OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH

TU DUBLIN

TECHNOLOGICAL UNIVERSITY DUBLIN

TU5353

PgCert Applied Machine Learning

Final Report

Deep Reinforcement Learning for the game of Tetris

by

Aleksandar Topuzovic, D23127501

Deep Reinforcement Learning for the game of Tetris

This report is submitted in partial fulfilment of the requirements for the

Postgraduate Certificate in Applied Machine Learning

to the

School of Computer Science, *TU Dublin*

Author: Aleksandar Topuzovic, D23127501

Supervisor: Ciaran Kelly, School of Computer Science

Date: 01.12.2024

Abstract

This project aims to develop an artificial intelligence agent that can learn to play the game of Tetris through Deep Reinforcement Learning (DRL). Tetris, while seemingly simple, is complex in practice and has been proven to be computationally challenging to solve. The agent will learn through trial and error, refining its strategies based on the outcomes of its actions to maximise its score over time.

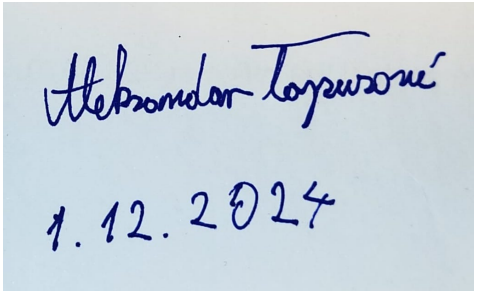
Using modern machine learning techniques and game simulation tools, the system will be implemented in Python to create an environment where the agent can learn and improve its gameplay. To manage the complexity of the problem, a simplified version of Tetris will be used, enabling effective training while preserving the core challenges of the game. The agent's performance will be evaluated against a baseline system that plays randomly, providing clear metrics for success.

The project will demonstrate the potential of machine learning to tackle complex sequential decision-making problems, while offering insights into how artificial intelligence can learn to master classic games through experience.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:



Acknowledgements

To my mother, for her unwavering love, support, and belief in me.

Table of Contents

1 Introduction

1.1 Project Background

1.1.1 Historical Origins

Tetris, one of the most influential video games ever created, emerged in June 1984 from the Moscow Academy of Sciences through the work of Alexey Pajitnov. Drawing inspiration from pentominoes, a childhood puzzle game, Pajitnov created what would become a global phenomenon during the final years of the Cold War (Plank-Blasko, 2015). The game's journey from a Soviet research institution to worldwide recognition exemplifies both its universal appeal and its mathematical elegance.

1.1.2 Game Mechanics and Design

At its core, Tetris operates on a 10x20 grid where players manipulate falling geometric pieces called tetrominoes. Each tetromino, composed of four connected squares, must be rotated and positioned to create complete horizontal lines. These seven distinct shapes (Figure 1) fall continuously, requiring players to make quick decisions about optimal placement. When a row is completely filled, it disappears, and all blocks above descend, creating a satisfying cascade effect. Players score points through various actions: placing pieces, clearing lines, and achieving multiple line clears simultaneously. The game continues until the stack of blocks reaches the top of the screen, making survival and space management crucial elements of success.

1.1.3 Cultural Impact

While the original version featured no music, the 1989 Game Boy release introduced what would become the game's iconic theme - an adaptation of the Russian folk song "Korobeiniki." This musical choice cleverly emphasised the game's Soviet origins, adding to its exotic appeal in Western markets (Plank-Blasko, 2015). Tetris's cultural significance extends beyond gaming, as evidenced by Temple's (2004) BBC documentary "Tetris: From Russia with Love" and a 2023 biographical thriller exploring the complex licensing battles that marked its global distribution.

1.1.4 The AI Challenge

Beyond its cultural impact, Tetris represents a fascinating challenge in artificial intelligence research. Despite its simple rules, the game embodies significant computational complexity. Demaine et al. (2003) proved that offline Tetris is NP-complete through a polynomial-time reduction from 3-Partition. The reduction shows that given a sequence of pieces and an initial board configuration, maximising the number of cleared rows is NP-hard. Specifically, they construct a Tetris game instance where rows can be cleared if and only if there exists a valid solution to the corresponding 3-Partition instance. This establishes that even with perfect knowledge of future pieces, finding an optimal solution is computationally intractable. This complexity arises from several factors:

1. Vast State Space: The standard 10x20 grid allows for approximately 7×2^{200} possible states (Algorta & Simsek, 2019), creating a challenging environment for decision-making algorithms.

2. Sequential Decision Making: Each piece placement affects future possibilities, requiring long-term strategic thinking.
3. Real-time Constraints: In practical play, decisions must be made within time constraints as pieces fall continuously.
4. Delayed Rewards: The impact of decisions may not be immediately apparent, as poor piece placement can lead to difficulties many moves later.

These characteristics make Tetris an ideal testbed for modern machine learning approaches, particularly reinforcement learning. While various AI techniques have been applied to Tetris over the decades, from simple heuristic-based systems to genetic algorithms, the game continues to challenge researchers in creating agents that can match human-level performance consistently.

The complexity and elegance of Tetris make it particularly suitable for exploring deep reinforcement learning techniques. Its well-defined rules and scoring system provide clear feedback for learning algorithms, while its strategic depth offers ample opportunity for discovering sophisticated playing strategies through experience. As computing power and machine learning techniques have advanced, new approaches to tackling this classic challenge have become possible, opening exciting avenues for research in artificial intelligence and game-playing systems.

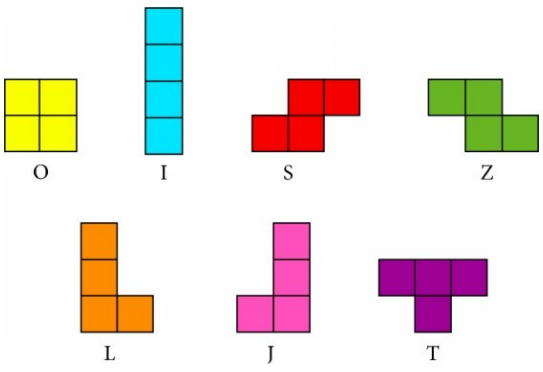


Figure 1 - The seven Tetris tetrominoes, each composed of four connected squares

Figure 2 - Screenshot from the original 1984 version showing the basic game layout

Figure 3 - The iconic Game Boy version that popularised Tetris globally

Figure 4 - Atari's Tetris cover art featuring the Kremlin, emphasising the game's exotic Soviet origins and cultural significance during the Cold War era

1.2 Project Description

This project investigates the application of deep reinforcement learning to create an autonomous Tetris-playing agent. Through systematic experimentation and evaluation, we aim to develop an agent capable of learning effective gameplay strategies directly from experience. The project encompasses three main components:

First, we design and implement a simplified Tetris environment that facilitates agent training while retaining the fundamental complexities that make Tetris an interesting challenge for artificial intelligence. This environment provides the foundation for applying reinforcement learning principles to develop effective piece placement and long-term planning strategies.

Second, we systematically evaluate the agent's performance through comprehensive metrics including cleared lines, survival time, and overall score. Through comparative analysis with baseline strategies, we assess the effectiveness of our learning approach and identify both strengths and limitations of the implementation.

Finally, we consolidate our findings into a cohesive report that documents the methods, results, and implications of this research. This documentation serves to advance our understanding of applying artificial intelligence to complex game environments and provides insights for future research in this field.

1.3 Project Aims and Objectives

The primary aim of this project is to develop and evaluate a Deep Reinforcement Learning agent capable of playing Tetris autonomously using Deep Q-Learning. The agent should demonstrate effective gameplay strategies and outperform random decision-making.

To achieve this main goal, the project is broken down into the following specific objectives:

Core Development

1. Design and implement a Tetris environment that:
    - Accurately simulates core game mechanics (piece movement, rotation, line clearing)
    - Provides a standardised interface using OpenAI Gym
    - Enables efficient training through fast state transitions
    - Captures essential gameplay metrics (score, lines cleared, piece statistics)
2. Develop a Deep Q-Network (DQN) agent that:
    - Processes the game state effectively
    - Learns optimal piece placement strategies
    - Makes decisions within reasonable time constraints
    - Maintains stable learning behaviour
3. Create a reward system that:
    - Incentivizes effective gameplay strategies
    - Balances immediate rewards (piece placement) with long-term planning (line clearing)
    - Discourages unfavourable board states
    - Provides clear learning signals

Training and Optimization

4. Implement comprehensive training procedures:
    - Configure and tune essential hyperparameters
    - Develop an experience replay mechanism
    - Monitor and analyse learning progress
    - Implement early stopping and model checkpointing
5. Establish performance metrics:
    - Average score per game
    - Lines cleared per game
    - Survival time
    - Game completion rate

Evaluation and Analysis

6. Compare the DQN agent against baselines:
    - Implement a random-action baseline agent
    - Conduct statistical analysis of performance differences
    - Evaluate learning efficiency and convergence

- Identify strengths and limitations of the approach

7. Visualisation and Demonstration:

    - Develop a basic ASCII representation for training monitoring
    - Create a graphical interface using Pygame for gameplay demonstration
    - Generate performance visualisation plots
    - Record example gameplay sequences

Documentation and Reporting

8. Provide comprehensive documentation:

    - Detailed technical documentation of implementation
    - Training configuration guidelines
    - Performance analysis results
    - Future improvement recommendations

1.4 Project Scope

This project focuses on developing a Deep Q-Learning agent for playing Tetris while maintaining reasonable computational requirements and implementation complexity. The following sections outline what is included within the project scope and what falls outside of it.

1.4.1 In Scope

Core implementation

- Development of a simplified Tetris environment with standard 10x20 grid
- Implementation of basic Tetris mechanics (piece movement, rotation, line clearing)
- Core DQN agent development using Stable Baselines 3
- Basic reward system implementation
- ASCII and Pygame-based visualisations
- Performance logging and basic metrics tracking

Training and Evaluation

- Training the DQN agent using specified hyperparameters
- Comparison against random-play baseline
- Basic performance analysis and visualisation
- Documentation of training procedures and results

Documentation

- Technical documentation of implementation
- Analysis of results
- Code comments and README files
- Project report and presentation

1.4.2 Out of Scope

Advanced Game Features

- Special game modes (e.g., marathon, sprint)
- Advanced scoring systems

- Piece preview/hold functionality
- Increasing difficulty levels
- Multiplayer capabilities
- Super rotation system

Advanced AI Techniques

- Implementation of other reinforcement learning algorithms beyond DQN
- Comparative analysis with other AI approaches (e.g., genetic algorithms, MCTS)
- Development of novel reinforcement learning techniques
- Advanced neural network architectures
- Multi-agent learning systems

Performance Optimizations

- Distributed training implementation
- GPU optimization
- Advanced memory management
- Real-time performance optimization

Additional Features

- Web-based interface
- Advanced visualisation tools
- Database integration
- API development
- Deployment infrastructure

This scope definition ensures the project remains focused on its core objectives while maintaining feasibility within the given timeframe and resources. The emphasis is on developing a functional DRL agent that demonstrates the application of deep Q-learning to Tetris, rather than creating a full-featured game or exploring multiple AI approaches.

1.5 Thesis Roadmap

Each chapter of this thesis has been structured to systematically present the development and evaluation of a Deep Reinforcement Learning agent for playing Tetris:

Chapter 2: Literature Review examines the theoretical foundations and prior work in applying artificial intelligence to Tetris. It explores fundamental concepts in reinforcement learning, deep Q-networks, and convolutional neural networks, while analysing existing approaches to creating Tetris-playing agents. The chapter also evaluates the technical frameworks and tools available for implementation.

Chapter 3: Design details the development of a comprehensive system architecture for a deep reinforcement learning Tetris agent. The chapter examines the design considerations for each major component: a custom Tetris environment built on the Gymnasium framework, a deep Q-network architecture inspired by successful approaches in Atari game environments, and a modular training system that enables efficient experimentation. It explores the rationale behind key design decisions, including state representation choices, reward structure formulation, and evaluation metrics selection. Special attention is given to creating a flexible

and extensible architecture that supports both research requirements and practical implementation constraints.

Chapter 4: Development details the practical implementation of the Tetris agent, documenting key technical decisions and solutions to challenges encountered. It covers the implementation of the core game mechanics, deep learning architecture, and training infrastructure. The chapter examines how theoretical concepts from reinforcement learning were translated into working code, including the details of state representation, reward calculation, and neural network architecture. Specific attention is given to performance optimizations and the development of comprehensive monitoring systems for tracking agent progress.

Chapter 5: Testing and Evaluation presents a systematic analysis of the agent's performance through both quantitative metrics and qualitative assessment. It describes the experimental methodology used to evaluate the agent, including comparison against baseline systems and statistical analysis of key performance indicators. The chapter examines the agent's learning progression and final capabilities across multiple evaluation criteria, including lines cleared, survival time, and strategic gameplay patterns. It also discusses the challenges encountered during testing and their implications for future improvements.

Chapter 6: Conclusions synthesizes the key findings from the research and places them within the broader context of artificial intelligence and game-playing systems. It critically examines the successes and limitations of the implemented approach, drawing insights about both the specific challenges of applying deep reinforcement learning to Tetris and the broader implications for complex game environments. The chapter concludes by proposing promising directions for future research, including potential improvements to the current implementation and new approaches that could address identified limitations.

2 Literature Review

2.1 Introduction

This chapter presents a systematic review of literature relevant to developing a Deep Reinforcement Learning agent for playing Tetris. The review serves multiple objectives: establishing theoretical foundations, examining existing implementations, and identifying current challenges and opportunities in this field. Through careful analysis of prior work, we aim to understand both successful approaches and remaining challenges in creating effective Tetris-playing agents.

Our literature review methodology followed a structured approach, utilizing key academic databases including Google Scholar, IEEE Xplore, ACM Digital Library, and ArXiv preprints. The search strategy employed combinations of terms including "Tetris," "reinforcement learning," "deep Q-learning," "game AI," and "deep neural networks" to identify relevant research. We focused particularly on papers from the last decade while including seminal works that established fundamental concepts.

