UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Development of a reinforcement learning agent to play Tetris

by

Andrew Murdoch
20734751

Mechatronic Project 488

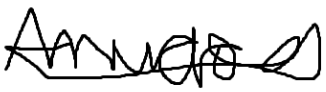
Final report

Study leader: Mr. JC Schoemann

July 2021

Plagiarism declaration

I have read and understand the Stellenbosch University Policy on Plagiarism and the definitions of plagiarism and self-plagiarism contained in the Policy [Plagiarism: The use of the ideas or material of others without acknowledgement, or the re-use of one's own previously evaluated or published material without acknowledgement or indication thereof (self-plagiarism or text-recycling)]. I also understand that direct translations are plagiarism, unless accompanied by an appropriate acknowledgement of the source. I also know that verbatim copy that has not been explicitly indicated as such, is plagiarism. I know that plagiarism is a punishable offence and may be referred to the University's Central Disciplinary Committee (CDC) who has the authority to expel me for such an offence. I know that plagiarism is harmful for the academic environment and that it has a negative impact on any profession. Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully (acknowledged); further, all verbatim copies have been expressly indicated as such (e.g. through quotation marks) and the sources are cited fully. I declare that, except where a source has been cited, the work contained in this assignment is my own work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment. I declare that have not allowed, and will not allow, anyone to use my work (in paper, graphics, electronic, verbal or any other format) with the intention of passing it off as his/her own work. I know that a mark of zero may be awarded to assignments with plagiarism and also that no opportunity be given to submit an improved assignment.



Signature: ..... Date: 2021/07/01

A. Murdoch

EXECUTIVE SUMMARY

Title of project
Development of a reinforcement learning agent to play Tetris.
Objectives
• Develop a suitable Tetris simulation environment for algorithm development. • Develop a model-free reinforcement learning agent utilising linear function approximation for Tetris. • Compare the effect of reward signals, feature representations, hyperparameters and learning methods on the training time and performance of the agent.
What is current practice and what are its limitations?
Recent attempts at creating artificial intelligence to play Tetris use non-linear function approximation, model-based reinforcement learning, or evolutionary algorithms. The training times reported for algorithms in literature are between 1 and 3 months.
What is new in this project?
The use of model-free reinforcement learning with linear function approximation applied to Tetris.
Was the project successful? How will it make a difference?
Yes, an agent was developed that could play a simplified version of Tetris after one hour of training. The solution method developed may be applied to industry related problems.
What were the risks to the project being a success? How were these handled?
The main risk was time constraint, as the deadline is in July. This risk was mitigated by thorough project planning and regular consultation with supervisor.
What contributions have/ will other students made/make?
N/A.
Which aspects of the project will carry on after completion and why?
There is no planned direct continuation of the project.
What arrangements have been made to expedite continuation?
N/A.

ECSA exit level outcomes

ECSA Outcome	Addressed in Chapters
ELO 1. Problem solving: Demonstrate competence to identify, assess, formulate and solve convergent and divergent engineering problems creatively and innovatively.	1, 5, 6, 7
ELO 2. Application of scientific and engineering knowledge: Demonstrate competence to apply knowledge of mathematics, basic science and engineering sciences from first principles to solve engineering problems.	3, 4, 5, 7
ELO 3. Engineering design: Demonstrate competence to perform creative, procedural and non-procedural design and synthesis of components, systems, engineering works, products or processes.	5, 7
ELO 4. Investigations, experiments and data analysis: Demonstrate competence to design and conduct investigations and experiments.	5, 6
ELO 5. Engineering methods, skills and tools, including information technology: Demonstrate competence to use appropriate engineering methods, skills and tools, including those based on engineering technology.	3, 4, 5, 6, 7
ELO 6. Professional and technical communication: Demonstrate competence to communicate effectively, both orally and in writing, with engineering audiences and the community at large.	Project proposal, Final report, oral presentation, poster

ELO 7. Impact of Engineering activity: emonstrate critical awareness of the impact of engineering activity on the social, industrial and physical environment.	1
ELO 8. Individual, team and multidisciplinary working: Demonstrate competence to work effectively, in teams and in multi-disciplinary environments.	All aspects of project
ELO 9. Independent learning ability: Demonstrate competence to engage in independent learning through well developed learning skills	2, 3, 4
ELO 10. Engineering Professionalism: Demonstrate critical awareness of the need to act professionally and ethically and to exercise judgment and take responsibility within own limits of competence.	All aspects of project

Contents

List of Figures viii

List of Tables x

List of symbols xi

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Project objectives and motivation . . . 3
- 1.3 Document outline . . . 3

2 Literature study 4
- 2.1 Standard Tetris . . . 4
- 2.2 Melax's simplified tetris . . . 5
- 2.3 Algorithms to solve Tetris . . . 6
  - 2.3.1 Rule-based decision makers . . . 6
  - 2.3.2 Evolutionary algorithms . . . 6
  - 2.3.3 Reinforcement learning . . . 7
- 2.4 Feature sets . . . 8

3 Markov decision processes 12
- 3.1 Formal definition . . . 12
- 3.2 The Agent environment interface . . . 12
- 3.3 Rewards and returns . . . 13
- 3.4 Policies and values functions . . . 14

4 Reinforcement learning 16
- 4.1 Policy iteration . . . 16
- 4.2 Temporal difference learning methods . . . 17
  - 4.2.1 Sarsa . . . 17
  - 4.2.2 n-step Sarsa . . . 18
  - 4.2.3 Sarsa(λ) . . . 18
  - 4.2.4 Q-learning . . . 19
- 4.3 Approximate solution methods . . . 19
  - 4.3.1 Linear function approximation . . . 20

4.3.2 Linear function approximation applied to learning methods 21

5 Applying reinforcement learning to Tetris 23
5.1 Training and testing loops . . . 23
5.2 Tetris simulation . . . 24
5.2.1 Playing field and tetromino implementation . . . 24
5.2.2 State and action spaces . . . 25
5.2.3 Feature vectors . . . 27
5.2.4 Reward function . . . 28
5.2.5 Tetris simulation logic . . . 28
5.3 Agent implementation . . . 28
5.3.1 Action selection . . . 30
5.3.2 Finding the approximate action-values . . . 30

6 Experiments and results 32
6.1 Reward Function . . . 33
6.2 Playing field feature representation . . . 35
6.3 Learning algorithms . . . 35
6.4 Hyperparameters . . . 37
6.4.1 Exploration rate . . . 37
6.4.2 Step size . . . 38
6.4.3 Reward discount rate . . . 39
6.4.4 Initial weight vector . . . 40
6.5 Final agent . . . 41

7 Final agent implementation 44
7.1 Widening the playing field . . . 44
7.2 Playing with standard Tetrominos . . . 48

8 Conclusion 49

A Pseudocode for various learning methods 51

B Results 55
B.1 Reward Signal . . . 55
B.2 Learning methods . . . 58

C Safety guidelines 60
C.1 Overview . . . 60
C.2 General laboratory safety . . . 60
C.2.1 Covid . . . 60
C.2.2 Emergency exit . . . 60
C.2.3 Equipment safety . . . 61

D Project schedule 62

E Project cost and techno-economic analysis **63**
E.1 Project costs 63
E.2 Technical impact 64
E.3 Return on investment 65
E.4 Potential commercialisation 65

List of References **66**

List of Figures

1.1 Tetris playing field 2

2.1 The tetromino shapes 4
2.2 Simplified tetromino shapes in Melax's Tetris 5
2.3 Melax's Tetris playing field 5
2.4 Tetris playing field with a few identified features 10

3.1 The agent environment boundary 13

5.1 Flow diagram for training loop 24
5.2 Playing field matrix 25
5.3 Tetromino matrix 25
5.4 Flow diagram for Tetris simulation 29
5.5 Flow diagram for agent 30

6.1 Learning curves for optimal values of individual reward features . 33
6.2 Learning curves for varying the reward for clearing a line 34
6.3 Learning curves for different feature vectors 36
6.4 Learning curves for various learning algorithms 37
6.5 Learning curves for varying values of exploration rate 38
6.6 Learning curves for varying step size values 39
6.7 Learning curves for varying reward discount rate values 40
6.8 Learning curves for varying initial weight values 41
6.9 Learning curve for the final agent using optimal parameters 42
6.10 Score distribution for final agent 42

7.1 Final solution playing field setup 45
7.2 Final solution flow diagram 46

A.1 Pseudocode for the Sarsa algorithm 51
A.2 Pseudocode for the n-step Sarsa algorithm 52
A.3 Pseudocode for the Sarsa(λ) algorithm 53
A.4 Pseudocode for the Q-learning algorithm 54

B.1 Learning curves for varying the penalty that an agent receives for losing a game 55

B.2 Learning curves for varying the penalty that an agent receives for increasing the maximum column height 56
B.3 Learning curves for varying the penalty that an agent receives for creating a hole 57
B.4 Learning curves for varying the penalty that an agent receives for making the playing field more uneven 58
B.5 Learning curves for n-Step Sarsa with varying n 59

D.1 Project proposed and actual timelines 62

E.1 Comparison of project proposed and actual budgets 64

List of Tables

2.1 Scores for clearing lines 5
2.2 A summary of Reinforcement learning algorithms in the literature 9
2.3 Significant feature sets 10

5.1 A summary of playing field feature representations 27

6.1 Standard parameters for test runs 32
6.2 Final reward signal coefficient values 34
6.3 Playing field feature representation performances in testing 35
6.4 Results for learning methods 36
6.5 Results for different exploration rates 38
6.6 Results for different step size values 39
6.7 Results for varying discount rate 40
6.8 Results for different initial weights 41

7.1 Scores for final solution 47

C.1 Contact details in case of emergency 60

E.1 Project proposed cost 63
E.2 Project actual cost 64

List of symbols

Mathematical expressions

$Pr\{X = x\}$	probability that a random variable X takes on the value x
$X \sim p$	random variable X selected from distribution $p(x) = Pr\{X = x\}$
$\mathbb{E}\|X\|$	expectation of random variable X, $\mathbb{E}\|X\| = \sum p(x)x$
Δ	partial derivative
∇	partial derivative of a column vector

Subscripts

$_\pi$	evaluate expression under policy π
$_*$	evaluate under the optimal policy, π_*
$_t$	at time t

Superscripts

$\hat{}$	function approximation
$'$	at the next time step

Agent hyperparameters

α	step size parameter
γ	discount parameter
λ	decay rate for eligibility traces
ϵ	probability of taking a random action while following ϵ-greedy policy

Markov decision processes and reinforcement learning

a	single action
A_t	action at time t
$\mathcal{A}$	set of all actions
s	single state
S_t	state at time t

$\mathcal{S}$	set of all states
r	reward
R_t	reward at time t
π	policy
$\pi(a\|s)$	probability of taking action a in state s under policy π
π_*	optimal policy
$v_\pi(s)$	true value of state s under policy π
$q_\pi(s,a)$	true value of state action pair (s,a) under policy π
$q_*(s,a)$	true value of state action pair (s,a) under the optimal policy π_*
T	final (terminating) time step of an episode
$Q(s,a)$	estimate of $q_\pi(s,a)$
G_t	discounted return following time t
$G_{t:t+n}$	n-step discounted return from $t+1$ to $t+n$
$\hat{q}(s,a,\mathbf{w})$	approximate value of state action pair (s,a), given a weight vector $\mathbf{w}$
$\mathbf{x}(s,a)$	feature vector
$\mathbf{w}$	weight vector
δ	temporal difference error

Chapter 1

Introduction

Reinforcement learning is a machine learning technique by which an agent is trained to make a sequence of decisions. The agent is allowed to interact with its environment, employing a trial and error approach to developing the policy by which it takes actions. It has found many interesting applications: recommender systems in Netflix and Spotify; stock trading systems; automated medical diagnoses and path planning for autonomous vehicles. These scenarios are difficult to simulate, making algorithm development tricky. Furthermore, deploying reinforcement learning agents in these environments involve risk: an incorrect choice may lead to financial loss or damage to equipment. For this reason, video games have become a popular test bed for algorithm development in the reinforcement learning research community. They are a safe environment that can be simulated and accelerated.

1.1 Background

Tetris is a popular video game created by Alexey Pajitnov in 1984 in which the player needs to place falling shapes, called tetrominos, on a 10x20 grid. Once the tetromino reaches the bottom or collides with another placed tetromino, it becomes fixed in place and the next tetromino starts falling from the top of the screen. When a row is entirely filled with tetromino blocks, it is cleared from the screen and all of the filled cells above it are shifted one row downwards. The goal of the game is to clear as many rows as possible before the stack of occupied cells reach the top of the screen, at which point the game ends. A picture of the playing field is shown in Figure 1.1. The game has increased in popularity since its conception, and has sold over 425 million copies, more than any other video game in existence.

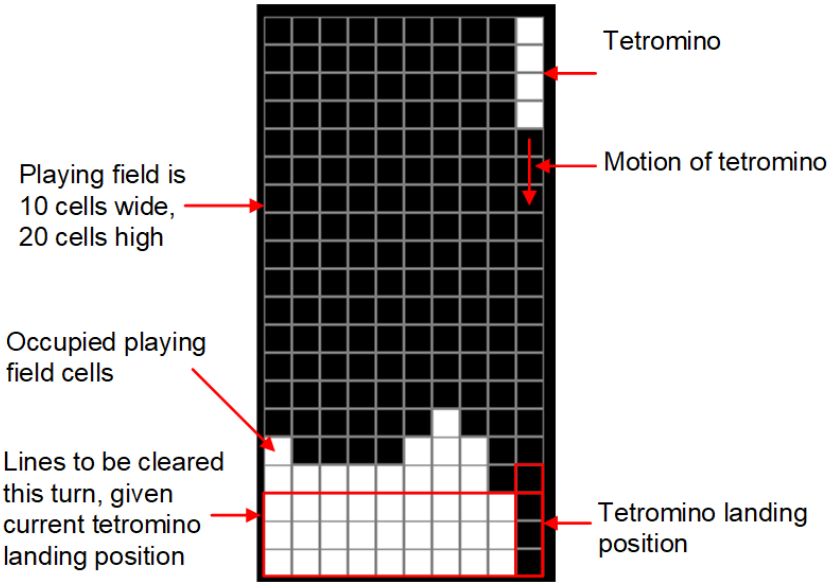


Figure 1.1: The Tetris playig field

Tetris has long been used as a testbed for algorithm development, and has gained a reputation for being difficult to solve. This is due to the large number of states the agent may encounter while playing: there are 200 grid cells that can each take on one of two values, resulting in an upper bound of 2^{200} possible playing field configurations. Accounting for the type, rotation, vertical and horizontal positions of the tetromino, the upper bound for the total number of states is $6.1 \cdot 10^{63}$. It has also been mathematically proven to be NP complete, meaning that it cannot be solved in polynomial time in any known way [1]. This has lead to the development of algorithms with exceedingly long training times. Amundsen [2], Szita and Lorincz[3], and Thiery and Scherrer [4] report training times of one month, while Bohm et al. [5] reports an execution time of three months to train an agent to play Tetris. These excessively long training times are a common problem that hinder algorithm development. Reducing the complexity of an algorithm may decrease its training time, but at the cost of reduced agent performance. A few simple techniques can be considered: model-free learning can be used as opposed to model-based learning. In model-free techniques, the agent learns directly from the environment without any domain knowledge. In model-based techniques, the agent constructs a model of its environment which it uses to plan ahead. Another simplification is to use linear regression as opposed to a neural network as a value function approximator. This approach is explained in Chapter 4.

1.2 Project objectives and motivation

This project, conducted by Mr A. Murdoch as part of the module Mechatronics Project 488, stems from a proposal by Mr. JC Schoemann to develop a reinforcement learning agent that is capable of playing Tetris. The goal is to develop the agent using a less computationally expensive training method than those found in the literature: model-free reinforcement learning with linear function approximation. There are three project objectives:

1. Develop a suitable Tetris simulation environment for accelerated algorithm development.

2. Develop and determine the suitability of a model-free reinforcement learning agent utilising linear function approximation for Tetris.

3. Compare the effect of reward signals, feature representations, hyperparameters and learning methods on the training time and performance of the agent.

The training times mentioned in the previous section of one to three months severely hinder algorithm development. An investigation into the suitability of using model-free learning and linear function approximation in complex simulation environments is pertinent because these techniques may result in faster training times. The question to be answered through this project is whether they will result in an agent with adequate performance. The value in finding this result for Tetris is that the methods developed in this project may be extended to real world scenarios.

1.3 Document outline

The literature review in Chapter 2 introduces the rules of standard Tetris as well as a simplified variant of thereof, called Melax's Tetris. Current approaches to solving Tetris are then discussed. Chapter 3 presents an introduction to markov decision processes, which is the framework for reinforcement learning. Chapter 4 serves as an introduction to reinforcement learning theory. The experimental setup used to train the agent is discussed in Chapter 5. Several simplifications made to the game are discussed here as well. Chapter 6 presents results from training with the these simplifications in place. Chapter 7 then presents solutions to remove the simplifications from the game.

