LEARNING TETRIS WITH REINFORCEMENT LEARNING

Submitted 2021-10-13, in partial fulfillment of
the conditions for the award of the degree **BIG DATA & ANALYTICS.**

IOANNIS TSIROVASILIS
ME1948

Supervised by MICHAEL FILIPPAKIS

School of Digital Systems University of Piraeus Greece

Abstract

The last decades, scientists have expressed an increasing interest for the game Tetris and more precisely for efficient algorithms that can score the most in-game points.

A plethora of approaches have been tried out, including genetic algorithms, linear programming, cross-entropy and natural policy gradient, but none of them competes with experts players playing under no time pressure.

In recent years, scientists have started applying reinforcement learning in Tetris as it displays effective results in adapting to video game environments, exploit mechanisms and deliver extreme performances.
Current thesis aims to introduce Memory Based Learning, a reinforcement learning algorithm which uses a memory that helps in the training process by replaying past experiences.

Contents

Abstract **1**

List of Figures **4**

1 Introduction **5**
1.1 Introduction . . . 5
1.2 Tetris . . . 5
1.3 State Representation . . . 6

2 Background Related Work **7**
2.1 Tetris . . . 7
2.2 Mathematical foundations of Tetris . . . 9
2.3 Solving NP-Complete Problems . . . 10
2.4 Exploration . . . 10
2.5 Existing applications . . . 11
2.6 Large state space successes . . . 11
2.7 Tetris Related Reinforcement Learning . . . 12
2.8 Relational Reinforcement Learning . . . 16
2.9 Reinforcement Learning Definitions . . . 18
2.10 State-Action Pairs Complex Probability Distributions of Reward . . . 20
2.11 Neural Networks and Deep Reinforcement Learning . . . 21

3 Deep Learning Neural Networks **23**
3.1 Neural Networks Definition . . . 23
3.2 Biological Neuron . . . 23
3.3 Model of Artificial Neural Network . . . 24
3.4 Feedforward Network . . . 25
3.5 Activation Functions . . . 26
3.5.1 Linear Activation Function . . . 26
3.5.2 ReLU (Rectified Linear Unit) Activation Function . . . 27
3.6 Perceptron . . . 27
3.7 Learning Rate . . . 27
3.8 Weights . . . 28
3.9 Back Propagation . . . 28
3.10 Stochastic Gradient Descent . . . 29
3.11 Adam Optimization Algorithm . . . 30
3.12 Deep Reinforcement Learning . . . 31

4 Design & Specifications **32**
4.1 Redesigning the Tetris State Space . . . 32
4.2 The Structure of a Reinforcement Learning Agent . . . 32
4.3 The Discovery of Transitions . . . 33
4.4 Exploring Amongst Transitions . . . 33
4.5 Update the Value Function . . . 33
4.6 Application Design . . . 34

5 Implementation **36**

5.1 Environment . . . 36
5.2 Q-Learning . . . 36
5.3 SARSA . . . 37
5.4 Memory Based Learning . . . 39
5.5 Rewards and Exploration . . . 40
5.6 State Representation . . . 40
5.7 Deep Neural Network Agent . . . 41

6 Evaluation **42**
6.1 SARSA Results . . . 42
6.2 QLearning Results . . . 43
6.3 Memory Based Results . . . 43

7 Conclusion **47**

References **48**

Appendices **50**

A Example: Source Code Samples **50**

List of Figures

2.1 Tetris Board 10x20 Square Blocks 8
2.2 "I" 8
2.3 "S" 8
2.4 "Z" 8
2.5 "O" 8
2.6 "T" 8
2.7 "L" 8
2.8 "J" 8
2.9 Melax's Reduced Tetrominoes 13
2.10 Melax's results as taken from Melax [1] 14
2.11 Bdolah and Livnat's results as taken from Bdolah and Livnat [2] 15

3.1 Biological Neuron 23
3.2 Artificial Neural Network Model 24
3.3 Single layer feedforward network 25
3.4 Multilayer feedforward network 26
3.5 Perceptron 27
3.6 Back Propagation 28

6.1 SARSA Results 42
6.2 QLearning Results 43
6.3 Memory Based Results 44
6.4 Initial Neural Network Comparison 45
6.5 Deeper Neural Network Comparison 46

Chapter 1

Introduction

1.1 Introduction

Reinforcement learning is a branch of artificial intelligence that focuses on achieving the learning process during an agent's lifespan. This is accomplished by giving the agent the ability to retain its circumstances, providing him with enough memory of the environmental events, and rewarding or punishing his actions in the context of a predefined reward policy. The drawback of traditional reinforcement learning is the exponential increase in agent's storage and exploration time requirements when there is a linear increase in the dimensions of the problem.

Tetris is a game created in 1985 by Alexey Pajitnov and, in the last decades, it has been an important research topic for both mathematics and artificial intelligence communities. Unfortunately, despite being a conceptually simple game, it is NP-complete [3].

In the context of this thesis, we try to apply reinforcement learning to Tetris, a mass challenge due to the size of the possible states that the game may reside at a given time. Our approach involves simplifying the Tetris game description and conducting a comparative analysis between 3 distinct reinforcement learning algorithms. This involves extracting the core information required to function intelligently from the full problem description. Reducing the problem description makes it possible to downgrade the complexity-related restrictions and lead to the broader application of reinforcement learning.

1.2 Tetris

Tetris consists of 7 types of pieces called *Tetrominoes* and more specifically they represent the letters "I", "O", "S", "Z", "L", "J" and "T" as depicted in figures 2.2 to 2.8. These pieces are formed by four square blocks and are spawned one at a time in the Tetris board, which is a rectangle of 20×10 square blocks, Figure 2.1. The user controls the spawned piece and is free to move it right or left infinite times inside the rectangle boundaries. The piece is falling one block-row at a time, with a frequency that is predefined and depends on the game *Level*. The user can accelerate the falling rate by pressing a relevant button. Another available move is the rotation of the pieces in a clock-wise manner. Suppose any of the square blocks of the falling piece collides, vertically, with the ground or another square block of an already fallen piece. In that case, it stops moving, stays in place, and two things happen after that: i) if a block-row *(Line)* is full of pieces' square blocks, then those squares blocks are removed, and all the upper structure moves one line down, ii)

the next piece is spawned and starts falling. The game ends when no more pieces can be spawned, i.e., when the structure is so tall that the next piece to be spawned will overlap with the structure. In the board, there is information about the cleared *Lines*, the current *Level*, which affects the falling rate and scoring awards, the *Next* piece to be spawned, and the current *Score*.

All these years, many different implementations, with slight or considerable variations, have been published. One of them is used in the official world Tetris tournament called *Classic Tetris World Tournament*.

1.3 State Representation

A factor that highly affects the performance of a Reinforcement Learning System is the state representation, i.e., a model describing the environment's current view. For example, in a backgammon game, the state could be the chips' positions and who is turn it is. The state representation can fully encapsulate the state of the game or contain redundant information. This is important as the size of the representation can have adverse effects on training time. A naive approach where the representation is the whole environment, i.e., the pixels of the screen, is shown to slowly converge by Andre and Russell [4]. What makes Tetris' case more difficult is the significant branching factor. The branching factor is the number of following possible states given the current state. This factor lies between 9 - 39 and depends on the different Tetrominoes' shapes and the rotations applied to them. For example, the "O" piece has no rotations, "I", "S", "Z" have two while "T", "L", and "J" have 4. In this paper, we evaluate the performance of agents using a state representation consisting of 4 features, namely, lines cleared, number of holes, total bumpiness, and total height.

Chapter 2

Background Related Work

This chapter aims to introduce the formal specifications of Tetris that define the domain of the problem. Later we study beyond the raw specification, and we discuss the mathematics of Tetris. Finally, we justify the adoption for solving Tetris with reinforcement learning, introduce the theory used throughout the thesis and talk about related research in reinforcement learning. Also, we end off with a review of previous attempts to apply reinforcement learning to Tetris.

2.1 Tetris

Tetris, shown in Figure 2.1, is established to the point that it awards its name to the category of puzzle games. Each variation may have a range of different tetrominoes. These tetrominoes represent the alphabet letters I, O, S, Z, L, J, and T as depicted in Figures 2.2 to 2.8

Tetrominoes can also be rotated and translated in the absence of obstructions.

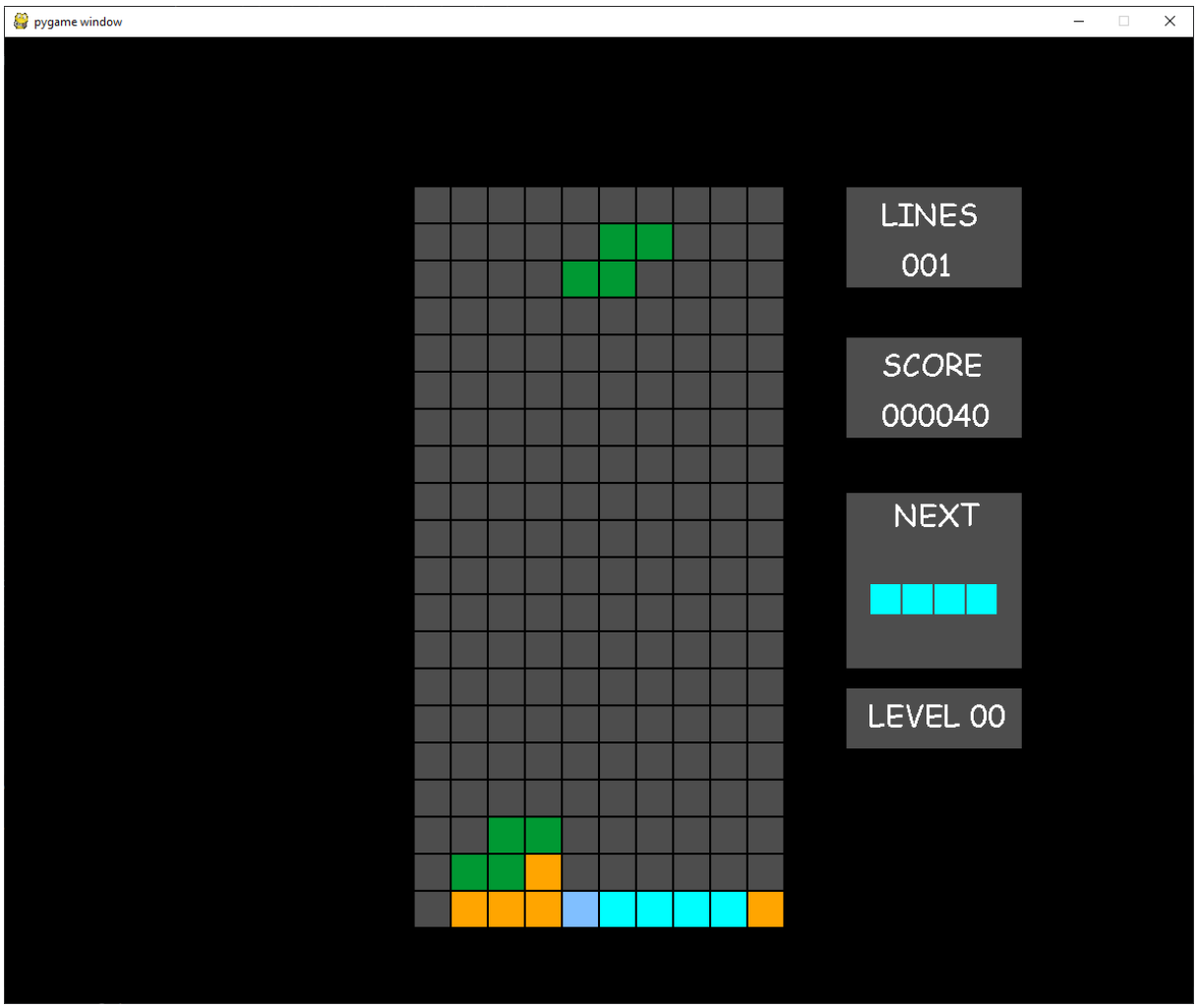


Figure 2.1: Tetris Board 10x20 Square Blocks

At the start of the game, a randomly selected tetromino spawns at the top center block of the board—the tetromino descents at a fixed speed, determined by the current level. If there is an obstacle right beneath, the tetromino stops descending, sets in place, and two things follow. First, the game erases the rows full of tetromino blocks, and the board

Figure 2.2: "I"

Figure 2.3: "S"

Figure 2.4: "Z"

Figure 2.5: "O"

Figure 2.6: "T"

Figure 2.7: "L"

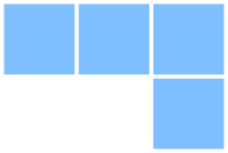


Figure 2.8: "J"

moves down by the number of erased rows in this step. Lastly, a new tetromino spawns, and the game continues until a tetromino cannot be placed at the top of the board.

Many different approaches of artificial intelligence and other fields have been applied to Tetris, making the comparison between works hard due to the plethora of variations in the rules and the mechanics that a game of Tetris can be implemented. However, as this thesis focuses on comparing three different reinforcement learning algorithms applied to a single variation of Tetris, it is easy to imply that there can be no implementation discrepancies, as specific guidelines define our implementation. Therefore the achieved results of the agents will be comparable. The following standard set is chosen for this research.

- Tetris has a board with dimensions 20x10
- Tetris has seven distinct piece (Figures 2.2 to 2.8)
- The current game tetromino is selected by a queue of initially seven randomly chosen pieces. This queue refreshes every time its length is less or equal to two by adding seven more pieces
- Points are awarded by combining each piece that lands and the number of cleared lines per step. Losing the game grants penalty points.

2.2 Mathematical foundations of Tetris

The possibility of generating a sequence of tetrominoes that guarantee the end of any Tetris games in a board of width 2(2n+1), with n being an integer, has been mathematically proven [5]. The latter is achieved by spawning alternating Z and S pieces into the board, which results in the gradual construction of taller and taller structures and eventually the termination of the game. Even if the agent were to play a flawless game of Tetris, the series of pieces that guarantee the game's termination are statistically inevitable after a long enough time (infinite period). The number of rows completed by a good Tetris player will follow an exponential distribution [6], owing to the stochastic nature of the game. Some Tetris games are more complex than others due to the pieces drawn and the order in which they are delivered, and the resulting performance spectrum can be mistaken for erratic behavior on the part of the player. Breukelaar et al. [3] prove Tetris is NP-complete problem. The arisen implication relates to the impossible computation of linearly searching the entire policy space and picking an ideal action. That is where approximation techniques fit, like reinforcement learning, to determine an optimal policy. One assumption reinforcement learning requires is that the environment has the Markov property [7]. Tetris satisfies the requirement mentioned above, as the state represents all the relevant information required to make an optimal decision at any instant in time. In other words, there is no historical momentum of the system's current state; thus, any future occurrence is entirely dependent on the system's current state. If we are handed control of a Tetris game at any point, we are as equipped to play from that point as we would be had we played up until that point.