The chapter is organized to progress systematically from theoretical foundations to practical implementations. Section 2.2 examines the problem context, establishing why Tetris presents unique challenges for artificial intelligence systems. Section 2.3 presents the core literature review, exploring fundamental concepts in reinforcement learning and examining various approaches to creating Tetris-playing agents, from early heuristic-based systems to modern deep learning solutions. Section 2.4 analyzes related work specifically focused on Tetris AI implementations. Section 2.5 examines the technical frameworks and tools available for implementation, while Section 2.6 synthesizes the requirements derived from this review.

Through this comprehensive review, we establish the theoretical groundwork for our implementation while identifying gaps in current approaches that our work aims to address. The insights gained directly inform the design decisions and implementation strategies described in subsequent chapters.

2.2 Problem Context

Tetris presents unique challenges for artificial intelligence systems, combining elements of real-time decision making, spatial reasoning, and long-term strategic planning. To understand these challenges, we must examine both the theoretical complexity of the game and the practical difficulties in creating effective playing strategies.

The fundamental challenge of Tetris lies in its computational complexity. Demaine et al. (2003) proved that Tetris is NP-complete in its offline form, where the sequence of pieces is known in advance. This classification puts Tetris in the same complexity class as many of computer science's most challenging problems, indicating that finding truly optimal solutions is computationally intractable. The proof was established by reducing Tetris to the NP-complete problem known as 3-Partition, demonstrating that the complexity of optimal play matches that of well-known computationally hard problems.

Beyond its theoretical complexity, Tetris poses the interesting challenge of having no guaranteed winning strategy. Brzustowski's (1992) work on defeating algorithms demonstrated that certain sequences of pieces, particularly involving S and Z tetrominoes,

inevitably lead to loss. These pieces force the creation of "shields" - specific formations of filled cells that cannot be dismantled without replicating them higher in the game board. The finite height of the Tetris well (typically 20 rows) means that these formations will eventually reach the top, leading to game over regardless of player skill or strategy.

From an AI perspective, several characteristics make Tetris particularly challenging:

1. Large State Space: This vast state space of 7 × 2^200 possible configurations creates an enormous space of possible game states that an AI must learn to evaluate.
2. Delayed Consequences: The impact of piece placement may not be immediately apparent, requiring the AI to develop long-term strategic understanding.
3. Real-time Decisions: In practical play, decisions must be made within time constraints as pieces fall continuously, adding pressure to the decision-making process.
4. Sparse Rewards: Meaningful rewards (line clears) occur relatively infrequently compared to the number of actions taken, making it harder for learning algorithms to identify successful strategies.

These characteristics make Tetris an excellent testbed for reinforcement learning approaches. While perfect play is provably impossible, the challenge becomes developing strategies that maximise survival time and score, closely matching how human players approach the game. The combination of clear rules, complex strategy, and impossible perfection makes Tetris particularly suitable for evaluating the capabilities of modern machine learning techniques.

2.3 Literature Review

A pivotal advancement in applying deep learning to games was introduced by Mnih et al. (2013) in their seminal work, Playing Atari with Deep Reinforcement Learning. This study demonstrated that convolutional neural networks (CNNs) could directly learn control policies from high-dimensional sensory input. By using raw pixel data as input and game score as a reward signal, their Deep Q-Network (DQN) achieved human-level performance across a diverse set of Atari 2600 games.

The DQN architecture represented a significant departure from previous approaches by combining feature extraction and decision-making within a single neural network. This end-to-end learning approach eliminated the reliance on manually engineered features prevalent in earlier systems, a particularly relevant advancement for Tetris where traditional AI implementations had heavily depended on pre-defined heuristics to evaluate board states.

A crucial innovation in their work was the introduction of experience replay, which enabled the storage and random sampling of past experiences during training. This mechanism effectively addressed the correlation between consecutive samples in sequential decision-making tasks. For Tetris, where the consequences of actions often manifest over multiple moves, this approach proves especially valuable in addressing the temporal credit assignment problem inherent in the game.

The authors further enhanced training stability through the implementation of a separate target network, periodically updated with the primary network's weights. This technique particularly addressed the inherent instability in learning value functions for environments with delayed rewards, a characteristic challenge in Tetris gameplay where the impact of piece placement may only become apparent several moves later.

Subsequent works, such as Deep Reinforcement Learning with Python by Gym and Sanghi (2021), provided valuable practical insights into implementing reinforcement learning systems. While not foundational in the theoretical sense, this book served as a bridge between the abstract principles of reinforcement learning and their application in contemporary software frameworks. Its guidance on using environments compatible with the Gymnasium interface and leveraging pre-built implementations of algorithms, such as Deep Q-Networks, significantly informed the technical development of this project. These practical resources complemented the theoretical foundation provided by seminal texts like Sutton and Barto's Reinforcement Learning: An Introduction, which remains a cornerstone in understanding the principles underlying reinforcement learning.

The success of DQN in Atari games demonstrated the algorithm's ability to master complex control policies using only raw pixels as input. Mnih et al. concluded that this approach could generalize to a wide range of reinforcement learning tasks beyond Atari, offering a robust framework for tackling environments with high-dimensional state spaces. Tetris, while offering immediate feedback for specific actions such as line clears, presents additional challenges due to its vast state space and the long-term strategic considerations required for effective gameplay. These characteristics make Tetris an intriguing testbed for extending the principles established by Mnih et al., particularly in exploring how DQN can handle delayed rewards and long-term planning within a dynamic, grid-based environment.

2.3.1 Reinforcement learning

Reinforcement Learning (RL) represents a fundamental approach in machine learning where agents learn optimal behaviours through interactions with their environment. Unlike supervised learning which relies on labelled training data, RL agents learn through a trial-and-error process, adjusting their behaviour based on the feedback received from their actions (Sutton and Barto, 2018). This feedback mechanism makes RL particularly suitable for game-playing tasks, where the quality of decisions can be evaluated through gameplay outcomes.

The core components of a reinforcement learning system include:

- **Environment**: The context in which the agent operates. In Tetris, this is the game board with its current piece configuration.
- **State (S):** A representation of the current situation. For Tetris, this includes the board state and the current falling piece.
- **Action (A)**: The choices available to the agent. In Tetris, these are piece movements and rotations.
- **Reward (R)**: Numerical feedback signals indicating the desirability of outcomes. For Tetris, rewards might be given for clearing lines or penalties for game over.
- **Policy (π)**: The strategy that maps states to actions, defining how the agent behaves.

These elements interact in a continuous cycle: the agent observes the current state, selects an action based on its policy, receives a reward, and observes the next state resulting from its action. The goal is to learn a policy that maximises the expected cumulative reward over time, not just immediate rewards.

A key challenge in reinforcement learning is the exploration-exploitation dilemma. Agents must balance exploring new actions to discover potentially better strategies (exploration) with

using known good strategies (exploitation). This is particularly relevant in Tetris, where different piece placement strategies might have long-term implications that are not immediately apparent.

The mathematical framework for RL typically uses Markov Decision Processes (MDPs), which provide a formal model for sequential decision-making. In an MDP, the next state and reward depend only on the current state and action, not on the history of previous states and actions. While Tetris can be modelled as an MDP, its large state space and delayed rewards make it a challenging environment for traditional RL approaches, motivating the use of deep learning techniques to handle this complexity.

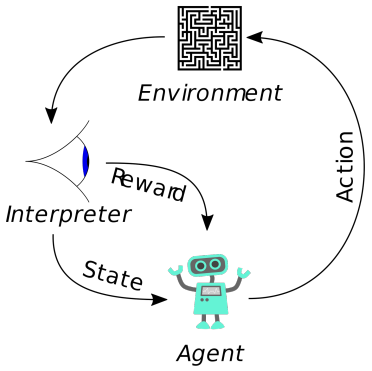


Figure 5 - The Reinforcement Learning interaction loop

2.3.2 Q Learning

Q-learning is a model-free reinforcement learning algorithm that enables agents to learn optimal action-value functions through direct environment interaction. Introduced by Watkins (1989) and further developed in Watkins and Dayan (1992), Q-learning stands out for its ability to learn effective policies without requiring a model of the environment's dynamics.

At its core, Q-learning maintains a table of Q-values, Q(s,a), representing the expected future reward for taking action 'a' in state 's' and then following the optimal policy thereafter. The Q-value update formula is shown in Figure 7.

$$Q^{new}(S_t, A_t) \leftarrow (1 - \underbrace{\alpha}_{\text{learning rate}}) \cdot \underbrace{Q(S_t, A_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \Big(\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(S_{t+1}, a)}_{\text{estimate of optimal future value}} \Big)$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

Figure 6: Q-learning update formula

In practice, Q-learning follows an iterative process:

1. Observe current state
2. Select an action (often using an ε-greedy strategy)
3. Execute the action and observe reward and new state
4. Update Q-values using the formula

5. Repeat until convergence

While Q-learning guarantees convergence to optimal values given sufficient exploration, it faces practical limitations in environments with large state spaces like Tetris. A simple 10x20 Tetris board has 2^200 possible states, making it impossible to store and update Q-values for every state-action pair. This limitation motivates the extension to Deep Q-learning, where neural networks are used to approximate the Q-function, allowing the algorithm to generalise across similar states and handle large state spaces effectively.

Algorithm 1: Q-learning Algorithm(off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$;
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ arbitrarily, except $Q(\text{terminal}, \cdot) = 0$;
foreach *episode* **do**
  Initialize S;
  foreach *step of episode* **do**
    Choose A from S using policy derived from Q (e.g., $\varepsilon - greedy$);
    Take action A, observe R, S';
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$;
    $S \leftarrow S'$;
  end
  until S is terminal
end

Figure 7 - Q-learning Algorithm

2.3.3 Deep Q Networks

Deep Q-Networks (DQN) represent a breakthrough in reinforcement learning by successfully combining deep neural networks with Q-learning principles. Introduced by Mnih et al. (2015) in their seminal work on Atari games, DQN demonstrated for the first time that deep reinforcement learning could achieve human-level performance across a variety of challenging games using only pixel inputs and game scores.

The key innovation of DQN is replacing the Q-table with a deep neural network that approximates the Q-function. This network takes a state representation as input and outputs Q-values for all possible actions, enabling the agent to handle complex, high-dimensional state spaces that would be impractical with traditional Q-learning.

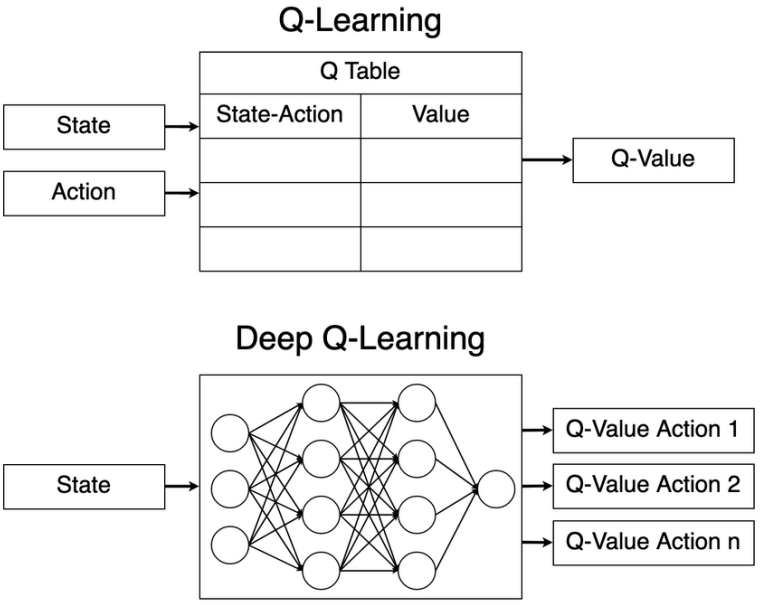


Figure 8 - Q-Learning vs Deep Q-Learning architecture comparison

DQN introduces several crucial mechanisms to stabilise the learning process:

Experience Replay: Instead of learning from consecutive samples, DQN stores experiences in a replay buffer and randomly samples from it for training. This approach breaks the correlation between consecutive samples and improves learning stability by allowing the agent to learn from rare but important experiences multiple times.

Target Network: DQN maintains two networks - the primary network for action selection and a separate target network for generating target Q-values. The target network is updated periodically with the primary network's weights, helping to reduce the moving target problem inherent in Q-learning.

For Tetris specifically, DQN offers several advantages. It can learn directly from the game board state, discovering useful features automatically rather than relying on hand-crafted ones. The network can also identify patterns and generalise across similar board configurations, a crucial capability given the game's vast state space.

2.3.4 Neural Networks for Game AI

Neural networks have become fundamental tools in modern game AI, particularly in reinforcement learning applications. Their ability to process complex input data and learn meaningful representations makes them especially suitable for game environments where traditional approaches might struggle with the high dimensionality of state spaces.

In game environments, neural networks typically serve two crucial functions. First, they act as function approximators, learning to map game states to action values or policies. Second, they serve as feature extractors, automatically learning to identify relevant patterns and structures in the game state that are important for decision-making.

For Tetris, two main neural network approaches are commonly used: multilayer perceptrons (MLPs) with hand-crafted features and convolutional neural networks (CNNs) processing raw board states. MLPs require careful feature engineering, where domain knowledge is used to extract relevant game characteristics such as holes, height differences, and line completion opportunities. While this approach can be effective, it requires significant expert knowledge and careful feature design.

In contrast, CNNs can work directly with the raw game board representation, automatically learning to recognize relevant patterns and features through their convolutional layers. This approach, demonstrated successfully by Mnih et al. (2015) in their work on Atari games, eliminates the need for manual feature engineering. In Tetris, CNNs can naturally capture spatial relationships in the grid-like board structure, learning to identify important game features such as holes, complete lines, and piece configurations through their hierarchical processing layers.

Our project implements and compares both approaches, investigating the trade-off between the explicit feature engineering required for MLPs and the automatic feature learning capability of CNNs. This comparison provides insights into the benefits and challenges of each approach in the context of Tetris, particularly focusing on the simplification of the development process when using CNNs versus the control and interpretability offered by carefully designed features with MLPs.

