performance of the basic network and very significantly so. This clearly shows the weakness of applying supervised learning to a state indirectly, only taking into account a followup state in its future. The controlled learning and it predecessor the supervised learning with rules show only a very minor improvement in efficiency, but also no deterioration.

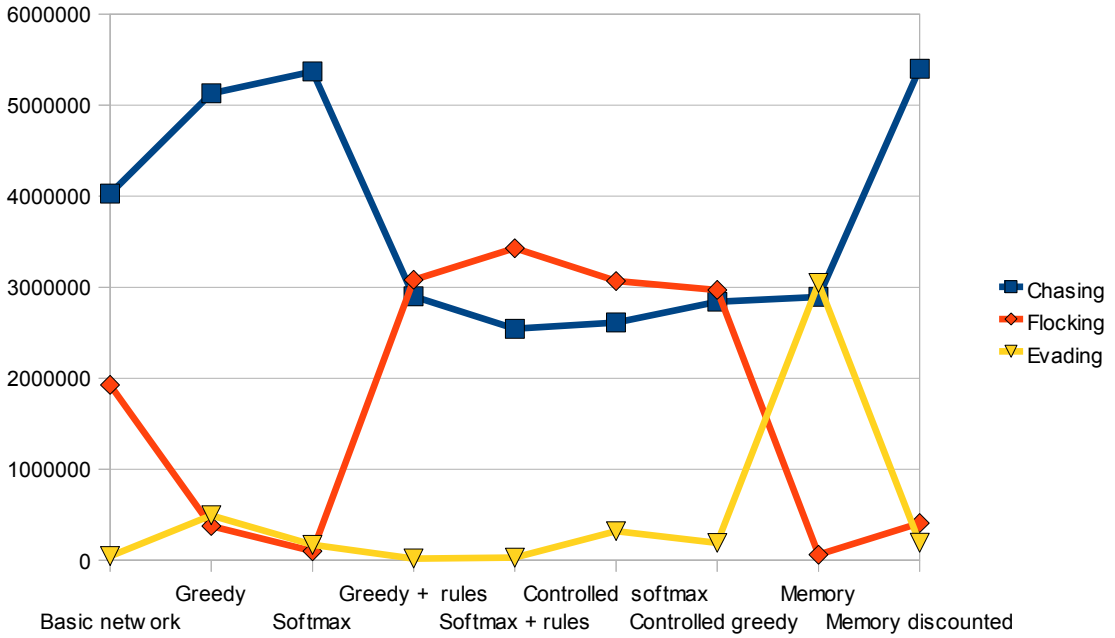

Figure 6.28 Sum of turns spent in a behaviour by all agents combined

Figure 6.28 Shows the overall behaviour of the algorithms. The pure supervised algorithms including the discounted memory algorithm show dominating chase behaviour with low evasion and flocking behaviour. As seen by the ratios this seems to be the most effective behaviour. The controlled and rules algorithms show a much higher flocking percentage on par with the chasing. It is interesting to note that although the behaviour seems to be near identical judging by the pure sums, the efficiency of the controlled learning is significantly higher than the one of the supervised learning with rules. The only notable difference between the two classes is the fact that the controlled learning has a higher evasion percentage. Although very small this seems to be important as the more efficient pure supervised learning algorithms also show a similar amount of evasion.

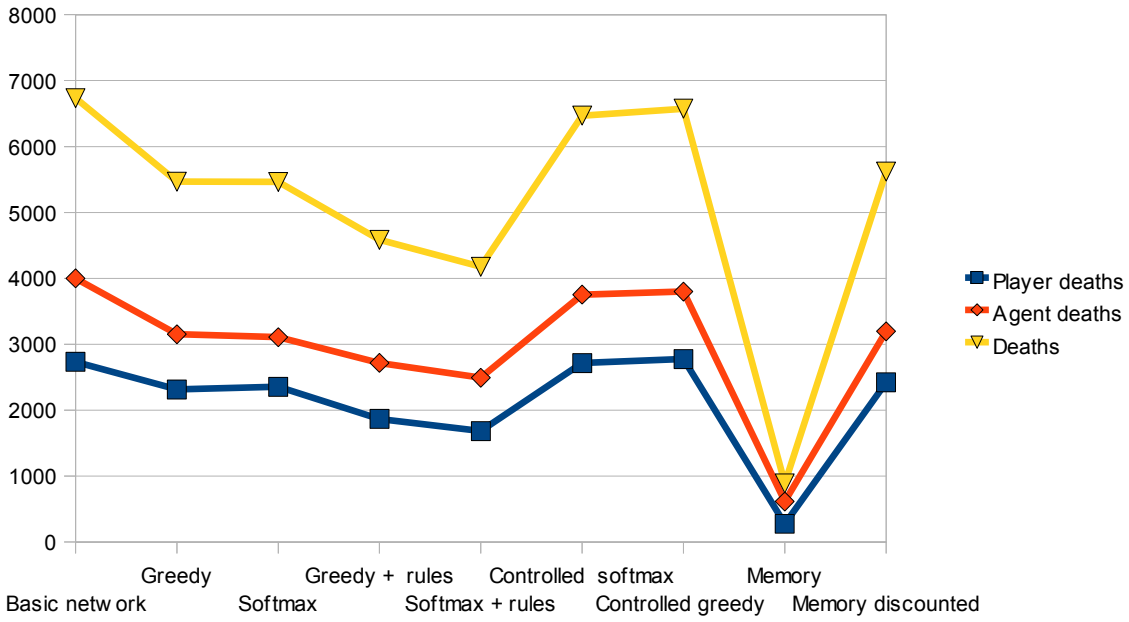

Figure 6.29 Total sum of player and agent deaths.

An interesting difference between the controlled learning and the supervised learning with added rules is the big gap in total death events during a game. Despite the near identical sum of turns spent exhibiting chasing and flocking the two different classes of algorithms obviously differ more than Figure 6.28 shows. When taking a look at the individual turns per state distribution in chapter 6.6 and 6.7 one can see that the controlled learning algorithm allows for a stronger variation in behaviour, the peak differences between chasing and flocking are higher and only sum up to the same average. The reason for this is the different frequency with which the rules were imprinted on the neural network. The supervised learning with rules used a higher frequency of 250 gamerounds while the controlled learning was only executed every 500 gamerounds. The supervised learning algorithms all show a very similar number of deaths with the obvious exception of the one with undiscounted memory.

6.11 Summary of chapter 6

The algorithms developed in chapter 5 have been tested more thoroughly and most of the observations about the AI agents behaviour have been confirmed. In addition several new aspects have been explored.

Supervised offline learning was successfully used to implement an AI for agents in a real time continuous environment using a neural network as basis.

Supervised online learning as proposed in chapters 5 and 6 was used to adapt the behaviour of AI agents to changes in their environment and also simply to change their behaviour from their predefined default AI routines.

Control rules which train the network parallel to the event driven learning were successfully used to steer the agent's behaviour into a wanted direction while still allowing for adaptation. The frequency of these control events can be used to adjust their influence on the overall learning process. The controlled learning algorithms both showed a balanced behaviour throughout the whole test.

It was statistically proven that the supervised learning changes and improves the behaviour although the gain in efficiency was smaller than hoped for.

Judging the AI by the criteria formulated in chapter 2 the following observations can be made:

Computational requirements:

- Speed: The tested algorithms are not tested for performance. They still show no signs of slowing down the game though. A short discussion of possible performance improvements will be given in chapter 9.
- Effectiveness: The controlled learning rules can be seen as a handwritten AI. Most of the implemented learning algorithms show better efficiency than it. It even has a negative impact on the overall performance when used in conjunction with one of the supervised learning algorithms.
- Robustness: The agents change the AI based on the ingame experience but some algorithms, like the basic supervised learning, tend to produce onesided behaviour after some time.
- Efficiency: The changes occur quite quickly, normally during the course of 4 to 5 timesteps, which corresponds to 3 to 4 minutes in real time if the rate of gamerounds is capped at about 30 rounds per seconds.

Functional requirements:

- Clarity: The short time changes in behaviour are easy to understand in most cases. Some of the more permanent long term changes are harder to understand at first glance.
- Variety: The agents visibly react to streaks of agent or player deaths and adapt their behaviour accordingly.
- Consistency: In all tests the time it took for the long term changes to take effect was quite similar. 5 to 6 timesteps for the supervised learning to fully convert the behaviour for example.
- Scalability: The change in damage rates made it easier for the agents to adapt. There is still a much bigger impact on efficency during the low damage segement than during the high damage segment though.

7. Reinforcement learning

Similar to chapter 6 several learning algorithms will be implemented and tested. All of them belong into the class of reinforcement learning as they directly reward or punish successful or failed actions. They do not propose a fixed alternative instead like the supervised algorithms but work on the basis of the current state of the AI agent at the moment the reward is handed out. The same neural network initialized with the same offline training as for the supervised algorithms is used to store the agent's action selection policy.

The following reinforcement learning algorithms are examined in this chapter:

- Complimentary reinforcement backpropagation algorithm (CRBP)
- Complimentary reinforcement backpropagation algorithm with discounted memory
- 0.5 rule reinforcement learning algorithm
- 0.5 rule reinforcement learning algorithm with memory

All of these are executed online.

7.1 Complimentary reinforcement backpropagation algorithm (CRBP)

The CRBP was first introduced by Ackley & Littman in 1990 **[AckleyLittman90]**. The main idea behind it is that whenever an action fails to generate reward, CRBP will try to generate a new action different to the old one **[Kaelbling et al. 96]**. This is achieved by training the neural network with a state action pair $<\underline{s},a>$ if the generated reward by a was 1 and by training it with the state action pair $<s,\underline{a}>$ if the reward was 0, where $\underline{a} = (1-a_1, 1-a_2, \ldots, 1-a_n)$. In the case of the Chase/Flock/Evade task the a_n correspond directly to the output values of the neural network.

The implementation used for the tests is shown in Figure 7.1.

The basic version was also extended with a short term memory in the same way as the supervised learning algorithms in chapter 6.

```
void CRBP(double distanceToPlayer)
{
        //if entity is alive
        if (agent.getHitpoints() >= 0.0)
        {
                //if entity was close while player died, reinforce current behaviour
                if (kill && (distanceToPlayer < CombatDistance))
                {
                        neuralNet.setInputs(agent.getInputs());
                        neuralNet.FeedForward();
                        neuralNet.Retrain(  neuralNet.GetOutput(0),
                                            neuralNet.GetOutput(1),
                                            neuralNet.GetOutput(2) );
                }
        }

        //if entity died, discourage current behaviour
        if (agent.getHitpoints() < 0.0)
        {
                neuralNet.setInputs(agent.getInputs());
                neuralNet.FeedForward();
                neuralNet.Retrain(  1.0 - neuralNet.GetOutput(0),
                                    1.0 - neuralNet.GetOutput(1),
                                    1.0 – neuralNet.GetOutput(2) );
        }
}
```

Figure 7.1 CRBP implementation

Basic CRBP

The basic CRBP shows a disproportional amount of flocking, a very low chasing and no evading. After an initial time of about 3 timesteps the algorithm has converged on the behaviour it will exhibit for the remainder of the test. There is no visible adaptation to the different player damage rates. As a result there are comparatively few player and agent deaths and the overall ratio is very low at 0.43.

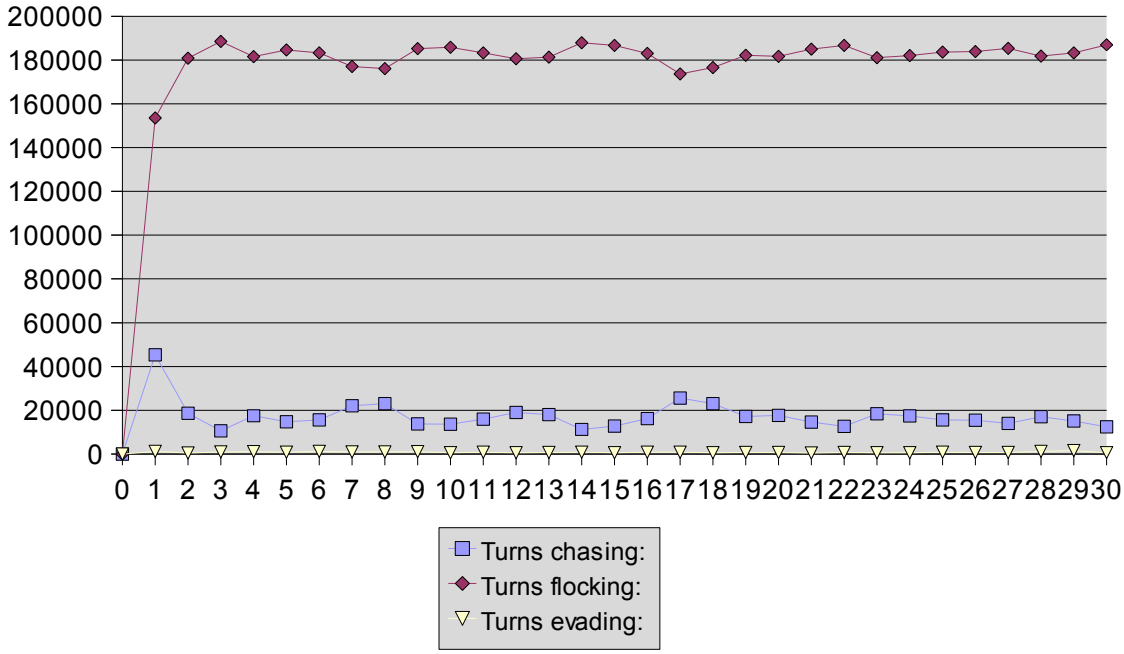

Figure 7.2 Turns per state distribution CRBP

Death ratio

The average death ratio of the network trained with CRBP is 0.43. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.48
- Low: 0.89
- High: 0.21

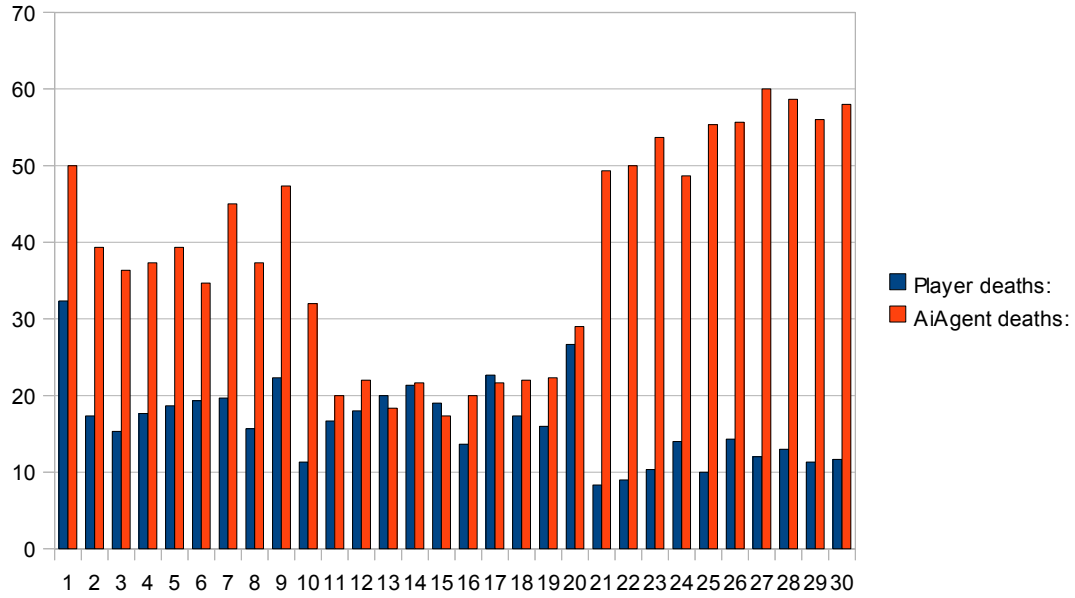

Figure 7.3 Death ratio CRBP

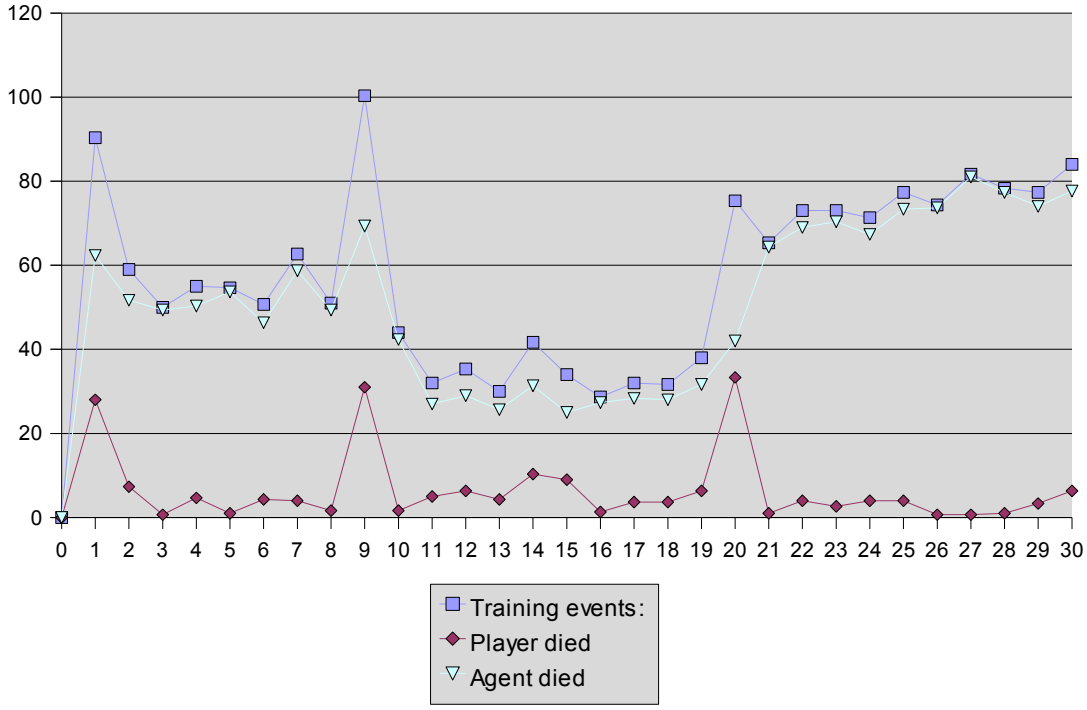

Figure 7.4 Training events CRBP

CRBP discounted memory

The version of the CRBP with discounted memory shows even less chasing. The memory amplifies the learning process already observed in its basic form.

