Minimax Search and Reinforcement Learning for Adversarial Tetris

Maria Rovatsou

Department of Electronic and Computer Engineering

Technical University of Crete

A thesis submitted in partial fulfillement for the

Diploma Degree

September 2009

Πολυτεχνείο Κρήτης

Αναζήτηση **MiniMax** και Ενισχυτική Μάθηση για Ανταγωνιστικό **Tetris**

Μαρία Ροβάτσου

Τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών

Σεπτέμβριος 2009

This work is dedicated to my loving family, my boyfriend and my friends that I owe every meaningful moment of my life, and to my supervisor that without his help and support I would not be able to present this work. Thank you all for your care and support it means everything to me.

Abstract

Game playing has always been considered an activity requiring a good level of intelligence and therefore has become a major research area within Artificial Intelligence and Machine Learning. This thesis focuses on Adversarial Tetris, a variation of the well-known Tetris game, introduced at the 3rd International Reinforcement Learning Competition in 2009. In Adversarial Tetris the mission of the player to complete as many lines as possible is actively hindered by an unknown adversary who selects the falling tetraminoes in ways that make the game harder for the player. In addition, there are boards of different sizes and learning ability is tested over a variety of boards and adversaries. This thesis describes the design and implementation of an agent capable of learning to improve his strategy against any adversary and any board size. The agent combines MiniMax search enhanced with Alpha-Beta pruning for looking ahead within the game tree and the Least-Squares Temporal Difference Learning (LSTD) algorithm for learning an appropriate state evaluation function over a small set of features. The learned strategies exhibit satisfactory performance over a wide range of boards and adversaries and our agent achieves good scores on the testing run of the competition.

Περίληψη

Η συμμετοχή σε παιχνίδια θεωρούνταν ανέκαθεν μια δραστηριότητα που απαιτεί ένα καλό επίπεδο νοημοσύνης και κατά συνέπεια έχει αποτελέσει ένα σημαντικό ερευνητικό χώρο στα πλαίσια της τεχνητής νοημοσύνης και της μηχανικής μάθησης. Η παρούσα διπλωματική εργασία εστιάζει στο παιχνίδι ανταγωνιστικό Tetris, μια παραλλαγή του γνωστού παιχνιδιού Tetris,το οποίο επινοήθηκε για τις ανάγκες του διαγωνισμού International Reinforcement Learning Competition το 2009. Στο ανταγωνιστικό Tetris ο σκοπός του παίκτη είναι να συμπληρώσει όσες περισσότερες γραμμές είναι δυνατόν ενώ παρεμποδίζεται ενεργά από έναν άγνωστο αντίπαλο ο οποίος επιλέγει τα ριπτόμενα τουβλάκια με τρόπους που δυσκολεύουν το παιχνίδι για τον παίκτη. Επιπρόσθετα, υπάρχουν πίνακες παιχνιδιού διαφορετικών διαστάσεων και η ικανότητα μάθησης εξετάζεται σε ένα πλήθος διαφορετικών πινάκων και αντιπάλων. Η παρούσα εργασία περιγράφει το σχεδιασμό και την υλοποίηση ενός πράκτορα ικανού να μαθαίνει να βελτιώνει τη στρατηγική του απέναντι σε οποιαδήποτε αντίπαλο και οποιοδήποτε μέγεθος πίνακα. Ο πράκτορας συνδυάζει αναζήτηση MiniMax εμπλουτισμένη με κλάδεμα α-β για να προελαύνει στο δένδρο του παιχνιδιού και τον αλγόριθμο Least-Squares Temporal Difference Learning (LSTD) για να μαθαίνει μια κατάλληλη συνάρτηση αξιολόγησης καταστάσεων του παιχνιδιού βάσει ενός μικρού συνόλου χαρακτηριστικών. Οι στρατηγικές που μαθαίνονται επιδεικνύουν ικανοποιητική απόδοση απέναντι σε ένα ευρύ φάσμα πινάκων και αντιπάλων και ο πράκτορας μας επιτυγχάνει καλές βαθμολογίες στη δοκιμασία του διαγωνισμού.

Contents

List of Figures

LIST OF FIGURES

Chapter 1

Introduction

Reinforcement Learning is a growing field of Machine Learning focusing on learning by trial-and-error. The agent is placed in an environment almost or fully unknown and learns to accomplish certain tasks by processing the information stemming from its interaction with the environment. This learning setup has been used as the basis for the introduction of many Reinforcement Learning algorithms that led to outgrowing performance of the agents in difficult tasks demanding intelligent behavior.

Skillful game playing has always been considered a token of intelligence, consequently Artificial Intelligence and its field Reinforcement Learning exploit games in order to exhibit intelligent performance. Another reason for choosing games for creating and testing learning algorithms is that they provide a complex environment that can be easily simulated and controlled in contrast with the real world.

A game that has become a benchmark, exactly because it involves a great deal of complexity along with very simple playing rules, is the game of Tetris. It consists of a board in which one of the seven available four-block tiles fall from the top to the bottom one after the other. The goal of the player is to place the pieces so that they form complete lines, which are eliminated from the board, lowering all blocks above. The game is over when a tile placed in the board reaches the top of the board. The environment of the game chooses randomly (typically, uniformly) the tile that is going to fall next and the player can fully manipulate the tile in order to place it at the desired position. The fact that the rules are simple should not give the impression that the task is simple. There are about 40 possible actions available to the player every time he has to decide for a placement and about 10^{64} possible states that these actions could lead to. This is hard even for a human player, let alone for an agent that has certain bounds on resources he can use.

Adversarial Tetris is a variation of Tetris that keeps the simplicity of playing rules and the complexity of the task and combines them with another aspect, adversity. The adversarial environment makes the task even more demanding and intriguing, as an unknown adversary tries to hinder the player from eliminating lines. The sole way for the adversary to achieve this is to choose pieces that augment the difficulty of completing lines for the player and can even “leave out” a tile from a whole game if it suits his adversarial game play.

Adversarial Tetris was formulated as a Markovian Decision Process for the needs of the 3rd Reinforcement Learning Competition. The competition started in March 2009 and ended on June 8, 2009. The domain of Adversarial Tetris had only two teams competing and unfortunately due to technical reasons our agent was not able to participate. The task however witholds its interest beyond the competition due to its challenging nature.

Our approach is to combine a Game Search algorithm in order to produce a strategy that will confront the adversary and minimize its effect on our agents game play and a Reinforcement Learning Algorithm that enables the agent to learn how to perform well in this game. The Minimax Search Algorithm combined with Alpha-Beta Pruning enables the agent to ”think” beyond his immediate move, at least see the opponent’s move in response to his own. However, this does not suffice as the agent must learn which actions will return high reward in the long run. The proposed agent exhibits a good learning performance and balances the criteria of maximizing his score in respect to the opponent’s moves, while trying not to lose during the game.

1.1 Thesis Overview

Chapter 2 contains a consice description of the theory and algorithms we applied in this thesis and provides some background knowledge for the methods discussed later on. It includes a short description of Tree Search algorithms, Minimax Search and Alpha-Beta Pruning, evaluation functions, as well as a quick introduction to Markov Decision Processes and Reinforcement Learning to clarify the context of learning that formed the basis for our agent. The problem of value prediction with function approximation is emphasized and last but not least the Least-Squares Temporal Difference learning algorithm that was used in this thesis is consisely described.

Chapter 3 provides a detailed presentation of the game of Tetris and its variation Adversarial Tetris along with the formalization of Adversarial Tetris as a Markovian Decision Process and the goal of this thesis with repect to the game. This chapter also surveys the related previous work on the game of Tetris.

Chapter 4 describes how we modelled the players actions for Tetris in order to make the agent more efficient in using the information he receives from the environment and the agent's architecture. In this chapter, the Minimax Game Tree formulation is described and the evaluation function that is used by the agent in order to estimate the value of different board depending on their configuration. Lastly, we propose an incremental version of the Least-Squares Temporal Difference Learning algorithm which is more appropriate for learning inthe context of a game.

Chapter 5 fills the reader with some more information on the RL-Competition and the RL-Glue Framework that provided the environment, the experiments, and the agent interface for our agent. Several technical details are provided about the Java implementation of our agent.

Chapter 6 includes the results of all the learning and testing experinments of this thesis. Experiments were run for representative parameter values to demonstrate different aspects of the learning problem and the properties of the learned agents.

Chapter 7.3 discusses the outcome of this thesis along with some future work that is proposed.

Chapter 2

Background

2.1 Search in Games

2.1.1 Tree Search

A search problem is typically described by a state space, an initial state, an action space, a successor function, a goal test, a cost function. The state space describes all the possible situations in which we are searching for a situation of interest (goal state) with certain properties. The initial state is where the search begins. The action space includes all the options available in each state; an action choice in some state leads to a number of successor states as dictated by the successor function. Each transition from one state to another comes with a cost described by the cost function. The goal of a search algorithm is to find a sequence of actions which when applied from the initial state generates a path through the state space leading to a goal state with a minimum total path cost.

There are several search algorithms which are based on the idea of generating a search tree from the initial state and they differ only in the strategy they use to expand the tree. A search tree is formed by generating a root node representing the initial state and by expanding the initial state and its successors recursively generating children nodes for all successor states of a node. In this sense, a search tree represents all possible paths through the state space starting from the initial state. Each node in the tree, in addition to the representation of the corresponding state, holds useful information about the action that lead to its creation, links to its predecessor and to its successor nodes, its depth in the tree from the root, the total path cost up to that node, etc.

There are some kinds of nodes of special importance in the tree. One is the root node that is already mentioned. Another kind is a fringe node, that is a node whose successor nodes have not been generated yet, i.e. the node has not been expanded yet. The set of nodes that have not been expanded at any time is called the fringe of the tree. Finally, the terminal or leaf nodes are nodes corresponding to states with no successors. That means that a search tree cannot be expanded from terminal nodes and represent dead-ends.

2.1.2 Game Trees

A one-player game can be easily cast as a search problem. The state space includes all possible states of the game, the initial state represents the beginning of the game, the actions are the player choices, the successor function includes the rules of the games, and the cost function is the rewarding scheme of the game that describes how the score is modified with each state transition. States describing the end of a game correspond to terminal nodes in the search tree. Computing a good strategy is equivalent to searching for a sequence of actions that leads to a game end with highest total score, the only difference being the change of optimization objective as we are now looking for the path of maximum total "cost". Therefore, given sufficient time an agent can search the entire game tree and find an optimal strategy (an optimal path) leading to the highest possible score. Thanks to the fact that game trees even for simple one-player games are enormous in size, interest in such games has not been lost and attention is focused on approximating optimal strategies as close as possible with limited computational resources.

An alternating game is a two-player game where player play in turn and typically have conflicting objectives; one is trying to maximize the score (maximizer), while the other is trying to minimize it (minimizer). These games are known as zero-sum games since what is won by one player is lost by the other and therefore the sum is always zero. This common type of game can also be modeled as a search problem, however with several modifications. As before, each node in the game tree represents the state of the game and information about which player is supposed to play at the current state. Therefore, the game tree is now organized in alternating levels depending on the player turn; each such level is called a ply. The root node of the tree corresponds to the initial state of the game and the terminal nodes correspond to states where the game has ended.