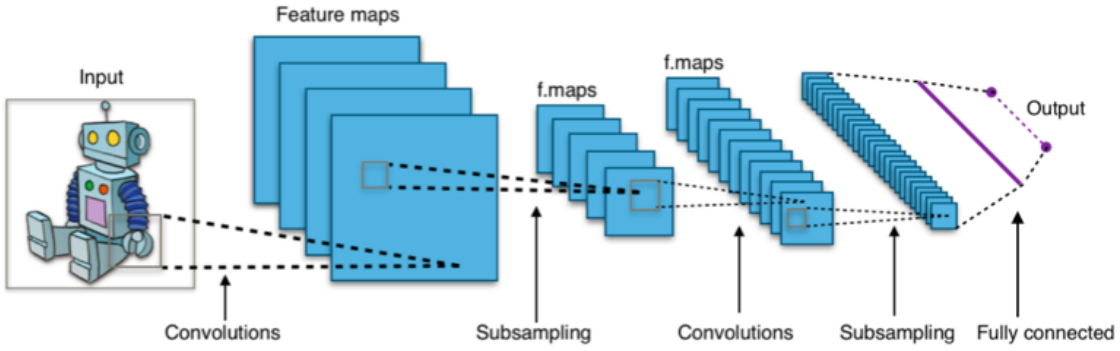


Figure 9 - Convolutional Neural Network Architecture

2.4 Related work in Tetris AI

The history of artificial intelligence approaches to playing Tetris spans several decades, showcasing the evolution of game-playing AI techniques. This progression provides valuable insights into both the challenges of creating effective Tetris agents and the various methods developed to address them.

2.4.1 Classical Approaches

Early attempts to create Tetris-playing AI systems primarily focused on rule-based and heuristic approaches. These methods typically evaluated potential piece placements using hand-crafted scoring functions based on features such as board height, number of holes, and line completions. One of the most successful classical approaches was developed by Pierre Dellacherie, whose simple yet effective six-feature system achieved remarkably good performance despite its straightforward implementation.

The key features traditionally used in these approaches include:

- **Aggregate Height**: Sum of all column heights
- **Complete Lines**: Number of full lines ready to be cleared
- **Holes**: Empty cells with filled cells above them
- **Bumpiness**: Sum of differences between adjacent columns
- **Wells**: Deep holes perfect for I-piece placement
- **Landing Height**: Height where the current piece is placed

Carr (2005) explored these classical approaches in depth, noting their effectiveness while highlighting their limitations. The main advantage of these methods was their computational efficiency and interpretability - each decision could be traced back to specific feature weights. However, they struggled with long-term strategic planning and could not adapt to new situations beyond their programmed rules.

Another significant classical approach was the use of genetic algorithms to optimise the weights of these feature-based evaluation functions. This automated the process of finding effective feature combinations, potentially discovering strategies that human designers might not consider. However, these approaches still relied on the predefined features and could not discover entirely new evaluation criteria.

The limitations of classical approaches, particularly their reliance on hand-crafted features and inability to learn from experience, ultimately motivated the exploration of more advanced machine learning techniques. These limitations become especially apparent when considering Brzustowski's (1992) proof that perfect play is impossible, suggesting that adaptive strategies capable of maximising survival time might be more effective than rigid rule-based approaches.

2.4.2 Reinforcement Learning Approaches

The application of reinforcement learning to Tetris marked a significant shift from static, rule-based systems to adaptive agents that could learn from experience. Carr's 2005 work "Applying Reinforcement Learning to Tetris" laid important groundwork by establishing Tetris as a Markov Decision Process (MDP), making it suitable for RL approaches while highlighting the challenges posed by its vast state space.

Early RL approaches to Tetris faced several key challenges. Lundgaard and McKee explored these in their work on "Reinforcement Learning and Neural Networks for Tetris," finding that while Q-learning agents could learn effective strategies, they struggled with the delayed nature of rewards in Tetris - actions that lead to game over might have been made many moves earlier. Their work also highlighted the challenge of designing appropriate reward functions, as simply rewarding line clears did not necessarily lead to robust long-term strategies.

A significant advancement came with the introduction of afterstate value functions, where the agent learns to evaluate board positions that would result from actions rather than the actions themselves. This approach, discussed by Carr (2005), helped reduce the complexity of the learning task by focusing on the consequences of actions rather than the actions themselves.

Stevens and Pradhan's work on "Playing Tetris with Deep Reinforcement Learning" introduced several important innovations:

- State representation techniques that balanced information content with computational efficiency
- The use of action grouping to simplify the learning task
- Transfer learning approaches to accelerate training

These early reinforcement learning approaches provided valuable insights into:

1. The importance of state representation in managing complexity
2. The role of reward design in shaping agent behaviour
3. The benefits of reducing the action space through clever problem formulation
4. The challenge of credit assignment in games with delayed rewards

While these approaches demonstrated the potential of reinforcement learning for Tetris, they also revealed limitations that would later be addressed by deep reinforcement learning methods. The key challenge remained finding effective ways to handle the game's large state space while learning meaningful long-term strategies.

2.4.3 Deep Learning Approaches

The application of deep learning to Tetris represents the latest evolution in AI approaches to the game, marked by the ability to learn complex features and strategies directly from raw game states. This advancement was largely inspired by Mnih et al.'s 2013 breakthrough in applying deep reinforcement learning to Atari games, which demonstrated that neural networks could effectively learn to play games directly from screen pixels.

Liu and Liu's work "Learn to Play Tetris with Deep Reinforcement Learning" marked a significant contribution by exploring various deep RL algorithms in both simplified and full Tetris environments. Their research tested multiple advanced architectures including Dreamer, DrQ, and PPO, providing valuable insights into the effectiveness of different deep learning approaches. A key finding was that while deep RL performed well in simplified Tetris environments, the complexity of the full game presented significant challenges, particularly in terms of exploration and sparse rewards.

Algorta and Simsek's comprehensive review "The Game of Tetris in Machine Learning" highlighted several key advantages of deep learning approaches:

- Automatic feature extraction from raw board states
- Ability to discover novel strategies beyond human-designed heuristics
- Potential for transfer learning between different board sizes and game variants

However, their work also identified persistent challenges:

- Long training times due to the need for extensive experience
- Difficulty in maintaining consistent performance across different game conditions
- The need for careful reward shaping to guide learning effectively

Recent approaches have focused on addressing these challenges through various innovations:

1. State representation optimization to reduce input complexity while preserving important game information
2. Novel network architectures specifically designed for grid-based games
3. Improved training techniques to handle delayed rewards and sparse feedback

The success of deep learning approaches in Tetris demonstrates both the power and limitations of current AI techniques. While these methods can learn sophisticated strategies without hand-crafted features, they often require significant computational resources and careful architecture design to achieve stable performance.

2.4.4 Current State-of-the-Art

The current state-of-the-art in Tetris AI reflects the ongoing challenge of creating effective game-playing agents. According to Algorta and Simsek's comprehensive review, while significant progress has been made in developing Tetris AI, achieving consistent high-level performance remains a complex challenge due to the game's vast state space and delayed reward structure.

Liu and Liu's recent work demonstrates that modern deep reinforcement learning approaches show promise in simplified Tetris environments but face challenges in scaling to full gameplay. Their experiments with various algorithms including Dreamer, DrQ, and PPO highlight that successful implementations require careful consideration of:

- State representation and action space design
- Environment modifications to expedite learning
- Reward shaping to guide effective strategy development

The most successful current approaches tend to balance the complexity of the learning task with practical implementation considerations. This includes making strategic simplifications to the game environment while maintaining the core challenges that make Tetris an interesting problem for AI research.

2.5 Technical Framework

The implementation of a deep reinforcement learning agent for Tetris requires a robust and efficient technical framework. The choice of technologies significantly impacts both development efficiency and system performance. This section examines the key technologies selected for this project and their roles in the implementation.

2.5.1 Python for Machine Learning

Python has established itself as the de facto standard for machine learning projects, offering a combination of simplicity, readability, and a rich ecosystem of specialised libraries. According to Raschka, Patterson, and Nolet (2020), Python's success in machine learning can be attributed to several key factors:

Development Efficiency

Python's clear syntax and straightforward structure allow developers to focus on implementing machine learning concepts rather than dealing with complex programming constructs. This is particularly valuable in research and experimental contexts where rapid prototyping and iteration are essential.

Scientific Computing Ecosystem

The language provides essential numerical computing tools through libraries such as:

- **NumPy**: Efficient manipulation of multi-dimensional arrays and matrices
- **SciPy**: Advanced scientific computing tools and algorithms

- **Pandas**: Data manipulation and analysis

Machine Learning Libraries

Python offers a comprehensive collection of machine learning frameworks that make implementing complex algorithms accessible:

- **Stable-Baselines3**: Reliable implementations of reinforcement learning algorithms
- **PyTorch**: Dynamic neural networks and automatic differentiation
- **Gymnasium**: Standardised environments for reinforcement learning

While Python's interpreted nature can pose performance limitations, many of these libraries leverage optimised implementations in lower-level languages such as C++ while maintaining Python's user-friendly interface. This combination of ease of use and computational efficiency makes Python particularly suitable for developing and experimenting with reinforcement learning agents.

The extensive Python community also provides abundant resources, documentation, and support, facilitating rapid problem-solving and knowledge sharing during development. This community-driven ecosystem has been crucial in establishing Python as the primary tool for machine learning research and development.

2.5.2 Deep Learning Frameworks

Modern deep learning frameworks have revolutionised the development of AI systems by providing robust, efficient implementations of complex algorithms while maintaining accessibility for researchers and developers. For this project, we leverage two primary frameworks that complement each other to create a powerful development environment.

At the heart of our implementation lies Stable-Baselines3, a deep reinforcement learning framework designed with reliability and usability in mind. Developed by Raffin et al. (2021), Stable-Baselines3 emerged from the need for dependable, well-tested implementations of reinforcement learning algorithms. The framework handles many of the intricate details of reinforcement learning, from experience replay to model updating, allowing developers to focus on the unique aspects of their applications. Its modular design enables customization where needed while providing sensible defaults that work well in practice.

Underlying this reinforcement learning capability is PyTorch, a dynamic deep learning framework that has gained widespread adoption in the research community. Paszke et al. (2019) designed PyTorch with a Python-first philosophy, creating an intuitive interface that feels natural to Python developers while maintaining high performance through its C++ backend. This approach has proven particularly valuable in research settings, where the ability to quickly prototype and debug ideas is crucial.

The synergy between these frameworks creates a development environment that balances power with usability. While PyTorch provides the fundamental building blocks for creating and training neural networks, Stable-Baselines3 builds upon this foundation to offer high-level reinforcement learning algorithms. This layered approach means we can leverage battle-tested implementations for standard components while maintaining the flexibility to customise elements specific to our Tetris application.

The combination proves especially valuable for our project, where we need both the reliability of established algorithms and the flexibility to adapt them to the unique challenges of Tetris. Whether we're implementing custom neural network architectures or fine-tuning training parameters, these frameworks provide the tools we need while abstracting away much of the underlying complexity.

2.5.3 Reinforcement Learning Tools

The development of reinforcement learning applications has been greatly simplified by the emergence of specialised tools that provide standardised interfaces and efficient implementations. At the core of modern reinforcement learning development is Gymnasium, a fork and successor of the original OpenAI Gym framework.

Gymnasium, as described by Towers et al. (2024), provides a standardised interface for reinforcement learning environments that has become the de facto standard in the field. This standardisation is crucial - it allows researchers and developers to focus on agent development while ensuring their work can be easily compared and reproduced. The framework's design reflects years of community experience in reinforcement learning research, offering a clean, intuitive API that handles the complexities of environment-agent interaction.

What makes Gymnasium particularly valuable for our Tetris implementation is its flexible environment specification system. Each environment implements a consistent interface with core methods like `step()` and `reset()`, while allowing for customization of observation spaces, action spaces, and reward structures. This means we can create a Tetris environment that faithfully represents the game while conforming to standardised practices that ensure compatibility with existing reinforcement learning algorithms.

The framework also provides essential tools for environment manipulation and performance optimization. Its vectorized environment implementations allow multiple game instances to run in parallel, significantly speeding up the training process. This is particularly valuable in Tetris, where agents need to experience many games to learn effective strategies. The ability to run multiple environments simultaneously means our agent can gather experience more efficiently, leading to faster learning and more robust performance.

By providing these standardised interfaces and efficient implementations, Gymnasium creates a robust development environment that handles many of the technical challenges inherent in reinforcement learning implementation. This allows us to focus on the unique aspects of our Tetris agent while building on proven, reliable foundations.

2.5.4 Visualisation and Monitoring Tools

Effective monitoring and visualisation of the training process is crucial in deep reinforcement learning projects, where understanding agent behaviour and learning progress can be challenging. Our implementation utilises several tools to provide insights into the training process and agent performance.

TensorBoard serves as our primary training monitoring tool. Originally developed for TensorFlow but now widely adopted across frameworks, TensorBoard provides real-time visualisation of training metrics. When used with Stable-Baselines3, it automatically tracks essential reinforcement learning metrics such as episode rewards, episode lengths, and loss values. This integration becomes particularly valuable in our Tetris implementation, where

we can monitor not only these standard metrics but also game-specific indicators like the number of lines cleared and individual action frequencies.

For game visualisation, we employ Pygame, a set of Python modules designed for writing video games. Pygame provides a simple yet effective way to render the Tetris game state, allowing us to visually verify the agent's behaviour and create recordings of gameplay. This visual feedback is invaluable during development, helping us identify potential issues in the environment implementation and understand the strategies being learned by the agent.

The combination of these visualisation tools creates a comprehensive monitoring system:

- TensorBoard: Training metrics, learning curves, and performance trends
- Pygame: Real-time gameplay visualisation and strategy analysis

This dual approach to monitoring enables both quantitative analysis of the agent's learning progress through TensorBoard's metrics and qualitative assessment of its behaviour through Pygame's visual representation. Such comprehensive monitoring is essential for understanding how different hyperparameters and architectural choices affect the agent's learning and performance.

2.6 Requirements Analysis

The comprehensive review of literature, existing solutions, and available technologies helps formulate a clear set of requirements for our Tetris AI system. Our analysis reveals opportunities to advance the field by addressing specific gaps in current implementations while providing valuable tools for researchers and developers.