2.3 Solving NP-Complete Problems

Attractive and promising solutions to problems outside of the range of computations of linear search methods can be discovered with biological processes emulations. Genetic algorithms and reinforcement learning are two such approaches. Genetic algorithms, for example, search directly in the solution (policy) space of a problem, giving birth to solutions amongst the fittest individuals to approach an optimal solution. On the other hand, reinforcement learning yields an environment to an agent that is subsequently left to explore for itself. The agent gets feedback directly from the environment through rewards or penalties and continuously updates its value function to achieve the optimal policy. Ideally, both methods converge on the best policy [8], although their different routes gear them towards particular problems. Additionally, Reinforcement learning offers a higher resolution than genetic algorithms, as genetic algorithms select optimal candidates at the population level. In contrast, reinforcement learning selects optimal actions at an individual level [8]. Every action taken under reinforcement learning is evaluated and driven towards the optimal action for that state, while genetic algorithms reward complete genetic strains regardless of the behavior of individual genes within the previous episode. Reinforcement learning is different from genetic algorithms by indirectly adjusting its policy by updating its value function, rather than introducing random variations directly to its policy and relying on the agent chancing upon a superior policy. A vast deal of information is conveyed in a Tetris game, and reinforcement learning enables the agent to capture this information and adapt accordingly. Furthermore, this would enable a directed real-time adjustment of the agent's policy rather than a global adjustment at the end of the game. Another consideration is that as the agent's performance improves, the number of rows completed in a game increases, and the lifespan of the agent increases. This does not negatively impact the reinforcement learning agent as it learns with every move but has an increasingly significant impact on the rate of improvement of the genetic algorithm agent since it learns with the end of each game. These traits indicate that reinforcement learning is better suited to solving Tetris than genetic algorithms.

2.4 Exploration

The agent can have one of a variety of exploration policies. However, regarding a purely greedy policy, the agent will always select the state transition to offer the most significant long-term reward. Even though this will immediately benefit the agent, it may fail to find the best policy in the long term. Using an ϵ-greedy method, the agent will choose the best state transition the majority of the time and take exploratory transitions the rest of the time. The frequency of these exploratory transitions is determined by the policy's value of ϵ. It is possible to vary ϵ to have an initially open-minded agent that proves its value function while its experience increases over time. One problem with the ϵ-greedy approach is that the agent explores aimlessly and is as likely to explore an unattractive avenue as it is to explore a promising one. Another approach is to start with a very high ϵ and gradually diminish it to zero. This encourages the agent to explore freely at the beginning but choose wiser action later when experience is gained. promising ones.

$$\epsilon = 1 - d/1500 \tag{2.1}$$

$$d = min(E, 1500) \tag{2.2}$$

The selection of 2.1 is driven by the fact that we did not have the necessary computing power to train the agent for a long time, so the number of 1500 is given as the maximum allowed episode for exploration.

2.5 Existing applications

Reinforcement learning performs well in small domains, and by using the insight offered by Sutton and Barto [7], the creation of an agent that plays simple games like Tic-Tac-Toe or Blackjack successfully is effortless. It it successfully applied to many sophisticated problems such as :

- Packet routing in dynamically changing networks [9]
- Robotic control [10]
- Acrobat [7]
- Chess [11]

Bellman is cited Sutton and Barto [7] as stating that reinforcement learning suffers from the "curse of dimensionality". In other words, the exponential increase in the system's complexity as the number of elements in it increases linearly. This tendency is responsible for reinforcement learning having relatively few successes in large state-space domains [12]. These successes include :

- RoboCup-Soccer Keep-Away [12]
- Backgammon [13]
- Elevator control [14]

2.6 Large state space successes

TD-Gammon

Tesauro [13] used reinforcement learning to train a neural network to play backgammon. The program was such a success that its first implementation (Version 0.0) had abilities equal to Tesauro's well-established Neurogammon 2 [13]. Furthermore, by Version 2.1, the TD-Gammon is regarded as playing at a level extremely close to that of the world's best human players and has influenced how expert backgammon players play [13]. The reliance on performance rather than established wisdom and the unbiased exploration leads, in some circumstances, to TD-gammon adopting non-intuitive policies superior to

those utilized by humans [13]. Backgammon is estimated to have a state-space larger than 10^{20}. This state-space was reduced by the use of a neural network organized in a multiplayer perception architecture. TD learning with eligibility traces was responsible for updating the weighting functions on the neural network as the game progressed. Another benefit associated with using reinforcement learning methods rather than pure supervised learning methods was that TD-gammon could be (and was) trained against itself [13].

RoboCup-Soccer Keep-Away

Sutton et al. [12] managed to train reinforcement learning agents to complete a successfully subtask of full soccer, involving a team of agents — all learning independently — keeping the ball away from their opponents. This implementation overthrew many difficulties, like having multiple independent agents functioning with delayed rewards and, most importantly, functioning in ample state space. The state-space problem was resolved with the use of linear tile-coding (CMAC) function approximation to reduce the state-space to a more feasible size [12].

2.7 Tetris Related Reinforcement Learning

We found three existing extensions of reinforcement learning to Tetris that all implement one- piece methods.

Reduced Tetris

Melax [1] applied reinforcement learning to a greatly reduced version of the Tetris game. His Tetris game had a well with a width of six, an infinite height, and the greatly reduced piece set shown in Figure 2.9. The game's length was dictated by the number of tetrominoes attributed to the game, which was set at ten thousand. Although the height of the Tetris board was infinite in theory, the active layer in which blocks were placed was two blocks high. Any placement above this level had the result of lower layers being discarded until the structure had a height of two. The number of discarded rows was kept as record by the game, which was used as a score for the agent's performance. This scoring approach resulted in better performance corresponding to a lower score. The two-block active height prevented the agent from lowering the block structure and completing rows that it initially failed to complete. This is different from traditional Tetris, where a player can complete a previously unfilled row after reducing the well structure and re-exposing it. Furthermore, since the pieces are drawn stochastically, and unfilled rows form an immutable blemish on the performance evaluation of the agent, this introduces a random aspect to the results. The agent was implemented using the TD(0) and was punished a hundred points for each level it introduced higher than the working height of the well. The agent of Melax achieved significant learning, as shown in Table 2.1. These results are shown in Figure 2.10.

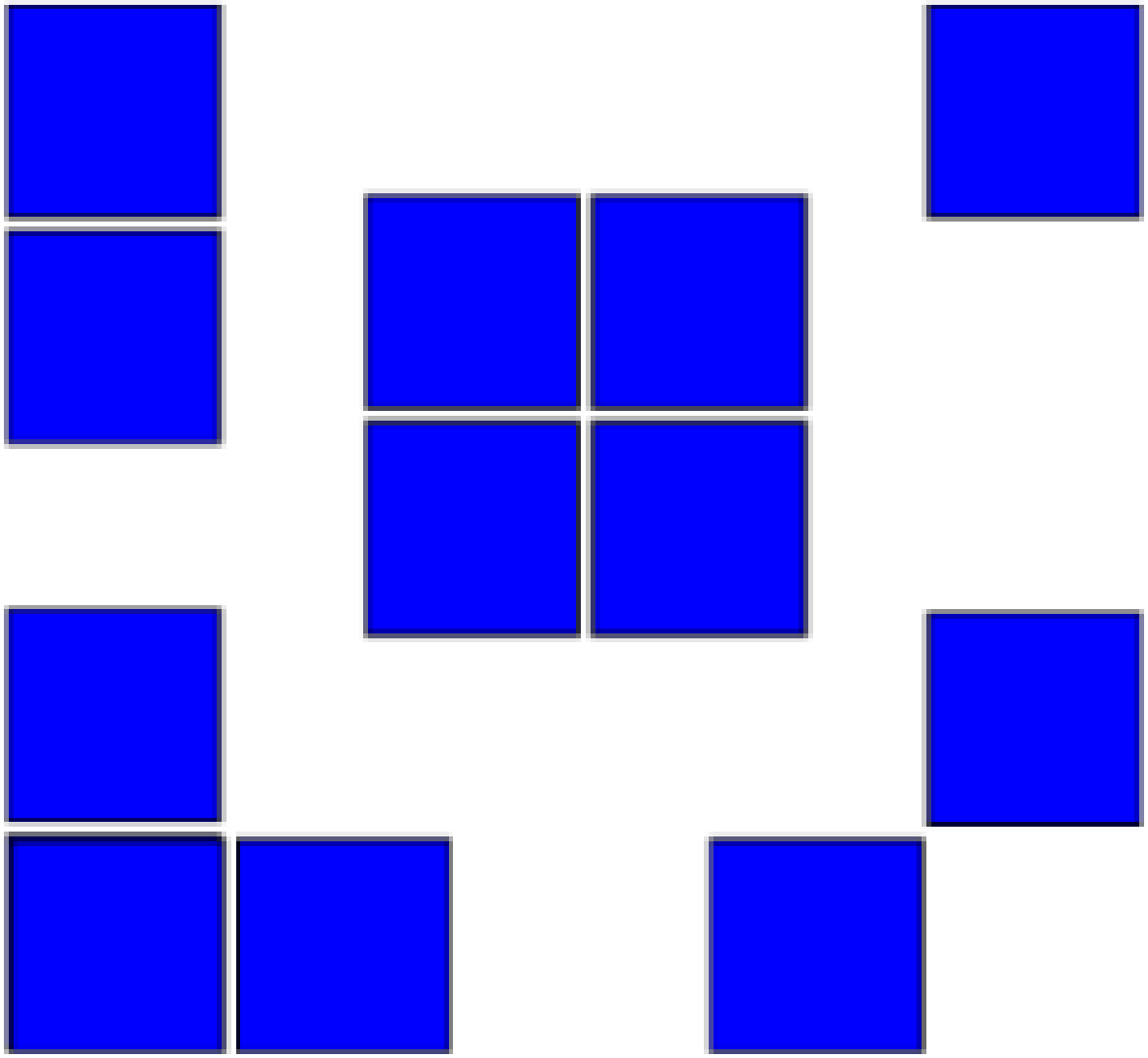


Figure 2.9: Melax's Reduced Tetrominoes

Game	Height
1	1485
2	1166
4	1032
8	902
16	837
32	644
64	395
128	303
256	289

Table 2.1: Table to test captions and labels.

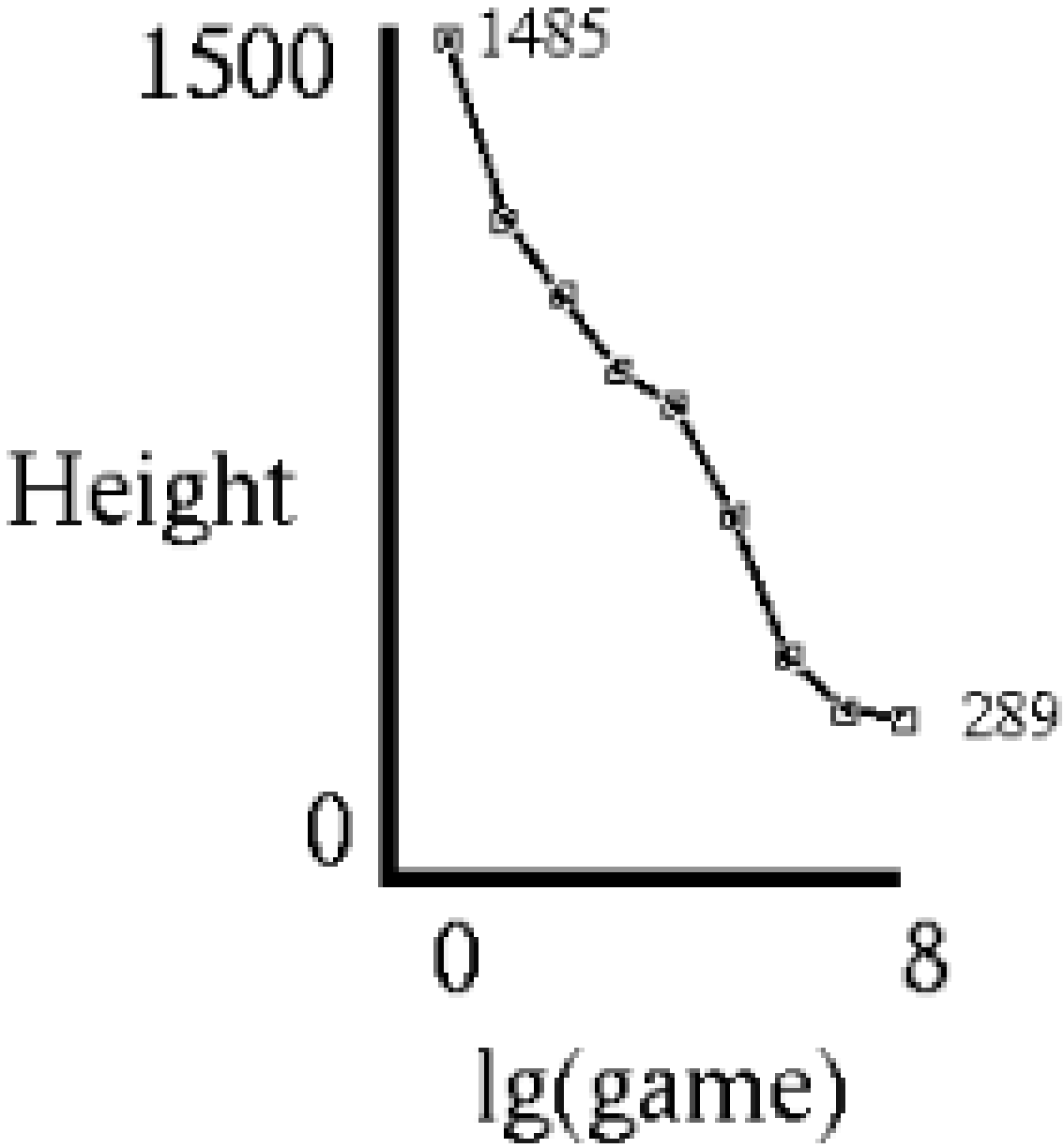
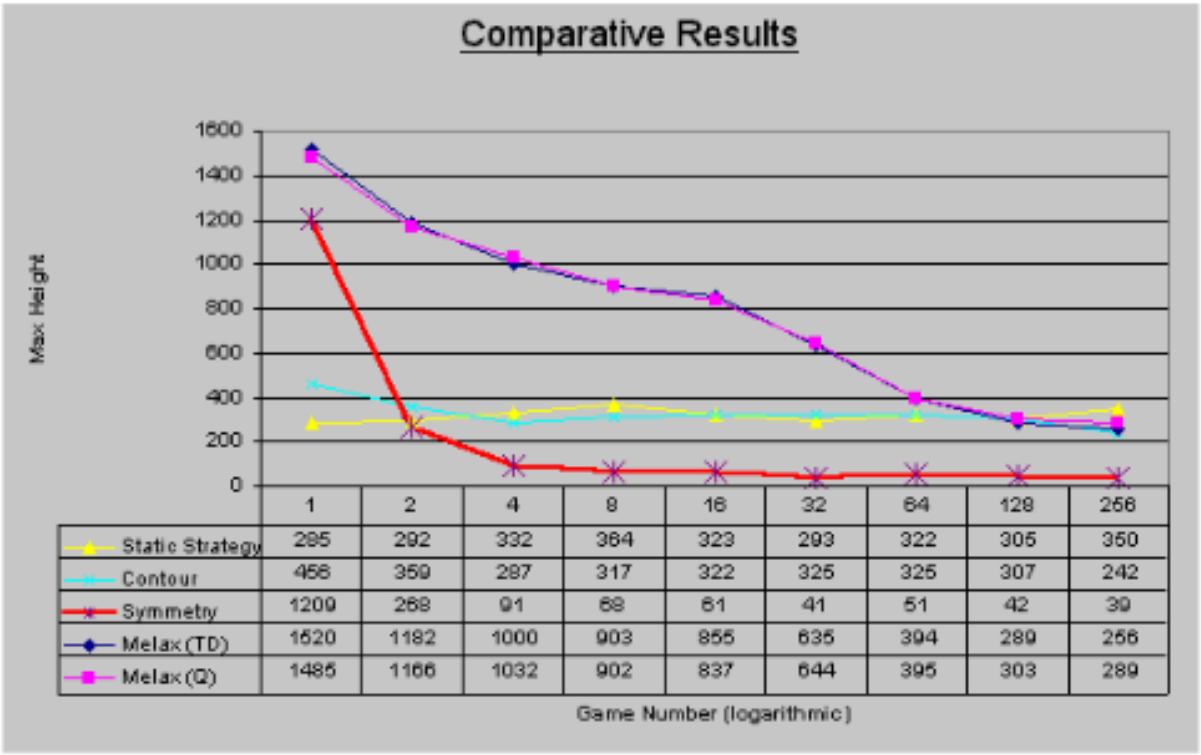


Figure 2.10: Melax's results as taken from Melax [1]

Mirror Symmetry

Max's approach was adopted and extended by Bdolah and Livnat [2], who investigated different reinforcement learning algorithms and introduced state-space optimizations. First, the state-space was reduced into two distinct approaches. In the first approach, subsurface information was discarded, letting only the contour of the game to consider. This approach was further divided into considering the contour differences as positive or negative. The second state-space reduction used mirror symmetry within Melax's well to reduce the number of different states.