Chapter 2

Literature study

This chapter summarises research work done on applying reinforcement learning to Tetris prior to this project. The standard version of Tetris is described, as well as a simplified version thereof considered by some researchers. A summary of previous attempts at training an agent to play Tetris is given, along with important feature sets used by researchers.

2.1 Standard Tetris

The Tetris guideline [6] is the current specification that all Tetris game products must conform to. The important rules of the game listed in the guideline are summarised in this section.

The visible playing field is 10 wide by 20 high. There are 7 tetrominos consisting of 4 cells each. They are shown in Figure 2.1, and their names are indicative of their shape.

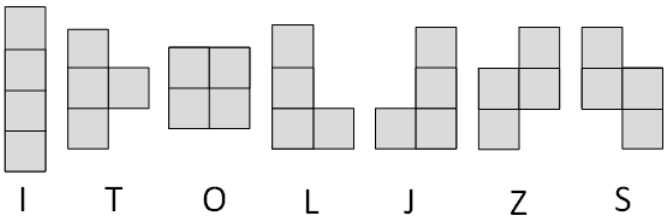


Figure 2.1: The tetromino shapes

There may only be 1 falling tetromino in the playing field at a time, so a new tetromino spawns only after the previous tetromino has landed on the playing field. The tetrominos each have an equal likelihood of spawning. They appear on the vertical centre line of the playing field, directly above the top most visible row. Once spawned, they move downwards at a rate proportional to the level number. The player controls the tetromino with the actions translate left;

translate right; rotate clockwise; soft drop (moves the tetromino downwards quickly) and hard drop (moves the tetromino to the bottom of the playing field instantly). Score is earned by filling an entire row of the playing field with tetromino cells, which is referred to as clearing a line. The score earned is dependent on the number of lines cleared in 1 move, shown in Table 2.1.

Lines Cleared	Score
1	1
2	3
3	5
≥ 4	8

Table 2.1: Scores for clearing lines

2.2 Melax's simplified tetris

Melax [7] simplified Tetris considerably to apply reinforcement learning to it in 1998. He kept the original set of actions, but changed the playing field and tetrominos considerably. The tetromino set he considered is shown in Figure 2.2.

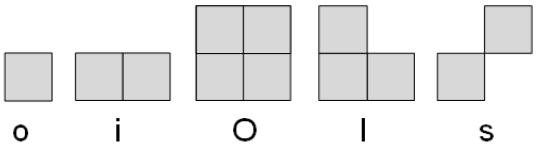


Figure 2.2: Simplified tetromino shapes in Melax's Tetris

In Melax's implementation the playing field is infinitely high, but the visible playing field are the rows of the highest occupied cell, and the row beneath it. Whenever a tetromino occupies a cell above the visible part of the playing field, the playing field shifts upwards. The cells beneath the visible playing field are then discarded. The agent is penalised for increasing the height of the visible playing field. The agent was limited to 10000 tetrominos dropped. This logic is demonstrated in Figure 2.3.

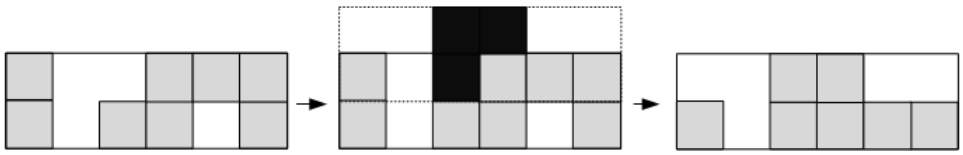


Figure 2.3: Melax's Tetris playing field, adapted from Carr [7].

2.3 Algorithms to solve Tetris

There have been many successful attempts at creating agents to play the standard version of Tetris. To compare agents, most researchers report score in terms of average number of lines cleared per game. Note that variations in reported scores may be partly due to subtle implementation differences between Tetris environments. Where training time was reported, it was given either in terms of the number of episodes (games played). The most common approaches to solving Tetris are rule-based decision makers, evolutionary algorithms, and reinforcement learning agents.

2.3.1 Rule-based decision makers

Rule-based decision makers are programmed to take actions deterministically using a series of if statements that check the playing field for certain conditions. The two most successful rule-based agents were developed by Delacherie, who's agent averaged 650000 lines cleared, and Fahey, who's agent cleared 7216290 lines in the 1 game it was tested. The drawback of this approach is that the decision maker has no ability to alter its policy to account for variations in the game implementation. Furthermore, the assumption is made that the programmer knows the optimal policy. This is often not true, and the decision maker's performance is limited to the level of policies developed by humans. Therefore, decision makers that learn dynamic policies (such as evolutionary algorithms and reinforcement learning) have the potential to outperform rules-based agents [7].

2.3.2 Evolutionary algorithms

Evolutionary algorithms are approaches to learn decision making policies that mimic biological evolution. They typically generate a population of agents with random policies for the entire state space. The agents' performances are evaluated according to a fitness function, and a new population of agents is generated by combining and randomly adding slight variations to policies from the previous generation's best performing agents. New generations are created until some stopping condition is reached.

Bohm et al. [5] developed evolutionary algorithms for Tetris in 2005, reporting 480 million lines cleared from an agent utilising a linear fitness function after 3 months of training time. Note that this score was achieved in a version of the game where the agent knows which tetromino will be spawned next, and as such, is difficult to compare with other results. Szita and Lorinez [3] then achieved 350 000 lines cleared in 2006 by applying principles of cross entropy to evolutionary algorithms, reporting a training time of one month. Thiery and Scherrer [8] further developed Szita and Lorinez's work by adding several features to their state representation, achieving 35 million lines cleared after training the algorithm for one month.

2.3.3 Reinforcement learning

As stated in the introduction, reinforcement learning is a machine learning technique whereby an agent is trained to make a sequence of decisions through its interactions with the environment. The set of variables completely describing the environment is called the state. In Tetris, the state is the playing field configuration (occupied and unoccupied cells), the tetromino type, position and rotation. Whenever the agent takes an action, it is rewarded (or penalised). This reward is typically based on the game scoring function. The agent assigns a value to each state that it encounters based on the reward that it receives following that state. The goal of reinforcement learning is to select actions which result in the maximum cumulative reward. It does so by selecting actions that lead to states with the highest value. In the case of Tetris, the state space is too large to be represented as in tabular form. Therefore, the value of a given state must be approximated with a function approximator. The discussion that follows is ordered by how the researchers treated the value function. See Table 2.2 for a summary of the reinforcement learning agents in the discussion.

Agents using linear value function approximation

An early application of reinforcement learning to Tetris was by Tsitsiklis and Van Roy [9], who achieved an average of 31 lines cleared over 100 games by using a model-based reinforcement learning technique called dynamic programming in 1996. An important contribution that they made to literature was to simplify the game by not allowing the tetromino to fall gradually. Instead, the tetromino appears in a staging area at the top of the playing field. The agent can position the tetromino within the staging area before dropping it, but does not have control over the tetromino once it is dropped. This simplifies the game in that the agent does not need to consider the vertical position of the tetromino [4].

One year after Tsitsiklis and Van Roy presented their research, Bertsekas and Ioffe [10] developed and applied temporal difference learning to Tetris, achieving 3200 lines. Temporal difference learning was an important breakthrough in the field of reinforcement learning, and is discussed in Chapter 4.

A more recent attempt at model-based learning was by Thiam et al. [11], who trained an agent using the Sarsa learning rule (a temporal difference method). They reported that their agent played $400 \cdot 10^3$ tetrominos in 1 game after training with $3.5 \cdot 10^9$ gaming pieces. Their decision to report the performance in terms of tetrominos played makes it difficult to compare their results with other researchers. They did, however, contribute to literature by investigating the effect of reward functions on agent performance. Rather than using the game scoring function, Thiam et al. used a reward function that rewards or penalised an agent every move based on the change in the

fitness function in Equation 2.1 during that move. This guided the agent to take desirable actions, significantly increasing its performance.

$$\begin{aligned}\text{fitness function} = 5 \cdot (\text{average column height}) + 16 \sum \text{holes} \\ + \sum (\text{column height differences})^2\end{aligned} \tag{2.1}$$

Agents using non-linear value function approximation

Recently, the focus has shifted from linear regression to neural networks for value function approximation. One such algorithm was devloped by Steven and Pradhan [12], who reported that their model-free deep Q-learning agent played 980 pieces in one game. Once again, the choice to report scores in terms of tetrominos played makes it difficult to compare the performance of their agent to others in literature. They did, however, contribute to literature by investigating the effect of grouping actions together. They found that considering the entire sequence of moves to place a tetromino on the playing field as a single action improved the performance of their agent.

Agents using parameterized policy iteration

An alternative technique to approximating the value function is to evaluate and improve policies directly. This is known as parameterized policy iteration. Kakade [13] reported 6800 lines using one such method called the natural policy gradient. The most recent attempt at policy iteration in the literature reviewed was by Gabillon et al. [14], who achieved 51 million lines cleared in 2013. Reinforcement learning methods that learn a parameterized policy are more complex but perform better than those that learn a value function.

2.4 Feature sets

Previously, the state was defined simply as the set of variables describing the environment. However, there are many choices as to how the state is represented to the agent. In Tetris, this means selecting a set of features to represent the playing field configuration. These features can be chosen cleverly to minimise memory requirements and improve the agent's performance. Important feature sets considered by researchers in the literature are summarised in Table 2.3. Some of these features are visualised in Figure 5.2.

Author	**Learning method**	**Value function/ parameterized policy**	**Model-based/ model-free**	**Function approximator**	**Score**
Tsitsiklis and Van Roy [9]	Dynamic programming	Value function	Model-based	Linear	31 lines
Thiam et al. [11]	Sarsa	Value function	Model-based	Linear	$400 \cdot 10^3$ tetrominos
Bertsekas and Ioffe [10]	Temporal Difference	Value function	Model-based	Linear	3200 lines
Stevens and Pradhan [12]	Q-learning	Value function	Model-free	Neural network	980 tetrominos
Kakade [12]	Natural policy gradient	Parameterized policy	Model-free	-	6800 lines
Gabillon et al. [14]	Approximate dynamic programming	Parameterized policy	Model-free	-	$51 \cdot 10^6$ lines

Table 2.2: A summary of Reinforcement learning algorithms in the literature

Feature Description	Tsitsiklis & Van Roy [9]	Bertsekas & Ioffe [10]	Dellacherie [15]	Bohm et al. [5]	Stevens & Pradhan [12]	Groß et al. [16]
Column heights	x	x				
Total number of holes	x	x	x	x		
Height differences between columns		x				x
Maximum column height		x		x		
Landing height of falling piece			x	x		
Row transition			x	x		
Column transition			x	x		
Height difference between highest and lowest columns				x		
Maximum well depth				x		
Lines cleared in last move				x		
Pixel array of playing field					x	

Table 2.3: Significant feature sets

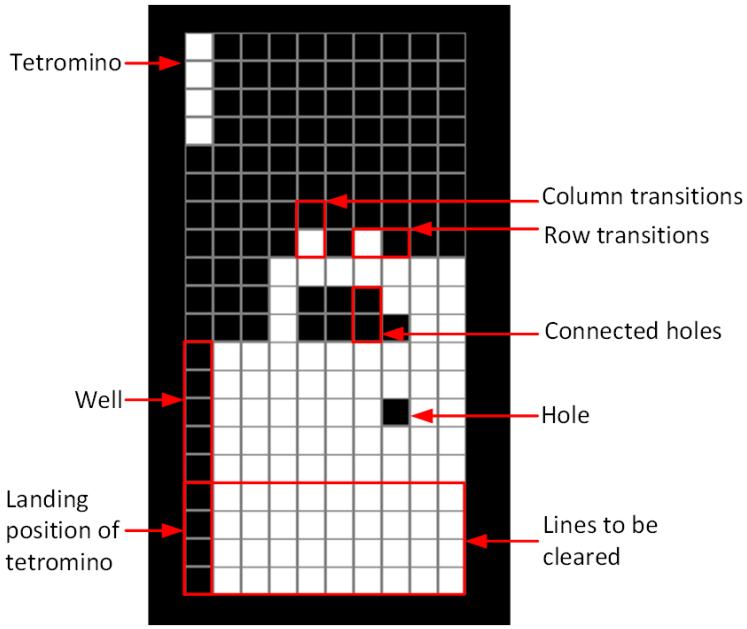


Figure 2.4: Tetris playing field with a few identified features

Three approaches to solving Tetris were discussed in this section: rule-

based decision makers, evolutionary algorithms and reinforcement learning. Rule-based decision makers have performed well, but cannot alter their policy. This gives learning agents an advantage, as they have the potential to improve. Although evolutionary techniques generally outperform reinforcement learning, Carr [7] argued in favor of adopting reinforcement learning approaches to Tetris, stating that while genetic algorithms evaluate the policies of an agent over the entire set of states, reinforcement learning evaluates the policy followed by an agent in every individual state. This pushes the policy followed in a specific state towards the optimal action for that state. Furthermore, due to the mechanisms by which the value function is updated, reinforcement learning approaches allow the agent to learn during a game, rather than to update its policies after the game is finished. This allows for faster training times.

Chapter 3

Markov decision processes

Having identified reinforcement learning as the solution technique for this project, we proceed with an introduction to the markov decision process (MDP). This is the specific type of decision sequence that reinforcement learning solves. This discussion is adapted from Silver's reinforcement learning course [17] and Sutton and Barto's reinforcement learning textbook [18].

3.1 Formal definition

A Markov decision process is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where

- $\mathcal{S}$ is a set of states
- $\mathcal{A}$ is a set of actions
- $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$ is a state transition probability matrix
- $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$ is a reward function
- $\gamma \in [0, 1]$ is a discount factor.

The next three sections explain how each element of this five-tuple is used to construct a sequential decision process that can be solved using reinforcement learning. The first point to discuss is how the agent and the environment must interact.

3.2 The Agent environment interface

In an MDP, the agent and the environment are defined as separate entities. They interact in a sequence of discrete time steps, $t = 0, 1, 2, 3, ...$. At each time step the agent receives a state representation $S_t \in \mathcal{S}$, and selects an action $A_t \in \mathcal{A}$. The environment then executes A_t, modifying the state according to $\mathcal{P}^a_{ss'}$, which is the probability of transitioning into a state given the current state and action taken. The time step t is incremented, and the environment

outputs a reward $R_{t+1} \in \mathcal{R}_s^a$ and next state representation S_{t+1}. The next state and reward is received by the agent at time step $t+1$. This gives rise to the trajectory

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ... \tag{3.1}$$

The interaction between the agent and environment is shown in Figure 3.1.

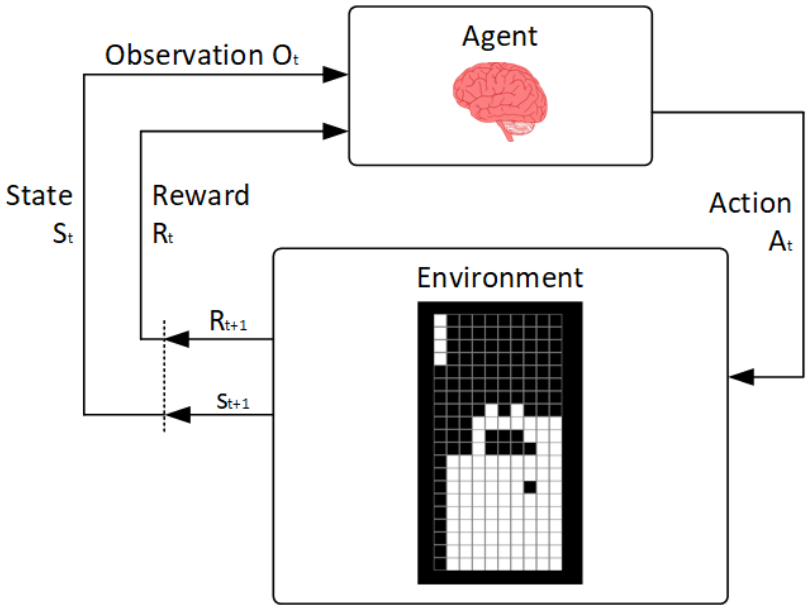


Figure 3.1: The agent environment boundary. Note, figure is adapted from [18]. Image of brain by openClipart-Vectors, pixabay (www.pixabay.com). Creative commons licence.