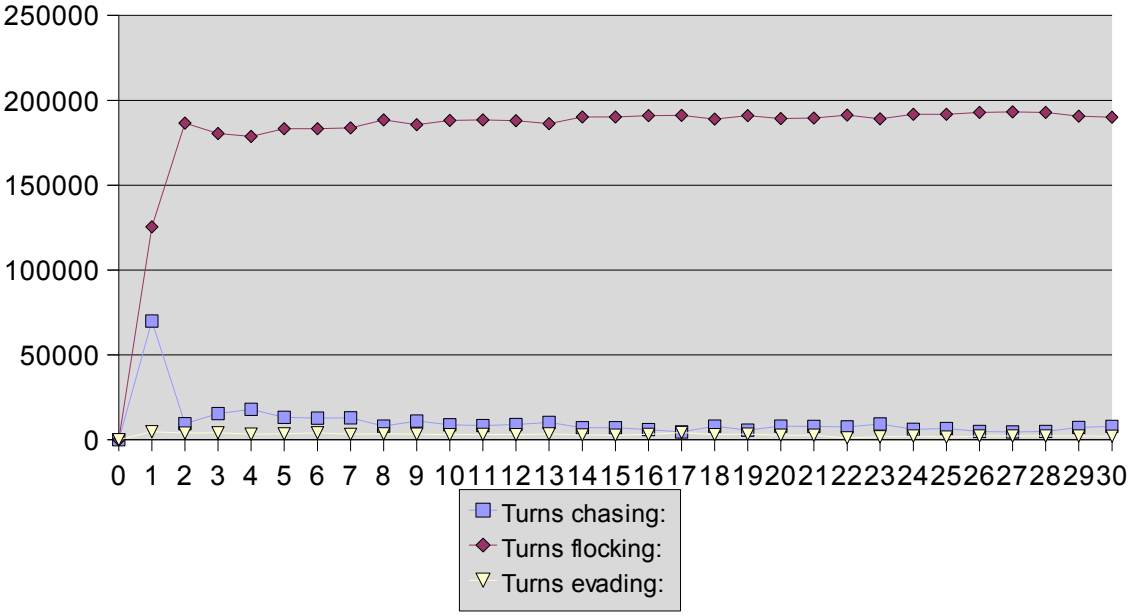

Figure 7.5 Turns per state distribution CRBP discounted memory

Death ratio

The average death ratio of the network trained with CRBP using discounted memory is 0.35. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.41
- Low: 0.76
- High: 0.16

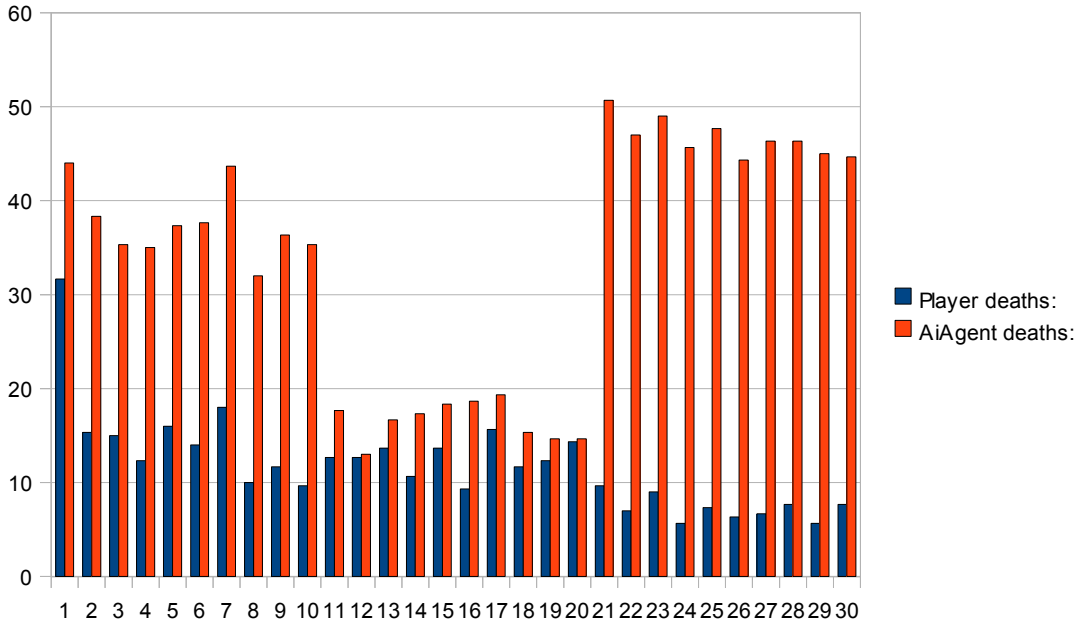

Figure 7.6 Death ratio CRBP discounted memory

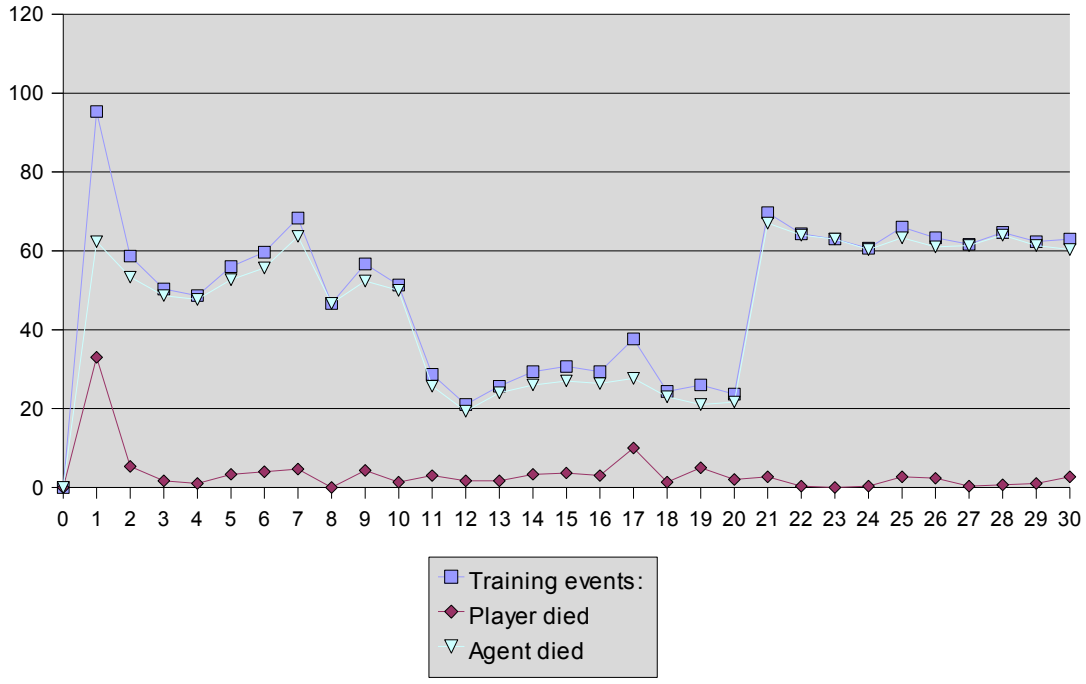

Figure 7.7 Training events CRBP discounted memory

7.2 0.5 rule reinforcement learning algorithm

Two main weaknesses were obvious in the behaviour produced by the CRBP. The domination of flocking and the near absence of chasing and evading. By closer examining the training rules and watching the agent behaviour this could be attributed to the following problems. Evading agents often were instrumental in damaging the player before they started to evade. Those agents then often died while evading because they could not escape out of the combat distance quickly enough. In such a case The earlier chase and the evade will be weakened. Similarly chasing gets punished when an agent dies while fighting the player not having the chance to escape. The CRBP will incorrectly punish the chase behaviour in this case and strengthen flocking and evading in this kind of situation. As a result of this flocking gets indirectly promoted on many occasions while the other two behaviours are rewarded in very few situations.

To lessen this problem the reinforcement learning algorithm was modified. Firstly the reward for flocking behaviour was deactivated as flocking agents can be considered to only accidentally attack the player. After a few tests it became obvious that this alone was not enough, so secondly the punishment of behaviours was weakened and an additional punishment for the flocking behaviour was introduced. This was done by using a weight of 0.5 on the outputs which are used as the inputs for the training of the neural network. The current behaviour which shall get punished is weighted with 0.5 while the other behaviours are fed into the retraining unweighted. Flocking is always weighted with 0.5 and therefore weakened, even if it wasn't the actual behaviour exhibited by the agent.

This new reinforcement learning algorithm was dubbed the 0.5 rule reinforcement algorithm and then tested with several selection strategies, with control rules and with short term memory.

The 0.5 Rule implementation:

```
void ReinforcementLearning(distanceToPlayer)
{
        //if entity is alive
        if ( (activeEntity.getHitpoints() >= 0.0))
        {
                //if entity was close while player died, reinforce current behaviour
                if (kill && (distance < CombatDistance))
                {
                        neuralNet.setInputs(agent.getInputs);
                        neuralNet.FeedForward();
                        behaviour = neuralNet.getBehaviour();
                        switch (behaviour)
                        {
                                case chasing:
                                neuralNet.Retrain(neuralNet.GetOutput(0),
                                                neuralNet.GetOutput(1),
                                                neuralNet.GetOutput(2));
                                break;
                                case flocking:
                                //skip training
                                break;
                                case evading:
                                neuralNet.Retrain(neuralNet.GetOutput(0),
                                                neuralNet.GetOutput(1),
                                                neuralNet.GetOutput(2));
                        break;
                        }
                }
        }
```

```
//if entity died, discourage behaviour
if (activeEntity.getHitpoints() < 0.0)
{
        neuralNet.setInputs(agent.getInputs);
        neuralNet.FeedForward();
        behaviour = neuralNet.getBehaviour();
        switch (behaviour)
        {
                case chasing:
                neuralNet.Retrain(0.5*neuralNet.GetOutput(0),
                                  0.5*neuralNet.GetOutput(1),
                                  neuralNet.GetOutput(2));
                break;
                case flocking:
                neuralNet.Retrain(neuralNet.GetOutput(0),
                                  0.5*neuralNet.GetOutput(1),
                                  neuralNet.GetOutput(2));
                break;
                case evading:
                neuralNet.Retrain(neuralNet.GetOutput(0),
                                  0.5*neuralNet.GetOutput(1),
                                  0.5*neuralNet.GetOutput(2));
                break;
        }
}
}
```

Figure 7.8 0.5 rule reinforcement learning algorithm

0.5 rule reinforcement learning algorithm, greedy selection

The new algorithm showed significantly improved behaviour. Flocking is still the overall strongest state exhibitied by the agents but chasing is back in an influental quantity, more than half as much as flocking. Evading plays a minor role as in most previous cases but is very perceivable and constant.

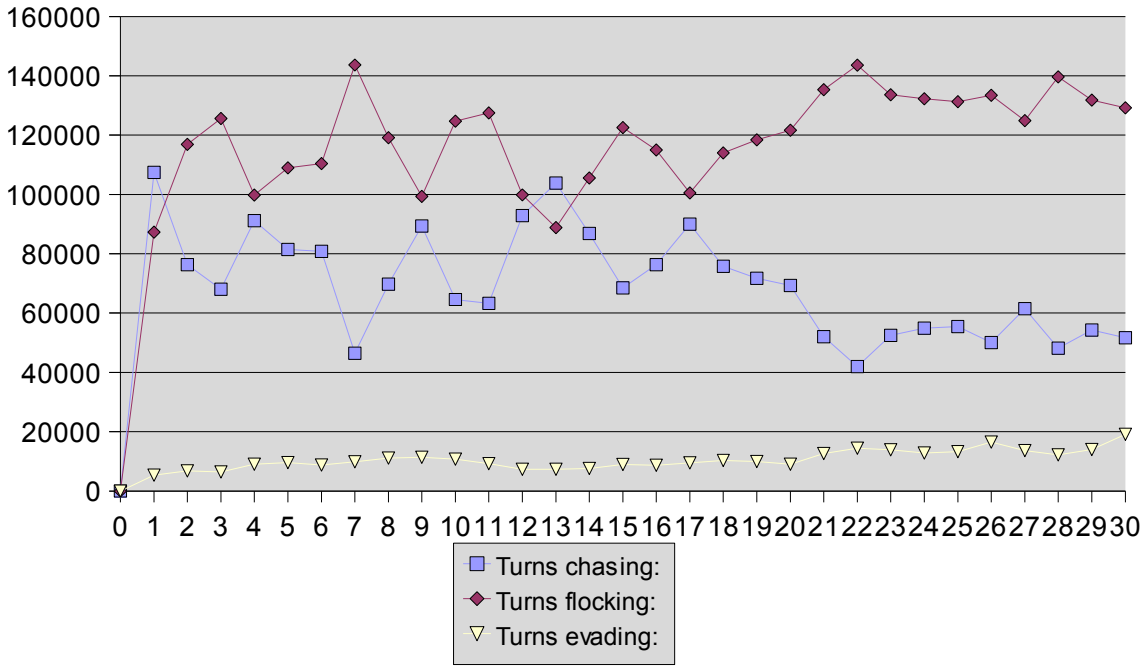

Figure 7.9 Turns per state distribution 0.5 rule with greedy selection

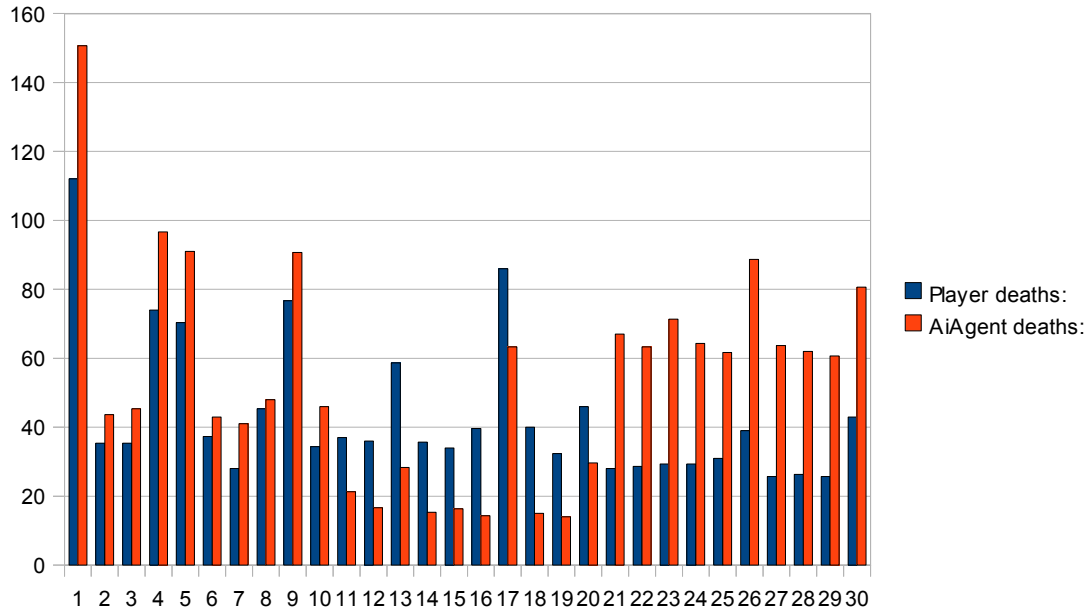

Figure 7.10 Death ratio 0.5 rule with greedy selection

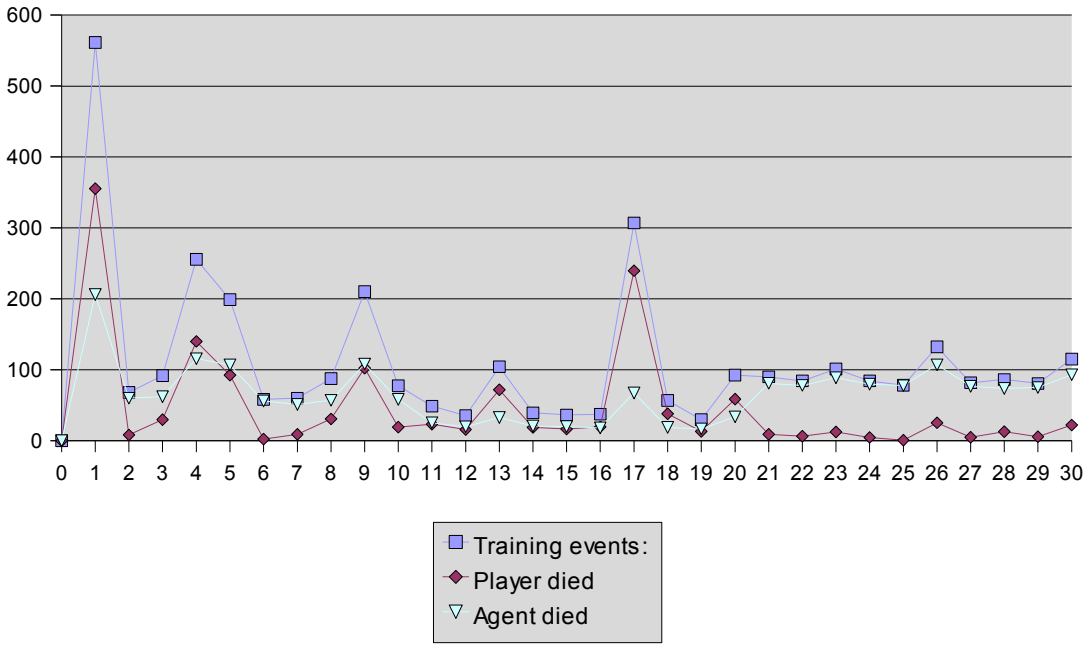

Figure 7.11 Training events 0.5 rule with greedy selection

Death ratio

The newly improved behaviour clearly shows in a very good death ratio. Not only is it higher than the one by the CRBP but it also surpasses the best ratio of the supervised learning algorithms examined in chapter 6.

The average death ratio of the network trained with the 0.5 rule algorithm and greedy selection is 0.81. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.79
- Low: 1.9
- High: 0.45

0.5 rule reinforcement learning algorithm, softmax selection

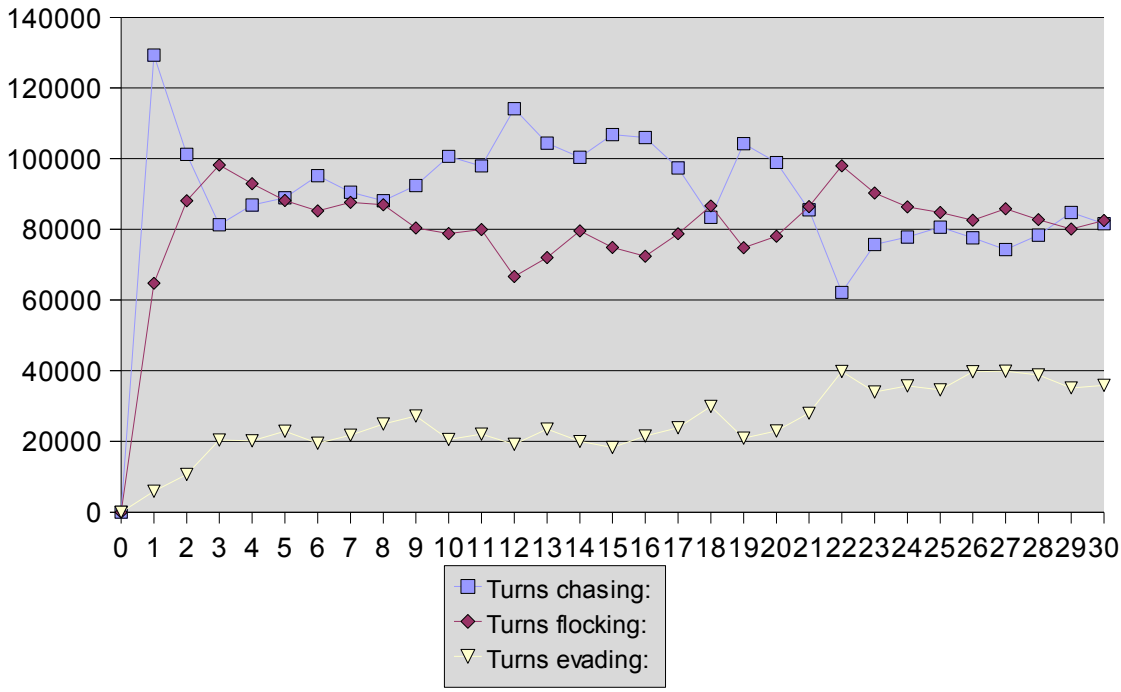

Figure 7.12 Turns per state distribution 0.5 rule with softmax selection

The softmax selection strategy again lowers the gap bewteen the different behaviours. Especially evading is used much more compared to the greedy action selection strategy.

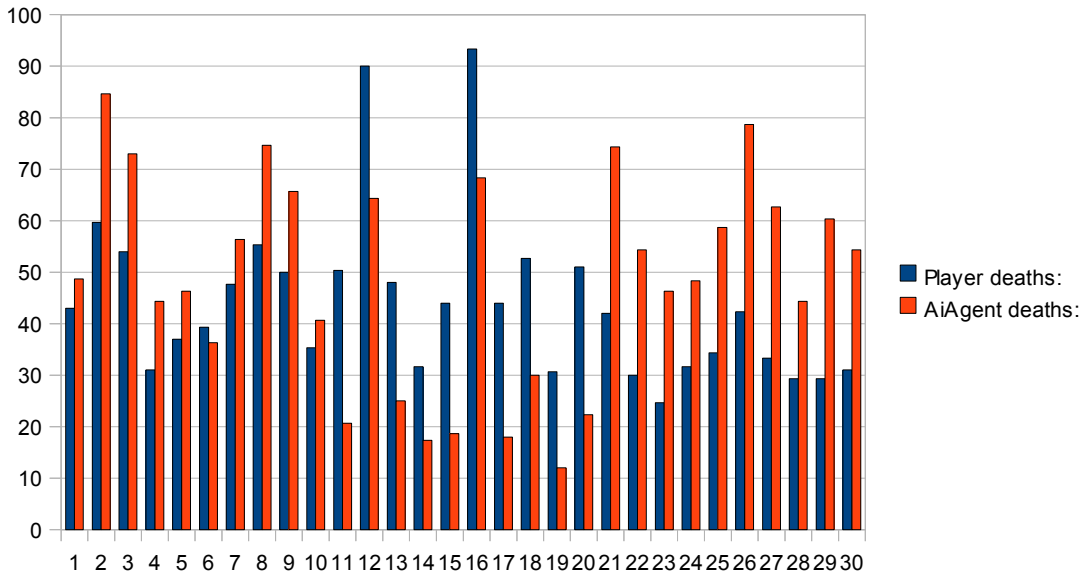

Figure 7.13 Death ratio 0.5 rule with softmax selection

The softmax selection is also noticeably more efficient as can be seen by the higher death ratio.

