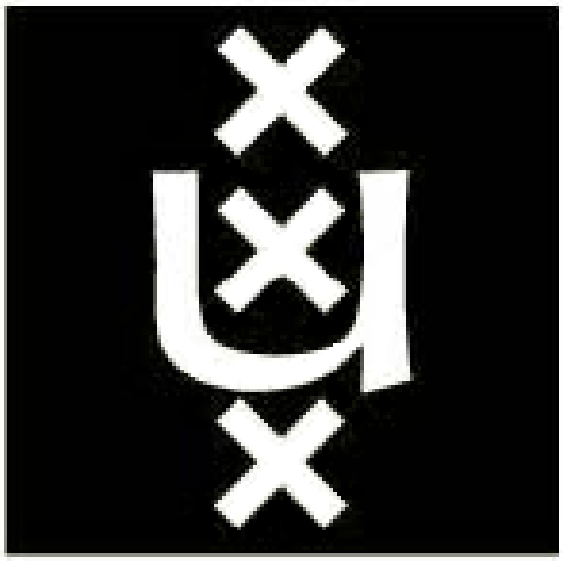
Cross-Entropy Method for Reinforcement Learning

Steijn Kistemaker
Heyendaalseweg 127 A1-01
6525 AJ Nijmegen
skistema@science.uva.nl

Thesis for Bachelor Artificial Intelligence
University of Amsterdam
Plantage Muidergracht 24
1018 TV, Amsterdam



Supervised by:
Frans Oliehoek
Intelligent Autonomous Systems group
Informatics Institute
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam
F.A.Oliehoek@uva.nl

Shimon Whiteson
Intelligent Autonomous Systems group
Informatics Institute
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam
shimon@whiteson.org

June 27, 2008

Abstract

Reinforcement Learning methods have been succesfully applied to various optimalization problems. Scaling this up to real world sized problems has however been more of a problem. In this research we apply Reinforcement Learning to the game of Tetris which has a very large state space. We not only try to learn policies for Standard Tetris but try to learn parameterized policies for Generalized Tetris, which varies from game to game in field size and the chance of a single block occuring. In comparison to a non-parameterized policy we find that a parameterized policy is able to outperform the non-parameterized policy. The increased complexity of learning such a policy using the Cross-Entropy method however reaches a limit at which the policy is too complex to learn and performance drops below the non-parameterized policy.

Contents

1 Introduction

1.1 Motivation

In everyday life we are constantly performing all kinds of tasks, from really simple to really complex tasks like driving a car. When driving a car we observe the world and respond (act) to it. However defining how we would react in all situations would be a hard (impossible?) task. Tasks like these are thus not suited for solving by a hand-coded approach. *Reinforcement learning* is a technique to deal with these kinds of problems where optimal actions have to be taken to maximize

a long-term reward and the solution of the problem is not clearly defined. The tasks are often formulated as a *Markov Decision Process* where decision making is modeled by mapping *states* to *actions*. To stimulate research in this field an annual *Reinforcement Learning competition* [1] is held to compare RL techniques in different domains. This year one of those domains is Tetris, which belongs to the class of tasks with a large state-space and where choosing the optimal action is hard. I have done my research in this domain because it lends itself for benchmarking RL techniques and, because of the large state space, it scales up to a real-world sized problem.

1.2 Standard Tetris

Tetris is a popular video game programmed by Alexey Pajitnov in 1985. Since its introduction the game spread throughout the world and gained much popularity. It has been implemented on about every game system every since and has spawned many variations to it.

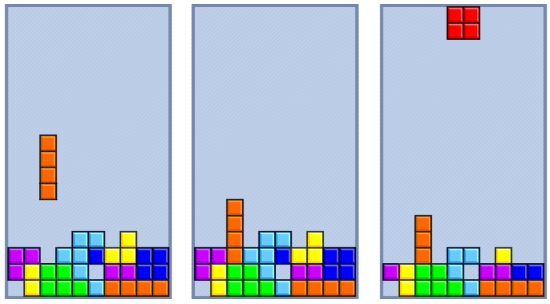


Figure 1: From left to right: A block falling, the construction of a filled line, the removal of the line and introduction of a new block

Tetris is a falling blocks game where blocks of different shapes called *tetrominoes* have to be aligned on the playing field to create horizontal lines without gaps. A standard field has width 10 and height 20 and uses 7 tetrominoes (see Fig. 2) which occur with equal probability. We will use the term 'block' for tetromino from now on. The standard Tetris also shows the next block, which gives the player the advantage of taking this into account when placing the current block. The goal of the game is to remove as many lines as possible by moving the blocks sideways and by rotating them 90 degree units to place them in the desired position[8]. While doing these rotations and translations the block drops down at a discrete fixed rate, so all actions must be done before the block hits an obstruction. When a block is placed so a line is filled completely, the line is removed and the complete contents of the field above it drops one line down. The game ends when the filled cells in the playing field reach up to the top of the field so that no new block can enter.

Even though the concept and rules of Tetris are simple, playing a good game of Tetris requires a lot of practice and skill. Extensive research has been done on the

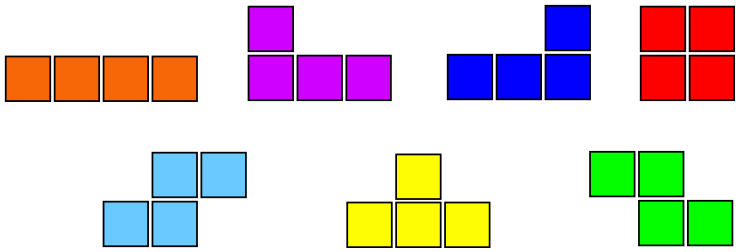


Figure 2: From left to right, top to bottom: I-block, J-block (or left gun), L-block (or right gun), O-block (or square), S-block (or right snake), T-block, Z-block (or left snake)

game of Tetris in a mathematical sense [11, 5, 7] and in AI research as well [8, 16]. Burgiel shows that a sequence of alternating S and Z blocks can not be survived [5]. Given an infinite sequence of blocks every possible sequence will occur. This proves that any game must ultimately end, so no perfect player could go on forever. Also Demaine et al. have shown that finding the optimal placement of blocks is NP-hard even if the complete sequence of blocks is known in advance [7]. This makes it a problem fit for Reinforcement Learning techniques, and learning techniques in general.