3.3 Rewards and returns

The goal of the agent is to maximise the expected value of the cumulative sum of a received discounted scalar reward signal, denoted the return. This is formally expressed as

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.2}$$

where $0 \leq \gamma \leq 1$ discounts the future rewards. This is useful for episodic tasks, as it controls how strongly present and future rewards are weighted. When $\gamma = 1$, the rewards in the far future count just as much as the immediate reward. When $\gamma = 0$, only the immediate reward matters. The return at any

time step can be related to the return at future time steps

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3+...}) \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{3.3}$$

3.4 Policies and values functions

To solve an MDP, a policy must be formulated. The policy is a probability distribution of actions that the agent might take, given the current state. Mathematically, this is expressed as

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]. \tag{3.4}$$

The first step in computing the policy is to attach a value to each state. The value of being in state s and taking action a, then following policy π thereafter is the action-value function, denoted $q_\pi(s, a)$. The action-value function q_π is represented in a table format, with each entry in the table corresponding to a state-action pair (s, a). The value of each entry is defined as expected return following the state-action pair corresponding to that entry:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, At = a]. \tag{3.5}$$

Equation 3.5 can decomposed into immediate reward plus the discounted value of the next state-action pair by substituting Equation 3.3 for G_t,

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]. \tag{3.6}$$

This equation relates the state-action pair's value to the value of the state action pair at the next time step. When the expectation $\mathbb{E}$ is removed, Equation 3.6 becomes the Bellman Expectation Equation:

$$q_\pi(s, a) = \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a'). \tag{3.7}$$

This equation states that the value of a state-action pair is the sum of the immediate reward and discounted averaged value of all possible state-action pairs reachable from (s, a), weighted by the state-transition probabilities and policy. Note that $q_\pi(s, a)$ is dependent on the policy. Therefore, the value of $q_\pi(s, a)$ can be improved by selecting a better policy. The goal of the agent (which is to maximise the accumulated reward) is achieved by finding the optimal action-value function $q_*(s, a)$. This is the maximum action-value function over all action-value functions:

$$q_*(s, a) = \max_\pi q_\pi(s, a). \tag{3.8}$$

The MDP is considered solved when $q_*(s, a)$ is known, because an optimal policy is found by selecting actions that maximise over $q_*(s, a)$:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \max_{a \in \mathcal{A}} q_*(s, a) \\ 0 & \text{otherwise.} \end{cases} \tag{3.9}$$

Lastly, as the value functions $q_\pi(s, a)$ is related to itself in the Bellman Expectation Equation 3.7, the optimal action-value function is related to itself by the Bellman Optimality equation:

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a'). \tag{3.10}$$

This equation states that the optimal action-value function at the current state is the sum of the immediate reward and discounted averaged optimal state-action pair values reachable from the current state by taking the optimal action. Reinforcement learning techniques solve the MDP by computing the Bellman Optimality Equation. They do this by iteratively predicting the value function of their current policy, $q_\pi(s, a)$, and improving it, until $q_\pi(s, a) = q_*(s, a)$.

This concludes the theory of MDPs, the decision process that reinforcement learning is designed to solve. Tetris can be modelled as an MDP by populating the MDP five-tuple in Section 3.1: $\mathcal{S}$ is the playing field and tetromino configurations, $\mathcal{A}$ is the set of actions available to the player, $\mathcal{P}_{ss'}^a$ is defined by the game dynamics, and R_s^a is the game scoring function. γ is a hyper-parameter that can be chosen. The next chapter will explain reinforcement learning methods in more detail, after which they will be applied to solve Tetris.

Chapter 4

Reinforcement learning

Having introduced the necessary theory surrounding MDPs, we now present the theory for reinforcement learning, an iterative solution technique to solve MDPs. Reinforcement learning solves the MDP by iteratively generating new policies, each better than the previous one, until the optimal policy is found. This is known as policy iteration. The theory presented in this chapter is adapted from Silver's reinforcement learning lectures [17] and Sutton and Barto's reinforcement learning textbook [18].

4.1 Policy iteration

The process of generating better policies happens in 2 steps:

1. Policy evaluation
2. Policy improvement.

In policy evaluation, an estimate of q_π, denoted Q, is made more accurate using an update rule:

$$\begin{aligned} Q_{new} &\leftarrow Q_{old} + \alpha[error] \\ &\leftarrow Q_{old} + \alpha[target - Q_{old}]. \end{aligned} \tag{4.1}$$

The Q values are updated towards a target - this target is dependent on the learning method, and is discussed in the next section. α is the step size parameter, controlling how strongly Q is updated towards the target. Policy improvement then generates a new policy for a state by choosing the state-action pair with the highest value:

$$\pi'(s) = \max_{a \in \mathcal{A}} Q(s, a). \tag{4.2}$$

The new policy generated is guaranteed to be better then the previous one. Policy iteration occurs until improvement stops (Q remains constant over policy iterations). At this point, the Bellman Optimality Equation 3.10 has been

satisfied. This means that the current value function is the optimal value function ($Q = q_*$) and the current policy is an optimal policy ($\pi = \pi_*$).

Three common techniques for performing policy iteration are dynamic programming, Monte Carlo methods and temporal difference learning. Dynamic programming does not sample from experience - the agent does not interact with the environment in a regular, sequential manner. Instead, the value of states are computed in any order. Because it does not sample from experience, dynamic programming requires a model of the environment. However, it is too computationally expensive to represent probabilistic models (the state transition and reward matrices). As such, model-free techniques are favored.

Monte Carlo methods and temporal difference learning avoid the need for a model by sampling from experience - the agent interacts with the environment in the way that a human would play a game. Monte Carlo samples from complete episodes, meaning that policy evaluation and improvement happens only after the episode is finished, and is based on the return for the entire episode. Temporal difference learning samples from incomplete episodes, and the policy is evaluated and improved at every time step. Temporal difference learning is preferred over Monte Carlo because the updates have lower variance and it tends to converge faster. It is therefore the learning method we will use to solve Tetris.

4.2 Temporal difference learning methods

As stated earlier, temporal difference learning updates Q towards q_π using Equation 4.1, then the the policy is improved using Equation 4.2 at every time step. There are several choices for the *target* in the update rule, with slightly different algorithms resulting from each choice. Sarsa, n-step Sarsa, Sarsa(λ) and Q-learning are common choices, and are discussed here.

4.2.1 Sarsa

The name Sarsa comes from the sequence of agent-environment interactions S_0,A_0,R_1,S_1,A_1.... The idea is to set the target in Equation 4.1 equal to the reward plus discounted value of the next state visited:

$$target = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}). \tag{4.3}$$

Note that Sarsa samples the next state and action. This means that the time step is at $t + 1$ when the Q value of the state at t is updated. With the substitution, the update rule is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(St, At)]. \tag{4.4}$$

4.2.2 n-step Sarsa

The Sarsa method updates $Q(S, A)$ toward the immediate observed reward and the bootstrapped value of the next state, while the Monte Carlo method mentioned earlier bases the update on the return received during the entire episode. It is often best to update $Q(S, A)$ based on reward for an intermediate number of steps. This because bootstrapping works best if it is over a length of time where a recognisable state change has occurred. The n-step Sarsa algorithm achieves this: The update at time step t is based on the reward received at the n next states and bootstrapped value at time step $t + n$. The return for all states up until $T - n$ is calculated using

$$G_{t:t+n} = R_{t+1} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), \quad n \geq 1, 0 \leq t < T - n. \tag{4.5}$$

To account for states less than n steps away from the terminal state T, the return from state $T - n$ to T is

$$G_{t:t+n} = G_t \text{ if } t + n \geq T. \tag{4.6}$$

The update rule then becomes:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \quad 0 \leq t < T. \tag{4.7}$$

4.2.3 Sarsa(λ)

Sarsa(λ) is the average of all n-step returns weighted by λ. This method generalises Monte Carlo and temporal difference learning, such that a spectrum of methods is produced that has Monte Carlo (when $\lambda = 1$) on one end, and Sarsa (when λ=0) on the other. This usually allows for more efficient learning, but it does present a problem: all the n-step returns are only available at the end of the episode. Although it is possible to perform the update a the end of the episode, known as the forward view of Sarsa(λ), it is not the preferred update mechanism. It is better to update at every time step, an approach known as the backwards view. This is made possible with the use of eligibility traces, denoted z. The eligibility trace is a vector of the same length as Q, and each element in the trace is associated with a state-action pair. For every non-visited state, the trace decays by a factor of λ at every time step

$$z_t(s, a) = \gamma\lambda z_{t-1}(S, A), \quad \forall (s, a) \in \mathcal{S}, \mathcal{A}, (s, A) \neq (S_t, A_t) \tag{4.8}$$

while the trace of the state-action pair at the current time step is incremented:

$$z_t(s, a) = \gamma\lambda z_{t-1}(S_t, A_t) + 1. \tag{4.9}$$

Thus, the eligibility trace can be thought of as an indicator of how recently a state was visited. λ controls how quickly the eligibility trace decays. The error for Sarsa(λ) the same as the Sarsa method:

$$\delta_t = R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t). \tag{4.10}$$

The value of every state-action pair is then updated based on the current error multiplied by the eligibility trace associated with that state. Thus, states that were visited recently see larger updates.

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha \delta_t z_t(s, a), \quad \text{for all } s, a \tag{4.11}$$

4.2.4 Q-learning

The policies followed by agents are typically ϵ-greedy. This means that they select the optimal (greedy) action $1 - \epsilon$ of the time, and exploratory actions ϵ of the time. Although this is useful in that the agent explores the state space, potentially finding more valuable states and better actions, an on-policy update rule (such as Sarsa) does not allow the agent to learn about the optimal policy while it explores (since explorations are usually sub-optimal actions). Q-learning is an off-policy update rule that allows the agent to learn the optimal policy while exploring. It does so by setting the target equal to the reward plus discounted value of the greedy action at the next state, rather than the actual action selected:

$$target = R_{t+1} + \gamma \max_a Q(S', a). \tag{4.12}$$

The update rule at every time step is then:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S', a) - Q(S_t, A_t)]. \tag{4.13}$$

This usually results in earlier convergence than the Sarsa method. Having introduced several learning algorithms, we move on to describing the method for approximating the state space.

4.3 Approximate solution methods

Earlier, the action-value function q_π was introduced as a table containing the value of all state-action pairs. However, recall from Chapter 2, Tetris's state space is too large to be represented in tabular form. Therefore, q_π is approximated using a functional form with a weight vector $\mathbf{w} \in \mathbb{R}^d$:

$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a) \tag{4.14}$$

Two common choices for the function approximator are linear regression algorithms or neural networks. As stated in the introduction, linear function approximation is the the chosen method for this project because it normally leads to shorter training times.

4.3.1 Linear function approximation

Linear function approximation prescribes methods for representing the state, calculating the approximate action-value $\hat{q}_\pi$, and updating the weight vector to make $\hat{q}_\pi$ more accurate.

Feature representation

In linear function approximation, the state is represented by a feature vector rather than a table:

$$\mathbf{x}(S, A) = \begin{pmatrix} x_1(S, A) \\ \vdots \\ x_n(S, A) \end{pmatrix} \tag{4.15}$$

The feature vector must contain sufficient information for the agent to choose actions in its environment. Furthermore, it is a sample of the state - the number of elements in the feature vector is less than or equal to the number of elements in the state. This sampling causes information to be lost through feature encoding. Therefore, the features are chosen to be as compact as possible while minimising information loss. This is best explained by means of an example. In Tetris, an accurate representation of a column is a binary vector, where 1s and 0s correspond to occupied and unoccupied cells, respectively. The number of permutations of this vector is 1048576. Although this is large, it contains all the information about the column - the height, the number and position of holes are represented. A feature that some Tetris researchers consider is the column height: this means that only the highest occupied cell in the column is represented. The number of permutations of the column height feature is 20, resulting in a feature vector that is 52428.8 times smaller than the binary feature. The small feature vector does result in information loss: the column height feature does not contain any information about the holes. However, it is possible that the agent only needs the column heights to train effectively. Thus, features are selected to retain only useful information. This is a trail and error process. Note that this discussion was simplified to one column: the complete feature vector for Tetris contains the tetromino information and cell configurations for the entire playing field. Feature vectors used by previous Tetris researches have been summarised in Section 2.4.

Calculating approximate action-values

The approximate action-value is calculated as the dot product between the feature and the weight vector

$$\hat{q}(S, A, \mathbf{w}) = \begin{pmatrix} 1(S = s_1, A = a_1) \\ \vdots \\ 1(S = s_n, A = a_n) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_n \end{pmatrix} \tag{4.16}$$

Each element in the weight vector corresponds to an element in the feature vector. Note that the feature vector is 1-hot encoded: all entries are zero except for the active state-action pair. This means that the dot product in Equation 4.16 equates to the weight corresponding to the active state.

Updating the weights

Linear regression is used to update the weight values so that $\hat{q}_\pi$ more accurately represents q_π. The method to find the weight vector update is to differentiate the mean square error between the true value q_π and the approximate value $\hat{q}_\pi$

$$\begin{aligned}\Delta\mathbf{w} &= -\frac{1}{2}\alpha\nabla_{\mathbf{w}}\mathbb{E}_\pi[(q_\pi(S,A) - \hat{q}(S,A,\mathbf{w}))^2] \\ &= \alpha\mathbb{E}_\pi[(q_\pi(S,A) - \hat{q}(S,A,\mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S,A,\mathbf{w})].\end{aligned} \tag{4.17}$$

The expectation is removed by sampling from experience

$$\Delta\mathbf{w} = \alpha(q_\pi(S,A) - \hat{q}(S,A,\mathbf{w}))\mathbf{x}(S,A). \tag{4.18}$$

Of course, the true value of q_π is not known. It is therefore substituted with a *target* to form the update rule:

$$\mathbf{w} = \mathbf{w} + \alpha(target - \hat{q}(S,A,\mathbf{w}))\mathbf{x}(S,A). \tag{4.19}$$

This equation updates only the weight corresponding to the active state. It is analogous to Equation 4.1 for the tabular case. The target is set by the learning method: Sarsa, n-step Sarsa, Sarsa(λ) and Q-learning. Using these algorithms in combination with linear function approximation allows them to be applied to much larger state spaces than with the tabular case.

4.3.2 Linear function approximation applied to learning methods

The algorithms discussed in Section 4.2 can be a adapted to use linear function approximation by substituting their targets into Equation 4.19. Then, rather than a state-value update, a weight update takes place.

Sarsa

The weight update for Sarsa is analogous to Equation 4.4:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma\hat{q}(S',A',\mathbf{w}) - \hat{q}(S,A,\mathbf{w})]\mathbf{x}(S,A). \tag{4.20}$$

n-step Sarsa

The return for n-step Sarsa remains unchanged:

$$G_{t:t+n} = R_{t+1} + ... + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}_{t+n-1}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad \text{if } n \geq 1, 0 \leq t < T - n \tag{4.21}$$

$$G_{t:t+n} = G_t \text{ if } t + n \geq T. \tag{4.22}$$

These Equations are analogous to the n-step returns in Equations 4.21 and 4.22. The weight update from Equation 4.7 becomes

$$\mathbf{w}_{t+n} = \mathbf{w}_{t+n-1} + \alpha[G_{t:t+n} - \hat{q}_{t+n-1}(S_t, A_t, \mathbf{w}_{t+n-1})]\mathbf{x}(S, A), \quad 0 \leq t < T. \tag{4.23}$$

Sarsa(λ)

The eligibility traces in the Sarsa(λ) algorithm are unaffected by linear function approximation. The error δ from Equation 4.10 becomes

$$\delta_t = R_{t+1} + \gamma \hat{q}_t(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t). \tag{4.24}$$

The update rule from Equation 4.11 stays the same, except weights are substituted for action-values

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t. \tag{4.25}$$

Q-learning

The weight update for Q-learning is analogous to Equation 4.13:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma \max_a \hat{q}(S', a, \mathbf{w}) - Q(S_t, A_t, \mathbf{w})]\mathbf{x}(S, A). \tag{4.26}$$

To summarise, reinforcement learning solves the Bellman Optimality Equation by iteratively evaluating and generating new policies. Several methods for doing this were discussed, and temporal difference learning was chosen as the method this project will implement. It was chosen over dynamic programming due to computational constraints, and over Monte Carlo because it converges faster. The temporal difference techniques each have a slightly different policy evaluation step. The techniques considered in this project are Sarsa, n-step Sarsa, Sarsa(λ) and Q-learning. Linear function function approximation was then introduced as a technique to approximate the action-values using a weight vector, as the actual number of action-values in Tetris are far too large to be represented in table form. Pseudocode for each algorithm with linear function approximation is provided in Appendix A. The next Chapter will explain how the theory was applied to solve Tetris.

Chapter 5

Applying reinforcement learning to Tetris

In this chapter, we address how the markov decision process and reinforcement learning theory presented in Chapter 3 and 4 is applied to train agents to play Tetris. The implementations for the training and testing loops, Tetris environment, and the agent are discussed. These programs were written in python using the numpy library.