Our system aims to empower researchers and developers in ways currently not possible with existing implementations. By providing a comprehensive platform for comparing traditional and modern approaches to Tetris AI, stakeholders will be able to directly evaluate the trade-offs between different methodologies. This includes understanding the relative merits of hand-crafted versus learned features, assessing various neural network architectures, and accessing well-documented implementations of deep reinforcement learning techniques applied to Tetris.

1. Dual Architecture Implementation

   Priority: Vital

   Source: Liu and Liu's work highlighting the need for comparative analysis

   The system must provide a comprehensive implementation of both MLP and CNN approaches. The MLP architecture will work with hand-crafted features, following traditional methodologies, while the CNN approach will learn directly from raw board states. This dual implementation enables direct comparison between classical and modern approaches, offering insights into the effectiveness of different methodologies in Tetris AI development.

2. Standardised Environment

   Priority: Vital

   Source: Gymnasium framework (Towers et al., 2024)

Creating a consistent and well-defined Tetris environment following Gymnasium standards is essential for ensuring reproducibility and compatibility. This standardisation enables other researchers to build upon our work and allows for fair comparisons between different approaches. The environment must handle all core Tetris mechanics while providing clear interfaces for agent interaction.

3. Efficient Training System

   Priority: Vital

   Source: Mnih et al.'s DQN implementation principles

   The training system must incorporate key components for efficient deep reinforcement learning, including parallel training environments, experience replay, and stable learning mechanisms. These elements are crucial for effective agent training, ensuring the system can learn complex strategies within reasonable time frames while maintaining stability throughout the training process.

4. Performance Monitoring

   Priority: Important

   Source: Raffin et al.'s Stable-Baselines3 methodology

   The system requires comprehensive monitoring capabilities to track and analyse agent performance. This includes tracking various metrics during training, from basic game performance indicators to detailed learning progress measurements. Such monitoring is essential for understanding agent behaviour, identifying issues, and validating improvements in the learning process.

5. Feature Engineering Framework

   Priority: Important

   Source: Classical approaches discussed by Carr (2005)

   A robust framework for implementing and testing traditional Tetris features is necessary for the MLP approach. This includes calculations for board characteristics like holes, bumpiness, and aggregate height. The framework must be flexible enough to allow experimentation with different feature combinations while providing consistent measurements for comparative analysis.

6. Vectorized Training

   Priority: Important

   Source: Modern RL practices from Stable-Baselines3

   The implementation must support parallel execution of multiple game environments to accelerate the training process. This vectorized approach significantly improves data collection efficiency and training speed, enabling more thorough experimentation and faster iteration cycles during development.

7. Comparative Metrics

   Priority: Vital

   The system must implement a comprehensive set of evaluation metrics that enable fair and thorough comparison between different approaches. These metrics should include not only traditional game performance measures like average score and lines cleared, but also training efficiency metrics and stability measurements. This multifaceted evaluation approach ensures a complete understanding of each method's strengths and weaknesses.

8. Visualisation System

   Priority: Desirable

   Source: Analysis needs identified in various implementations

   A real-time visualisation system is needed to observe and analyse agent behaviour during both training and evaluation. This system should provide clear visual feedback about the agent's decision-making process and game performance, facilitating deeper understanding of learned strategies and potential areas for improvement.

9. Implementation Documentation

   Priority: Important

   Source: Best practices from reviewed implementations

   Comprehensive documentation must cover all aspects of the system, from architectural designs to training procedures. This documentation should provide clear explanations of implementation choices, configuration options, and usage guidelines, ensuring that other researchers can understand and build upon our work.

10. Reproducibility Guidelines

    Priority: Important

    Source: Scientific methodology requirements

    Detailed guidelines for reproducing experiments and results are essential. These should include specific hyperparameter settings, environmental configurations, and step-by-step procedures for training and evaluation, ensuring that our research findings can be independently verified and extended.

11. Hyperparameter Tuning

    Priority: Desirable

    Source: Common challenge identified in literature

    The system should provide tools and frameworks for systematic hyperparameter optimization. This includes methods for exploring different parameter combinations

and evaluating their impact on agent performance, helping to identify optimal configurations for different approaches.

12. Model Checkpointing

    Priority: Important

    Source: Standard practice in deep learning

    Regular saving of model states and training progress is necessary for both analysis and recovery purposes. This feature enables long-term training stability by providing recovery points and allows for detailed analysis of the learning process over time.

2.7 Conclusions

This literature review chapter has established the theoretical foundations, examined existing approaches, and identified key requirements for developing a deep reinforcement learning agent for Tetris. Through analysis of both historical and contemporary works, we have traced the evolution of Tetris AI from early rule-based systems to modern deep learning approaches.

Our examination began with understanding the fundamental complexity of Tetris, established through Demaine et al.'s proof of NP-completeness and Brzustowski's analysis of gameplay limitations. These theoretical foundations highlight why Tetris remains a challenging and interesting problem for artificial intelligence research.

The review of theoretical frameworks, from basic reinforcement learning principles to advanced deep Q-networks, provides the necessary background for our implementation. Particularly significant is Mnih et al.'s groundbreaking work on DQN, which demonstrated the potential of deep reinforcement learning in game environments. This work, combined with more recent advances in the field, shapes our approach to the Tetris challenge.

Our analysis of existing solutions reveals a clear progression in Tetris AI development:

- Classical approaches demonstrated the effectiveness of hand-crafted features
- Early reinforcement learning attempts highlighted the challenges of large state spaces
- Modern deep learning solutions showed promise while revealing ongoing challenges in training efficiency and stability

The technical framework discussion identified powerful tools and libraries that make our implementation possible. Python's scientific computing ecosystem, combined with specialised frameworks like Stable-Baselines3 and Gymnasium, provides a robust foundation for developing and evaluating our agent.

From this comprehensive review, we have derived twelve key requirements, ranging from vital architectural needs to desirable features for analysis and evaluation. These requirements will guide our implementation, ensuring that our work both builds upon existing knowledge and contributes meaningful insights to the field.

The next chapters will detail how we translate these requirements into a practical implementation, addressing the identified gaps in current approaches while leveraging the strengths of modern deep learning frameworks.

3 Design

3.1 Introduction

This chapter presents the design and implementation approach for developing an artificial agent capable of learning to play Tetris using Deep Reinforcement Learning (DRL). Drawing from successful implementations in recent literature, particularly Mnih et al.'s (2013) work on Atari games, we designed a system that combines a custom Tetris implementation with a Deep Q-Network (DQN) agent, creating a comprehensive learning environment.

Our design philosophy centred on three core principles: modularity, allowing independent development and testing of components; extensibility, supporting future enhancements and experimentation; and observability, enabling thorough analysis of the learning process. This approach follows established practices in deep reinforcement learning research and software development industry while addressing the specific challenges of the Tetris environment.

The system consists of three main components:

1. **Game Core**: Implements the fundamental Tetris mechanics, state management, and environment interface. This component ensures accurate game simulation while providing a standardised interface for agent interaction.
2. **Learning System**: Contains the DQN agent, featuring custom convolutional neural networks for feature extraction and Q-value estimation. This component handles the learning process and decision-making.
3. **Support Systems**: Provides infrastructure for training, visualisation, and evaluation. These systems enable monitoring of the learning process and assessment of agent performance.

The chapter systematically details each component of our design approach. Section 3.2 examines our environment design, describing how we balanced authentic Tetris mechanics with reinforcement learning requirements. Section 3.3 presents our architecture model design, detailing the neural network structure and how it processes game states to select actions. Section 3.4 explores our training design, covering experience collection, reward structures, and learning optimizations. Section 3.5 details our evaluation design, establishing the frameworks for assessing agent performance through both quantitative metrics and qualitative analysis. Through this systematic examination, we establish clear motivations for our implementation choices while maintaining focus on core research objectives.

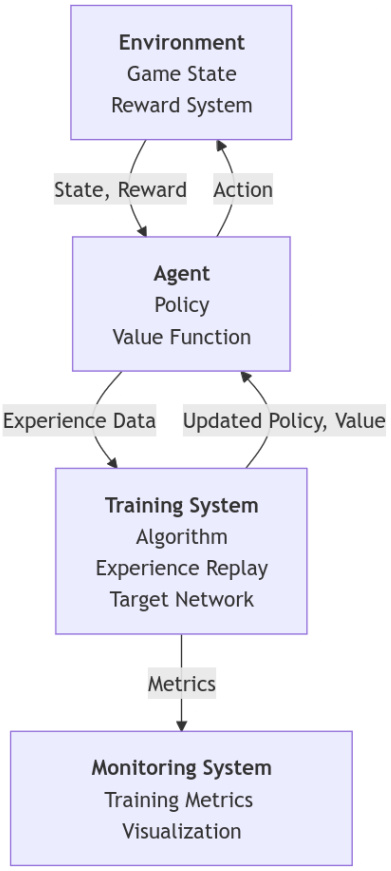


Figure 10 - Architecture Diagram

3.2 Environment Design

The Tetris environment design balances authentic gameplay mechanics with the specific requirements of reinforcement learning training. Following the standardised Gymnasium interface (Towers et al., 2024), our environment provides a consistent API for agent interaction while maintaining the core elements that make Tetris an engaging learning challenge.

The environment uses the classic 20x10 grid dimension that has been standard since the original 1984 release. A key design decision was the simplification of the piece set to five tetrominoes (I, O, T, L, J), deliberately excluding S and Z pieces. This choice was motivated by Brzustowski's (1992) analysis showing that these pieces can force eventual game termination, and their exclusion allows the agent to focus on learning fundamental stacking and line-clearing strategies during the initial training phase.

The state representation uses a simple array structure where each cell in the 20x10 grid contains either a 0 (empty) or a 255 (filled). The action space is implemented with four fundamental moves: left translation, right translation, clockwise rotation and hard drop,

This simplified action set maintains all strategically significant options while reducing the complexity of the learning task. The hard drop action, in particular, helps accelerate training by reducing the number of steps required to place each piece.

Following Stevens and Pradhan's (2016) work, our reward structure uses a fitness function that evaluates board state quality based on multiple factors, such as stacking height, line completions, holes, and surface smoothness. This function balances immediate rewards for clearing lines with penalties for creating unfavorable board conditions, encouraging efficient and strategic gameplay.

The environment tracks several key metrics during gameplay to enable comprehensive evaluation of agent performance:

- Lines cleared: Total number of rows eliminated
- Pieces placed: Count of successfully placed tetrominoes
- Board statistics:
    - Bumpiness: Sum of height differences between adjacent columns
    - Holes: Count of empty cells with filled cells above them
    - Column heights: Tracking minimum, maximum, and aggregate heights

These metrics serve both as training feedback and evaluation criteria, allowing detailed analysis of agent behaviour and learning progress.

This environment design creates a framework that simplifies the learning task while maintaining the strategic depth that makes Tetris an interesting challenge for reinforcement learning. The standardised interface ensures compatibility with existing reinforcement learning algorithms while the comprehensive metrics enable thorough analysis of agent performance.

3.3 Architecture Model Design

The deep learning architecture for the Tetris agent is built around a Deep Q-Network (DQN) model, which processes game states and estimates optimal actions for the agent. This design leverages convolutional neural networks (CNNs) to automatically extract meaningful features from the spatial structure of the game board, an approach inspired by the success of similar techniques in game AI research (e.g., Atari games by Mnih et al., 2015).

The CNN component transforms raw game board observations into a compact, high-dimensional feature representation, capturing spatial patterns such as potential line completions, stacking strategies, and problematic areas like holes. This automatic feature extraction eliminates the need for manually engineered features, enabling the agent to discover strategies through gameplay experience.

The extracted features are passed to fully connected layers that estimate Q-values for all possible actions. This dual structure—feature extraction by the CNN and decision-making through dense layers—enables the agent to process complex game states and learn effective strategies in a scalable and efficient manner.

To stabilize training, the architecture incorporates experience replay and a target network. These mechanisms mitigate the challenges of correlated data and non-stationary targets, ensuring robust learning in the dynamic environment of Tetris.

This architecture balances computational efficiency and learning capacity, offering a foundation capable of handling the vast state space and real-time decision-making demands of Tetris. By combining CNNs with reinforcement learning principles, the model effectively

addresses the challenges of spatial reasoning and sequential decision-making inherent in the game.

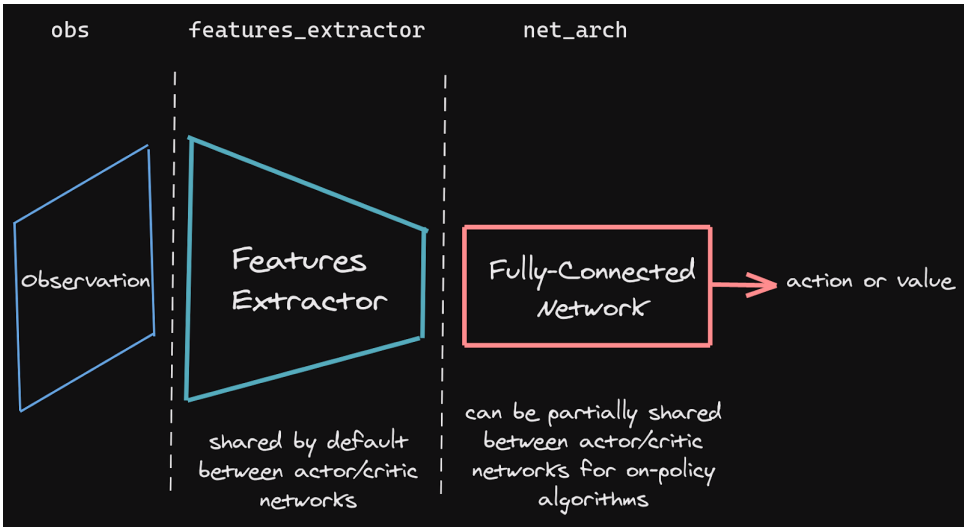


Figure 11 - Stable Baselines3 NN Architecture

Figure 12 illustrates the complete network architecture, showing the progression from input state through feature extraction and Q-value estimation stages to the final action outputs.