	1	2	4	8	16	32	64	128	256
Static Strategy	285	292	332	364	323	293	322	305	350
Contour	458	359	287	317	322	325	325	307	242
Symmetry	1209	268	91	68	61	41	51	42	39
Melax (TD)	1620	1182	1000	903	855	635	394	289	256
Melax (Q)	1485	1166	1032	902	837	644	395	303	289

Figure 2.11: Bdolah and Livnat's results as taken from Bdolah and Livnat [2]

Both optimizations appear to have significantly improved the performance of the agent, judging by the chart shown in Figure 2.11. There are, however, some troubling viewpoints to these results.

The results of mirror symmetry are far superior to the results achieved by any other method. This optimization effectively ignored one reflection of duplicated states and thus should have sped up the learning process while converging on the same solution. The accelerated learning is apparent, but the results show that the mirror symmetry led to adopting a distinctly different policy to that adopted by the original agent. This means that the value function must have converged on different values, negating the original assumption that the values are identical and necessarily redundant. The contour approach extended the perceptive ability of the agent and maintained information below the original two-layer structure. This enabled the agent to reduce the well structure throughout the game continually. The results in the figure seem to indicate swift learning. By the end of the first game, the agent settles on a policy that produces a result far superior to the original Melax result. Despite the dubious results associated with the mirror symmetry optimization, it is a sound suggestion that is legitimate in the Tetris game defined by Melax. This optimization is equally legitimate in the full game since every tetromino in the standard tetromino set is mirrored within the set. Incorporating this in reducing our final state space would roughly halve the number of states required in describing the Tetris well.

2.8 Relational Reinforcement Learning

Relational reinforcement learning was utilized to the full board of Tetris problem by Driessens [15]. Relational reinforcement learning is different from traditional methods in its structuring of the value function. For example, rather than storing every possible state in a table, the relationship between the elements in the environment is utilized in developing a reduced state space. This state information is then stored in a decision tree structure. Driessens approached the problem with three different relational regression methods [15] which he developed throughout his thesis. The first of these regression methods had already proven itself with the successful extension of reinforcement learning to Digger. Driessens results for full Tetris are shown in Table 2.2. The RRL-RIB agent reached its optimal policy within fifty training games. In the four hundred and fifty subsequent training games, this policy was not improved upon. The RRL-KBR

Regression Method	Learning Games — Average Completed Rows	
RRL-TG	5000	10
RRL-RIB	50	12
RRL-KBR	10-20	30-40

Table 2.2: Relational regression results [15]

The agent reached a better policy earlier than the other regression methods. However, it then rather unexpectedly unlearned its policy after a further twenty to thirty games. Since this is a full implementation of Tetris, its results can be compared against other one-piece artificial intelligence methods. The best-hand-coded competitor completes an average of six hundred and fifty thousand rows per game, and the best dynamic agent, utilizing genetic algorithms, completes an average of seventy-four thousand rows per game [6]. Driessens results are not impressive in light of the competition and are very poor even for human standards. Driessens attributes the reduced functionality to Q-learning, stipulating that Q-learning requires a reasonable estimate of the future rewards in order to function correctly and that the stochastic nature of Tetris critically limits the accuracy of these estimates. Since his regression methods were derived from Q-learning, this inadequacy impacted all of his methods. Furthermore, Q-learning is known to be unstable [12], [16] when incorporated in function approximation, and this could certainly have contributed to the poor performance reflected in the above results. Despite the final results of Driessens's agent, the idea of exploiting the internal relationships present within the Tetris well as a means of reducing the state space is an attractive one.

Tsitsiklis & Van Roy [17] formulate Tetris as a Markov decision problem using a 200-dimensional binary vector to represent each state and a 7-dimensional binary vector for the pieces. As a Tetris board is a 20x10 rectangle of blocks, each element of the state vector matches precisely one block of the board with values 1or 0 if the block is occupied by a piece or not, respectively. As for the piece vector, all the elements are 0 except for the ones that occupy a block. Using feature-based value iteration with features being the height of the board and the number of holes, they achieved 32 cleared lines.

Bertsekas & Ioffe [18] tried out an approximate version of the λ-policy iteration method as well as an optimistic version. Use a linear feature-based approximation architecture

with optimistic λ-policy iteration. The two methods performed similar results, with the best being optimistic, achieving a score of 3,200 cleared lines in a 20x10 Tetris board. The used featured are the height of each column, the differences in heights between two consecutive columns, the maximum height, and the number of holes.

Scherrer [19] revisits the work of Bertsekas & Ioffe [18] of the approximate version of λ-policy iteration method and challenges their paradoxical observation. More precisely, they wrote *"An interesting and somewhat paradoxical observation is that a high performance is achieved after relatively few policy iterations, but the performance gradually drops significantly. We have no explanation for this intriguing phenomenon, which occurred with all of the successful methods that we tried"*. He reproduced the issue and found out that it was a minor bug in the algorithmic implementation. With this fix, the previously mentioned algorithm achieves an approximate average score of 4,000 cleared lines.

Lagoudakis et al. [20] research two different algorithms. The Least-Squares Q-Learning, an extension of the Least-Squares Temporal Difference algorithm that learns a state-action value function much like Q-Learning. Due to limitations, including approximation biases and poor sample utilization, they eventually study the Least-Squares Policy Iteration (LSPI) algorithm, a model-free form to approximate policy iteration and efficiently uses training samples collected in any arbitrary manner. The features they use regarding Tetris are the maximum height in the current board, the number of holes, the sum of absolute height differences between adjacent columns, the mean height, the change of these quantities in the next step, the change in score, and a constant term. The LSPI achieves an average score between 1,000 and 3,000 per game.

Farias & Van Roy [21] examine the linear programming approach to approximate Dynamic Programming, which requires a linear program solution with a tractable number of variables but a potentially large number of constraints. They do so using randomized constraint sampling, a technique capable of producing reasonable solutions to the linear program in approximate Dynamic Programming. They formulate Tetris as a stochastic control problem with an intractable state-space. To approximate the cost-to-go function, they use a linear combination of the following 22 basis functions: Ten basis functions for the height of each column Nine basis functions for the absolute difference between heights of successive columns. One basis function for the maximum column height One basis function for the number of holes and a constant basis function The performance of the approach reaches 4,274 scores.

Bohm et al. [22] apply evolutionary algorithms to the game of Tetris. The program chooses the best move by rating possible future views of the board based on a rating function of a weighted sum of several sub ratings. The evolutionary algorithm is used to find the optimal weights. The rating function takes into account the current piece as well as the next one to be spawned, and it is mainly composed of more specific rating functions, which are: 1) The maximum height of the board 2) The number of holes 3) Connected holes which is the same as the number of holes but vertically connected unoccupied blocks count as one hole 4) The number of cleared lines 5) The differences between the highest and the lowest heights of the respective columns 6) The depth of the deepest well, i.e., empty blocks between columns with a width of one. 7) Sum of all wells 8) Landing height, i.e., the height that the last piece was placed 9) Number of occupied blocks 10) weighted blocks are the same as the number of occupied blocks but multiplied by the number of

rows they reside in. 11) Sum of all horizontal transitions between occupied/unoccupied blocks 12) Sum of all vertical transitions between occupied/unoccupied blocks They split the experiments using two rating functions. One using the features 1-6 and the other with 7-12. Their best results in a 20x10 board were 859,520.

2.9 Reinforcement Learning Definitions

Reinforcement learning can be comprehended through the concepts of environments, rewards, states, actions, and agents. Capital letters often denote sets of things, while lowercase letters denote a specific instance of the same thing; e.g., A is all possible actions, while a is a specific action contained in the set.

- Agent: An agent takes actions; for example, Super Mario navigating a video game or a drone making a delivery.
- Action (A): A is the set of all the possible moves the agent could make. An action has to be self-explanatory, but it must be noted that agents often choose from a list of possible and discreet actions. The list may include running left or right, jumping low or high, standing still, or crouching in video games. In the stock market, the list may include holding, buying, or selling any one of the available securities and their derivatives. Regarding handling aerial drones, alternatives would include many different velocities and accelerations in 3D space.
- Discount factor: The discount factor is multiplied by the future rewards discovered by the agent to dampen the effect of these rewards on the agent's action choice. It is designed to mark future rewards as less worthy than immediate rewards; i.e., it enforces short-term greed in the agent. They are usually expressed with the Greek lower-case letter gamma: γ. Say γ is 0.8, and there is a reward of 5 points after ten timesteps, the current value of that reward is

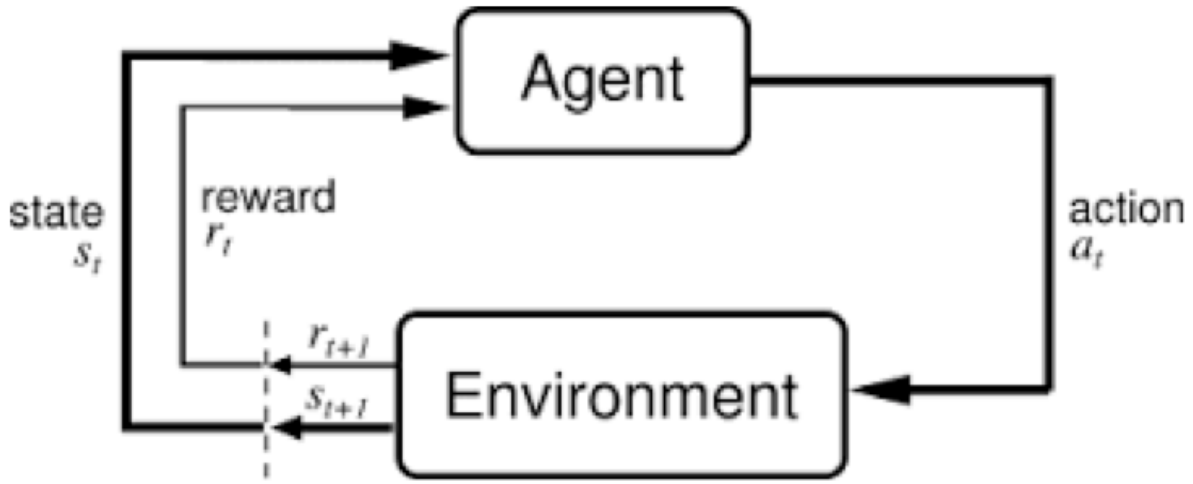
  $$0.8^{10} \times 5$$

  . A discount factor of 1 makes future rewards worth just as much as immediate rewards.
- Environment: The world in which the agent moves and reports feedback back to the agent. The environment takes the agent's present state and action as input and returns it as output, the reward, and its next state. From the agent's perspective, the environment could be the rules of society or the laws of physics that process people's actions and determine their consequences.
- State (S): A state is an immediate and concrete situation in which the agent finds itself, i.e., a particular place and moment, an instantaneous configuration that puts the agent with other vital things such as obstacles, tools, prizes, or enemies. It can be any future situation or the current reported back by the environment.
- Reward (R): A reward is the feedback metric by which we measure the failure or success of an agent's actions in a particular state. For example, in the Super Mario

video game, he is awarded points if Mario touches a coin. From any state, an agent can send output in the form of actions towards the environment. The environment returns the new state (which resulted from acting on the previous state) to the agent and rewards any. Rewards can be immediate or delayed. Thus, they effectively evaluate the agent's action.

- Policy (π): The policy an agent employs is the strategy to determine the following action based on the present state. It is a map between states and actions.
- Value (V): The expected long-term return with the discount included, as opposed to the short-term reward R. $V_\pi(s)$ under policy π is the expected long-term return of the current state. The further into the future a reward occurs, the lower it is its estimated value.
- Q-value (Q): Q-value is similar to value, except that it takes the current action a as an extra parameter. $Q_\pi(s, a)$ refers to the long-term return of action from the current state s under policy π. Thus, Q is a map between state-action pairs to rewards.

To sum up, environments are functions that take as input an action taken in the current state and output the next state and the reward; agents input the new state and reward and output the following action.



In the feedback loop depicted above, the subscripts denote the time steps t and t+1, with each referring to different states: the state at moments t and t+1.

Reinforcement learning judges actions by the produced results. It is goal-oriented and aims to learn sequences of actions that lead an agent to achieve its goal or maximize its value function. Here are some examples:

- In video games, the main goal is to finish the game by collecting the most points, so each additional point obtained through the course of the game is going to affect

the subsequent behavior of the agent; i.e., the agent may learn that it should touch coins, dodge meteors or shoot battleships to maximize its score.

- In the real world, the goal may be for a robot to travel a distance between two points A and B, and every centimeter the robot moves closer to point B could be counted as points.

2.10 State-Action Pairs Complex Probability Distributions of Reward

Reinforcement learning's goal is to select the best-known action for any provided state, which means the actions need to be ranked and assigned values relative to one another. What we are measuring is the value of state-action pairs since those actions are state-dependent.

We map state-action pairs to the values expected to be produced using the Q function. The Q function takes as input a state and action and outputs the probable reward.