5.1 Training and testing loops

The agent and environment interaction, represented by Figure 3.1, occurs within the training loop. Additional to the interaction shown in this Figure, the training loop performs the update rule (Sarsa, n-Step Sarsa, Sarsa(λ), or Q-learning). Separate training loops were developed for each of the update rules, as they require slightly different implementations. Psuedocode for each of these algorithms is presented in the appendix A. However, the general program flow, shown in Figure 5.1 is the same for all of them. This figure shows that for every episode, the training loop starts a new game. Then, at every time step, the agent and Tetris simulation (the environment) interact. The agent selects an action, which the Tetris simulation executes, outputting a next state and reward. The weight vector is then updated using the update rules specified in the previous chapter. There is then a check to see if the game should be ended. If the game is not over, the agent-environment action continues. There are two mechanisms by which the game can end: either the agent loses, or it reaches a limit set on the number of lines it is allowed to clear. This limit was placed on the agent to produce acceptable learning curves and prevent runaway training times. If the game should end, the training loop records the number of lines cleared and duration of the episode. The training loop then starts a new episode by initialising a new game, while keeping the weight vector unchanged. This process continues until the specified episode limit is reached. Once the agent is trained, it can be tested using the testing

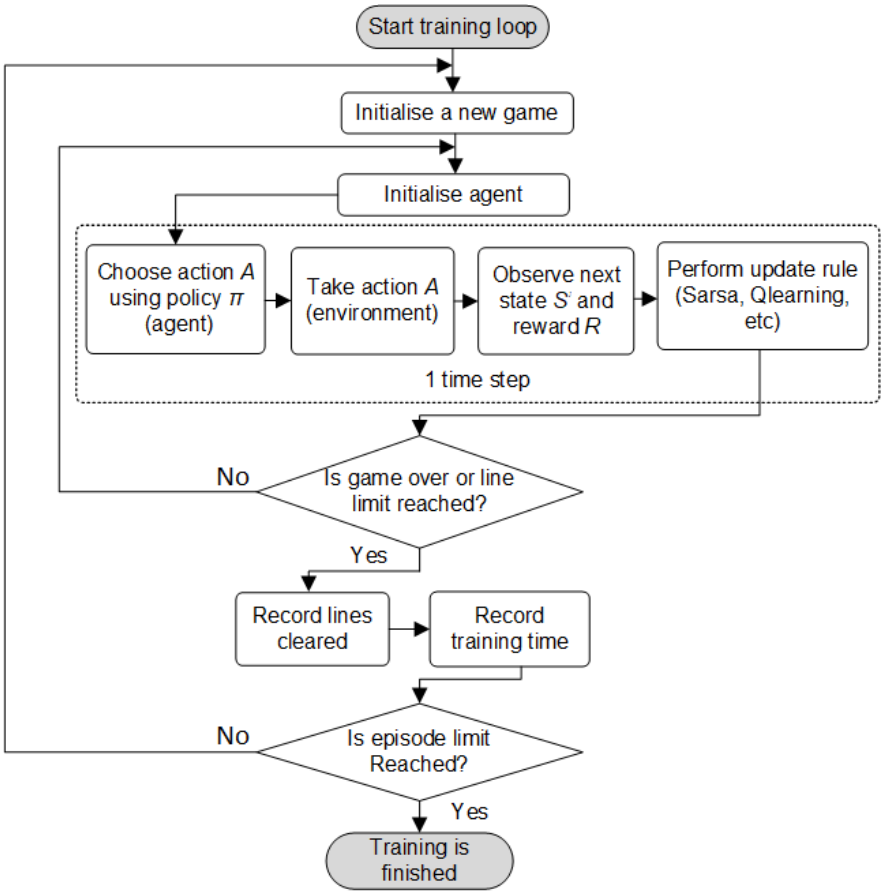


Figure 5.1: Flow diagram for training loop

loop. This loop is equivalent to the training loop in Figure 5.1, except that the update rule is excluded and the lines cleared limit is removed.

5.2 Tetris simulation

The Tetris simulation controls the playing field and tetromino during 1 time step of the training loop. It takes an action as input, executes the action by making changes to the playing field, and outputs the next state and reward. We start the discussion by explaining how the playing field and tetromino were represented in memory.

5.2.1 Playing field and tetromino implementation

The playing field is represented by a matrix of binary values that indicate whether a cell is occupied, as shown in Figure 5.2.

The tetrominos are represented by a separate matrix, shown in Figure 5.3. This matrix can be thought of as a second layer that slides on top of the

playing field matrix, as shown by the green box in Figure 5.2. Therefore, the playing field cells that the tetromino occupy are not encoded as ones on the playing field matrix. This is necessary for the tetrominos to be encoded into the state separately from the playing field. The top left corner of this matrix specifies the position of the tetromino on the playing field. The cells occupied by the tetromino in the tetromino matrix are specified by a tuple, such as $i = \langle(0,0),(1,0)\rangle$. Every tetromino type and rotation has its own unique tuple and identifier. Thus, the tetromino is specified by four values: The type, rotation, x and y coordinates.

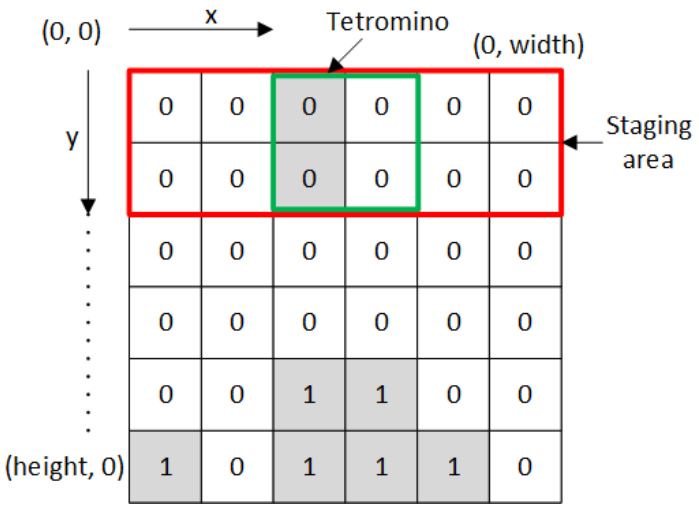


Figure 5.2: The playing field representation in memory. 1 indicates an occupied cell, whereas 0 indicates an unoccupied cell.

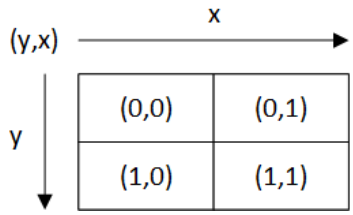


Figure 5.3: Tetromino representation in code. (y,x) specifies the position of the tetromino on the playing field matrix. The tetrominos are encoded as tuples of the coordinates shown in the grid. These coordinates are relative to (y,x).

5.2.2 State and action spaces

The agent is only allowed to select actions that move the tetromino. The action set available to the agent is translate left, translate right, rotate clockwise and

drop. This matches the action set of standard Tetris, bar the soft drop action.

In Tetris, the state specified by the playing field matrix, tetromino type, rotation, x and y position. The Tetris simulation stores these values only for the current state. From Chapter 3, recall that the action-values correspond to unique state-action pairs. This requires that each permutation of the state be represented in memory. However, there is an issue with this: the size (number of permutations) of the state is the product of the number of permutations of the playing field; number of tetromino types; rotations; possible and y positions. This equates to $6.106 \cdot 10^{63}$, which is far too large to be represented in memory. Another issue associated with large state spaces is long training times: since the agent must visit a state multiple times for it to develop a good policy for that state, it takes longer to develop policies for every state in a large state space. Therefore, several simplifications are made to the Tetris simulation to reduce the state space size to make training an agent feasible.

Narrowing the playing field

The size of the state space associated with the playing field increases exponentially with the width of the playing field. Therefore, the playing field was narrowed to six columns to significantly reduce the size the size of the state space. Six was chosen to match the width of Melax's Tetris. Chapter 7 addresses removing this constraint.

Simplified tetrominos

Early results showed that agents playing with large tetrominos performed poorly and took long to train. This is because there are very few good tetromino placements relative to the state space size. A subset of Melax's simplified tetrominos, consisting of the o, O, i and l shapes from Figure 2.2 were used. The s shape was excluded because it increased training time significantly. Reducing the number of tetrominos not only made training easier, it also reduced the state space size. Chapter 7 addresses removing this constraint.

Tetromino staging area

Instead of allowing the tetrominos to fall gradually, the staging area setup considered by Tsitsiklis and Van Roy [9] was used. It is shown in Figure by the red box in Figure 5.2. The staging area is two rows high because Melax's tetrominos are at most two cells in height and length. This setup is equivalent to the original game under the condition that the agent (had it been playing the original game) adopted a policy to position and drop the block before it moves downwards once. In fact, this policy is encouraged by penalising the agent at every time step. Using the staging area has the advantage of making the vertical position of the tetromino irrelevant, meaning we can exclude it from the state space.

5.2.3 Feature vectors

The state space size with the simplifications mentioned above in place is $(2^6)^{20} \cdot 8 \cdot 6 = 6.38 \cdot 10^{37}$. This is still too large to be represented in memory. Therefore, linear function approximation is used to represent the state space as a compact feature vector, $\mathbf{x}(S, A)$. The feature vector contains the action, playing field representation, rotation, and x position. We focused our attention on the representing the playing field as a set of features, since it is the largest contributor to the state space size. Our playing field feature sets, summarised in Table 5.1, are based on those on those in Table 2.3 from the literature review. The features were clipped to achieve manageable feature vector sizes.

The first approach was to sample a portion of the playing field matrix. Two rows, corresponding to the maximum column height and the row beneath it were considered. Two rows were chosen because Melax's tetrominos are at most two rows tall. The choice to position these two rows at the top of the playing field was to ensure the section of the playing field that the agent can see is always accessible.

The next approach was to use the column heights. This approach yielded a more compact feature vector than using the unprocessed playing field, allowing four rows to be considered.

A feature indicating whether a column has holes in it was then used in the feature vector. Holes were by far the most common feature considered by researchers in the literature. However, they contain no information about the contours or occupied cells in the playing field. Therefore, the holes feature was combined with column heights. The combination of holes and column heights produced an extremely large feature vector. As such, the column heights were clipped at two rows to keep the feature vector a manageable size.

The final feature considered was height differences between columns, truncated to ± 2. The feature vector produced was the most compact, owing to the fact that there is one less height difference than the number of columns.

Feature	**Playing field size formula**	**parameter value**	**Total feature vector size**
n row matrix	$(2^n)^{width}$	$n = 2$	786432
Column height clipped at n rows	$(n+1)^{width}$	$n = 4$	3000000
Column height clipped at n rows and holes	$(n+1)^{width} \cdot 2^{width}$	$n = 2$	8957952
Height differences between columns, clipped at -n and n rows	$(2n+1)^{width-1}$	$n = 2$	600000

Table 5.1: A summary of playing field feature representations

5.2.4 Reward function

The reward function in the original Tetris is based only on the number of lines cleared in a time step, shown in Table 2.1. This produces sparse reward, resulting in slow training times. This is because clearing a line involves some strategy, so the agent must take a large number of random actions before it clears a line by accident and receives any reward at all. Therefore, a continuous reward function was considered. This reward function compares selected reward features before and after taking a move:

$$\begin{aligned}\text{reward} = c_1 \cdot (\text{lines cleared}) + c_2 \cdot (\text{time step penalty}) + \cdot c_3 \\ \cdot (\text{game over penalty}) + c_4 \cdot \Delta(\text{maximum column height}) \\ + c_5 \cdot \Delta(\text{total holes}) + c_6 \cdot \Delta(\text{unevenness})\end{aligned} \tag{5.1}$$

where the lines cleared reward matches the values from Table 2.1; unevenness is defined as the squared height differences between columns; the time step penalty is given on every turn that the agent does not clear a line; and the game over penalty is zero on all turns except when the agent loses, in which case it is one. The game ends when any column enters into the staging area. The constants associated with each feature are determined experimentally in Chapter 6 to minimise training time while maximising agent performance.

5.2.5 Tetris simulation logic

Now that motivatons for state space representations and reward function has been given, we describe the logic for the Tetris simulation. The flow diagram is shown in Figure 5.4. The simulation receives the agents chosen action. If the action is anything besides drop, the tetromino remains in the staging area, and a check is performed to see if the move is valid (it does not move the tetromino past the boundaries of playing field). If the action is valid, the tetromino is shifted. A time step penalty is given to the agent regardless of whether the action was valid. If the action was to drop the tetromino, the tetromino is moved downwards until it collides with the bottom of the playing field or an occupied cell. The playing field cells that the tetromino occupy are then set equal to 1. Full lines are cleared from the playing field by moving the rows above the full line downwards. The features are calculated from the state after the lines are cleared so that they reflect the cleared line on the current time step. A check is then performed to see if the agent has lost the game. If so, a flag is set. The training loop will then end the game and initialise a new one. After the game over check is performed, the reward is given.

5.3 Agent implementation

The agent contains the feature and weight vectors, mechanisms for selecting an action, and functions for evaluating the approximate action-value. Note

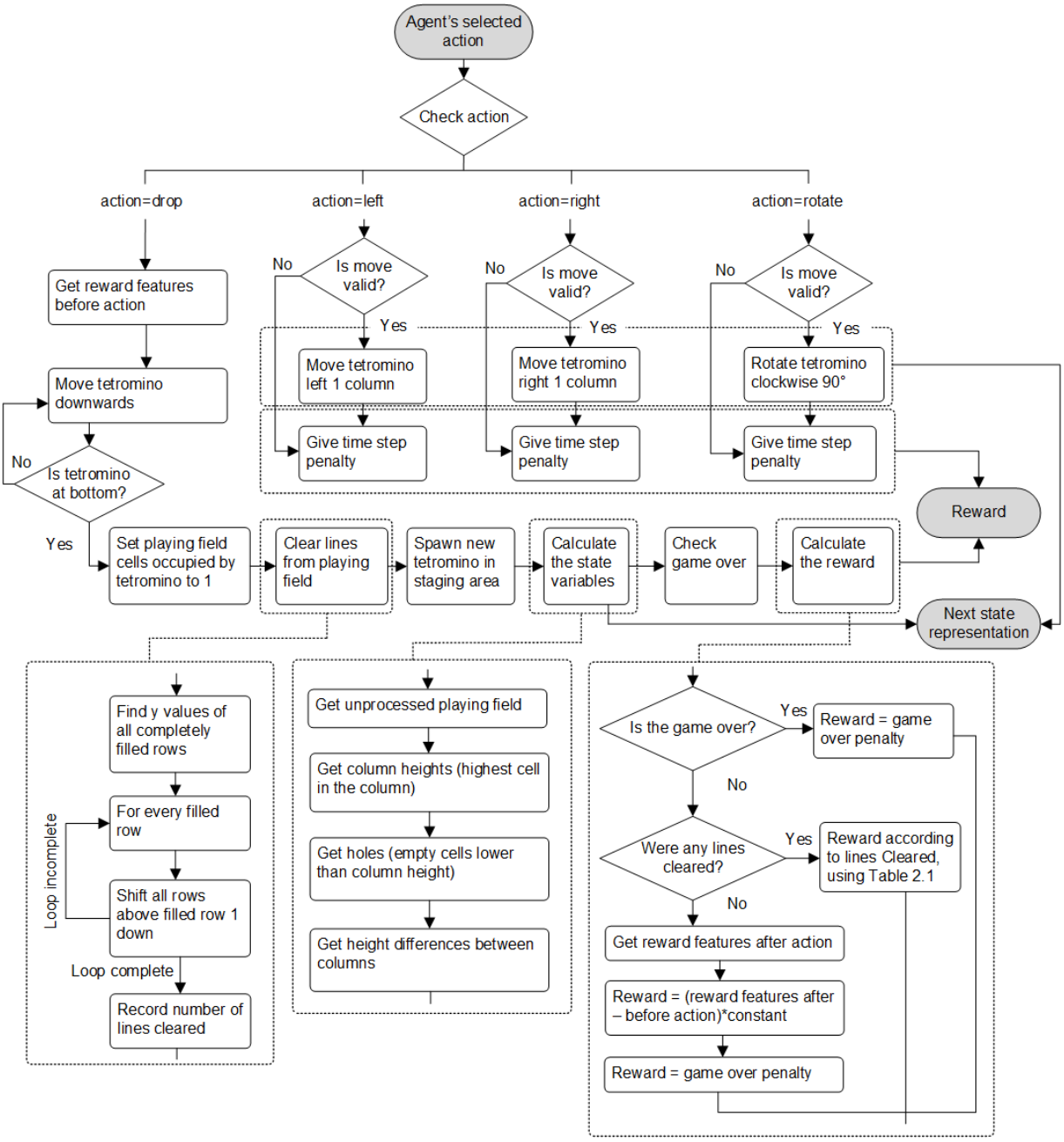


Figure 5.4: Flow diagram for Tetris simulation

that the feature vector was discussed in Section 5.2.3.

5.3.1 Action selection

The task of the agent is to select actions based on an ϵ-greedy policy. This means that it selects greedy and exploratory actions at a ratio of ϵ to $1 - \epsilon$. The exploratory actions are random, while the greedy actions are selected by evaluating the values of the state-action pair for every available action, then selecting the action with the highest value. This process is shown by the flow chart in Figure 5.5. The next section deals with calculating the state-action pair values.

