SmartAgent — Creating Reinforcement Learning Tetris AI

Samuel J. Sarjant

October 28, 2008

Abstract

For an NP-complete problem with a large amount of possible states, such as playing the popular videogame of Tetris, learning an effective artificial intelligence (AI) strategy can be hard using standard machine learning techniques because of the large number of examples required. Reinforcement learning, which learns by interacting with the environment through actions and receiving reward based on those actions, is well suited to the task. By learning which actions receive the highest rewards, the AI agent becomes a formidable player. This project discusses the application of reinforcement learning to a Tetris playing AI agent which was entered into the Reinforcement Learning Competition 2008 and presents the results and conclusions formed from its development and performance.

Contents

Chapter 1

Introduction

Ever since artificial intelligence could be created through programming code, programmers have always dreamed of creating software which has the ability to learn for itself. If a self-thinking software module (or agent) could simply take to a task and complete it with minimal help from a human expert, then the process of software design would be changed drastically, decreasing the time needed for programmers to spend on designing artificial intelligence algorithms.

Reinforcement learning is one of the first steps towards this self-learning, autonomous agent ideal. Reinforcement learning is basically the process of learning what to do in an environment by exploring and exploiting what has been learned. A large problem that a reinforcement learning agent faces is how to store the information it has learned efficiently and effectively while conforming to time and storage constraints. This project investigates the effectiveness of a reinforcement learning based agent within the environment of the popular videogame Tetris. However, to emphasise the need for self-learning, the environment is not fixed as ordinary Tetris. In every separate scenario, there are subtle changes to the standard Tetris environment that the agent must realise and account for when formulating its strategy.

Throughout its development, the agent presented in this project adopts and modifies strategies and algorithms found in other Tetris AI papers. The development of the agent involved many iterations, each building on the last. The final iteration was used to compete against other reinforcement learning agents in a worldwide reinforcement learning focused competition held in 2008. The results of this competition are one of the performance measures for the agent, shown as comparisons of agent performance between other competing teams. Another measure of performance is whether the iterations of this agent were logical, progressive iterations, so each of these agent iterations are compared against one-another.

In the final sections of this paper, a summary of conclusions drawn from the agent's performance is given and any possible future work is detailed to those that would continue development on the agent presented within.

Chapter 2

Background

Some concepts used in this paper require background and understanding to properly explain the agent's development. This project is not the first to use reinforcement learning or create a Tetris AI agent, and other approaches are briefly mentioned in this chapter. The competition upon which the agent was developed for is also explained here, as well as the machine learning suite known as WEKA which could have useful applications for future work.

2.1 Reinforcement Learning

In 'Reinforcement Learning: An Introduction' by Richard S. Sutton and Andrew G. Barto [13], reinforcement learning is defined as:

> Reinforcement learning is learning what to do — how to map situations to actions — so as to maximize a numerical reward signal

In essence, it is a process of independent learning, where the agent must discover which actions work best in which situations. The agent discovers these favourable actions by trying them out and observing any reward obtained from them. Rewards are given as a number, where a higher number is more favourable than a low one. The reward may not be an immediate reward — it may take a sequence of actions before a reward is received.

Reinforcement learning differs from supervised learning (as seen in standard machine learning) in that the agent does not know if an action taken is the 'right' action to take. The agent can only determine if an action is better than an alternative action by comparing the rewards seen on each. Although supervised learning is a perfectly acceptable method of learning, sometimes it is difficult to define the 'right' action to take in certain environments.

Problems in reinforcement learning are commonly given as *Markov Decision Processes* (MDPs) [13]. A problem is an MDP if the state and action spaces are finite in number, also making it an *episodic* problem. If the problem does not have specific states or the actions are not finite, the problem is *continuous*.

The exit criterion for a problem is usually a goal state of some sort but can be many things. When an agent satisfies the exit criteria, the time it spent in the environment is known as an *episode*.

Commonly, the agent stores the knowledge of an environment in a structure known as a *policy*. This is a structure that maps state-action pairs to associated values. By recording the rewards seen from particular states when an action is performed, the policy can grow in size and thus increase the agent's knowledge base. Often the full reward is not explicitly stored within the policy, instead a function is used to update the values and give an estimate towards the *actual* reward. This is because a reward can often fluctuate, even if the state and action remain the same. The basic function for updating an action value is as follows [13]:

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$

The $StepSize$ can be any value, but usually something small like 0.1 works well. $Target$ is the reward received at the time of the update.

One of the key problems in reinforcement learning is on-line learning, or the problem of *exploration vs exploitation*. This is the problem of choosing whether to further explore the environment for better rewards or exploit a known reward path. The difference between on-line and off-line learning is that on-line learning gets no extra training time to test strategies, it must adapt on-the-fly. So, to get the maximal reward, an agent needs to find the high reward actions and exploit them at the same time. But to find these actions, it has to try actions it has not chosen before. The agent has some sort of method to go about this task, a simple method being ε-greedy action selection. With a probability of ε, the agent chooses a random exploratory action but the rest of the time plays greedily (chooses the action with the highest reward). There are more sophisticated methods, such as the one used by this agent, as seen in Section 3.1.2.

2.1.1 Successful Applications of RL

One of the most successful applications of reinforcement learning is TD-Gammon, developed by Gerald Tesauro, a backgammon playing reinforcement learning agent [15]. Backgammon is an ancient two-player game in which the players take turns rolling dice and moving their checkers in opposite directions in a race to remove the checkers from the board. A player wins when they remove all of their checkers from the board, and receive one point for their win. If a player removes all of their checkers from the board without letting their opponent remove any, they win two points. During play, a player can choose to increase the stakes with what is known as a 'doubling cube'. When a player chooses this, the stakes of the win double and the player receives double points if they win. Two additional factors make the game more complex. First, a player can 'hit' another player's checker if it is the only checker present in the column, sending it back to before the start. This attacked piece has to re-enter the field before

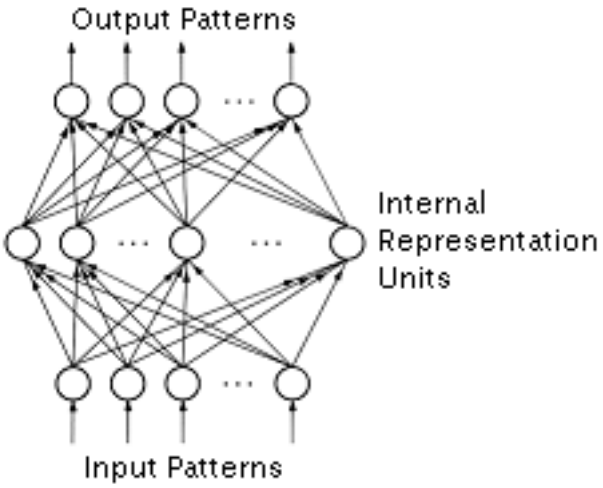

Figure 2.1: An illustration of the multilayer perception architecture used in TD-Gammon's neural network. Figure reproduced from [15].

any other of the player's piece can be moved. Second, it is possible to create 'anchors' by placing your own pieces on top of one another, blocking the other player from placing their piece on that column.

Because the game has such a large state space (estimated at 10^{20} unique states), supervised learning or playing via table look-up is infeasible. TD-Gammon plays backgammon by combining TD-learning with a neural network (Figure 2.1). TD-Gammon's method of learning works by letting the agent change the weights present in the neural network in order to maximise its overall reward. The reward in backgammon is received when one player has won or lost, with different levels of reward depending on the type of win and how the doubling cube was used. To appropriate this reward among the actions taken during the game, the agent uses *temporal-difference* (or TD) learning to change the network weights and grow the network.

Initially, the TD-Gammon agent knew nothing about the game, not even basic strategies. However, within the first few thousand games of the agent playing against itself, it quickly picked up some basic strategies (attacking an opponent's checker and creating anchors). After several tens of thousands of games, more sophisticated strategies began to emerge. Because the agent was able to grow the network, it was able to self-learn important features of the board and use them for better play. This agent was already as good as some backgammon algorithms around at the time, but not quite as good as master human players.

By incorporating some knowledge and basic strategies about the backgammon environment into the raw encoding, the agent became a much better player, rivalling many advanced human players. The agent became such a good player that it was able to create strategies never realised by backgammon experts be-

Figure 2.2: The seven tetrominoes used in Tetris.

fore. TD-Gammon is now considered to be one of the best backgammon playing entities in the world.

This application of reinforcement learning highlights several important concepts:

- The ability to learn new strategies and methods of play. The agent was able to create superior strategies never realised by human players before, something a hand-coded algorithm simply could not do because it is limited by the designer.
- The power of learning. The agent was able to formulate base strategies within a few iterations of self-learning and intermediate strategies after several ten thousand iterations. The agent also became the best backgammon playing AI players, something hand-designed algorithms have not achieved thus far.
- The problem of delayed-reward learning. Even though the reward in backgammon is not given until the game is completed, the agent was still able to quickly and effectively learn how to play.

Other notable applications of reinforcement learning are: KnightCap, a chess playing program that combines TD(λ) with game-tree search [2]; RoboCup, an international competition in which robots play soccer against one another, utilising reinforcement learning as one of the strategies [16]; Robot Juggling, a robotics control algorithm using reinforcement learning to learn how to juggle a 'devil stick' [12].

2.2 Tetris

Tetris is a popular videogame in which a random tetromino (from a possible seven tetrominoes) falls at a set speed down the field while the player moves it about to make it land in such a way that full horizontal lines of tetrominoes are created. Figure 2.2 shows the possible tetrominoes in Tetris, where each is a different combination of four joined units. The player can control the horizontal movement of the falling tetromino as well as being able to rotate the tetromino 90° in either direction, provided the action can be done without intersecting another block or the boundaries of the field. The player can also 'drop' the

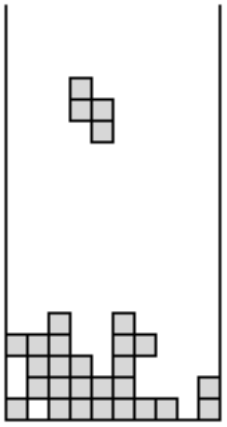

Figure 2.3: An example Tetris game in progress.

tetromino, which will cause the tetromino to go to the lowest valid position directly beneath its current position. When a tetromino has been dropped, or can fall no further, if the tetromino completes one or more horizontal lines, those lines are removed, the blocks above the completed lines move down the same number as lines removed, and the player receives a score based on how many lines were removed. When the tetromino settles, a new piece spawns at the top of the field and the process repeats. As the game goes on, the tetrominoes fall at a progressively faster rate, making it harder for the player to put the tetromino where they want it. The game ends when the field becomes too full for new pieces to spawn.

In the standard definition of Tetris, the field size is set as 10 blocks wide by 20 blocks high, the probability of each piece is equal, and the points scored when a line is made is proportional to the number of lines made squared. In most versions of Tetris where a human is the player, the next tetromino to come is displayed as additional information to help players set up strategies. This is known as two-piece Tetris. In one-piece Tetris, only the current falling tetromino is known.

Although the rules of Tetris are simple, playing the game effectively can be difficult, especially with one-piece Tetris. It has been proved mathematically that Tetris is NP-hard [6], which means that finding the optimal strategy for play cannot be solved in polynomial time, even if all the pieces are known in advance. This quality makes manually creating AI for Tetris difficult, which in turn makes it a good candidate for reinforcement learning techniques.

2.2.1 AI Approaches

Designing an AI agent for Tetris has been done before, in two separate ways: using a fixed strategy and basing it on self-learning. Some of the methods that have been invented and their respective results are discussed below.

Fixed-Strategy Approaches

A fixed-strategy approach is an approach that is tuned to one particular problem definition and behaves the same way each time throughout the scenario. In Tetris terms, these are algorithms designed by a programmer/s or trained beforehand to play a particular Tetris scenario (usually the standard one as defined in Section 2.2).

Although fixed-strategy approaches are usually quite effective on their respective MDP, they may not work so well on different MDPs. This is the main fault with fixed-strategy approaches and the reason why they would be unsuitable for the RL Competition (Section 2.3). Another problem with hand-tuned fixed-strategy algorithms are that they are only as good as the programmer who coded them. This is because the designer of the algorithm assumes they know the best way to play, when there could indeed be a better way.

The best one-piece algorithm[1] was created by Pierre Dellacherie and achieved

[1] Results from 2005 [7].

an average of about 650000 rows per game, with some games going up to 2 million rows. His algorithm was created around his ideas of optimal play and tuned as he observed the agent's performance. In a way, his algorithm was created using human reinforcement learning.

A two-piece algorithm created by Colin P. Fahey achieved excellent results, clearing over 7 million lines in a single game [5]. The additional knowledge of the next piece made his algorithm superior to Pierre's and holds the record of the best Tetris algorithm in the world[2].

Genetic (or evolutionary) algorithms [11] have been used to create effective Tetris playing agents as well. Although a genetic algorithm based agent can be used on a variety of scenarios, it requires extensive pre-processing to determine the correct 'chromosome' to use for play. Because these genetic agents require so much pre-processing and only use one fixed strategy during gameplay, they are regarded as fixed-strategy approaches.