Running the agent through lists of state-action pairs is the process of reinforcement learning, observing the resulting rewards, and adapting the Q function's predictions to those rewards until the best path is accurately predicted. This prediction is the policy.

Reinforcement learning can be interpreted as modeling a complex probability distribution of rewards concerning many state-action pairs. This is one of the reasons reinforcement learning is paired with a Markov decision process, a sampling method from a complex distribution to deduce its properties.

After a bit of time spent employing a Markov decision process to approximate the probability distribution of rewards over state-action pairs, a reinforcement learning algorithm may tend to repeat actions that lead to reward and cease to test alternatives. Thus, there is a tension between exploiting available rewards and continued exploration to discover new actions that also lead to victory. Much like oil companies have the dual function of pumping crude out of known oil fields while drilling for new reserves, in the same way, reinforcement learning algorithms can be programmed to both explore and exploit to varying degrees in order to assure that they do not pass over rewarding actions at the cost of known winners.

Reinforcement learning is an iterative process. In its most interesting applications, it does not begin by knowing which rewards state-action pairs will produce. Instead, it learns those relations by running through states repeatedly as athletes or musicians iterate through states to improve their performance.

2.11 Neural Networks and Deep Reinforcement Learning

Neural networks are function approximators, particularly useful in reinforcement learning when the state space or action space is too large to be wholly known.

A neural network can be used as a value function or policy function approximator. That is, neural networks can learn to map states to values or state-action pairs to Q values. Thus, rather than use a lookup table to store, index, and update all possible states and their values, which is impossible with enormous problems, we can train a neural network on samples from the action or state-space to learn and predict how valuable they are relative to our target in reinforcement learning.

As with all neural networks, they use coefficients to approximate the function relating inputs to outputs. Their learning consists of finding the fitting coefficients, or weights, by iteratively adjusting those weights along gradients that promise minor errors.

More specifically, Q maps state-action pairs to the highest combination of immediate reward with all future rewards that later actions in the trajectory might harvest. Here is the equation for the Q function, from Wikipedia:

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

After assigning values to the expected rewards, the Q function selects the state-action pair corresponding to the highest so-called Q value.

At the start of reinforcement learning, the neural network coefficients may be initialized stochastically or randomly. Then, using feedback from the environment, the neural network can use the difference between the expected reward and the ground-truth reward to adjust the weights and improve the interpretation of state-action pairs.

This feedback loop is similar to the backpropagation of error in supervised learning. Nevertheless, supervised learning begins with knowledge of the actual labels the neural network is trying to predict. Its goal is the creation of a model that maps different images to their corresponding names.

Reinforcement learning relies on the environment to feed it a scalar number in response to any new action. However, the rewards the environment returns can be delayed, varied, or affected by unknown variables, introducing noise to the feedback loop.

This leads us to a complete expression of the Q function, which considers the immediate rewards produced by action and the delayed rewards returned several timesteps deeper in the sequence.

As human beings, the Q function is recursive. Thus, just as calling the wetware method human() contains within it another method human(), of which we are all the fruit, calling the Q function on a given state-action pair requires us to call a nested Q function to predict the value of the following state, which in turn depends on the Q function of the state after that, and so forth.

Chapter 3

Deep Learning Neural Networks

3.1 Neural Networks Definition

An artificial Neural Network is an efficient computing system whose central theme is analogous to biological neural networks. Artificial Neural Networks are also named "parallel distributed processing systems", "artificial neural systems," or "connectionist systems." Artificial Neural Network acquires a vast collection of units that are interconnected in a pattern that allows communication between them. Also, these units are simple processors which operate in parallel.

3.2 Biological Neuron

A nerve cell neuron is a biological cell that processes information. There is a vast number of neurons, approximately 10^{11} with numerous interconnections, approximately 10^{15}.

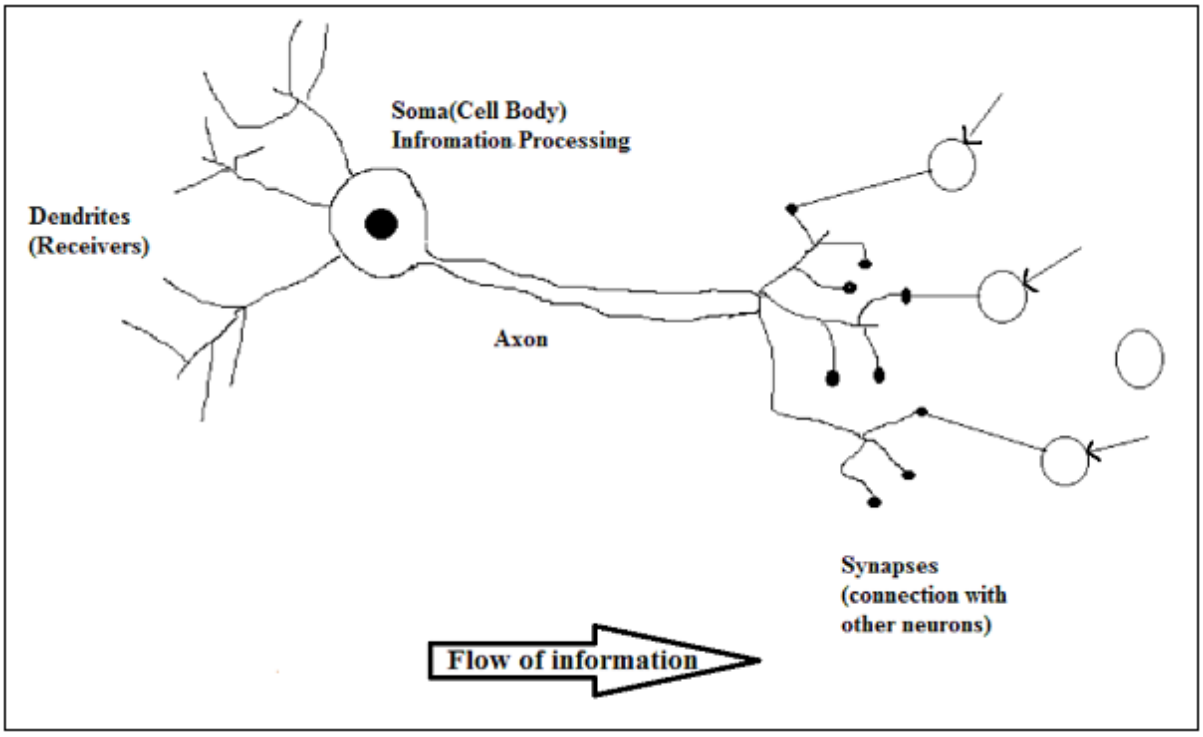


Figure 3.1: Biological Neuron

As shown in the above Figure 3.1, a typical neuron consists of the below-mentioned four parts, with the help of which we can explain its working.

- Dendrites They are tree-like branches responsible for receiving the information from other connected neurons. In other words, they are like the ears of the neuron.
- Soma It is the neuron's cell body responsibility for processing information they have received from dendrites.
- Axon It is just like a cable through which neurons send the information.
- Synapses It is the connection between the axon and other neuron dendrites.

3.3 Model of Artificial Neural Network

Figure 3.2 represents the general model of ANN followed by its processing.

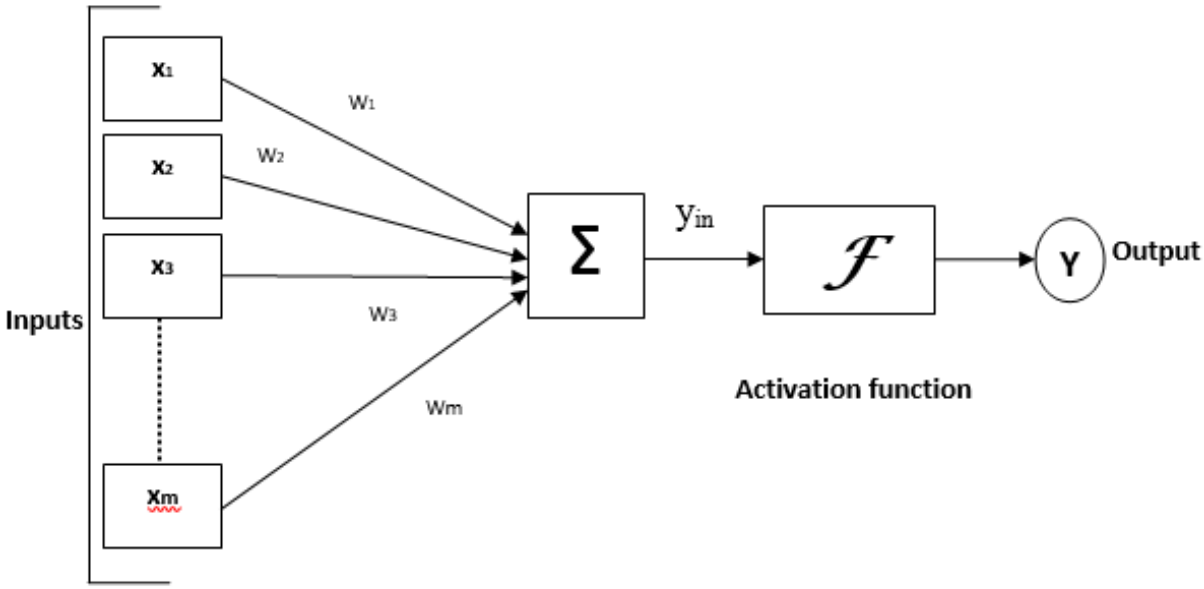


Figure 3.2: Artificial Neural Network Model

For the above general model of an artificial neural network, the net input can be calculated as follows

$$y_{in} = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + ...x_m \cdot w_m$$

The output is calculated by applying the activation function over the network input.

$$Y = F(y_{in})$$

3.4 Feedforward Network

It is a network having processing nodes in layers, and all the nodes of a layer are connected to the nodes of the previous layer. Each connection has a weight. There is no feedback loop, meaning information flows in one direction, from input to output. There are two types of feedforward networks

- Single layer feedforward network This neural network has only one weighted layer. In other words, the input layer is fully connected to the output layer (Figure 3.3).
- Multilayer feedforward network This neural network consists of more than one weighted layer. Intermediate layers are called hidden layers (Figure 3.4).

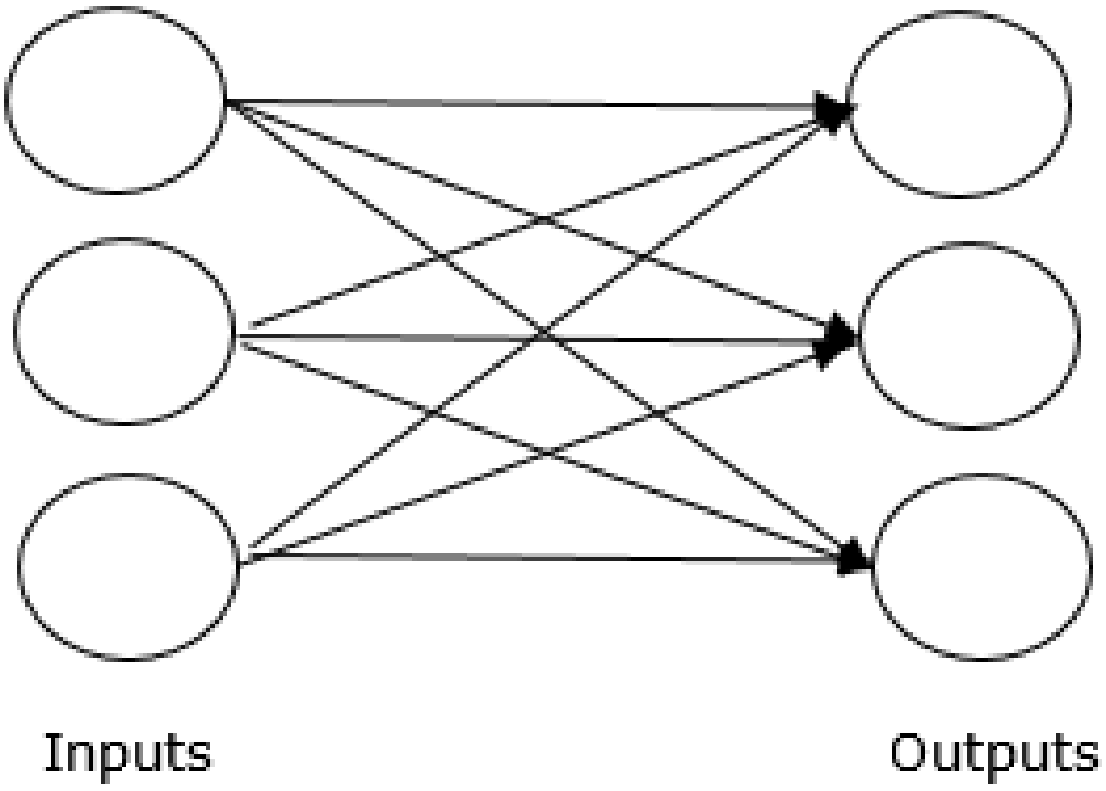


Figure 3.3: Single layer feedforward network

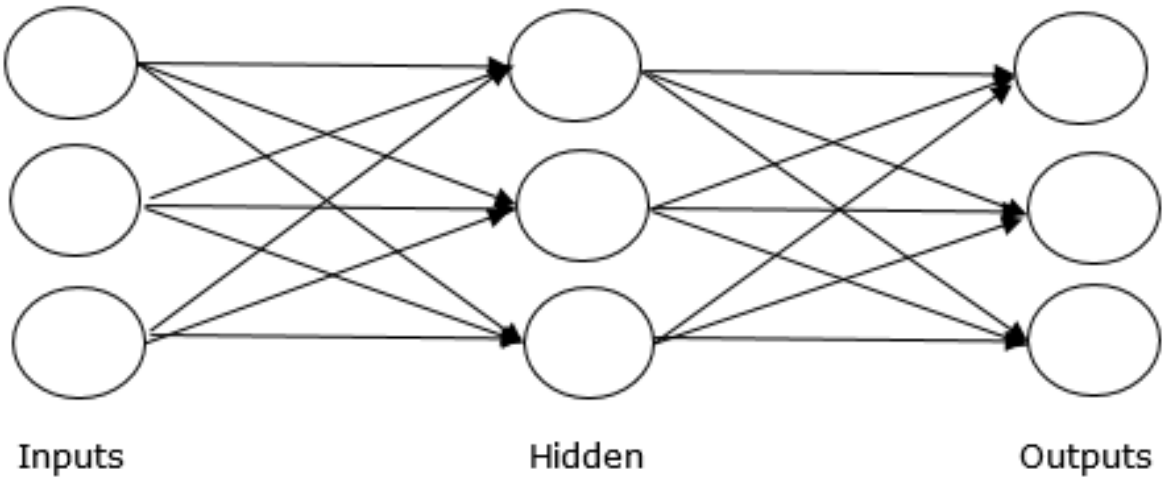


Figure 3.4: Multilayer feedforward network

3.5 Activation Functions

It may be defined as the effort applied over the input to obtain an exact output. In neural networks, we can also apply activation functions over the input to get the exact output. The followings are some activation functions of interest.