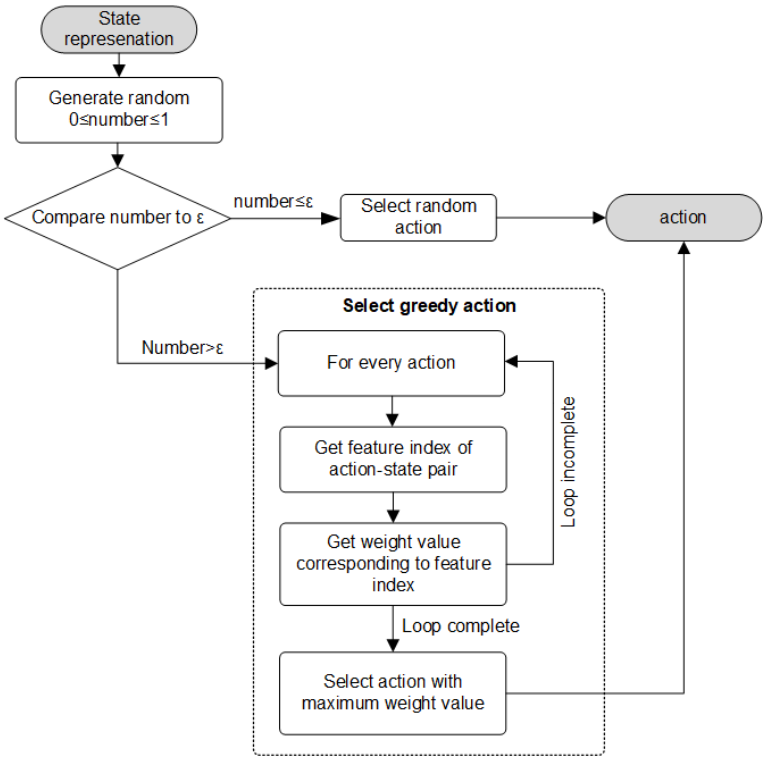


Figure 5.5: Flow diagram for agent

5.3.2 Finding the approximate action-values

Recall from Section 4.3.1, in linear function approximation, the value of a state-action pair is calculated by the dot product of the weight vector and feature vector:

$$\hat{q}(S, A, \mathbf{w}) = \begin{pmatrix} 1(S = s_1, A = a_1) \\ \vdots \\ 1(S = s_n, A = a_n) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_n \end{pmatrix} \tag{5.2}$$

Since the feature vector is 1-hot encoded, the action-value is the value of the weight vector element corresponding to the active state-action pair. This operation is equivalent to indexing into the weight vector corresponding to the active state-action pair using an integer. It was found that indexing with an integer was more computationally efficient than using a dot product because it avoids large vector computations on every time step. Therefore, the indexing method was used. This significantly improved training times for Sarsa, n-step Sarsa and Q-learning. The method was also used to find the error for Sarsa(λ). However, large vector computations could not be avoided for this method because, by its definition, all weights are updated at every time step.

In summary, this chapter has presented the methods for applying the reinforcement learning theory discussed in the previous chapters to Tetris. It was clear that training on the standard version of Tetris was not possible due to its large state space. Therefore, the game was simplified and function approximation used to represent the state space. In the next chapter, we will present the training and testing results of agents that were trained on the simplified version of Tetris.

Chapter 6

Experiments and results

The previous chapter described the setup that was used to apply reinforcement learning to Tetris. Several simplifications were made to the game to allow this to happen. Using this setup, experiments were performed to determine the optimal reward function, playing field feature representation, learning method and hyperparameters. These parameters were optimised for both training time and performance while testing. The results are used as the basis for the solution in Chapter 7. Note that all the learning curves in this chapter show the average lines cleared by 50 agents trained with all parameters set to the values in table 6.1, except for the one being varied. Furthermore, each graph shows a sample of 100 evenly spaced episodes from the total number of episodes in the training run. The learning curves were plotted with the python library Matplotlib.

During testing, the performance of the agents were measured as the average number of lines cleared over 100 games with the lines cleared limit removed. Averages are given to conform to the reporting style in literature.

Parameter	**Selected value**
Learning method	Sarsa
Feature vector	Height differences between columns, truncated to ± 2
γ	0.9
α	0.1
ϵ	0.01
initial weights	0

Table 6.1: Standard parameters for all training runs presented in this chapter

6.1 Reward Function

The reward function was described in Equation 5.1. Each component of this reward function was tested individually, starting with the reward for clearing a line alone. The reward feature game over penalty was then tested by setting all other rewards to zero, apart from the reward for clearing a line. The coefficient of the game over penalty was then varied over several training runs. The column height, total holes and uneveness were tested likewise. The learning curves for all the coefficients tested for every reward feature are shown in Appendix B. The learning curve in Figure 6.1 show the coefficient for each reward feature that resulted in the fastest convergence.

The reward signal using the scoring mechanism from the game converged at around 15000 episodes. Each reward feature worked to some degree. A game over penalty improved the agent's convergence, confirming Steven and Pradhan's [12] results. Penalising the agent for creating holes improved the training time but the average score did not converge to 100 lines cleared, despite the fact that it produces desirable behaviour (there cannot be any holes in a line for it to be cleared). Penalising the agent for increasing the unevenness of the board causes the agent to learn the fastest, but the score also does not converge to 100 lines cleared. This reward feature may create undesirable behaviour, as the unevenness of the playing field can be decreased at the expense of creating a hole. The best reward feature in terms of increased performance and training time was penalising an increase in maximum column height.

The optimal values for each individual reward feature were then used to-

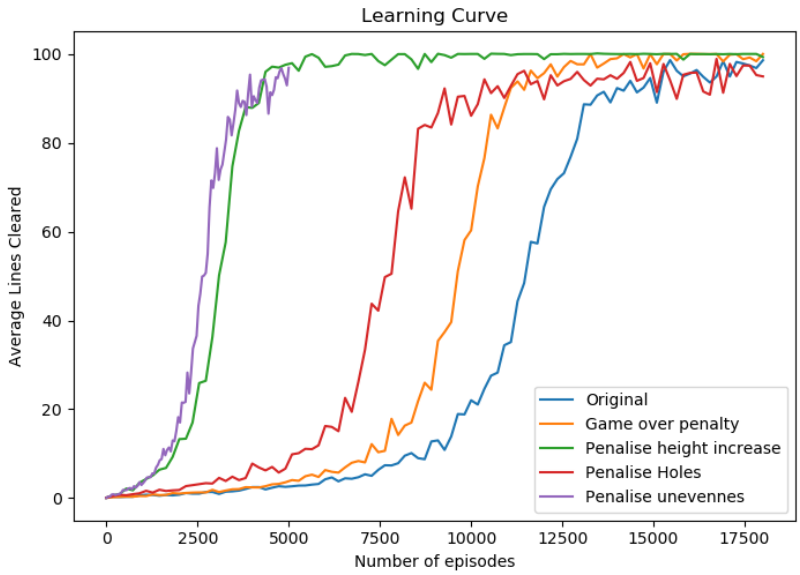


Figure 6.1: Learning curves for optimal values of individual reward features

gether in a single reward function. Although this resulted in an acceptable learning curve, the agent took long to train (1638 seconds). This was explained when observing the agents behaviours when testing: it repeatedly took reversing moves. This meant that the learned optimal action in certain states was never to drop the tetromino. The behaviour and training time issues were fixed by adding a small penalty at every time step and multiplying the reward for clearing a line by a constant. The training time after the reward for clearing a line was increased was 617 seconds. The learning curves for the multiplied reward signals are shown in Figure 6.2. The green line, with the reward for clearing a line set to the games score multiplied by 10, is the final reward function. Values for the final reward signal are presented in Table 6.2.

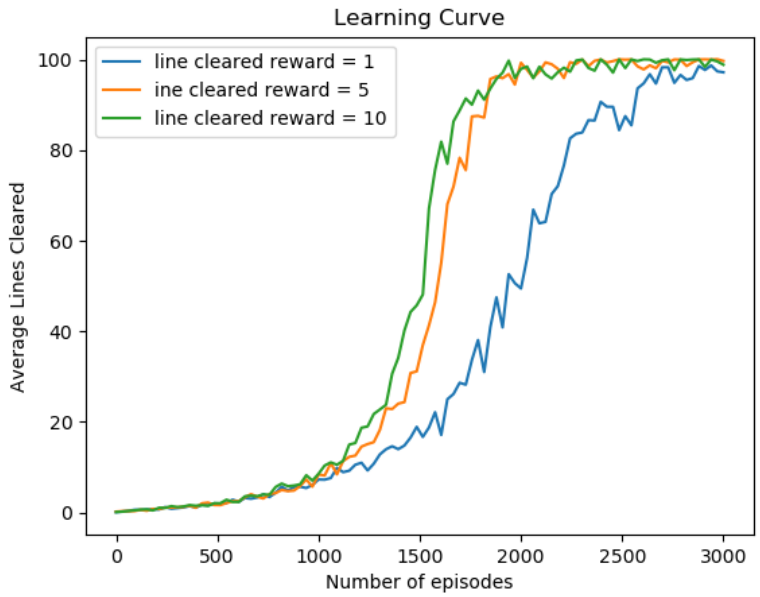


Figure 6.2: Learning curves for varying the reward for clearing a line

Reward feature	**Coefficient value**
Reward for clearing a line	10
Time step penalty	0.1
Game over penalty	2
Height increase penalty	1
Create holes penalty	2
Increase unevenness penalty	0.1

Table 6.2: Final reward signal coefficient values

6.2 Playing field feature representation

The learning curves for each of the playing field feature representations introduced in Table 5.1 is shown in Figure 6.3. Table 6.3 shows the performance of each feature in testing.

The first feature vector in Table 5.1, consisting of the row corresponding to the highest occupied cell and the row beneath it, was a naive choice. This was because sections of the playing field were not be visible to the agent when the maximum column height was more than two rows higher than the rest of the playing field. When this happens, the tetromino falls through the visible rows. The result is that the agent places the tetrominos badly, creating holes. However, the learning curve shows that feature did achieve some lines cleared.

The column height feature introduced a limitation on the agent: The visible section of the playing field becomes inaccessible if the column heights are greater than four. The performance of the column height feature is substandard. Adding holes to the column height feature significantly increased the size of the feature vector without improving the performance of the column height feature considerably.

The height difference feature outperformed the other features by a large margin in testing, and its training time was satisfactory. Aside from producing the most compact feature vector, the height differences are not limited to any specific portion of the playing field. Instead, all height differences less than ± 2 are accurately represented.

Feature	**Average lines cleared while testing**
n row matrix	80.90
Column height clipped at n rows	0.73
Column height clipped at n rows and holes	3.22
Height differences between columns, clipped at -n and n rows	726.78

Table 6.3: Playing field feature representation performances in testing

6.3 Learning algorithms

This section addresses the performance of each of the algorithms presented in Chapter 4. Learning curves for each algorithm are shown in Figure 6.4, and the performances of each algorithm while testing are presented in Table 6.4.

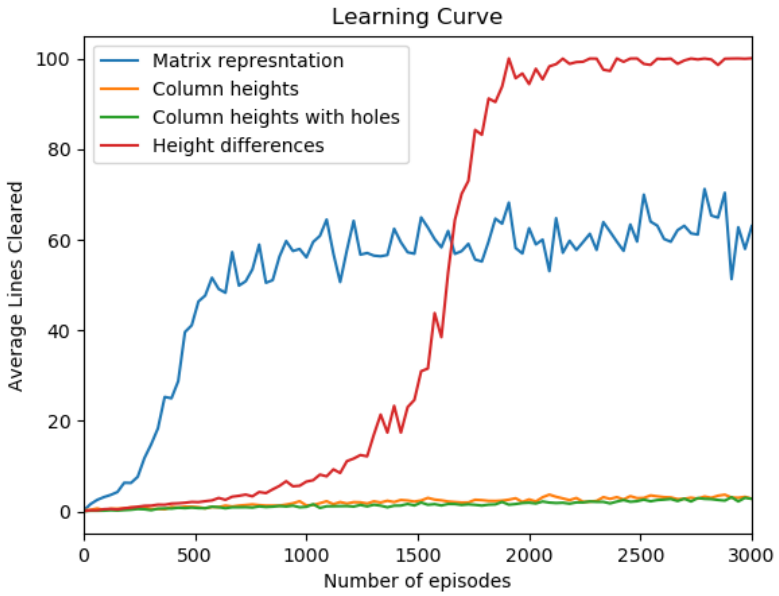


Figure 6.3: Learning curves for different feature vectors

Learning method	**Average lines cleared while testing**	**Time per training run (s)**
Sarsa	3487	765
n-step Sarsa, $n = 3$	1019	615
Sarsa(λ)	–	–
Q-learning	1045	1190

Table 6.4: Learning algorithms performance in testing and training times

The learning curves for n-step Sarsa with varying n is presented in Appendix B. In summary, $n = 3$ was optimal, increasing n beyond three significantly decreased the performance of the agent.

The learning curve in Figure 6.4 shows that Sarsa trained the in the same time as 3-step Sarsa. However, after hyperparameter tuning (on both Sarsa and 3-step Sarsa), Sarsa outperformed 3-step Sarsa by a significant margin in testing. Results for and The tests for Sarsa(λ) were aborted after one day of training without producing a single agent. The reason for the long training time was identified as the multiplication of the eligibility trace: computing this trace at every time step is computationally expensive. Q-learning scored similarly to 3-step Sarsa but took significantly longer to train. Sarsa is therefore chosen as the preferred learning algorithm.

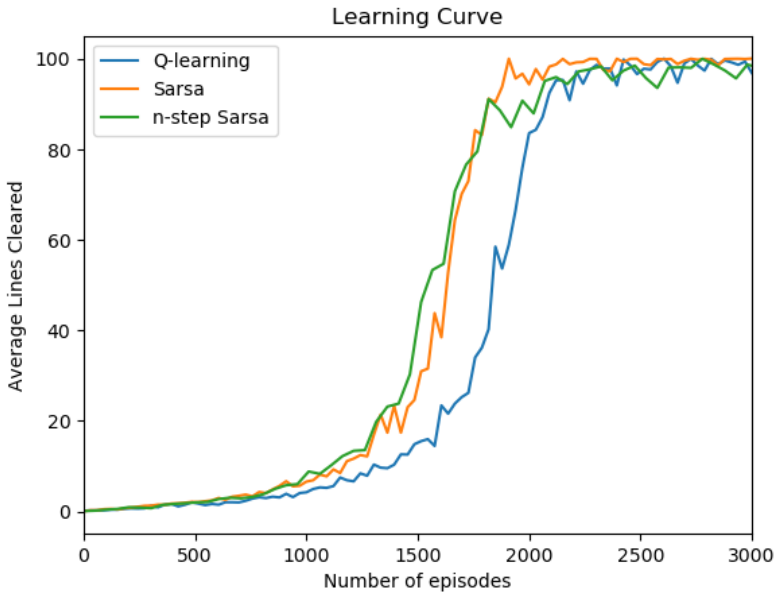


Figure 6.4: Learning curves for various learning algorithms, $n = 3$ for n-step Sarsa

6.4 Hyperparameters

The hyperparameters ϵ (exploration rate), α (step size), γ (reward discount factor), and the initial values of the weight vector were tested to find their optimal values for agent performance and training time. Each parameter was tested by varying its value while keeping the other hyperparameters set to the values from Table 6.1.

6.4.1 Exploration rate

Learning curves and testing results for different values of ϵ are shown in Figure 6.5 and Table 6.5 respectively. There was a strong correlation between the exploration rate and training time. Agents with low exploration rates took considerably longer (in seconds) to train than agents with higher exploration rates. This was explained by observing the agent behaviour while testing: the agent would often enter into a cycle of reversing moves (left then right, etc.) An agent would need to make an exploratory move to break the cycle of reversing moves. This means agents with low exploration rates will get stuck in cycles of reversing moves for longer, causing training time to increase. ϵ was chosen as 0.01 due to both training time and performance considerations.

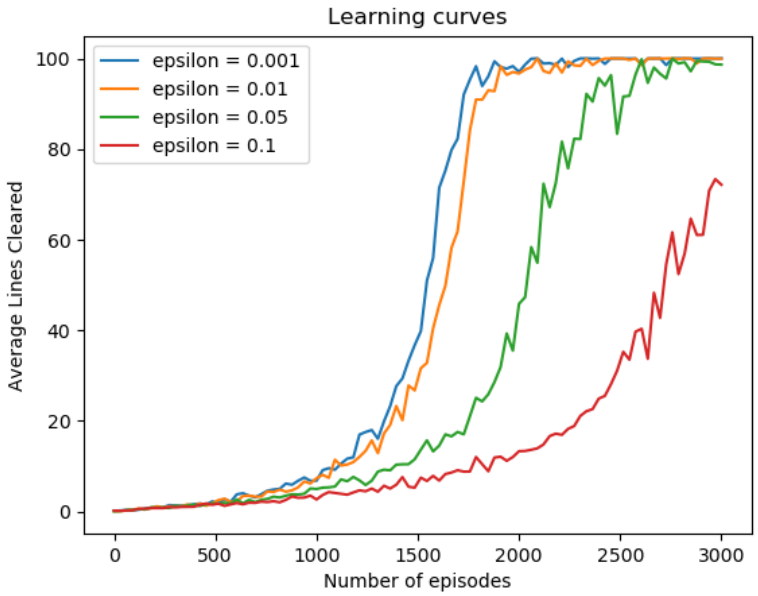


Figure 6.5: Learning curves for varying values of exploration rate

ϵ (exploration rate)	Average lines cleared while testing	Time per training run (s)
0.001	860	3600
0.01	3487	765
0.05	918	477
0.1	66	270

Table 6.5: Scores in testing and training time for different exploration rates

6.4.2 Step size

Learning curves for α are shown in Figure 6.6, and results while testing and training times in Table 6.6. Large step sizes resulted in both better training times and agent performances. However, setting the step size too high can cause learning to become unstable. α was chosen as 0.1 because it resulted in the highest score while testing, and because it resulted in stable learning.