This architecture provides a solid foundation for learning complex Tetris strategies while maintaining efficient computation and stable learning dynamics. The careful balance of feature extraction and Q-learning components creates a system capable of capturing both the spatial patterns and strategic elements necessary for successful Tetris gameplay.

3.4 Training design

The training system implements a comprehensive approach to developing and evaluating deep reinforcement learning agents for Tetris. The design centers around three key components: the core learning architecture, experience management, and evaluation framework.

The learning architecture employs a Deep Q-Network (DQN) algorithm, selected for its proven effectiveness in discrete action spaces like Tetris. The design incorporates an ε-greedy exploration strategy with decay over time, balancing the need for initial environment exploration with eventual exploitation of learned strategies.

Experience replay serves as a crucial component of the training system. The design includes a replay buffer to store and learn from past experiences, enabling the agent to revisit and learn from important game situations multiple times. This approach helps break temporal correlations in the training data and improves learning stability.

The training pipeline leverages parallel environment processing to accelerate learning and provide diverse experience data. Multiple game instances run simultaneously, allowing the agent to encounter a broader range of game states and scenarios during training. This parallel approach not only speeds up the training process but also contributes to more robust strategy development.

The evaluation framework tracks both standard reinforcement learning metrics and game-specific indicators. Beyond conventional episode rewards and training losses, the system monitors Tetris-specific metrics including lines cleared, pieces placed, and board state characteristics. This comprehensive monitoring enables thorough analysis of agent development throughout the training process.

To ensure reproducibility and systematic experimentation, the design includes a configuration-based experiment management system. This approach enables clear specification of experimental parameters and maintains consistent testing conditions across different training runs. Each experiment maintains its own logging and checkpoint system for tracking progress and preserving promising models.

The training progression follows several distinct phases, from initial exploration of game mechanics to the development of sophisticated strategies. The design implements early stopping mechanisms based on performance metrics, optimizing training duration while preventing potential degradation of learned strategies. Regular model checkpoints enable analysis of strategy development over time.

This training design creates a robust framework for developing and evaluating Tetris agents. The combination of proven learning algorithms, efficient experience collection, and comprehensive evaluation metrics enables detailed analysis of agent performance while maintaining reproducibility of results.

3.5 Evaluation Design

The evaluation design establishes a framework for comprehensively assessing the agent's performance through both quantitative measurements and visual analysis techniques. The system tracks four core performance indicators: score progression, cleared lines, pieces placed, and survival time. These fundamental metrics are supplemented by sophisticated board state analysis including surface smoothness, hole formation, and height distributions.

The visualization design incorporates three complementary modes: a lightweight text display for development, a graphical interface for detailed analysis, and a programmatic interface for automated evaluation. This multi-modal approach enables both rapid feedback during development and thorough analysis of learned strategies.

Following established practices in reinforcement learning evaluation (Sutton and Barto, 2018), the system includes a systematic comparison framework using random-play agents as baselines. This provides clear benchmarks for measuring improvement while accounting for the stochastic nature of both learning and gameplay. The evaluation infrastructure splits into automated and interactive components, enabling both systematic data collection and detailed behavioral analysis.

This comprehensive design creates a framework that reveals not just whether the agent learns, but how it develops and applies strategies during gameplay. The combination of quantitative metrics and qualitative analysis tools provides the necessary insight into both performance outcomes and strategic development.

3.6 Conclusions

The design of our Tetris-playing agent introduces practical optimizations for the specific challenges of Tetris gameplay. The system architecture employs a modular design pattern that

separates three key components: the core game mechanics, learning algorithms, and evaluation framework. This separation facilitates both development iteration and empirical analysis.

The environment implementation captures the fundamental strategic elements of Tetris while providing a standardised interface for agent-environment interaction. We incorporate both visual and command-line interfaces for development and debugging, complemented by a comprehensive metrics collection system for quantitative performance analysis.

The modular architecture enables straightforward extension through multiple vectors: integration of alternative learning algorithms, expansion of game mechanics, and refinement of evaluation metrics. The systematic approach to both architecture and evaluation creates a robust foundation for investigating key questions in reinforcement learning and artificial game intelligence.

4 Development

4.1 Introduction

This chapter details the implementation of our deep reinforcement learning agent for Tetris, documenting the successes and challenges encountered during development. Built upon established frameworks including Stable Baselines 3 (Raffin et al., 2021), Gymnasium (Towers et al., 2024), and PyTorch (Paszke et al., 2019), our implementation journey covered several critical components: the core Tetris game engine, reinforcement learning environment, deep learning architecture, optimizations for performant environments, and comprehensive monitoring and visualization systems.

The chapter begins with an examination of the Tetris game engine implementation, including state representation, action space design, and core gameplay mechanics. We then explore the development of our custom Gymnasium environment, which provides the standardized interface necessary for reinforcement learning. The deep learning implementation section follows, detailing our DQN architecture and the custom convolutional neural networks developed specifically for processing Tetris game states. Subsequently, we discuss the performance optimizations implemented, including vectorized environments and hardware acceleration capabilities. The chapter concludes with our approaches to monitoring and visualization, detailing the systems developed for tracking training progress and analyzing agent behavior.

This chapter aims to systematically outline the design and development of each major component, highlighting how they collectively contribute to achieving a high-performing Tetris-playing agent. Through this examination, we explore not just what was implemented, but why specific approaches were chosen and how they evolved through the development process.

4.2 Game implementation

In implementing the Tetris game core, we faced several interesting challenges in translating the design principles established in Chapter 3 into working code. Our journey through this implementation revealed important insights about performance optimization and system architecture that shaped our final solution.

Our first major challenge was finding an efficient way to represent and manipulate the game board. The sheer scale of this challenge became apparent when considering Algorta and Simsek's (2019) observation that Tetris has an estimated state space of 7 × 2^200. We chose to build our foundation on a 20x10 NumPy array, developing a dual-purpose encoding system where 0 represents empty cells and values 1-7 represent different tetromino types. This elegant solution allowed us to maintain simple binary checks for game logic while simultaneously supporting colour coding for visualisation. The action space is clearly defined through an enumeration:

```
@unique
class Actions(IntEnum):
        """
        Enum class representing possible actions in the Tetris game.
        """
        LEFT = 0
```

```
RIGHT = 1
ROTATE = 2
HARD_DROP = 3
```

Figure 12 - Enum representing tetromino actions

This enum provides a clean interface between the environment and agent while ensuring type safety. The HARD_DROP action was included to accelerate training by reducing the number of steps required to place each piece.

We structured our implementation to maintain a clear separation between the environment and learning agent. This led us to develop the TetrisGame class as a central coordinator, handling the complex interplay of game mechanics through specialised methods. We found that representing each tetromino as a 2D NumPy array, combined with an integer-based colouring system, provided an elegant solution for both identification and rendering purposes.

To speed up the game engine we precomputed all possible tetromino rotations. By storing each piece's rotated versions (0°, 90°, 180°, 270°) in a lookup dictionary, we eliminated the need for runtime rotation calculations, improving performance during gameplay while maintaining our ability to modify rotation behaviour as needed.

A crucial component of our implementation is the comprehensive metrics tracking system, which provides essential feedback for evaluating board states and guiding the learning process. The _analyze_board method performs real-time analysis of critical board characteristics, employing efficient column-wise traversal to calculate multiple metrics simultaneously. For each column, the method tracks the height of the highest occupied cell, counts holes (empty cells with filled cells above them), and calculates surface smoothness through height differences between adjacent columns. This approach, inspired by Dellacherie's successful heuristics (Fahey, 2003), enables the computation of five key metrics: bumpiness (aggregate height differences between adjacent columns), total holes, column heights array, minimum height, maximum height, and aggregate height. The metrics computation avoids expensive matrix operations by utilizing direct array indexing and maintains O(n) complexity where n represents the total number of cells in the grid. These metrics prove invaluable both for reward calculation during training and for analyzing the agent's strategic development over time.

Building upon established Tetris scoring conventions, we implemented a two-tier reward system that combines immediate action feedback with strategic gameplay incentives. Following classic Tetris scoring rules (Carr, 2005), the system awards points for line clearances using an exponential scale: 40 points for a single line, 100 for two lines, 300 for three lines, and 1200 points for a tetris (four lines cleared simultaneously). Additionally, our implementation includes a small score bonus for hard drops, awarding one point per cell the piece travels downward, encouraging efficient piece placement while maintaining strategic depth.

Throughout our development process, we made several surprising discoveries about performance optimization. Perhaps most unexpected was our finding that traditional loop-based operations significantly outperformed vectorized NumPy operations for our specific board size (20x10) and tetromino pieces (4x4 maximum). This insight led us to adopt a hybrid approach, using NumPy for state representation while relying on efficient loop-based operations for game mechanics.

```
	def _is_valid_position(self, position: Tuple[int, int], tetromino:
np.ndarray) -> bool:
	"""Check if the tetromino position is valid."""
	assert self.grid is not None, "self.grid should not be None"
	y, x = position
	height, width = tetromino.shape

	if y < 0 or y + height > self.board_height or x < 0 or x + width >
self.board_width:
		return False

	for i, row in enumerate(tetromino):
		for j, cell in enumerate(row):
		if cell and self.grid[y + i, x + j]:
			return False
	return True
```

Figure 13 - Loop based collision detection

We created both random and bag-based piece generation systems, with the latter following the standard 7-bag approach common in modern Tetris implementations. The decision to implement this system through an abstract base class with concrete strategy implementations proved valuable, allowing us to easily experiment with different generation approaches while maintaining consistent behaviour across our experiments.

```
class TetrominoGenerator(ABC):
	"""
	Abstract base class for tetromino generators that defines the common
	Interface and shared initialization logic.
	"""

	def __init__(self, tetrominoes: List[str], rng: Generator):
	"""
	Initialize the generator with a list of tetrominoes and a random number
	generator.

	Args:
		tetrominoes: List of tetromino types
		rng: NumPy random number generator
	"""
	  self.tetrominoes = tetrominoes
	  self.rng = rng
	  self.bag: List[str] = []

	@abstractmethod
	def get_next_tetromino(self) -> str:
	"""Get the next tetromino from the generator."""
	  pass

	def reset(self, rng: Generator):
	"""Reset the generator"""
	self.rng = rng
```

Figure 14 - Base random tetromino generator

The integration of our Tetris implementation with the Gymnasium framework (Towers et al., 2024) formed a foundational component of our reinforcement learning system. Following established software best practices, we developed a hierarchical environment structure that promoted modularity while maintaining strict adherence to standardized interfaces.

At the core of our implementation lies the BaseTetrisEnv class, which establishes the fundamental interaction layer between the learning agent and the game engine. This base class implements the standardized Gymnasium interface and defines the action space using gymnasium spaces.Discrete type, representing the four possible moves: left translation, right translation, rotation, and hard drop. The decision to use a discrete action space reflects the natural discreteness of Tetris gameplay mechanics.

```
class BaseTetrisEnv(gym.Env):
       """Base Tetris Gym Environment"""

       metadata = {
         "render_modes": ["pygame", "ansi", "rgb_array", "human"],
         "render_fps": 4,
       }

       def __init__(
         self,
         grid_size=(20, 10),
         tetrominoes: Optional[List[str]] = None,
         render_mode=None,
         piece_gen=None,
       ):
         super().__init__()

         self.render_mode = render_mode
         self.grid_size = grid_size
         self.tetrominoes = tetrominoes
         self.ticks_per_drop = ticks_per_drop
         self.state: Optional[Dict] = None
         self.game = TetrisGame(
              grid_size=self.grid_size,
              tetrominoes=self.tetrominoes,
              rng=self.np_random,
              piece_gen=piece_gen,
         )

         self.renderer: Optional[TetrisRenderer] = None
         # set renderer

         self.action_space = spaces.Discrete(len(Actions))

       def reset(self, seed=None, options=None) -> Tuple[np.ndarray | Dict,
Dict]:
       """Reset the environment to start a new game."""
         super().reset(seed=seed)
         self.game.rng = self.np_random
         self.state = self.game.reset()
         return self._get_observation(), self._get_info()

       def step(self, action: int):
         self.state, game_over, drop_distance, lines_cleared =
self.game.step(Actions(action))
         reward = self.calculate_reward(game_over, drop_distance, lines_cleared)

         observation = self._get_observation()
         info = self._get_info()

         # observation, reward, terminated, truncated, info
         return observation, reward, game_over, False, info

       def _get_info(self) -> Dict[str, Any]:
         assert self.state is not None, "self.state should not be None"
         return {
              "score": self.state["score"],
```

```
            "lines_cleared": self.state["lines_cleared"],
            "pieces_placed": self.state["pieces_placed"],
            "current_ticks": self.state["current_ticks"],
            **self.state["metrics"],
        }

    def render(self):
        """Render the current state of the game."""
        assert self.state is not None, "self.state should not be None"
        if self.renderer is not None:
            return self.renderer.render(self.state)

    def _get_observation(self):
        """Observation return"""
        pass

    def calculate_reward(self, game_over, drop_distance, lines_cleared):
        """Reward calculation"""
        pass
```

Figure 15 - Base gym environment

Building upon this foundation, we implemented the ImageTetrisBaseEnv class to provide state representation as grayscale images:

```
class ImageTetrisBaseEnv(BaseTetrisEnv):
    """Tetris environment with image observation"""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.observation_space = spaces.Box(
                low=0,
                high=255,
                shape=(1, self.grid_size[0], self.grid_size[1]),
                dtype=np.uint8,
        )
    def _get_observation(self) -> np.ndarray:
        """Return the current state of the game grid."""
        grid = self.state["grid"].copy()
        np.putmask(grid, grid > 0, 255)
        return np.expand_dims(grid, axis=0).astype(np.uint8)
```

Figure 16 - Tetris Environment with image observation

This representation strategy follows the approach demonstrated by Mnih et al. (2013) in their seminal work on deep reinforcement learning for Atari games, where raw visual input proved effective for learning complex game strategies. The single-channel grayscale representation reduces input dimensionality while preserving essential spatial information about the game state.