As an example, consider the game tree shown in Figure 2.1 for the well-known Tic-Tac-Toe game. Tic-Tac-Toe is played by two players on a 3×3 board. One player draws Xs and the other Os placing them on empty squares of the board. The player who completes a series of three of his own symbols in any direction wins. If the board is filled without anyone completing any series of symbols the game ends in a draw. We assume that our two players are **Max** and **Min**. Max has the X and Min the O. The initial state of the game is when the board is blank. Let's say that Max plays first. Max has nine choices at the beginning of the game. Every choice of Max can lead to a possible state of the game. These states are the successor states of the initial state. In the game tree there are now nine states with an X on a different place; the nodes at this level correspond to a ply. When Max has made his choice, Min has eight choices given the Max's choice. So, for every state that was created before with one X there will be eight successors with an O in a different place. As they play in alternating turns the game tree covers all possible game evolutions. The nodes at the ninth ply are terminal nodes, that is states where the game has ended. In each terminal node, a payoff is given to each of the two players. A reward for the winner and a penalty for the loser ($+1$ if Max wins, -1 if Min wins, 0 for a draw).

2.1.3 MiniMax Search

Every path down a two-player alternating game tree represents alternating player choices. Therefore, a single player cannot really drive the game into a desired terminal node, because the two players have conflicting goals and therefore will try to reach different terminal nodes. As a result, each player will try to choose actions that will increase his probability of winning and will reduce his probability of loosing. Given that the strategy of the opponent is typically unknown an agent searching such a game tree can only assume that the opponent will play optimally.

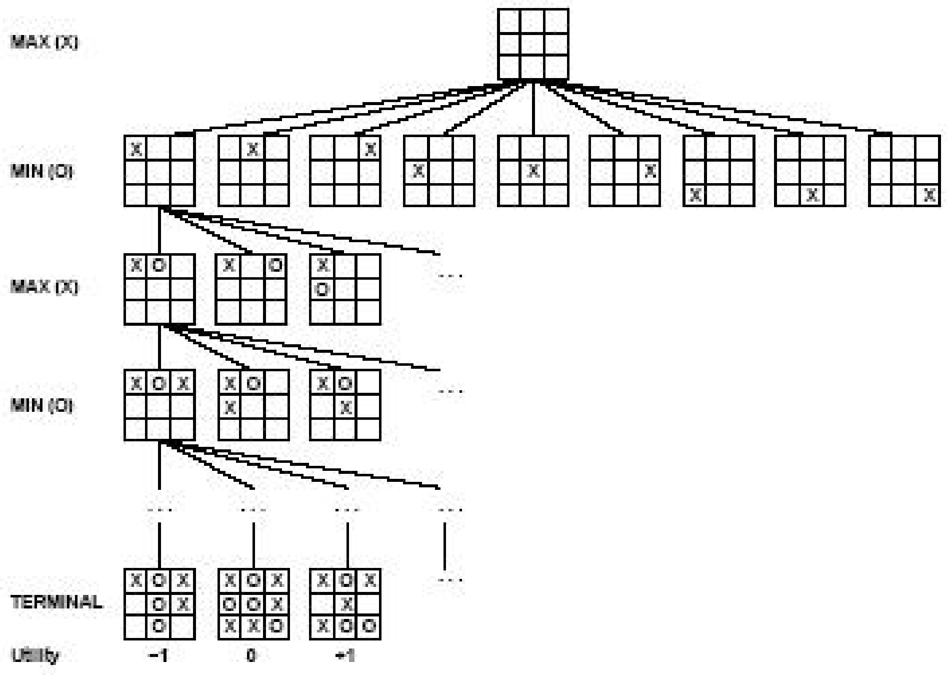

Figure 2.1: Tic-Tac-Toe game tree

This is the only safe assumption that can be made in the absence of any other information. Any other assumption will lead to strategies that can eventually be exploited by the opponent.

These ideas gave rise to the **Minimax Search** algorithm [1] for two-player zero-sum games. MiniMax search generates the game tree in order to compute the minimax value of every node, representing the utility of any given node to the player we are in favor. The computation of the minimax value is done according to the way Max and Min play. Max wants to maximize his payoff, so he wants to choose the children node with the greatest value. Min, on the contrary, wants to minimize Max's payoff, so as to maximize his own payoff, so he chooses the children node with the least value. Given this pattern repeats, the algorithm uses a recursive computation for the utility of any node n: The complete MiniMax algorithm from the maximizer's point of view is shown in Algorithm 1. Clearly, the algorithm will have to reach the leaf nodes of the tree to terminate the recursion and return their utility. The recursion unwinds from the leaves towards the root

$$\textsc{MiniMax Value}\ (n) = \begin{cases} \textsc{Utility}\ (n), & \text{if } n \text{ is a terminal state} \\ \max_{s\ \in\ \text{Successors}(n)} \textsc{MiniMax Value}\ (s), & \text{if } n \text{ is a Max node} \\ \min_{s\ \in\ \text{Successors}(n)} \textsc{MiniMax Value}\ (s), & \text{if } n \text{ is a Min node} \end{cases}$$

of the tree backing up the values that the players choose in each node. Once the minimax value of the root is updated the Max player is ready to make a decision and choose this action that leads to the child node where the minimax value at the root came from. Recursively, one can extract the full optimal strategy leading to the terminal node whose utility was backed up all the way to the root. If both players play optimally, the game will end in that terminal node. Note that even if the Min player chooses to play a suboptimal strategy, the minimax strategy guarantees that the resulting score for the Max player will be no less that the minimax value at the root. Applying MiniMax to the game tree of Tic-Tac-Toe will bring a value of 0 to the root, as expected, indicating the fact that optimal play by both agents will definitely lead to a draw. A minimax value of $+1$ or -1 at the root would indicate an inherent advantage given to one player over the other.

Notice that a game tree does not necessarily have to begin from the initial state of the game; the root node can be any intermediate state of the game. A game tree can be formed from any state where a player needs to make a decision and the resulting minimax strategy will be the best strategy from the current state. The game tree is a principled way of looking ahead into the game and predicting possible evolutions. This fact is exploited by MiniMax to create optimal strategies.

2.1.4 Alpha-Beta Pruning

The main problem with Minimax Search is the space and time complexity. Its time complexity is $O(b^m)$ and its space complexity if all successor nodes are produced at once is $O(bm)$, with b being the number of legal moves of a player (the branching factor of the tree) and m the number of plies. This complexity

Algorithm 1 The MiniMax algorithm for search in game trees.

```
action := MiniMax Decision (state)
inputs: a state
returns: an action
  $u \leftarrow$ MaxValue (state)
  return action $\in$ Successors(state) with value $u$

utility value := MaxValue (state)
inputs: a state
returns: a utility value
  if TerminalTest(state) = true then
    return Utility(state)
  end if
  $u \leftarrow -\infty$
  for all $s \in$ Successors(state) do
    $u \leftarrow \max(u,$ MinValue (state))
  end for
  return $u$

utility value := MinValue (state)
inputs: a state
returns: a utility value
  if TerminalTest(state) = true then
    return Utility(state)
  end if
  $u \leftarrow +\infty$
  for all $s \in$ Successors(state) do
    $u \leftarrow \min(u,$ MaxValue (state))
  end for
  return $u$
```

makes the algorithm inapplicable, if no optimizations are used. The most common optimization to MiniMax is the Alpha-Beta Pruning algorithm.

Alpha-Beta Pruning [1] is an enhancement to MiniMax that computes the same minimax values, however by pruning away parts of the tree that are not going to change the MiniMax decision at the root. The main idea is that if there is a better choice for a player at the parent node or at any predecessor then the current node and entire subtree underneath can be eliminated or get *pruned*. This pruning is safe since the pruned subtrees will not affect the computation of the minimax values. The value of a Max node can be estimated even before processing all its successor nodes. The current max value is a lower bound to its final value; if this lower bound exceeds the value of some Min choice higher up in tree, it will definitely lead the Min to avoid the current branch and therefore it can be pruned. A similar argument holds for pruning Min nodes.

In order for the pruning to be done effectively, the value of the best choice along a path must be accessible to the successor nodes. For this reason, two values are propagated to the successor nodes: α and β. α is the value of Max's best choice along the path and β is the value of Min's best choice. Pruning at the Max nodes is based on the value of β; if the current value of a Max node exceeds β, Min will choose the branch with the lower β value. At every Max node α is updated before it is passed to lower levels. Correspondingly, at Min's nodes the β value is updated and the nodes are pruned based on the value of α. An example of Alpha-Beta Pruning on a simple game tree is shown in Figure 2.2; the nodes in gray will not be expanded while computing the minimax values. The complete Alpha-Beta Pruning algorithm is shown in Algorithm 2.

The Alpha-Beta Pruning algorithm can reduce the time complexity of Minimax Search to $O(b^{\frac{m}{2}})$ in the best case. The reduction depends on the processing order of the successor nodes and their values. If the ordering is optimal the time complexity will be $O(b^{\frac{m}{2}})$, however for a random ordering the average time complexity will be $O(b^{\frac{3m}{4}})$. In the worst case, there will be no reduction at all and the time complexity will be exactly the same as that of MiniMax.

Algorithm 2 The MiniMax algorithm with α-β Pruning for search in game trees.

```
action := MiniMax Alpha-Beta Decision (state)
inputs: a state
returns: an action
  $u \leftarrow$ MaxValue (state, $-\infty$, $+\infty$)
  return action $\in$ Successors(state) with value $u$

utility value := MaxValue (state, $\alpha$, $\beta$)
inputs: a state, $\alpha$, $\beta$
returns: a utility value
  if TerminalTest(state) = true then
    return Utility(state)
  end if
  $u \leftarrow -\infty$
  for all $s \in$ Successors(state) do
    $u \leftarrow \max(u$, MinValue (state, $\alpha$, $\beta$))
    if $u \geq \beta$ then
      return $u$
    end if
    $\alpha \leftarrow \max(\alpha, u)$
  end for
  return $u$

utility value := MinValue (state, $\alpha$, $\beta$)
inputs: a state, $\alpha$, $\beta$
returns: a utility value
  if TerminalTest(state) = true then
    return Utility(state)
  end if
  $u \leftarrow +\infty$
  for all $s \in$ Successors(state) do
    $u \leftarrow \min(u$, MaxValue (state, $\alpha$, $\beta$))
    if $u \leq \alpha$ then
      return $u$
    end if
    $\beta \leftarrow \min(\beta, u)$
  end for
  return $u$
```

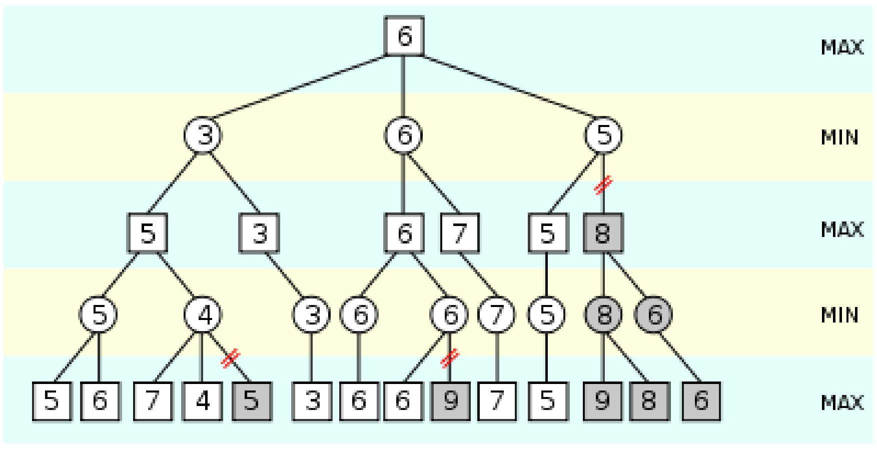

Figure 2.2: Alpha-Beta Pruning

2.1.5 Evaluation Functions

For any non-trivial game, the corresponding game tree grows rapidly at exponential rates and therefore running MiniMax with or without Alpha-Beta Pruning is simply impractical for making decisions in reasonable time. In practice, these algorithms can only expand the game tree up to a small depth in the order of 5 to 10, depending on the game. So, unless the current state of the game is near the end of the game with only a few moves left, MiniMax will not be able to return an optimal decision in reasonable time. However, the most important decisions in a game are probably those taken early on or around the middle of the game. How can MiniMax and Alpha-Beta become useful in such cases?