'An Evolutionary Approach To Tetris' by Niko Böhm, Gabriella Kókai and Stefan Mandl [4] describes an optimal agent for playing two-piece Tetris which is created via the evolutionary algorithm process. Their algorithm works by defining the field as a group of 12 unique numerical features, and evaluating the field after both pieces are theoretically placed. The sum of the features determines the choice of move, with a higher sum being more desirable. Each feature had a weight associated with it which could be mutated during the reproduction stage of the algorithm. The sum was also determined by the *rating-function*, which could affect some of the feature weights in one of three ways: linear, exponential and exponential with displacement. For example, if the pile height feature (the current height of the pile) rose from two to four, using a linear rating-function would see this change equal to the pile height rising from 15 to 17. With an exponential rating-function, the pile height rising from 15 to 17 is worse than the height rising from two to four.

Because training a good agent for playing a game of Tetris can take a very long time, they reduced the field size to 6×12 to shorten the algorithm's running time. On the 10×20 field, which was only run once due to the large amount of time it took to complete, their agent made an average of over 1.3 million lines over five games [4]. Although these results are quite good for a non-hand-coded algorithm, it takes far too long (three months! [4]) to train the agent.

Learning-Based Approaches

A learning-based approach is an approach where the agent learns how to play the game by itself, with minimal instruction from the creator of the agent, during the course of gameplay. In Tetris, an agent will not be required to learn basic ideas — such as which command makes the piece move left, these will be given by the programmer — but will have to learn an overall piece placement strategy for the particular MDP.

[2]Results from 2005 [7].

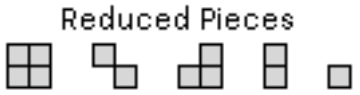

Figure 2.4: The reduced pieces used in Bdolah and Livnat's paper [3].

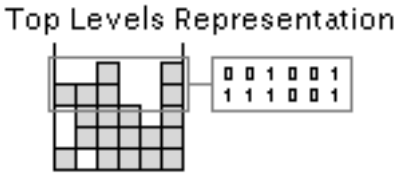

Figure 2.5: The well can be approximated into a smaller state using the top two levels.

The key advantage of learning-based agents is that they can adapt to a scenario automatically, without having to be redefined by the programmer. However, this adaptability often compromises performance, because it takes time for an agent to become specialised in a particular MDP. Also, because the agent learns how to play by itself, it may create effective strategies that a human player would not normally consider.

'Reinforcement Learning Playing Tetris' by Yael Bdolah and Dror Livnat [3] concerned a reduced version of Tetris, much like the paper it was modelled on [10]. Instead of the standard Tetris specification, some small changes were made. Instead of using pieces made up of 4 units, possible pieces were constrained within a 2×2 grid creating 5 unique pieces of differing unit counts (Figure 2.4). Because the pieces were smaller, the field size was also reduced to a width of 6 and an infinite height. Because a computer is playing to learn, rather than to win, placing a height limit on the well would only slow learning.

They chose the strategy of reducing the state space to the top two rows of the relevant part of the well (Figure 2.5). Where M = height of the highest column, this representation only holds the information of row M and $M-1$, stored in a binary 2D array. This results in $2^{2*6} = 4096$ states. This figure was further reduced by utilising the symmetry of the pieces and states into only 2080 states.

The agent used these states to record rewards received when a piece was placed onto them. This information could then be used by the agent to make educated decisions on where to place the pieces for maximal reward.

'Adapting Reinforcement Learning to Tetris' by Donald Carr [5] took the same direction as this project by initially drawing upon Bdolah and Livnat's paper and expanding upon the concepts covered.[3] Similar to the project described in this report, Carr realised that the top-two-rows strategy in Bdolah and Livnat's paper could not be used in full Tetris so, like this project, he

[3]Note that this paper was not discovered until later into the project and the similarities between the two projects are purely coincidence.

represented the field as a group of contoured substates. However, Carr mainly worked with the reduced Tetris specification as given by Bdolah and Livnat's paper and Figure 2.4. Although he obtained good results for the reduced version of Tetris [5], when he used full Tetris pieces, the results were dismal, worse than an average human player [5].

'Learning Tetris Using the Noisy Cross-Entropy Method' by István Szita and András Lörincz [14] was a paper which created an intelligent agent to play the full standard version of Tetris. One of the teams (Loria INRIA - MAIA) in the RL Competition (Section 2.3) used this paper as part of their agent's design. This paper's approach involved maintaining a distribution of solutions to the problem, rather than a single strategy. The agent chooses actions by evaluating the current tetromino in every possible position and rotation in the well and choosing the best one, the position that gets the largest $V_w(s)$ as given by the value function:

$$V_w(s) := \sum_{i=1}^{22} w_i \phi_i(s)$$

where $\phi_i(s)$ is the value of a particular feature of the well, from 22 different possible features (on a standard ten-wide well). The 22 features comprise the maximal column height, individual column heights, differences of column heights and the number of holes. w_i is the weight of the particular value, which changes as the agent plays.

The way the agent learns is by adjusting the weights to suit the MDP properly using the cross-entropy method. This method works by starting with an initial family of weights — likely not the ideal weights — and modifying them towards the ideal weight by using the mean value of the family and a calculated bias. Over time, this family becomes a near-ideal set of solutions to the problem, producing an effective agent.

A problem noted in the paper is that the agent often settles on a local maximum of solutions, stifling it from learning further. To address this, they injected noise into the algorithm to offset the agent from local maxima.

2.3 RL Competition 2008

The (Second Annual) Reinforcement Learning Competition 2008 is a competition designed to foster research on reinforcement learning in a competitive environment. The competition began on October 15$^{\text{th}}$, 2007 and finished on July 1$^{\text{st}}$, 2008. Throughout the competition, the participants could design and test their agents against training, proving, and later testing, environments. In the training stage, the participants can design and test an agent on local training software provided by the competition for immediate results, but this does not provide a good measure of agent performance. In the proving stage, the participants can 'prove' their agent on the competition server to see how well it performs over multiple MDPs against other agents in the competition. The testing stage is similar to the proving stage except that the agent can only

be submitted once and the MDPs have been slightly changed to add in some uncertainty.

There were six different domains in the 2008 competition:

Mountain Car For the mountain car problem, the agent must drive an underpowered car up a steep mountain slope. The goal for the agent is to get as high as it can up this mountain by driving at a particular acceleration. The car's movement is described by two continuous variables, position and velocity, and one discrete acceleration variable.

Real-Time Strategy In the real-time strategy (or RTS) problem, the goal is to destroy the agent's opponent's base. To do this the agent must effectively use the two types of units available to it: workers and marines. Workers can gather materials by finding mineral patches, which are used to train further workers or marines. Marines are used for combat, which are ideal units for destroying the agent's opponent's base or units. The agent's actions are defined as the action chosen for each of the agent's units.

Helicopter Hovering In the helicopter hovering problem, the agent must control a helicopter without crashing it. Hovering a helicopter is a challenging problem with high dimensional, asymmetric, noisy, nonlinear, non-minimal phase dynamics. This problem leaves no room for error, due to the steep penalty for crashing.

Keepaway Soccer The Keepaway Soccer problem is similar to soccer, except that one team attempts to keep the ball within a limited region on the field while the other team attempts to force it from that region.

Polyathlon The Polyathlon problem is a broad problem in which the agent needs to be able to learn anything without prior knowledge or pre-training. In the Polyathlon, the programmer is unable to tune the agent to the domain, making this problem the ultimate self-learning problem.

Tetris As explained in Section 2.2, Tetris is a falling blocks game where the player tries to achieve the highest score possible while keeping the height of the pile down. The RL Competition has slightly differing rules for this version, formalised by Van Roy [8]. At each step, the agent chooses an action, the action is performed and the piece falls by one. The pieces only fall when the agent chooses an action and only one action can be performed per step. The available actions to perform are: move piece left, move piece right, rotate piece left, rotate piece right, drop piece, or do nothing.

An interesting change to the standard Tetris specification is that the agent is not penalised for losing a game.[4] The agent's only reward value occurs when it makes a line/s. This reward can be a higher order polynomial ad

[4]This was not realised until late in the agent's development.

not just linear to the number of lines made, depending on the MDP. So the emphasis in this version is on making lines, not survival. This was likely chosen because a computer is not affected by the increase in speed that normally makes it harder for a human to play Tetris. Also, it could be because the emphasis in the competition is on learning, and by letting the agent continue learning without punishing it for losing, the agent can learn at a fast rate.

Another change, related to this idea, is that the agent is tested over a set number of *steps*, rather than *episodes*. A step in Tetris is equal to a single action, or a piece falling by one unit. Once again, this rule does not penalise agents that fail frequently. But it does change the typical strategy of playing Tetris as shown by the winning team's strategy (Section 4.1).

Each of these domains are modified in a small way during proving and testing to make using a fixed-strategy agent an impractical approach. Because of these changes, reinforcement learning is an ideal method for creating an agent, but incorporating previous knowledge about the domain into the agent (except for Polyathlon) is allowed. By doing so, the agent would only have to learn the strategies for the problem, rather than learning how to interact with the problem itself.

The results and discussion of the Tetris domain of the competition are in Section 4.1.

2.4 WEKA

WEKA (Waikato Environment for Knowledge Analysis) [18] is a machine learning software suite used primarily for data mining tasks (Figure 2.6). WEKA was developed at the University of Waikato and is freely available for download under the GNU General Public License.[5] WEKA supports several standard data mining tasks: data preprocessing, clustering, classification, regression, visualisation, and feature selection.

Data mining is the process of sorting through large amounts of data and finding relevant patterns or information. Data is typically characterised as a group of *instances*, where each instance is made up of several *attributes* and a class attribute. The class value is what the machine learning algorithms are attempting to predict based on the attributes of the instance. For instance, a common dataset used to explain data mining is the *iris* dataset. In this dataset, the set of attributes are $\{sepallength, sepalwidth, petallength, petalwidth\}$, where each attribute value is numerical and the class attribute is the type of iris (Iris-setosa, Iris-versicolor, Iris-virginica). Machine learning algorithms attempt to identify which ranges of values in each attribute define each type of iris flower by learning from a 'training set,' which is a percentage of the entire dataset. The accuracy of an algorithm is calculated by testing its model on a separate 'testing set' and recording how many errors were made.

[5] Go to http://www.cs.waikato.ac.nz/ml/weka/ for the download

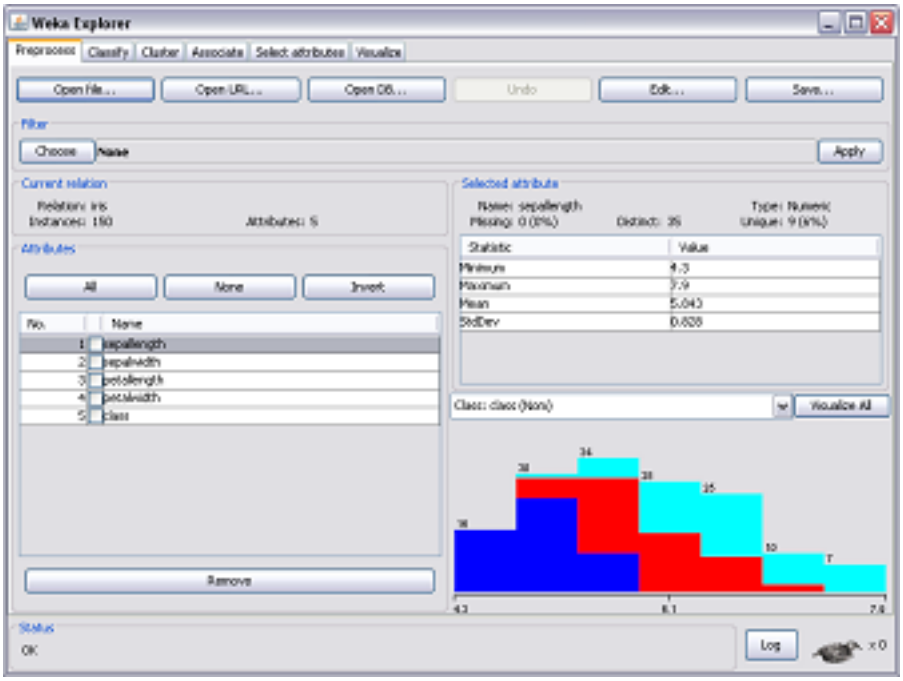

Figure 2.6: A screenshot of the WEKA 3.4.12 Explorer window

Because WEKA is open source software, existing filters and algorithms can be modified or completely new ones can be created and added to the suite. WEKA already contains a wide range of machine learning algorithms, some built to suit standard attribute-instance data files, others able to take on more obscure data files, such as relational data files.

Although WEKA contains many algorithms and tools for various situations, it does not contain any reinforcement learning algorithms.[6] This is probably because the input for reinforcement learners (a reactive environment) is fundamentally different to that of supervised learners (attribute-instance learning). There is room for a reinforcement learner because technically, given a complete data file that contains every state, it could be used to simulate an environment in which an agent can explore, returning rewards given by the resulting state instance.

[6] As at WEKA v3.5.8

Chapter 3

Implementation and Development

The agent (known as SmartAgent in the competition) went through multiple design iterations before it competed in the RL Competition, incorporating concepts and ideas from other papers at each stage. Through each iteration, the agent's performance got progressively better, with some iterations being more effective than others. In the later stages of the agent's development, some external tools were used to assess and improve the agent's performance. In this chapter, the details of each design iteration and the tools used in the later stages will be explained, as well as reasons for each design choice.