1.3 Generalized Tetris in short

In our research we will however be foccusing on a new variation on Standard Tetris, called *generalized Tetris* as defined by the *Reinforcement Learning Competition* [1]. Quite some work has already been done on solving the problem of Standard Tetris in AI, and a general approach has been developed [4, 16]. Generalized Tetris however poses somes new problems because it introduces a parameterized version of the game. Here every seperate game is parameterized by the width and height of the field, which is directly observable, and the probability of each block occuring, which has to be infered. Generalized Tetris thus poses new problems because a generalized or parameterized tactic has to be learned and the block probabilities have to be inferred, which introduces insecurity about which state the player is in. We will try to tackle the problem of state insecurity and develop ways to learn a parameterized tactic.

1.4 Overview of thesis

In the rest of this thesis we will introduce the theoretical background in section 2, which introduces the concepts of Markov Decision Processes, Reinforcement Learning and the Cross-Entropy method. In section 3 we will explain the general approach for learning Tetris policies and discuss the previous work on which we are building on. In section 4 we will explain how generalized Tetris differs from

Standard Tetris and how we plan to solve the new problems it poses. In section 5 we will introduce our experiments, the measures to validate these experiments and the results themselves. In section 6 we will state our general findings and conclude about the effectiveness of the approach taken, ending with section 7, discussing the validity of our results and introduce some interesting unanswered questions on which further research could be done.

2 Background

2.1 Markov Decision Process

The Markov decision process (MDP) is a mathematical framework for modeling tasks that have the *Markov property*. This *Markov property* defines a memory-less system where the current state summarizes all past experiences in a compact way so that all relevant information for decision making is retained [15]. The transition probability of going from a certain state to the next state is thus independent of the previous states.

The Markov decision process can be defined by states $s \in S$, actions $a \in A$ and the state transition function, given by (1), which gives the probability of going from *s* to *s'* with action *a*.

$$P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a) \tag{1}$$

For every state transition at time *t* a reward r_t is given where the expected value of the next reward is given by (2).

$$R_a(s, s') = E(r_{t+1} | s_t = s, a_t = a, s_{t+1} = s') \tag{2}$$

The state *s* is the collection of sensations recieved by the decision maker. This is not restricted by immediate (sensoric) sensations but can also be some internal sensation retained by the decision maker. We can now define the Markov property in a more precise way. The decision process is said to have to *Markov property* if it satisfies (3), or (4) when rewards are taken into account [15].

$$Pr(s_{t+1} = s' | s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = Pr(s_{t+1} = s' | s_t, a_t) \tag{3}$$

$$\begin{aligned} Pr(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \ldots, s_0, a_0, r_0) \\ = Pr(s_{t+1} = s', r_{t+1} = r | s_t, a_t) \end{aligned} \tag{4}$$

2.2 Reinforcement Learning

Reinforcement Learning is a learning technique to learn optimal actions to recieve a maximum amount of (possibly) delayed reward by interacting with the environment [15]. The agent learns by acting in its environment, recieving a positive or

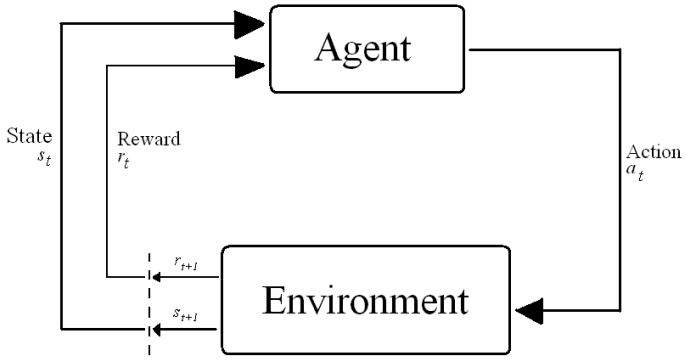


Figure 3: The agent-environment interaction in Reinforcement Learning (from [15])

negative reward, and adjusting its preference for the taken action. The tasks in Reinforcement Learning are often modelled as MDPs in which optimal actions over the state space *S* have to be found to optimize the total return over the process.

In Fig. 3 we can see the interaction between the agent and the environment. At timestep *t* the agent is in state *s* and takes an action $a \in A(s)$, where *A(s)* is the set of possible actions in *s*. It then transitions to state s_{t+1} and recieves a reward $r_{t+1} \in \Re$. Based on this interaction the agent must learn the optimal policy $\pi : S \rightarrow A$ to optimize the total return $R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T$ for episodic task (with an endstate at timestep T), or $R_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1}$, where $0 \leq \gamma \leq 1$ for a continuous task.

In Reinforcement Learning, learning a policy is based on estimating *value functions* which expresses the value of being in a certain state. When the model of the environment is known, one can predict to what state each action will lead, and choose the action that leads to the state with the highest value. When such model is unavailable the agent can either try to learn the model or learn values for *state-action pairs*. The value for a certain state following policy π is given by the Bellman Equation in (5), where $\pi(s, a)$ is the chance of taking action *a* in *s*.

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\{R_t | s_t = s\} \\ &= E_{\pi}\Big\{ \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} \Big| s_t = s \Big\} \\ &= \sum_{a} \pi(s,a) \sum_{s'} P_a(s,s')[R_a(s,s') + \gamma V^{\pi}(s')] \end{aligned} \tag{5}$$

The task of the agent is to learn the *optimal policy* π^* by learning the optimal state-value function $V^*(s) = \max_{\pi} V^{\pi}(s)$ for all $s \in S$.

In reinforcement learning several techniques have been developed to learn the

optimal value function or approximate it. Dynamic programming techniques such as value iteration and policy iteration can solve the problem exactly, but require a complete model of the environment. *Monte Carlo methods* are used to learn from on-line from experience and do not require a model of the environment. A disadvantage of this technique however is that it learns only after finishing an episode and can thus not be applied to continuing tasks. A third method called *temporal-difference learning* bridges the gap between dynamic programming and Monte Carlo methods by being able to learn from experience and learning during the episode by bootstrapping (updating estimates based on other estimates).