The key lies in finding an evaluation function which, given any game state, returns a numeric value that estimates the minimax value of the corresponding node. The availability of such a function allows cutting-off the MiniMax search at a certain depth and considering the nodes there as terminal nodes giving them values using the evaluation function. These values propagate to the root exactly as if they were payoff values. The net gain is that the decision at the root is based on previewing a significant number of moves ahead the current state. Even if the evaluation function is somewhat inaccurate, MiniMax with cut-off will yield good decisions. Note that Alpha-Beta Pruning can be helpful at doubling in the best case the depth at which cut-off is invoked, therefore allowing for even better decisions.

There is no recipe for designing perfect evaluation functions as they strongly depend on the game and its state space. In most cases, experience and domain expertise can be helpful in isolating a number of useful state features that can be used to learn a parametric evaluation function. The basic background behind the learning techniques used in games are outlined in the next section.

2.2 Sequential Decision Making and Learning

2.2.1 Markov Decision Processes

The task of a decision maker is greatly simplified if each decision is based only on the current state information and not on the entire history of states in the past. This implies that the state description is rich enough to retain all the needed past information. This independence from the past property is known as the **Markov property** [2]. It should be clear that in a Markovian process, a state needs not retain the information of how it was produced or the whole sequence of everything that lead to it, as long as the information that needed for succeeding states can be produced. For example, in chess, if we have all the information about the board, i.e. which pieces are where, we have all the information we need in order to produce future states; the sequence of actions which led the pieces to their positions is not needed. The most important consequence of the Markov property is that state transitions depend only on the current state and the action chosen, and therefore the decision in each state can be based without loss solely on the current state.

A **Markov Decision Process** (MDP) is a formalization used in describing sequential decision problems, whereby a decision maker needs to make sequences of decisions with long-term effects. An MDP is a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where:

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions

- $\mathcal{P}$ is a Markovian transition model, where $\mathcal{P}(s'|s,a)$ is the probability of transition to state s' when taking action a in state s
- $\mathcal{R}$ is a reward function, where $\mathcal{R}(s,a,s')$ is the reward for a transition from state s to s' with action a
- γ is the discount factor for future rewards, $\gamma \in [0,1)$

An MDP is a discrete-time process and the objective to choose actions at every step so as to maximize the cumulative reward in the long-term. A policy π is the "strategy" of the agent, that is a function that chooses an action in each state. Each policy when adopted for making decisions yields a long-term reward. The quality of a policy over the state space is characterized by its value function, which expresses the expected, total, discounted reward:

$$V^{\pi}(s) = E_{\alpha_t \sim \pi : s_t \sim \mathcal{P}} \left(\sum_{t=0}^{\infty} \gamma^t r^t \;\middle|\; s_0 = s \right) \tag{2.1}$$

A policy that maximizes the value function over all states in the state space is called an optimal policy. Provided the full model of an MDP, an optimal policy can be derived using any of the following methods: Value Iteration, Policy Iteration, Linear Programming.

2.2.2 Reinforcement Learning

Reinforcement Learning [1, 2, 3] is the problem of learning a good policy in an unknown enviroment which could be described as an MDP model, but this model is not availalble. Learning is based on interacting with the environment and observing the effects (rewards, transitions) of action choices in different states. The training data in reinforcement learning typically come in the form of (s,a,r,s') samples, meaning that at some point in time the agent chose action a in state s, received a reward r, and observed a transition to state s'. The agent does not have any prior knowledge so he must try different actions in different states in order to see which ones are good and which are bad in each state. An action will influence the next state of the environment, but it may also influence all succeeding ones. These facts reveal two important aspects of reinforcement learning: the *trial-and-error* approach and the *delayed reward*.

Problems in reinforcement learning come in two flavors: *Prediction* and *Control*. In a prediction problem the agent executes a fixed policy and the goal is to learn the value function for that policy. On the contrary, in a control problem the goal of the agent is to learn a good policy that maximizes the expected return.

Reinforcement learning methods are also divided into two categories: *on-policy* and *off-policy*. An on-policy learning method requires the agent to execute the policy π he is learning about, whereas an off-policy method gives the freedom to the agent to execute any arbitrary policy.

2.2.3 Value Function Approximation

In both prediction and control problems, the representation of a value function is a central issue. In finite state spaces, the simplest form of representation is a tabular form, whereby a big table stores one value of each state. In practical problems involving huge or continuous state spaces, this approach is simply impractical. The solution is to approximate the value function in some compact way at the cost of lost precision. *Value function approximation* [2, 4] refers to methods of generalizing a compact value function over the entire state space. In this manner, V^π is not seen as a real-valued vector, but rather as a function that maps values to states. It is parameterized with a parameter vector w. An alteration in the vector of parameters, often called *weights*, changes the whole estimation of V^π in several states. This means that a sample in any specific state may affect a series of values in other states.

The most common type of value function approximation involves the use of a *linear approximation architecture*. In these architectures, the approximate value is produced as the weighted sum of a number of *features* or else called *basis functions*. The parameters of the approximator are the weights multiplying the features. Any state s is mapped to a feature vector first:

$$\phi(s) = \big(\phi_1(s), \phi_2(s), ..., \phi_k(s)\big)^\top \tag{2.2}$$

and to a real value as follows:

$$\tilde{V}^{\pi}(s) = \sum_{j=1}^{k} \phi_j(s) w_j \tag{2.3}$$

The problem of learning a value function is now reduced to a problem of learning an appropriate set of parameters (weights). The number of basis functions and their representation is a lot smaller than the tabular representation of V^{π} and as a result they are used in cases where the state and action spaces are so big that its practically impossible to represent.

The basis functions of a linear architecture must be *linearly independent* to avoid singularities. The choice of basis functions can be seen as a way to impart prior knowledge to the system as they capture important knowledge about the state of the eneviroment.

2.2.4 Temporal Difference Learning

Temporal Difference (TD) learning [2] is an on-policy method for prediction. It utilizes the fact that value differences between temporally-close states (one step apart) are observable through the reward received during the transition from one state to another. The TD update equation for a sample (s, a, r, s') is the following:

$$V(s) \leftarrow V(s) + \eta \left(r + \gamma V(s') - V(s)\right) \tag{2.4}$$

where η is a small positive value known as the learning rate. This simple update equation assumes that the learned value function is stored in a table. If the value function is approximated by a linear architecture with k features, the update equation for a sample (s, a, r, s') is modified as follows to update the weights directly:

$$\forall i = 1, 2, \ldots, k,\ w_i \leftarrow w_i + \eta \phi_i(s) \left(r + \gamma \tilde{V}(s') - \tilde{V}(s)\right) \tag{2.5}$$

where $\tilde{V}(s) = \phi(s)^{\top} w$. TD is attractive because of its simplicuty. It has also been shown that TD converges given a sufficient large number of samples.

2.2.5 Least Squares Temporal Difference Learning

The **Least Squares Temporal Difference Learning** (LSTD) algorithm [5, 6] is another on-policy prediction method, similar to TD learning. The main difference with TD learning is that LSTD works only with linear approximation architectures and processes a batch of samples collectively entering them into a linear system, which is solved at the end to yield the learned weights of the approximation. The linear system of LSTD corresponds to a fixed-point approximation of the value function; for k features the size of the systems is $k \times k$ and takes the form $Aw = b$, where A is a $(k \times k)$ matrix and b is a $(k \times 1)$ vector. The LSTD update equations are the following for any sample (s, a, r, s').

$$A \leftarrow A + \phi(s)\big(\phi(s) - \gamma\phi(s')\big)^\top \tag{2.6}$$

$$b \leftarrow b + \phi(s)r \tag{2.7}$$

A full description of the LSTD algorithm is shown in Algorithm 3.

LSTD makes efficient use of the samples to find an appropriate set of weights for the linear architecture without requiring a learning rate parameter and independently of the order of sample presentation. Given the same approximation architecture, in the limit of infinite samples, TD and LSTD will everntually converge to the same solution.

2.3 Learning in Games

The connection between games and learning should now be apparent. Playing a game is a sequential decision problem. Most games are Markovian and can be described by an MDP. Following a specific strategy corresponds to executing a fixed policy. If the opponent is considered to be a “noisy” part of the environment, then the minimax value on the nodes of the game tree is simply the value function of our strategy/policy in that state. If the minimax value at the root of the tree corresponding to state s was backed up from the child of action a coming from some terminal node corresponding to state s' and the payoff/reward received between the two states was r, then we have exactly the samples (s, a, r, s') needed to solve the prediction problem of learning a good evaluation function of any given game. These connections are exploited in the context of this thesis, as described in detail in the following chapters.

Algorithm 3 Least Squares Temporal Difference Learning Algorithm

```
$w^\pi$ := LSTD $(D, k, \phi, \gamma, \pi)$
**inputs**:
$D$: Set of samples $(s, a, r, s')$ collected using policy $\pi$
$k$: Number of basis functions
$\phi$: Basis functions
$\gamma$: Discount factor
$\pi$: Policy whose value function is to be approximated
**outputs:**
$w^\pi$: the parameters of the approximate value function

  $A \leftarrow 0$ // $(k \times k)$ matrix
  $b \leftarrow 0$ // $(k \times 1)$ vector
  **for** all samples $(s, a, r, s') \in D$ **do**
    $A \leftarrow A + \phi(s)\big(\phi(s) - \gamma\phi(s')\big)^\top$
    $b \leftarrow b + \phi(s)r$
  **end for**
  $w^\pi \leftarrow A^{-1}b$
  **return** $w^\pi$
```

Chapter 3

Problem Statement

Even from the the origins of Artificial Intelligence, games played an important role, as games where thought as complex, demanding, and intriguing environments that could be used in order to test "intelligent" behavior for an agent. Reinforcement Learning exploits game in the same way. Many learning algorithms were created with a game as a motivating starting point. Tetris is a game that has become a standard benchmark in Reinforcement Learning.