3.1 Agent Development

3.1.1 Initial Designs

The initial learning strategy was inspired by the paper 'Reinforcement Learning Playing Tetris' by Yael Bdolah and Dror Livnat [3] (Section 2.2.1).

Because their strategy kept the total amount of stored information down by approximating the field into a two-row bitmap, rather than the naive approach of storing an entire Tetris field, it was used as a base point for this project's agent. However, this project deals with the full specification of Tetris, which uses pieces made up of four units, so only looking at the top two rows would be inadequate. The agent would have to look at the top four rows of the well to get the full amount of required information. Because the full specification also has larger, more complex pieces, the state space could not easily be reduced symmetrically (it is possible to do so by swapping symmetrical pieces with one another, such as L-piece for J-piece). Furthermore, the number of columns in the competition can be anything larger than six (though usually less than 13). The state space grows exponentially with the number of columns.

Using a standard Tetris well specification (with ten columns), the number

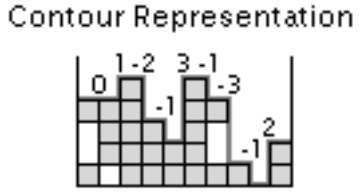

Figure 3.1: The field can be approximated as an array of height differences between columns.

of states in this modified top four rows strategy gives $2^{4*10} = 1{,}099{,}511{,}627{,}776$ states, which is beyond unacceptable because it would not only take a very long time to learn each state's expected reward, but also because storing that amount of states would need about 4TB worth of space. Another flaw is that this method under-represents the entirety of the field by only looking at the top X rows of the field, when there may be more useful features beneath.

Clearly a different design was required. In Bdolah and Livnat's paper [3] there was another approach mentioned which was not as successful on their reduced Tetris, but may be better on the full Tetris specification. At the least, it would have a smaller state space than the previous strategy.

As seen in Figure 3.1, they represented the field as an array of height differences between neighbouring columns, which still holds enough information for an agent to make a good decision, disregarding 'holes'. Because a height difference of three is no different than a height difference of 30 to the agent (only the I-piece (Section 2.2) could be effectively used in these situations), the height values can be capped between $\{-3, \ldots, 3\}$ to keep the state space down.

Using this representation, the number of states on a standard Tetris field is $7^{10-1} = 40{,}353{,}607$ states, which is still too much for effective learning. Even reducing the state space by decreasing the possible values to $\{-2, \ldots, 2\}$, the number of states is $5^{10-1} = 1{,}953{,}125$ states.

3.1.2 V1.0: Contoured Substate Representation

As the method of representing the field as an array of height differences still yields too many states, SmartAgent version 1.0 used a contoured substate representation instead. Because each Tetris piece, or tetromino, can be of a maximum size of four, the agent need only store information how a piece fits within four particular columns. By only storing parts of the field, as width-four-substates as seen in Figure 3.2, the policy size can be shrunk down to a small size.

When each substate is represented using the aforementioned height difference method (with the values capped between $\{-2, \ldots, 2\}$), giving three values per substate, the number of states is only $5^3 = 125$ states. With a total of 48 different piece configurations within a substate, the total size of the policy becomes $125 \times 48 = 6000$ individual elements. On a standard ten column field, a total of seven unique substates could fit onto it, with the substates intersecting (but not overlapping) one-another. Because the substates intersect one-another,

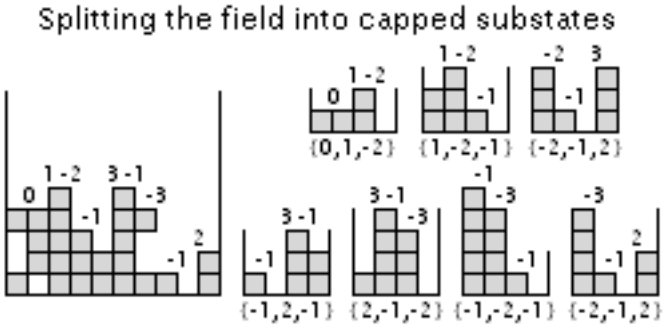

Figure 3.2: By only storing parts of the field and restricting the values, the policy size can be kept low while providing enough information to learn.

when a piece falls, it could fall within multiple substates and the agent can take advantage of this and record the reward for each substate, resulting in faster learning. For example, in Figure 3.2, if a vertically-aligned S-piece (Section 2.2) was placed in the hollow near the centre so it fits perfectly, it would be captured by three substates: $\{1, -2, -1\}$, $\{-2, -1, 2\}$ and $\{-1, 2, -1\}$.

Another benefit of subdividing the field is that during piece placement, parts of the field can be ignored as possible positions if they are clearly unsuitable candidates (perhaps that part of the field is too high or unreachable with the number of piece movements). A substate's height is given by the height of the lowest point in the substate. If a substate is at least four rows higher than the lowest substate, then it is removed from possible substates, as it can likely be ignored as a possible substate for placement due to being too high.

Using this strategy, the agent can learn how to play Tetris by placing tetrominoes at positions on the field and updating the policy by recording where the tetromino landed in particular substates with the reward received from that move. The function used for storing reward is a simple Monte-Carlo update method [13], shown here:

$$V(s,t) = V(s,t) + \alpha(reward - V(s,t))$$

where s is the substate, t is the tetromino, including its position and rotation with regards to the substate, α is the rate of learning (usually 0.1), and $V(s,t)$ is the value of the substate-tetromino pair within the policy, with an initial value of zero.

Once a knowledge storing strategy has been established, the general algorithm of the program can be defined:

1. The piece enters the field, is identified and its position captured by the agent.

2. The agent chooses between placing it exploratorily or exploitatively (see Section 2.1).

- **Exploration**

  (a) Cool the 'temperature' (see below).

  (b) Choose a random substate from the current field substates.

  (c) Choose a random valid location and rotation within the chosen substate, move the piece to that location and drop it.

- **Exploitation**

  (a) Choose the best substate s on the field by using the policy reward lookup function $V(s, t)$ where t is the current piece.

  (b) Rotate and move the piece to the given substate and drop it.

3. Observe the reward (even if the reward is zero) and store in the appropriate piece-substate pair/s in the policy.

In the competition version of Tetris, an agent has to choose an action each step. In order to place a piece in a goal position, a series of actions are required to move and rotate the piece into the correct position. Within the algorithm, moving a piece to a goal position is achieved by an automatic helper function that simply moves, rotates, and drops the piece to the position in a minimal number of steps.

The agent still has one more problem to take care of: how to properly explore the environment while maintaining a high reward rate. Initially the agent was given a simple ε-greedy strategy (a fixed probability of choosing exploratorily) but because it needs to exploit high reward paths more when its knowledge base is sufficiently large, this strategy was changed to something a little more flexible.

Simulated annealing [11] is an algorithm based on the metal cooling process known as 'annealing,' which is the process of heating glass or metal to a high temperature and then gradually cooling it. It is a proven optimisation technique [13] that can be applied in many situations by starting with a high temperature and then gradually 'cooling' that temperature over time. In this situation, that temperature represents the probability of choosing an exploratory action, which starts at 1.0 (100%) and this is gradually 'cooled' or decreased by a factor of 0.99 each time the agent chooses to make an *exploratory* move. Therefore, as the 'temperature' decreases, the probability of making an exploratory move decreases and thus, because cooling only occurs when an exploratory move is made, the rate of cooling decreases. In summary, the probability of choosing exploratorily is initially quite high, but as it decreases, the rate of cooling decreases as well. This strategy allows the agent time to explore the domain while slowly settling on a greedy style of play.

3.1.3 V1.1: Variable Sized Substates

The first version of SmartAgent played in a simple manner — when exploring, it chose a random substate and a random piece orientation within that substate. But even though high substates were removed from the possible substates to

choose from, the agent continued to place pieces on these high points regardless. This was a major problem, because the piece could be placed in such a way that it hangs over an empty space, creating a 'cave' sort of structure, which is a problem for the contour representation of the field, which only looks straight down and ignores any empty space beneath a filled space.

This problem was caused by an inherent flaw in the substate strategy: each substate was of a fixed length (four) and the substate's height was determined by the lowest point in the substate. As shown in Figure 3.3, the height culling works on the leftmost substate, which is at least four rows higher than the lowest substate. However, the lowest substate also happens to contain a high column, but is still considered valid because it also contains the lowest point. And the middle substate is clearly an unsuitable substate, but is still valid because its lowest point is less than four rows above the lowest substate.

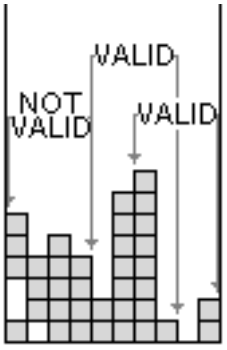

Figure 3.3: Substates are still (erroneously) considered valid if at least one column is less than four rows higher than the lowest column.

To address this issue, the size of a substate needs to be of a variable size, so that all columns within the substate are of roughly the same height. So instead of using the fixed-size substates, variable-sized substates are used, their lengths determined by the differences in height between neighbouring columns, with a maximum length of four. When scanning the field for substates, if the difference in height is three or more, the state is cut and a new substate is started after the height difference. Using this method, the substates created from a field may intersect one-another if they are both of size four, or will represent smaller parts of the field if they are of size three or less.

One problem with this new representation is that substates of width one will be undefinable, as there is no height difference between columns. For this purpose, three different special states have been created:

- A well at the beginning of the field (i.e. contour array $\{4, \ldots\}$).
- A well at the end of the field (i.e. contour array $\{\ldots, -4\}$).
- A well within the field (i.e. contour array $\{\ldots, -4, 4, \ldots\}$).

These three special states are all ideal candidates for the I-piece (Section 2.2) to be placed into. There are other types of one-wide substates ($\{\ldots, 4, 4, \ldots\}$), but they are useless to the agent and can be ignored.

By introducing variable-sized substates, the total number of possible substates has grown as well. The exact number is equal to:

$$numStates = \underbrace{(5^3 = 125)}_{\text{size 4 substates}} + \overbrace{(5^2 = 25)}^{\text{size 3 substates}} + \underbrace{(5^1 = 5)}_{\text{size 2 substates}} + \overbrace{3}^{\text{size 1 substates}} = 158$$

This small disadvantage is offset by an opportunity variable-sized substates create: superstate updating.

Superstate Updating

Given a substate of size three with the values of $S_3 = \{a, b\}$, the same substate can be seen in larger substates $S_{4a} = \{a, b, c\}$ and $S_{4b} = \{c, a, b\}$, so therefore any rewards gained by S_3 can also be applied to S_{4a} and S_{4b} with some small changes to the landed tetromino's relative position. This 'superstate' updating can be done to substates of size two as well, applying rewards to all size three and four superstates. Special substates can also benefit by rewarding mid-field special states whenever an edge special state is rewarded.

Given this method of learning propagation, to learn at a maximum rate, the agent should only capture the minimal surrounding substate of the landed piece and let the superstate updating store the reward in all necessary substates. This strategy of multiple substate updating speeds up the learning process and makes for a better overall agent.

3.1.4 V1.2: Semi-Guided Placement

The other issue present in V1.0 (and V1.1) was that during exploration the agent would place the pieces randomly on the field (with some bias towards lower areas). This is a major problem because it inhibits the agent's learning. Because of the nature of the Tetris environment, rewards are only given when a line is made. The problem is that the agent needs to know how to make a line. But to learn that, the agent needs to receive reward from making a line in the first place. It is a vicious cycle in which the agent may make a line by random chance, but the odds of doing so are slim.

To break from this cycle, the agent needs a little help placing pieces when exploring to receive rewards. By guiding the placement of pieces using the current piece's structure, there is a higher chance of receiving reward. In Figure 3.4 the current piece is an L-piece (Section 2.2) and according to the piece's structure, it would fit best straight down or to the far left. A horizontal I-piece (Section 2.2) is best placed on a flat plane, such as a fresh field, while a vertical I-piece can be placed anywhere, but is best used in the special states (Section 3.1.3).

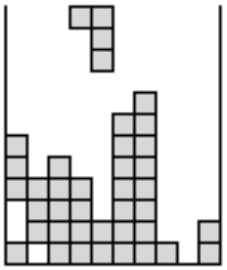

Figure 3.4: The current piece fits best directly beneath and to the far left.

Of course, the ideal position may not always be available on the current field, so if this is the case, a close approximation is used instead. The approximation is determined by changing the ideal contour structure $\{c_0, \ldots, c_n\}$ by a small amount at either end. Because the contour can be changed in two ways, the function gives two

new contours. The function is as follows:

$$f(\{c_0, \ldots, c_n\}) = \begin{matrix} \{c_0 + 1, \ldots, c_n\} \\ \{c_0, \ldots, c_n - 1\} \end{matrix}$$

If neither of these outputs are found on the field, the function is applied again to each output until one of three things occur:

1. The contour is found on the field.
2. The contour cannot be mutated any more due to the bounds of the representation ($\{-2, \ldots, 2\}$).
3. The recursion level goes deeper than two.

In the first case, if the contour is found, it is checked against all other found contours at the same recursion level and the lowest one is chosen. In the second case, the recursion stops. As a last resort, in all other cases, a random position and orientation is chosen on the field.

There is a special case for special states, as they cannot be mutated in this manner. If there are special states present, the lowest one is chosen. Otherwise, the piece is put next to the largest height difference in the field or next to a wall.