The value function *V(s)* learned is classicaly represented by a table with the values stored for each state. Because real-world problems can have a state space too large to represent by a table several techniques have been developed to approximate the value for each state. This can be done by using a linear value function with weights ω and features ϕ defining the value of a state as in (6) for *n* features. These weights can then be learned by gradient-descent methods or parameter optimization techniques such as *genetic algorithms* [10] and the *Cross-Entropy Method* [6].

$$V(s) = \sum_{k=1}^{n} \omega_k \cdot \phi_k(s) \tag{6}$$

2.3 Cross-Entropy Method

The *Cross-Entropy (CE) method* is a technique frequently used for rare event simulation and optimization [6]. The CE method provides a simple way to deal with combinatorial optimization problems in an efficient way. The basic steps of CE consist of generating random data samples and maintaining a distribution of good samples according to some scoring mechanism to generate new samples from. CE can be applied to reinforcement learning by learning (optimizing) a value function [6, 16]. One of the possibilities is to learn the weights for a linear value function as in (6). The task for the CE method is to optimize the weights of a value function as in (7), which determines the actions of the agent. The CE method then optimizes $S(\vec{\Omega})$, which is a real valued scoring function of the performance of the agent using weight vector $\vec{\Omega} = |\omega_1, \ldots, \omega_n|$ [6].

$$\vec{\Omega}^* = \arg\max_{\vec{\Omega}} S(\vec{\Omega}) \tag{7}$$

The CE method starts with an initial distribution $\chi_0 \in \Xi$, where Ξ is the probability density function defining parametric family of χ. The goal is to find a distribution χ^* with an optimal score of α^*. Since the probability of getting a high-valued sample, with α near α^*, is very small we need a way of improving our distribution χ. When we now find a distribution χ_1 in the high valued samples such that $\alpha_1 \geq \alpha_0$ we can replace our initial distribution χ_0 by χ_1 and resample from here.

In this research a gaussian distribution is used such that $\chi_t \in N(\mu_t, \sigma_t^2)$. We draw a set of N sample vectors $\vec{\Omega}_1, \ldots, \vec{\Omega}_N$ with scores $S(\vec{\Omega}_1), \ldots, S(\vec{\Omega}_N)$ of which the best $[\rho \cdot N] = \beta$ samples are selected, where $0 < \rho < 1$. From this elite set of β samples the new $\vec{\mu}_{t+1}$ and $\vec{\sigma}_{t+1}^2$ are given by (8) and (9), denoting the indices of these elite samples by i.

$$\vec{\mu}_{t+1} = \frac{\sum_{i=1}^{\beta} \vec{\Omega}_i}{\beta} \tag{8}$$

$$\vec{\sigma}_{t+1}^2 = \frac{\sum_{i=1}^{\beta} (\vec{\omega}_i - \vec{\mu}_{t+1})^T (\vec{\Omega}_i - \vec{\mu}_{t+1})}{\beta} \tag{9}$$

3 Feature-based approach to Tetris

3.1 Afterstate evaluation

A lot of research in reinforcement learning applied to Tetris has been done after Tetris was first formalized as an MDP by [17]. Because of the large state space, approximately $0.96 \cdot 2^{199}$ [8], standard reinforcement learning techniques (such as tabular TD) cannot be applied here. Therefore feature based approaches have been taken to develop skilled Tetris AI. All these approaches are based on evaluating so called *afterstates* and choosing the right action to reach this state.

In the Tetris environment the state is defined by a discrete valued vector containing:

1. bit map: binary representation of the current board
2. bit vector: indentifying the falling block

The action space is defined as two discrete valued variables determining the final translation and final rotation.[1] When the block is placed in its final position, the block is dropped down. With all the dynamics known we can predict the state the system will be in after we have finished this action. Because we can enumerate all possible actions, we can enumerate all possible after states (the so called look ahead). In combination with an afterstate evaluation function we can choose the 'best' afterstate, and thus the best action. The pseudocode for this is given below.

[1] This is different from the game Tetris where the block has to be manipulated while it is falling down.

Main Algorithm($currentfield$)

```
1  for each step
2       do
3          if new block entered
4             actionTuple ← FINDBESTAFTERSTATE(currentfield,newBlock)
5  return actionTuple
```

findBestAfterState($currentfield$,$newBlock$)

```
 1  maxscore ← −∞
 2  for all translations t
 3       do
 4          for all rotations r
 5               do
 6                  resultingfield ← AFTERSTATE(newBlock,currentfield,t,r)
 7                  score ← SCOREFIELD(resultingfield)
 8                  if score > maxscore
 9                     bestAction ← {t,r}
10                     maxscore ← score
11  return bestAction
```

3.2 Representing the value function

3.2.1 Features

Because the state space of Tetris is so large, a higher level description of the field is needed. Idealy one would like a description that incorporates all important aspects of the state, but has a low dimensionality. Often a trade off between these two must be made.

In this research we have adopted the feature set as described by [4] which is a combination of their own features, three features used in Pierre Dellacherie's (PD) hand coded algorithm and two features used by Colin Fahey (CF) [8] in his Tetris AI. We have however removed the "Blocks" feature used by Colin Fahey because it is obsolete in combination with the "Removed Lines" feature.[2] We will give an overview here of the used features as described in [4]. In Fig. 4 some of the features are represented graphicaly. The following text is an excerpt from [4].

1. *Pile Height*: The row (height) of the highest occupied cell in the board.

[2]The "Blocks" feature is the count of the number of filled cells in the playing field. However since every block adds 4 filled cells to the field, when no lines are removed this feature will have the same value for all compared fields at one time. The difference in comparison between different fields for this feature can thus only be caused by the removal of lines. The "Removed Lines" feature however already encodes for this more directly, so the "Blocks" feature is rendered obsolete.