3.1 Tetris

Tetris is a video game created in 1984 by Alexey Pajitnov, a Russian computer engineer. The game is played on a board of dimension 10×20 using seven kinds of simple tiles, originally called *tetraminoes*. All of them are composed of four colored blocks (*minoes*) forming a total of seven different shapes (Figure 3.1). Every tetraminoe has its own unique color. The rules of the game are very simple. The tiles are falling from the top of the board and the goal is to place them on the board so that lines are completed without gaps. The player has six actions available: rotate the tile by 90 degrees clockwise or counterclockwise, move the tile one step to the right or to the left, drop the tile down, or do nothing. The player can move the tile as long as it is *falling*. Once the tile rests on top of existing tiles in the board it cannot be moved anymore. When a line is fully completed it is eliminated from the board. The game ends when a resting tile reaches the top of the board. To prevent this, the player has to complete as many

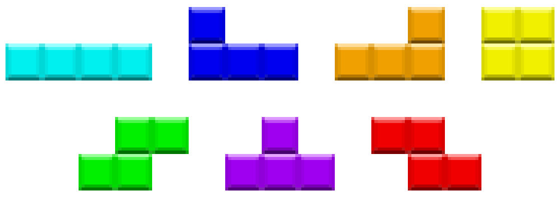

Figure 3.1: The seven tetraminoes.

lines as possible. If the player completes more than one lines simultaneously he is rewarded more than if he completed the same number of lines one by one. Figure 3.2 shows a snapshot of the board during a typical Tetris game.

Tetris is a very demanding and intriguing game, even though its rules are very simple. It needs a skillful player, in the sense that it not easy to form a strategy in this game and it needs a lot of experience in order to perform well. It is proven by E. D. Demaine, S. Hohenberger, and D. Liben-Nowell [7] that a strategy that tries to maximize the number of completed rows, to maximize the number of the lines eliminated simultaneously, to minimize the board height, or to maximize the number of tetraminoes placed in the board before the game ends is an **NP-complete** problem. Furthermore, it is shown that an optimal strategy is **NP-hard**, even to approximate it. This inherent difficulty is one of the main reasons why this game is widely used in order to test Reinforcement Learning algorithms.

Tetris was first formulated as a Markovian Decision Process (MDP) in 1996 by John Tsitsiklis and Benjamin Van Roy [8]. The state space of the game is spanned by two binary vectors. One vector has 200 dimensions and represents the 10×20 board. Each bit in this vector is 1, if there is a mino in the corresponding cell of the board, and 0, if the corresponding cell is empty. The second binary vector has 7 dimensions and it only has one active bit that represents the falling tile. In the MDP model there are as many decisions as there are actions, about 40 (10 columns and 4 rotations). If an action is taken in a state which includes the current board and the falling tile, the next state includes the new board with the falling tile placed at the corresponding position (less any removed lines) and a new falling tile. This MDP is deterministic as with a given tile, action and board

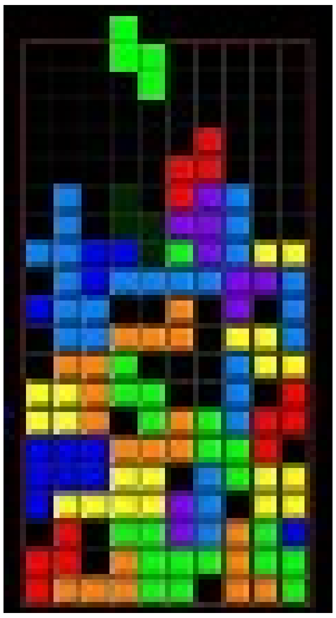

Figure 3.2: Tetris board.

configuration it is definite what the next state will be. The distribution over the falling tiles is uniform, so the seven tiles are chosen with equal possibilities. The reward function gives positive numerical values according to the lines completed by the player exactly as the original game. In this MDP formulation the goal is to maximize the long-term reward. The only difference with the video game is that a state does not include any preview information about the falling tile in the next state.

3.2 Adversarial Tetris

At the recent Reinforcement Learning Competition in 2009 [9] the model of the MDP by John Tsitsiklis and Benjamin Van Roy [8] was used as a basis for modelling a variation of Tetris, called **Adversarial Tetris**. This variation has a number of differences with the original game that was described above. The main difference is the adversarial aspect of Tetris. At this variation the new falling tile generator of the environment of the game is replaced by an opponent. The tiles are not randomly chosen anymore, but are chosen with the motivation to hinder the formation of complete lines on the board. The adversary may never choose a tile, if it helps the player to complete lines.

The MDP model of the Adversarial Tetris was formulated according to the model of the original Tetris described above. There are some differences though. The main difference is the fact that the distribution of falling tiles is non-uniform. Another difference is that the dimension of the board varies in height and width. A last difference is that the state includes the current position and rotation of the falling tile in addition to the configuration of the board. This means that the state is produced like the frames of the video game. For example, a state has the new falling tile at the top of the board and the configuration of the rest of the board. The next state will have the same falling tile, but placed one level down wherever the action of the player moved it from its previous position. If the player has not dropped the tile, it will be still floating in the board. The game does not advance to the next time step, unless the player chooses an action first, therefore in theory the player has infinite time to make an action choice.

The RL-Competition offers a generalized MDP model for Adversarial Tetris which is fully specified by four parameters (the height and width of the board and the adversity and type of the opponent). For the needs of the competition 20 instances of this model were specified with the parameters shown in Table 3.2. It should be noted that the adversity and the type of the opponent were not known until after the end of the competition and even now it is unclear how the adversity value and the type number affect the behavior of the opponent.

Another point that needs to be noted, although it is not a difference, is that the reward for eliminating multiple lines at once is greater than the reward for eliminating the same number of lines one by one. The competition reward function assigns positive numerical values to eliminating lines but the reward given differs from MDP to MDP, so that the total reward from all MDP can be normalized and fitted to the *difficulty* of every MDP. However, the competition framework does not penalize the agent for losing a game. Another important aspect is that there is no time restriction on making decisions.

In the 3rd RL-Competition the cumulative reward of the agent was being tracked for any number of steps involving multiple continuous games. Participation in the competition was achieved through a test run, which required the

MDP Number	Width	Height	Adversity	Type
0	6	16	0.194066	1
1	10	21	0.554109	1
2	7	19	0.172945	0
3	7	21	0.437863	1
4	11	22	0.474675	2
5	6	18	0.020393	2
6	6	19	0.027500	1
7	8	21	0.067066	2
8	9	21	0.232561	1
9	6	21	0.337115	3
10	9	25	0.587398	1
11	9	20	0.228675	3
12	9	20	0.039145	2
13	7	18	0.475067	3
14	11	22	0.241642	2
15	6	21	0.324070	2
16	7	22	0.402114	2
17	10	24	0.152057	2
18	10	24	0.025430	3
19	6	20	0.278261	2

Table 3.1: The 20 instantiations of the generalized MDP used for the competition.

learning agent to play continuously for a large number of steps (in the order of millions) and accumulate as much reward as possible, while the underlying MDP was being switched in some unknown way. The purpose was to demonstrate successfull learning over a wide range of game variations represented by the above MDPs and rapid adaptation to a changing environment. Every team in order to compete had to complete only one testing run.

The competition ended on June 8, 2009. Due technical difficulties only two teams were able to successfully participate. The results were announced at the RL-Competition Workshop on June 18, 2009 in Montreal, Canada during the

International Conference on Machine Learning. The winner was the team of Rutgers University, USA with a score of 1,500,000, followed by the team from the Nanjing University, China with a score of about 1,300,000.

3.3 Thesis Goal

In this thesis we try to address the problem of learning for Adversarial Tetris as descibed above with one difference. As mentioned, in the competition the cumulative reward of the agent is being tracked for any number of steps involving multiple continuous games. This leads to an agent that has only one goal, to maximize the number of completed lines over a number of steps independently of how many games he lost. There lies the alteration of our approach to this problem. In our case the agent is penalized for losing a game. In this manner, the agent has to achieve a more complex goal, to complete as many lines as possible, but within a single game. This formalization is closer to the task a human player has to achieve when playing this game. This modification, however, does not affect the use of the environment and the experiments given by the RL-Competition. It only affects the criteria that drive the agent's behavior. The environment and experiments that we used are those given by the RL-Competition committee.

3.4 Related Work

There is a lot of work on Tetris since its formulation as an MDP in 1996 till now. Tsitsiklis and Van Roy [8] created the model in order to apply their Approximate Value Iteration algorithm using a linear architecture with a few basis functions for value function approximation. On the same route was the work of Bertsekas and Tsitsiklis in 1996 [10]. Also, Bertsekas and Ioffe [11] in the same year introduced a Temporal-Difference Policy Iteration algorithm that also used a linear architecture in order to approximate a cost function. In 2001, Kakade [12] introduced a gradient-descent method, named natural policy gradient, and applied it to Tetris. In 2002, Lagoudakis, Parr, and Littman applied Least-Squares approach to Tetris. Another approach to Tetris was taken by Ramon and Driessens [13] who modeled it as a relational reinforcement learning problem and applied a regression technique using Gaussian processe to predict Q values. Another approach to Tetris was that of de Farias and Van Roy in 2006 [14] who used the technique of randomized constraint sampling in order to approximate the optimal cost function with the aid of a linear architecture of basis functions. The same year Szita and Lőrincz introduced the noisy cross-entropy method [15] for reinforcement learning problems and applied it to Tetris in order to prove its efficiency. In the context of the *Second Reinforcement Learning Competition* in 2008 Thiery [16] used λ -Policy Iteration to solve a linear architecture based on the features of D. P. Bertsekas and Ioffe [11]. This approach outperformed all previous work at the time.

The work described above was done based on the original form of Tetris game. There has been unpublished work on Adversarial Tetris in the context of the *Third RL Competition* in 2009. The only published part of this work was presented at the ICML Workshop in 2009 [17] by the team of Rutgers University. They applied look-ahead tree search algorithm where the opponent in each MDP was taken as a fixed probability distribution over falling tiles learned from data and the cross entropy method for the approximation of the value function. There is also unpublished work by the team of Nanjing University, China.

Chapter 4

Our Approach

4.1 Player Actions

In the original form of the video game of Tetris the player has a time limit for his decision, which is the time a tile needs to fall completely. In Adversarial Tetris the tile is falling one step downwards every time the agent chooses an action and the agent has no time restrictions in order to make a decision. The action can be one from the six available low-level actions for the step-by-step manipulation of the tile, meaning that the player can move the falling tile only one step at a time and the environment adds a downward move of one step to the player's move. The player has six low-level choices: move the tile left or right, rotate it clockwise or counterclockwise, drop it, and do nothing. Using a sequence of these actions the player can place the tile to any desired position. There is no unique sequence of such actions in order to place a tile to a specific position and with a certain rotation. For starters, there are many equivalent rotations for most of the tiles, in addition to the fact that one clockwise rotation is equivalent to two clockwise rotations on the same tile and so on. Furthermore, the player can reach a certain position by moving left and right, but also by moving right and then left, or by repeating this pattern a couple of times. There is no restriction apart from the configuration of the board for the agent's choise of low-level actions.