This method of using semi-guided placement greatly increased the rate of reward, and thus, rate of learning for the agent. Combined with the superstate updating from V1.1, the agent was quickly filling the policy with values and using them for better play. In the Tetris environment though, lines are made from more than one well guided piece. It requires several pieces to be put in the right places to make a line. Therefore, it would make sense to reward the pieces that helped make a line as well.

3.1.5 V1.3: Eligibility Trace

An eligibility trace [13] is a technique used in reinforcement learning for rewarding not only the action that directly received the reward, but also the actions leading up to that one. It is used because often a reward is not received from a singular action but rather from a sequence of actions. In Tetris, this is just the case.

The trace works by maintaining a linear fixed-length history of pairs of pieces and minimal surrounding substates (Section 3.1.3) where the piece landed (in this case, $length = 10$). When a reward is received, all the items in the trace receive a fraction of the reward, with regards to their age in the trace. So the older the pair, the less of the reward that pair receives. But a pair at the head of the trace (the action that triggered the reward) receives the full reward. The formula for linearly appropriating the reward is as follows:

$$R_i = \frac{trace_{length} - pos_i}{trace_{length}}$$

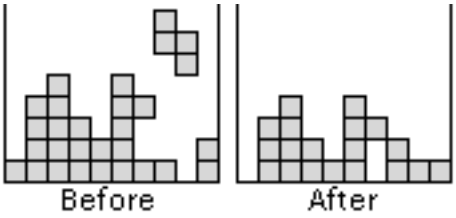

Figure 3.5: The before and after field states for an S-piece.

where pos_i is the position within the trace, starting at index zero.

By using semi-guided placement, superstate updating and an eligibility trace, the agent receives rewards at a fast rate and learns quickly. The combination of superstate updating and the eligibility trace is especially effective, as every element in the trace is of a minimal size and every superstate of that element receives reward when the trace is rewarded.

3.1.6 V1.4: Field Evaluation Placement

Although semi-guided placement increased the agent's performance, the overall performance was still rather lacklustre. The agent would still make major mistakes and place pieces in less than ideal positions. The problem was that the agent would only determine how pieces fit on the field without attempting to also fill lines. So, as an experiment, a new method of placing pieces was formulated, using Böhm, Kókai and Mandl's paper 'An Evolutionary Approach To Tetris' [4] as a key idea for this method.

Using field evaluation functions that determine the worth of a position after the piece had dropped worked well in that paper (and other effective Tetris agents). Also, Åsberg's paper states (regarding Tetris AI agents) that:

> "... a genetically evolved agent policy is definitely the way to go [1]."

By evaluating the field on a set of features, a piece can be tested in every possible position on the field and the afterstate that looks best according to the evaluation function will be used as the position to move the piece to.

Using the most relevant parameters from Böhm et al. and modifying them slightly, as well as using Tetris intuition, these five parameters were created:

Bumpiness This feature is a measure of the bumpiness of the field, or more accurately, the sum of the absolute height differences between neighbouring columns. Given a field state of x columns, the contour array would be defined as $\{c_0, c_1, \ldots, c_{x-1}\}$, where c_i is the height difference between column_i and column_{i+1}, the bumpiness feature is equal to $\sum_{i=0}^{x-1} |c_i|$. This parameter was inspired by and modified from the *Altitude Difference* and *Sum of all Wells* parameters. In Figure 3.5, the bumpiness value of the before state is 17. In the after state, the bumpiness value is equal to 13.

Chasms This feature is a count of the number of chasms present on the field. A chasm is defined as a 1-wide gap which is at least three units deep, i.e. a gap only a vertical I-piece (Section 2.2) could properly fit into. This value is squared to emphasise the importance of keeping this value to minimum. This value was inspired by and modified from the '*well*' parameters. In Figure 3.5, the chasm value in both before and after states is one, due to the chasm present on the left side of both states.

Height This feature measures the maximum height of the pile. An empty field has a height value of zero. This value is squared to emphasise the importance of keeping height down. This value is the same as the *Pile Height* parameter. In Figure 3.5, the height value of the before field is five. In the after state, the height value is four.

Holes This feature counts the total number of holes in the field. A hole is defined as an unoccupied unit which is underneath an occupied unit. This counts unoccupied units several units beneath an occupied unit. In another sense, this is a part of the field that cannot be 'seen' directly down from the top of the field. This parameter is the same as the *Holes* parameter. In Figure 3.5, both the before and after states have a holes value of two, even though the before field's holes are not completely surrounded by blocks.

Lines This feature is equal to the number of lines made after the piece has been placed. This value can only be between zero and four. This parameter is the same as the *Removed Lines* parameter. In Figure 3.5, the lines value is equal to one in the afterstate.

By combining the values of these parameters together, with the *Lines* parameter multiplied by negative one, a field 'cost' can be calculated for a particular afterstate. By testing the piece in every possible position and evaluating the afterstate, the best move can be calculated by placing the piece in the piece position with the lowest 'cost.' Because this strategy was originally an experiment, the policy was still recording how pieces fit into the substates for use in exploitative play.

Not all the parameters from Böhm et al. were used; several were shown to be of little use [4] while others were excluded in order to keep the size of the parameter set down: *Connected Holes* was roughly the same as *Holes*, it just counted adjacent holes as a single hole; *Landing Height*, which recorded the height of the fallen tetromino, was shown to be useless; *Blocks*, which counted the number of occupied units on the field, was also shown to be useless; *Weighted Blocks*, which counted the blocks but assigned weights depending on the height of the block, was shown to be semi-useful, but was chosen to be excluded in order to keep the number of parameters low; and *Row* and *Column Transitions*, which sums the occupied/unoccupied transitions horizontally and vertically respectively, measured roughly the same quality as *Bumpiness* and *Holes* and was therefore omitted as well.

Evaluating the field using the raw, unweighted parameters would have the agent place pieces in good, but not great positions. The agent would create too many holes. So, to fix this, some coefficients were added to the calculation to weight the importance of each feature. The values settled upon for version 1.4 are $\{1, 2, 2, 8, 2\}$ which multiply to each feature respectively.

This new strategy created a much better agent — when playing exploratorily. Because the agent was playing by the substate strategy during exploitative play (the substate data having been captured during the exploratory field evaluation phase), it would place pieces in average-to-poor positions. So version 1.4 of SmartAgent became a fixed-policy agent which always used the field evaluation strategy with the same coefficients. This sort of strategy is unacceptable for a competition with an emphasis on adaptability. The next iteration of the agent developed attempted to overcome this limitation.

3.1.7 V1.5: Mutating Parameter Sets

In Böhm et al.'s paper 'An Evolutionary Approach To Tetris,' the ideal agent was found after a series of mutations on the weights of the features. The effectiveness of a set of weights was able to be determined by how well the agent performed using those weights, or how long the agent was able to survive a game of Tetris. But in the competition, the agent needs to be able to learn faster than in a game-by-game approach — it needs to learn on-line.

In version 1.5 of SmartAgent, the standard evolutionary algorithm was reworked so that a set of weights would only have to be tested over a small number of steps to get an estimate of the effectiveness of the weights. Much like the genetic algorithm, the agent would maintain a set of 'chromosomes' (parameter sets: $P = \{w_1, w_2, w_3, w_4, w_5\}$) and these chromosomes would mutate independently or with each other to create new and hopefully better weights for the particular MDP of Tetris. A more formal description of the algorithm is given below:

1. Start with an initial parameter set (such as $\{1, 2, 2, 8, 2\}$) and run over N (typically $N = 20$) pieces, storing the total reward gained and the # steps used with the parameter weights.

2. Store the parameter set in the sorted policy, where the order of the parameter weights is determined by the total reward gained divided by the number of pieces the set was in use for.

3. Choose between exploration and exploitation based on the current 'temperature' (see Section 3.1.2).

- Exploration

   (a) Cool the 'temperature' (see Section 3.1.2).

   (b) 'Mutate' (see below) a new parameter set using the best parameter sets from the policy.

(c) Run this new parameter set over N pieces, storing the total reward gained during its use.

(d) If the mutant is not in the policy, insert it into the correctly sorted position. If it is, update the total reward and # pieces values, and recalculate its 'worth' and position in the policy.

- Exploitation

(a) Choose the best parameter set from the policy and run it for N pieces, updating the total reward and # pieces values, and recalculating the 'worth' and position in the policy.

4. Return to step 3.

As with the substate strategy, choosing between exploratory and exploitative actions is still done with simulated annealing, but the choice only happens every N moves. The policy in version 1.5 of SmartAgent is now made up of parameter sets with associated 'worths,' calculated by dividing the total reward gained while the parameter set was in use by the total number of steps the parameter set has been used for. As the number of steps the parameter set is tested on grows large, this worth approaches the actual value of the parameter set. To keep track of the most successful parameter sets, the policy is sorted by this worth, so the best sets can be used for mutation (see below). Because there are a large number of possible parameter sets, the size of the policy is capped at 20, discarding information about any parameter sets sorted beyond 20^{th} in order to minimise memory costs.

When a new parameter set is mutated from the policy, it is done by a random choice of one of the four methods:

Basic parameter mutation Creates a new parameter set by modifying a randomly chosen weight value from the best parameter set in the policy ($P_{best} = \{w_1, w_2, w_3, w_4, w_5\}$) by multiplying or dividing it by β (typically $\beta = 5$) so that $P_{mutant} = \{w'_1, w'_2, w'_3, w'_4, w'_5\}$ such that one of $w'_x = w_x \overset{\times}{\div} \beta$ and all other $w'_{y \neq x} = w_y$. The parameter set is then normalised so the smallest $w = 1.0$.

Mid-point child Can only be performed if the policy has more than one element. This mutation creates a new child from the top two parameter sets in the policy ($P_{bestA} = \{w_{A1}, w_{A2}, w_{A3}, w_{A4}, w_{A5}\}$ and $P_{bestB} = \{w_{B1}, w_{B2}, w_{B3}, w_{B4}, w_{B5}\}$) such that the new child is the average of the two parameter sets. Therefore, $P_{mutant} = \{\frac{w_{A1}+w_{B1}}{2}, \frac{w_{A2}+w_{B2}}{2}, \frac{w_{A3}+w_{B3}}{2}, \frac{w_{A4}+w_{B4}}{2}, \frac{w_{A5}+w_{B5}}{2}\}$. The parameter set is then normalised so the smallest $w = 1.0$.

Trend child Can only be performed if the policy has more than one element. This mutation creates a new child from the top two parameter sets in the policy ($P_{bestA} = \{w_{A1}, w_{A2}, w_{A3}, w_{A4}, w_{A5}\}$ and $P_{bestB} = \{w_{B1}, w_{B2}, w_{B3}, w_{B4}, w_{B5}\}$) such that the new child follows the trend of

the two sets. If P_{bestA} is better than P_{bestB} (as determined by their order in the policy), the child would be defined as $P_{mutant} = \{w_{A1} \times \frac{w_{A1}}{w_{B1}}, w_{A2} \times \frac{w_{A2}}{w_{B2}}, w_{A3} \times \frac{w_{A3}}{w_{B3}}, w_{A4} \times \frac{w_{A4}}{w_{B4}}, w_{A5} \times \frac{w_{A5}}{w_{B5}}\}$. And *vice-versa* if P_{bestB} is better than P_{bestA}. The parameter set is then normalised so the smallest $w = 1.0$.

Crossover child Can only be performed if the policy has more than one element. This mutation creates a new child from the top two parameter sets in the policy ($P_{bestA} = \{w_{A1}, w_{A2}, w_{A3}, w_{A4}, w_{A5}\}$ and $P_{bestB} = \{w_{B1}, w_{B2}, w_{B3}, w_{B4}, w_{B5}\}$) such that the new child is one of the resulting children from a standard genetic algorithm crossover as shown in Russell and Norvig's 'Artificial Intelligence: A Modern Approach' [11]. For example, a pair of resulting children could be $P_{mutantA} = \{w_{A1}, w_{A2}, w_{B3}, w_{B4}, w_{B5}\}$ and $P_{mutantB} = \{w_{B1}, w_{B2}, w_{A3}, w_{A4}, w_{A5}\}$. Both parameter sets are then normalised so their smallest $w - 1.0$. Because this method nets two children, one of the children are used immediately while the other is stored for next time an exploratory action is chosen.

3.1.8 V1.6: Competition Version

This last iteration of development before the competition ended focused on optimising parameters and only making minor improvements to the learning strategy.

Before and After Field Evaluations One of the changes added into the algorithm to help the learning process was using the before and after field's scores to help judge a parameter sets worth. More specifically, before a parameter set is evaluated over N pieces, the base value of the field is calculated by evaluating the field with the unweighted parameter set $P_{unweighted} = \{1, 1, 1, 1, 1\}$. Then after the parameter set has been run over the N pieces, the field is evaluated again with the unweighted parameter set and the difference between the score ($before - after$) is added to the parameter set's worth. The after field is also taken when the agent loses an episode, so that the parameter set is punished for losing the game. There is one exception to this addition, which is the very first parameter set the agent runs when starting a new field. Because the field is empty, the agent cannot possibly improve upon it and so the current parameter set is not modified by the before and after field evaluations.