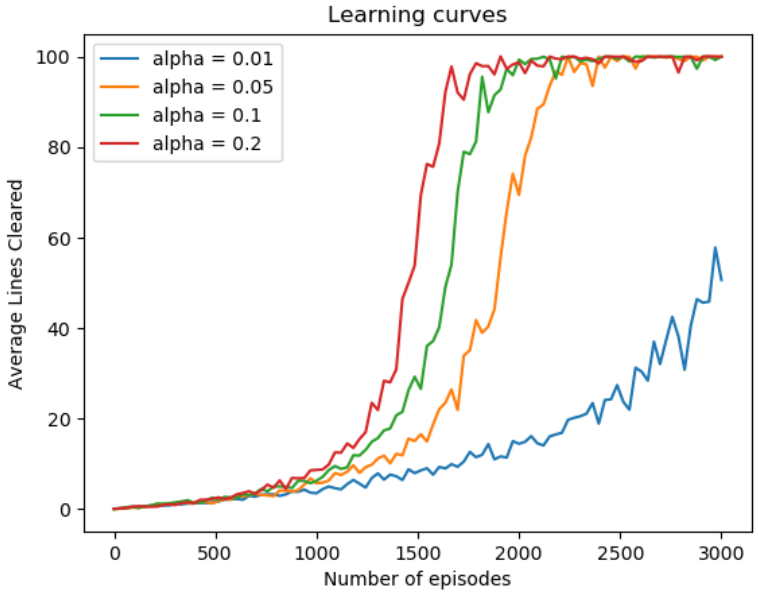


Figure 6.6: Learning curves for varying step size values

α **(learning rate)**	**Average lines cleared while testing**	**Time per training run (s)**
0.01	28	1245
0.05	874	957
0.1	3487	765
0.2	2918	604

Table 6.6: Scores in testing and training time for different step sizes

6.4.3 Reward discount rate

Figure 6.7 and Table 6.7 show the learning curves and performance in testing and training times for iterating γ. Although setting the discount rate low resulted in long training times, the score did not deviate far below the optimal value. Increasing the discount rate significantly shorten training times, but setting the discount rate too high impeded performance. The optimal discount rate was 0.9 for agent performance, resulting in an average training time.

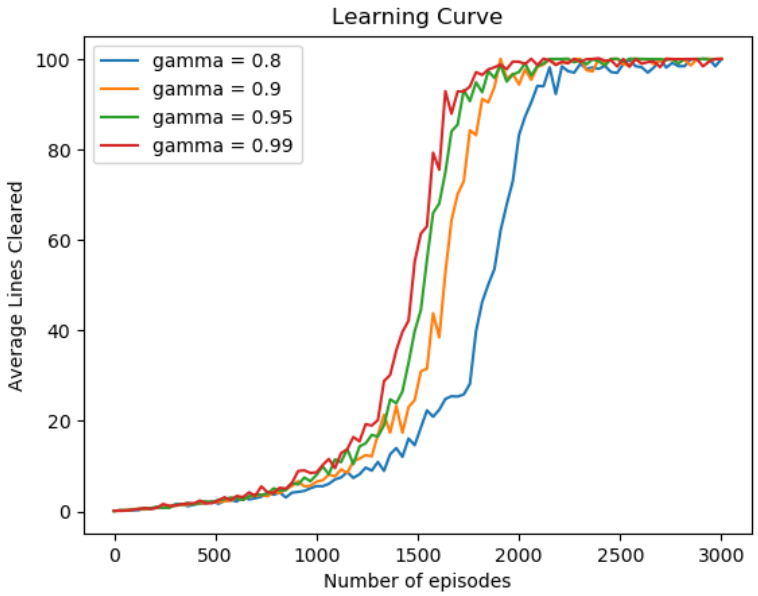


Figure 6.7: Learning curves for varying reward discount rate values

γ (Reward discount factor)	Average lines cleared while testing	Time per training run (s)
0.8	3037	1559
0.9	3487	765
0.95	3146	655
0.99	1351	656

Table 6.7: Scores in testing and training time for different reward discount rates

6.4.4 Initial weight vector

The learning curves for setting the initial weights to different values is shown in Figure 6.8, results for agent performances and training times are shown in Table 6.8. Setting the initial weights influences the behaviour of the agent significantly at the start of training. Exploration is encouraged by setting the weights slightly high, and discouraged by setting the weights slightly low. Changing the weights did not affect the number of episodes the agent converged in, but did influence the training time (in seconds) significantly. Setting the weights to anything but zero caused a significant increase in training time.

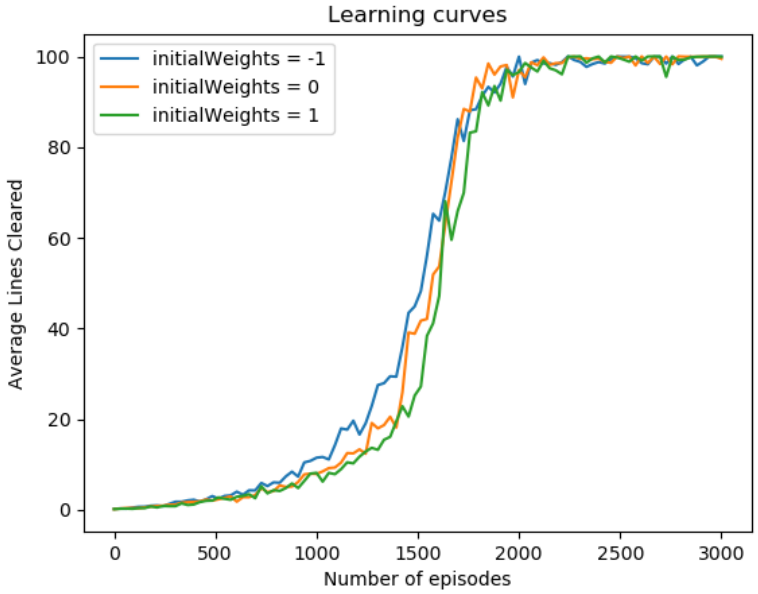


Figure 6.8: Learning curves for varying initial weight values

Initial Weights	Average lines cleared while testing	Time per training run (s)
-1	–	4273
0	3487	765
1	727	1741

Table 6.8: Scores in testing and training time for different initial weights. The test for initial weights=-1 were aborted due to excessively long training times.

6.5 Final agent

An optimal set of parameters for agent performance and training time was determined from the above experiments. The chosen learning method and feature vector was Sarsa and height differences between columns truncated to ± 2, respectively. The reward discount factor γ is chosen as 0.9 because it performed the best in testing. The step size parameter α is 0.1 due to performance and stability considerations. The learning rate ϵ is set to 0.01 because of performance and time considerations. Lastly, the weights were initialised to zero. Notice that the hyperparameter values are equal to those listed in Table 6.1. By varying the hyperparameters around these values, we have proven that the hyperparameters selected are at or near a local optimal point for agent performance. The learning curve using these hyperparameters

is shown in Figure 6.9. The agent trained for 765 seconds, before scoring an average of 3487 lines cleared over 100 games in testing. The blue area in the figure shows the area shows the difference between the first and third quartiles of as a measure of score spread while training, while Figure 6.10 shows the distribution of scores while testing. This distribution is heavily skewed to the right - the score for most of the games were under 3000, but there are outlier games where the agent scores a large number of lines.

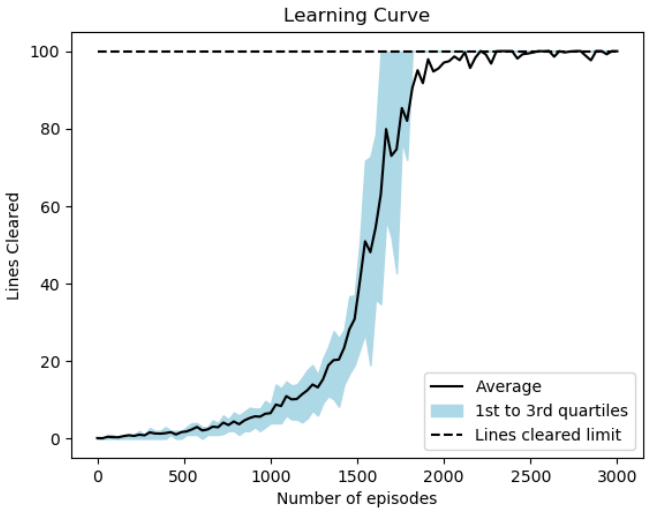


Figure 6.9: Learning curve for the final agent using optimal parameters

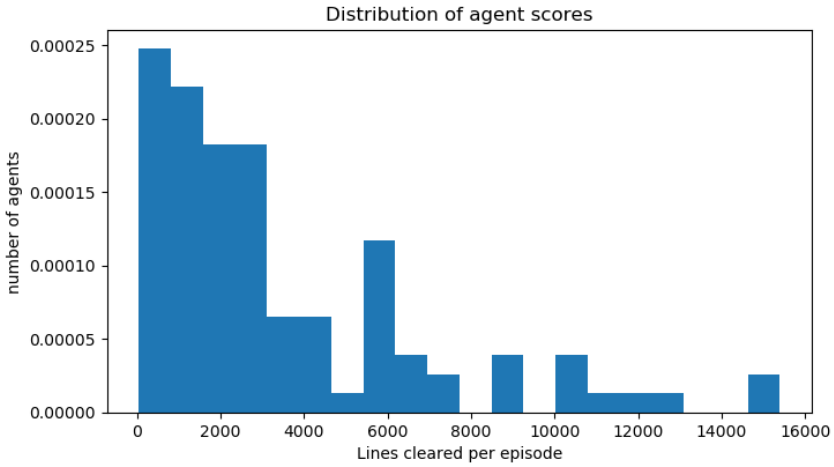


Figure 6.10: Density plot showing distribution of scores in testing (without lines cleared limit) for the final agent

In this chapter, the optimal reward signal, feature vector, learning methods and hyperparameters were found for the simplified version of Tetris described in Chapter 5. In doing so, the agent's training time to convergence was improved from 15000 episodes to less than 2500 episodes. This is valuable, as it is now possible to train agents to a high performance level quickly. The final agent scored an average of 3487 lines cleared per game after 12 minutes of training. This was considered good enough to justify attempts at removing the simplifications from the Tetris simulation. These attempts are discussed in the next chapter.

Chapter 7

Final agent implementation

In Section 1.2, the project aim was described as developing model-free reinforcement learning agents utilising linear function approximation methods for Tetris. In Chapters 5 and 6, it was shown that the feature and weight vectors increase exponentially with the state space. To reduce the size of the weight vector and make training with linear function approximation feasible, the Tetris simulation was simplified. The two most significant differences between the game considered in Chapters 5 and 6 and standard Tetris were:

1. Narrowing the playing field
2. Using Melax's simplified tetrominos

Performing tests with these simplifications in place was advantageous because it revealed which reward signals and features worked well for Tetris. With this knowledge in hand, attempts were made to remove these simplifications without redefining the project aim. This meant that more complex techniques such as model-based learning were considered outside the scope of this project. This Chapter summarises our solutions to widening the playing field and playing with standard Tetrominos. The two issues are solved individually.

7.1 Widening the playing field

For this section, Melax's full set of tetrominos were used (including the s shape). The s tetromino introduces significant difficulty, as there are often very few positions where it can be placed without creating a hole. With this set of tetrominos, and using column heights truncated at ± 2, the weight vector is 750000 elements large. Widening the playing field to 10 increases the weight vector to $781.25 \cdot 10^6$. This state space was too large to produce results in a reasonable time: an agent trained on the standard size playing field scored 0 lines cleared in testing after 35999 episodes and 1 hour of training.

A solution that that keeps the number of states small while playing on the 10 column wide field is to train the agent on the 6 wide playing field, then

let it play on the full size playing field by only allowing to see a 6 column wide portion of the large playing field at any given time. If the agent places the tetrominos perfectly in the visible portion of the playing field, then it should be able to clear lines. As part of the implementation, a rule-based decision maker is used to select the portion of the playing field that is visible to the reinforcement learning agent, and to move the tetromino into the visible portion so that the reinforcement learning agent can take control of it. The playing field is depicted in Figure 7.1

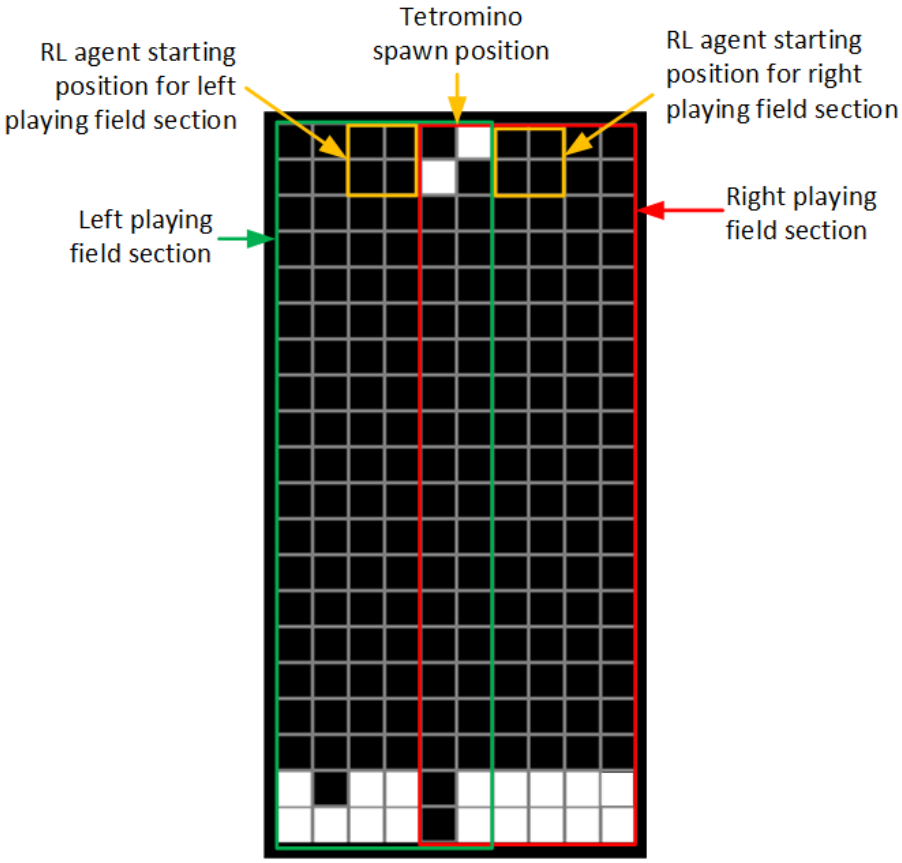


Figure 7.1: Wide playing field divided into two visible playing field sections each 6 wide, on either side of the playing field

The logic for the testing loop is shown in Figure 7.2. It is analogous to the training loop shown in Figure 5.1. At every time step, the rules based agent checks to see if a new tetromino has spawned on that turn. If one has spawned, then the playing field section with the lowest average height is selected as the visible portion of the playing field. This ensures the agent plays at the same height in all sections of the playing field. The rule-based then takes control of the tetromino, moving it towards the starting position of the visible playing field section. This is the default spawning position of the tetromino of the