Playing at the level of the six low-level actions ruins the idea of a two-player alternating game, as the opponent's turn is only once after several turns of the player. Also, the branching factor of six in each choice of the player would lead to an intractable game tree, even before the falling tile reaches a permanent position in the tree. Finally, this aspect of having many sets of low-level actions leading to the same configuration of the board and thus repeated states in the game tree would lead to another unnecessary growth of the tree. These observations lead us to consider an abstraction of the player's moves, high-level actions, that are the actions that bring the tile from the top of the board to its final position. These actions referred as high-level actions are interpreted by the agent to a sequence of low-level actions that are selected only by the criteria to be the minimum action set that can be created for the desired position.

The importance of this abstraction is more evident when looking at the game tree. We assume that Max is our player and Min the opponent. The game tree for alternating playing is formed by creating the nodes that represent the actions of Max followed by the nodes that represent the actions of Min and so on. However, if there was no abstraction, Max would have to make an arbitrary set of moves till the tile was placed and then it would be Min's turn. As a result, the game tree would not a balanced structure. Moreover, all these nodes that would be created in between the first occurence of the new tile till its final placement would not provide the game tree with any more useful information than the final node representing the node with the tile's final position.

On the other hand, the opponent (Min) does not only decide which tile will be the next falling tile in the board, but also with which rotation it will be placed at the top of the board. This means that the opponent has as many actions as there are tiles multiplied by the number of possible rotations to these tiles, which is $4 \times 7 = 28$. However, not all these actions are needed in order to represent the opponent's moves, as the player (Max) can use low-level actions to rotate the tile to another configuration. For the great majority of cases the original rotation of a tile does not affect the final placement of the tile. This is not true only for a small number of board configurations, where the pile of the tiles has almost reached the board's top. The number of those cases is so small and their effect on the agent's game play is minimal, as these conditions will inevitably lead to a loss within a short time. Consequently, the drawback of ommitting them is negligible compared to the benefit of reducing the braching factor at Min nodes from 28 to just 7. This reduction becomes even more imperative, given that the branching factor of Max is already high.

In summary, both players will choose high-level moves. There are 7 choices for the Min (the 7 tiles) and at most 40 choices for the Max (10 columns, 4 rotations).

4.2 Minimax Game Tree

The **Minimax Game Tree** represents all the possible paths of action sequences of the two players Max and Min playing in alternating turns. The game tree is needed in order for the player to form a strategy against its opponent, so its formation should take place when it is the agent's turn to play. Therefore, our player forms a new tree every time the opponent make his move choosing a tile.

Max using the high-level actions has as many possible actions as are the possible placements of the falling tile on top of the board configuration. The number of these placements, however, varies according to the height and width of the board, the falling tile and its number of minimum rotations, and the configuration of the board. These conditions define a varying branching factor for Max, which in our case is our agent. This factor may vary from 0 for the terminal states, were there are no places left in the board for the falling tile to the maximum numerical value for given board dimensions, tile and an empty board. For example, in an empty board of size 10×20 there are 9 places available for the square tile and since its shape does not differentiate over rotations, there are going to be 9 actions available. An approximation of the upper bound of the board's branching factor for Max is $4 \times width$ and taking into consideration the rotations that have no effect to the tiles posture the branching factor for an empty board is $19 \times width - 28$ - this is computed on the bases of the shape and the meaningful rotations.

Since we have declared what the actions of each player are and when the game tree needs to be created we can now describe the procedure of its formation. The initial state of the game tree is the current board configuration in addition to the falling tile, where Min has already played and it is Max's turn to play, so the root of the tree is a Max node. The successor function for a Max node creates all

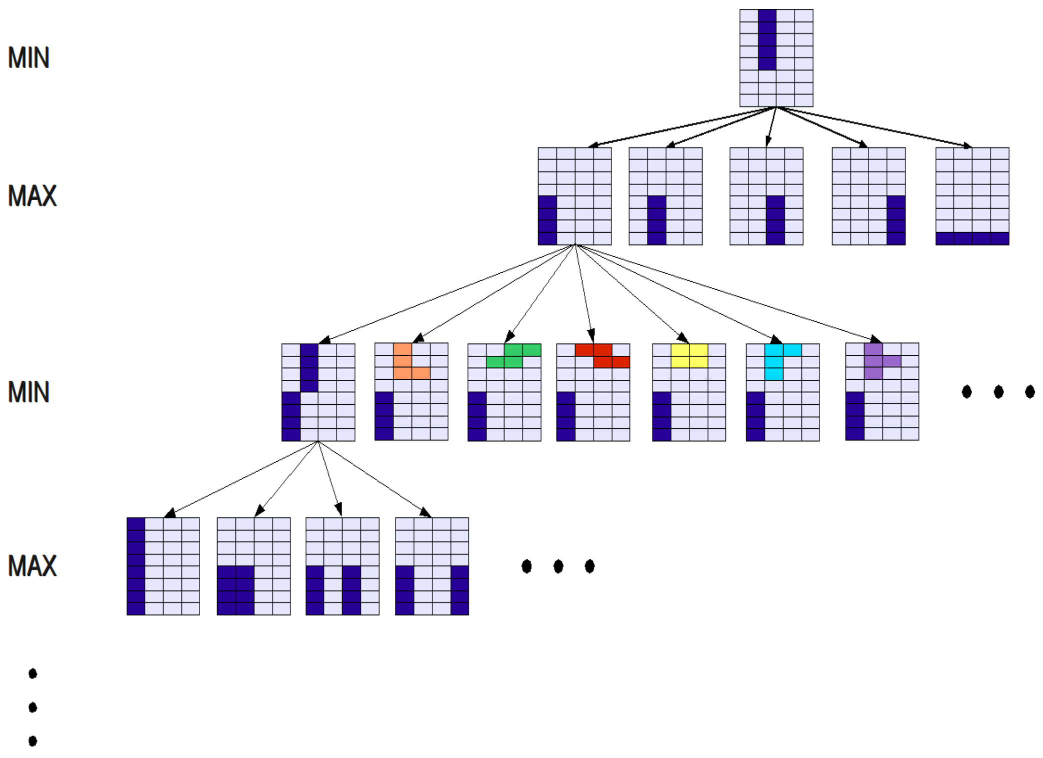

Figure 4.1: Adversarial Tetris MiniMax game tree.

of its children by placing the falling tile to all legal positions of the board. For each of these nodes, according to the Minimax Search algorithm, the successor function creates 7 children one for every choice of action of Min. Figure 4.1 depicts a partial game tree for Adversarial Tetris. The expansion of the tree continues until the cut-off depth of the tree is reached. The cut-off depth is the depth where the algorithm stops expanding. The utility of the nodes at the cut-off depth is estimated by an evaluation function described below.

The computation of the Minimax Value is done with the aid of the Alpha-Beta Pruning in order to prune any nodes and sub-trees that do not contribute to final decision of the player. However, the pruning of nodes by the Alpha-Beta Pruning algorithm does not eliminate the effect of such a big branching factor of the game tree and the depth that our computational resources permit does not exceed the cut-off depth 3.

4.3 Evaluation Function Approximation

The estimation of a board configuration whether in favor or against our agent is done by an evaluation function, which also implicitly determines the agent's policy. It is the means of the agent to perceive the effects of its actions and learn how to perform well in this environment. In order to learn a good evaluation function that suits our goals we apply Reinforcement Learning for learning a state value function V. This value function in this environment with the large state space can not be computed but must be approximated.

We are using a linear architecture in order to approximate the value function. This linear architecture is formed by a set of basis functions $\phi(s)$, and a set of weights w. We have issued two possible sets of basis functions with different properties which will eventually lead to two different agents. The first set includes 6 basis functions for characterizing the board as shown in Table 4.3.

Feature	Description
1.0	a constant term
h_{max}	the maximum height of the board
$\tilde{h}$	the mean height of the board
L	holes, the total number of cells below placed tiles on the board
$\sum_i \|h_i - h_{i+1}\|$	the sum of absolute column differences in height
G	gaps, the empty cells above placed tiles up to max height

Table 4.1: Set of State Features

The second set of basis functions uses a separate block of the 6 basis functions described above for each one of the 7 tiles of Tetris, giving a total of 42 basis functions. This is proposed because with the previous set the agent learns which boards and actions are better for him, but does not associate them to the falling tiles that these actions manipulate. The same action on different tiles, even if the board is unchanged, may have a totally different effect; ignoring the type of tile falling in the current board leads to a less efficient behavior. This second set of basis functions take into account both the board and the falling tile and therefore is appropriate for evaluating Max nodes. It should be noted that only one block of size 6 is active in any state, the block corresponding to the falling block.

4.4 Incremental Least-Squares Temporal Difference Learning

We applied a variation of **Least-Squares Temporal Difference Learning** algorithm in order to learn a good set of parameters for our value function. The need for modifying the original LSTD algorithm stems from the fact that the underlying agent policy is determined through the values given to states by our evaluation function, which are propagated to the root; if these values changes, so does the policy, therefore it is important to discard old data and use only the recent ones for learning. To this end, we used the technique of **exponential windowing**. According to this technique, the weights are updated in regular intervals called **epochs**; each epoch may last for several decision steps. During an epoch the underlying value function and policy remain unchanged, therefore all data collected are inserted into the matrices A and b of LSTD. At the completion of the epoch, the system is solved and the weights are updated. In the next epoch, the matrices A and b of LSTD are not initialized to 0, but to the A and b matrices of the previous epoch discounted by a parameter μ. This initialization is useful because the previous data are not completely eliminated, but are weighted less and less as they become older and older. Their influence to the solution of the system varies according to the numerical value of μ which takes values between 0 (no influence) to 1 (full influence). A value of 0 leads to singularity problems due to the shortage of samples within a single epoch, however a value around 0.95 offers a good balance between recent and old data with exponentially decayed weights. We call this variation **Incremental LSTD**; a full description of the algorithm is given in Algorithm 4 (t indicates the epoch number).

In order to accomodate a wider range of objectives we used a rewarding scheme that encourages line completion (positive reward), but discourages loss of a game (negative reward). We balanced these two objectives by given a reward of +1 for

Algorithm 4 Incremental Least Squares Temporal Difference Learning

```
$w^t, A^t, b^t$ := incLSTD $(D^t, k, \phi, \gamma, w^{t-1}, A^{t-1}, b^{t-1}, \mu)$
**inputs**:
$D^t$: Samples $(s, a, r, s')$ collected using policy $\pi^{t-1}$ derived from $w^{t-1}$
$k$: Number of basis functions
$\phi$: Basis functions
$\gamma$: Discount factor
$w^{t-1}$: the parameters of the approximate value function at epoch $t-1$
$A^{t-1}$: $(k \times k)$ matrix $A$ at epoch $t-1$
$b^{t-1}$: $(k \times 1)$ vector $b$ at epoch $t-1$
$\mu$ : the exponential windowing factor
**outputs:**
$w^t$: the parameters of the approximate value function at epoch $t$
$A^t$: matrix $A$ at epoch $t$
$b^t$: matrix $b$ at epoch $t$

  if $t == 0$ then
    $A^t \leftarrow 0$
    $b^t \leftarrow 0$
  else
    $A^t \leftarrow \mu A^{t-1}$
    $b^t \leftarrow \mu b^{t-1}$
  end if
  for all samples $(s, a, r, s') \in D$ do
    $A^t \leftarrow A^t + \phi(s)\big(\phi(s) - \gamma\phi(s')\big)^\top$
    $b^t \leftarrow b^t + \phi(s)r$
  end for
  $w^t \leftarrow {A^t}^{-1} b^t$
  return $w^t, A^t, b^t$
```

each completed line and a penalty of -10 for each game lost. We set the discount factor to 1 ($\gamma = 1$) since rewards/penalties do not loose value independently of when they occur.