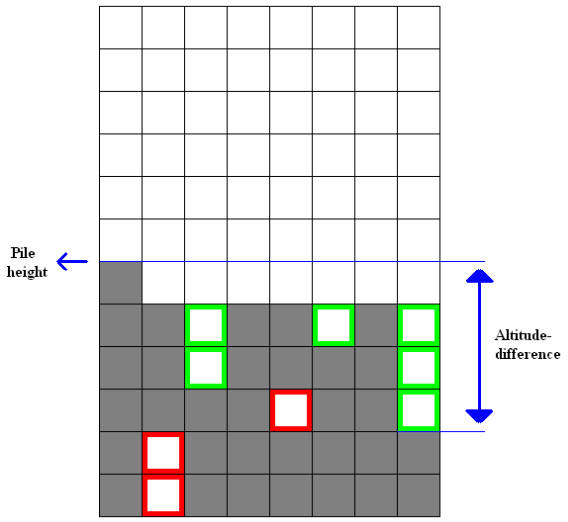


Figure 4: A graphical representation of some of the features used. In red the "Holes"; the two holes above eachother would be counted as one by "Connected Holes". In green the "Wells" on the board. The rightmost well has the "Maximum Well Depth" of 3. Also the "Pile Height" and "Altitude Difference" are depicted here.

2. *Holes*: The number of all unoccupied cells that have at least one occupied above them.
3. *Connected Holes*: Same as *Holes* above, however vertically connected unoccupied cells only count as one hole.
4. *Removed Lines*: The number of lines that were cleared in the last step to get to the current board.
5. *Altitude Difference*: The difference between the highest occupied and lowest free cell that are directly reachable from the top.
6. *Maximum Well Depth*: The depth of the deepest well (with a width of one) on the board.
7. *Sum of all Wells (CF)*: Sum of all wells on the board.
8. *Weighted Blocks (CF)*: Sum of filled cells but cells in row n count n-times as much as blocks in row 1 (counting from bottom to top).
9. *Landing Height (PD)*: The height at which the last tetramino has been placed.
10. *Row Transitions (PD)*: Sum of all horizontal occupied/unoccupied-transitions on the board. The outside to the left and right counts as occupied.
11. *Column Transitions (PD)*: As Row Transitions above, but counts vertical transitions. The outside below the game-board is considered occupied.

3.2.2 "Naive" Linear Architecture

To evaluate a Tetris field multiple types of value functions can be used. For instance, an exponential value function can be used to express the fact that an increase in pile height at the top of the field is much worse than an increase in pile height at the bottom of the field [4]. Simple linear functions however have been proven to lead to good results as well [16]. One of the value functions we will use can be defined as in (10), where $\phi_i(s)$ is the value of feature ϕ_i given s. [3]

$$V(s) = \sum_{i=1}^{|\phi|} \omega_i \cdot \phi_i(s) + \omega_0 \tag{10}$$

This value function has been proven to work well when trained on one MDP setting. We will, however, apply it to a set of MDPs with different characteristics. The optimal weights for every MDP seperately could be different, so we want to learn a general weight set that performs well on all MDPs in the testset.

3.3 Learning the value function

Learning a value function as described in (10) can be done in many ways, and several Reinforcement Learning techniques have been applied to learn these value functions for the game of Tetris. Tsitsiklis et al. used a very simple feature representation in combination with table-based value iteration in which they showed that it is possible to tackle the problem of dimensionality by using a feature state description [17]. As we can see however in table 1 their approach does not result in very good play.

Several other approaches have been taken such as policy iteration [2], genetic algorithms [4] and the cross-entropy method [16] to optimize the Tetris policy. They however tried to learn the weights for a feature set, using to linear and exponential value functions [16]. No clear optimal form for the value function however has been found and conflicting results have been obtained. It is argued that the optimal form of the value function is dependent on the field size [4].

As we can see in table 1 a high variation in scores is obtained. However most of the early results were directed on showing the possibilities of different learning techniques applied to Tetris. Only later in [3, 4] much work has been done on feature engineering, resulting in a significant increase in performance. Surprisingly however, [16] have obtained good performance - in comparison with the best handcoded approach (Dellacherie [8]) - with a somewhat simplistic feature set and a linear value function. They expect that using better features and possibly an exponential value function will result in a significant increase in performance. At the moment only the results of [16, 3, 4] compare competitively with the best hand coded approach.

[3] Because we are comparing the values *V(s)* for different states we will not use ω_0 in our implementation.

Technique	Average removed lines	Reference
Value iteration	32	[17]
$\lambda-$Policy iteration	3,183	[2]
Natural policy gradient	$\approx 6,800$	[12]
Least squares policy iteration	1,000 - 3,000	[13]
Handcoded by Dellacherie	$631,167$	[8]
Relational RL + Kernel-based regression	50	[14]
Genetic Algorithm	$586,130$	[3]
Bootstrapped dynamic programming	4,274	[9]
Cross-Entropy with decreasing noise	$348,895$	[16]

Table 1: Overview of learning techniques applied to Standard Tetris and their results in chronological order.

4 Generalized Tetris

4.1 Generalized Tetris in the Reinforcement Learning Competition

The competition domain given by the *Reinforcement Learning Competition* is based on the Van Roy (1995) specification of the Tetris problem. The competition domain however differs from it, and from the standard Tetris, in a few significant ways. In van Roy's specification the agent chooses a translation and rotation as to specify its final position. In the competition version however, the agent chooses to rotate or translate the block one space as it drops down. This gives more control over the block (like in the standard Tetris), but has a limiting effect on what positions can be reached from the top position.[4] Although it is possible to control the block at every timestep we decided not to use this feature and act like a final translation and position at the top can be chosen when a new block enters the field, and is then followed by a drop down. Although this reduces some of the possibilities it makes the action space a lot simpler. We are using an after state evaluation as described earlier. The highest scoring after state is selected, and the sub-actions for translating, rotating and dropping to reach that after state are planned.

Each MDP in the competition domain is defined by 9 parameters, of which 2 define the size of the playing field (directly observeable in each state) and 7 define the chances for each of the 7 different blocks occuring (we will call this the block probability). These block probabilities are not known in advance and are not directly observable. The field size and block probabilities can also vary from MDP to MDP. In Standard Tetris the block probabilities are uniformly distributed, but in the competition domain they are drawn from a non-uniform distribution.

[4] As opposed to standard Tetris, where multiple actions can be done per timestep where the block drops one line down.