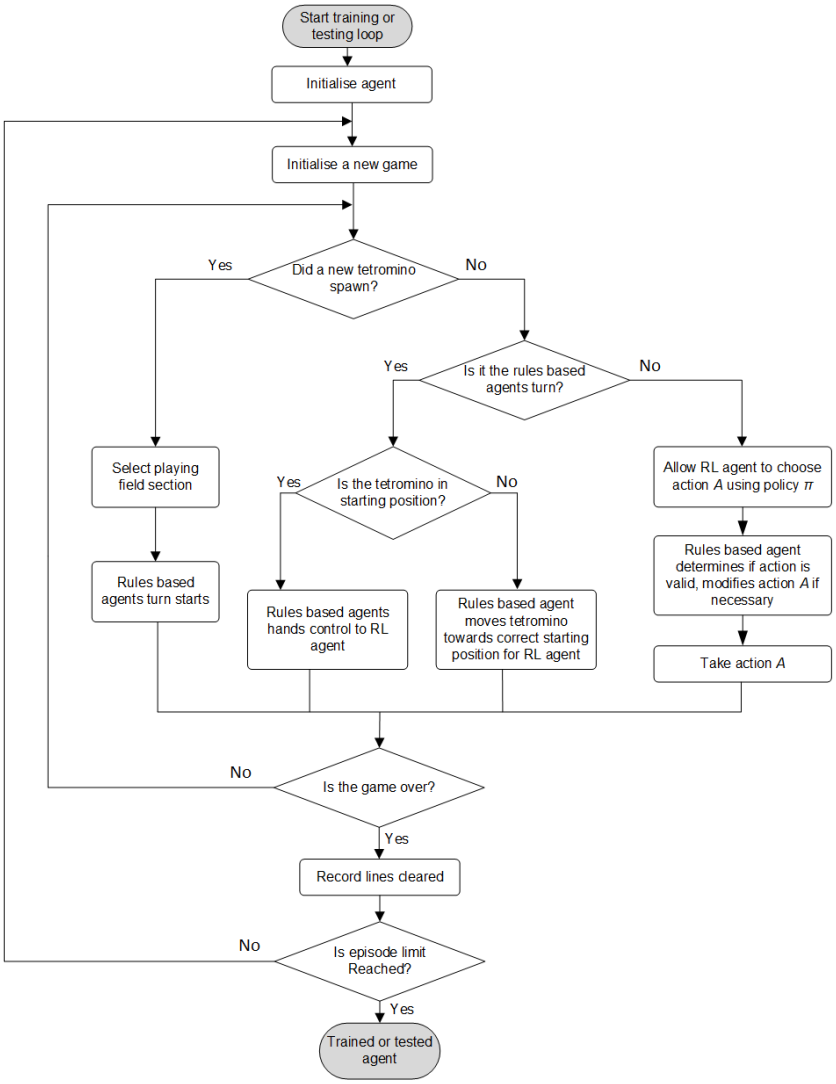


Figure 7.2: Flow diagram for solution to allow an agent trained on a narrow playing field to play on the standard size playing field

smaller playing field that the agent trained on. The rule-based decision maker hands control of the tetromino over to the reinforcement learning agent when the tetromino reaches the starting position. The reinforcement learning agent is then allowed to take actions to place and drop the tetromino in its section of the playing field. The rule-based decision maker checks each of these actions to ensure that the reinforcement learning does not move the tetromino beyond the visible section of the playing field (this would cause an error, since there are no weight indexes for states outside of the visible playing field). If this happens, the rules based agent forces the tetromino to drop.

Three playing field scenarios were considered when testing this solution:

1. Three visible playing field sections (left, middle and right), each four columns wide with one column of overlap between sections

2. Two visible playing field sections (left and right), each five columns wide and no overlap between sections

3. Two visible playing field sections (left and right), each six columns wide with two columns of overlap between sections (this setup is depicted in Figure 7.1)

For each of these scenarios, the agent was trained for one hour with the maximum lines cleared limit removed (recall that the lines cleared limit was in place to plot learning curves) on a playing field that matched the size of the visible playing field section on the large playing field. Results for each scenario are shown in Table 7.1.

Playing field section width	**Lines cleared on small playing field**	**Lines cleared on large playing field**
4	68166.44	9.52
5	4250.93	12.24
6	33736.56	83.44

Table 7.1: Scores for different agents trained on small playing fields, deployed on the standard playing field

The agents trained effectively on the small playing fields. However, the agent's performance were drastically reduced when inserted into the large playing field setup. This is because the definition of the Markov decision process has changed: it is now only partially observable, and the agents do not have access to all the relevant information to select the optimal action taking the entire playing field into account. Reconsider Figure 7.1, but assume the reinforcement learning agent is in control of the tetromino and the right side of the playing field is visible. If (as in training) the visible section was the entire playing field, the agent could drop the tetromino in its current position, clearing the second row. However, because second row is not complete on the left side of the playing field, the line does not clear. Instead, the move creates a hole in the first row. Thus, behaviour learned while training becomes ineffective when applied to the large playing field. Furthermore, the strategy learned by the smaller playing field sections are more incompatible with the large playing field.

However, despite not being trained on, and only being able to see a portion of the large playing field, the agent that was trained on the six column wide playing field was able to clear lines on the large playing field. This is a fair improvement over the score of zero from the agent that was trained on the large playing field. This is a novel finding - no researcher in the literature reviewed attempted to play Tetris by training the agent on a smaller playing field, or allowing the agent to work together with a rule-based decision maker.

7.2 Playing with standard Tetrominos

This section addresses issue of playing with the standard Tetris tetrominos, without widening the playing field. After the standard tetrominos were inserted into the game, the Tetris simulation was modified made so that the staging area height was 4 rows. Furthermore, the playing field representation was changed to height differences between columns truncated to ± 3 (to account for the larger tetrominos).

The reward function values from Table 6.2 were used to train the agent. This decision was made due to time considerations - reiterating the reward functions is a timely exercise because the agent takes much longer to train with the standard tetrominos than with Melax's tetrominos. The agent was trained for 1 million episodes, taking approximately 24 hours. The final performance in testing over 100 games was 31.28 lines cleared per game on average. This matches the performance of Tsitsiklis and Van Roy [9], except on a small playing field. This agent was placed into the setup described in Section 7.1, socring an average of 0.02 lines cleared.

This chapter presented solutions to remove two simplifications made to the Tetris simulation: Playing with Melax's simplified tetrominos and narrowing the playing field. The solution for widening the playing field was to train the agent on a narrow playing field, then transfer it onto the large playing field. The large playing field is represented to the agent by only making a portion of it visible. This visible portion is chosen by a rule-based agent. An agent trained on a 6 column wide playing field achieved an average of 83.44 lines cleared on the large playing field using this setup. An agent was then trained to play with the standard tetrominos on a 6 column wide playing field, achieving an average of 31.28 lines.

Chapter 8

Conclusion

Tetris has become popular as a test bed for algorithm development in the reinforcement learning community due to its ability to be simulated and accelerated. There have been many attempts at training agents to play Tetris, but the research focus has predominantly been on scoring as many points as possible. This has led to the development of algorithms with training times exceeding a month. Long training times such as these hinder algorithm development, and is the issue that this project set out to address. The research question asked was whether the use of less complex techniques than those presented in the literature could shorten training times while still resulting in an agent with an acceptable level of performance for Tetris. The method chosen for the task was model-free reinforcement learning with linear function approximation. The value of addressing this issue for Tetris is that the techniques developed for this project can be applied to real world problems in industry.

The purpose of the project has been achieved by training an agent to play on a standard Tetris playing field with a set of simplified tetrominos. The most significant challenge posed to the project was playing on the standard size playing field. The state space increased exponentially when increasing the width of the playing field, making training on the standard playing field unfeasible. The solution to this was to train the agent on a smaller playing field, then transfer the agent onto the standard size playing field. This was done by making a only a portion of the standard playing field visible to the agent. The agent played the game together with a rule-based decision maker which decided where the visible portion of the playing field would be. An agent that was trained on a six column wide playing field for one hour scored an average of 83.44 lines using this setup on the standard size playing field. Thus, an acceptable level of performance was achieved in a much shorter training time than those reported in literature.

Furthermore, an investigation into the optimal reward functions and feature vectors took place. Optimal reward function penalised the agent at every time step, for losing a game, increasing the maximum height, creating holes, and increasing the unevenness of the playing field, and only rewarding the agent

for clearing a line. The optimal playing field representation in the feature vector were the height differences between columns truncated to the tetromino height. These findings may be useful for future Tetris researchers looking to improve the performance of their agents.

The fact that an agent was trained on a small playing field and tested on a full size playing field means that reinforcement learning algorithms can be trained in simpler environments than what they are deployed in. This is an especially useful finding for applications where the state space cannot be reduced sufficiently to for a reinforcement learning agent to train, or where simulating agent experience is difficult: the solution may be to use engineering ingenuity rather than more computational power.

Our recommendations for future work on training reinforcement learning algorithms for Tetris are to investigate the use of neural networks as function approximators, or to use model-based techniques to play on the large playing field without having to limit the visibility of the agent.

Appendix A

Pseudocode for various learning methods

Algorithm parameters: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialise value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily
Loop for each episode:
    $S, A \leftarrow$ initial state and action of episode
    Loop for each step of the episode:
    | Take action A, observe R and S'
    | If S is terminal:
    | | $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R - \hat{q}(S, A, \mathbf{w})]\mathbf{x}(S, A)$ (update rule)
    | | Go to next episode
    | Choose A as a function of $\hat{q}(S', \cdot, \mathbf{w})$
    | $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})]\mathbf{x}(S, A)$ (update rule)
    | $S \leftarrow S'; A \leftarrow A'$
Until S is terminal

Figure A.1: Pseudocode for the Sarsa algorithm, adapted from Sutton and Barto's reinforcement learning textbook [18].

Algorithm parameters: Step size $\alpha \in (0, 1]$, small $\epsilon \in (0, 1]$, positive integer n
Initialise value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily

Loop for each episode:
    Initialise and store $S_0 \neq$ terminal
    Select and store an action $A_0 \sim \pi(\cdot|S_0)$
    $T \leftarrow \infty$
    Loop for $t = 0, 1, 2, 3...$:
        If $t < T$, then:
            Take action A_t
            Observe and store R_{t+1}, S_{t+1}
            If S_{t+1} is terminal, then:
                $T \leftarrow t + 1$
            else:
                Select and store $A_{t+1} \sim \pi(\cdot|S_{t+1})$
        $\tau \leftarrow t - n + 1$, τ is the state being updated
        If $\tau \geq 0$:
            $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
            If $\tau + n < T$, then:
                $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$ (get target)
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha[G - \hat{q}(S_\tau, A_\tau, \mathbf{w})]\mathbf{x}(S, A)$ (update rule)
    Until $\tau = T - 1$

Figure A.2: Pseudocode for the n-step Sarsa algorithm, adapted from Sutton and Barto's reinforcement learning textbook [18].

input: a function $\mathcal{F}(s, a)$ returning set of indices of active features for s, a
Algorithm parameters: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$, trace decay rate $\lambda \in [0, 1]$
Initialise $\mathbf{w} = (w_1, ..., w_d)^\intercal \in \mathbb{R}^d$, $\mathbf{z} = (z_1, ..., z_d)^\intercal \in \mathbb{R}^d$
Repeat for each episode:
    Initialise S
    Choose $A \sim \pi(\cdot|S)$
    $\mathbf{z} \leftarrow 0$
    Repeat for each step of the episode
    | Take action A, observe reward R and next state S'
    | $\delta \leftarrow R$
    | Loop for i in $\mathcal{F}(S, A)$
    | | $\delta \leftarrow -w_i$
    | | $z_i \leftarrow -z_i$
    | If S' is terminal, then:
    | | $w \leftarrow w + \alpha\delta\mathbf{z}$
    | | Go to next episode
    | Choose $A' \sim \pi(\cdot|S')$
    | Loop for i in $\mathcal{F}(S', A')$:
    | | $\delta \leftarrow \delta + \gamma w_i$
    | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
    | $z \leftarrow \gamma\lambda\mathbf{z}$
    | $S \leftarrow S'; A \leftarrow A'$

Figure A.3: Pseudocode for the Sarsa(λ) algorithm, adapted from Sutton and Barto's reinforcement learning textbook [18].

Algorithm parameters: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialise value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily
Loop for each episode:
    Initialise S
    Loop for each step of the episode:
    | Choose action A as a function of $\hat{q}(S', \cdot, \mathbf{w})$
    | Take action A, observe reward R and next state S'
    | If S is terminal:
    | | $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R - \hat{q}(S, A, \mathbf{w})]\mathbf{x}(S, A)$ (update rule)
    | | Go to next episode
    | $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma \max_a q(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})]\mathbf{x}(S, A)$ (update rule)
    | $S \leftarrow S'$
Until S is terminal

Figure A.4: Pseudocode for the Q-learning algorithm, adapted from Sutton and Barto's reinforcement learning textbook [18].

Appendix B

Results

This Appendix presents the experiment results of reward features and learning methods that were not included in Chapter 6 for brevity.

B.1 Reward Signal

Figure B.1 shows that adding a game over penalty improves convergence, confirming Steven and Pradhan's result [12]. Different values for the game over penalty yielded roughly the same learning curve. Using a value of -2 caused the agent to train the quickest, but only marginally.

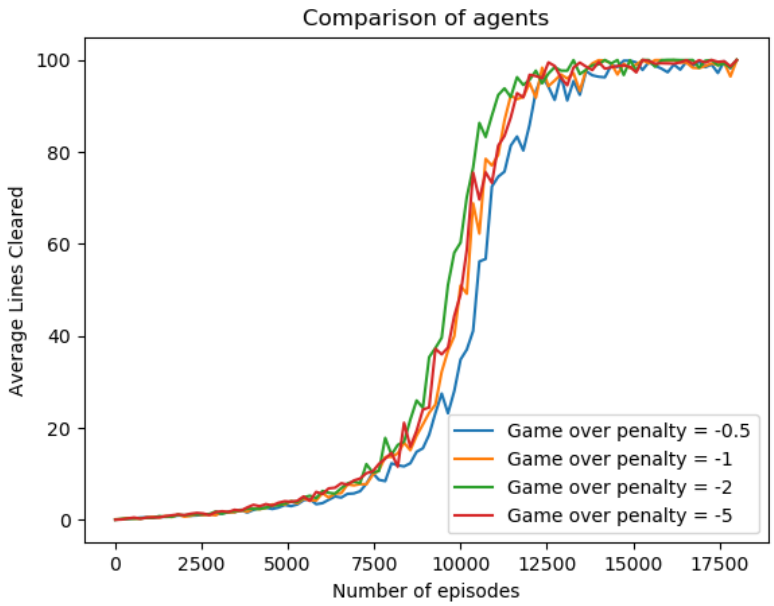


Figure B.1: Learning curves for varying the penalty that an agent receives for losing a game

Penalising the agent for increasing the maximum column height is a strong reward feature, as shown in Figure B.3. This may be because the feature is linked to the game over mechanism, since the game ends when the maximum column height exceeds the playing field height. Thus, agents that are encouraged to avoid stacking tetrominos should more effectively avoid game over. Convergence improves when the penalty is increased to a value of 0.5, after which there is little performance gain for penalising the agent more severely.

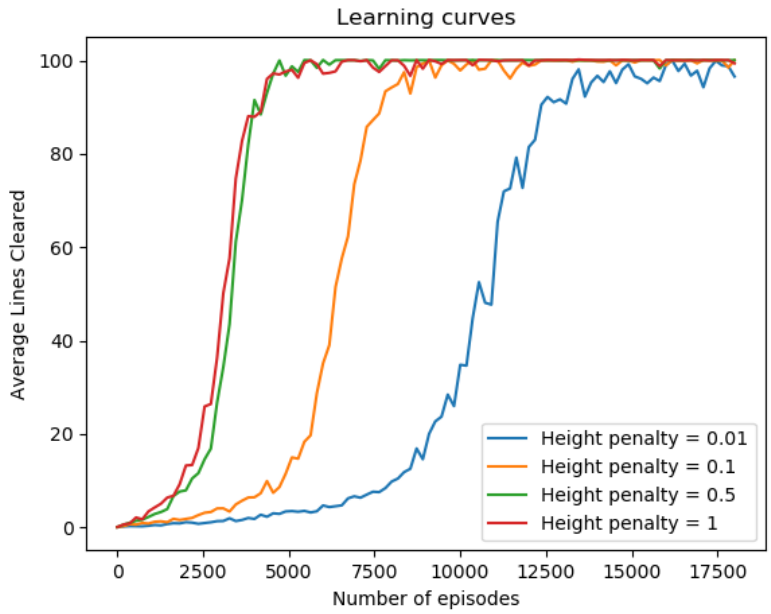


Figure B.2: Learning curves for varying the penalty that an agent receives for increasing the maximum column height

Another strong reward feature is the total number of holes in the playing field, as shown in Figure B.3. Avoiding holes is important because a line cannot be cleared if it has a hole in it. Increasing the penalty for creating a hole results in marginal performance gain.