Chapter 5

Implementation

5.1 RL-Glue Framework and the RL-Competition

RL-Glue [18] is an open-source *framework* that provides an interface that different agents, environments and experiments can ran together without the need to be written at the same programming language. It is a *set of common guidelines* provided to the reinforcement learning community in order to share and compare different agents and environments without much effort. The main functionality of it is that it is generalized, which means that the basic interface of an experiment, an agent or an experiment is available. This helps so that the code in order to connect the agent, the environment and the experiment.

The RL-Glue interface includes **RL-Glue Core Project**. The Core Project provides software in order to produce direct-compile project or a stand-alone server for running socket projects. The latter is used in order experiments, agents and environments to be linked over a socket protocol locally or over the Internet.

RL-Glue Extensions is a separate project that includes multi-language support. In *direct compile* mode all the components of a Reinforcement Learning Project are compiled together into a single executable program if it is written in C/C++. The other mode that is available, is the *socket* mode. The *socket* mode is a mode that involves linking of the an agent, an environment and an experiment over the Internet or locally. These three components can be written in any language from those supported. The *language-specific* software that allows programs from different languages to connect via the RL-Glue interface is called *codec*. The available codecs are the following: C/C++, Java, Matlab, Python and Lisp.

The RL-Glue Project uses the **task specification language, *task spec*** [19] . The basic notion of *task spec* language is to have a task specification standard. It is a string generated by the environment and is given to the agent as a parameter in *agent-init*. It is the representation of the interaction of the environment and the agent. It gives the information to the agent that is needed in order to form the "perception" of the environment. More specifically, it gives a representation of the observations, the actions and the range of rewards.

This thesis is based on the domain of the RL-Competition of Adversarial Tetris. The RL-Competition utilized the open-source project of RL-Glue. This provided an interface for the agent, the environment and the experiments. The environment and the experiments are software programs provided by the competition. The agent interface was provided from the RL-Glue Project; based on that a sample agent was made available from the competition organizing team.

The **technical details** [9] of the competition software refer to the observation and the action space. The **observation space** is high dimensional and a discrete valued space. It contains a representation of the current configuration of the *board* as a binary **bit map**, a **bit vector** for the *falling piece* and **two integers** that declare the board size, the number of *rows* and *columns*. The **action space** is one dimensional and discrete. It defines the manipulation of the falling piece. All actions have one integer value.

5.2 The agent architecture

The agent was written in **Java** programming language as it is supported by RL-Glue. The agent is based on the sample code given by the RL-Competition team and is compatible with the **RL-Glue agent interface**. The **Eclipse IDE** [20] was used at the development of the code and the **Sun Java Platform JRE** version 6 [21]. The operating system that it was programmed and used was a Debian based Distribution of Linux, **Ubuntu** [?]. This program has needs at least 2 GB RAM to run and preferably more than 3Gb. It was run to 64-bit systems but it can run in 32-bit systems also. It can run in any platform that

RL-Glue does. Which means in all operating systems and in any kind of system. The drawback is that it cannot be parallelized fully but it was tested and run to the Grid of the university.

The agent interface has the methods of **init, start, step, end, clean up, freeze** and **message** available to be used for the connection with the environment and the experiment. In our implementation the methods init, step, end and clean up were implemented and used. There were some classes in order for the **game tree** to implemented and for **board** to be represented as an entity. The methods of the agent analyzed the observation of the environment given by **task spec**. The agent used a **model** of the environment in order to produce the possible *future* states of the environment and form the game tree. Since the *states* of the MDPs represented at every *time step* the current placement of the *falling* piece and the actions were done at each time steps the agent did not make a *decision* at each time step. The agent needed to make a decision every time there was a *new falling piece*. As a result, the game tree was produced at that time so did the update of the learning algorithm. The decision of the agent was not an action but a *placement* of the new piece. In that way, the agent's decision was a *strategy* that included the *minimum* set of actions in order to place the piece to the desired placement on the board. The set was actions is referred above as "minimum" because there are many equivalent actions that can lead a piece to one specific placement with a specific orientation, the set that was used was the one with the less steps needed for the placement to be completed.

The learning algorithm's computation of the vector of *weights* was done at specified intervals of steps not games. The updates needed to be done only when new *experience* was offered to the agent and that was the case only when a decision was to be made and an *estimation* of the current board configuration. In order for the linear system to be solved a Java API was used, **Jama** package [22].

It is pointed out several times in this thesis that the *negative* payoff for losing a game is not given by the environment. The numerical value of this was given to the agent at the game tree. At the game tree we defined its value and there it contributed to the decision making of the agent. Also the cut off depth of the tree is defined there. The agent was tested at cut off depth one, three and five.

At cut off depth three the game tree was nothing more than a generator of the model of the environment.

The agent through its **step** method selected an action at a time as far as the environment is concerned, regardless of the fact that all the sequence of actions for a piece was already decisioned. The agent is able to perform in all the MDP's of the environment and to run in all the experiments that were given from the RL-Competition.

5.3 Environment and Experiments

The environment and the experiments were implemented by the organizing committee of the RL-Competition [9]. The **environment** includes the board, the pieces distribution and the *opponent*, the generator of the pieces' distribution. At the time that the agent was completed we had no knowledge of the opponent and its behavior apart from its goal. After the ending of the competition this code was released to publicity and we could see that there were more than one agents that played the role of the opponent with a sort of a scaling to their "evilness". However till this date we have no specific information about these opponents that varied from an MDP to another. The Adversarial Tetris was **generalized** as there were parameters that specified a certain MDP of the generalized problem. There were twenty parameters from zero to nineteen, at each of them another MDP was specified with different dimensions of board and opponent hardness. Most of our experiments are done at the MDP with parameter 1, as its board dimensions were similar to the original Tetris board.

The **experiment** software included a console trainer for Java, a GUI trainer and there was the proving software and the testing software. Unfortunately, because this agent did not compete at the competition and there were no local test runs done, there are not any results from that experiment software. Exactly because the formalization of this game is episodic the experiments were done in episodes. Every episode was a game. Every game was independent from the other as an episodic task would demand.

Chapter 6

Results

6.1 Experimental Design

Our learning experiments were conducted over a period of 400 epochs, where each epoch is a series of 8, game steps. At the end of each learning epoch the LSTD system is solved and the weights of the value function are updated. Therefore, the total learning steps were 3,200. Learning was conducted only on MDP 1 which has board dimensions that are closer to the board dimensions of the original Tetris. The exponential windowing parameter was set to 0.95. Learning takes place only at the root of the tree in each move, as learning to the internal nodes leads to a great degree of repetition biasing the learned evaluation function.

The agent with the first set of basis functions (= 6) learns by backing up values from depth 1 and from depth 3. This choice is dictated by the fact that this set of basis functions ignores the choice of the Min and therefore it would be meaningless to expand the tree beyond Min nodes which are found at odd depths. The second agent with the larger set of basis functions ($k = 42$) learns by backing up values from depth 2. This set of basis functions takes the action choice of the Min explicitly into account and therefore it makes sense to cut-off the search at Max nodes, which are found at even depths.

The testing experiments involved playing 500 games per MDP. The first agent, who learned with cut-off depth 1, was tested for cut-off depths 1 and 3. The same agent, who learned with cut-off depth 3, was tested for cut-off depths 1 and 3.

The second agent, who learned with cut-off depth 2, was tested for minimax tree cut-off depths 2 and 4.

6.2 Experimental Results

6.2.1 Learning Performance

Learning performance for the agent with the first set of 6 basis functions.

The first learning experiment is the agent to learn using a game tree with cut off depth 1 shown in figure 6.1. The second learning experiment is the same agent learning using a game tree with cut off depth 3 as shown in figure 6.2.

Learning Results for the agent with the second set of basis functions that combine the previous 6 basis functions with the falling piece is shown in Figure 6.3 .

6.2.2 Playing performance

In order to test the playing performance of the first agent we have run the following experiments. The first experiment was the agent with the 6 basis functions utilizing the weights learned at cut-off depth 1 play in all MDPs with cut-off depth 1 and then with cut-off depth 3. Another experiment was the same agent but this time utilizing the weights learned at cut-off depth 3 play in all MDPs with cut-off depth 1 and afterwards with cut-off depth 3. The average performance of this agent is 544 steps and 44 lines for all MDPs and 366 steps and 16 lines at MDP 1. In the table 6.2.2 all the results of the testing runs are shown.

The second agent proposed played in the environment of the adversarial tetris for all MDPs available utilizing the weights learned at the cut off depth 2 of the minimax tree, having cut off depth of the minimax tree 2 while playing. The has an average performance of 222 steps and 44 lines per game over all MDP's and 197 steps and 16 lines on MDP 1. Analytically the performance of the agent is shown in the table 6.2.2.