The benefits of using this addition to the parameter set worth strategy are not that the agent learns which parameter sets are good, these should already be obvious by the positive reward they receive for making lines, but that it learns which parameter sets are bad, as parameter sets that perform poorly will likely receive a negative reward for lowering the worth of the field. Of course, a parameter set that receives a terrible combination of tetrominoes during its evaluation will receive a negative reward, which is an unfortunate side effect of the strategy. Conversely, a parameter set

could receive an perfect combination of tetrominoes. In the end, these extreme cases average out and the relative difference between roughly equal parameter sets remains the same.

After the before and after evaluations were added to the agent's strategy, the performance did not appear to improve or degrade. Although, the performance was being judged based on how the agent appeared to play and not based upon experimental findings. Because this addition theoretically helped the agent, it was kept as part of the agent's strategy.

Emergency Play Often, when the height of the blocks get too high, the agent would break down and quickly fail the episode. When evaluating the field at every possible piece position, the agent does not check if the piece can be moved to the goal position in the number of available moves. Also, because the risk of loss is greatest when the field is near-full, it would be best if the agent could focus all its efforts on lowering the pile of blocks. This was the problem that led to the formulation of 'emergency play.[1]'

Emergency play is the strategy of having a separate policy with separate parameter sets for the upper half of the field. When the height of the blocks reaches the upper half of the field, the agent would switch into emergency play mode using the best emergency parameter set from the emergency policy. The initial emergency policy consists of a single parameter set tuned to focus on keeping height down (such as $\{1, 1, 32, 1, 256\}$). If the agent remains in emergency mode for some time, parameter sets are mutated from and added to the policy as usual.

Theoretically, the strategy looks appealing. However, there are some key issues that make using it problematic. Firstly, the agent should be focusing on keeping the height down naturally anyway. If an agent is performing well, it is doing so by keeping the height of the pile down, which is what emergency play is for, making it redundant. Secondly, the agent has to learn effective parameters for two separate policies. This makes learning good parameter sets take twice as long, if not longer due to the nature of the emergency policy. Because the emergency policy aims to lower the pile height down into the regular policy's domain again, the time that the agent spends learning in the emergency policy's part of the field is significantly lower than the regular policy's learning time. Finally, if only focusing on one or two parameters in the parameter set proves to be the best way to reduce the field height, why does the agent not learn this in the first place?

When this strategy was applied to the agent, it performed poorly and so it was discarded. As an alternative to emergency play, the *height* parameter was modified to be cubed instead of squared. This made the agent put more focus into keeping the pile height low in a smoother way than switching to emergency play mode could achieve, and consequentally also resulted in better play.

[1] Under the misassumption that survival was the main goal.

Maximum Well Depth The paper 'An Evolutionary Approach to Tetris' [4], describes a parameter called *maximum well depth*. This parameter measures the depth of the deepest well on the field, where a well is defined as a one-wide gap which has higher columns to either side. Although this parameter is similar to the *chasms* parameter used in this paper, it differs in an important aspect: it measures the depth of a chasm, rather than the total number of chasms. When this parameter was swapped for the *chasms* parameter, the agent's performance increased significantly. Although both parameters give useful information, *maximum well depth* was used in place of *chasms* because of its performance advantage over *chasms*. It is unknown exactly why it would give such a performance boost other than the fact that deeper wells are discouraged using *maximum well depth*, leading to a smoother field which is generally a more favourable field.

During this iteration of agent development, various parameters were tweaked, notably the initial parameter set of the agent. This was set close to the best parameter set for the standard Tetris field (10×20) because this setting was roughly the 'centremost' MDP for the Tetris MDPs the agent was being tested upon. Other parameters were optimised as well; such as the rate of cooling for the simulated annealing (Section 3.1.2); β, which controls the amount that the parameter sets mutate; N, the number of pieces to evaluate a parameter set on, among other minor parameters.

So, the state of the agent before the competition closed can be summarised as such:

1. The agent begins with a default initial parameter set ($P_{initial} = \{11250, 1, 10, 55000, 1\}$), which it runs over N ($N = 20$) pieces before mutating a new parameter set to run. Note that the 'temperature' is cooled at this point by the cooling rate (cooling rate $= 0.99$).

2. The current parameter set is used as a weights vector for features of the field and determines where the agent places the falling tetrominoes.

3. The initial parameter set is stored within the policy with its worth calculated as the reward gained over the N steps divided by N.

4. This new parameter set is mutated using the 'Basic parameter mutation,' where a random parameter weight is multiplied or divided by β ($\beta = 5$).

5. The new parameter set is run over N steps, recording the reward and the pre and post-field difference, and stored within an appropriate position in the policy depending on its worth.

6. An exploratory action is chosen with probability 'temperature,' otherwise the best parameter set from the policy is re-used for another N steps.

7. If an exploratory action is chosen, the 'temperature' is cooled again and a new parameter set is created by a random mutation from the set of four possible mutations.

8. Return to step 5 and continue *ad infinitum*.

3.1.9 Other Directions

As well as optimising the parameters as described in Section 3.1.8 during the final weeks of the competition, two substantial strategy changes were trialled that could be defined as separate agents. Although these separate agents were not used in the competition, they were kept for later version comparisons.

SmartAgent-Adaptive

The first altered agent was dubbed SmartAgent-Adaptive, due to the adaptive simulated annealing method that it utilises. Simulated annealing is a good strategy for slowly settling on an effective parameter set but it requires that the cooling rate is set to a good value initially otherwise the agent may either settle on a sub-optimal exploitative parameter set too early or explore for too long and lose effectiveness. Adaptive simulated annealing (ASA) [9] solves this issue by adjusting the rate of cooling depending on how well the agent is performing. So, if the agent is having trouble finding a parameter set for a particular MDP, the cooling rate is slowed, or even reversed in extreme cases. If the agent has found an effective parameter set, the cooling rate increases to capture this particular combination of weights.

In theory, this should work well. But, as with emergency play, in practise there are problems. Although ASA eliminates the need to set the cooling rate parameter, in turn it introduces new parameters, such as the threshold value for determining whether a parameter set is performing well (this could differ depending on the MDP, for example, an impossible MDP where the best worth is to lose at the slowest rate possible), and the rate of change for the cooling rate changes. These extra parameters made the algorithm quite volatile and the agent would often settle on an exploitative action quickly. Given more research, ASA may prove a valuable tool for the agent, but because the end date for the competition was too close, further work was discontinued.

SmartAgent-Probable

The second altered agent was dubbed SmartAgent-Probable, because the agent makes use of the observed distribution of the tetrominoes. The main difference between SmartAgent-Probable and SmartAgent V1.6 is that the agent performs a one-piece lookahead with probabilistic weighting. This means that the agent bases its moves on what the field would look like after two pieces have fallen. To do this, it simulates placing the current piece in each one of the possible positions, then for every piece (from the seven tetrominoes available, Section 2.2), finds the best position for them on the post-first-piece-fallen field. The probability comes in by multiplying this second piece's worth by the probability of that piece being the next piece. So now the best piece position is not decided

by how the field would look after the first piece has fallen, it is now a sum of the seven post fields after the second possible piece has fallen.

Initially the agent does not know anything about the probability distribution of the tetrominoes, so for the first hundred pieces, the agent plays assuming the probability of each piece is equal. During this time, the agent is monitoring the types of pieces falling and recording the observed distribution. Because this strategy uses these probabilities, the best position given by the SmartAgent V1.6 agent may differ from the SmartAgent-Probable agent due to the agent setting the field up for more probable pieces.

Although this strategy worked well enough, it was computationally expensive and therefore much slower. On a ten column field, the average number of unique positions a tetromino can be placed in is:

$$avnum(pos) = \frac{\overbrace{17 \text{ positions} \times 6}^{\text{I, T, S, Z, J, L-pieces}} + \overbrace{9 \text{ positions}}^{\text{O-piece}}}{7} = 15.86 \text{ (2 d.p.)}$$

However, using SmartAgent-Probable, the amount of positions to look at on a ten column field is equal to $15.86 \times 15.86 \times 7 = 1760.78$. This is about 111 times slower than the V1.6 strategy.

Because this strategy was so slow, it was not submitted as the final testing agent to the competition, even though it was likely a better playing agent than SmartAgent V1.6.[2] It was assumed that slow agents would be punished or disqualified but it was later found[3] that there was no restriction on agent processing time.

3.2 Other Tools

3.2.1 Tetris Workshop

Although the competition provided some software for agent training, it was very limited in its output. The console (or text-based) trainer would only record how many steps each episode lasted for and the GUI[4] trainer only displayed a graphical version of the field state, the number of lines completed, the current piece ID (if it was not apparent from the field state), the episode number, and the number of steps passed in the episode and in total. The main problem with these trainers was that the MDP details were not visible, so one had to either decode the task specification given to the agent or visually inspect the field for the MDP size, and just guess at the piece distribution and reward exponent. Both trainers also had annoying user interface problems as well. To start the trainer, the agent had to be loaded up in a separate process from the trainer

[2] Performance was only compared in basic measures such as visual comparison in the final weeks of the competition.

[3] About a month after the competition ended.

[4] Graphical User Interface

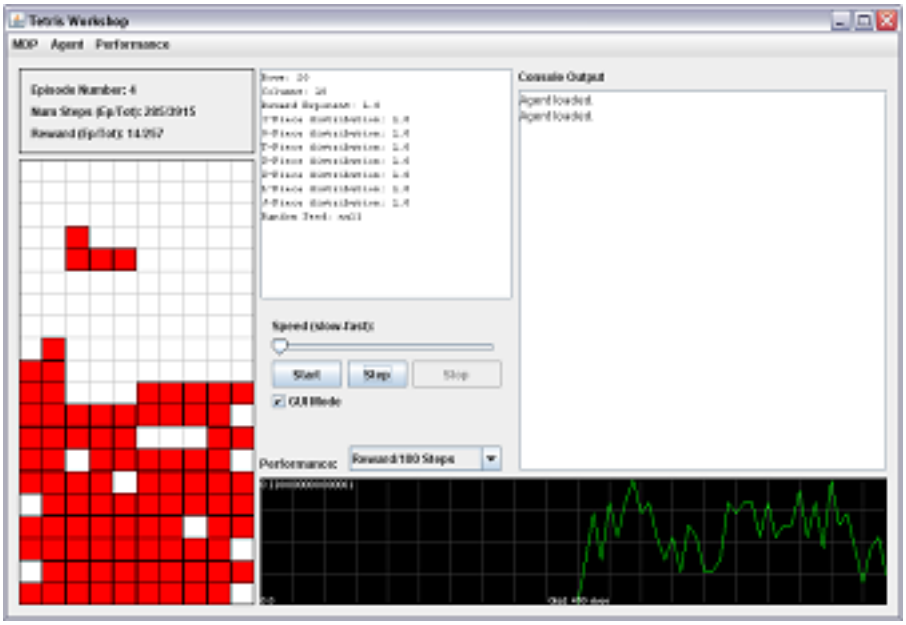

Figure 3.6: A screenshot of Tetris Workshop GUI.

and the two would communicate over a port. This process had to be repeated every time the agent was run, and quickly became tiresome.

The console and GUI trainers were insufficient for mass experiments and performance comparisons so a custom Tetris training environment program was created, dubbed Tetris Workshop GUI (See Figure 3.6). It has much more flexibility in setting up the environment and agent and provides useful performance graphs to measure and compare agent performances. Listed below are the key features of Tetris Workshop GUI:

On-the-fly agent loading As stated previously, to load an agent in the provided competition software, the agent must be loaded in a separate process and connected to a port. With Tetris Workshop GUI, an agent can be loaded with a typical load file dialog and it's instantly ready to go. This made the testing process much more streamlined and user friendly.

Customisable MDP settings A custom MDP can be created by specifying the size, piece distribution, reward exponent, and random seed (if any). If an MDP has a random seed, the same seed value will always spawn the exact same sequence of pieces for a game. This is very useful for comparing different agents against one-another, as at least the piece sequence will remain the same, lowering the variance in agent performance due to differing piece sequences.

Additionally, predefined competition MDPs and previously saved MDPs can be loaded in for use.

Dynamic multi-agent performance graph As well as providing numerical output of the agent's performance in the form of reward gained and steps passed, both per episode and in total, the performance is also shown on a dynamic performance graph at several scales. The agent performance can be viewed at powers-of-10 steps for the x axis, as well as reward per episode (with the x axis as the number of episodes) and total reward (x axis as total steps passed).

As well as displaying the current agent's performance, another agent's total performance or reward per episode performance can be loaded and displayed on the graph for comparison purposes.

Experiment mode When comparing multiple agents over multiple MDPs, doing it manually is a time-consuming and labourious chore. It is for this reason that an experiment mode was put into Tetris Workshop GUI, so multiple agents could be run over multiple MDPs and have their results saved to files for later comparisons. The experiment mode allows for setting up any number of MDPs and any number of agents, tested over a number of repetitions per MDP (to reduce variance) with a step number limit per repetition. The experiment panel also includes a progress bar, which states how much time has elapsed, what percentage of the experiment is complete, and an estimated time left value.

In the initial experiment mode, the agents were run over every MDP with r repetitions per MDP, producing a single file per agent which contains the average total reward gained over all MDPs. Although this produced accurate results for agent comparisons, because of the amount of averaging, the performance lines were very linear and did not appear to show much evidence of learning.