The final layer of our environment hierarchy, ImgHTetrisEnv, implements the specific reward structure that guides the learning process. Our reward function combines immediate gameplay feedback with strategic considerations:

```
class ImgHTetrisEnv(ImageTetrisBaseEnv):
    """Image observation and heuristic award"""
    GAME_OVER_PENALTY = -50
    HEIGHT_PENALTY = -0.51
    LINE_REWARD = 0.76
    HOLE_PENALTY = -0.36
    BUMPINESS_PENALTY = -0.18
```

```
def calculate_reward(self, game_over, drop_distance, lines_cleared):
  """Custom reward function."""
  reward = 0
  holes = self.state["metrics"]["holes"]
  bumpiness = self.state["metrics"]["bumpiness"]
  height = self.state["metrics"]["sum_height"]

  new_fitness = (
       self.HEIGHT_PENALTY * height
       + self.LINE_REWARD * lines_cleared
       + self.HOLE_PENALTY * holes
       + self.BUMPINESS_PENALTY * bumpiness
  )
  change_in_fitness = new_fitness - self.fitness
  self.fitness = new_fitness
  reward += change_in_fitness

  if game_over:
       reward += self.GAME_OVER_PENALTY

  return reward
```

Figure 17 - Tetris Environment reward definition

This reward formulation follows the fitness function demonstrated by Stevens and Pradhan (2016) in their Tetris implementation, balancing immediate rewards for line clearance against strategic penalties for unfavorable board states.

Through this hierarchical design, our implementation achieves both modularity and extensibility while maintaining compatibility with modern reinforcement learning frameworks. This architecture facilitated systematic experimentation with different state representations and reward structures throughout our research.

Our implementation journey demonstrated the practical value of our core design principles. The modular structure we adopted allowed us to independently test and modify components, while our extensible architecture supported the addition of new features without disrupting existing functionality. The comprehensive monitoring capabilities we built into the system proved invaluable during development and testing, providing clear insights into system behaviour and learning progress.

Through careful attention to these principles, we created a robust foundation for exploring deep reinforcement learning in Tetris. Our implementation not only met our immediate research needs but also provided the flexibility to adapt and extend the system as our understanding of the problem domain grew.

4.3 Deep Learning Implementation

Our deep learning architecture combines a specialized convolutional neural network for feature extraction with a Q-network for action-value estimation. The TFEDLR feature extractor, inspired by Stevens and Pradhan's (2016) successful CNN architecture for Tetris and Mnih et al.'s (2013) work on deep reinforcement learning, processes raw game states through convolutional layers to learn relevant spatial patterns. This extracted representation feeds into a Q-network with fully connected layers that estimate action values. The architecture enables end-to-end learning of both feature extraction and value estimation through environmental interaction, building on established approaches while addressing Tetris-specific challenges.

The TFEDLR feature extractor is implemented as a subclass of the BaseFeaturesExtractor class from the Stable Baselines 3 library. This allows the CNN to be seamlessly integrated into the DQN agent's policy network, ensuring compatibility with the broader reinforcement learning framework.

The CNN architecture of the TFEDLR extractor consists of the following components:

Convolutional Layers: The extractor employs three convolutional layers, each with a different number of kernels and kernel sizes. The first layer uses 32 kernels of size 5x5, the second layer uses 64 kernels of size 3x3, and the third layer uses 64 kernels of size 3x3. This progression of kernel sizes and depths is intended to capture increasingly complex features in the input observations.

Activation Function: The ReLU (Rectified Linear Unit) activation function is applied after each convolutional layer to introduce non-linearity and help the network learn more complex representations.

Pooling Layers: Two max-pooling layers each with a 2x2 kernel and a stride of 2 are applied after second and third convolutional layers. These pooling layers reduce the spatial dimensions of the feature maps, allowing the network to capture more abstract and higher-level features.

Flattening: The output of the final convolutional layer is flattened to a one-dimensional vector, preparing the features for the subsequent linear layers.

Linear Layer: A fully connected (linear) is added after the convolutional portion of the network. This linear layer maps the flattened convolutional features to the desired feature dimension (e.g., 256)

```
class TFEDLR(BaseFeaturesExtractor):
       """
       A custom feature extractor for Tetris game observations using a
Convolutional Neural Network (CNN).
       The network consists of three convolutional layers with max pooling
followed by a linear layer.
       """

       def __init__(
         self,
         observation_space: spaces.Box,
         features_dim: int = 256,
         num_kernels: tuple[int, int, int] = (32, 64, 64),
         kernel_sizes: tuple[int, int, int] = (5, 3, 3),
         pooling_kernel_sizes: tuple[int, int] = (2, 2),
       ):
         super().__init__(observation_space, features_dim)
         n_chan = observation_space.shape[0]

         self.cnn = nn.Sequential(
              # First conv layer
              nn.Conv2d(n_chan, num_kernels[0], kernel_size=kernel_sizes[0],
stride=1, padding=1),
              nn.ReLU(),
              # Second conv layer
              nn.Conv2d(num_kernels[0], num_kernels[1],
kernel_size=kernel_sizes[1], stride=1, padding=1),
              nn.ReLU(),
              nn.MaxPool2d(kernel_size=pooling_kernel_sizes[0], stride=2),
```

```
            # Third conv layer
            nn.Conv2d(num_kernels[1], num_kernels[2],
kernel_size=kernel_sizes[2], stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=pooling_kernel_sizes[1], stride=2),
            nn.Flatten(),
        )

        with torch.no_grad():  # Don't track operations, we just need to
calculate the output size
            example_input =
torch.as_tensor(observation_space.sample()[None]).float()
            cnn_output_dim = self.cnn(example_input).shape[1]

        self.linear = nn.Sequential(nn.Linear(cnn_output_dim, features_dim),
nn.ReLU())

    def forward(self, observations) -> torch.Tensor:
        """Process board through CNN and linear"""
        board_features = self.cnn(observations)
        return self.linear(board_features)
```

Figure 18 - Custom Feature Extractor

The TFEDLR feature extractor functions as a core component of the neural network architecture, integrated directly into the DQN agent's policy network. Following the convolutional layers of TFEDLR, the network architecture includes two fully-connected layers, each containing 256 units, which serve as the Q-network for action-value estimation. This combination of convolutional feature extraction followed by fully-connected layers for value estimation creates a complete neural network capable of processing the raw game state and predicting action values.

```
    policy_kwargs = {
        "features_extractor_class": TFEDLR,
        "features_extractor_kwargs": {"features_dim": 256},
        "net_arch":[256, 256],
        "activation_fn": torch.nn.ReLU,
    }

    model = DQN(
        "CnnPolicy",
        env,
        policy_kwargs=policy_kwargs,
        …
    )
```

Figure 19 - Custom feature extractor integration with the model

By using the features_extractor_class and features_extractor_kwargs arguments, the Stable Baselines 3 DQN implementation is able to seamlessly integrate the custom TFEDLR feature extractor into the policy network. This allows the agent to leverage the specialized feature representation learned by the CNN while benefiting from the robust DQN training algorithm and infrastructure provided by the Stable Baselines 3 library.

The design of the TFEDLR feature extractor, with its combination of convolutional and linear layers, aims to capture the essential spatial and structural characteristics of the Tetris game board that are crucial for effective decision-making. The progressive increase in kernel sizes and depths, combined with pooling layers, enables the network to extract a rich set of spatial

and structural features from the input observations, which are then passed on to the Q-value estimation components of the DQN agent.

Custom feature extractor, integrated within the broader DQN framework, forms a key part of the overall deep reinforcement learning system developed for the Tetris game agent.

The DQN model's configuration incorporated several carefully tuned hyperparameters based on empirical testing and established research. Following Mnih et al. (2013)'s foundational work, we implemented experience replay with a substantial buffer size of 500,000 transitions, enabling the agent to learn from a diverse set of historical experiences. The learning process utilized a learning rate of 5e-5 with the Adam optimizer, and a discount factor (γ) of 0.99 to balance immediate and future rewards. Training occurred in mini-batches of 64 samples, with the target network updated every 2000 steps to maintain stability, a technique that proved effective in stabilizing deep reinforcement learning training (Mnih et al., 2013). The exploration strategy employed an ε-greedy approach with initial exploration rate (ε) of 1.0, decreasing linearly to 0.01 over 30% of the total training duration. The policy network architecture consisted of two fully connected layers of 256 units each following the feature extractor, with ReLU activation functions, adapting the approach demonstrated by Stevens and Pradhan (2016) in their Tetris implementation. To ensure stable initial learning, the agent accumulated 50,000 steps of experience before beginning training updates, allowing the replay buffer to populate with a representative sample of experiences. The system supported checkpointing at 10,000-step intervals and included buffer saving capabilities, enabling training interruption and resumption while maintaining learning progress. This configuration proved effective in balancing the exploration-exploitation trade-off while maintaining stable learning dynamics throughout the training process.

4.4 Performant environments

Our implementation leveraged Stable-Baselines3's vectorized environment capabilities to enhance training efficiency. Vectorized environments enable simultaneous interaction with multiple environment instances, allowing the agent to gather more diverse experiences per training step compared to single-environment training (Raffin et al., 2021).

We implemented support for both DummyVecEnv and SubprocVecEnv parallel environment wrappers. While SubprocVecEnv distributes environments across separate processes, our initial experiments revealed that its synchronization overhead outweighed the benefits for our computationally lightweight Tetris implementation. DummyVecEnv, which creates multiple environment instances that run sequentially in the main process, proved sufficient for gathering diverse experiences while avoiding the inter-process communication overhead associated with SubprocVecEnv.

```
env_kwargs = … # parameters for the Tetris environment
vec_env_cls = SubprocVecEnv if args.subproc else DummyVecEnv
env = make_vec_env("Tetris-imgh", env_kwargs=env_kwargs, n_envs=4,
vec_env_cls=vec_env_cls)
```

Figure 20 - Vectorised environments

This design allowed for easy switching between parallel implementation strategies while maintaining consistent environment behavior. The number of parallel environments could be configured through command-line arguments, providing flexibility for different training

scenarios and computational resources.
Our implementation includes automatic hardware acceleration capabilities, dynamically selecting the most efficient computational backend available on the system. The following code demonstrates this functionality:

```
device = "cpu"
if torch.backends.mps.is_available():
       device = "mps"
elif torch.cuda.is_available():
       # PyTorch config for better GPU usage
       torch.backends.cudnn.benchmark = True  # Enable auto-tuner
       torch.backends.cuda.matmul.allow_tf32 = True  # Allow TF32 on Ampere
       torch.backends.cudnn.allow_tf32 = True
       device = "cuda"
```

Figure 21 - Automatic acceleration detection

This architecture enables seamless utilization of hardware acceleration across different platforms. On systems with NVIDIA GPUs, the implementation leverages CUDA capabilities, while on Apple Macs, it utilizes Metal Performance Shaders (MPS) for acceleration. For NVIDIA GPUs, we implemented additional optimizations including enabling the cuDNN auto-tuner and TF32 precision on compatible hardware. When no specialized hardware is available, the system gracefully falls back to CPU computation, ensuring broad compatibility across different computing environments.

The automatic backend selection enhances the system's portability while maintaining optimal performance characteristics for each platform. This proved particularly valuable during development and experimentation across different computing environments, allowing us to leverage available hardware acceleration without requiring manual configuration.

4.5 Monitoring System

We implemented a comprehensive monitoring system that tracks both training metrics and gameplay performance through multiple complementary approaches.

At the core of our monitoring infrastructure lies TensorBoard, already tightly integrated with Stable Baselines 3 because of its robust capabilities in visualizing training dynamics in real-time. The system tracks essential performance indicators throughout training, encompassing both gameplay outcomes like episode scores, lines cleared, and pieces placed, as well as board state characteristics such as bumpiness and hole counts. Board state metrics, currently recorded at the end of the game when the agent has already lost, could be improved by tracking their averages throughout the entire gameplay. Nevertheless they have proved useful at tracking the agent progress. Beyond these game-specific metrics, the system monitors fundamental training indicators including loss values and exploration rates through a custom callback system.

To capture these gameplay-specific metrics, we implemented a custom callback that integrates with the Stable Baselines 3's training loop. This callback triggers at each episode completion, gathering metrics from the environment and logging them through TensorBoard's event writing system. All metrics are written as tfevents files into the log directory, enabling both real-time monitoring during training and post-hoc analysis.

```
class EpisodeEndMetricsCallback(BaseCallback):
```

```
        """
        Custom callback for logging episode score and lines cleared to TensorBoard
only at the end of each game.
        """
        def _on_step(self) -> bool:
        # Check if any environment is done (end of episode)
        for i, done in enumerate(self.locals["dones"]):
                if done:
                info = self.locals["infos"][i]
                score = info.get("score", 0)
                lines_cleared = info.get("lines_cleared", 0)
                pieces_placed = info.get("pieces_placed", 0)
                bumpiness = info.get("bumpiness", 0)
                holes = info.get("holes", 0)
                # Log to TensorBoard at the end of the episode
                self.logger.record_mean("episode/score", score)
                self.logger.record_mean("episode/lines_cleared", lines_cleared)
                self.logger.record_mean("episode/pieces_placed", pieces_placed)
                self.logger.record_mean("episode/bumpiness", bumpiness)
                self.logger.record_mean("episode/holes", holes)
        return True…
        )
```

Figure 22 - Custom callback mechanism

To analyze the collected metrics, we developed three complementary visualization approaches. The primary interface uses TensorBoard's web-based solution, providing interactive plots and the ability to compare multiple training runs. For more detailed analysis, we created custom Jupyter notebooks that load the tfevents files directly, enabling sophisticated statistical analysis and custom visualizations using libraries like matplotlib and seaborn.