3.5.1 Linear Activation Function

Also called the identity function because it performs no input editing. It can be defined as

$$F(x) = x$$

Sigmoid Activation Function

Sigmoid outputs values between 0 and 1. It is positive in nature and always bounded, which means its output cannot be less than 0 and more than 1. Furthermore, sigmoid is increasing in nature, which means more the input higher would be the output. It can be defined as

$$F(x) = \frac{1}{1 + exp(-x)}$$

3.5.2 ReLU (Rectified Linear Unit) Activation Function

The rectifier or ReLU (Rectified Linear Unit) activation function is an activation function defined as the positive part of its argument:

$$F(x) = max(0, x)$$

3.6 Perceptron

Perceptron is the primary operational unit of artificial neural networks. It employs supervised learning rule and can classify the data into two classes.

Perceptron consists of a single neuron with a non-trivial number of inputs having adjustable weights, but the neuron's output is 0 or 1 depending on the threshold. It also has a bias whose weight is always 1. The following figure shows a perceptron representation

Perceptron has the following three basic elements:

- Links It would have a set of connection links, which carries a weight including a bias always having weight 1.
- Adder It adds the input after they are multiplied with their respective weights.
- Activation function It limits the output of a neuron.

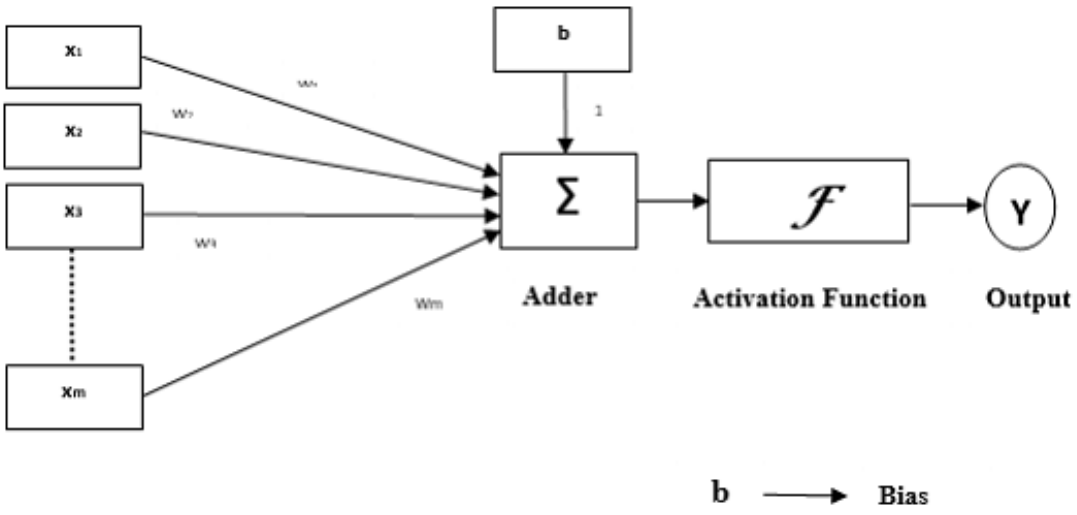


Figure 3.5: Perceptron

3.7 Learning Rate

The learning rate is a hyperparameter that controls how much the model should change in response to the estimated error each time the weights are updated. The learning rate

selection is challenging as too small values may result in a lengthy training process that could get stuck. In contrast, on the other hand, a value too large may result in learning a sub-optimal set of weights very quickly or an unstable training process.

3.8 Weights

Weight is the parameter in a neural network that transforms input data within the network's layers. A neural network is a set of neurons. Within each neuron, there is a set of inputs, a weight, and a bias value. As an input enters the neuron, it gets multiplied by the weight value, and the resulting output is either passed to the next layer or observed. Often the weights of a neural network live within the hidden layers of the network.

3.9 Back Propagation

Back Propagation Neural Network is a multilayer neural network consisting of the input layer, at least one hidden layer, and the output layer. As its name suggests, back-propagating will take place in this network. The error calculated at the output layer propagated back towards the input layer by comparing the predicted output and the actual output.

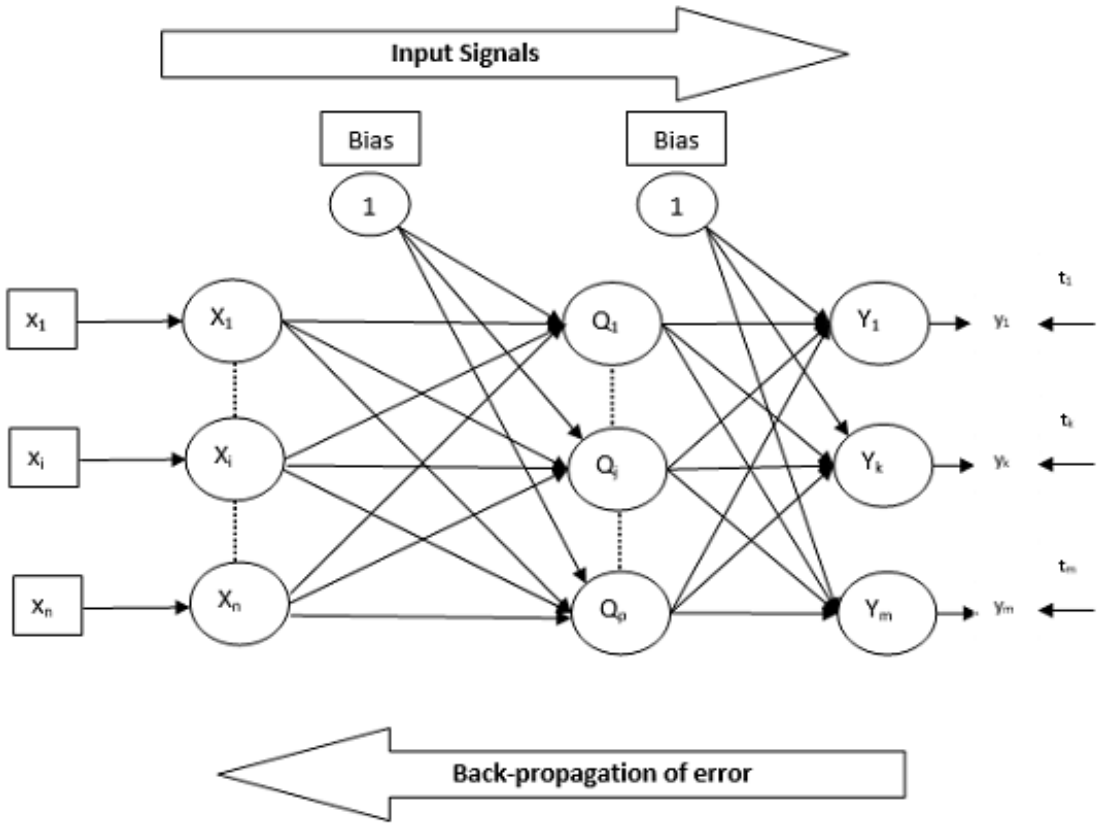


Figure 3.6: Back Propagation

Every neuron is connected with another neuron through a connection link. Every connection link is associated with a weight that contains information about the input signal. This is the most important information for neurons to solve a specific problem because the weight usually excites or inhibits the signal that is being communicated. In addition, each neuron has an internal state, which is called an activation signal. Output signals, produced after combining the input signals and activation rule, may be sent to other units.

3.10 Stochastic Gradient Descent

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks.

The standard gradient descent algorithm updates the parameters θ of the objective $J(\theta)$

as,

$$\theta = \theta - \alpha \nabla E[J(\theta)]$$

where the expectation is approximated by evaluating the gradient and cost over the full training set. Stochastic Gradient Descent (SGD) does away with the expectation in the update and computes the gradient of the parameters using only a single or a few training examples. The new update is given by,

$$\theta = \theta - \alpha \nabla J(\theta; x(i), y(i))$$

with a pair $(x(i), y(i))$ from the training set.

Generally, in SGD each parameter is updated by computing with respect to a minibatch or a few training examples as opposed to a single example. There are two reasons for this: first, it reduces the variance in the parameter update and leads to more stable convergence and secondly it allows the computation to take advantage of optimized matrix operations that can be used in a vectorized computation of the gradient and const. A common minibatch size is 256, although the optimal size of the minibatch can vary for different applications and architectures.

In SGD, the learning rate α is typically smaller than a corresponding learning rate in batch gradient descent due to the much more variance in the update. Choosing the best learning rate and schedule (i.e., changing the value of the learning rate as learning progresses) is considered a difficult task. One common method that works well in practice is using a small enough constant learning rate that provides stable convergence in the initial epoch (complete pass through the training set) or two of training and then halves the learning rate's value as convergence slows down. A better approach is evaluating a held-out set after each epoch and anneal the learning rate when the change between epochs is below a small threshold. This tends to give good convergence to a local optimum. Another commonly used schedule is to reduce the learning rate on each iteration t as $ab + t$ where a and b dictate the initial learning rate and when the annealing begins. More advanced

methods include using a backtracking line search to find the optimal update.

One final but essential point regarding Stochastic Gradient Descent is how the data are presented to the algorithm. If the data is given in exact order, this may bias the gradient and lead to poor convergence. Generally, an excellent method to avoid this is to shuffle the data before each training epoch randomly.

3.11 Adam Optimization Algorithm

The Adam optimization algorithm extends stochastic gradient descent that has recently seen wider adoption for deep learning applications in natural language processing or computer vision. Adam is an optimization algorithm that can be used in the place of the classical stochastic gradient descent procedure for updating network weights iterative based on the training data.

Adam is different from classical stochastic gradient descent.

Stochastic gradient descent keeps a single learning rate (denoted as alpha) for all weight updates, and the learning rate is not changing during training.

A learning rate is kept for each network weight and separately adapted as learning unwinds. The method computes individual learning rates for different parameters from estimates of first and second moments of the gradients.

The authors explain Adam as the combination of the advantages of two other extensions of stochastic gradient descent. Particularly:

Adaptive Gradient Algorithm (AdaGrad) which maintains a per-parameter learning rate that improves the performance on problems containing sparse gradients (e.g.,computer vision and natural language problems). Root Mean Square Propagation (RMSProp) also maintains per-parameter learning rates, which adapt based on the average of current magnitudes of the gradients for the weight. This means the algorithm performs well on online and non-stationary problems (e.g., noisy).

Adam connects the benefits of both AdaGrad and RMSProp.

Adam uses the average of the second moments of the gradients other than adapting the parameter learning rates based on the mean as in RMSProp.

Notably, the algorithm computes the exponential moving average of the gradient and the squared gradient, while the parameters beta1 and beta2 control the decreasing rates of these moving averages.

The starting value of the moving averages and beta1 and beta2 values, which are close to 1.0, result in a bias of moment estimates towards zero. This bias is overcome by first calculating the biased estimates before then calculating bias-corrected estimates.

3.12 Deep Reinforcement Learning

Deep reinforcement learning is the combination of artificial neural networks and reinforcement learning which helps software agents learn how to accomplish their goals. In other words, it unites function approximation and target optimization, mapping states and actions to the rewards they lead to.

While neural networks are responsible for recent artificial intelligence breakthroughs in domains like machine translation, compute vision and time series prediction – they also combine with reinforcement learning algorithms in the creation of something astounding like Deepmind's AlphaGo, an algorithm that beat the world champions of the Go board game.

Reinforcement learning attributes to goal-oriented algorithms that learn ways of achieving a complex objective or the maximization along a particular dimension after many steps; e.g., they can maximize won points in a game after many moves. Reinforcement learning algorithms may start from a clean slate, and under the proper conditions, achieve superhuman performance. Much like a pet incentivized by treats and scolding, these algorithms are punished when they take the wrong decisions and rewarded if they make the right ones.

Reinforcement algorithms that use deep neural networks can beat human experts playing numerous Atari video games [23], Starcraft II [24] and Dota-2 [25]. While that may sound quite trivial to non-gamers, it is actually a vast improvement compared to reinforcement learning's historical accomplishments, and the state of the art is rapidly progressing.

Reinforcement learning solves the hard problem of immediate actions correlation with the production of the delayed outcomes. Like humans, reinforcement learning algorithms usually have to wait to see the weight of their decisions. They operate in a environment with delayed returns, where it can be hard to understand which action leads to which outcome after many time steps.

Reinforcement learning algorithms are slowly and gradually performing better and better in real-life environments when choosing from an arbitrary number of possible actions, instead of the limited options of a repeatable video game. In other words, they are beginning to achieve goals in the real world.

Chapter 4

Design & Specifications

In this chapter, we reduce the state space of Tetris by adopting assumptions and discuss the possible consequences of each assumption. We then design the reinforcement learning agent and consider the processes it requires. Finally, we end the chapter by considering the structure of the whole application.

4.1 Redesigning the Tetris State Space

Traditional reinforcement learning uses a tabular value function that associates a value with every state. Thus, the primary design consideration is how the Tetris state space can be reduced without discarding pertinent information. Since the full Tetris well, shown in Figure 2.1, has dimensions twenty blocks deep by ten blocks wide, there are two hundred block positions in the well that can be either occupied or empty.

Assumption 1 The position of each block on screen is not a consideration that is factored into every move by a human player. We limit ourselves to simply considering the sum of heights of each column.

Assumption 2 The height of each column is pretty irrelevant except perhaps when the height of a column starts to approach the top of the well. The importance lies in the relationship between successive columns rather than in their isolated heights.

Assumption 3 The height of each column is not the only consideration a human player has to think. For example, holes that form between structures are the natural enemy of Tetris. Therefore, one crucial aspect of the game is the reduction of these holes.

Assumption 4 In order to avoid losing the game, it is essential to clear lines as the game progresses. Having in mind that clearing more rows at once awards more points, the possible number of row clearance at each step is important. This maxes out at four, due to I piece's (Figure 2.2) height, which is four.

4.2 The Structure of a Reinforcement Learning Agent

We set out to create an agent that functions within the reduced state representation developed above. We decided to compare three algorithms where the agent can consider the tetromino currently allocated to it in the course of each move. The agent's behaviour

can be separated into distinct processes.

- Discover transitions
- Choose amongst transitions using exploration policy
- Update value function

Each of these processes is now discussed in depth in the following subsections

4.3 The Discovery of Transitions

This method discovers the transitions available from the current state. The agent makes use of a conceptual game that exists purely for its benefit, isolating any conceptual manipulations from the full game. The agent copies the block formation from the real Tetris well into its conceptual well before performing each of the possible transitions with the current tetromino. Each transition is defined by four parameters. These are the number of translations and rotations, the resulting reward and the value of the resulting state. Every unique transition is added to a list of possible transitions.