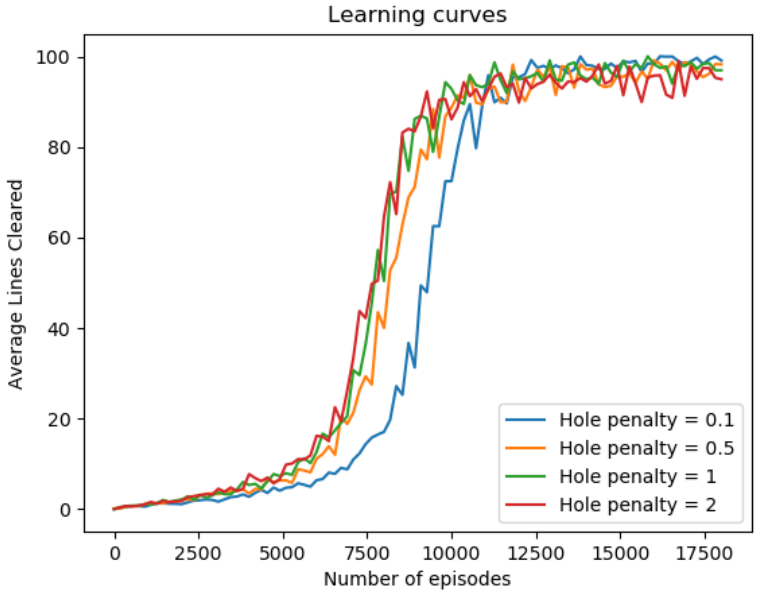


Figure B.3: Learning curves for varying the penalty that an agent receives for creating a hole

Figure B.4 shows the learning curve for an agent trained with only the unevenness reward feature. Penalising the agent for increasing the unevenness of the board allows it to train very quickly, but results do not converge at 100 lines cleared. Note that this reward feature may encourage undesirable behaviour, such as decreasing the unevenness of the playing field at the expense of creating a hole.

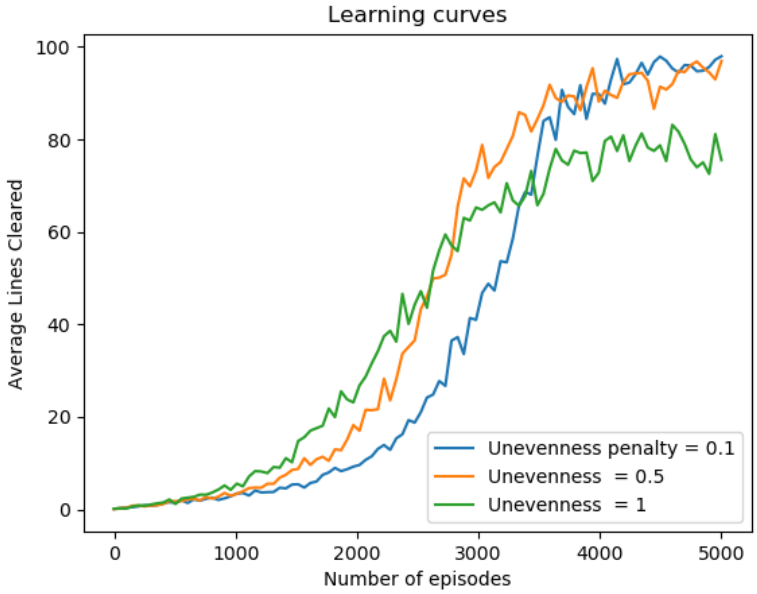


Figure B.4: Learning curves for varying the penalty that an agent receives for making the playing field more uneven

B.2 Learning methods

The results for n-step Sarsa considering different numbers of steps are shown in Figure B.5. It shows that increasing the number of steps beyond 3 decreases the performance of the agent. This may be because value estimates that take into account states far into the future (after a new tetromino is spawned) are noisy, since tetrominos spawn randomly.

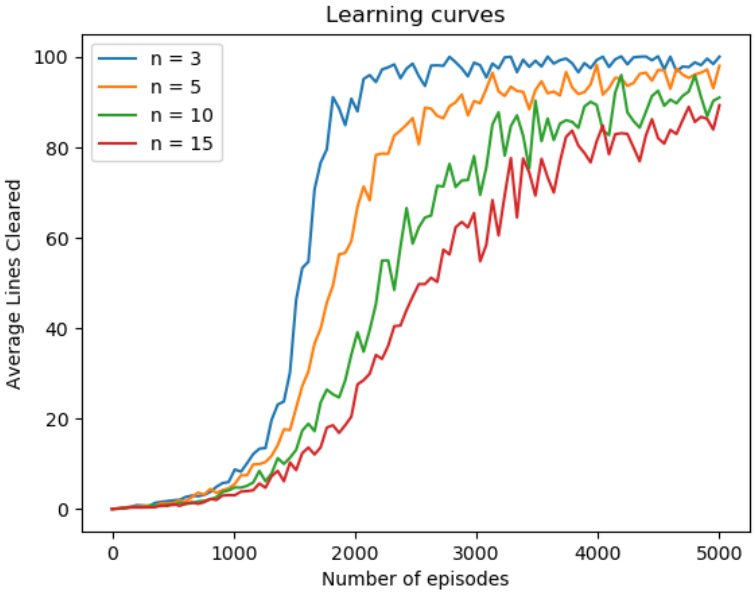


Figure B.5: Learning curves for n-Step Sarsa with varying n

Appendix C

Safety guidelines

contact Name	Room Nr.	Contact details
Mr. JC Schoeman	E418	jcschoeman@sun.ac.za
Dr. Jordaan	E420	wjordaan@sun.ac.za
Campus security	-	0218082333
Stellenbosch mediclinic	-	0218612000

Table C.1: Contact details in case of emergency

C.1 Overview

This project involves developing a reinforcement learning algorithm for Tetris. It is purely theoretical and programming based, with no physical experiments performed. However, the algorithms were trained on a computer in the electronic systems laboratory. Safety procedures for the laboratory are given:

C.2 General laboratory safety

C.2.1 Covid

The following protocols will be adhered to, due to covid-19:

- Sanitise hands when entering and exiting the laboratory
- Wear mask when inside laboratory
- Social distance by remaining at least 1.5m away from peers in laboratory

C.2.2 Emergency exit

The emergency exit for the electronic engineering building is to the right of the electronic systems laboratory's entrance.

C.2.3 Equipment safety

Computers are generally sensitive to power trips. As a precautionary measure, the loadshedding schedules will be checked on a daily basis, and the computer turned off and unplugged before loadshedding starts. It will only be plugged back in once loadshedding is over. Furthermore, the project files will be backed up online regularly to prevent data loss. Lastly, the workstation will be kept neat and clean.

Appendix D

Project schedule

Figure D.1 shows the proposed and actual timeline of planned activities. The project was planned to start on the Monday the first of March 2021 and end on Friday the ninth of July, one week before the deadline on the 16th of July. The project fell behind near the start when the literature review took longer than expected. Furthermore, several investigations ran in parallel and took longer than expected. The development of the final concept and report writing was delayed. The project ended on the 15th of July, one day before the deadline.

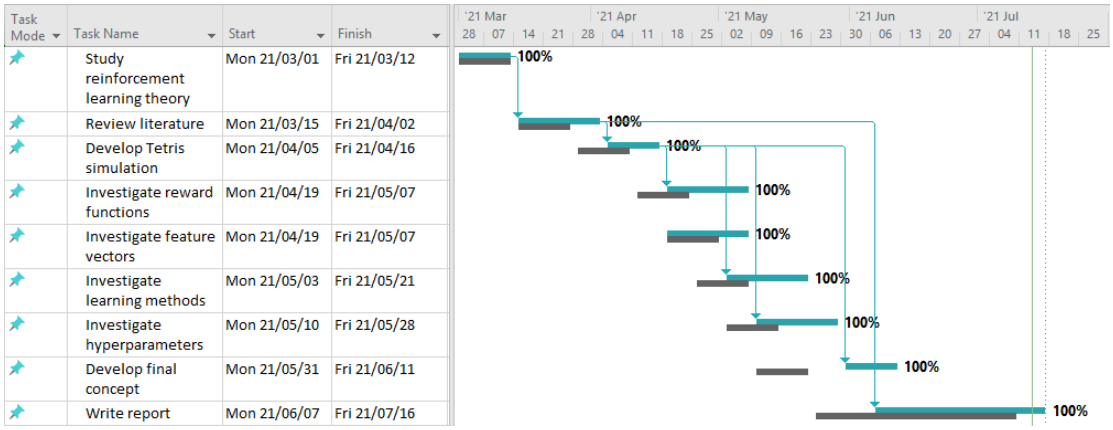


Figure D.1: Project proposed and actual timelines

Appendix E

Project cost and techno-economic analysis

E.1 Project costs

The proposed and actual project costs are shown in Tables E.1 and E.2 respectively. A comparison of planned and actual costs for each activity is shown in Figure E.1. Overall, the project took 60 hours longer than planned and ran over-budget by R29475. The project went over-budget on the reinforcement learning study, literature review and report writing activities. Although the investigations into reward functions, feature vectors, learning methods and hyperparameters took longer than expected, they remained on budget. This is because they required less engineering time than expected, but the experiments took long to run. Time spent waiting for experiments to finish was counted in facilities use but not engineering time.

			Engineering time		Facility use		Capital cost	Total
Nr.	Activity	hrs	rate (R/hr)	R	rate (R/hr)	R	R	R
1	Study reinforcement learning theory	50	450	22500	7,5	375		22875
2	Review literature	50	450	22500	7,5	375		22875
3	Develop Tetris simulation	50	450	22500	7,5	375		22875
4	Investigate reward functions	50	450	22500	7,5	375		22875
5	Investigate feature vectors	50	450	22500	7,5	375		22875
6	Investigate learning methods	50	450	22500	7,5	375		22875
7	Investigate hyperparameters	50	450	22500	7,5	375		22875
8	Develop final concept	50	450	22500	7,5	375		22875
9	Finalise report	150	450	67500	7,5	1125		68625
	Capital cost: computer						15000	15000
	Totals	550		247500		4125	15000	266625

Table E.1: Project proposed cost

Nr.	Activity	Engineering time hrs	rate (R/hr)	R	Facilities use rate (R/hr)	hrs	R	Capital cost R	Total R
1	Study reinforcement learning theory	100	450	45000	7,5	100	750		45750
2	Review literature	80	450	36000	7,5	80	600		36600
3	Develop Tetris simulation	60	450	27000	7,5	60	450		27450
4	Investigate reward functions	30	450	13500	7,5	130	975		14475
5	Investigate feature vectors	50	450	22500	7,5	100	750		23250
6	Investigate learning methods	50	450	22500	7,5	100	750		23250
7	Investigate hyperparameters	10	450	4500	7,5	60	450		4950
8	Develop final concept	30	450	13500	7,5	50	375		13875
9	Finalise report	200	450	90000	7,5	200	1500		91500
	Capital cost: computer							15000	15000
	Totals	610		274500		880	6600		296100

Table E.2: Project actual cost

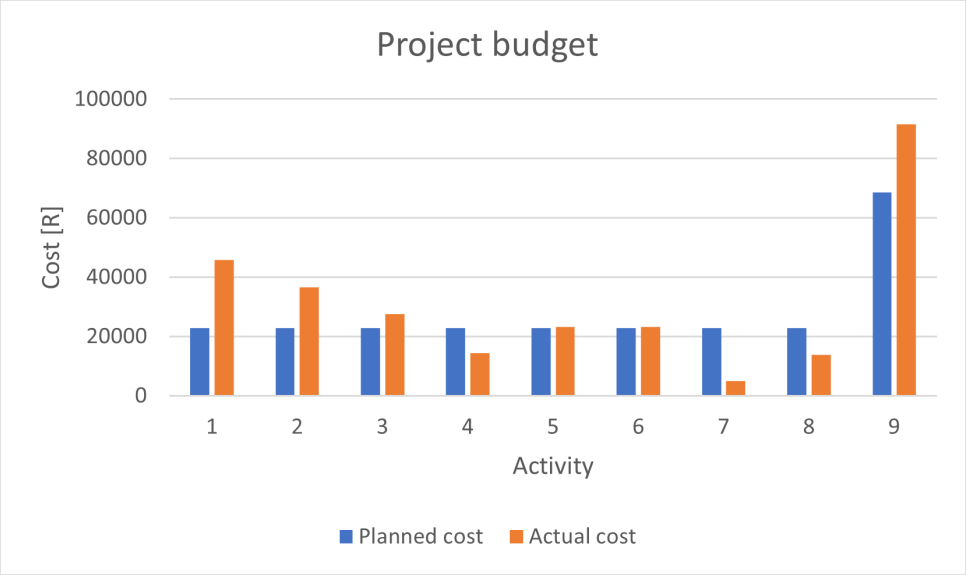


Figure E.1: Comparison of project proposed and actual budgets

E.2 Technical impact

Linear function approximation is simpler technique and often requires less training time that non-linear function approximation techniques such as neural networks. The reduction in training time makes algorithm development easier, but at the cost of agent performance.

A model-free reinforcement learning agent using linear function approximation was trained to play Tetris with simplified tetrominos. The agent was trained on a small playing field and tested on a full size playing field. This finding - that agents can be trained in a smaller MDP than what they are deployed in, is useful in applications where the state space cannot be reduced sufficiently to for a reinforcement learning agent to train, or where simulating agent experience is difficult.

E.3 Return on investment

Reinforcement learning has many industry applications: self driving cars; trading and finance; natural language processing; automated medical diagnoses; and recommender systems for online content services. These are lucrative fields, and successful agents may generate return on investment. Furthermore, the idea that was introduced in this project, to train an agent on a smaller MDP than the one it is deployed in, is applicable in industry, as large state spaces plague many real-world applications.

E.4 Potential commercialisation

If improvements are made to enable the agent to play the standard Tetris game, then the project result may be commercialised as a computer opponent in multiplayer Tetris.

List of References

[1] Graham, S.: Mathematicians Prove Tetris Is Tough. https://www.scientificamerican.com/article/mathematicians-prove-tetr/, 2002. [Online; accessed 8-June-2021].

[2] Amundsen, J.B.: A comparison of feature functions for Tetris strategies. , no. June, p. 78, 2014.

[3] Szita, I. and Lorincz, A.: Learning tetris using the noisy cross-entropy method. *Neural Computation*, vol. 18, no. 12, pp. 2936–2941, 2006. ISSN 08997667.

[4] Thiery, C., Scherrer, B., Thiery, C., Scherrer, B., Controllers, B. and Computer, I.: Building Controllers for Tetris To cite this version :. pp. 3–11, 2009.
Available at: https://hal.inria.fr/inria-00418954/document

[5] Böhm, N., Kóokai, G. and Mandl, S.: An Evolutionary Approach to Tetris. *Proceedings of the 6th Metaheuristics International Conference (MIC2005)*, pp. 137–148, 2005. ISSN 00222429.
Available at: https://faui20a.informatik.uni-erlangen.de/publication/download/mic.pdf

[6] Blue Planet Software: 2009 Tetris Design Guideline. 2009.
Available at: https://tetris.fandom.com/wiki/Tetris_Guideline

[7] Carr, D.: Applying reinforcement learning to Tetris. *Business*, pp. 1–15, 2005.

[8] Thiery, C. and Scherrer, B.: Improvements on learning tetris with cross entropy. *ICGA Journal*, vol. 32, no. 1, pp. 23–33, 2009. ISSN 13896911.
Available at: https://hal.inria.fr/inria-00418930/document

[9] Tsitsiklis, J.N. and Van Roy, B.: Feature-based methods for large scale dynamic programming. *Machine Learning*, vol. 22, no. 1-3, pp. 59–94, 1996. ISSN 08856125.
Available at: https://www.mit.edu/$\sim$jnt/Papers/J060-96-bvr-feature.pdf

[10] Bertsekas, D.P. and Ioffe, S.: Temporal Differences-Based Policy Iteration and Applications in Neuro-Dynamic Programming. *Electrical Engineering*, vol. 1996, no. August 1996, pp. 1–19, 1997.
Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.7759&rep=rep1&type=pdf

[11] Thiam, P., Kessler, V. and Schwenker, F.: A reinforcement learning algorithm to train a tetris playing agent. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8774, pp. 165–170, 2014. ISSN 16113349.

[12] Stevens, M. and Pradhan, S.: Playing Tetris with Deep Reinforcement Learning. 2016.

[13] Kakade, S.: A natural policy gradient. *Advances in Neural Information Processing Systems*, 2002. ISSN 10495258.

[14] Gabillon, V., Ghavamzadeh, M. and Scherrer, B.: Approximate dynamic programming finally performswell in the game of Tetris. *Advances in Neural Information Processing Systems*, pp. 1–9, 2013. ISSN 10495258.

[15] Fahey, C.: Tetris ai, computer plays tetris.
Available at: `http://colinfahey.com/tetris/tetris_en.html`

[16] Groß, A., Friedland, J. and Schwenker, F.: Learning to play Tetris applying reinforcement learning methods. *ESANN 2008 Proceedings, 16th European Symposium on Artificial Neural Networks - Advances in Computational Intelligence and Learning*, , no. April, pp. 131–136, 2008.
Available at: `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.218.4954&rep=rep1&type=pdf`

[17] Silver, D.: Lectures on reinforcement learning. URL: `https://www.davidsilver.uk/teaching/`, 2015.

[18] Sutton, R. and Barto, A.: *Reinforcement Learning, An Introduction*. 2nd edn. MIT Press, Cambridge, Massachusetts, 2020. ISBN 9780262039246.