Death ratio

The average death ratio of the network trained with the 0.5 rule algorithm and softmax selection is 0.91. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.79
- Low: 1.81
- High: 0.56

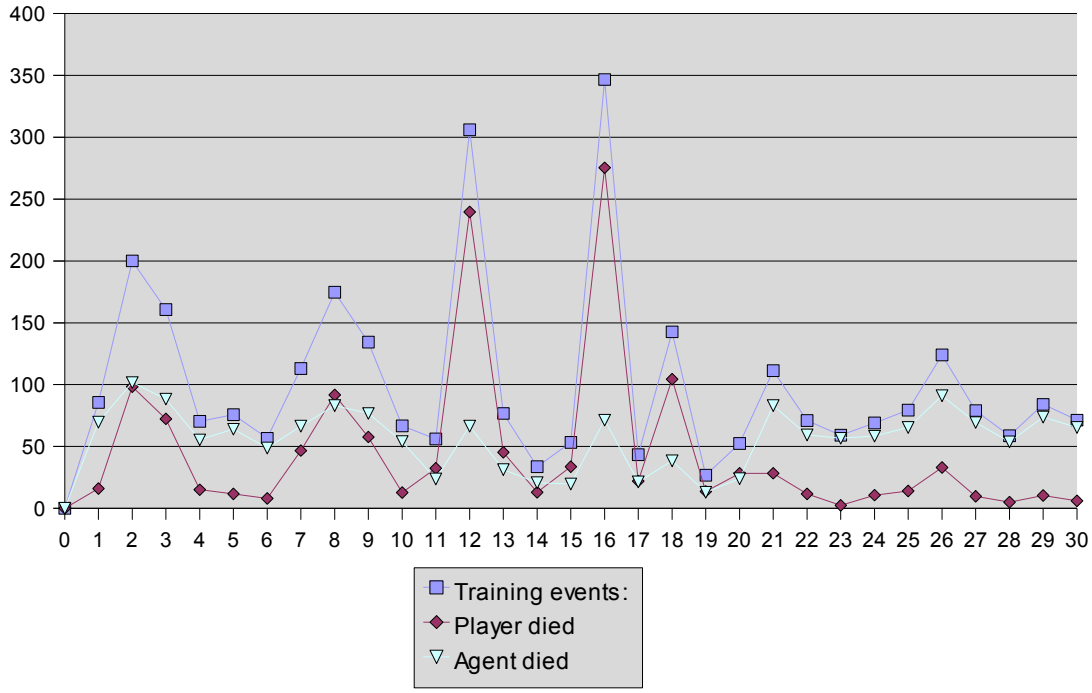

Figure 7.14 Training events 0.5 rule with softmax selection

Controlled 0.5 rule reinforcement learning algorithm

The controlled learning rules strongly dampens the reinforcement learnings efficiency boost and overall influence on the agents. Figure 7.15 shows the typical swinging between the two dominating behaviours. The absolute player and agent death numbers are also higher than the respective numbers for the uncontrolled algorithms. An effect the controlled learning rules also had on the supervised learing.

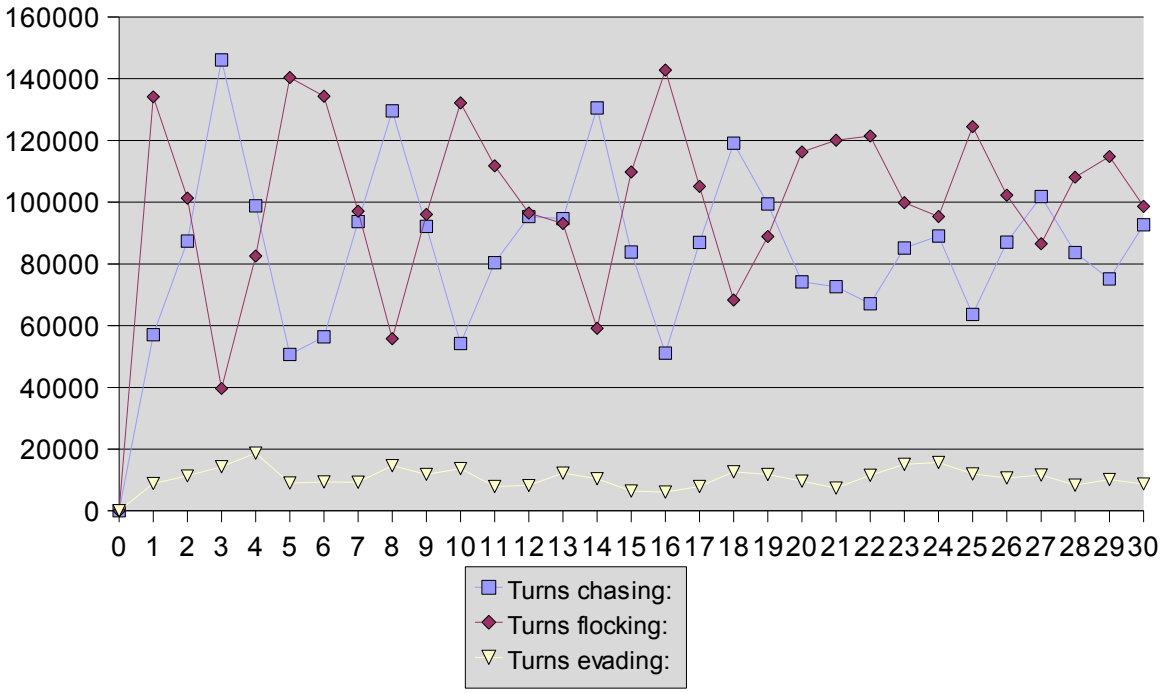

Figure 7.15 Turns per state distribution controlled 0.5 rule with softmax selection

Death ratio

The average death ratio of the network trained with the controlled 0.5 rule algorithm and softmax selection is 0.77. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.73
- Low: 1.59
- High: 0.45

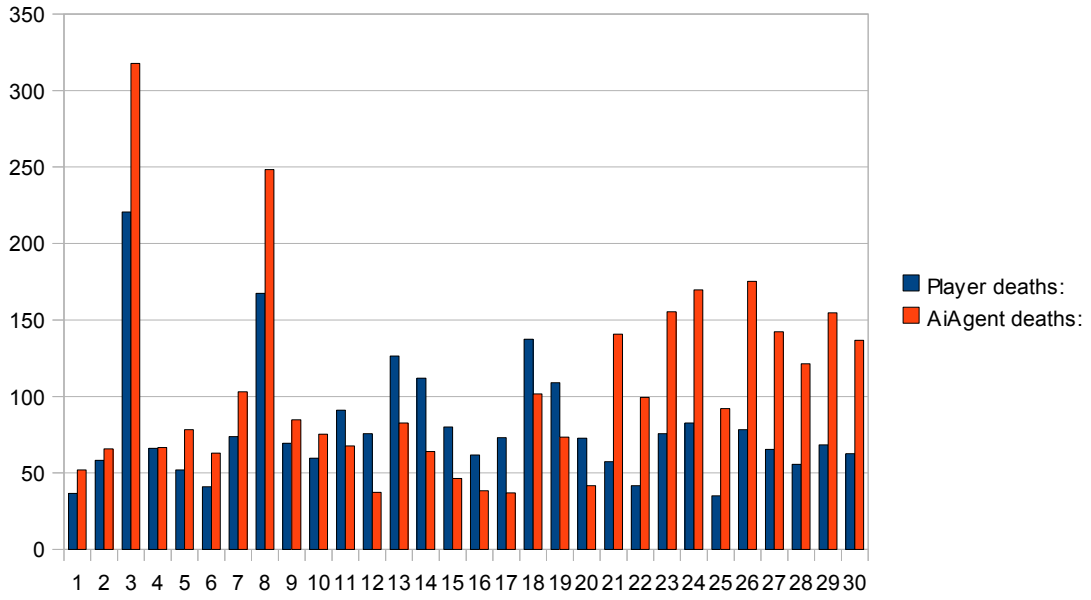

Figure 7.16 Death ratio controlled 0.5 rule with softmax selection

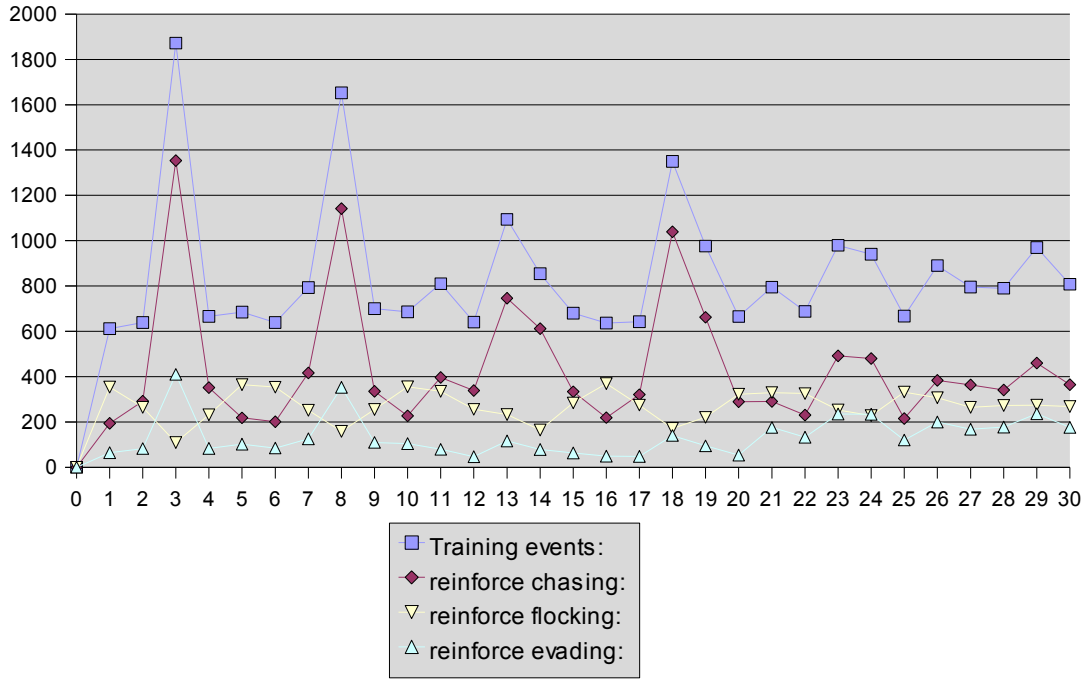

Figure 7.17 Training events controlled 0.5 rule with softmax selection

7.3 0.5 rule reinforcement learning algorithm with memory

The memory was implemented similarly to the one for the supervised learning. The only major difference was the fact that the retraining of the neural network has to recreate the old outputs belonging to the saved past inputs while the supervised algorithm could simply take the target value from the learning algorithm to calculate the errors.

0.5 rule reinforcement learning algorithm with memory

The simple short term memory strongly enhances the effect of the learning algorithm. The evasion behaviour is boosted much stronger and is able to even overtake the flocking behaviour, during the high player damage segment it even comes close to top the chase behaviour. The overall effect on the efficiency and variation of the agent's AI is quite negative. The death ratio is significantly lower than the version of the algorithm without the memory and the behaviour more monotonous.

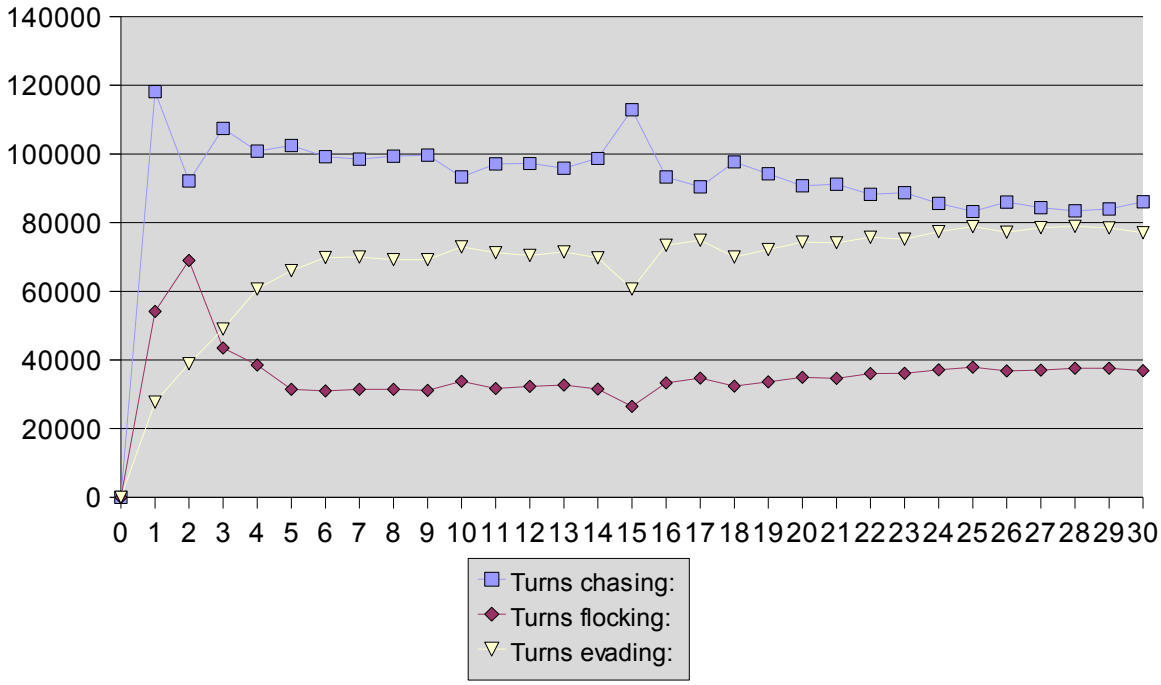

Figure 7.18 Turns per state distribution 0.5 rule with softmax selection and memory

Death ratio

The average death ratio of the network trained with the 0.5 rule algorithm and memory is 0.51. The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.68
- Low: 0.87
- High: 0.28

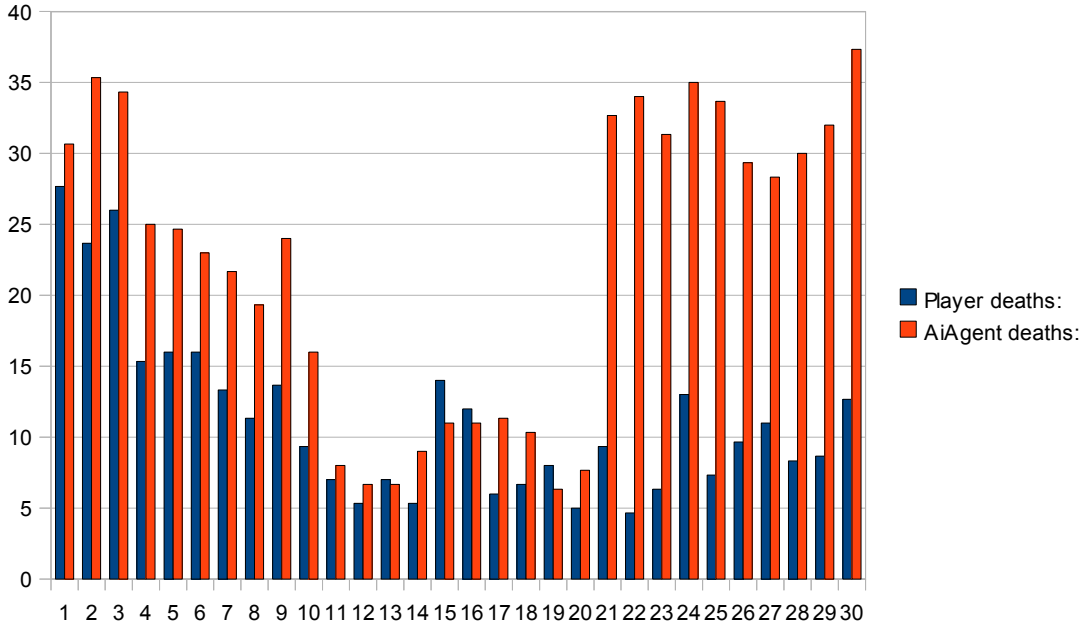

Figure 7.19 Death ratio 0.5 rule with softmax selection and memory

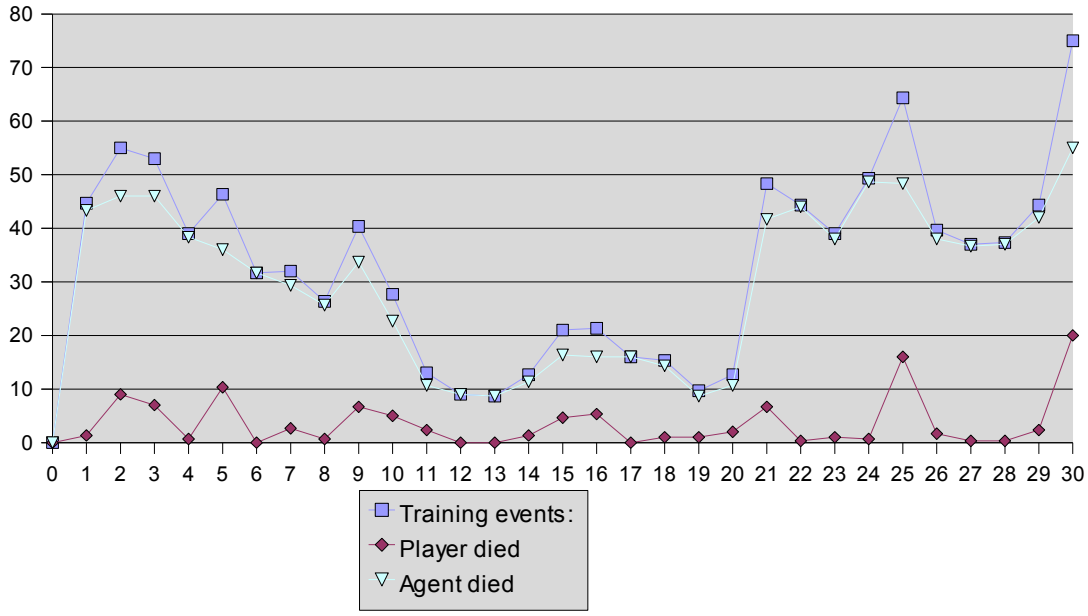

Figure 7.20 Training events 0.5 rule with softmax selection and memory

Controlled 0.5 rule reinforcement learning algorithm with discounted memory

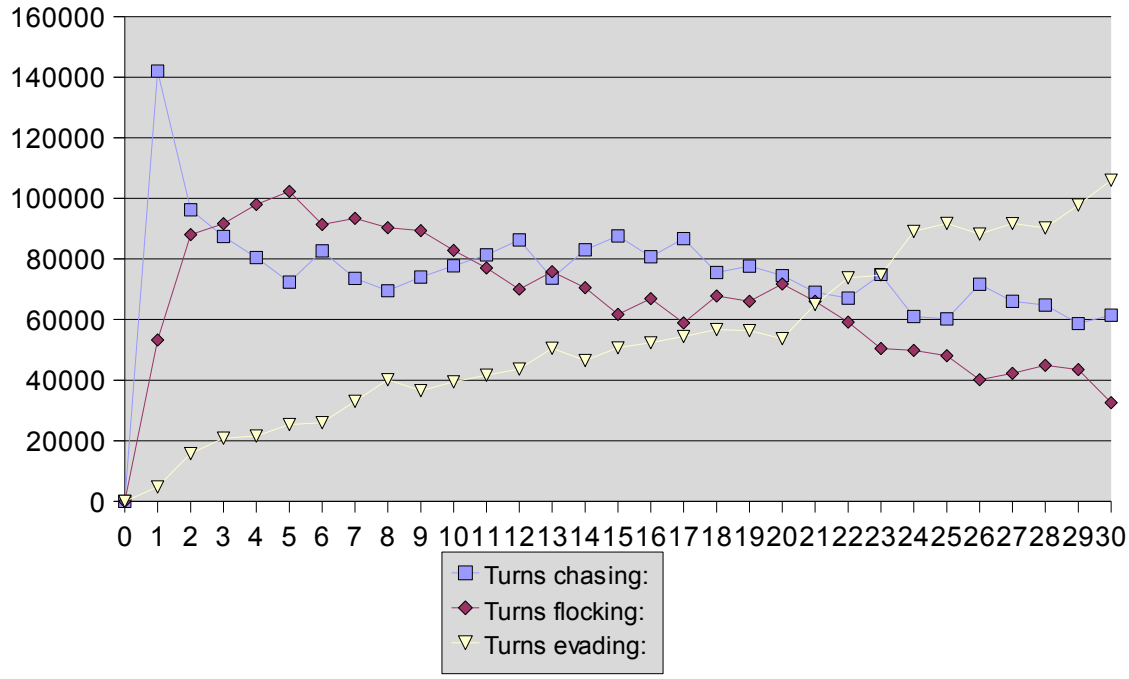

Figure 7.21 Turns per state distribution, Softmax selection and discounted memory

The discounted memory has a much more moderate impact on the learning. The overall curve in Figure 7.21 looks more akin to Figure 7.12 the behaviour from the memoryless reinforcement learning, just with boosted ascents and descents. Interesting to note is also the leap of the evasion behaviour to dominate in the high damage segement.