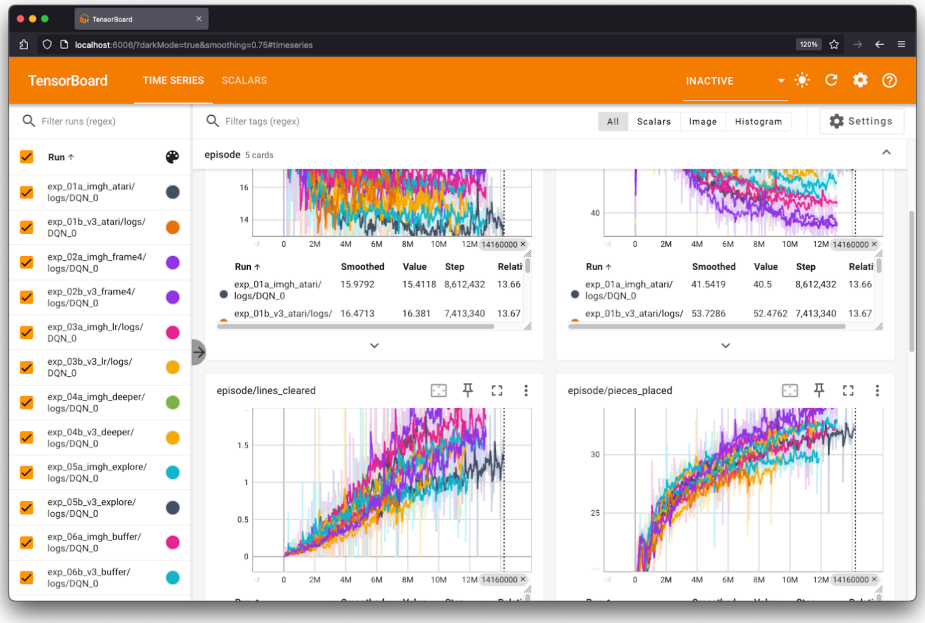


Figure 23 - Tensorboard web interface

Recognizing the need for monitoring training progress in resource-constrained environments like remote servers, we also implemented a lightweight console-based visualization system both using raw ascii characters and plotille library.. This allows us to generate ASCII-based plots directly in the terminal, providing essential feedback about training progress without requiring a graphical interface or web browser.

This system proved invaluable during development, as it helped us identify and address several key issues. For instance, it revealed that the agent initially struggled with exploration, leading us to adjust the exploration fraction parameters. The multi-faceted monitoring approach also helped validate our reward function design by showing clear correlations between improving gameplay metrics and decreasing loss values.

This comprehensive monitoring system follows best practices established in deep reinforcement learning research (Raffin et al., 2021), while adding domain-specific metrics crucial for understanding Tetris gameplay. The combination of real-time monitoring and detailed post-hoc analysis capabilities provided the insights necessary to refine our training approach and verify the agent's learning progress.

4.6 Visualisation system

The development of an effective visualisation system played a crucial role in our Tetris agent's implementation. Recognizing the importance of both practical development and comprehensive analysis, we implemented a multi-mode rendering framework that catered to the diverse needs of our project.

Our visualisation system comprised three distinct rendering modes, each tailored to serve a specific purpose within the project's lifecycle. The first mode, the ASCII renderer was implemented to ensure broad compatibility across different computing environments without imposing additional dependencies. By representing the game state using standard text characters, this visualization method functions reliably even in environments limited to terminal-based output, such as remote servers or containerized deployments. While modern systems are generally capable of handling more sophisticated rendering methods without significant performance impact, the ASCII implementation provides a lightweight, portable solution that aids in development and debugging across diverse computing environments.

For the purposes of demonstrating the agent's gameplay to stakeholders and observers, we incorporated a PyGame-based rendering mode. This graphical interface presented a visually engaging and true-to-life representation of the Tetris game, complete with colour-coded tetromino pieces, animated gameplay, music and overlaid statistical information. The PyGame renderer's ability to capture the visual elegance of the Tetris game was invaluable in showcasing the agent's learned strategies and conveying the complexity of the problem domain to a wider audience.

Finally, to facilitate in-depth analysis of the agent's decision-making process and performance characteristics, we implemented a third rendering mode that produced RGB array outputs. This format allowed for seamless integration with Jupyter notebooks and other data visualisation tools, enabling us to generate high-quality gameplay recordings and perform detailed post-hoc evaluations of the agent's behaviour.

The design of these three rendering modes was guided by the principle of a common interface, ensuring that the core functionality required for visualising the game state was

abstracted away from the specific implementation details of each mode. This approach allowed us to easily swap between rendering methods without the need to significantly modify other components of the system, promoting code reuse and maintainability.

At the heart of this common interface was the TetrisRenderer base class, which defined the essential methods and attributes required for rendering the game state. Subclasses of this base class, such as TetrisPyGameRenderer, TetrisASCIIRenderer, and TetrisRGBArrayRenderer, implemented the specific rendering logic for each mode, adhering to the shared interface and ensuring a consistent API for the rest of the system to interact with.

```
class TetrisRenderer(ABC):
        """Base class for Tetris renderers"""

        @abstractmethod
        def render(self):
        """Concrete implementation needs to implement this method"""
        pass

        @abstractmethod
        def close(self):
        """Concrete implementation needs to implement this method"""
        pass
```

Code 24 - Base Renderer class

By adopting this modular and extensible design for our visualisation system, we were able to efficiently address the varied needs of our Tetris agent development process. The seamless integration of these rendering modes allowed us to continuously monitor the agent's progress, effectively communicate its capabilities to stakeholders, and perform in-depth analyses to gain deeper insights into the learned strategies and decision-making process.

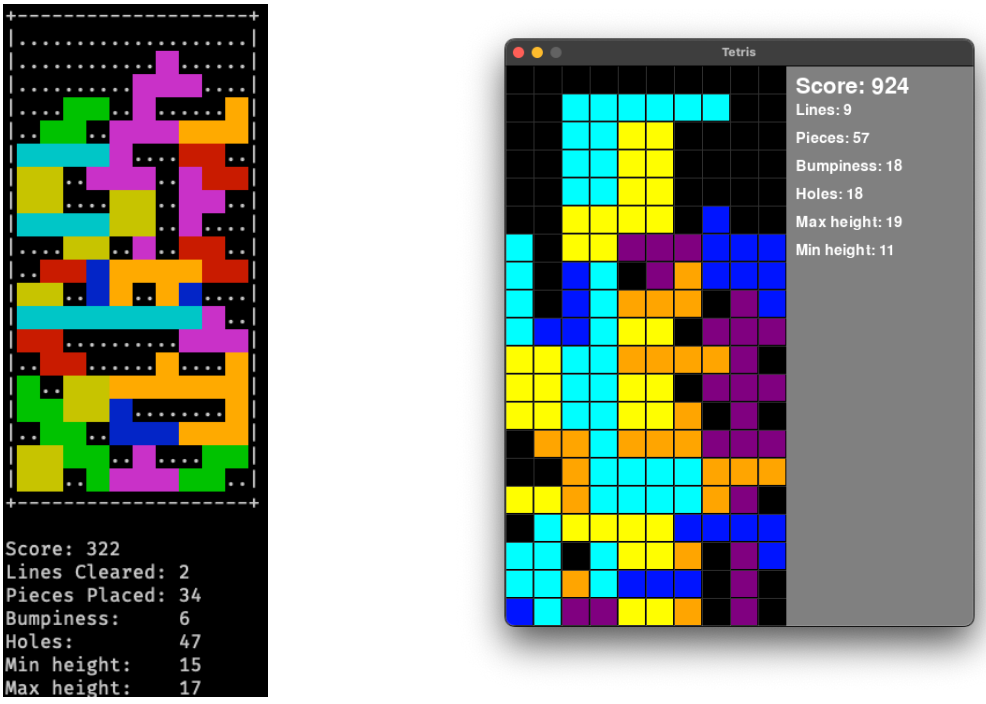


Figure 25 - ASCII and Pygame interfaces

4.7 Training Management

To facilitate systematic exploration of different training configurations and efficient resource utilization, we developed a comprehensive training management system. This system centers around a modular experiment orchestration framework that manages concurrent training runs while ensuring robust error handling and resource optimization.

Our framework utilizes YAML configuration files to define experiment parameters, as shown in Figure 26. Each configuration specifies key training parameters including the environment type, network architecture, learning rates, and exploration strategies. An example configuration is shown below:

```
---
experiments:
       - work_dir: "exp_01"
         env_name: "Tetris-imgh"
         num_envs: 4
         extractor_name: "TFEDLR"
         net_arch: [256, 256]
         buffer_size: 500000
         batch_size: 64
         learning_rate: 5e-5
         exploration_fraction: 0.3
         exploration_initial_eps: 1.0
         exploration_final_eps: 0.01
         tetrominoes: ["I", "O", "T", "L", "J"]
         checkpoint: true
         checkpoint_interval: 10000
         timestamps: 100000
```

Figure 26 - Example experiment configuration file

The system launches multiple training processes in parallel, with each process governed by its own configuration parameters. A monitoring subsystem continuously tracks process health and automatically restarts failed processes within predefined limits. Running multiple experiments concurrently enables efficient utilization of available GPU resources. The learning system includes automatic checkpointing and detailed logging to facilitate analysis of training progression and performance across different configurations.

The modular architecture separates experiment configuration from execution logic, enabling rapid iteration of different hyperparameter combinations without modifying core training code. This separation proved particularly valuable during our investigation of optimal network architectures and learning parameters.

This infrastructure enabled systematic exploration of the hyperparameter space while maintaining experimental reproducibility. The framework's reliability and flexibility significantly accelerated our investigation of effective training strategies for the Tetris environment, allowing us to efficiently identify promising configurations for further optimization.

4.8 Implementation infrastructure

Our implementation infrastructure provides comprehensive control over agent training through a sophisticated command-line interface and robust model persistence system. These components create a flexible yet reproducible experimental framework that supports systematic exploration of different training approaches.

The command-line interface implements a parameter configuration system using argparse python module. This system exposes all key training parameters through the command line, enabling parameter modification without any code changes:

```
parser.add_argument("--gamma", type=float, default=0.99)
parser.add_argument("--buffer-size", type=int, default=500000)
parser.add_argument("--batch-size", type=int, default=64)
parser.add_argument("--learning-rate", type=float, default=5e-5)
parser.add_argument("--exploration-fraction", type=float, default=0.3)
parser.add_argument("--exploration-initial-eps", type=float, default=1.0
...
```

Figure 27 - Argument parsing

The parameter system fully integrates with a YAML-based experiment framework that enables definition of parameter sets for systematic experimentation. Every configuration parameter in the experimentation system is translated to a command line parameter to be used with the learning system.

The model persistence system handles saving and loading of trained models while providing flexibility in training continuation. During model loading, the system allows modification of training parameters. This capability enables fine-tuning of training parameters between sessions, such as adjusting learning rates or exploration strategies, without losing trained weights.

The system supports most parameter modifications during training continuation, including changes to learning rates, exploration parameters, and environment configurations. However, structural changes to the model, such as modifying the feature extractor or network architecture, are not supported as they would require architecture reconciliation.

For example, adjusting the Tetris environment configuration and training parameters can be accomplished through simple command-line arguments:

```
python learn2.py --continue \
    --model-file model.zip \
    --learning-rate 1e-5 \
    --exploration-final-eps 0.05 \
    --env-name Tetris-imgh \
    --tetrominoes I O T L J S Z \
    --piece-gen rnd
```

Figure 28 - Argument parsing

The flexibility of this persistence system proves particularly valuable for transfer learning scenarios. For instance, an agent initially trained on a simplified set of tetrominoes (I, O, T, L, J) can be fine-tuned to handle the full piece set by loading the pre-trained model and adjusting the environment configuration to include S and Z pieces. This approach allows the agent to build upon previously learned strategies while adapting to new challenges following similar transfer learning principles demonstrated in other deep reinforcement learning applications

This infrastructure creates a robust foundation for experimentation while maintaining reproducibility, following software development best practice. The combination of flexible parameter management and comprehensive model persistence enables systematic exploration

of different training approaches while ensuring training progress can be preserved and modified as needed.

4.9 Issues

The development and experimentation phases of this project encountered several significant challenges, particularly in securing adequate computational resources for training the deep reinforcement learning models. Initially, development began on a MacBook Pro 2008, but this quickly proved inadequate for the computational demands of neural network training. The ageing hardware not only lacked raw processing power but also ran an older version of macOS that prevented the use of PyTorch MPS (Metal Performance Shaders) acceleration, eliminating any possibility of leveraging GPU acceleration on the local machine.

These limitations led to the exploration of cloud-based alternatives, beginning with Google Colab. While Colab initially seemed promising with its GPU and TPU support, the free credits were quickly exhausted. The platform's pricing structure proved problematic, offering compute units without clear indication of their translation to actual computation time. This lack of transparency made it difficult to plan and budget for extended training sessions.

The search for alternatives led to Lightning.ai, which offered an attractive package including a free "studio" CPU-based environment and 22 hours of GPU time. The platform's integration with Visual Studio Code through a web interface seemed ideal, allowing development and experimentation through a browser. However, the 4-hour runtime limitation on free sessions proved challenging, and several GPU hours were inadvertently consumed when sessions extended beyond this limit, resulting in unexpected charges.

These experiences highlighted the critical importance of finding a platform that could balance computational requirements with cost constraints. The search ultimately led to examining offerings from both runpod.io and vast.ai. After careful evaluation, vast.ai emerged as the optimal solution, despite its more complex setup requirements. The platform offered extremely competitive pricing with both on-demand and interruptible options, providing the flexibility needed for extended training sessions.

Beyond the computational challenges, one of the most significant risks identified at the project's outset was the compressed time frame coupled with limited availability for development. With development time primarily restricted to weekends, maintaining project momentum became a considerable challenge. To mitigate these constraints, significant evening work became necessary to achieve meaningful progress and maintain the high quality standards set for the project. While the initial environment setup was completed quickly with promising results, the learning process did not proceed as initially envisioned.

An early pivotal moment in the project came during the implementation of the DQN algorithm. The initial approach involved writing the reinforcement learning algorithm from scratch, but this proved to be more challenging than anticipated. The custom implementation was plagued by efficiency issues and bugs, and even when finally operational, its limitations became increasingly apparent. This led to a strategic decision to pivot to using Stable Baselines 3, a choice that aligned with the literature review findings. This decision, made early enough in the project timeline, prevented any significant impact on the overall schedule while providing valuable insight into the importance of standardised reinforcement learning libraries for research projects.