In the competition a limited test-set of MDPs is drawn from a predetermined distribution on which can be trained. Evaluation takes place on the performance of the agent on a seperate proving set. A limited number of steps is given to obtain a maximum score.[5] Whenever the field is filled in such a way that no new block can enter the field is cleared and the game continues with a new block until the step-limit is reached.

For our research we planned to learn a generalized value function over a set of MDPs. To do this we are using the naive linear value function as described above for comparison, and have developed two value functions parameterized by the block probabilities. In our research we use a test set with equal field widths an do not try generalize over the field dimensions.

4.2 Learning the block distribution

Because the block distribution is not directly observable, but does give crucial information for decision making this block distribution has to be estimated in some way. Luckily this block distribution can be easily approximated by keeping a count of the number of occurences of each block. The chance of a single block occuring can then be estimated as $P(block) = \frac{\#occurences\_block}{\#total\_occurences}$. This will however be very susceptible to strong variation early on in the game and could lead to bad decisions. A possible solution to this is to use a confidence term which expresses the confidence in the correctness of the current block distribution. This confidence would be 0 at the start of the game and increase to 1 as the game continues. We have chosen to assume a uniform block distribution at first and use a linear increase in confidence which becomes 1 after 1000 total block occurences. The approximated chance of a single block occuring is then expressed as $P(block) = conf \cdot \frac{\#occurences\_block}{\#total\_occurences} + (1 - conf) \cdot \frac{1}{7}$.

To incorporate this knowledge of the block distribution in our process of decision making we have developed two techniques, being parameterized value functions (as described in subsequent sections) and parameterized look ahead. For the parameterized look ahead we increase the look ahead to a search depth of 2 (two ply). We have however no knowledge of the next piece, so we will have to enumerate all possible afterstates for all possible next pieces. We incorporate the knowledge of the block distribution by weighting the look ahead scores for each block according to its probability of occuring. Pseudocode for this algoritm is given below.[6]

[5] Every single action counts as a step.

[6] The function-call *findBestAfterState* on line 10 returns the *maxscore* in this case, instead of the *best action* as defined in 3.1.

findBestAfterStateTwoPly($currentfield$,$newBlock$)

```
1  maxscore ← −∞
2  for all translations t
3      do
4          for all rotations r
5              do
6                  resultingfield ← AFTERSTATE(newBlock,currentfield,t,r)
7                  score ← 0
8                  for all blocks b
9                      do
10                         scoreThisBlock ← FINDBESTAFTERSTATE(resultingfield,b)
11                         score ← score + P(b) * scoreThisBlock
12                 if score > maxscore
13                    bestAction ← {t,r}
14                    maxscore ← score
15 return bestAction
```

4.3 Generalized Value functions

4.3.1 Generalized Linear Architecture

Because the linear value function in (10) does not take the parameters of the different MDPs into account it is bound to have little succes generalizing when the different MDPs are not very much alike. Therefore we have developed a value function that is dependent on the block probability parameters. However since they sum up to 1 we can define each MDP by 6 parameters. We want to model the pairwise dependencies of each feature / parameter pair. This way we can model the influence of each parameter on each feature in a seperated manner. The value function is given in (11) where π_j denotes the value for the j'th parameters. Because we express each MDP by 6 ($= |\pi|$) parameters we need a set of offset weights defined by ω_l in the last term of the value function.

$$V(s) = \sum_{i=1}^{|\phi|} \sum_{j=1}^{|\pi|} \omega_{i,j} \cdot \pi_j \cdot \phi_i(s) + \sum_{l=1}^{|\phi|} \omega_l \cdot \phi_l(s) \tag{11}$$

For this value function however we assume a linear dependency between the feature and the parameter, which might not be the case. Although this value function is much more expressive than the naive linear function, it will fail to express all pairwise depencies in a correct manner when they are not linear.

4.3.2 Generalized Tilecoded Architecture

We have developed a third value function to overcome the problem of linear dependencies, but still keep the expressiveness of modeling the influence of each

parameter on each feature by pairwise dependencies. We have therefor developed a value function that performs tilecoding in the parameter space. In this function every feature / parameter pair is represented by a vector of weights and a binary vector selecting one of these weights. Because of the limited variation in the MDP test-set we have decided to use 3 tiles in the parameter space subdividing it into the regions of $\pi_j < 0.10, \quad 0.10 \leq \pi_j \leq 0.18, \quad$ and $\pi_j > 0.18$. When a larger test-set with more variation is available a more fine-grained tiling can be used. The value function is defined in (12) where the first vector evaluates the truth of each expression resulting in 0 or 1.

$$V(S) = \sum_{i=1}^{|\phi|} \sum_{j=1}^{|\pi|} \begin{bmatrix} \pi_j < 0.10 \\ 0.10 \leq \pi_j \leq 0.18 \\ \pi_j > 0.18 \end{bmatrix}^T \cdot \begin{bmatrix} \omega_{i,j,1} \\ \omega_{i,j,2} \\ \omega_{i,j,3} \end{bmatrix} \cdot \phi_i(s) + \sum_{l=1}^{|\phi|} \omega_l \cdot \phi_l(s) \quad (12)$$

Using this value function the influence of each parameter on each feature, while removing the linear dependency because a weight for each range of a parameter value in pair with a feature can be learned. However increasing the complexity of the value function will make it harder to find the optimal weights, which could have a negative effect on its learning abilities.

5 Experiments and results

To research the effectiveness of the different value functions we have set up a number of experiments to validate the different approaches. Training times for Tetris, however, are very long due to the fact that the training time scales linearly with the performance of an agent, and the fact that a large number of games need to be played to verify the performance of an agent with high confidence. Because time in this project was limited we needed a way to lower the training times. We also wanted to stay as close to the performance criteria as set by the Reinforcement Learning Competition [1]. In the competition every test MDP is played for 5 million steps. Whenever the blocks in the field reach the top, the field is cleared and the game continues until the step limit is reached. We have however decided to end the agent playing the MDP when the blocks in the field reach the top (like a normal game ending in standard Tetris). We did this to ensure fast termination of bad playing agents. To also ensure practical training times for good playing agents we have also limited the runs per MDP by 500.000 steps. The score of an agent per MDP was calculated as $\frac{\#lines\_removed}{\#steps\_used} \cdot (5 \cdot 10^6)$ (We will call this the RLC metric). Every MDP was played twice by an agent, and the score per MDP was the average of the two runs. The total score of the agent was the sum of the scores over all played MDPs. We ran the Cross-Entropy method with 100 agents and $\rho = 0.1$ resulting in a population of 10 elite agents. We initialized the first generation of agents from a gaussian distribution as described earlier, with $\mu = 0$ and $\sigma^2 = 100$. As a test set we used six MDPs defined by the parameters as described in table 2.