As a result of this much improved overall behaviour, the efficency is the highest of all the algorithms tested so far.

Death ratio

The average death ratio of the network trained with the 0.5 rule algorithm and discounted memory is 1.09 The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 1.06
- Low: 2.87
- High: 0.65

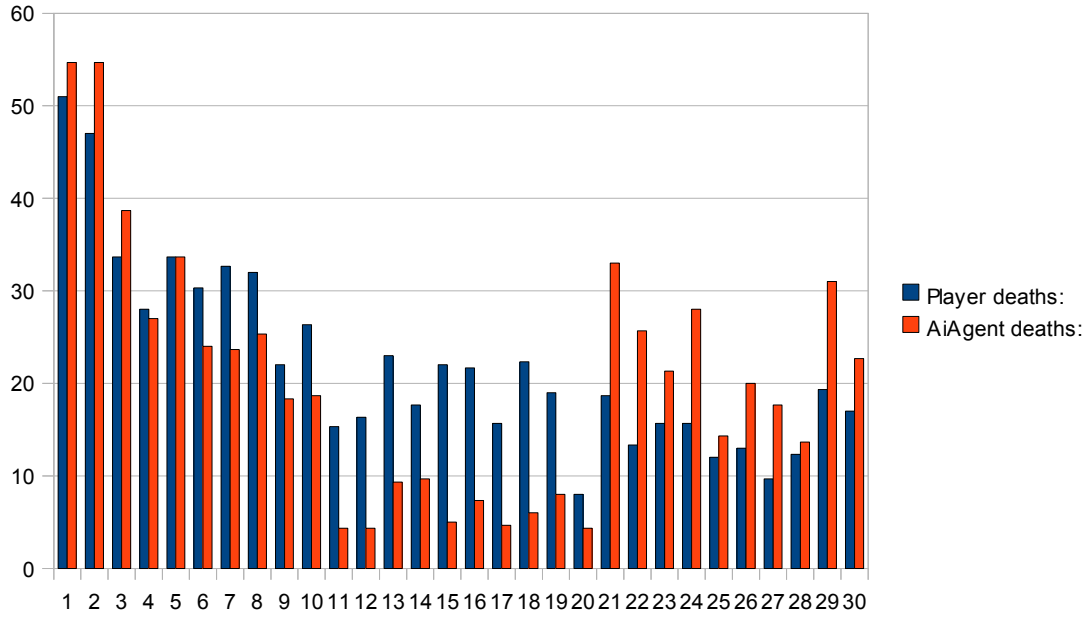

Figure 7.22 Death ratio 0.5 rule with softmax selection and discounted memory

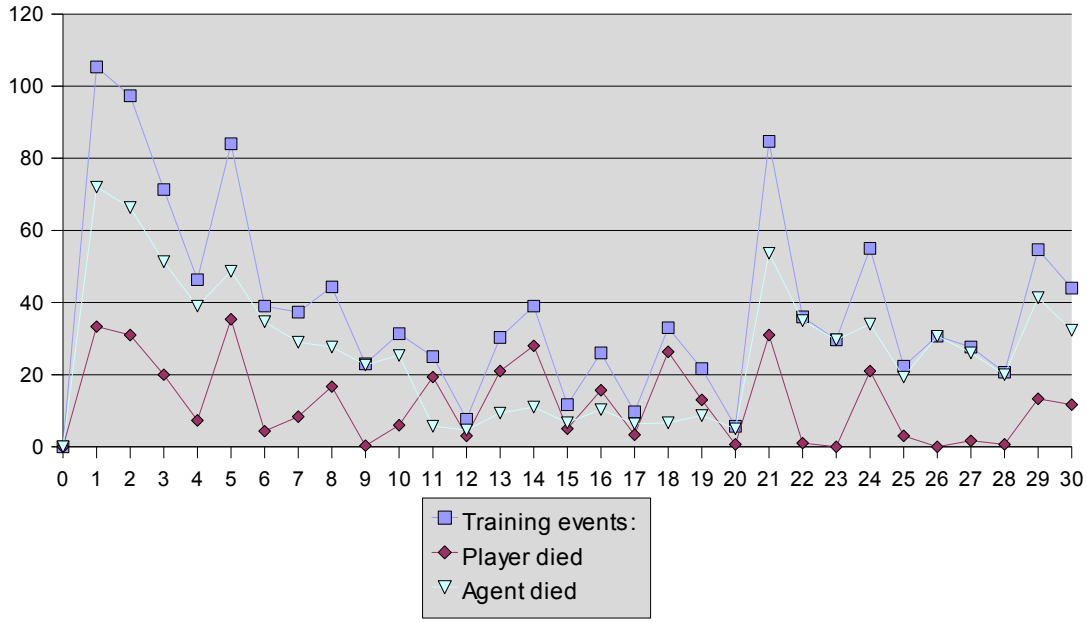

Figure 7.23 Training events 0.5 rule with softmax selection and discounted memory

7.4 Summary of chapter 7

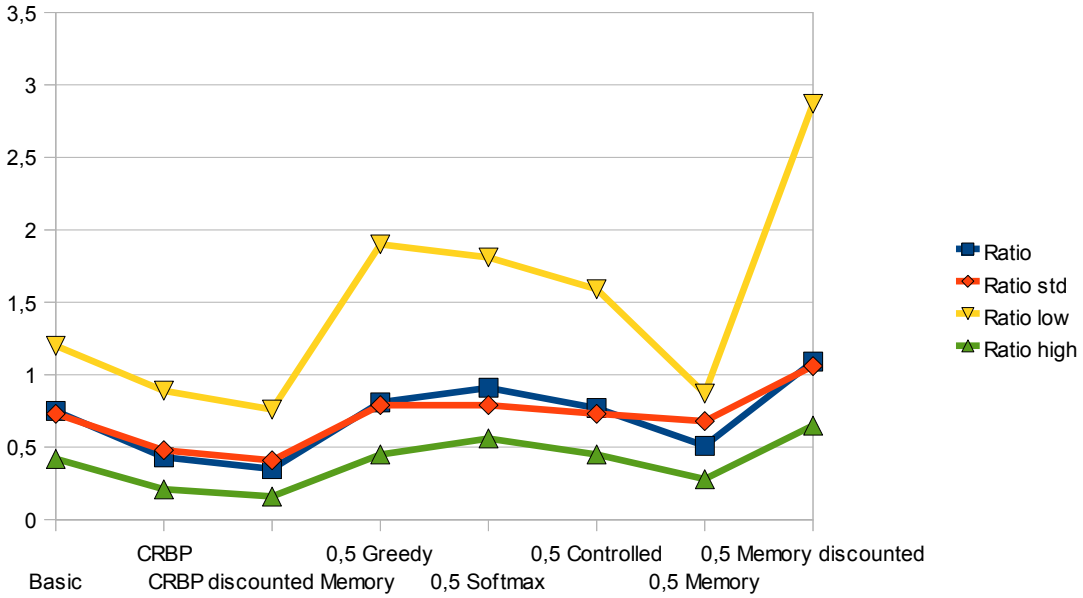

Figure 7.24 Death ratio comparison

The CRBP algorithm was a big dissapointment. It led to the worst efficiency and also to very monotonous, inflexible behaviour. The opposite of what was intended. Obviously it is not simply possible to transfer a reinforcement learning algorithm to a similar, yet different task.

The specially tailored algorithm based on the CRBP on the other hand is the most efficient of the algorithms examined in chapters 6 and 7 and also exhibits interesting and varied behaviour, a very important aspect of gaming AI. Figure 7.25 shows a near identical amount of turns spent flocking and chasing and also a noticeable amount of evasion behaviour. A slight surprise is the big gap between the algorithms with discounted and undiscounted memory. The same disparity could be observed in chapter 6 and hints at either a big superiority of the concept of the discounted memory or a weakness in the implementation of the simple memory.

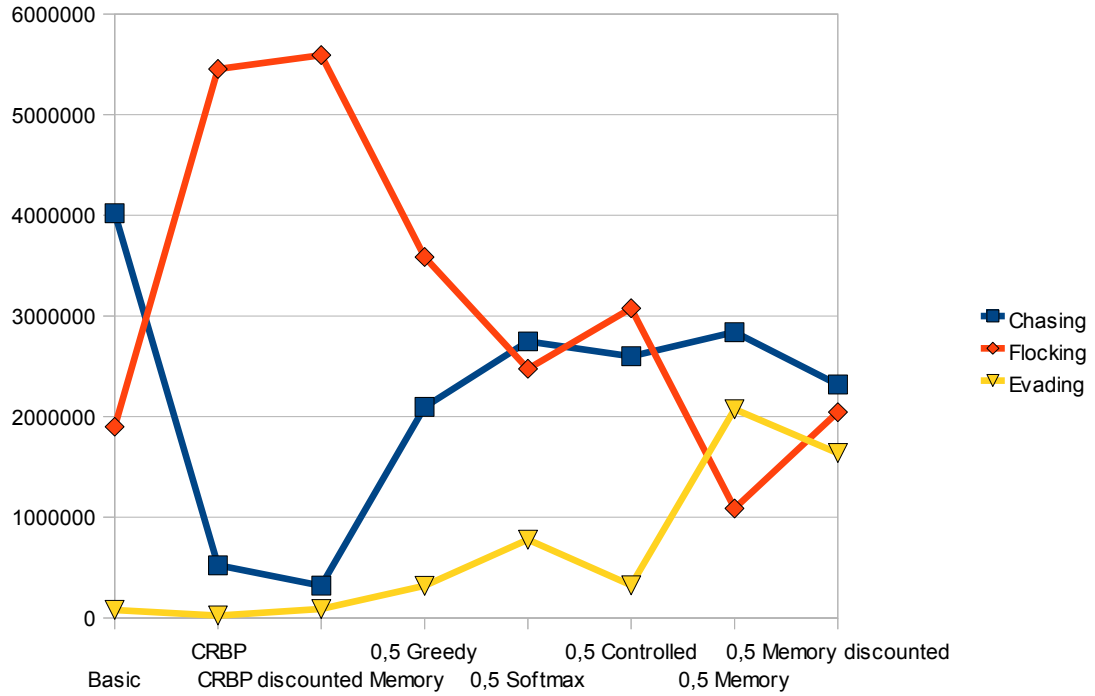

Figure 7.25 Sum of turns spent in a behaviour by all agents combined

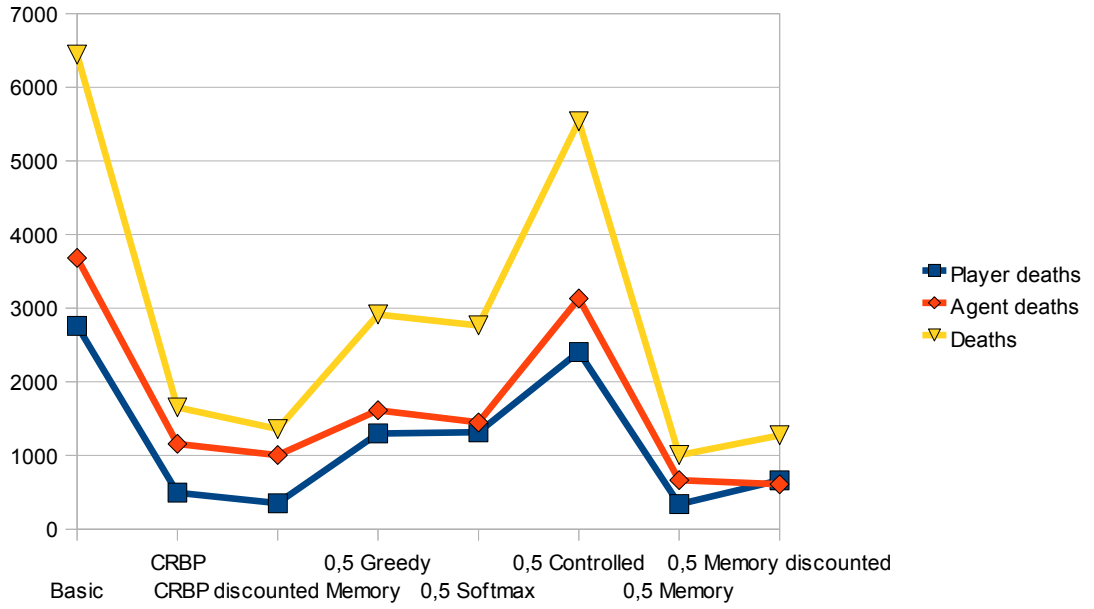

Figure 7.26 Sum of player and agent deaths

Comparison to the supervised learning algorithms.

The reinforcement learning led to superior efficency and also to a more varied behaviour. It made bigger changes to the basic neural network which can also be seen in the much lower number of overall player and agent deaths in Figure 7.26. compared to Figure 6.29. It has the potential to be clearly better. On the other hand there also was a bigger variance in the results from the tests which do not show in the statistics which are averaged from a big number of samples. It takes more work and testing to ensure proper operation of a reinforcement learning algorithm. The 0.5 rule algorithm also had to be very specifically modified to suit the Chase/Flock/Evade task while the supervised learning algorithms used a much simpler pattern.

Summary of chapter 7

Chapter 7 introduced reinforcement learning to create adaptive agent AI on basis of a neural network. The same goals as in chapter 6 were achieved. The new algorithms are more efficient but also potentially more unpredictable and harder to implement and test.

8. Variations of the network

Besides supervised and reinforcement learning operating on the single small neural network, two other network structures were examined. Firstly a bigger network was implemented using more inputs and hidden nodes. Secondly a combination of two small networks was tested. The goal was to see if these modification would have a positive impact on the overall behaviour and effectiveness and thereby judge the suitability of the original, small network for the solution of the Chase/Flock/Evade task. The tests with the single, bigger network were executed early during the process of designing the learning algorithms and used to decide if the small network was sufficient to solve the Chase/Flock/Evade task satisfyingly. Therefore no reinforcement learning algorithm was implemented for it. The two network structure was examined to show alternatives to the small network and to test the viability of the developed techniques when dealing with multiple neural networks.

8.1 Big network

The new network has the same three layer structure like the old one. It uses six inputs, four outputs and five hidden layer nodes.

The inputs

To properly exploit the possible gain of using a bigger network the inputs are complemented by two more values, raising the total number of inputs to six. These are the *death ratio* and the *degree of encirclement*. The other inputs are the same as for the small neural net.

Death ratio

The death ratio is the player death to agent death ratio averaged over the course of the last 5000 gamerounds. This allows the network to judge its past performance. The hyperbolic tangent (tanh(death ratio)) is used to scale the death ratio to fit as input.

Degree of encirclement

The degree of encirclement is a measurement allowing the agents to know if they are in a good position to attack. This is done by a method which takes into account the number of AI agents in each of the four quadrants around the player and weighs each agent by its distance to the player. If the AI agents are distributed evenly around the player and many are close the degree of encirclement will return a high value, if the agents are all situated on the same side of the player and far away it will return a low number. The degree is scaled to lie between 0.0 and 1.0 like all other inputs.

Basic big network

The first conditioning is done offline with a slightly modified training set from the small network, extended with values for the new inputs. The tests run for the new network are identical to the ones from chapters 6 and 7.

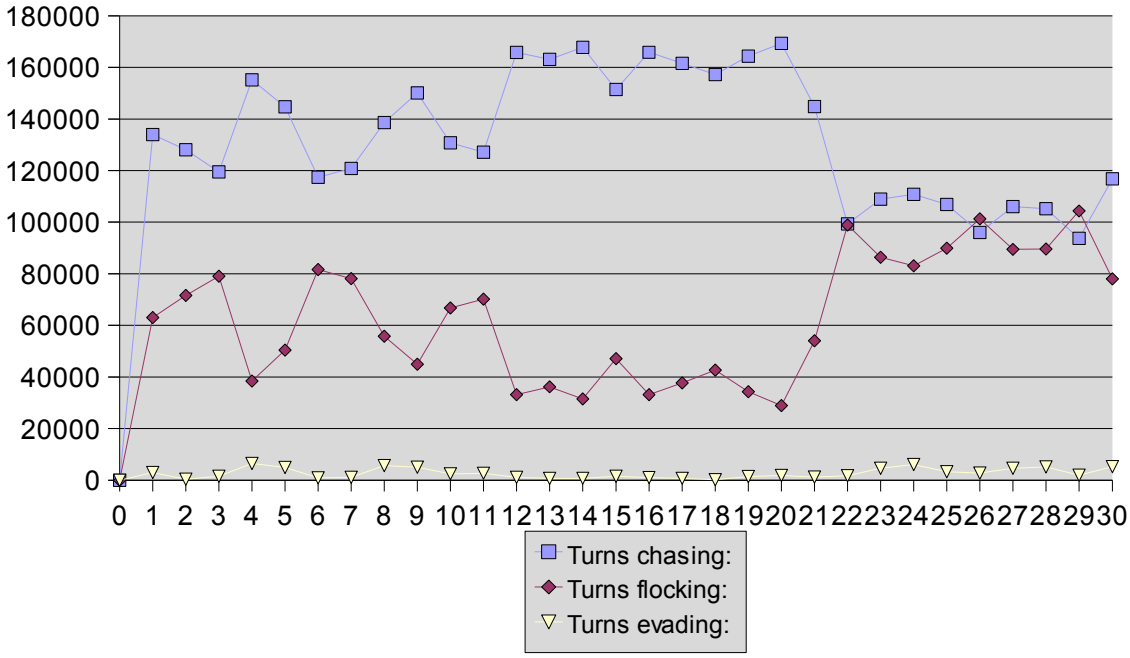

Figure 8.1 Turns per state distribution, basic big network

Unsurprisingly the absolute numbers of turns spent in the different states are similar to the basic small network albeit a stronger evasion can be observed, nearly twice as much as for the small neural net. Chasing dominates the standard and low damage segments while flocking gets nearly as strong as chasing during the high damage segment. Interestingly the variation between the behaviours is quite different though. The small network showed a constant swing between chasing and flocking while the bigger network shows a smoother behaviour and more pronounced differences betweens the test stages. This can also be seen in the better death ratio at 0.75 which rivals that of the small network with supervised learning.

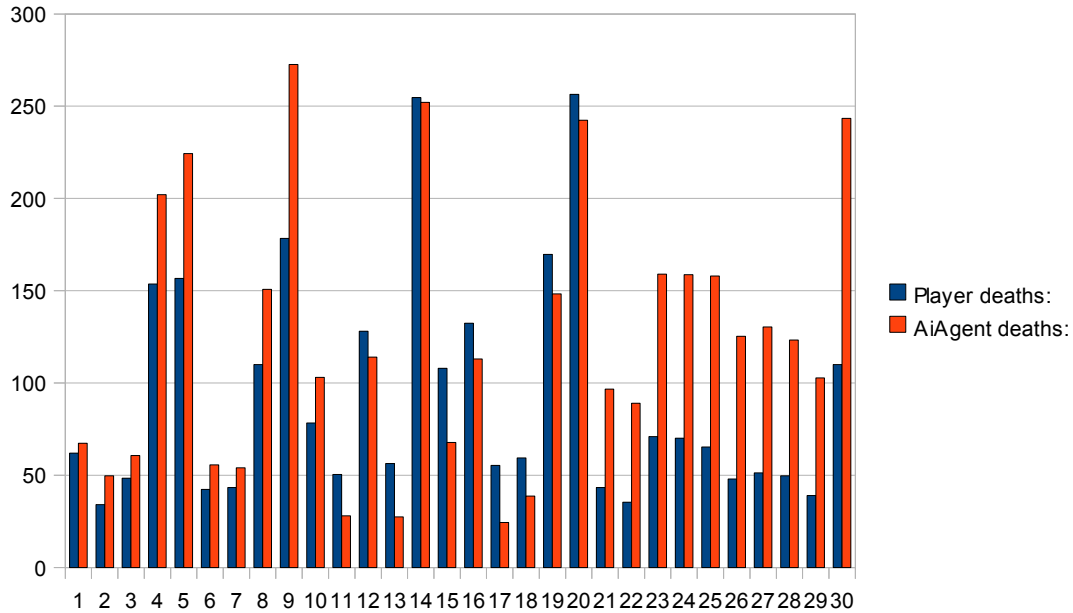

Figure 8.2 Death ratio, basic big network

Death ratio

The average death ratio of the basic big network is 0.75 The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.73
- Low: 1.2
- High: 0.42

Big network supervised learning

The supervised learning algorithm used for the small network is identical to the one developed for the original one.