The output of the experiment mode was later changed to match the format of the competition's output graph, in which the final results file for each agent contained a concatenation of each MDPs performance joined together into one graph. For instance, if an experiment contained ten MDPs, the output graph would be a concatenation of the agent's performance for each of the ten MDPs (See Figure 3.7). This view shows how the agent did on each MDP, and shown together with other agents' results, lets the user see on average how each agent did on each MDP.

The Tetris Workshop GUI was created after the competition was over, so it was not used for the testing of each agent during the competition. It was able to use the predefined MDP details because at the time of its creation, the competition code had been released.

3.2.2 WEKA

One of the methods trialled for optimising the initial parameters in SmartAgent V1.6 used WEKA (see Section 2.4 for details on WEKA) to create a model for determining the initial parameter set ($P_{initial}$) for an MDP based on the

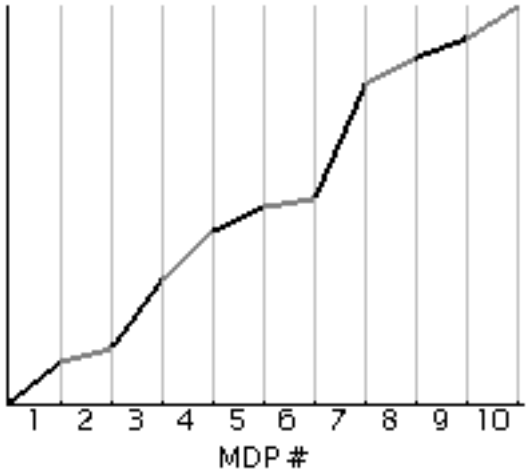

Figure 3.7: An example performance graph for an agent run over 10 MDPs using the later experiment mode.

dimensions of the field. If an initial parameter set starts off as a good play strategy, it is more likely that the agent will find an optimal parameter set quicker.

The data file used for training the parameter set creating model contained instances produced by SmartAgent's policy. Each instance was made from the best parameter set in the policy after a large number of steps had passed (over a million) for each predefined MDP. Ten instances were gathered from each MDP, so the agent had to re-learn an MDP ten times, resulting in a 200 instance (20 MDPs $\times$ 10 repetitions) data file. Each instance in the dataset was made up of nine attributes: the number of rows and columns, and seven attributes for the each piece's observed distributions (as the actual piece distribution was not given at this stage). The class values being predicted (as there were five of them) were the weights for each parameter in the parameter set.

Some elements of SmartAgent's main algorithm were changed for the purpose of data collection. The simulated annealing process was altered for maximal exploration of parameter sets (while maintaining the best parameter sets in the sorted policy) such that the temperature remained at 0.5 and never cooled (in essence, an ε-greedy strategy where $\varepsilon = 0.5$). Also, the initial parameter set weights were initialised as $P_{initial} = \{1, 1, 1, 1, 1\}$. These changes were made so that the best parameter set for each MDP could be found without bias towards the standard initial parameter set while avoiding premature convergence on a sub-optimal parameter set.

The results were dismal and proved to be of little use in determining the initial parameter set due to the model's low classification accuracy. This was probably due to the way the weights of the parameter sets were defined. Because the parameter sets were normalised so that the smallest parameter in the set was equal to one, parameter sets of roughly the same weightings would look wildly

different. For instance, an agent using parameter set $P_A = \{11250, 1, 10, 55000, 1\}$ will act in a very similar manner when using $P_B = \{11250, 0.5, 10, 55000, 1\} \Rightarrow \{22500, 1, 20, 110000, 2\}$, but the two resulting sets look quite different due to the larger numbers present in P_B. Furthermore, there is a huge problem with the attributes used in the data file. Although the number of rows and columns of an MDP are given, the piece distribution is not. Without this information, the model would be unable to make an accurate prediction until an observed piece distribution could be created. Once a sufficient amount of steps had passed to create an accurate piece distribution, the model could be used to predict a suitable parameter set. In the meantime, the agent would proceed as normal, creating its own parameter sets until the model was able to generate a useful parameter set. Further work could have gone into getting better results but was discontinued due to the time constraints of the competition.

Chapter 4

Experiments and Results

4.1 Competition Results

On July 1st 2008, the Second Annual Reinforcement Learning Competition ended. In the preceding month, competitors were able to submit their agents for a final testing run, which determined their placing in the competition. Between July 6th-9th, the 25th annual International Conference on Machine Learning took place in Helsinki, Finland, where the competition winners were announced and the results were published online.

SmartAgent was entered under the team name 'SmartCraft' and placed fifth in the Tetris domain of the competition. Throughout all ten MDPs, SmartAgent was roughly the fifth best agent, as shown in Figure 4.1. The SmartAgent version that was submitted was version 1.6. It was not known at the time that the agents did not have a time restriction for choosing moves, so SmartAgent-Probable was not used as the competition entrant, when it may have actually performed better than SmartAgent V1.6. The final numerical results are shown in Table 4.1.

Team	Score (Reward)
Loria INRIA - MAIA	6301413
The Cobras	6148543
NIPG	5796860
Enigma	5207626
SmartCraft	4967817
NJU RLOF	4323872
Saras	1359954
Massive Attack	1069379

Table 4.1: The final numerical score of the teams in the Tetris domain of the RL Competition 2008. SmartAgent had team name SmartCraft and finished fifth.

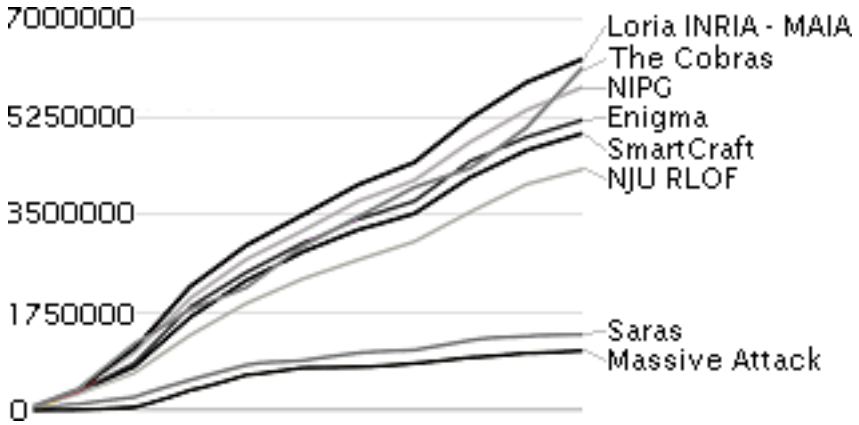

Figure 4.1: The results of RL Competition 2008. SmartAgent had team name SmartCraft and finished fifth. Figure reproduced from [17].

In Figure 4.1, the competitors follow the same reward trend, all except 'The Cobras' (second place winners of the 2008 RL Competition). The results shown for their team's agent appears to be quite different to the others and may have used a drastically different playing strategy to the others. When contacted and asked what their strategy for play was, their reply stated that they used:

> "...a very simple 1-step lookahead with a linear combination of a fixed set of features. The weights for each feature were learned using a policy search algorithm."

This strategy appears to be similar to SmartAgent-Probable, which uses a one-step lookahead and a mutating, weighted parameter (or feature) set. Perhaps if SmartAgent-Probable would have been entered, this project's team may have placed higher in the competition. In Section 4.2, the various SmartAgent versions will be compared against one-another in a simulated competition environment.

Team Loria INRIA - MAIA (the first place winners of the 2008 RL Competition) were also contacted and their agent utilised some unique strategies that perhaps gave them the upper hand over the rest of the competition agents. Their approach involved playing with a good fixed-policy agent while recording data such as piece distributions and reward exponents for a number of steps and then optimising the feature weights of the agent off-line (simulating an agent playing Tetris, allowing an optimal policy to be created 'behind-the-scenes'). This off-line optimisation does take some time, but the competition has no bounds on agent decision time.

Because the competition is judged over a fixed number of steps, an agent should place pieces in the minimal possible number of steps. Team Loria INRIA - MAIA took advantage of this restriction by putting some bias towards placing pieces in positions that require less steps.

They also used a strategy that altered their agent's actual performance, resulting in their agent appearing weaker on the competition leaderboard. During the proving stage, their agent would play as normal for 95% of the MDP, then go into 'suicide mode,' which changed the agent's strategy to achieving minimal reward (e.g. drop every piece immediately).

That team put a lot of effort into the competition by creating their own off-line simulator and testing environment which they ran many experiments upon to test which parameters worked best. According to their experiments, using a parameter set similar to the one this project used is good for survival, but not as good for obtaining maximal reward. A parameter set which takes into account all column heights and height differences achieves a greater overall reward, even if it does not survive as long.

4.2 Version Comparisons

After the competition had concluded, and the results were published, Tetris Workshop GUI (Section 3.2.1) was created to formally test the performances of the created SmartAgent versions against one-another. Initially, the experimenter produced a graph of the agent's average performance over all MDPs but this was later changed to produce a graph of all MDP performances concatenated together.

Because there is a large difference in performances between V1.3 and V1.4, all results figures have been separated into two groups: V1.1, V1.2 and V1.3; and V1.4, V1.5, V1.6, SmartAgent-Adaptive and SmartAgent-Probable. SmartAgent V1.0 was excluded because it was not stable enough to perform in the experiments. The separated graphs are able to show the performances more clearly than if all performances were joined on one graph.

4.2.1 Averaged Performances

The first experimenter of TetrisWorkshopGUI produced a single graph which showed the average performance of an agent over all MDPs. The agent also ran over each MDP a number of times to reduce variance and these repetitions were averaged as well. Because of the large amount of averaging, the resulting performance line appears quite linear and only curves if there is a definite pattern in an agent's performance.

Figure 4.2 shows the averaged performances of the first three (fully-working) agents: SmartAgent V1.1, SmartAgent V1.2, and SmartAgent V1.3. Each agent's performance was produced by running an agent over each MDP three times, with 24000 steps per MDP, where each MDP was controlled by a random seed to ensure equality of pieces. A total of 20 MDPs were used, numbered as MDP 0-19 from the competition predefined MDPs. The figure clearly shows the increases in performance with each iteration of the agent, although the results are not that impressive. Table 4.2 shows the actual rewards of the agents after 24000 steps. Although SmartAgent V1.3's average number of pieces per reward

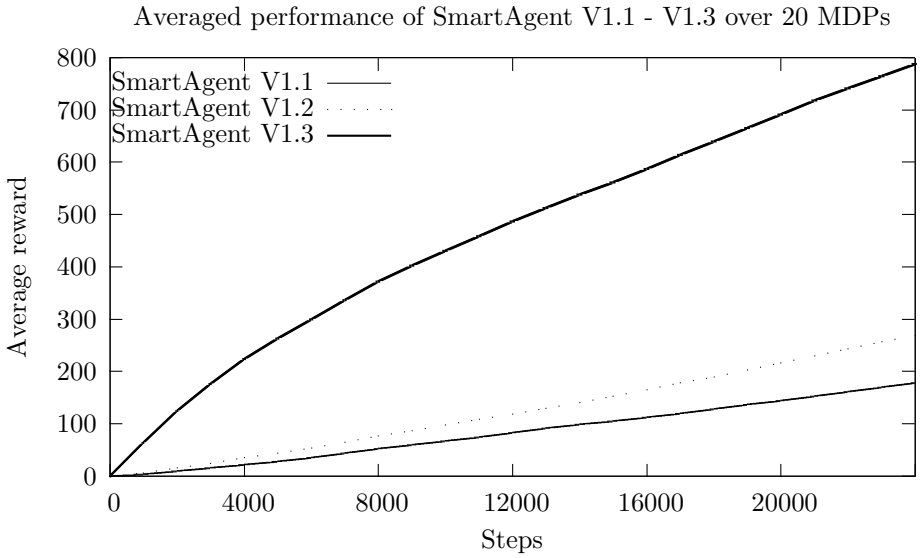

Figure 4.2: The averaged performance of SmartAgent 1.1, 1.2 and 1.3 over 20 MDPs with three repetitions per MDP. Each agent is run over 24000 steps three times per MDP and the performances are averaged.

Agent	Reward	$\frac{\text{Steps}}{\text{Reward}}$	$\frac{\text{Pieces}^{\dagger}}{\text{Reward}}$
SmartAgent V1.1	178.3	134.6	26.9
SmartAgent V1.2	269.1	89.2	17.8
SmartAgent V1.3	787.2	30.5	6.1
SmartAgent V1.4	2463.9	9.7	1.9
SmartAgent V1.5	2406.3	10.0	2.0
SmartAgent V1.6	2450.2	9.8	2.0
SmartAgent-Adaptive	2507.8	9.6	1.9
SmartAgent-Probable	2581.6	9.3	1.9

† Assuming it takes an average of five steps to place a piece.

Table 4.2: The averaged reward gained after 24000 steps over 20 MDPs with three repetitions and other rough calculations of performance. Clearing multiple lines and reward exponents are disregarded.