As a second performance testing, the second agent proposed played in the adversarial environment for all MDPs available utilizing the weights learned at

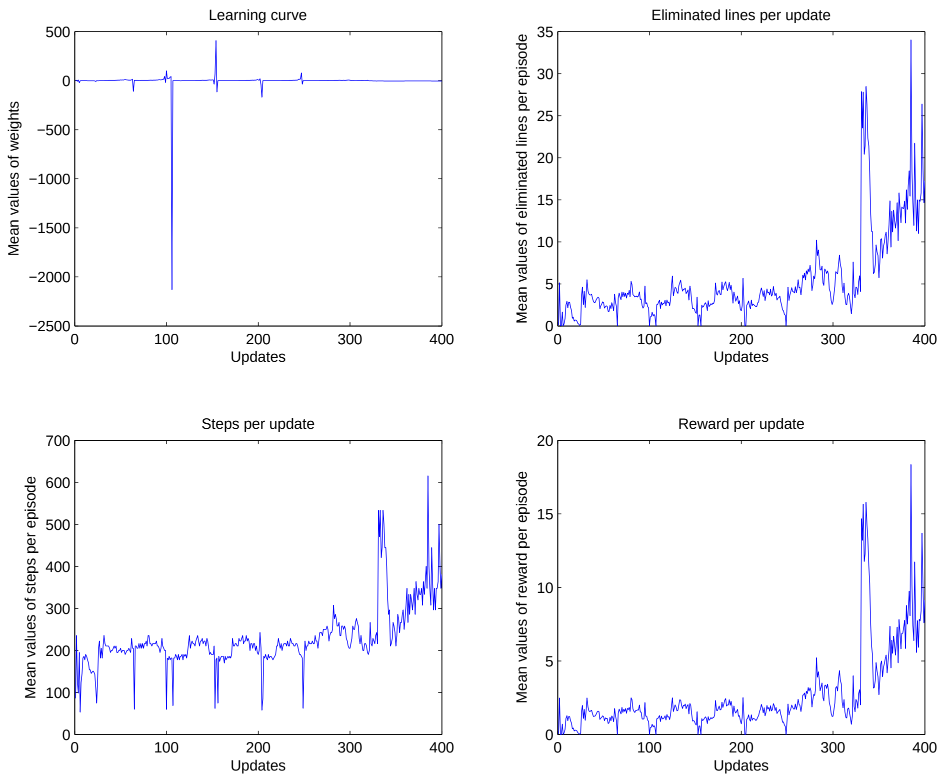

Figure 6.1: Learning performance for the agent with 6 basis functions learning at game tree cut-off depth 1

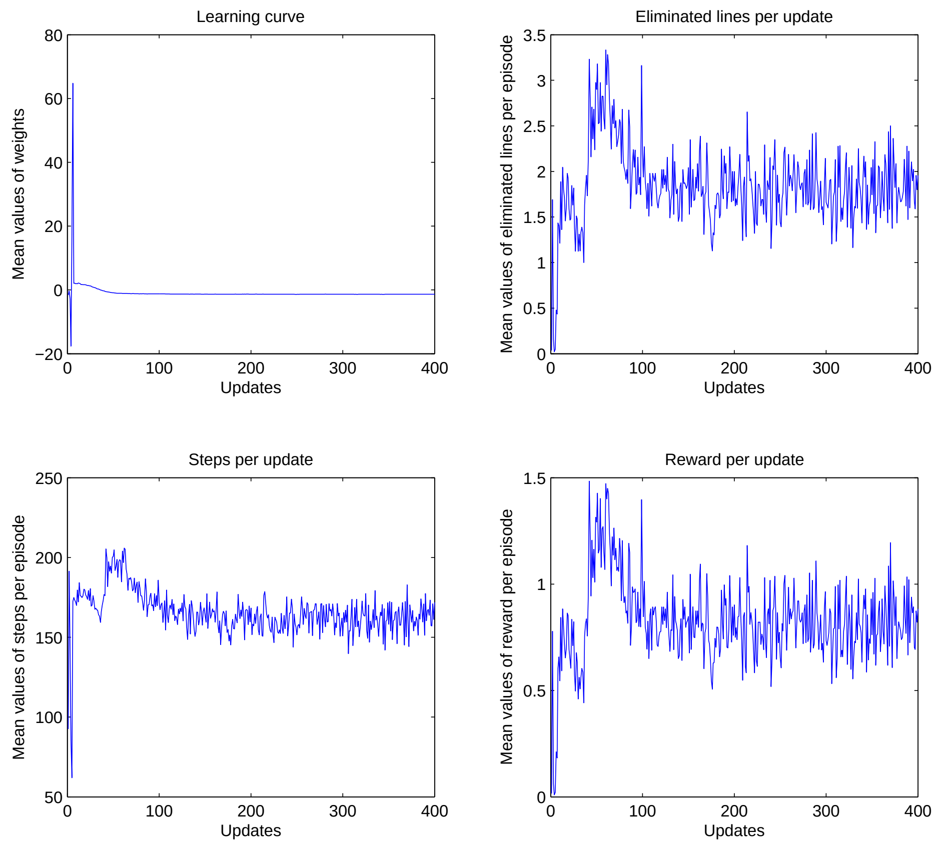

Figure 6.2: Learning performance for the agent with 6 basis functions learning at game tree cut-off depth 3

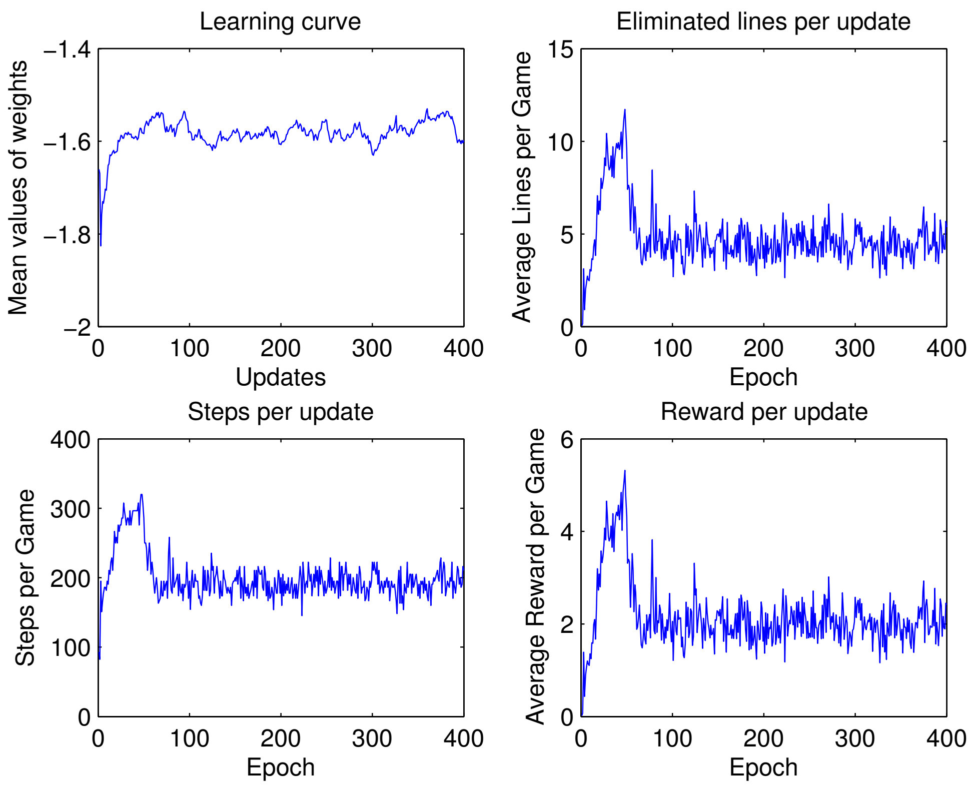

Figure 6.3: Learning performance for the agent with 42 basis functions learning at game tree cut-off depth 2

the cut off depth 2 of the minimax tree, having cut off depth of the minimax tree 4 while playing.

MDP		Lines			Reward			Steps	
	Min	Ave	Max	Min	Ave	Max	Min	Ave	Max
0	0	28.44	133	0	5.61	29	37	246.39	936
1	0	16.08	66	0	8.45	36.66	98	366.26	1048
2	0	31.81	180	0	6.55	37.34	54	348.13	1589
3	0	27.79	112	0	6.39	26.84	69	327.41	1035
4	0	21.93	93	0	10.9	46.67	76	514.77	1628
5	0	54.90	249	0	8.7	39.56	38	431.95	1726
6	0	41.78	202	0	6.62	32.59	38	344.68	1395
7	3	91.18	497	0.68	21.15	115.56	122	997.59	4928
8	0	54.15	284	0	16.43	86.52	68	755.15	3380
9	0	27.71	145	0	6.12	34.17	61	261.95	1041
10	1	22.62	95	0.36	9.97	43.48	109	426.77	1286
11	0	43.2	163	0	15.41	55.21	52	614.33	2
12	1	103.83	455	0.25	27.26	119.72	104	1312.71	5381
13	0	25.15	114	0	6.19	29.58	41	288.34	1047
14	0	54.47	258	0	20.23	97.32	117	1001.05	4127
15	1	24.22	122	0.14	5.34	26.44	72	241.97	917
16	0	27.26	106	0	7.81	30.39	54	328.99	1002
17	1	65.74	271	0.33	26.25	115.37	118	1047.04	3823
18	0	82.94	440	0	30.86	168.16	121	1273.60	6013
19	2	40.73	178	0.33	6.89	30.03	79	343.23	1276

Table 6.1: Testing results for the first agent playing at depth 1 with weights learned with a game tree with cut-off depth 1

MDP		Lines			Reward			Steps	
	Min	Ave	Max	Min	Ave	Max	Min	Ave	Max
0	0	2.97	16	0	0.48	3.22	25	66.56	164
1	0	0.82	8	0	0.37	3.98	67	127.22	236
2	0	2.07	12	0	0.41	2.42	29	76.78	181
3	0	2.31	14	0	0.51	3.35	39	86.63	201
4	0	1.01	9	0	0.45	4.66	67	150.91	306
5	0	4.04	23	0	0.62	3.51	32	81.70	217
6	0	3.59	16	0	0.55	2.45	31	80.93	178
7	0	3.02	18	0	0.68	4.07	44	119.23	274
8	0	2.32	17	0	0.66	4.92	47	121.08	349
9	0	3.65	14	0	0.61	2.36	33	88.55	167
10	0	1.59	8	0	0.59	3.95	53	122.81	246
11	0	1.93	9	0	0.56	4.64	48	111.05	221
12	0	2.52	15	0	0.64	3.86	47	120.88	290
13	0	2	16	0	0.42	4.05	30	73.312	196
14	0	1.39	10	0	0.51	3.67	61	158.16	318
15	0	3.21	14	0	0.54	2.70	32	84.99	168
16	0	2.19	11	0	0.48	2.75	37	87.54	188
17	0	1.65	14	0	0.57	5.29	81	162.19	359
18	0	1.82	14	0	0.61	4.80	79	163.78	349
19	0	4.10	18	0	0.68	3.12	32	87.95	199

Table 6.2: Testing results for the first agent playing at depth 3 with weights learned with a game tree with cut-off depth 1

MDP		Lines			Reward			Steps	
	Min	Ave	Max	Min	Ave	Max	Min	Ave	Max
0	0	15.16	63	0	2.40	10.4	28	156.93	507
1	0	8.72	50	0	3.91	21.64	66	265.43	798
2	0	14.07	45	0	2.79	9.08	38	202.7	479
3	0	15.63	48	0	3.43	10.74	39	228.1	505
4	0	9.61	49	0	4.17	22.91	74	328.21	931
5	0	20.77	90	0	3.11	14.05	44	202.26	675
6	0	19.35	74	0	2.91	11.63	42	196.1	544
7	0	19.64	79	0	4.41	18.09	43	295.57	882
8	0	17.29	70	0	4.98	21.70	67	323.6	970
9	0	17.61	58	0	2.93	11.78	51	194.01	468
10	0	11.83	41	0	4.42	16.53	57	296.83	641
11	0	16.12	96	0	4.74	27.35	42	300.63	1241
12	0	24.21	120	0	6.19	30.9	48	394.72	1517
13	0	13.80	44	0	2.89	9.52	49	195.59	432
14	0	16.81	59	0	5.92	21.67	67	424.62	1141
15	1	16.73	51	0.14	2.69	9.24	46	189.1	435
16	0	15.65	57	0	3.38	12.16	53	232.94	563
17	0	22.32	75	0	7.82	28.43	66	463.86	1151
18	0	21.95	78	0	7.31	26.91	74	461.57	1217
19	0	19.97	83	0	3.22	13.62	43	202.03	630