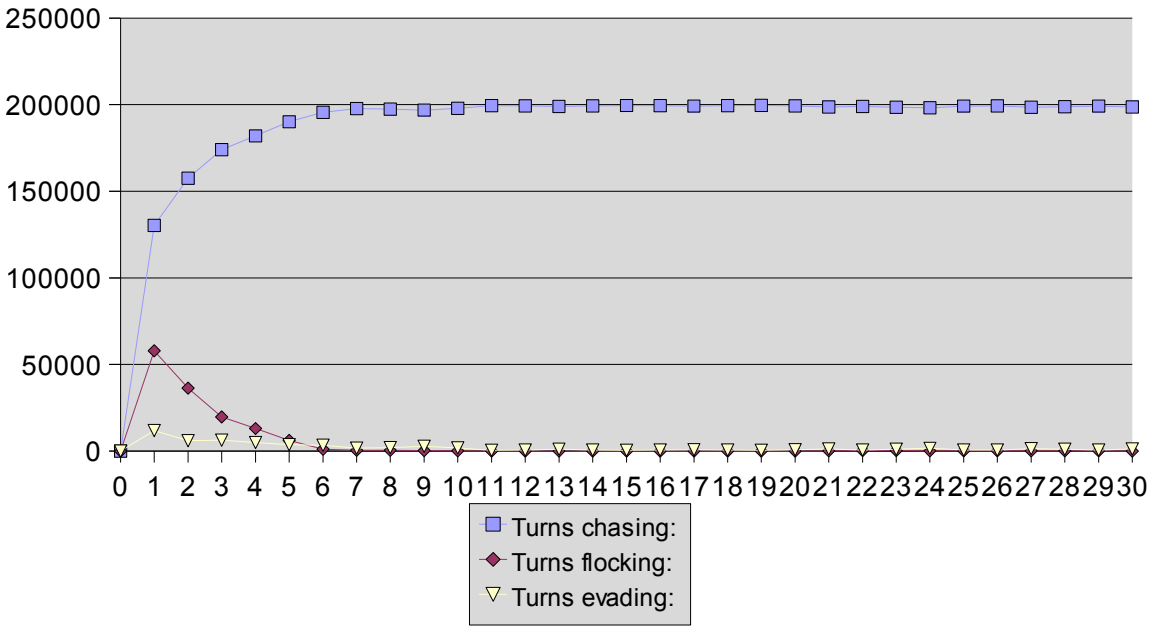

Figure 8.3 Turns per state distribution, big network supervised learning

The behaviour exhibited by the supervised big network is similar to the one by its smaller counterpart. Chasing becomes the dominant behaviour during the first 6 timesteps and flocking and evading get suppressed. This suppression effect is also slightly faster than for the small network, which shows when comparing the total number of turns spent in the different states.

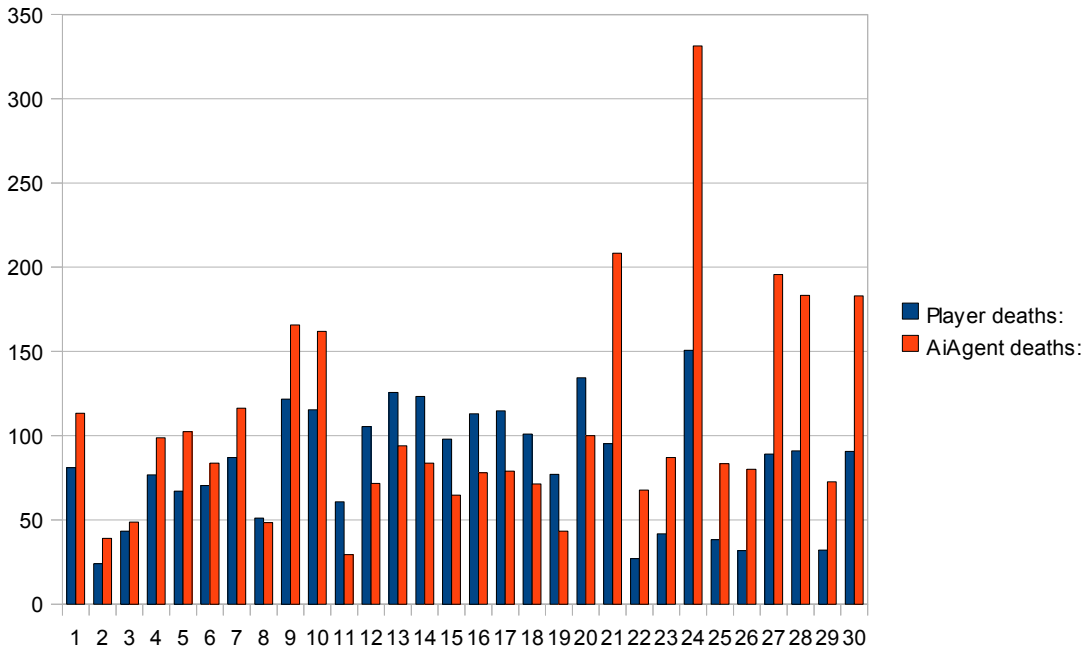

Figure 8.4 Death ratio, big network supervised learning

Death ratio

The average death ratio of the big network trained with supervised learning is 0.78 The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.75
- Low: 1.47
- High: 0.46

The death ratio again shows a slight advantage in efficiency compared to the smaller network.

Figure 8.5 shows that evading is constantly promoted but simply too weak in comparison to the chase behaviour to be of any impact on the behaviour.

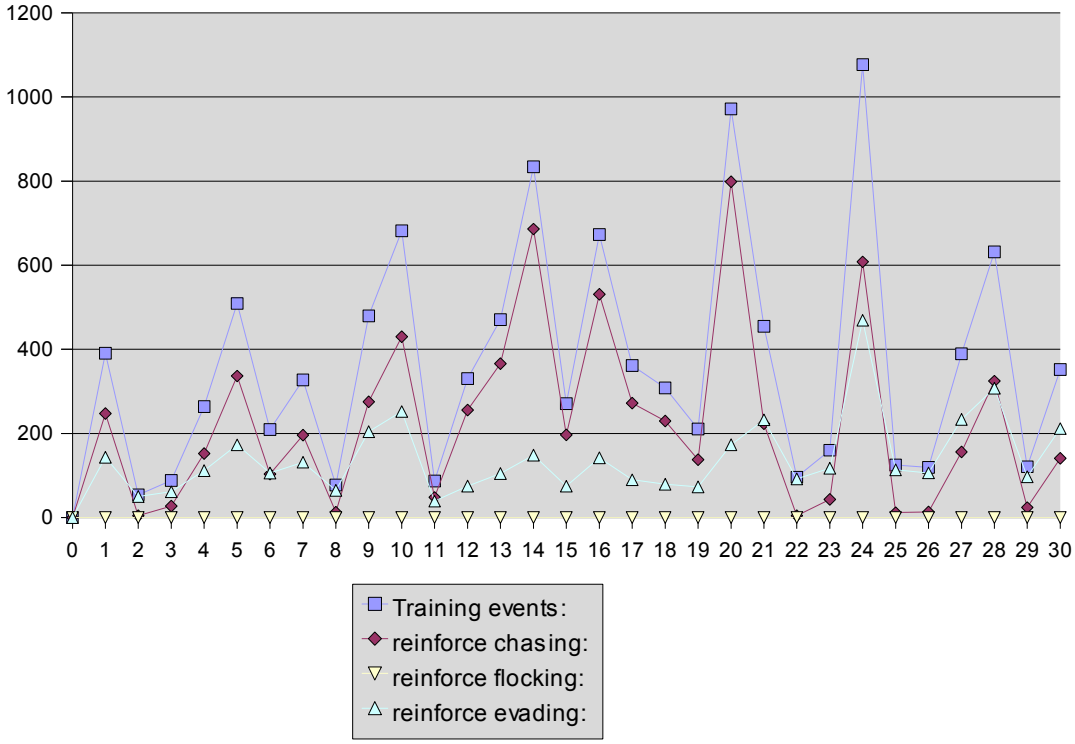

Figure 8.5 Training events, big network supervised learning

Big network, controlled supervised learning

The same controlled supervised learning algorithm introduced in chapter 5.7 was used to train the bigger network.

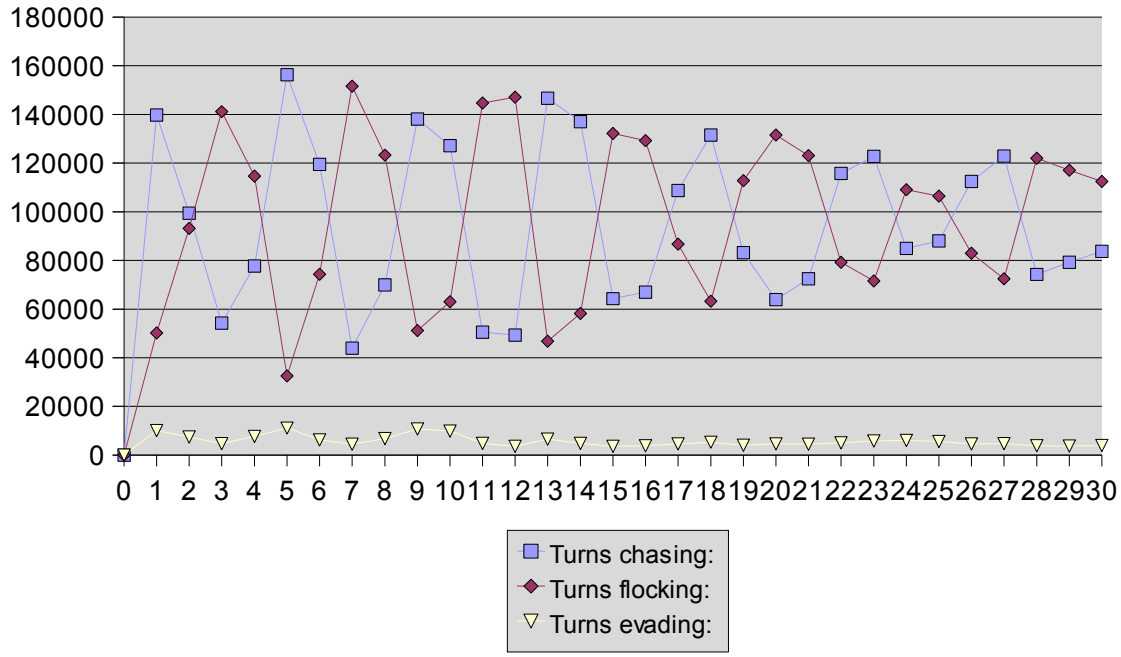

Figure 8.6 Turns per state distribution, big network controlled supervised learning

Figure 8.6 clearly shows the typical swinging back and forth between chasing and flocking resulting from the controlled learning rules which promote flocking. During the later stages of the tests, when the differences between the controlled and supervised learning decrease, the variance between the behaviours also gets smaller. Although still weak evading is stronger than in the basic and pure supervised learning networks.

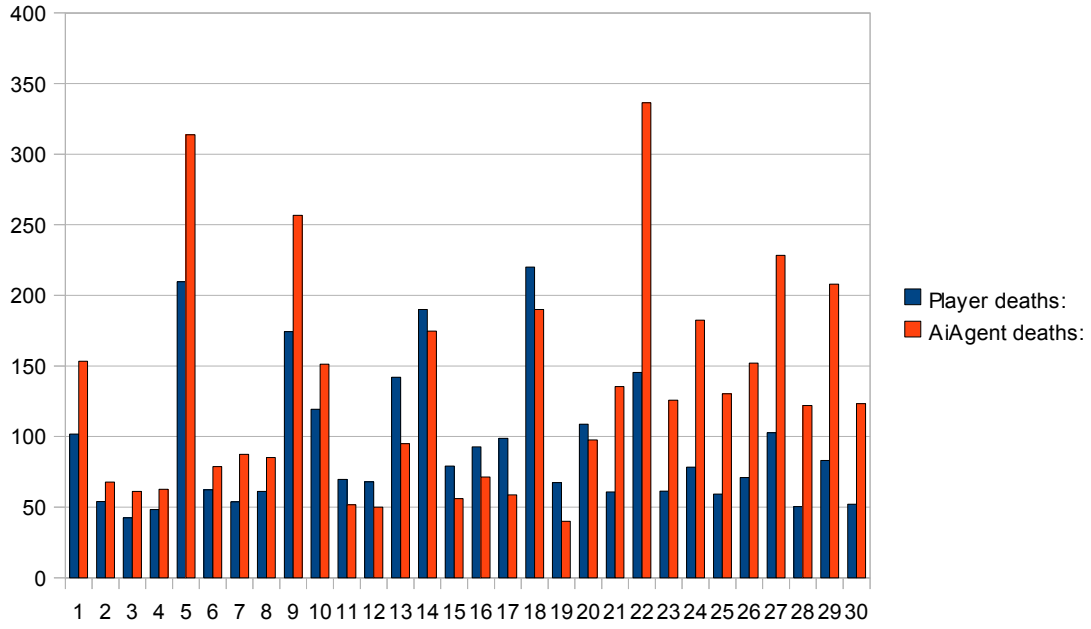

Figure 8.7 Death ratio, big network controlled supervised learning

Death ratio

The average death ratio of the big network trained with controlled supervised learning is 0.72 The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.7
- Low: 1.28
- High: 0.44

The lowest death ratio for the big network can be observed while using the controlled supervised learning although the ratio during the later test segments is slightly higher. This is consistent with the effect seen when using it in concert with the other learning types. The control rules force a more varied behaviour at the expense of some efficency and learning speed.

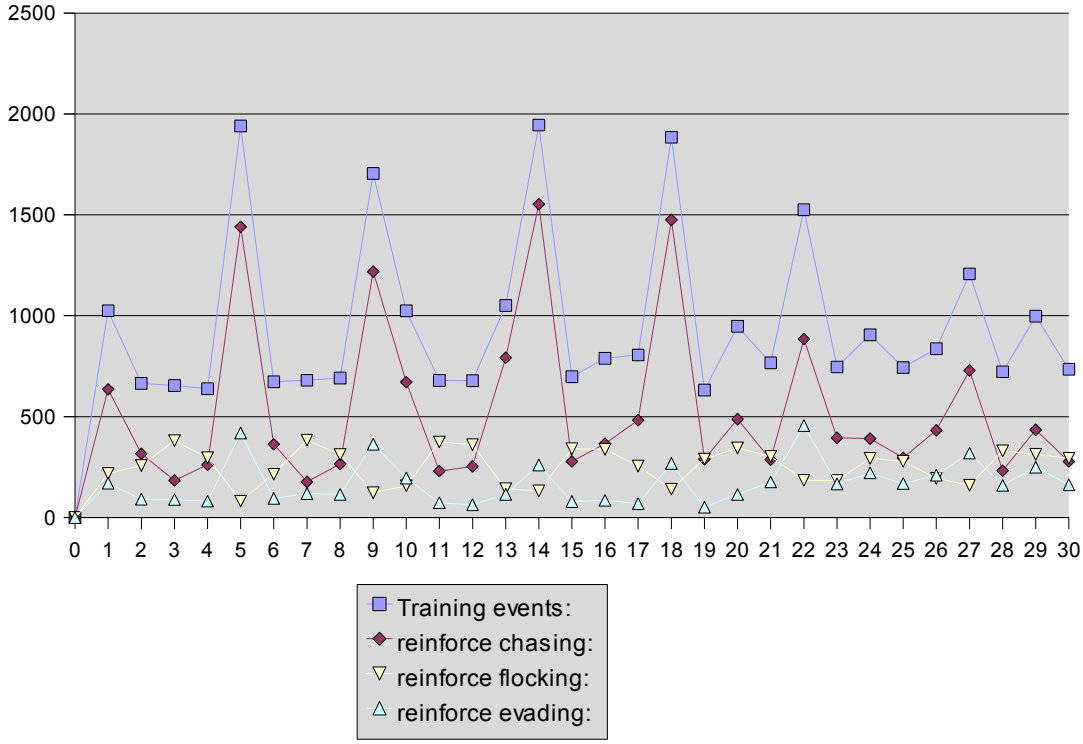

Figure 8.8 Training events, big network controlled supervised learning

Summary big network

The patterns observed while using the learning algorithms with a smaller network are repeated. Supervised learning raises the efficiency but also leads to monotonous behaviour while controlled learning provides variation but a drop in ratio. This can be seen in Figures 8.9 and 8.10.

Compared to the small neural network the bigger one shows an overall rise in efficiency. Although notable the rise is quite small and therefore the decision was made to use the smaller network for the more complex algorithms of the reinforcement learning as it is also capable of solving the Chase/Flock/Evade task in a sufficient manner. A more complex task obviously would profit more from the complexity inherent in bigger networks, especially the higher number of computable inputs.

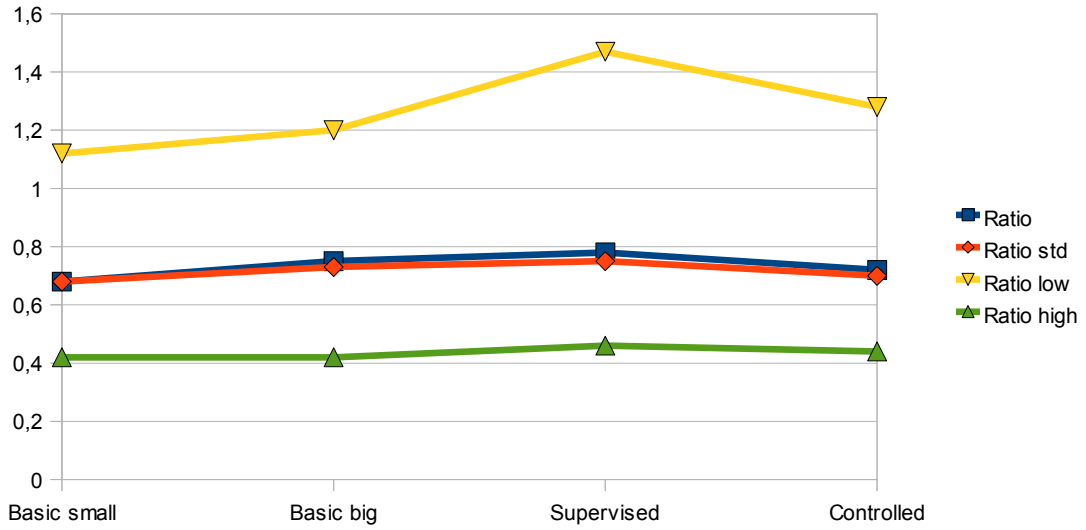

Figure 8.9 Death ratio comparison, big network

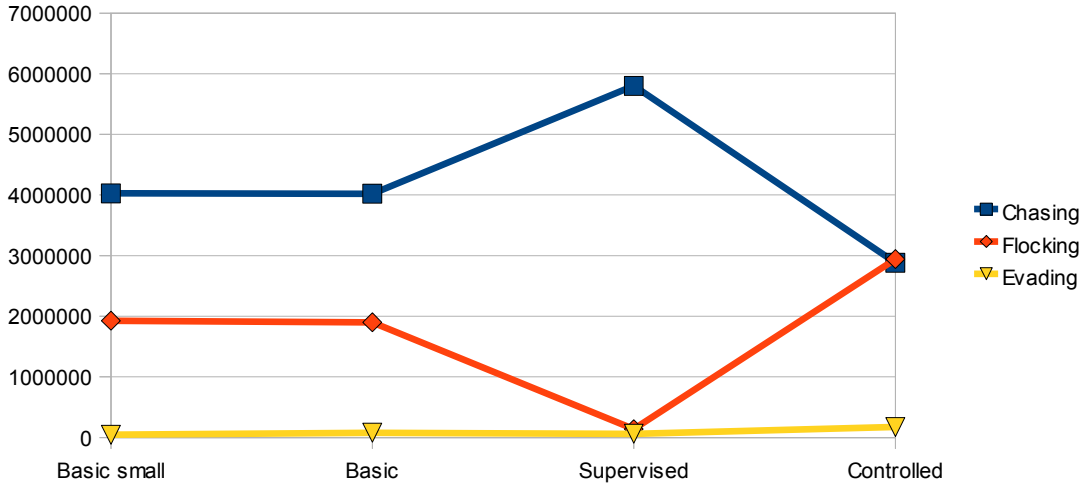

Figure 8.10 State distribution comparison, big network

Figure 8.11 shows that all tested versions of the big network lead to a similar number of agent and player deaths as their smaller network counterparts. No noticeable difference in overall behaviour can be seen.