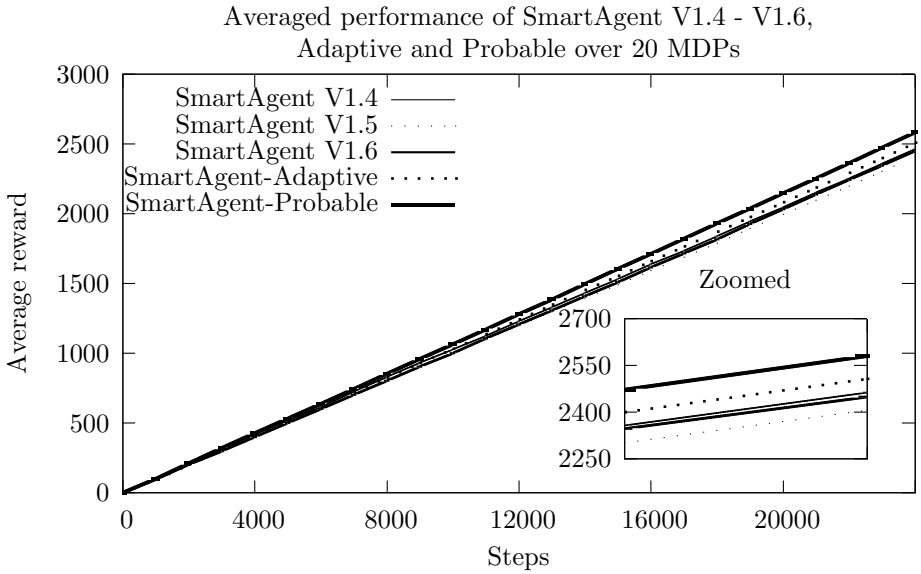

Figure 4.3: The averaged performance of SmartAgent 1.4, 1.5, 1.6, Adaptive and Probable over 20 MDPs with three repetitions per MDP.

is not terrible, it is not sufficient to keep the agent alive, because on the average ten column field, the minimal number of pieces required for a line is three, and placing about six pieces before clearing a line will cause the pile height to grow beyond the field.

An interesting feature of Figure 4.2 is SmartAgent V1.3's curve. It starts out with a steeper gradient than V1.1 or V1.2, but gradually becomes shallower, implying that the agent becomes less intelligent than when it started. Because this degrading in performance is not present in SmartAgent V1.2, the transition from semi-guided placement to exploitative substate play via the simulated annealing process is probably not the cause. SmartAgent V1.3's key feature was the inclusion of an eligibility trace to reward a history of piece-substate pairs, which is probably the reason for this behaviour. Because the trace awards a fixed-length history of piece-substate pairs, 'bad' movements are awarded if a later move causes a reward. Even if they are only awarded a small fraction of the actual reward, they appear as better moves to the agent than a piece-substate pair that has not had any award value.

Figure 4.3 shows the average performances of the rest of the SmartAgent versions: SmartAgent V1.4, SmartAgent V1.5, SmartAgent V1.6, SmartAgent-Adaptive and SmartAgent-Probable. As with the other agents, these agents ran over each MDP three times, with 24000 steps per MDP, for each of the 20 MDPs, with a random seed for equality. The results for these agents are a little more mixed than the previous figures results. Although SmartAgent

V1.4 was a fixed-strategy agent, it actually performed better than SmartAgent V1.5 and SmartAgent V1.6. These results indicate that adaptability may not be needed at all to effectively play different variants of Tetris. However, this performance graph has been heavily averaged and is misrepresenting towards the actual agents' performances over each MDP (shown more clearly later in Figure 4.5).

Another possible explanation for V1.4's comparable performance to V1.5 and V1.6 is that V1.4 does not have to spend time testing out various parameter sets, whereas V1.5 and V1.6 spend the first part of their lifetime testing parameters to see which ones work well. This could be the reason that SmartAgent-Adaptive performed so well. When SmartAgent-Adaptive finds a good parameter set, it increases the cooling rate to converge quicker. This leads to less exploration and more exploitation.

As expected, SmartAgent-Probable performed better than all of the other agents. This was probably because of the lookahead strategy SmartAgent-Probable uses. However, the margin between it and all of the other agents is fairly small showing that the performance gain of the lookahead strategy is minimal.

Table 4.2 shows the actual rewards of the agents after 24000 steps. Each of the later agents require only two pieces to clear a line, which theoretically, is impossible. However, the table is disregarding reward exponents and clearing multiple lines, so this makes more sense and shows that each of the later agents are making use of the reward exponent and clearing multiple lines at a time.

4.2.2 Concatenated Performances

The later experiment mode showed the agents' performances in a different manner: each agent's performance in each MDP is concatenated together to create a longer, more jagged line than the previous experiment's output line. The resulting graph more clearly shows how an agent performed in each MDP and highlights odd agent behaviour when viewed together with other agents. The results shown in Figures 4.4 and 4.5 use the same experiment setup: each agent is run over 20 MDPs with 256000 steps per MDP without repetitions and with a random seed for equality of pieces. Repetitions were not used in this experiment because the competition format only had one repetition, any variance should balance out because of the large number of steps and MDPs, and running the experiments would take too much time. 256000 steps were used to more closely approximate the competition's 5 million steps per MDP than the previous experiment results (Section 4.2.1) without taking too long to run the experiment. As with the previous experiment, the 20 MDPs used were the predefined MDPs from the competition, numbered MDP 0-19. Each figure is shown with tics marking the boundaries of each MDP.

Figure 4.4 shows the concatenated performances of the first three (fully working) agents: SmartAgent V1.1, SmartAgent V1.2, and SmartAgent V1.3. As with the averaged performances graph (Figure 4.2), the increases in performance with each iteration are clear. Though it is harder to see on this graph,

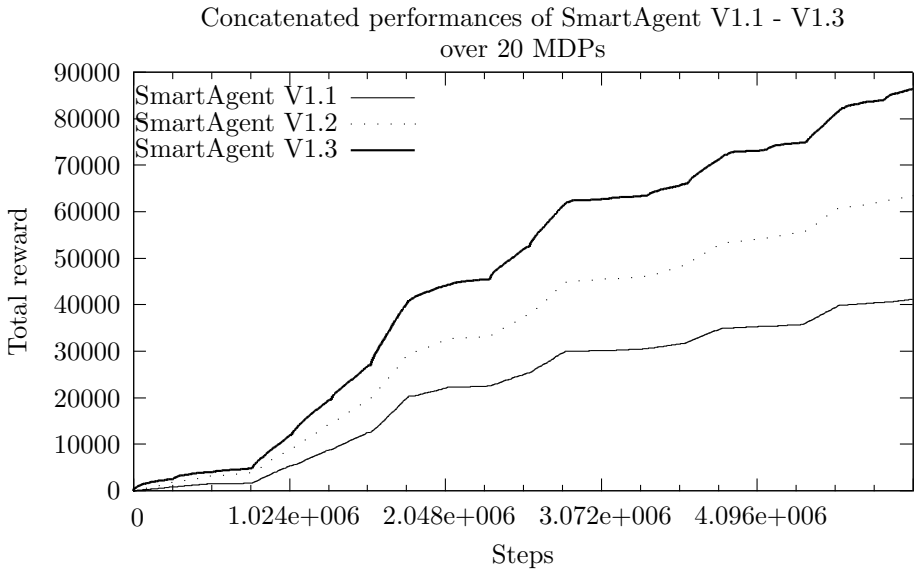

Figure 4.4: The concatenated performance of SmartAgent 1.1, 1.2 and 1.3 over 20 MDPs.

SmartAgent V1.3 still has the problem of degrading performance, noted by the steep slope at the start of each MDP which become less steep before becoming linear as the steps pass. An interesting feature of this graph is that the amount of increase between each iteration's performances appear to be the same. This shows that each iteration of development was a logical one and development was always progressive.

Figure 4.5 shows the concatenated performances of the rest of the SmartAgent versions: SmartAgent V1.4, SmartAgent V1.5, SmartAgent V1.6, SmartAgent-Adaptive and SmartAgent-Probable. As expected, SmartAgent-Probable was the best performing agent among the five, with SmartAgent-Adaptive performing slightly better than SmartAgent V1.6. Once again, SmartAgent V1.4 is better than SmartAgent V1.5, but does not perform as well as SmartAgent V1.6. This could be because SmartAgent V1.6 is given more time to create a better parameter set than in the averaged graph (Figure 4.3). It could also simply be random variance in the favour of SmartAgent V1.6 (or disfavour of SmartAgent V1.4). What is clear from the two graphs is that SmartAgent V1.5 is the least effective agent among the later agents.

4.2.3 Episodic Performance

Although SmartAgent-Probable only performed slightly better than the other agents in terms of total reward, there was another performance measure that the

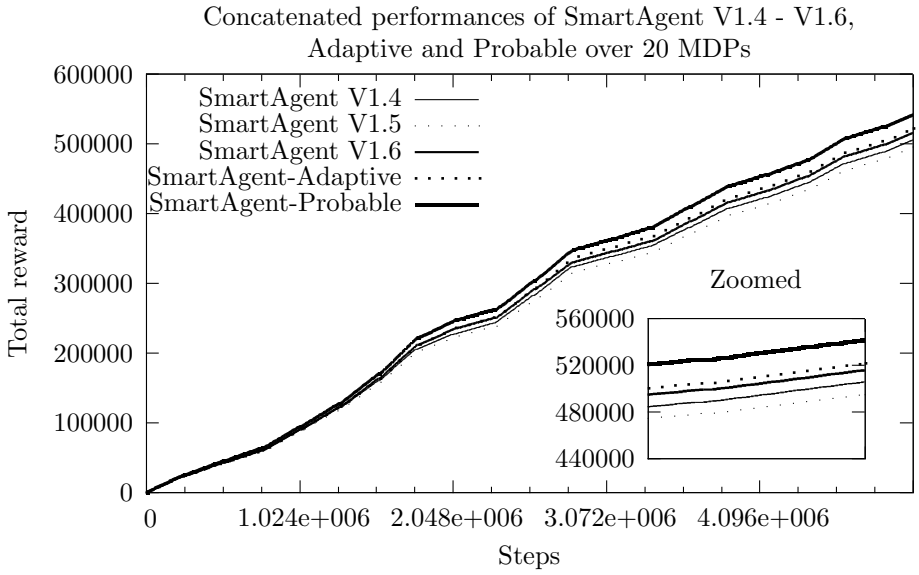

Figure 4.5: The concatenated performance of SmartAgent 1.4, 1.5, 1.6, Adaptive and Probable over 20 MDPs.

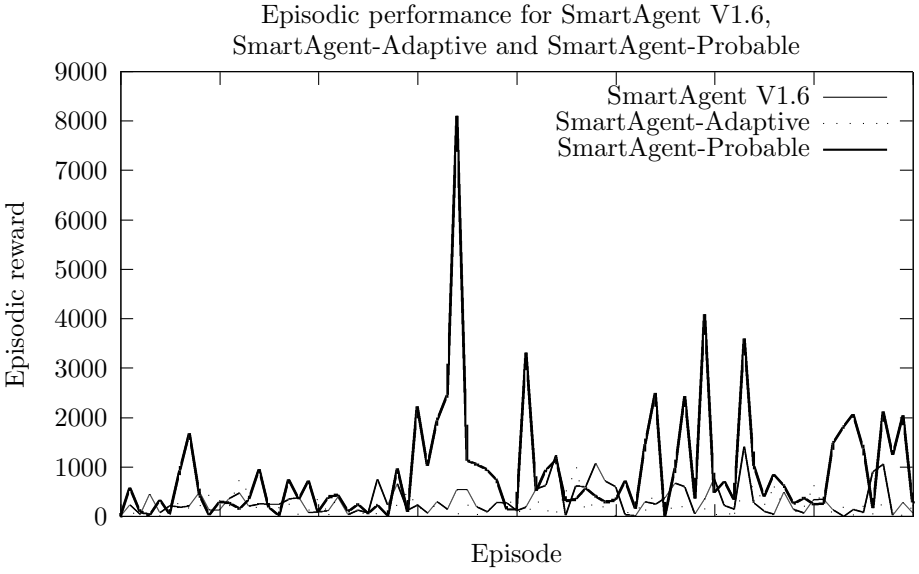

Figure 4.6: The episodic performance of SmartAgent V1.6 and SmartAgent-Probable.

Tetris Workshop GUI experimenter was able to measure: reward per episode. Shown in Figure 4.6 is a graph of performance measured by reward per episode on an arbitrary MDP. SmartAgent-Probable clearly performs better than the other two agents per episode, surviving longer per episode and generally making more reward. These results lead to the conclusion that SmartAgent-Probable would make a better survival agent than the other SmartAgent versions, even if it is a little slow in making decisions. With some optimising of the SmartAgent-Probable algorithm, it could be a fast, effective survivalist agent. However, the RL Competition is judged on total reward gained, so survivalism is not an ideal strategy.

Chapter 5

Future Work

Although this project covered a lot of what it set out to do, there is always room for further improvement. Because SmartAgent finished fifth in the competition, there is room for improvement to elevate its placing. By drawing on what has been learnt throughout the development and incorporating elements from other agents in the competition, SmartAgent could be improved in several ways.

The first thing to change would be how the agent played the game of Tetris. Because the competition was judged on a finite number of steps where failing an episode held no negative reward, Team Loria INRIA - MAIA (Section 4.1) held an immediate advantage by the way their agent was tailored. SmartAgent could easily be changed to be tailored more closely to the competitions rules by changing certain elements of how it plays:

- The parameters used in the parameter set should be changed to features more accommodating to the competition. Although the parameters SmartAgent uses are good for survival, there are other ones that may be better for achieving maximal reward. Team Loria INRIA - MAIA (Section 4.1) used a large parameter set consisting of many parameters grouped from various sources[14]. SmartAgent's parameter set could be changed to use some of the parameters they used, such as individual features for each column's height, or could use parameters similar to these to attempt to obtain a larger overall reward rather than to attempt to survive.