4.4 Exploring Amongst Transitions

ϵ-greedy is implemented as competing exploration policy. ϵ-greedy is a deviation from the greedy policy and is implemented within the greedy method by giving the agent a fixed probability of choosing randomly amongst the available transitions. There are therefore two methods that accept a range of possible transitions and return a single transition. Optimistic exploration is achieved by initialising the value function with values slightly larger than the largest anticipated value. This value is easily determined when dealing with purely negative rewards, since all states will have negative values, and therefore zero is an optimistic value. When dealing with positive rewards, the easiest approach is to set the agent to explore for a large period of time, look at the predicted values and adopt a value slightly larger than the largest value for the starting values.

4.5 Update the Value Function

The transition selected by the exploration policy is taken in the full game. The results of this transition are used in updating the agent's value function. The update functions for our three approaches are shown by equations in algorithms 1, 2 and 3. The update method accepts the current index, destination index and a reward. In the SARSA agent, this reward is interpreted as the reward associated with taking the action from the current state. In Q-Learning, the reward defers from SARSA in that it also considers the rewards of taking an action in the next step. In Memory-Based Learning, we train the agent at the end of each episode by replaying a batch from the previous episodes' moments. In

implementing those agents, the state-value table had to be extended to contain every transition off each state. The number of transitions is dictated by the number of different tetrominoes, the number of translations and the number of different orientations. Each tetromino accounts for a specific set of translations and orientations based on its physical nature. For example, the O piece (Figure 2.5) can move up to four blocks to the right or left, and have no rotations. In other words, the number of the available translations for O piece is 8. This drastically decreases the state space of the game as it removes a great deal of redundant information depending on the tetromino set used.

4.6 Application Design

We designed a Tetris game from first principles in order to have complete control over the structure of the game and familiarise ourselves with the required methods. The application can be readily divided up into the following logical classes

- The Game Window
- The Graphics Manager
- The Game Controller
- The Tetris class
- The Tetrominoes List
- The Tetris player

The game window is the display that all graphics are rendered. The graphics manager is responsible for sending to the game window the appropriate information for the game to take place. The game controller has a number of responsibilities listed below:

- Initialize score
- Track game status (playing, game over)
- Creating random tetromino sequences for future spawns
- Spawning a piece
- Moving a piece vertically and horizontally
- Check collisions with walls and other already placed tetrominoes
- Count number of holes
- Count bumpiness (sum of the differences of heights between consecutive columns)
- Count maximum height

The Tetris class, requires a game controller and a graphics manager to function. It is the core unit that connects everything together. Other than using the game controller to ensure the execution of the game, it is responsible for informing the dependent entities about the board status, the current reward and the result of applying one step (one action) to the environment.

The Tetris player, which can either be instantiated as an agent or human player, is an object that uses the exposed methods of Tetris class in order to access the basic controls of the game. if it is a human, a piece can be moved using the arrow keys. On the other hand, if it is an agent, it automatically moves based on the decisions it takes.

The tetrominoes list contain the necessary information about the piece that the game controller needs to know in order to apply the respective mechanics, such as rotational manipulation and translation. All tetromino transitions which occur within the board are checked and performed in the Game Controller. The Game Controller and the artificial agent (Tetris player) are all objects that will need to change in the course of the investigation. This structuring allows for seamless swapping between different game definitions and artificial agents. As long as the agent implements the correct interface, the theory guiding the actions of the agent can subscribe to any artificial intelligence method. This follows the reasoning, outlined by the strategy design pattern [26], that competing algorithms should implement a common interface and therefore be seamlessly interchangeable. We would expect any object-orientated Tetris game to deal with a large number of tetromino objects, and the performance penalty introduced by instantiating large numbers of simple objects warrants consideration. This is addressed by conventional design patterns and corresponds to a fly-weight design pattern, as discussed in Gamma et al. [26]. Rather than having the game continually recreating individual tetrominoes within the list of available tetrominoes, it is preferable to create every possible tetromino once and subsequently pass out a reference to the relevant tetromino. This optimisation is piece-specific and is implemented in the Tetris class by instantiating a dictionary mapping each of the tetrominoes to the corresponding range of orientations and translations. The first time a tetromino is assigned a rotation or translation, a respective value is accessed in this dictionary.

Chapter 5

Implementation

5.1 Environment

The simulation of the Tetris environment that is used is a custom implementation that obeys the Classic Tetris World Tournament rules, except that pieces can only be rotated clock-wise and not in both directions. In addition, the training agent decides the move it will follow each time a piece is spawned, and this move cannot be changed until the spawn of the next one.

As epoch, we assume one whole game, i.e., from the very first piece until the game is over. Every epoch, the current state is calculated as soon as a piece is spawned. After that, all following possible states are generated, based on the current's piece transforms and rotations. These following states are fed into the agent who chooses to take an action (move) in random or based on the most rewarding next state. The action is applied to the Tetromino, which in turn ends up falling on the board, and the next piece is spawned. This cycle continues until a game ends, triggering a reset and a new game starts.

In Tetris, an agent must take specific action sequences to receive a reward by clearing lines, while there are a lot of others that can lead to no rewards. If the latter happens frequently, more exploration is needed before a reward is found resulting in sparse rewards and impossible learning. This is avoided by reducing the placing of one tetromino from many actions to one.

5.2 Q-Learning

Q-Learning is an Off-Policy algorithm for Temporal Difference learning. It can be proven that given sufficient training under any ϵ-soft policy, the algorithm converges with probability 1 to a close approximation of the action-value function for an arbitrary target policy. Q-Learning learns the optimal policy even when actions are selected according to a more

exploratory or even random policy [27]. The procedural form of the algorithm is:

Algorithm 1: Q-Learning

Input: the policy π to be evaluated
Where: $Q()$ is the Q function
s is some state
S is the set of all states
a is an action
r is the reward given by taking action a
s' is the next state
a' is the next action
α is the learning rate
γ is the discount rate
ϵ is given by (5.2)
Initialize $Q(s, a) = 0, \forall s \in S$
while *true* **do**
  Initialize s as starting state
  while s *is not terminal state* **do**
    Choose a from s using policy derived from Q Take action a, observe r and next state s'
    Choose a' from s' using policy derived from Q
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{\alpha} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  end
end

This procedural approach can be translated into steps as follows:

- Initialize the Q-values table, Q(s, a)
- Observe the current state, s
- Choose an action, a, for that state based on the given policy
- Take the action, and observe the reward, R, as well as the new state, s'
- Update the Q-value for the state using the observed reward and the maximum reward possible for the next state
- Set the state to the new state, and repeat the process until a terminal state is reached

5.3 SARSA

The SARSA algorithm is an On-Policy algorithm for TD-Learning. The major difference between it and Q-Learning, is that the maximum reward for the next state is not necessarily used for updating the Q-values. Instead, a new action, and therefore reward, is

selected using the same policy that determined the original action. The name SARSA actually comes from the fact that the updates are done using the quintuple $Q(s, a, r, s', a')$. Where: s, a are the original state and action, r is the reward observed in the following state and s', a' are the new state-action pair. The procedural form of SARSA algorithm is comparable to that of Q-Learning [27]:

Algorithm 2: SARSA

Input: the policy π to be evaluated
Where: $Q()$ is the Q function
s is some state
S is the set of all states
a is an action
r is the reward given by taking action a
s' is the next state a' is the next action α is the learning rate
γ is the discount rate
ϵ is given by (5.2)
Initialize $Q(s, a) = 0, \forall s \in S$
while *true* **do**
  Initialize s as starting state
  Choose a from s using policy derived from Q
  while s *is not terminal state* **do**
    Take action a, observe r and next state s'
    Choose a' from s' using policy derived from Q
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
    $a \leftarrow a'$
  end
end

5.4 Memory Based Learning

Algorithm 3: Memory Based Learning

Input: the policy π to be evaluated
Where: $Q()$ is the Q function
s is some state
S is the set of all states
a is an action
r is the reward given by taking action A
α is the learning rate
γ is the discount rate
ϵ is given by (5.2)
$minSample$ is the minimum replay sample required to start training
$replayMemory$ is the replay memory
$batchSize$ is the number of samples to choose from replay memory
Initialize $Q(s, a) = 0, \forall s \in S$
while *true* **do**
  Initialize s as starting state
  while s *is not terminal state* **do**
    Choose a from s using policy derived from Q
    Take action a, observe r and next state s'
    $replayMemory.push((s, s', R))$
  end
  if $replayMemory.length \geq minSample$ **then**
    $batch \leftarrow replayMemory.sample(batchSize)$
    for $(s, s', R) \in batch$ **do**
      $Q(s, a) \leftarrow Q(s, a) + a[r + \gamma Q(s', a')]$
    end
  end
end

Memory Based is an algorithm thatthat uses a replay memory. Every generated state of the environment is inserted into a fixed size memory. This memory is a FIFO list. The goal of the agent is to maximize its rewards by constructing some policy. The policy is responsible for deciding the action that moves the environment from one state to another. The memory-based learning method tries to approximate a Q function, $Q(s, a)$, which estimates the value of the following states. In other words, $Q(s, a)$, provides information on how much value the state s will yield in the future. Unlike episodic or step-by-step RL method, where the Q function is improved per episode or per step accordingly, in the memory-based method, the value function starts improving after a minimum amount of state sample has been gathered. The policy is ϵ-greedy, meaning that it will either take highest rewarding action or a random action depending on ϵ.

5.5 Rewards and Exploration

Each block placement acquires 1 point of reward. Each lose grants a -2 reward. In each move, there can be a simultaneous line clearance $c \in [0, 4]$. The reward given based on c can be seen in equation (5.1).

$$r \leftarrow r + 10c^2 \tag{5.1}$$

The squared increase in rewards based online clearance, has shown that it helps the agent look for clearing more lines at once rather thanthan clearing them one by one, which is closer to a human's play style.

Exploration is an important part in reinforcement learning. The exploration rate is given by the formula (5.2), where d is the decay factor and E is the number of running epoch, meaning that ϵ slowly decays to 0 in the first 1500 episodes.

$$\epsilon = 1 - d/1500 \tag{5.2}$$

$$d = min(E, 1500) \tag{5.3}$$

This causes the agent to take random actions with decreasing rate, which is standard in reinforcement learning, as initially the agent has no experience and needs to explore more frequently while after some episodes it needs to exploit the knowledge it acquired through exploration. This is very similar to how humans learn tasks by trying strategies when faced with something new, but use experience on known tasks.

The policy relies on the value function $Q(s, a)$ which is the expected reward for each state. While for smaller problem $Q(s, a)$ can be represented by a table with each item representing the expected value of state-action pairs, in the case of Tetris where the set of states is huge, it is not feasible. That's why an approximator is required. The role of the approximator takes a Deep Neural Network.

5.6 State Representation

The state representation that was used i a 1D vector with four features:

- **Line Clearance**: The number of cleared lines if found in this state.
- **Holes**: The number of empty blocks with at least one piece block above them.
- **Total Bumpiness**: Sum of the differences of heights between pairs of consecutive columns.
- **Total Height**: Sum of heights of each column.

5.7 Deep Neural Network Agent

The agent used is a fully connected deep neural network.

Initially, we deployed a smaller neural network with the following characteristics:

- **Input Layer**: The input layer accepting the state representation.
- **Hidden Layer 1**: The first hidden layer consists of 32 neurons and uses the ReLU activation function.
- **Hidden Layer 2**: The second hidden layer consists of 32 neurons and uses the ReLU activation function.
- **Output Layer**: The output layers consists of 1 neuron which represents the expected reward given the input state. The activation function is linear.
- **Loss Function**: Mean Squared Error is used as loss function.
- **Optimizer**: Adam optimizer is used.
- **Learning Rate**: Learning rate is set to 0.001.

But we would like to compare the results of the afore mentioned model to the results of a deeper network containing:

- **Input Layer**: The input layer accepting the state representation.
- **Hidden Layer 1**: The first hidden layer consists of 32 neurons and uses the ReLU activation function.
- **Hidden Layer 2**: The second hidden layer consists of 64 neurons and uses the ReLU activation function.
- **Hidden Layer 3**: The first hidden layer consists of 64 neurons and uses the ReLU activation function.
- **Hidden Layer 4**: The second hidden layer consists of 32 neurons and uses the ReLU activation function.
- **Output Layer**: The output layers consists of 1 neuron which represents the expected reward given the input state. The activation function is linear.
- **Loss Function**: Mean Squared Error is used as loss function.
- **Optimizer**: Adam optimizer is used.
- **Learning Rate**: Learning rate is set to 0.001.

Chapter 6

Evaluation

6.1 SARSA Results

Figure 6.1 depicts the results of SARSA algorithm. It is clear that SARSA is not an algorithm that converges to this specific domain. After 3000 episodes and two deep learning agents of different depths, it could not learn how to play the game by clearing lines. We believe this happens due to the fact that SARSA selects actions a and a' on the same step using the given ϵ-greedy policy which increases randomness. In a game with a vast space-state, like Tetris, it is usual for an approach to find difficulties in approximating a large percentage of the space-state.

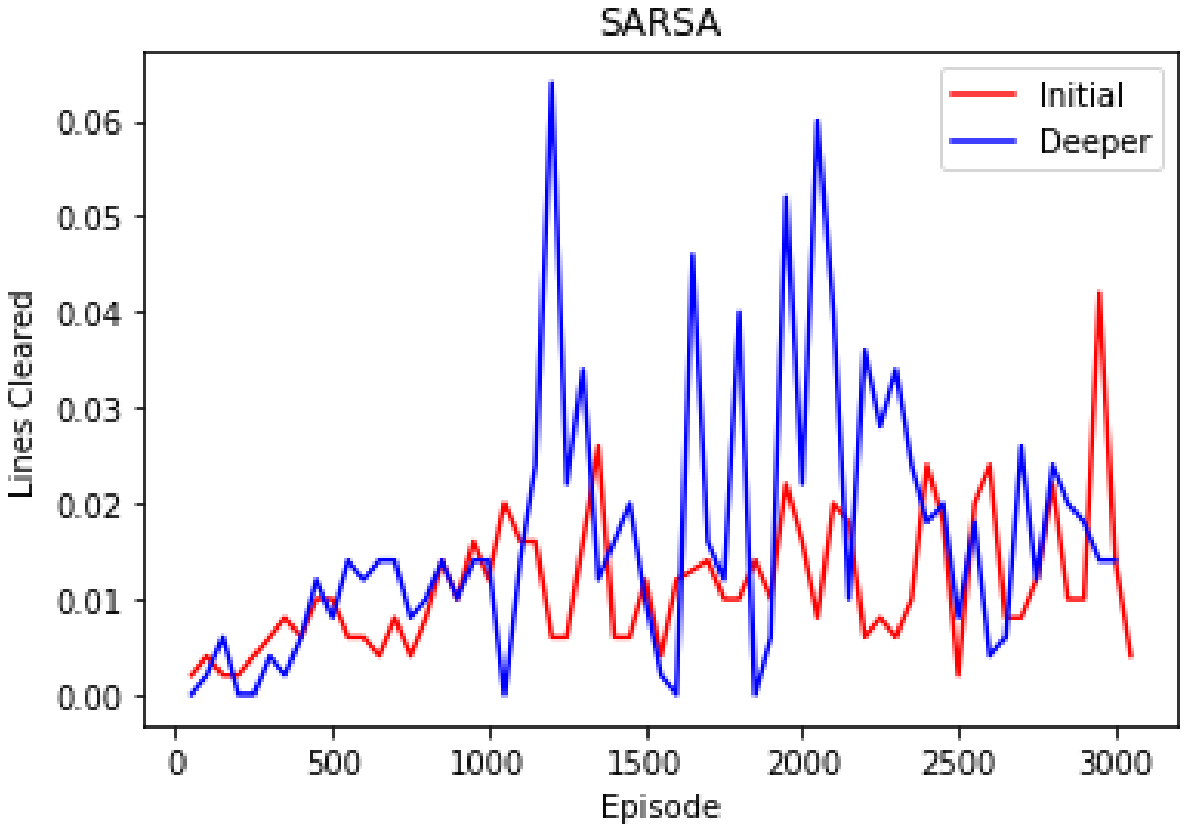


Figure 6.1: SARSA Results

6.2 QLearning Results

Figure 6.2 shows the results of QLearning approach. In contrast with SARSA, QLearning, on the initial agent, seems to be able to learn very little from the game. Max cleared lines of 8 during the exploration period can be interpreted as a random event to happen. Between 800 and 2800 episodes, we observe no actual improvement. After 2800 episode an increased performance is seen, but due to the maximum set limit of 3000 episodes we can not be sure that there could be a continues improvement towards optimum.