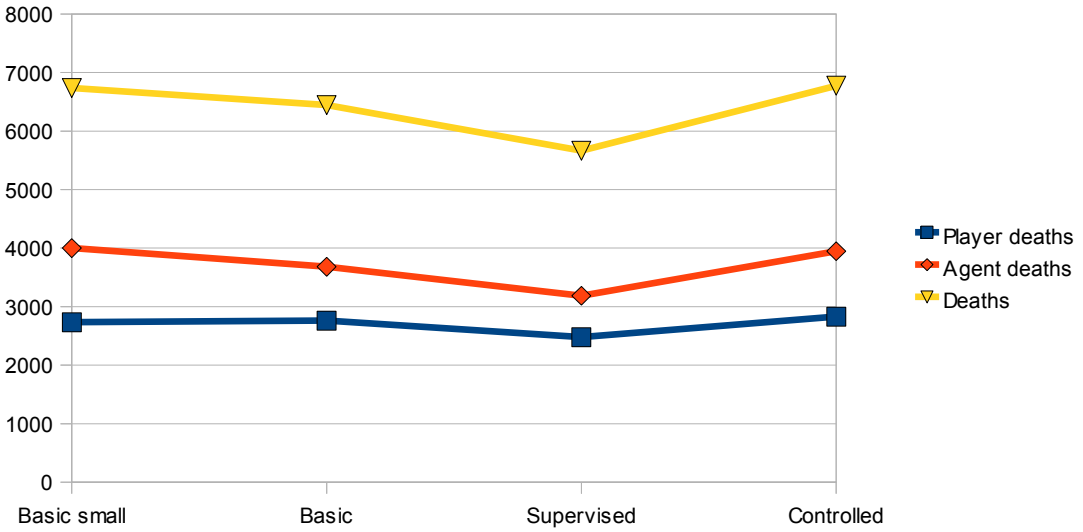

Figure 8.11 Total player and agent deaths comparison, big network

8.2 Two net architecture

Adding the second neural network

The idea behind using a second neural network was to to create a better coordination between the AI agents. The goal was to develop higher level tactics like surrounding the player and better group dynamics. To allow for a better assessment of the game situation the second network counts the number of AI agents in close proximity to the player and rates the situation in order to advise the AI agents to become more aggressive or defensive, it also takes their relative position to the player into account. This new network is called the “situational awareness” network. The original network was slightly redefined and named the “tactical” network.

In reinforcement learning terms the situational awareness net can also be seen as partly fulfilling the role of a state value function, whereas the tactical net represents the action selection policy.

The flocking behaviour was switched for a “move to target” behaviour called maneuvering, this behaviour replaces the flocking. An independent method which takes into account the relative position of the AI agent to the player and the result of the encirclement function is called to choose an appropriate goal for maneuvering agents.

To aid the situational awareness network in judging the overall situation two helper methods were defined to produce correctly scaled input. These are identical to the ones describe in chapter 8.1 under *degree of encirclement* and *death ratio*.

One judges the degree of encirclement of the player by taking into account the number of AI agents in front, behind, left and right of the player, weighted by their distance. The other simply calculates the death ratio during the last 1000 gamerounds giving the neural net a way to judge the past progress.

These two measurements in addition to the pure number of AI agents in close proximity to the player allow the net to evaluate the game situation independent of a single agent's own condition. The situational awareness network then creates a judgement of the overall situation and communicates this to the single agents which then take their

own, personal situation into account to make a decision.

Maneuvering

An agent in the maneuvering state will move to a set goal. This goal is decided upon by the degree of encirclement function in addition to examining the overall positioning of the agents relative to the player. The space around the player is divided into four quadrants relative to its position and direction: Front right, front left, back right and back left. The goal for the agent in the maneuvering state is always the least occupied quadrant on the side of the player closer to the agent. This ensures that the maneuvering agents will always try and surround the player before they attack.

The new in- and outputs

Inputs to the situational awareness network.

- Input 0: Degree of encirclement
- Input 1: Number of AI agents close to player
- Input 2: Player engaged or not
- Input 3: Death ratio during the last 1000 gamerounds

Outputs of the situational awareness network.

- Output 0: Attack advice
- Output 1: Maneuver advice
- Output 3: Evade

The outputs of the situational awareness network are treated as general advice to the single agents by using them as inputs for the tactical network.

Inputs to the tactical network.

- Input 0: Output 0 of the situational awareness net (attack)
- Input 1: Output 1 of the situational awareness net (maneuver).
- Input 2: Distance to player / MaxDistance
- Input 3: Hitpoints

Outputs of the tactical network.

- Output 0: Chase
- Output 1: Maneuver
- Output 2: Evade

The evade output of the situational awareness network is not directly fed into the tactical net. It is only used to indirectly influence the probabilities of the other two outputs during the controlled learning.

Basic two network architecture

Two net architecture basic training

The two net architecture is trained using one independent training set for each network.

The training of two networks proved to be more difficult. As a result the maneuvering dominated the initial behaviour. When examining the networks the situational awareness network could be held responsible for this. Its basic version favours the maneuver behaviour so strongly that the other inputs to the tactical net are overruled.

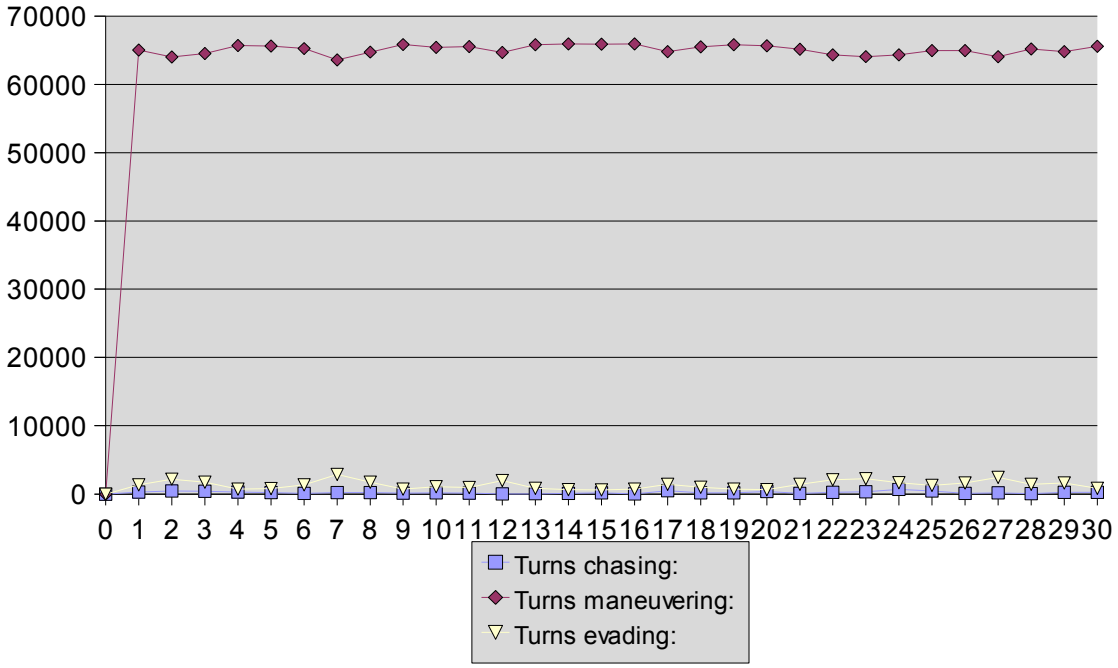

Figure 8.12 Turns per state distribution, basic two net architecture

This leads to a very weak behaviour during the high damage segment as can be seen in figure 8.13. The average death ratio is still quite high though as the maneuver behaviour leads to a homogenous movement of the AI agents. Therefore it is hard for the player to achieve a good performance throughout the game.

Death ratio

The average death ratio of the basic two net architecture is 0.8 The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.88
- Low: 2.0
- High: 0.47

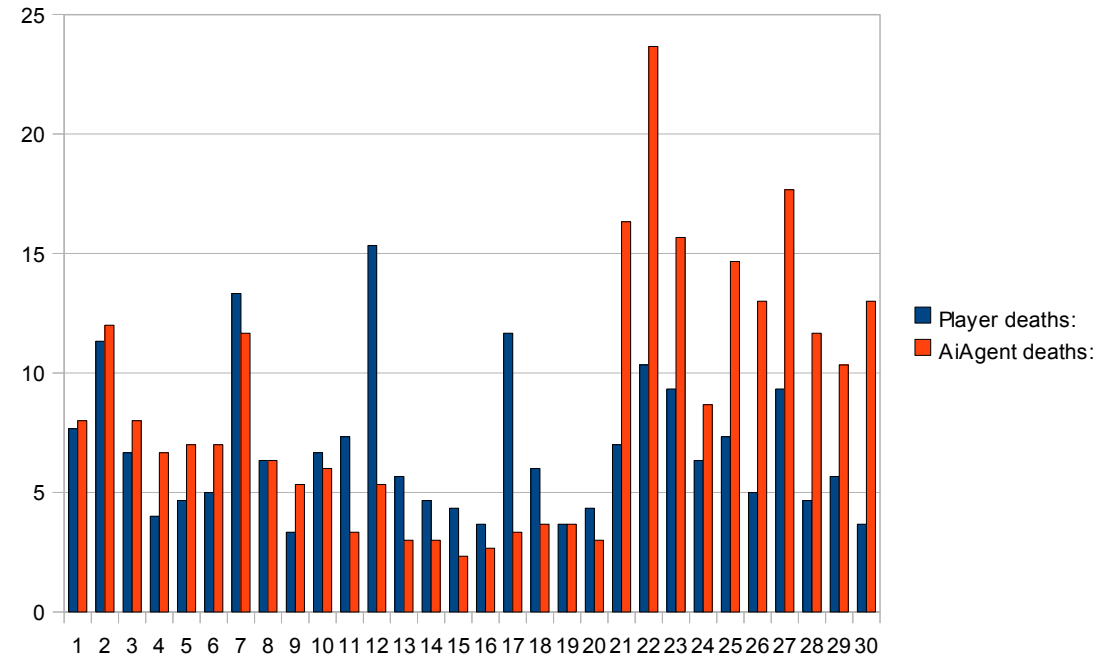

Figure 8.13 Death ratio, basic two net architecture

Two net architecture supervised learning

A *supervised* learning algorithm for the two net architecture was implemented which is an adaptation of the algorithm for the single net architecture.

Supervised learning algorithm for the two net architecture

The supervised learning algorithm for the two net architecture can be separated into two separate algorithms for each of the two networks.

The algorithm for the tactical net is identical to the one used for the single small network:

```
//supervised learning for tactical net
//if agent is alive
if ( (activeEntity.getHitpoints() >= 0.0))
{
        //if entity was close while player died, reinforce chase behaviour
        if (kill && (distance < CombatDistance))
        {
                tacticalNet.Retrain(0.9, 0.1, 0.1);
        }
}
```

```
//if agent died, encourage evasive behaviour
if (activeEntity.getHitpoints() < 0.0)
{
        tacticalNet.Retrain(0.1, 0.1, 0.9);
}
```

Figure 8.14 Supervised learning, tactical network

The algorithm for the situational awareness network takes into account the number of agents that died while trying to kill the player. If the player was killed and less than two agents died in the process this is seen as a success and the network is trained to advise the chase behaviour in a similar situation. If the player was not killed but at least one agent died the network is trained to advise the use of maneuvering.

```
//supervised learning situational awareness net
if (kill && agentDeathsThisRound < 2)
{
        situationalAwarenessNet.Retrain2(0.9, 0.1, 0.1);
}
else
{
        if (!kill && agentDeathsThisRound > 0)
        {
                situationalAwarenessNet.Retrain2(0.1, 0.9, 0.1);
        }
}
```

Figure 8.15 Supervised learning, situational awareness network

Comparable to its single net architecture counterpart, the supervised learning algorithm converges to relatively monotonous behaviour during the first third of the test. It does so even stronger as the second network speeds up the training effect.

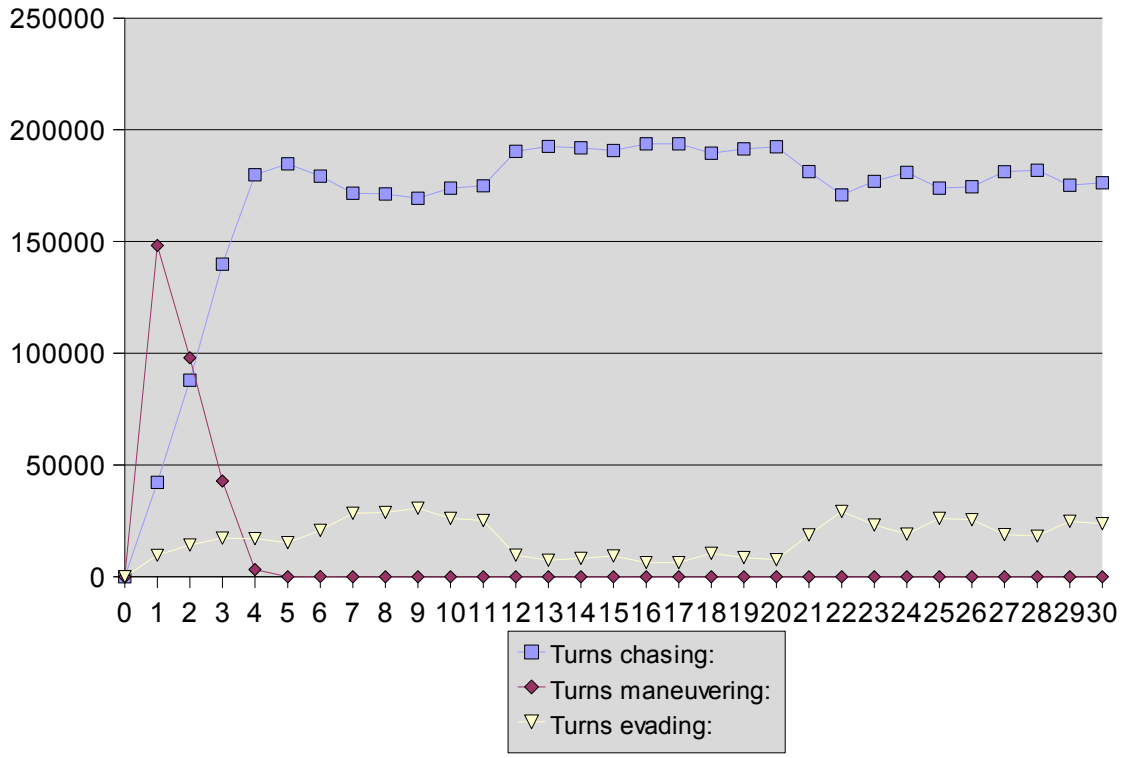

Figure 8.16 Turns per state distribution, two net architecture supervised learning

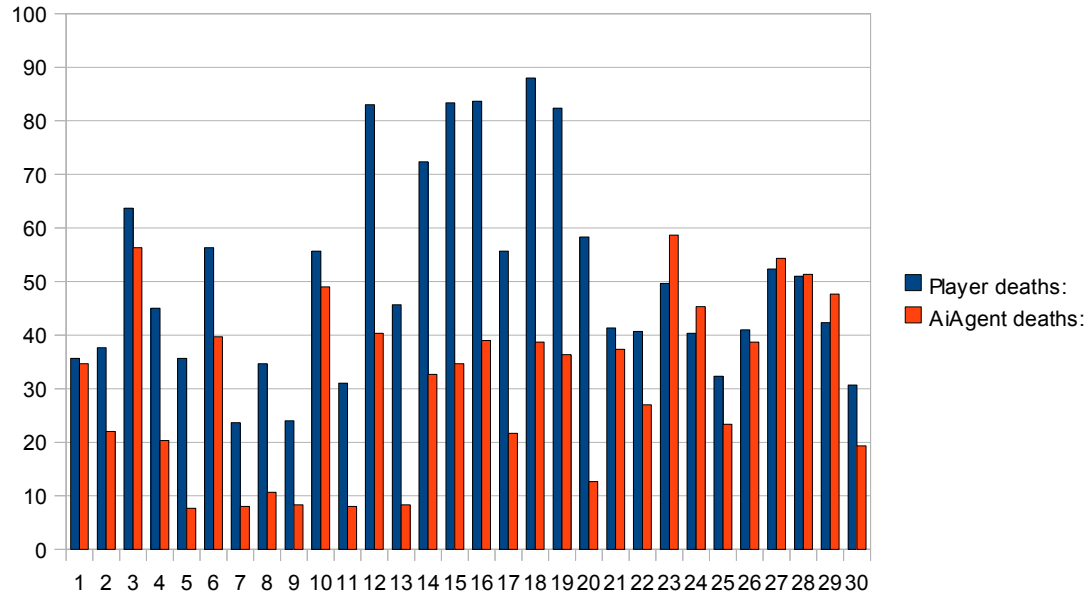

Figure 8.17 Death ratio, two net architecture supervised learning

Interestingly the death ratio is very high at 1.63.

Death ratio

The average death ratio of the supervised two net architecture is 1.63 The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 1.61
- Low: 2.51
- High: 1.05

Two network architecture controlled supervised

As for the single net supervised algorithm a set of control rules is implemented to guarantee a more varied behaviour. It consists of three parts: One which trains the tactical net to adhere to several specific input values, one which trains it to specifically follow the advice from the situational awareness network and one which trains the situational awareness network to steer towards a default assessment of the overall situation.

Their implementation is shown in Figure 8.18

```
//if agent is dying -> evade
if ((activeEntity.getHitpoints())  < 10.0 && (distance < CombatDistance))
{
        tacticalNet.Retrain(0.1, 0.1, 0.9, activeEntity);
}
else
{
        //if player not too close -> maneuver
        if ((situationalAwarenessNet.GetOutput(1) < 0.4)
        &&(distance> CombatDistance) )
        {
                tacticalNet.Retrain(0.1, 0.9, 0.1);
        }
        else
        {
                //if agent is too far away -> maneuver
                if(distance > CombatDistance)
                {
                        tacticalNet.Retrain(0.1, 0.9, 0.1);
                }
```

```
                        //if agent is close to player and not alone -> attack
                        else
                        {
                                if ( distance < 2 * CombatDistance &&
                                        situationalAwarenessNet.GetOutput(0) > 0.4)
                                {
                                        tacticalNet.Retrain(0.9, 0.1, 0.1);
                                }
                        }
                }
}

//train tactical net  to follow situational awareness net
if (situationalAwarenessNet.GetOutput(0) > 0.4)
{
                tacticalNet.Retrain(0.9, 0.1, 0.1);
}
else
{
                if (situationalAwarenessNet.GetOutput(1) > 0.4)
                {
                        tacticalNet.Retrain(0.1, 0.9, 0.1);
                }
                else
                {
                        tacticalNet.Retrain(0.1, 0.1, 0.9);
                }
}

//retrain situational awareness net
if ( (degree > 0.4) && (playerDeathToAgentDeathRatio > 0.5) )
{
                situationalAwarenessNet.Retrain(0.9, 0.1, 0.1);
}
```

```
else
{
        if ( (degree < 0.3) && (playerDeathToAgentDeathRatio > 0.4) )
        {
                situationalAwarenessNet.Retrain(0.1, 0.9, 0.1);
        }
        if ( (degree < 0.2) && (playerDeathToAgentDeathRatio < 0.2) )
        {
                situationalAwarenessNet.Retrain(0.1, 0.1, 0.9);
        }
}
```

Figure 8.18 Controlled learning, two net architecture

As can be seen in Figure 8.19 the control rules successfully prevent the agents from learning a onesided behaviour. Only the evade behaviour is comparatively low but still noticeable.