- When choosing a goal position, not only should the agent be more aware of whether it's actually possible to place the piece there (pre-simulate the movement steps to get to the goal and check for validity), but the agent should also take into account how many steps it takes to get to a position. If presented with two positions, pos_A and pos_B, where pos_A is a slightly better position than pos_B, in the SmartAgent V1.6 strategy the agent would automatically choose pos_A. However, say that pos_A takes five movements to get to, but pos_B only requires one, pos_B is actually the better position to go to (still assuming it is only slightly worse than

pos_A) because it requires less movements and thus, causing a higher rate of tetrominoes per steps.

- Because there is no negative reward associated with failing an episode under the competition rules, if the Tetris field becomes too full for effective play, the agent's best action could be to start over by 'committing suicide.' This would replace the old, complicated field with an empty new state where obtaining reward is easier with a minimal negative side effect (it will take a few tetrominoes before a reward can be made).

- As it was found that there is no time limit on an agent's performance, SmartAgent-Probable could potentially be used for optimal piece placement. Although the agent could potentially be extended further by looking ahead for even more steps, the processing time for each lookahead step increases drastically and would not be prudent.

- If there still is no time limit for an agent, off-line learning could be used in the same way that Team Loria INRIA - MAIA used it. An agent could simply play as per normal until it has a large enough amount of data to approximate a Tetris environment to perform off-line testing on. This drastically reduces the amount of on-line exploration time and allows for instant exploitative play.

Although SmartAgent V1.3's performance was eclipsed by the performances of SmartAgent V1.4 onwards, it may achieve a greater performance if more experimentation was performed on the eligibility trace. Perhaps a trace length of ten is too long or too short, perhaps the agent would perform better if the reward was distributed with exponential decay rather than linear decay. A sort of trace could be implemented into SmartAgent V1.5 onwards by rewarding parameter sets that are similar to the current parameter set with similarity measured by some sort of threshold function. For instance, the parameter set $P_A = \{1, 2, 3, 4, 5\}$ is similar to $P_B = \{1, 2, 3, 4, 6\}$ so when P_B is rewarded, P_A receives some proportion of reward as given by a similarity function $Similarity(P_x, P_y)$ which returns a number between 0.0 and 1.0 (inclusive). If $P_x = P_y$, the function returns 1.0. If P_x is dissimilar to P_y by a certain threshold, the function returns 0.0. Otherwise, some number between 0.0 and 1.0 is returned relating to the similarity of the parameter sets.

A possible direction for agent development that was not pursued in full detail because of time constraints was the use of adaptive simulated annealing. Section 4.2 indicates that the ASA agent received the best results (apart from SmartAgent-Probable), even though it may settle on a sub-optimal parameter set. The results could be improved upon by fully integrating and optimising ASA in SmartAgent's process. As stated in Section 4.2, SmartAgent-Adaptive may have an advantage over the simulated annealing agents because it spends less time exploring and quickly converges on a good solution.

Although SmartAgent-Probable achieved good results, it did so in a very long amount of time. As stated in Section 3.1.9, SmartAgent-Probable is roughly

111 times slower than the other agents, which limits its usefulness. Future work could go into improving the algorithm for efficiency, such as culling parts of the field to place pieces upon, or only simulating a portion of the next most probable pieces.

Another promising area for future work is Tetris Workshop GUI. There are possible additions to it that could produce interesting results and allow the program to take in a larger range of Tetris agents:

- An obvious addition is allowing the user to play with the computer. Not only would this be enjoyable, it could also provide useful input for an agent capable of learning by example. By integrating player-computer interactions, other features could be added in as well, such as showing the best position as given by the computer when the player is playing or even simply a competitive game of Tetris between the player and the computer agent.

- As stated previously, when training a Tetris playing agent, failing an episode has no effect towards its learning and is a restriction made purely for survivalist agents or human players. So, an alternative performance measure is to let the number of rows in the field be infinite and the effectiveness of an agent is measured by the pile height of the blocks. Bdolah and Livnat [3] used this performance measure in their experiments. When showing the field in the GUI, only the top portion of the pile height needs to be shown, both to the user and the agent, and a label showing the current height of the pile.

- SmartAgent was created for one-piece Tetris, where only the currently falling tetromino is known. A possible extension is incorporating the more traditional two-piece Tetris, where the current and next tetromino are shown, which will allow an agent to set up better strategies for play. Colin Fahey's fixed-policy agent is a two-piece agent and has been able to clear over 7 million lines [7]. SmartAgent-Probable is already halfway towards being a two-piece agent except that it does not know what the next piece will be.

Something this project failed to do was compare reinforcement learning methods to traditional machine learning methods. The comparisons within the competition and this project were all between other reinforcement learning agents. As an alternative comparator, a simple static agent could be created using WEKA to define the strategy. In Section 3.2.2, an attempt was made to use WEKA to determine the initial weights by training a classifier on data produced by SmartAgent. Although the initial results proved to be of little use, given more work and effort, they could be improved upon.

Instead of predicting the best initial parameters, a static agent could use the observed features of the field to predict the best position for the piece on the field by training a WEKA classifier on example data files created by SmartAgent itself. The data files could be created by running the SmartAgent

as usual, but storing each move decision and reward given by the decision. Each instance in the file would be made up of attributes defining the piece location and the field state, or more specifically: piece type, piece position, piece rotation, and height differences between each neighbouring column. The class value would be equal to the number of lines removed with that particular piece placement. Once trained, the classifier would then, given a piece type, determine the best position and rotation by expected reward much in the same way that SmartAgent determines the best as given by field features.

There are some problems with creating a static agent in this manner. The static agent may be overfit to an MDP and be unable to cope with MDPs of different size. Also, the static agent is really just a dynamic agent in a fixed state as it has been trained on data files created by a dynamic agent. Because of this, the static agent is unlikely to perform better than the dynamic agent, especially on differing MDPs.

Another avenue of exploration involving WEKA is to create an agent that still uses a machine learning classifier to make decisions but uses an incremental classifier to choose the best positions. Like the static agent, the incremental model would predict the best position on the field by evaluating expected reward. The difference is that the incremental agent would learn from its choices and incorporate the results into furture decisions. Although the incremental agent may need some pre-training to learn a basic initial classifier, it will not require as much pre-training as the static agent because it can learn new strategies as it plays.

Because the incremental agent has the ability to learn, it also has the opportunity to explore possible movements and train a better classifier. As with SmartAgent, the incremental agent would decide whether to place a piece exploitatively or exploratorily, where the piece will be placed in the best predicted position for exploitative play, and placed in one of the top x positions for exploratory play (where x is an arbitrary value, less than or equal to the total number of possible positions), as predicted by the incremental classifier. After a piece has been placed, the position, necessary field details and reward are input into the incremental classifier, which uses the data to refine its strategy.

WEKA currently does not have incremental regression algorithms[1], but because it is open-source, an incremental classifier could be created from a pre-existing static classifier. As with TD-Gammon [15], a neural network could be used, which changes the weights and number of neurons dynamically with the data. In a sense, an incremental classifier is the policy of the agent, used to store and predict movements.

[1] As at WEKA v3.5.8

Chapter 6

Conclusion

The results in Section 4.2 clearly show a steady increase in agent performance over each iteration of development (with the exception of SmartAgent V1.4). The performance especially increases when the agent strategy changes from substate-based play to field evaluation-based play. In order to play Tetris properly, an agent needs to be able to evaluate the entire field when making choices for moves, rather than a subsection of the field. If an agent only looks at a subsection, it cannot know what will make lines, only what will fit best in that particular subsection.

An important conclusion drawn was that the agent required more than pure reinforcement learning. In every SmartAgent version, the agent had assistance in playing Tetris. SmartAgent V1.0 contained the most basic assistance, in the form of placing pieces. In the competition interface of Tetris, placing a piece often requires a series of actions in order to manipulate the piece's orientation. By including a helper function that automatically places pieces by returning an appropriate string of actions to the environment, an agent can focus on learning meta-strategies for effectively playing Tetris, rather than learning base strategies on how to place pieces. The later versions all included other helper functions, such as V1.2's guided exploration and V1.4's field evaluation placement. In a competitive environment, the agent needs to be as effective as possible in order to win, so it should be given as much help as possible, rather than requiring it to learn every little base-level concept from scratch. Team Loria INRIA - MAIA did this well by training their agent off-line and generating an 'instant' exploitative agent.

This feature of the competition — allowing an agent infinite time to make a decision — is a large, counter-intuitive loophole in the rules that can be exploited to create static, non-reinforcement learning agents. As stated previously, Team Loria INRIA - MAIA used this loophole to train an agent offline after simply observing the domain and recording basic features of the MDP. Because of this loophole, reinforcement learning agents did not have an advantage in the competition over off-line, pseudo-static agents and the testing grounds were not ideal performance measures.

A curious result from SmartAgent V1.3's performance was that it un-learned how to play over time. It began each MDP with a strong learner (with regards to V1.1 and V1.2), but gradually played less effectively than how it began, eventually settling on a sub-optimal strategy. Although the exact reason for this behaviour is unclear, it is likely that the eligibility trace affected the performance both positively and negatively. The trace clearly gave SmartAgent V1.3 an advantage over SmartAgent V1.2 and made it learn fast initially, but somehow it affects the policy negatively as well, degrading the performance over time. The degrading performance is likely directly related to the simulated annealing 'temperature,' implying that exploitative decisions learned during exploratory play are to blame. In SmartAgent V1.3 these exploitative decisions are shaped not only by the piece-substate pairs that receive reward, but also by a history of piece-substate pairs leading up to that reward. Some of those pairs present in the history could have been substandard movements, but they are rewarded for simply being in the history. As mentioned in Chapter 5, the trace was not optimised and could have benefit from a smaller trace, or a different reward appropriation algorithm.

Even though SmartAgent-Probable and SmartAgent-Adaptive have shown themselves to be the most effective agents, they are not better than the other higher-level agents by a significant amount. It is doubtful that they will change SmartCraft's (see Section 4.1) placing in the competition much. Because every version of SmartAgent was created with the primary goal of survival, which was not the competitions primary goal, each iteration of agent was only comparable to previous iterations, rather than other competing agents. If the feature set is changed to cover features more beneficial to pure reward collection and totally disregarding survival, the agent may place higher in the competition.

The application of reinforcement learning to a Tetris-playing AI agent was shown to be successful, but required more than pure reinforcement learning. An agent needs guidance towards playing the game, a decent foothold to start learning, and an effective method of balancing exploration with exploitation. In comparison to static methods, reinforcement learning methods are only slightly better, showing that an effective static method could be used in all Tetris scenarios rather than a complex learner.

Bibliography

[1] J. Åsberg. Artificial Tetris players, 2007. http://frt.fy.chalmers.se/cs/cas/courses/seminar/071126/asberg.pdf.

[2] J. Baxter, A. Tridgell, and L. Weaver. Knightcap: A chess program that learns by combining td(λ) with game-tree search. In *Proceedings of the 15th International Conference on Machine Learning*, pages 28–36. Morgan Kaufmann, 1998.

[3] Y. Bdolah and D. Livnat. Reinforcement Learning Playing Tetris, 2000. http://www.math.tau.ac.il/~mansour/rl-course/student_proj/livnat/tetris.html.

[4] N. Böhm, G. Kókai, and S. Mandl. An Evolutionary Approach to Tetris. In *6th Metaheuristics International Conference*, pages CD–ROM, 2005.

[5] D. Carr. Adapting Reinforcement Learning to Tetris, 2005. http://research.ict.ru.ac.za/g02c0108/HnsThesis.pdf.

[6] E. D. Demaine, S. Hohenberger, and L. D. Nowell. Tetris is Hard, Even to Approximate. Technical Report MIT-LCS-TR-865, Massachussets Institue of Technology, Boston, 2002.

[7] C. P. Fahey. Tetris AI: Computer Plays Tetris, 2003. http://www.colinfahey.com/tetris/tetris_en.html.

[8] V. F. Farias and B. V. Roy. Tetris: A study of randomized constraint sampling. *Probabilistic and Randomized Methods for Design Under Uncertainty*, 2006.

[9] L. Ingber. Adaptive simulated annealing (ASA): Lessons learned. *Control and Cybernetics*, 25:33–54, 1996.

[10] S. Melax. Reinforcement Learning Tetris Example, 1998. http://www.melax.com/tetris.html.

[11] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[12] S. Schaal, C. G. Atkeson, and S. Vijayakumar. Real-time robot learning with locally weighted statistical learning. In *In International Conference on Robotics and Automation*, pages 288–293. IEEE Press, 2000.

[13] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, March 1998.

[14] I. Szita and A. Lörincz. Learning Tetris using the noisy cross-entropy method. *Neural Comput.*, 18(12):2936–2941, 2006.

[15] G. Tesauro. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38(3):58–68, 1995.

[16] U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, editors. *RoboCup 2007: Robot Soccer World Cup XI, July 9-10, 2007, Atlanta, GA, USA*, volume 5001 of *Lecture Notes in Computer Science*. Springer, 2008.

[17] S. Whiteson, B. Tanner, and A. White. 2008 Reinforcement Learning Competition, 2008.
`http://rl-competition.org`.

[18] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, June 2005.