Is is also clear, that when training on the deeper neural network, the agent wasn't able to learn anything about the game. Perhaps deeper networks require adjustments and tweaks to the hyperparameters in order to find the golden rule.

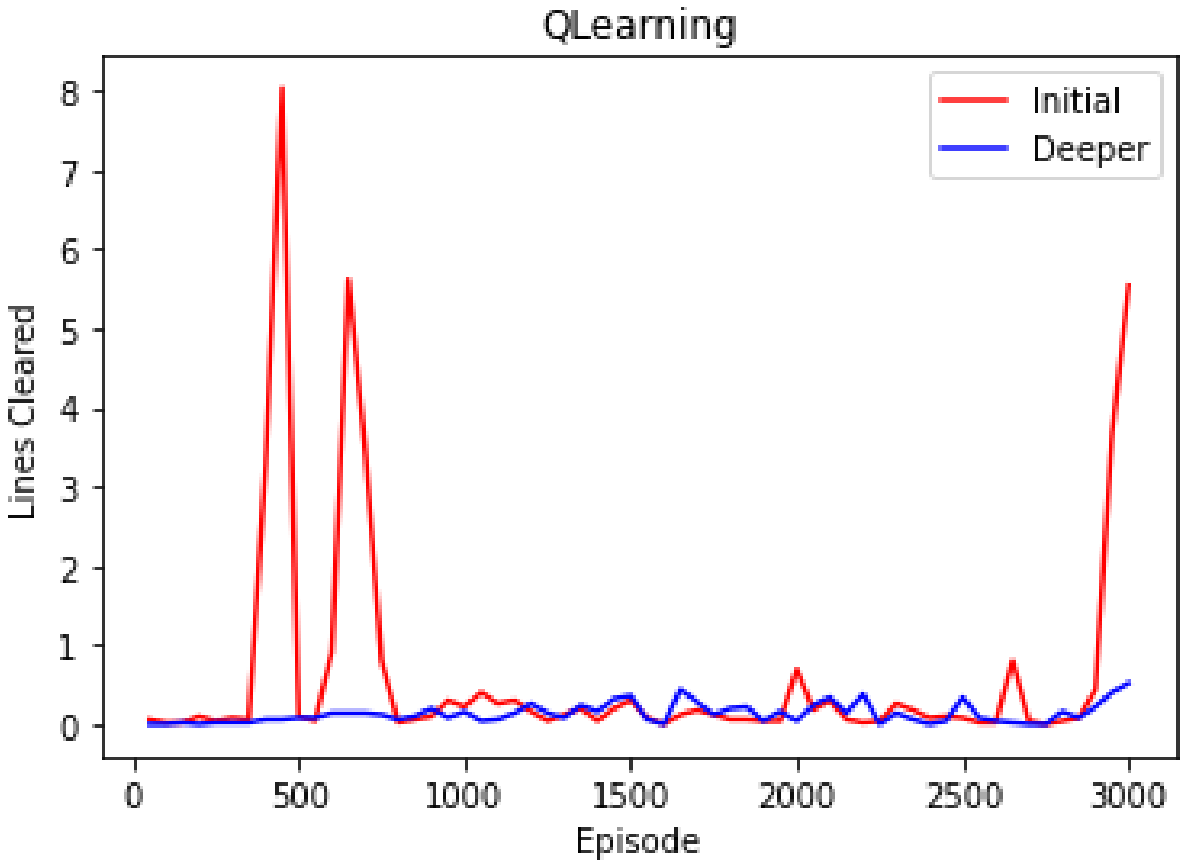


Figure 6.2: QLearning Results

6.3 Memory Based Results

At Figure 6.3 the results of Memory Based Learning can be seen. It is by far the most promising approach compared to SARSA and QLearning. In the graph 6.3, we observe that after 1500 episodes, the point where exploration stops, there's an increase in the performance of the agent reaching approximately the number of 35 cleared lines on the initial neural network. We strongly believe that this massive difference compared to other

algorithms is due to the fact that Memory Based agent uses a memory of past experiences for training. This seems to help because the agent can replay the best moves from a set of bad or good experiences. This is close to how we, humans, operate. By knowing the best action of our previous states, we can repeat and improve our behavior. This is something that lacks from previous approaches of SARSA and QLearning.

When training on the deeper network, we also observe an increase in cleared lines between 1500 and 2000 episodes. The maximum number is 20. After that the performance starts dropping. We believe this happens due hyperparameter tuning. Due to the lack of computer power we could not experiment on more setups and compare results with different hyperparameters.

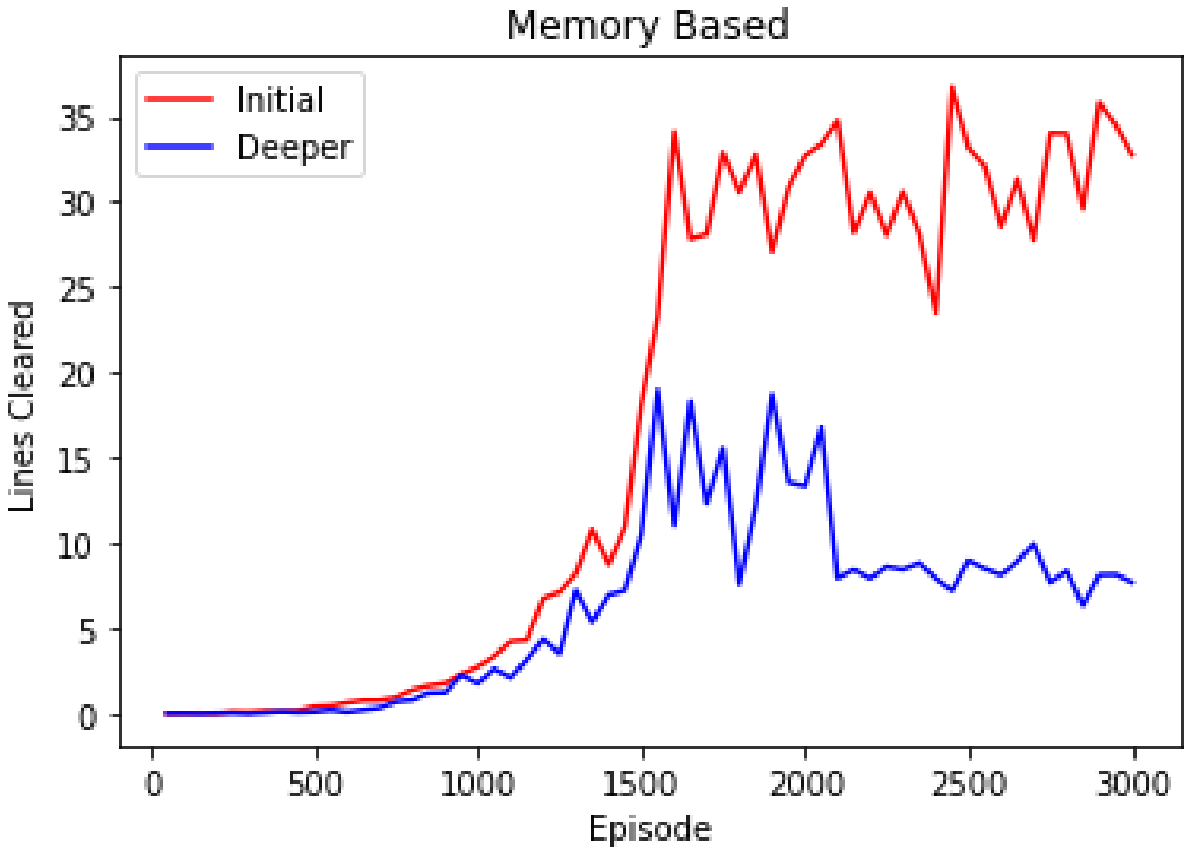


Figure 6.3: Memory Based Results

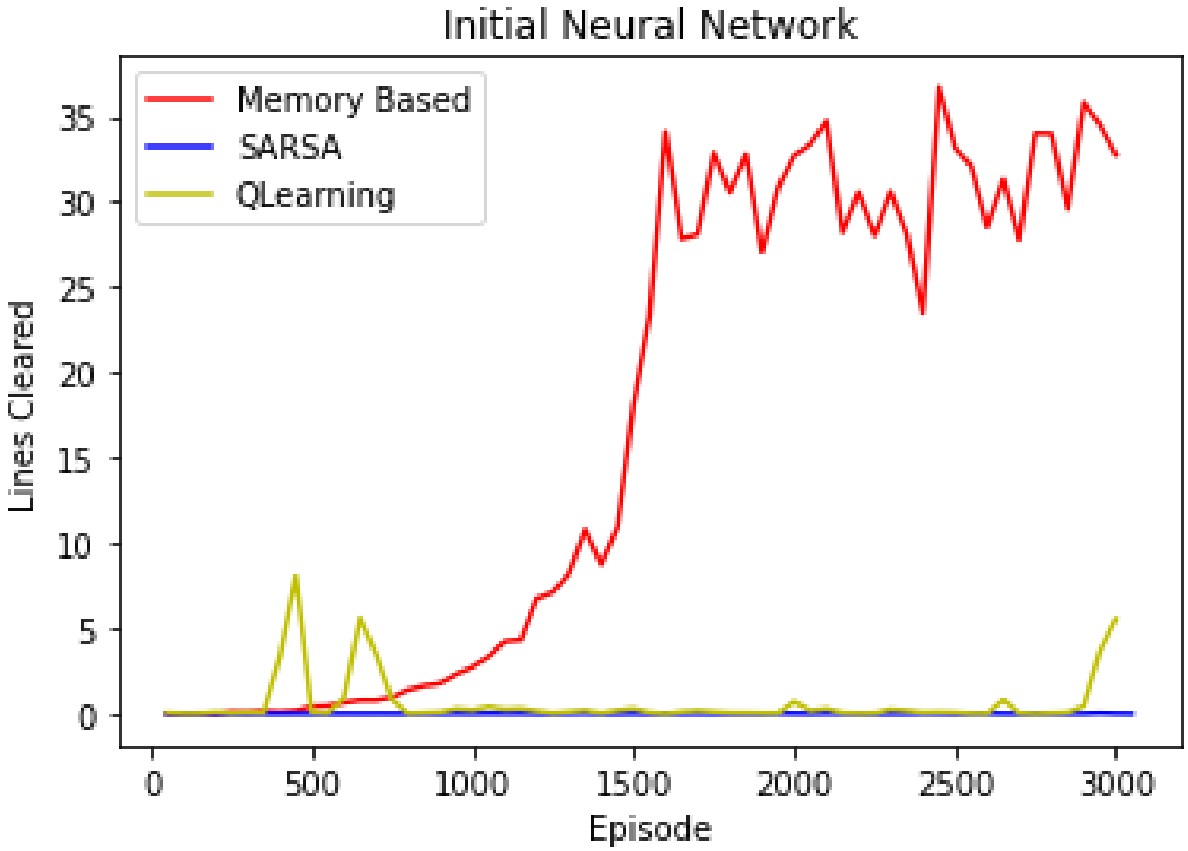


Figure 6.4: Initial Neural Network Comparison

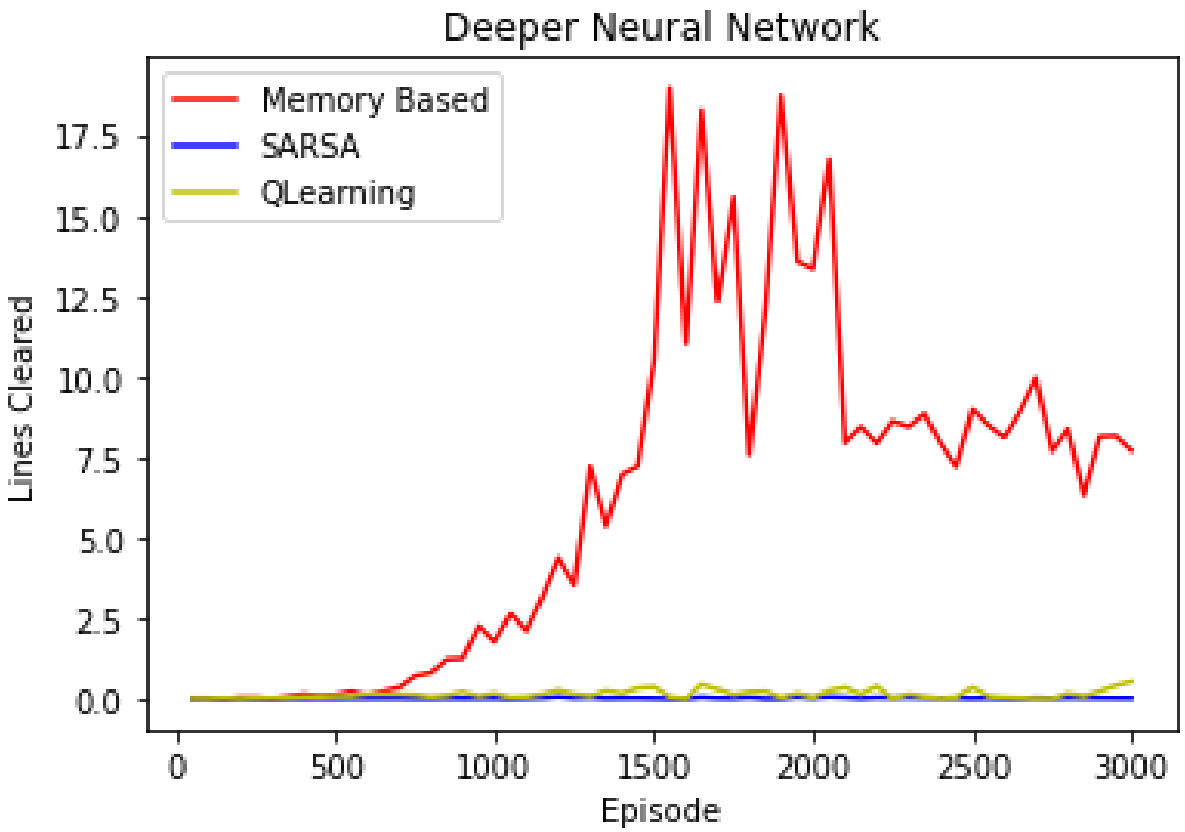


Figure 6.5: Deeper Neural Network Comparison

Chapter 7

Conclusion

In this thesis we presented an approach to reducing the state space of Tetris. We tried to identify redundancy in the game description, and thereby reduce the amount of information required to play intelligently, down to a handleable kernel. We compared two famous algorithms, SARSA and QLearning, with Memory Based Learning, a reinforcement learning algorithm that relies on a replay memory to train and predict actions. Our SARSA and QLearning, agent was found to be lacking in ability, and was completely overshadowed by the performance of the Memory Based agent.