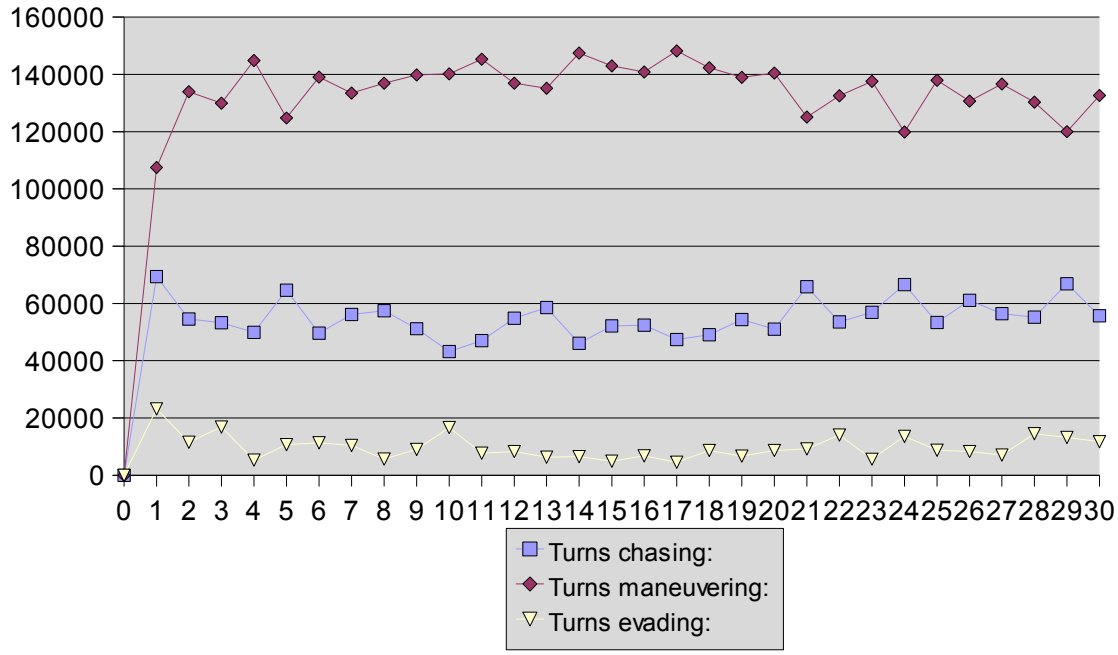

Figure 8.19 Turns per state distribution, two net architecture controlled supervised

The controlled behaviour is less efficient though, similar to the single network architectures. It is still superior to the basic network.

Death ratio

The average death ratio of the controlled supervised two net architecture is 0.87 The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.96
- Low: 1.86
- High: 0.53

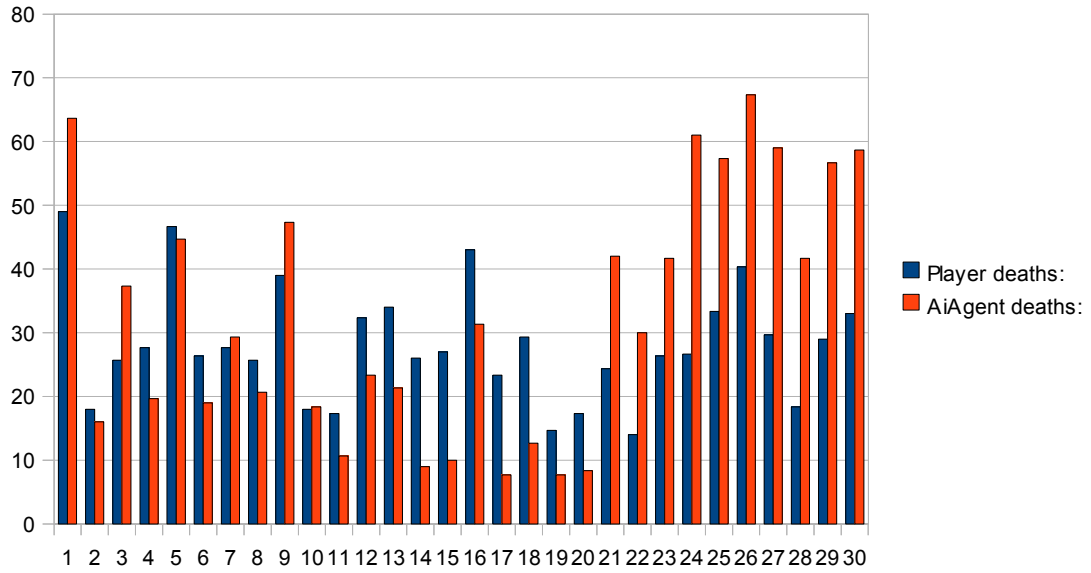

Figure 8.20 Death ratio, two net architecture controlled supervised learning

Two network architecture reinforcement learning

The reinforcement learning algorithm for the tactical network is an adaptation of the 0.5 rule algorithm. The situational awareness network needed a completely new algorithm. It is based on the overall number of agent deaths suffered when the player got killed to judge the assessment of the situation. If more than two agents died it is seen as a failure and the state and state evaluation is punished, if less than two agents died the current assessment is rewarded.

The implementation of the reinforcement learning is shown in Figure 8.21

```
//reinforcement learning for tactical net
//if agent is alive
if ( (activeEntity.getHitpoints() >= 0.0))
{
        //if entity was close while player died, reinforce current behaviour
        if (kill && (distance < CombatDistance))
        {
                tacticalNet.FeedForward();
                behaviour = tacticalNet.GetMaxOutputID();
                switch (behaviour)
                {
                        case chasing:
                        tacticalNet.RetrainMemory(tacticalNet.GetOutput(0),
                                                  tacticalNet.GetOutput(1),
                                                  tacticalNet.GetOutput(2));
                        break;
                        case maneuvering: //skip reinforcing of maneuvering behaviour
                        break;
                        case evading:
                        tacticalNet.RetrainMemory(tacticalNet.GetOutput(0),
                                                  tacticalNet.GetOutput(1),
                                                  tacticalNet.GetOutput(2));
                        break;
                }
        }
}
//if agent died, discourage current behaviour
if (activeEntity.getHitpoints() < 0.0)
{
        tacticalNet.FeedForward();
        behaviour = tacticalNet.GetMaxOutputID();
        switch (behaviour)
        {
                case chasing:
                        tacticalNet.Retrain(0.5*tacticalNet.GetOutput(0),
                                            0.5*tacticalNet.GetOutput(1),
                                            tacticalNet.GetOutput(2));
                break;
```

```
                    case maneuvering:
                        tacticalNet.Retrain(tacticalNet.GetOutput(0),
                                            0.5*tacticalNet.GetOutput(1),
                                            tacticalNet.GetOutput(2));
                        break;
                        case evading:
                        tacticalNet.Retrain(tacticalNet.GetOutput(0),
                                            0.5*tacticalNet.GetOutput(1),
                                            tacticalNet.GetOutput(2));
                        break;
            }
}

//reinforce situational awareness network
if (kill && agentDeathsThisRound < 2)
{
        situationalAwarenessNet.Retrain2(situationalAwarenessNet.GetOutput(0),
                                         situationalAwarenessNet.GetOutput(1),
                                         situationalAwarenessNet.GetOutput(2));
}
else
{
        if (!kill && agentDeathsThisRound > 1)
        situationalAwarenessNet.Retrain(0.5*situationalAwarenessNet.GetOutput(0),
                                        0.5*situationalAwarenessNet.GetOutput(1),
                                        situationalAwarenessNet.GetOutput(2));
        if (!kill && agentDeathsThisRound > 0)
        situationalAwarenessNet.Retrain(0.5*situationalAwarenessNet.GetOutput(0),
                                        0.5*situationalAwarenessNet.GetOutput(1),
                                        situationalAwarenessNet.GetOutput(2));
}
```

Figure 8.21 Reinforcement learning, two net architecture

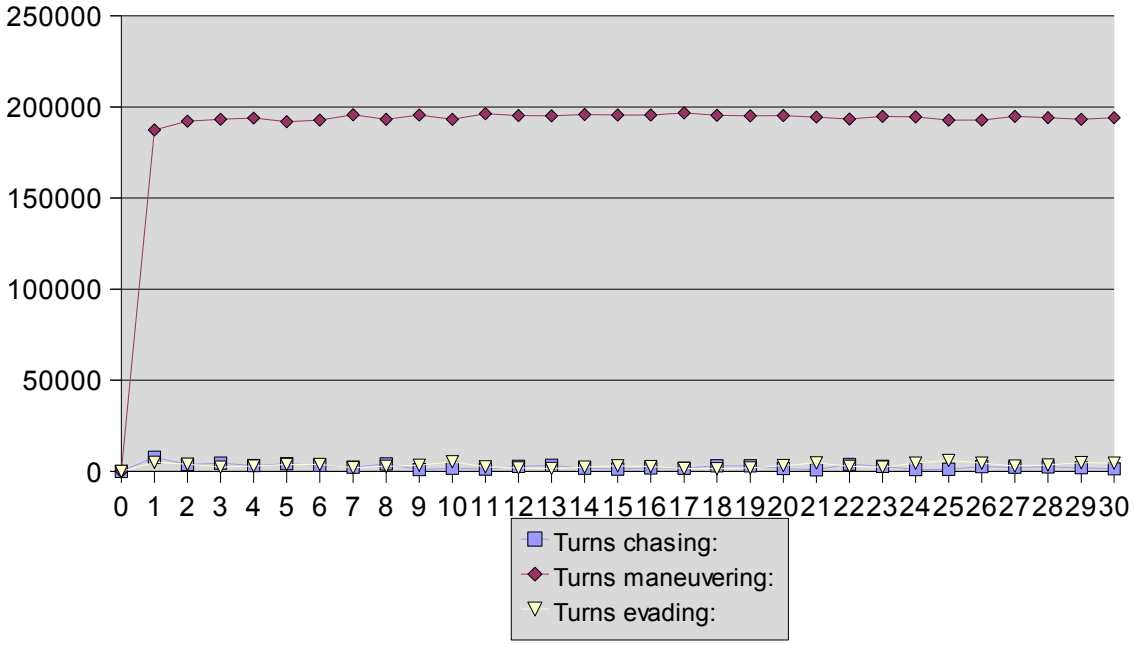

Figure 8.22 Turns per state distribution, two net architecture reinforcement learning

The reinforcement learning leads to a monotonous behaviour similar to the supervised learning. In this case maneuvering is the one dominant state where the supervised algorithm prefered chasing combined with a solid amount of evading.

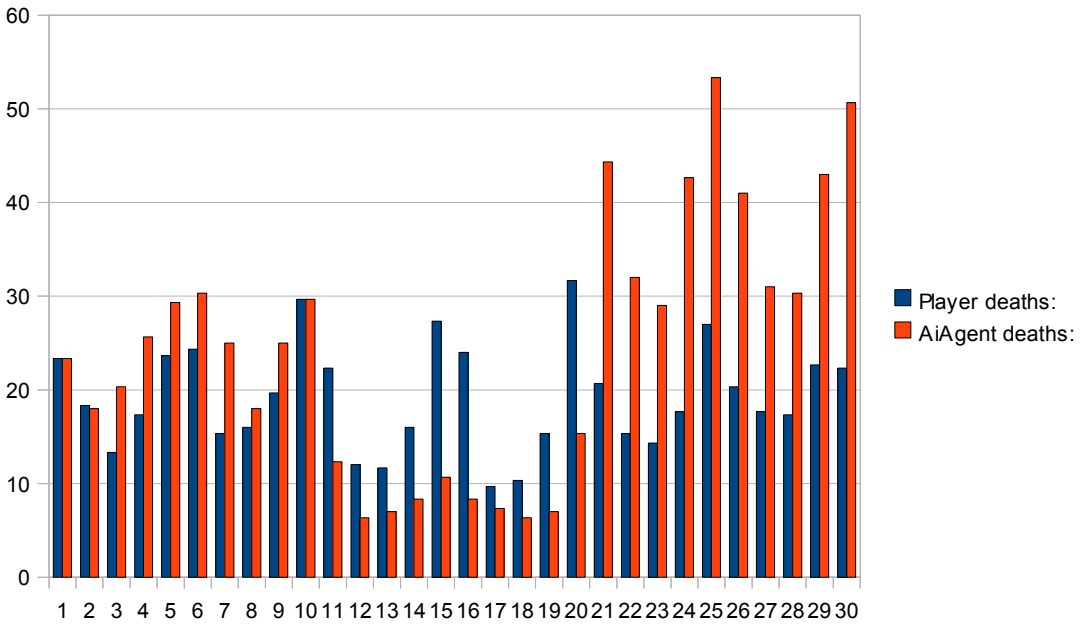

Figure 8.23 Death ratio, two net architecture reinforcement learning

This leads to a relatively bad death ratio as the AI agents are unable to adapt to the high damage segment of the test.

Death ratio

The average death ratio of the two net architecture with reinforcement learning is 0.79 The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.82
- Low: 2.03
- High: 0.49

Two network architecture controlled reinforcement learning

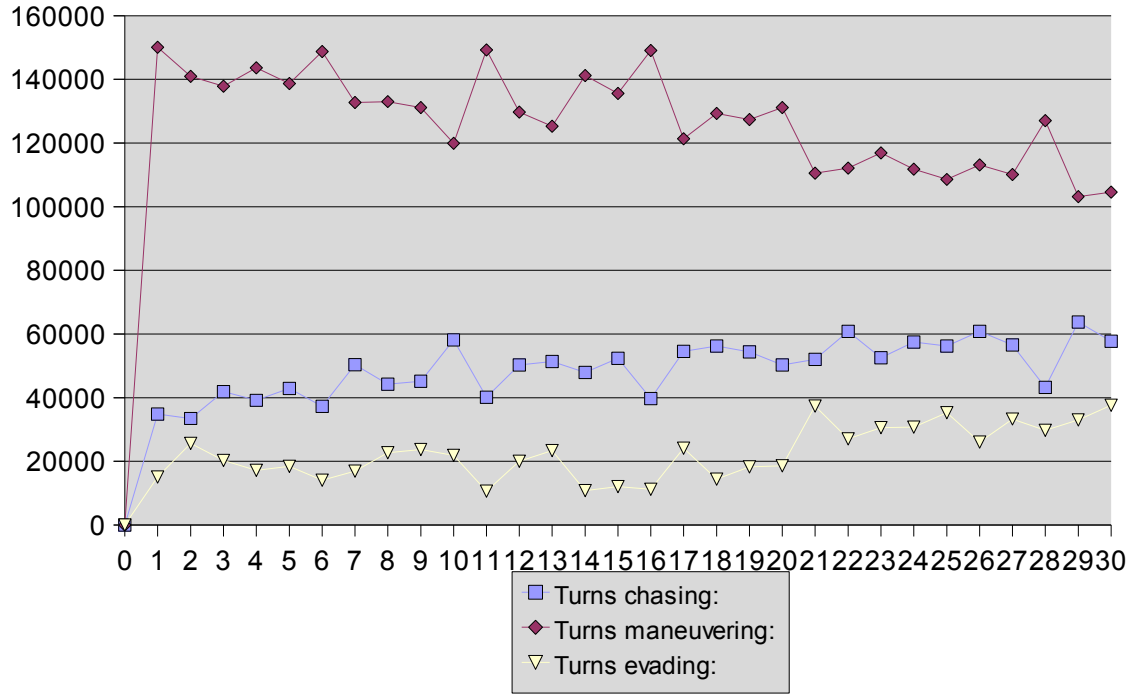

Figure 8.24 Turns per state distribution, two net architecture controlled reinforcement

The control rules are able to promote chasing and evading strongly enough to lead to a more varied behaviour. Maneuvering is still the prefered state of the AI agents though.

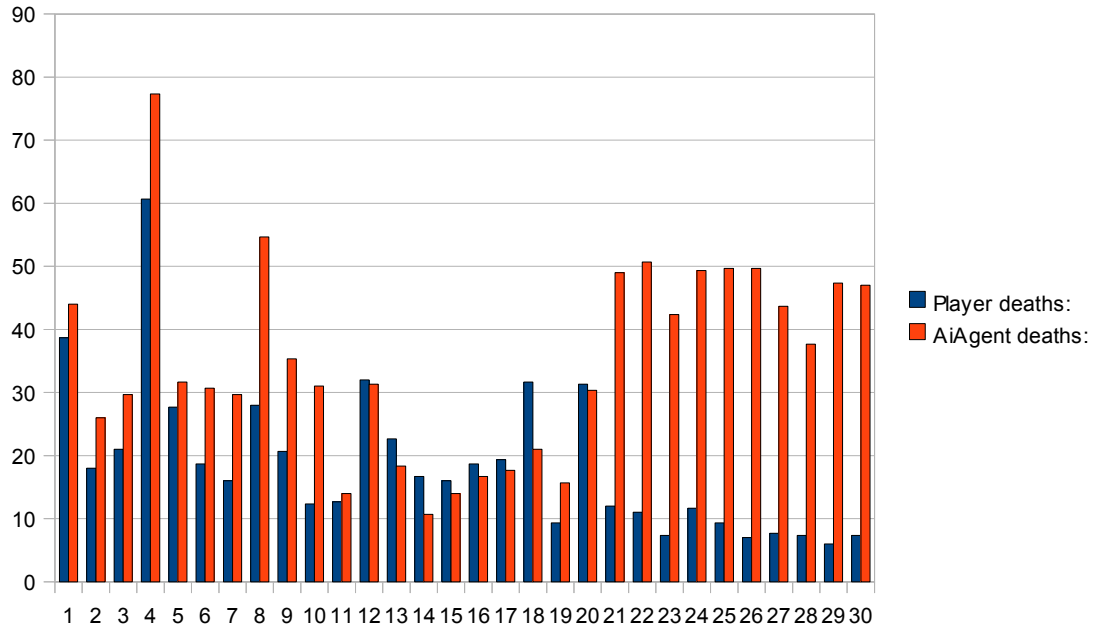

Figure 8.25 Death ratio, two net architecture controlled reinforcement learning

The combination of the control rules and the reinforcement algorithm for the two net architecture weakens the efficiency of the reinforcement learning. The death ratio is the lowest of all the two net algorithms.

Death ratio

The average death ratio of the two network architecture using controlled reinforcement learning is 0.53 The average death ratios for the standard damage, lower damage and higher damage are as follows:

- Standard: 0.67
- Low: 1.11
- High: 0.19

Summary two net architecture

The two net architecture proved to be more difficult to work on, especially considering it was examined late in the project and therefore less time was spent on adjusting the learning algorithms specifically for its special needs. As a result the behaviours exhibited during the tests were quite varied.

The supervised algorithms showed the best results as they were easier to implement and adapt. This shows most obviously in Figure 8.26. The pure supervised algorithm has the highest death ratio with the controlled variant coming in second. The reinforcement learning on the other hand even dropped the ratio below the basic network. As with all the other algorithms the controlled learning lowers the efficiency but raises the variety in behaviour as can be seen in Figure 8.27.