Fieldsize	P(I)	P(O)	P(T)	P(Z)	P(S)	P(J)	P(L)
7 x 17	0.2365	0.2809	0.1299	0.0084	0.3079	0.0221	0.0143
7 x 18	0.2102	0.2022	0.0077	0.2159	0.1489	0.0596	0.1555
7 x 19	0.1420	0.1755	0.1270	0.1701	0.0580	0.1765	0.1509
7 x 20	0.2314	0.1548	0.1007	0.1563	0.0064	0.1209	0.2295
7 x 21	0.2492	0.0902	0.0897	0.1888	0.1370	0.1939	0.0512
7 x 23	0.1993	0.1408	0.0422	0.2702	0.0023	0.2115	0.1337

Table 2: Overview of the six training MDPs and their parameters. All block probabilities are the empericaly determined relative occurences after 10.000 blocks, rounded to 4 decimals

5.1 One ply vs Two ply

Besides testing the possibilities of a generalized value function for a parameterized MDP set we wanted to research the possibilities of a one ply search as described in 3.1 versus a two ply search as described in 4.2, where all possible placements of all possible next blocks are also calculated, and their scores are weighted by the probability of each block occuring.

We trained all six MDPs seperately with the Naive Lineair value function as in (10), which is meant for training on a single MDP, and compared their relative performance. In training the score of an agent was its average number of lines removed over 5 runs on the same MDP. Each MDP was trained on 3 times, resulting in 3 different agents (weightsets) for every MDP. After training we evaluated each learned weightset by running it 10 times on the MDP it was trained for. The scores per MDP per agent is the average over these 10 performance runs. We introduce two measures of variance, of which one is the *performance variance* σ_p, which is the standard deviation of one weightset over the 10 performance runs. The second is the *training variance* σ_t which is the standard deviation of the 3 per agent scores.

The results of this experiment are shown in Fig. 5, the performance variance and average training variance are reported in table 3. As we can see in Fig. 5 the two ply look ahead gives a significant benefit over the one ply look ahead. An average improvement of factor 4.1 over all six MDPs was found. This however goes at the cost of $\#rotations \cdot \#translation \cdot \#blocks \approx 150$ more field evaluations, and thus an equal increase in training time. We have therefore chosen not to use this function in further experiments. Seperate experiments have shown that the optimal one ply action is also the optimal two ply action in 85% of the cases. The optimal two ply action was found to be in the top 25% of the one ply actions in 97% of the cases. With this information a heuristic can be developed which reduces the evaluation time for two ply search by $\frac{1}{4}$'th and make suboptimal decisions in only 3% of the cases.

In table 3 we can see an increase in variance between one ply and two ply, both in the training variance as well as in the performance variance. Although the two ply look ahead results in better scores, it does not result in more consistent play,

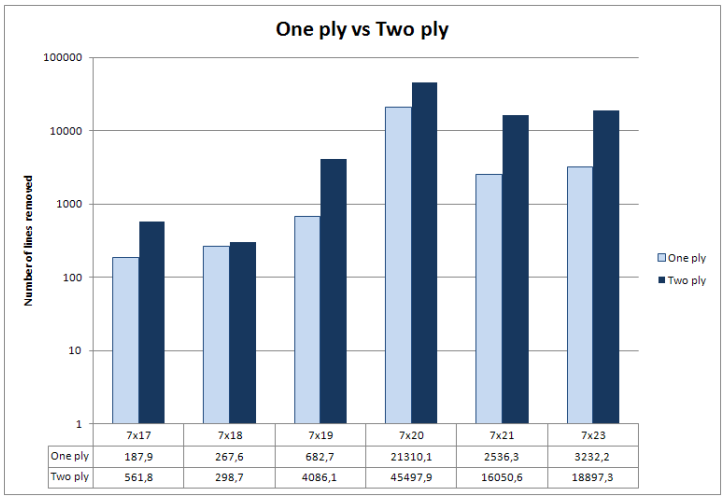


Figure 5: Results for one ply vs two ply look ahead. Shown are the averaged scores per MDP over 10 performance runs with 3 weightsets.

meaning that it will score much better on average, but it still able to play games with really low scores. Because the range of possible scores for the two ply is much larger, this results in a larger performance variance for the two ply look ahead. The fact that the training variance also increases indicates that 10 runs per MDP still might not be enough to get a good (consistent) estimate on the capabilities of the agent.

	One ply		**Two ply**	
	σ_p	σ_t	σ_p	σ_t
7 x 17	19.2415	9.6664	113.0699	32.8602
7 x 18	71.5738	11.0708	67.4893	32.6390
7 x 19	264.8725	49.0567	959.2267	607.4347
7 x 20	5194.5877	1239.8945	5519.3921	4590.7046
7 x 21	853.1617	209.8297	2736.9810	1303.2935
7 x 23	664.4103	392.9186	3633.9439	1814.2882

Table 3: The variance measures for one ply vs two ply look ahead.

5.2 Three value functions

To train the three different value functions ((10), (11), (12)) we have used the same MDP testset as described above. The stop condition for learning was based on the sum of the variance in all weights. The threshold value was 2, 10, 20 respectively for the three different value functions. If the sum of the variances dropped below this value learning was terminated. We increased this threshold value for the more complex value functions because they use more weights and take longer to sta-

bilize. By increasing the threshold we can perform training runs within practical time.

All three value functions had 10 training runs, resulting in a total of 30 learned weightsets. After learning we performed performance test runs where the complete testset of MDPs was run, and every MDP was played 10 times by each agent. We wanted to compare these results with eachother and with the specialized training runs. To compare the learned weightsets with the specialized one ply training runs we have measured the average performance on a single MDP as a percentage of the scores of the specialized training runs (as described in the previous section). To compare the different value functions with eachother we have used the RLC Metric. The scores reported are the sum over all MDPs, where the score per MDP was the average score over the 10 performance runs.