References

[1] Stan Melax. Reinforcement learning tetris example. 1998. https://melax.github.io/tetris/tetris.html.

[2] Yael Bdolah and Dror Livnat. Reinforcement learning playing tetris. 2000. http://www.math.tau.ac.il/ mansour/rl-course/student$_proj/livnat/tetris.html$.

[3] Ronbreukelaar, Erik D.demaine, Susanhohenberger, Hendrik Janhoogeboom, Walter Kosters, and Davidliben-nowell. Tetris is hard, even to approximate. *International Journal of Computational Geometry Applications*, 14, 11 2011.

[4] David Andre and Stuart J. Russell. State abstraction for programmable reinforcement learning agents. In *Eighteenth National Conference on Artificial Intelligence*, page 119–125, USA, 2002. American Association for Artificial Intelligence.

[5] Heidi Burgiel. How to lose at tetris. *The Mathematical Gazette*, 81(491):194–200, 1997.

[6] Colin P. Fahey. https://www.colinfahey.com/tetris/tetris.html.

[7] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction.

[8] Clinton Brett McLean. Design, evaluation and comparrison of evolution and reinforcement learning models. 2001.

[9] Justin Boyan and Michael Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in Neural Information Processing Systems*, 6, 10 1999.

[10] Hajime Kimura, Kazuteru Miyazaki, and Shigenobu Kobayashi. Reinforcement learning in pomdps with function approximation. pages 152–160, 01 1997.

[11] Jonathan Baxter. Knightcap : A chess program that learns by combining td ( ) with game-tree search. 1998.

[12] Peter Stone, Richard S. Sutton, and Gregory Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[13] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[14] R. Crites and A. Barto. Improving elevator performance using reinforcement learning. In *NIPS*, 1995.

[15] Driessens. Relational reinforcement learning. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven.

[16] Sebastian Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. In D. Touretzky J. Elman M. Mozer, P. Smolensky and A. Weigend, editors, *Proceedings of 4th Connectionist Models Summer School*. Erlbaum Associates, June 1993.

[17] John N. Tsitsiklis and Benjamin van Roy. Feature-based methods for large scale dynamic programming. *Mach. Learn.*, 22(1–3):59–94, January 1996.

[18] Dimitri Bertsekas and Sergey Ioffe. Temporal differences-based policy iteration and applications in neuro-dynamic programming1. 04 2012.

[19] Bruno Scherrer. Performance bounds for policy iteration and application to the game of tetris. *J. Mach. Learn. Res.*, 14(1):1181–1227, April 2013.

[20] Michail G. Lagoudakis, Ronald Parr, and Michael L. Littman. Least-squares methods in reinforcement learning for control. In *Proceedings of the Second Hellenic Conference on AI: Methods and Applications of Artificial Intelligence*, SETN '02, page 249–260, Berlin, Heidelberg, 2002. Springer-Verlag.

[21] Vivek F. Farias and Benjamin Van Roy. *Tetris: A Study of Randomized Constraint Sampling*, pages 189–201. Springer London, London, 2006.

[22] Niko Böhm, Gabriella Kókai, and Stefan Mandl. An evolutionary approach to tetris. In *The Sixth Metaheuristics International Conference (MIC 2005)*, August 2005 2005.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[24] Oriol Vinyals et al.

[25] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[27] Raphael van Uffelen Tim Eden, Anthony Knittel. *Reinforcement Learning*. http://www.cse.unsw.edu.au/ cs9417ml/RL1/algorithms.html.

Appendix A

Example: Source Code Samples

Training

```
from collections import deque

import numpy as np
from tensorflow import keras
from tensorflow.keras.layers import Dense, Input
import GameController as GameController
import TetrisAI as Tetris
from time import sleep
import random
from statistics import mean
import sys

# Configuration paramaters for the whole setup
seed = 42
gamma = 0.95  # Discount factor for past rewards
epsilon = 1.0  # Epsilon greedy parameter
epsilon_min = 0  # Minimum epsilon greedy parameter
epsilon_max = 1.0  # Maximum epsilon greedy parameter
epsilon_interval = (
    epsilon_max - epsilon_min
)  # Rate at which to reduce chance of random action being taken

epsilon_stop_episode = 1500
epsilon_decay = (epsilon - epsilon_min) / epsilon_stop_episode
log_every = 50
batch_size = 512
epochs = 1
mem_size = 20_000
memory = deque(maxlen=mem_size)
replay_start_size = 2000
ACTIONS = [(transform, rotation) for transform in range(0 - 5,
    GameController.BOARD_WIDTH - 5) for rotation in range(4)]

INPUT_SHAPE = 4

print("MemoryBased")
sleep(1)


def create_q_model():
    # Network defined by the Deepmind paper
    model = keras.models.Sequential()
    model.add(Input(shape=INPUT_SHAPE, dtype="float32"))
    model.add(Dense(32, activation="relu"))
```

```
    model.add(Dense(32, activation="relu"))
    model.add(Dense(1, activation="linear"))
    model.compile(loss='mse', optimizer='adam')
    return model


def get_best_state(states):
    '''Returns the best state for a given collection of states'''
    max_value = None
    best_state = None

    if epsilon >= np.random.rand(1)[0]:
        # Take random action
        return random.choice(list(states))
    else:
        for s in states:
            value = predict_value(np.reshape(s, [1, INPUT_SHAPE]))
            if not max_value or value > max_value:
                max_value = value
                best_state = s

    return best_state


def predict_value(state):
    """Predicts the score for a certain state"""
    return model.predict(state)


def add_to_memory(current_state, next_state, reward, done):
    """Adds a play to the replay memory buffer"""
    memory.append((current_state, next_state, reward, done))


def train(batch_size=32, epochs=3):
    """Trains the agent"""
    n = len(memory)

    if n >= replay_start_size and n >= batch_size:

        batch = random.sample(memory, batch_size)

        # Get the expected score for the next states, in batch (better
    performance)
        next_states = np.array([x[1] for x in batch])
        next_qs = [x[0] for x in model.predict(next_states)]

        x = []
        y = []

        # Build xy structure to fit the model in batch (better
    performance)
        for i, (state, _, reward, done) in enumerate(batch):
            if not done:
                # Partial Q formula
                new_q = reward + gamma * next_qs[i]
            else:
                new_q = reward
```

```
100
101             x.append(state)
102             y.append(new_q)
103
104         # Fit the model to the given values
105         model.fit(np.array(x), np.array(y), batch_size=batch_size,
    epochs=epochs, verbose=0)
106
107
108 # The first model makes the predictions for Q-values which are used to
109 # make a action.
110 model = create_q_model()
111
112 # Experience replay buffers
113 action_history = []
114 state_history = []
115 state_next_history = []
116 rewards_history = []
117 done_history = []
118 episode_reward_history = []
119 running_reward = 0
120 episode_count = 1
121 frame_count = 0
122 # Number of frames to take random action and observe output
123 epsilon_random_frames = 50000
124 # Number of frames for exploration
125 epsilon_greedy_frames = 100_000.0
126 # Maximum replay length
127 # Note: The Deepmind paper suggests 1000000 however this causes memory
    issues
128 max_memory_length = 10_000
129 # Train the model after 4 actions
130 update_after_actions = 4
131 # How often to update the target network
132 update_target_network = 10000
133 # Using huber loss for stability
134 loss_function = keras.losses.Huber()
135 env = Tetris.Tetris()
136 total_lines_cleared = 0
137 action_count = 0
138 TAG = 'another_state'
139 update_after_episodes = 1
140
141 singles = []
142 doubles = []
143 triples = []
144 tetrises = []
145
146 with open('reports/report_{}.csv'.format(TAG), 'w') as f:
147     print('Created')
148     f.write('Episode,Single,Double,Triple,Tetris,Total,Score\n')
149 try:
150     while True:  # Run until solved
151         done = False
152         episode_reward = 0
153         single = 0
154         double = 0
155         triple = 0
```

```
        tetris = 0
        episode_lines_cleared = 0
        if episode_count % 10 == 0:
            print("Episode: " + str(episode_count))
            print('Total lines cleared: ' + str(total_lines_cleared))
        while not done:
            env.step(None)
            current_state = env.get_state(False)
            possible_next_states = env.get_next_states()
            best_state = get_best_state(possible_next_states.values())

            best_action = None
            for action, state in possible_next_states.items():
                if state == best_state:
                    best_action = action
                    break
            if best_state[0] == 4:
                tetris += 1
            elif best_state[0] == 3:
                triple += 1
            elif best_state[0] == 2:
                double += 1
            elif best_state[0] == 1:
                single += 1

            reward, done = env.step(best_action)

            action_count += 1

            episode_reward += reward

            add_to_memory(current_state, possible_next_states[
best_action], reward, done)

            current_state = possible_next_states[best_action]

            if done:
                episode_lines_cleared = env.gameController.lines
                total_lines_cleared += episode_lines_cleared
                env.gameController.state_initializer()
                break

        episode_reward_history.append(episode_reward)
        singles.append(single)
        doubles.append(double)
        triples.append(triple)
        tetrises.append(tetris)
        # Update every fourth frame and once batch size is over 32
        if episode_count % update_after_episodes == 0:
            train(batch_size, epochs)

            if epsilon > epsilon_min:
                epsilon -= epsilon_decay

        # Logs
        if log_every and episode_count and episode_count % log_every ==
0:
            avg_score = mean(episode_reward_history[-log_every:])
```

```
            min_score = min(episode_reward_history[-log_every:])
            max_score = max(episode_reward_history[-log_every:])
            avg_singles = mean(singles[-log_every:])
            avg_doubles = mean(doubles[-log_every:])
            avg_triples = mean(triples[-log_every:])
            avg_tetrises = mean(tetrises[-log_every:])
            avg_total = mean(singles[-log_every:]) + 2 * mean(doubles[-log_every:]) \
                        + 3 * mean(triples[-log_every:]) + 4 * mean(tetrises[-log_every:])

            # Update running reward to check condition for solving
            with open('reports/report_{}.csv'.format(TAG), 'a') as f:
                f.write('{},{},{},{},{},{},{}\n'.format(episode_count, avg_singles,
                                                        avg_doubles, avg_triples,
                                                        avg_tetrises, avg_total, avg_score))
        episode_count += 1
except SystemExit:
    model.save('models/_{}'.format(TAG))
```

Testing

```
from collections import deque

import numpy as np
from tensorflow import keras
from tensorflow.keras.layers import Dense, Input
from MemoryBased import GameController as GameController, TetrisQLearning as Tetris
from time import sleep
import random
from statistics import mean
import sys

# Configuration paramaters for the whole setup
seed = 42
gamma = 0.95  # Discount factor for past rewards
epsilon = 1.0  # Epsilon greedy parameter
epsilon_min = 0  # Minimum epsilon greedy parameter
epsilon_max = 1.0  # Maximum epsilon greedy parameter
epsilon_interval = (
    epsilon_max - epsilon_min
)  # Rate at which to reduce chance of random action being taken

epsilon_stop_episode = 1500
epsilon_decay = (epsilon - epsilon_min) / epsilon_stop_episode
log_every = 50
batch_size = 512
epochs = 1
mem_size = 20_000
memory = deque(maxlen=mem_size)
replay_start_size = 2000
ACTIONS = [(transform, rotation) for transform in range(0 - 5, GameController.BOARD_WIDTH - 5) for rotation in range(4)]

```

```
INPUT_SHAPE = 4

print("MemoryBased")
sleep(1)

def get_best_state(states):
    '''Returns the best state for a given collection of states'''
    max_value = None
    best_state = None

    for s in states:
        value = predict_value(np.reshape(s, [1, INPUT_SHAPE]))
        if not max_value or value > max_value:
            max_value = value
            best_state = s

    return best_state


def predict_value(state):
    """Predicts the score for a certain state"""
    return model.predict(state)


def add_to_memory(current_state, next_state, reward, done):
    """Adds a play to the replay memory buffer"""
    memory.append((current_state, next_state, reward, done))


def train(batch_size=32, epochs=3):
    """Trains the agent"""
    n = len(memory)

    if n >= replay_start_size and n >= batch_size:

        batch = random.sample(memory, batch_size)

        # Get the expected score for the next states, in batch (better
    performance)
        next_states = np.array([x[1] for x in batch])
        next_qs = [x[0] for x in model.predict(next_states)]

        x = []
        y = []

        # Build xy structure to fit the model in batch (better
    performance)
        for i, (state, _, reward, done) in enumerate(batch):
            if not done:
                # Partial Q formula
                new_q = reward + gamma * next_qs[i]
            else:
                new_q = reward

            x.append(state)
            y.append(new_q)

        # Fit the model to the given values
```

```
88          model.fit(np.array(x), np.array(y), batch_size=batch_size,
        epochs=epochs, verbose=0)
89
90
91  # The first model makes the predictions for Q-values which are used to
92  # make a action.
93  model = keras.models.load_model('models/b_1')
94
95  print(model.summary())
96  sleep(2)
97
98  episode_count = 1
99  # Using huber loss for stability
100 env = Tetris.Tetris()
101 total_lines_cleared = 0
102 episode_reward = 0
103 try:
104     while True:  # Run until solved
105         done = False
106         episode_lines_cleared = 0
107         if episode_count % 10 == 0:
108             print("Episode: " + str(episode_count))
109             print('Total lines cleared: ' + str(total_lines_cleared))
110             print('Episode reward: ', )
111         episode_reward = 0
112         while not done:
113             env.step(None)
114             current_state = env.get_state(False)
115             possible_next_states = env.get_next_states()
116             best_state = get_best_state(possible_next_states.values())
117
118             best_action = None
119             for action, state in possible_next_states.items():
120                 if state == best_state:
121                     best_action = action
122                     break
123
124             reward, done = env.step(best_action)
125
126             episode_reward += reward
127
128             current_state = possible_next_states[best_action]
129
130             if done:
131                 episode_lines_cleared = env.gameController.lines
132                 total_lines_cleared += episode_lines_cleared
133                 env.gameController.state_initializer()
134                 break
135
136         episode_count += 1
137 except SystemExit:
138     print(0)
```