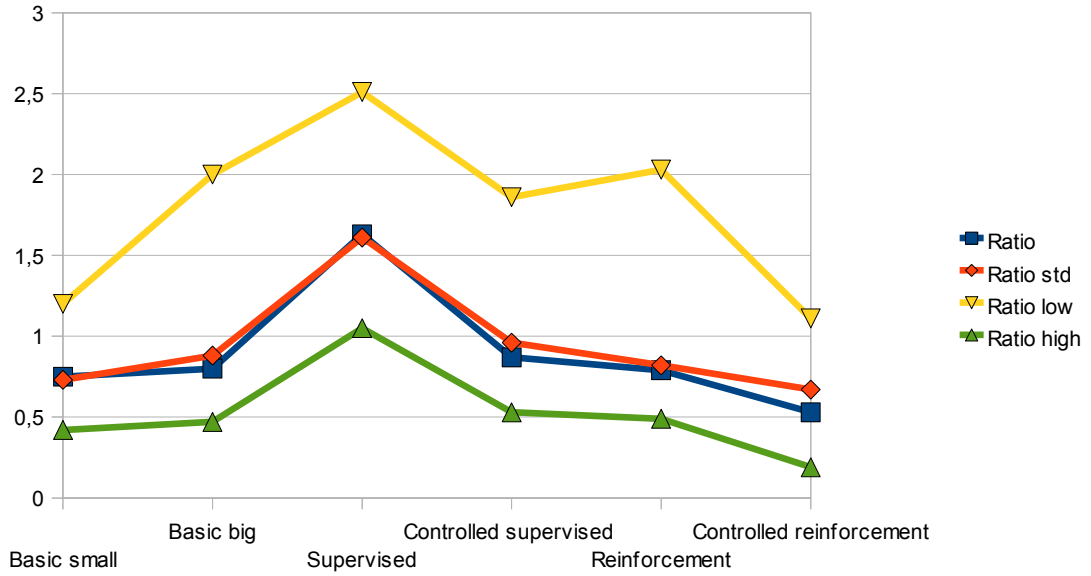

Figure 8.26 Death ratio comparison, two net architecture

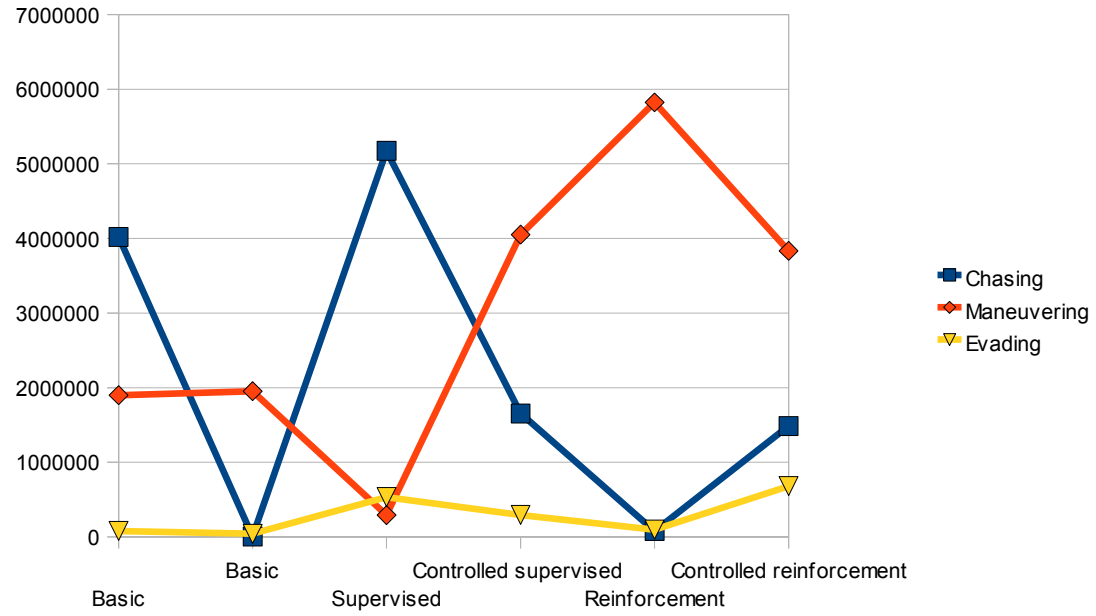

Figure 8.27 State distribution comparison, two net architecture

Figure 8.28 shows the overall impact of the exchange of the maneuvering for the flocking. The total numbers of player and agent deaths is much lower than the ones for the single network architecture. This stems from the maneuver behaviour and the situational awareness network which influence the agents to wait relatively long and only attack in greater numbers. This means the agents will spend more time preparing the attack and thereby cause less player deaths but also suffer less losses themselves.

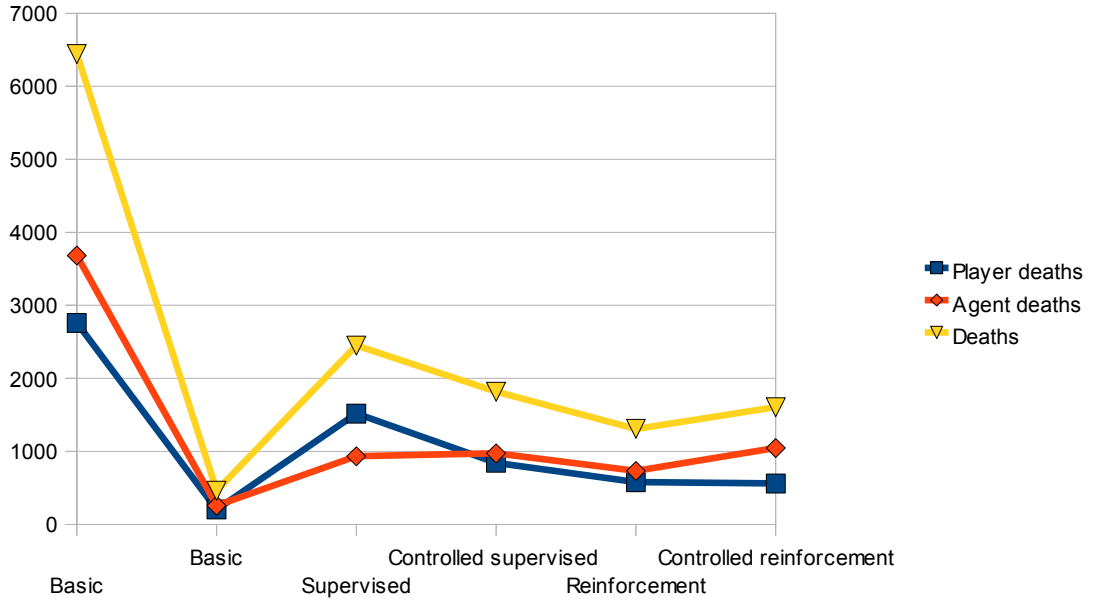

Figure 8.28 Total player and agent deaths, two net architecture

8.3 Summary of chapter 8

Two alternative network structures for the neural network were tested in chapter 8. The bigger single network proved to be slightly better than its smaller counterpart but the overall improvements in efficiency and behaviour were not big enough to warrant using it instead of the small network, which is fully capable of solving the requirements needed for the AI in the scope of the Chase/Flock/Evade task.

To test the two network structure the task had to be slightly modified by replacing the flocking behaviour with the maneuvering. This was done to give the situational awareness network a bigger influence and a more challenging problem to solve. As a result the overall efficiency could be raised for the supervised learning. A direct comparison to the single network architectures is not feasible though, as the maneuver behaviour automatically leads to a better positioning of the AI agents relative to the player, compared to the quasi random flocking movement. The reinforcement learning also proved to be very hard to implement at a satisfying level. This partly stems from the problem in having both networks contributing to the behaviour while it would probably be easier to let only the situational awareness network decide when to attack or flee. This shows a structural weakness of the task definition. The Chase/Flock/Evade task is better suited to be solved by a single network. The two net architecture would be better suited to a more complicated task. In the context of the Chase/Flock/Evade task the two networks often contradict each other which leads to the difficulties in using them to create an intelligent and stable behaviour.

9 Discussion and outlook

9.1 Discussion

Main goals

The goal of this thesis was the development of adaptive AI agents working in a continuous real time environment with the help of artificial neural networks. To a good extent this goal was reached.

AI agents were developed being able to react to changing circumstances in the environment, in the shape of playing against a human player and a simple AI, and also to changes in the amount of damage dealt by the player. This was achieved by the generalization capabilities inherent in neural networks and the application of supervised and reinforcement learning techniques. The algorithms examined in this thesis did improve the effectiveness and also the behaviour of the agents. The downside of the pure learning algorithms was their tendency to create onesided behaviour which might be more effective but is also often unwanted in the context of modern videogames as it can create a boring playing experience for a human player. To counter this effect the concept of controlled learning was introduced which allows to mix the learning effect with handcoded, clearly defined behaviour. This weakened the effect of the learning but created better defined AI. The motivation for this is, showing that it is possible to control the unwanted side effects of automatically learning algorithms and making them viable to be used in commercial games.

Additional questions

The task definition made for this thesis also creates more questions. Are the explored techniques only viable for small, simple tasks like the Chase/Evade/Flock problem? Or do they have the potential to be useful for more complex AI? To solve these questions more work has to be done as proposed in chapter 9.3.

9.2 Evaluating the AI

In chapter 3.5 several criteria were defined to evaluate online learning. They are repeated here for convenience.

Computational requirements:

- Speed: Online learning must be computationally fast as it takes place during runtime.
- Effectiveness: Online learning must create effective game AI, at least as good as manually created code.
- Robustness: It has to be able to deal with randomness.
- Efficiency: The wanted behaviour has to emerge swiftly enough (after a low number of trials) to be useful ingame.

Functional requirements:

- Clarity: The emergent behaviour has to be easy to understand.
- Variety: The learning process has to yield a variety of behaviour so that the game does not get boring.
- Consistency: The average number of trials before the wanted behaviour emerges has to be consistent, there should not be a huge variety in the amount of time it takes to learn.
- Scalability: The learning process must be able to scale to the difficulty level of the game.

Speed

The performance of the developed algorithms was not a focus of this thesis. They were sufficiently fast to not slow down the simulation and most of the computational time is spent for the graphics engine. Nonetheless it can be marked that there are several easy ways of improving the performance of the learning structures devised during this thesis. Firstly the learning is based on training events happening ingame, the number of training events processed during a set number of timesteps can be easily capped at a set amount to stabilize the number and time of needed calculations. Secondly the algorithms can easily be shifted to a thread independent of the normal game AI. A copy of the neural network could be easily used to learn, while another copy is used to do the actual ingame AI. The learning network running parrallel for a fixed amount of time and then getting exchanged with the ingame network. The network training process itself could also be spread over a set number of gameticks.

Effectiveness

The online learning methods were able to improve the effectiveness of the AI agents compared to the basic, offline trained network and also compared to the handcoded controlled learning rules, which are infact a form of manually created code comparable to an AI using a state machine.

Robustness

The AI was able to deal with a human player as good as it was with the computer controlled one. The AI also adapted to changes in the damage rates.

Efficiency

Depending on the test and algorithm used the AI clearly showed adaptation in very short time. Sometimes during the course of only 5 to 6 agent or player deaths. The longer term effect of the supervised and reinforcement learning algorithms normally took effect after a 5 to 6 timesteps, which equal about 3 to 4 minutes in real time.

Clarity

This is a weakness of some of the examined algorithms. Sometimes early versions showed hard to understand behaviour and needed a lot of tuning to work as intended. The 0.5 rule algorithm could be seen as a good example. To be able to better deal with this, it would help to create tools helping to better monitor the relation between the in-

and outputs of the neural network(s) and also to implement more built in data collection.

Variety

The pure learning algorithms often created slightly monotonous behaviour, heavily favouring one or two of the 3 states. This is also due to the simple task and should be examined on more complex problems. The controlled learning algorithms always showed variety, albeit this was mainly the effect of the handcoded control part of them.

Consistency

The time it took the learning algorithms to converge on their usual behaviour was sufficiently consistent.

Scalability

The learning speed can be easily lowered and the controlled learning frequency can be adjusted to allow for dificulty scaling.

9.3 Outlook

Further uses

The methods represented throughout this thesis are also compatible to other, already better researched learning algorithms.

Evolutionary offline algorithms could be used to train neural networks in games, while methods similar to those represented here could then provide online adaptivity to make game AI more varied and interesting.

Furthermore the neural networks used during this thesis were easily compatible with conventional methods that produced input for them. Just one way of incorporating the highly flexible neural networks into an existing gaming AI. This divide and conquer idea is also represented by the usage of small interconnected networks rather than one big monolithic one. All these ideas should make it more attractive for game developers to finally start using more advanced AI techniques.

Neural network toolbox

Future work in this field could be the development of tools designed to easily create neural networks and adjust their parameters, and also to provide a better understanding of the design and tuning of neural networks. A better monitoring of the network variables could also greatly help in understanding the results produced by a network, allowing for easier adjustments. A toolset could also contain methods to automatically create a number of different neural networks, not only the standard multilayer feed forward one, to compare the effect of the different types of networks when used ingame.

Implementing the learning techniques into a real game

The only way to conclusively decide if the techniques introduced in this thesis are capable of creating good AI for commercial games is of course the implementation of some of the learning algorithms into a full game. This could either be done by programming a small game from the gound up and base its complete AI on the usage of a hierarchy of neural networks implementing online learning algorithms. Alternatively a published commercial game could be used as the basis for a modification, exchanging some part of its AI with learning neural networks. Some older games are available as open source and would be ideal for this. Instead a modification of a newer game could be tried. The difficulty here is that normally only a small part of the AI is accessible in such a case, often as scripts.

Bibliography

[AckleyLittman90]

David H. Ackley , Michael L. Littman. Generalization and Scaling in Reinforcement Learning. In Touretzky, D. S. (Ed.), *Advances in Neural Information Processing Systems 2,* pp. 550-557. San Mateo, CA. Morgan Kaufmann 1990.

[Bauckhage et al. 03]

Christian Bauckhage, Christian Thurau, and Gerhard Sagerer. Learning Human-like Opponent Behavior for Interactive Computer Games. *DAGM-Symposium 2003,* pp 148-155. Springer 2003.

[Boyan92]

Justin A. Boyan. Modular Neural Networks for Learning Context-Dependent Game Strategies. Department of Engineering and Computer Laboratory, B.S., University of Chicago 1992.

[BryantMiikkulainen03]

Bryant, B. D. and Miikkulainen, R. Neuroevolution for adaptive teams. In *Proceedings of the 2003 Congress on Evolutionary Computation (CEC 2003), volume 3,* pp 2194-2201. IEEE Press 2003

[Campbell02]

M.S. Campbell. Knowledge discovery in Deep Blue. In *Communications of the ACM 42(11)*, pp 65-67. Association for Computing Machinery 1999.

[Chan et al. 04]

Ben Chan, Jörg Denzinger, Darryl Gates, Kevin Loose. Evolutionary behavior testing of commercial computer games. In *Proceedings of the 2004 Congress on Evolutionary Computation (CEC2004), volume 1,* pp 125 – 132. IEEE Press 2004.

[Coulom02]

M. Rémi Coulom (2002). Apprentissage par renforcement utilisant des réseaux de neurones, avec des applications au contrôle moteur (Reinforcement Learning Using Neural Networks, with Applications to Motor Control). Laboratoire Leibniz-IMAG dans le cadre de l'Ecole Doctorale Ingénierie pour le Vivant : Santé, Cognition, Environnement.

[Fogel et al. 04]

Fogel, D. B., Hays, T. J., and Johnson, D. R. A platform for evolving characters in competitive games. *In Proceedings of 2004 Congress on Evolutionary Computation*, pp. 1420–1426. Piscataway, NJ. IEEE Press 2004.

[GallagherRyan03]

Gallagher, M. and Ryan, A. Learning to play Pac-Man: An evolutionary, rule-based approach. In *Proceedings of the 2003 Congress on Evolutionary Computation (CEC 2003), volume 4*, pp. 2462–2469, Piscataway, NJ. IEEE Press 2003.

[Graham et al. 04]

Ross Graham, Hugh McCabe & Stephen Sheridan (2004). Neural Networks for Real-time Pathfinding in Computer Games. In *Proceedings of ITB Research Conference 2004.*

[Haugeland85]

Haugeland, John (1985). *Artificial Intelligence: The Very Idea*. Cambridge, Mass. MIT Press 1985.

[Kaelbling et al. 96]

L. Kaelbling, M. Littman, A. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research 4*, pp. 237-285. San Mateo, CA., Morgan Kaufmann 1996.

[Ladebeck08]

Manuel Ladebeck September (2008). Applying Dynamic Scripting to “Jagged Alliance 2”. Diploma thesis. Department of Computer Science Knowledge Engineering Group. Technical University Darmstadt, Germany.

[LairdLent00]

Laird, J. E., and van Lent, M.. Human-level AI’s killer application. Interactive computer games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence* , pp. 1171-1178. Menlo Park, CA, AAAI Press 2000.

[Lockett et al. 07

Alan J. Lockett, Charles L. Chen, and Risto Miikkulainen. Evolving Explicit Opponent Models in Game Playing. *Genetic and Evolutionary Computation Conference 2007 (GECCO-2007)* pp. 2106-2113. ACM 2007

[Lucas05]

S. M. Lucas. Evolving a neural network location evaluator to play Ms. Pac-Man. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*. Piscataway, NJ. IEEE Press 2005.

[Michie61]

D. Michie . Trial and error. In S. A. Barnett and A. McLaren (Eds.), *Science Survey, Part 2*, pp. 129–145. Harmondsworth, U.K.: Penguin 1961.

[Miikkulainen et al. 06]

Risto Miikkulainen, Bobby D. Bryant, Ryan Cornelius, Igor V. Karpov, Kenneth O. Stanley, and Chern Han Yong. Computational Intelligence in Games. Gary Y. Yen and David B. Fogel (editors), *Computational Intelligence: Principles and Practice*, pp. 155-191. IEEE 2006.

[MoriartyMiikkulainen95]

David Moriarty and Risto Miikkulainen. Discovering complex Othello strategies through evolutionary neural networks. *Connection Science, 7(3),* pp. 195-209, 1995.

[Neumann05]

Gerhard Neumann. The Reinforcement Learning Toolbox, Reinforcement Learning for Optimal Control Tasks. Diploma Thesis. Institut für Grundlagen der Informationsverarbeitung, University of Technology Graz 2005.

[Pfau08]

Jens Pfau. Plans as means for guiding a reinforcement learner. Master's thesis. Intelligent Agent Laboratory Department of Information Systems, University of Melbourne 2008.

[Pollack98]

J.B Pollack and A. D. Blair. Co-evolution in the successful learning of backgammon strategy. *Machine Learning* 32(1), pp. 225–240, 1998.

[RevelloMcCartney02]

T. Revello and R. McCartney. Generating war game strategies using a genetic algorithm. In *Evolutionary Computation (CEC02), Proceedings of the 2002 Congress on Evolutionary Computation*, volume 2, pp. 1086–1091, Piscataway, NJ, IEEE 2002.

[Reynolds87]

C. W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioral Model in Computer Graphics. In *SIGGRAPH '87 Conference Proceedings* 21(4), pp. 25-34. MIT Press 1987.

[Richards et al. 97]

Norman Richards, David E. Moriarty, Paul McQuesten, Risto Miikkulainen. Evolving Neural Networks to Play Go. *ICGA 1997* , pp. 768-775.

[Schaeffer et al. 96]

Jonathan Schaeffer , Robert Lake, Paul Lu , Martin Bryant , Dr. Marion. Chinook: The World Man-Machine Checkers Champion. *AI Magazine* 17(1), pp. 21-29, 1996.

[Smith01]

Timothy Adam Smith. Neural Networks in RTS AI. Bachelor of Engineering Thesis. School of Computer Science and Electrical Engineering, University of Queensland, Australia 2001.

[Spronck05]

Pieter Spronck. Adaptive Game AI. Ph.D. thesis, Maastricht University Press, Maastricht, The Netherlands 2005.

[StanleyMiikkulainen04]

Kenneth O. Stanley, Risto Miikkulainen. Evolving a Roving Eye for Go. *Genetic and Evolutionary Computation Conference 2004 (GECCO-2004)* pp. 1226-1238. Springer 2004.

[Stanley et al. 05]

Kenneth O. Stanley, Bobby D. Bryant, Risto Miikkulainen. Evolving Neural Network Agents in the NERO Video Game. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG'05)*. Piscataway, NJ: IEEE, 2005.

[Sutton99]

Richard S. Sutton (1999). *Computational Learning Theory, 4th European Conference (EuroCOLT 99)*, pp 11-17. Springer 1999.

[Sutton98]

Richard S. Sutton & A. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press 1998.

[Sutton88]

Richard S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine Learning 3,* pp. 9-44. Kluwer Academic Publishers Boston 1988.

[Tesauro95]

Gerald Tesauro . Temporal difference learning and td-gammon. *Communications of the ACM,* 38(3), pp. 58-68. Association for Computing Machinery 1995.

[Tesauro94]

Gerald Tesauro. TD-Gammon, a self-teaching backgammon program achieves master-level play. *Neural Computation 6(2)* pp. 215–219. MIT Press 1994.

[Tresp et al.93]

Tresp V., Hollatz J., Ahmad S. (1993). Network structuring and training using rules based knowledge. *Advances in Neural Information Processing Systems 5,* pp. 871-878. San Mateo CA, Morgan Kaufmann 1993.

[Watkins89]

Christopher Watkins. Learning from delayed rewards. PhD thesis, University of Cambridge, England 1989.

[Wiering95]

Marco A. Wiering. TD Learning of Game evaluation functions with hierarchical network architectures. Department of Computer Systems Faculty of Mathematics and Computer Science , University of Amsterdam 1995.

[Wiering05]

Marco A. Wiering. QV(Lambda)-learning: A New On-policy Reinforcement Learning Algorithm. In *Proceedings of the 7th European Workshop on Reinforcement Learning*, D. Leone (editor), pp. 17-18, 2005.

[Yannakakis et al. 04]

Yannakakis G, Levine, J., and Hallam, J.. An evolutionary approach for interactive computer games. In *Evolutionary Computation (CEC2004) Congress on Evolutionary Computation, volume 1,* pp. 986–993. Piscataway, NJ, IEEE 2004.

[Yoshioka et al. 98]

Taku Yoshioka, Shin Ishii, Minoru Ito. Strategy acquisition for the game "Othello" based on reinforcement learning. In *The Fifth International Conference on Neural Information Processing (ICONIP'R98), Proceedings*, pp. 841-844. IOA Press 1998.