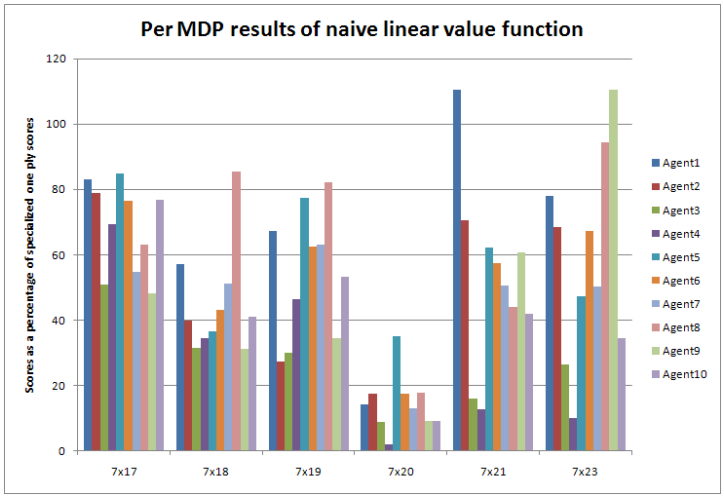


Figure 6: Per MDP results of the 10 naive linear learned weightsets as a percentage of the specialized one ply scores.

In Fig. 6 we can see a typical distribution of per MDP scores as a percentage of the specialized one ply scores. All agents seem to make the same trade off more or less between the different MDPs. The same was observed with the other two value functions, thus we will now use the average of the 10 agents per value function as a measure.

In Fig. 7 we can see the average performance of all three value functions over all MDPs. We can see in the 'average' column of bars that a maximum of 70% of the specialized one ply scores is reached by the Naive linear value function. As an average 50% of the specialized score however seems to be maximum reachable. Somewhat surprisingly we see here that the Naive linear value function, which has the most limited representation scores best using this metric. The more complex, but more expressive, value functions clearly score lower here.

In Fig. 8 we can see the performance of the different value functions on what they are selected on during training. The goal of the learning process is to maxi-

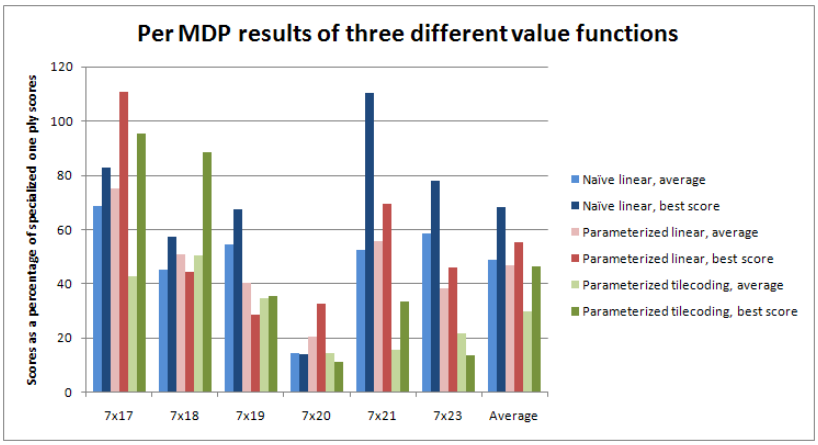


Figure 7: Per MDP results of the 3 different value functions as a percentage of the specialized one ply scores. Both the average over 10 training runs and the maximum score in the 10 training runs are reported.

mize the score as set by the RLC metric. In this figure the average obtained scores over the 10 training runs per value function are shown in combination with their standard deviations. We also show the maximum score obtained per value function in red. We can see here that although the Naive linear architecture scores better as a percentage of the specialized training runs, the Pairwise linear Architecture scores better on the RLC metric. We can also see that the average results of the Pairwise tilecoded architecture compares badly to the other value functions, but the best training run compares much better to the other two training runs. This is also represented by the high variance in scores using the Pairwise tilecoded architecture (as denoted by the error bar).

6 Conclusion

Looking back on our results we can say with high confidence that the two ply look ahead gives a significant benefit, even though the next block is not known. This howevery comes at a great cost of computational complexity. When this however can be reduced by a heuristic pruning method a two ply look ahead could be used in an effective way.

The results of the three different value functions however are less clear. Although the Pairwise Linear architecture outperforms both other value functions, we also observe that the Naive Linear architecture outperforms the Pairwise Tilecoded architecture. Because this last value function is at least equally expressive as the Naive Linear value function we expect that the large number of weights to be learned by the Pairwise Tilecoded architecture proved to be too complex. The large variance in scores of this value function confirms this idea, because it shows that although the average results are lower than the others, it can find solutions that

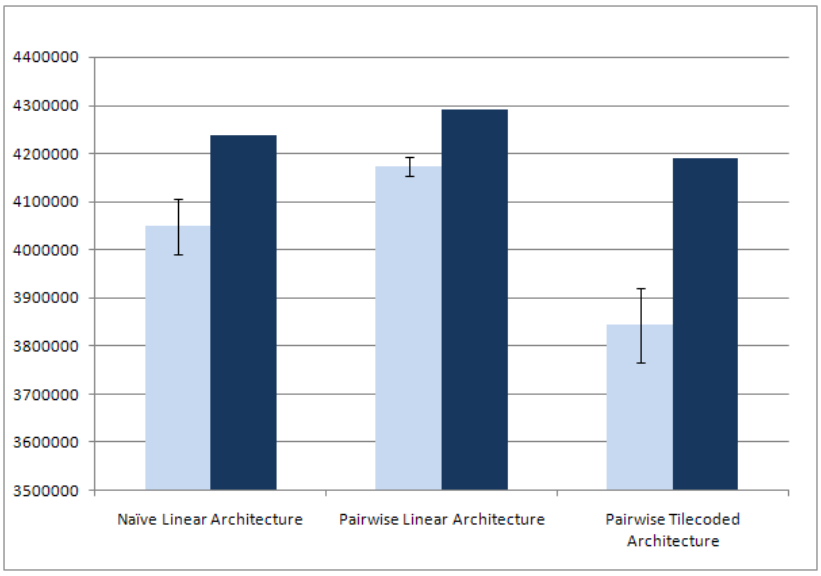


Figure 8: In light blue: Scores obtained using the RLC metric for the three different value functions. The error bars denote the standard deviation over the 10 training runs. In dark blue: The best score over 10 training runs.

are at least equally good.