Table 6.3: Testing results for the first agent playing at depth 1 with weights learned with a game tree with cut-off depth 3

MDP		Lines			Reward			Steps	
	Min	Ave	Max	Min	Ave	Max	Min	Ave	Max
0	0	2.99	14	0	0.47	2.64	26	71.62	165
1	0	1.68	11	0	0.76	6.18	43	157.71	306
2	0	3.36	14	0	0.67	2.83	33	107.05	217
3	0	3.72	15	0	0.83	3.35	28	121.41	235
4	0	1.73	13	0	0.76	7.64	46	176.41	407
5	0	3.64	17	0	0.56	2.75	31	83.18	189
6	0	3.42	17	0	0.52	2.6	32	83.88	187
7	0	4.16	19	0	0.93	4.52	40	149.14	315
8	0	3.67	16	0	1.05	4.34	34	162.41	337
9	0	3.2	16	0	0.55	3.09	37	91.79	192
10	0	2.98	14	0	1.11	5.39	50	181.88	333
11	0	3.42	17	0	0.97	6.19	33	151.79	351
12	0	4.08	16	0	1.03	4.12	33	160.99	334
13	0	3.45	16	0	0.73	3.44	24	102.31	213
14	0	2.28	13	0	0.83	4.77	38	185.73	430
15	0	2.92	15	0	0.49	3.13	36	88.57	184
16	0	3.71	18	0	0.79	3.92	43	125.59	259
17	0	2.88	16	0	0.99	5.29	50	201.3	425
18	0	3.16	13	0	1.04	4.48	53	210.37	386
19	0	4.33	20	0	0.71	3.28	33	96.81	223

Table 6.4: Testing results for the first agent playing at depth 3 with weights learned with a game tree with cut-off depth 3

MDP		Lines			Reward			Steps	
	Min	Ave	Max	Min	Ave	Max	Min	Ave	Max
0	0	10.80	44	0	1.7	7.32	26	120.81	369
1	0	4.76	24	0	2.13	11.04	55	196.67	476
2	0	7.12	36	0	1.44	7.06	33	124.53	378
3	0	8	35	0	1.79	7.83	38	143.93	403
4	0	5.24	23	0	2.25	11.46	42	241.19	566
5	0	18.52	69	0	2.84	10.54	30	185.38	529
6	0	18.22	69	0	2.8	10.86	33	186.11	525
7	0	14.81	61	0	3.34	14.02	46	251.56	712
8	0	12.17	47	0	3.51	13.6	51	273.32	674
9	0	15.47	47	0	2.53	8.69	42	176.31	404
10	0	8.29	28	0	3.08	10.06	58	253.34	484
11	0	10.01	50	0	2.77	16.77	47	234.89	721
12	0	10.87	50	0	2.98	16.77	50	246.18	721
13	0	16.01	62	0	4.1	15.96	49	313.44	848
14	0	6.55	29	0	1.37	6.08	31	115.92	322
15	0	10.78	38	0	3.9	13.96	34	343.41	768
16	0	15.84	46	0	2.53	8.53	36	180.83	394
17	0	7.71	35	0	1.66	8.24	40	143.81	410
18	0	12.16	42	0	4.15	14.21	80	336.82	788
19	0	14.14	69	0	4.67	23.38	72	365.78	1144

Table 6.5: Testing results for all MDPs for the second agent at minimax tree cut off depth 2

MDP		Lines			Reward			Steps	
	Min	Ave	Max	Min	Ave	Max	Min	Ave	Max
0.0	0.0	1.284	12.0	0.0	0.199	1.904	24.0	45.924	135.0
1	0.0	1.122	12.0	0.0	0.504	5.300	50.0	133.298	307.0
2	0.0	1.026	11.0	0.0	0.207	2.422	33.0	64.134	176.0
3	0.0	1.250	10.0	0.0	0.278	2.237	32.0	76.544	163.0
4	0.0	1.468	10.0	0.0	0.660	5.940	53.0	169.338	352.0
5	0.0	1.740	15.0	0.0	0.264	2.291	27.0	58.624	164.0
6	0.0	2.0	11.0	0.0	0.305	1.683	31.0	64.206	150.0
7	0.0	1.910	13.0	0.0	0.431	2.940	43.0	102.744	242.0
8	0.0	1.896	13.0	0.0	0.548	3.762	45.0	124.496	279.0
9	0.0	2.292	17.0	0.0	0.385	3.093	36.0	74.544	194.0
10	0.0	1.806	10.0	0.0	0.673	3.953	57.0	151.252	280.0
11	0.0	1.754	12.0	0.0	0.494	4.386	44.0	117.140	259.0
12	0.0	1.912	15.0	0.0	0.494	3.862	37.0	116.302	294.0
13	0.0	1.048	10.0	0.0	0.219	2.026	28.0	59.220	156.0
14	0.0	1.952	13.0	0.0	0.712	5.141	42.0	177.108	415.0
15	0.0	2.016	13.0	0.0	0.333	3.269	34.0	73.500	168.0
16	0.0	1.268	9.0	0.0	0.262	2.549	38.0	81.708	172.0
17	0.0	2.142	12.0	0.0	0.746	3.967	60.0	176.402	364.0
18	0.0	2.420	21.0	0.0	0.820	7.367	75.0	179.514	447.0
19	0.0	2.298	15.0	0.0	0.379	2.462	32.0	70.230	176.0

Table 6.6: Testing results for all MDPs for the second agent at minimax tree cut off depth 4

Chapter 7

Discussion, Conclusions and Future Work

7.1 Discussion

Adversarial Tetris is a game which witholds the complexity of Tetris and combines it with adversity. In this thesis we proposed an agent that will use a tree search algorithm in order to form the game tree and enable it to confront the adversary of the game. However, because of the very large action and state space the tree cannot be expanded to a great depth. In this thesis we did not expand the tree beyond the cut off depth of 4. The great state and action space has also a great effect to the evaluation function. We propose a linear architecture for the approximation of the State-Value Function as the computation of the State Value function would have been impractical. The linear architecture that we issue are two sets of basis functions that we have been experimented on. The first set of basis functions involves the features that we have provided to the agent in order to evaluate the utility of a board and the second combines these features with the currently falling piece. The policy of the agent derives from the Value Function approximation at every decision step, as the approximation is improving, the policy of the agent also improves, until a point where the approximation has converged and the quality of the policy does not change any more. The learning algorithm in order for this improvement in policy to be achieved is a variation of the Least-Squares Temporal Difference Learning that involves one parameter

μ which alters the behavior of the algorithm according to its numerical value. If μ is 0 the Incremental Least-Squares Temporal Difference is exactly the same with the original form of LSTD. In our case μ is non-zero as we use exponential windowing in order to weight the older samples in respect to the newer samples.

7.2 Future Work

The work that has been done in this thesis can have many expansions and experimentations for the future. A lot of experimentation can be done as far as the basis functions sets are concerned. It can be explored more how to achieve a better approximation of the State Value-Function. What is not included in this thesis because we could not overcome some technical difficulties is a testing run with the environment of the competition, however is one of our first future goals.

7.3 Conclusions

The results of this work have not gone far beyond the results of the Reinforcement Learning Competition as we can estimate at the time, however we can say that this work has proposed an architecture of an agent that has a good learning performance and because the opponent of the MDP that it was tested had one of the harder opponents it can play equally well and even better to othe MDPs regardless of the board dimensions. The sets of basis functions although that have a very small number enable the agent to form a good approximation of the State Value Function and the learning algorithm is able to converge in a relatively small number of iterations.

References

[1] Stuart Russel and Peter Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2 ed., 2003. 8, 11, 15

[2] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning:An Introduction*. The MIT Press, 1 ed., 1998. 14, 15, 16, 17

[3] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *JOURNAL OF ARTIFICIAL INTELLIGENCE RESEARCH*, vol. 4, pp. 237–285, 1996. 15

[4] Michail G. Lagoudakis and Ronald Parr, "Least-squares policy iteration," *Journal of Machine Learning Research*, vol. 4, pp. 1107–1149, 2003. 16

[5] S. J. Bradtke, A. G. Barto, and P. Kaelbling, "Linear least-squares algorithms for temporal difference learning," in *Machine Learning*, pp. 22–33, 1996. 18

[6] J. A. Boyan, "Least-squares temporal difference learning," in *In Proceedings of the Sixteenth International Conference on Machine Learning*, pp. 49–56, Morgan Kaufmann, 1999. 18

[7] E. D. Demaine, S. Hohenberger, and D. Liben-nowell, "Tetris is hard, even to approximate," tech. rep., 2003. 22

[8] J. N. Tsitsiklis and B. V. Roy, "Feature-based methods for large scale dynamic programming," in *Machine Learning*, pp. 59–94, 1994. 22, 23, 26

[9] "RL Competition 2009." 23, 38, 40

[10] D.P. Bertsekas and J.N. Tsitsiklis, *Neuro-dynamic programming* . Athena Scientific, 1996. 26

[11] D. P. Bertsekas and S. Ioffe, “Temporal differences-based policy iteration and,” in *Applications in Neuro-Dynamic Programming, Lab. for Info. and Decision Systems Report LIDS-P-2349, MIT*, 1996. 26, 27

[12] S. Kakade, “A natural policy gradient.” 26

[13] J. Ramon and K. Driessens, “On the numeric stability of gaussian processes regression for relational reinforcement learning,” in *In ICML-2004 Workshop on Relational Reinforcement Learning*, pp. 10–14, 2004. 26

[14] V. F. Farias and B. Van Roy, *Probabilistic and Randomized Methods for Design Under Uncertainty, chapter Tetris: A Study of Randomized Constraint Sampling*. Springer-Verlag UK, 2006. 27

[15] I. Szita and A. Lorincz, “Learning tetris using the noisy cross-entropy method,” *Neural Computation*, vol. 18, p. 2006. 27

[16] Chistofe Thiery, “Controle optimal stochastique et le jeu de Tetris,” Master’s thesis, Universite Henri Poincare – Nancy I, Juin 2007. 27

[17] John Asmuth, Monica Babes, XinyiCui, SergiuGoschin, BaiyangLiu, Chris Mansley, Paul Ringstad, Kevin Sanik, Brian Schubert, Daniel Shields, RavneetSingh, TingtingSun, FengmingWang, Ari Weinstein, John Wilder, Michael Wunder, Yan Xiong, SaeHoonYi, “The RL Competition as a Class Project.” ICML Workshop 2009, 2009. 27

[18] ww.rl-community.org, “RL-Glue.” http://glue.rl-community.org/. 37

[19] www.rl-community.org, “Task Specification Language.” url:http://glue.rl-community.org/Home/rl-glue/task-spec-language. 38

[20] Eclipse Foundation, “Eclipse IDE.” `http://www.eclipse.org/`. 38

[21] Sun Microsystems, “Java Sun Platform JRE 6.” url:http://java.sun.com/javase/downloads/index.jsp. 38

[22] Joe Hicklin Cleve Moler Peter Webb , from Mathworks and Ronald F. Boisvert Bruce Miller Roldan Pozo Karin Remington, from NIST, “Jama: A Java Matrix Package.” url:http://math.nist.gov/javanumerics/jama/. 39