Using the parameterized Pairwise Linear value function we have found which clearly outperforms the non-parameterized value function. We can thus say that parameterized value functions can be used to generalize over a set of parameterized MDPs. Since there is no comparitive work at the moment, we can not say with high confidence that the form of the value functions used in this research are good in general. Especially the Pairwise Linear architecture requires a linear dependence between the features and parameters. Although this seems to be the case in this research we do not expect this to hold for other domains.

7 Discussion

Although we have found that the Pairwise Linear value function performs best in our setting, we have made some choices that could strongly affect our results. Due to time limitations we capped the number of games per MDP per agent to 2 games. We have used the average of this score as the score of the particular agent on that MDP. Tetris is however known to have large variance in its scores. This means that only 2 games per MDP probably is not enough to determine the performance of the agent with high confidence. This can negatively affect the learning curve because bad playing agents can play 2 good games and get selected in the elite population, or good playing agents can play 2 bad games and not get selected. We think that when more time is available increasing the number of games per MDP would result

in a more stable learning curve and would lead to better results.

Because of the large number of weights for the parameterized value functions we have also increased the variance stop limits for the weights to ensure practical training times. It could however be the case that performance will increase even more, if the weights were finetuned (thus bringing the variance in the weights down to 0). Further research could be done to make these effects more clear.

One of the questions still unanswered in our approach to learning a generalized value function is how to generalize between different field sizes. We have focussed on one field width in particular, ignoring this problem completely. However since the field sizes are discrete and a drastic difference in policies for even and uneven width is possible, tile-coding in this parameter space could fail to express this difference, or simply result in a separate value function for every field size. When different field sizes can be clustered in groups with the same characteristics a Nearest Neighbour approach could be used.

In this research we traded off faster training times for raw perfomance by not using the two ply search. Further research could be done by using this two ply search which handles the block probability parameters in an intuitive way. We expect that taking these block probabilities into account could give a performance increase of several orders of a magnitude and make a clearer distinction between the different types of value functions.

Another improvement which could be made when all paramaters can be controled, and thus sufficient training data is available, is to extend on the tile-coding approach. At the moment we are assuming that every parameter has a separate effect on each feature. It could however be that combinations of parameters have an effect on each feature. The one-dimensional tile-coding in the parameter space can then be extended to a two dimensional tile coded parameter space, where every parameter-pair has a weight for every feature. An example of such a value function is given in (13), where $C(j,k)$ gives the set of all combinations of the possible values for j and k.

$$V(S) = \sum_{i=1}^{|\phi|} \sum_{\substack{j,k \in C(j,k) \\ j \in \{1,..,|\pi|\}, \\ k \in \{1,..,|\pi|\}}} \left(\begin{bmatrix} \pi_k < 0.10 \\ 0.10 \leq \pi_k \leq 0.18 \\ \pi_k > 0.18 \end{bmatrix} \begin{bmatrix} \pi_j < 0.10 \\ 0.10 \leq \pi_j \leq 0.18 \\ \pi_j > 0.18 \end{bmatrix}^T \right) \cdot \begin{bmatrix} \omega_{i,j,k,1} & \omega_{i,j,k,2} & \omega_{i,j,k,3} \\ \omega_{i,j,k,4} & \omega_{i,j,k,5} & \omega_{i,j,k,6} \\ \omega_{i,j,k,7} & \omega_{i,j,k,8} & \omega_{i,j,k,9} \end{bmatrix} \cdot \phi_i(s) + \sum_{l=1}^{|\phi|} \omega_l \cdot \phi_l(s) \quad (13)$$

This idea can be extended into an N-dimensional tile coded parameter space, where N is the number of parameters, and a weight for each feature is given by the complete combination of all N parameters. However increasing the number of weights, will make it ever more complex to learn the right values. There will

also have to be a set of training MDPs with sufficient variation to learn all these weights, and not leave unlearned combinations. However, it would be interesting to investigate whether a generalized policy could be learned this way when such a sufficient training set is available.

References

[1] Reinforcement learning competition 2008. http://www.rl-competition.org.

[2] Bertsekas and Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[3] Bohm, Kokai, and Mandl. Evolving a heuristic function for the game of tetris. *Proc. Lernen,Wissensentdeckung und Adaptivitat*, 2004.

[4] Bohm, Kokai, and Mandl. An evolutionary approach to tetris. In *MIC2005: The Sixth Metaheuristics International Conference*, 2005.

[5] Burgiel. How to lose at tetris. *Mathematical Gazette*, 1997.

[6] de Boer, Kroese, Mannor, and Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 2004.

[7] Demaine, Hohenberger, and Liben-Nowell. Tetris is hard, even to approximate. In *Proceedings of the 9th International Computing and Combinatorics Conference (COCOON 2003)*, 2003.

[8] Fahey. Tetris ai. *http://www.colinfahey.com/tetris*, 2003.

[9] Farias and van Roy. *Tetris: A study of randomized constraint sampling*, pages 189–201. Springer-Verlag, 2006.

[10] Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[11] Hoogeboom and Kosters. How to construct tetris configurations. 2004.

[12] Kakade. A natural policy gradient. In *Advances in Neural Information Processing Systems*, pages 1531–1538. NIPS, 1996.

[13] Lagoudakis, Parr, and Littman. Least-squares methods in reinforcement learning for control. In *Proceedings of the Second Hel lenic Conference on AI*, 2002.

[14] Ramon and Driessens. On the numeric stability of gaussian processes regression for relational reinforcement learning. In *ICML-2004 Workshop on Relational Reinforcement Learning*, 2004.

[15] Sutton and Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[16] Szita and Lorincz. Learning tetris using the noisy cross-entropy method. *Neural Computation*, 2006.

[17] Tsitsiklis and van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 1996.