Perhaps one of the most challenging aspects of the project was managing scope and knowing when to stop implementing new features and experimenting with improvements. The constant temptation to add features or make improvements had to be balanced against the practical constraints of the project timeline. This required a conscious effort to avoid letting perfect become the enemy of done, particularly given the fixed timeline and limited availability. The experience emphasised the importance of clear scope definition and pragmatic approach to improvements in research projects with fixed constraints.

These challenges significantly influenced both the development process and final implementation approach. The experience provided valuable insights into the practical considerations of deploying deep reinforcement learning projects, particularly regarding the balance between computational resources, time management, and scope control. These lessons will prove valuable for future research efforts in this domain, highlighting the importance of careful planning and pragmatic decision-making in similar projects

4.10 Conclusions

The development phase of our Tetris-playing agent has yielded a robust and modular implementation that successfully integrates modern deep reinforcement learning techniques with classic game mechanics. Through careful architectural decisions and strategic use of established frameworks, we created a system that not only serves our immediate research needs but also provides a flexible foundation for future experimentation.

Our implementation's modular design proved particularly valuable during development, with clear separation between the core game mechanics, learning systems, and visualization components. This modularity enabled independent testing and refinement of each component, significantly accelerating the development cycle. The decision to build upon Gymnasium's standardized interface (Towers et al., 2024) for our environment implementation has ensured compatibility with a wide range of reinforcement learning algorithms while maintaining clean separation between the environment and agent logic.

The choice of Stable-Baselines3 as our reinforcement learning framework has been validated through development, providing reliable implementations of core algorithms while allowing the flexibility needed for our custom neural network architectures. The framework's built-in monitoring capabilities, combined with our custom callback system, enabled comprehensive tracking of training progress and agent behavior.

Our technical choices regarding state representation and reward structures were informed by previous research, particularly the work of Stevens and Pradhan (2016) in their DQN implementation for Tetris. The decision to use simple binary state representations, combined with carefully crafted reward functions, created an effective learning environment while maintaining computational efficiency.

The implementation challenges we encountered, particularly in managing computational resources and handling training stability, led to valuable insights and improvements in our system design. The development of the vectorized environment system, following the approach demonstrated by Raffin et al. (2021), significantly improved training efficiency while maintaining code clarity.

While our implementation successfully met its core objectives, it also revealed several areas for potential improvement. The current reward structure, while effective, could benefit from

more sophisticated shaping to better guide the learning process. Additionally, the visualization system, though functional, could be enhanced to provide more detailed insights into the agent's decision-making process.

These conclusions from the development phase have set a strong foundation for the evaluation and analysis presented in subsequent chapters, while also highlighting promising directions for future improvements to the system.

5 Testing and Evaluation

5.1 Introduction

This chapter presents a comprehensive evaluation of our Deep Q-Learning agent's performance in playing Tetris. The evaluation process examines both quantitative metrics and qualitative aspects of the agent's gameplay, providing insights into the effectiveness of our implementation and the learning process.

To ensure robust and reproducible results training was conducted on a rented GPU environment through vast.ai over a period of 5 days. The hardware configuration consisted of an NVIDIA RTX A4000 graphics card paired with an AMD Ryzen 5 5600X 6-core processor and 24GB of system RAM. This setup provided the computational power necessary for training deep neural networks while maintaining cost-effectiveness. The software environment was standardized using a PyTorch-based Docker container (pytorch/pytorch:2.5.1-cuda12.4-cudnn9-runtime), ensuring consistency across training sessions.

The implementation utilized Python 3.11.10 as the primary programming language, leveraging several key libraries including Stable Baselines3 2.3.2 for reinforcement learning algorithms, PyTorch 2.5.1 for neural network operations, and Gymnasium 1.0.0 for the Tetris environment. Additional supporting libraries included Pandas 2.2.3 and NumPy 1.26.4 for data manipulation, and TensorBoard 2.18.0 for monitoring training progress.

We conducted our evaluation across multiple episodes to account for the stochastic nature of both the learning process and Tetris gameplay. The evaluation metrics were carefully chosen to provide insights into different aspects of the agent's performance, including:

1. Score progression and final scores achieved
2. Number of lines cleared per episode
3. Survival time measured in pieces placed
4. Board state analysis including metrics like holes and bumpiness

The evaluation process incorporated both automated metrics collection and visual gameplay analysis, allowing us to understand not just how well the agent performed numerically, but also the strategies it developed. This comprehensive approach follows best practices established in recent deep reinforcement learning research (Algorta & Şimşek, 2019).

Results were gathered across 500 evaluation episodes to ensure statistical significance, with performance compared against a random-action baseline to quantify the improvement achieved through learning. The following sections detail our testing methodology, present the results obtained, and analyze the implications of our findings.

5.2 Baseline Comparison

Analysis of 500 training episodes reveals significant improvements in the DQN agent's gameplay compared to random actions. While the random agent failed to clear a single line across all episodes, the trained agent consistently cleared an average of 15.89 lines per game. This demonstrates successful learning of fundamental Tetris strategies.

The trained agent's survival capabilities proved substantially stronger, placing 70.77 pieces on average compared to the random agent's 12.83. Episode durations extended nearly 17-fold, from 44.97 to 796.86 timesteps, indicating sophisticated piece placement strategies and board management.

Score improvements show similar dramatic gains, with the DQN agent averaging 677.31 points versus 93.21 for random play. However, the relatively high standard deviation (±195.40) suggests some variability in performance, likely due to the stochastic nature of piece sequences and occasional suboptimal decision-making.

Interestingly, the cumulative reward metric shows an apparent decrease (-147.74 vs -125.12) despite improved gameplay performance. This seeming contradiction stems from the reward calculation mechanism - rewards are assessed per action rather than per game outcome. Since the DQN agent survives significantly longer, placing 5.5 times more pieces than the random agent, it accumulates more reward/penalty events. Each piece placement affects board fitness metrics like holes and bumpiness, generating corresponding rewards or penalties. A more meaningful comparison might be reward per piece placed: the DQN agent averages -2.09 reward per piece (-147.74/70.77) versus -9.75 for random play (-125.12/12.83), demonstrating more efficient gameplay despite the lower cumulative reward.

The consistency of these results across 500 episodes, particularly the zero standard deviation in random agent line clearing, provides strong statistical confidence in the DQN agent's learned capabilities.

Metric	DQN Agent	Random Agent	Improvement
Score	677.31 ± 195.40	93.21 ± 16.37	627%
Lines Cleared	15.89 ± 4.57	0.00 ± 0.00	∞
Pieces Placed	70.77 ± 11.37	12.83 ± 1.70	451%
Episode Length	796.86 ± 128.99	44.976 ± 12.72	1672%
Reward	-147.74 ± 4.17	-125.12 ± 12.88	-18%

Table 1 - Performance comparison over 500 episodes (mean ± standard deviation)

5.3 Performance Analysis

The training process revealed several interesting patterns in the agent's learning trajectory. Examining the training metrics over approximately 600 million timesteps provides valuable insights into the agent's development and performance characteristics.

The episode score (Figure 29) shows a clear upward trend, with the agent improving from initial random performance to achieving scores consistently above 600 points. However, the learning curve exhibits considerable variance, which is typical in deep reinforcement learning for complex environments like Tetris. This variance can be attributed to the stochastic nature of both the exploration process and the game mechanics themselves.

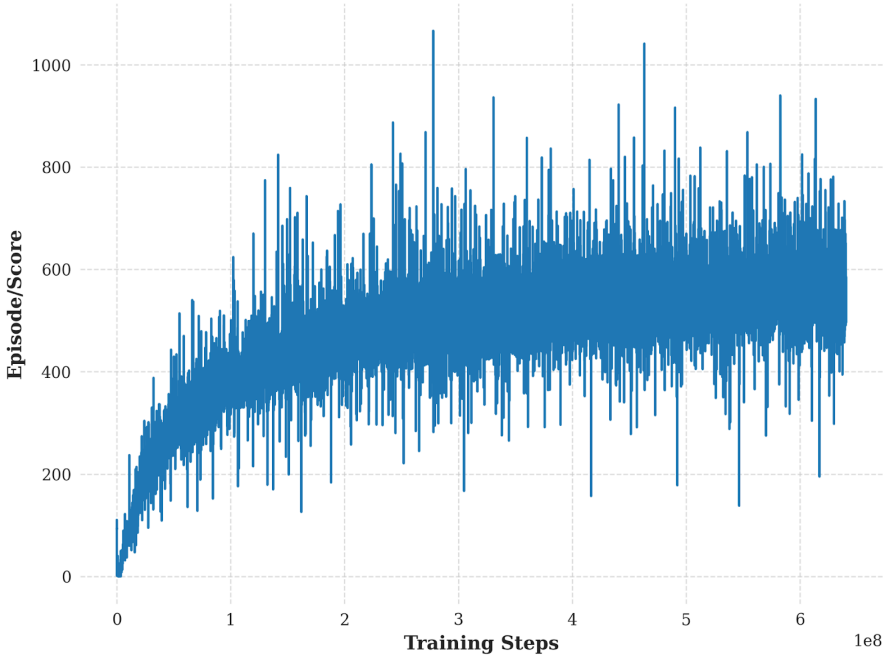


Figure 29 - Episode score

Line clearing ability (Figure 30) demonstrates similar improvement patterns to the overall score. The agent learned to clear multiple lines consistently, averaging approximately 15 lines per episode by the end of training. This metric's progression suggests the agent developed an understanding of piece placement strategies that extend beyond immediate rewards to achieve longer-term objectives.

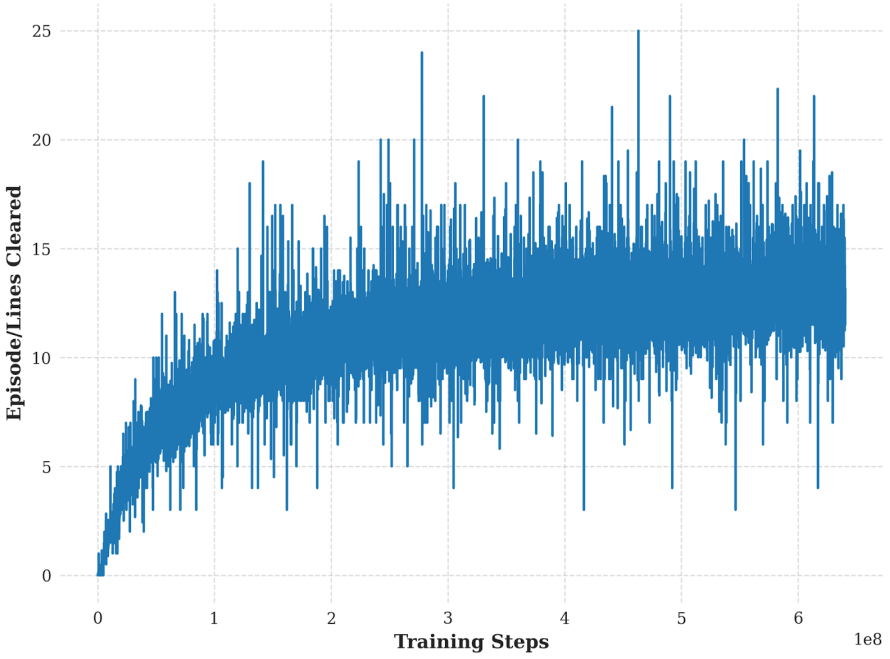


Figure 30 - Episode line clears

The pieces placed metric (Figure 31) provides perhaps the clearest indication of the agent's improving competence. From an initial baseline of around 20 pieces per episode, the agent learned to place approximately 70 pieces consistently, representing a substantial improvement in survival time and board management skills. This improvement aligns with findings from previous research showing that piece placement count serves as a reliable indicator of Tetris agent performance (Stevens and Pradhan, 2016).

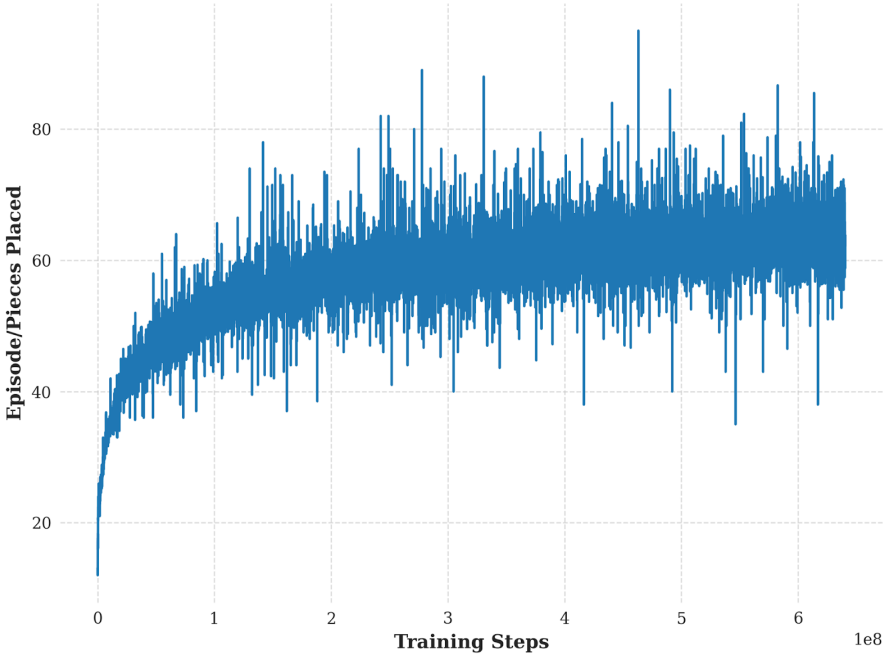


Figure 31 - Episode Pieces placed

Analysis of the end-of-episode metrics reveals interesting insights into the agent's strategic development. The bumpiness metric (Figure 32) shows a gradual decrease over time, indicating that the agent learned to maintain a more even surface profile - a key strategy for successful Tetris play (Carr, 2005). Similarly, the hole count metric (Figure 33) shows a downward trend, suggesting the agent learned to avoid creating covered empty spaces that limit future piece placement options.

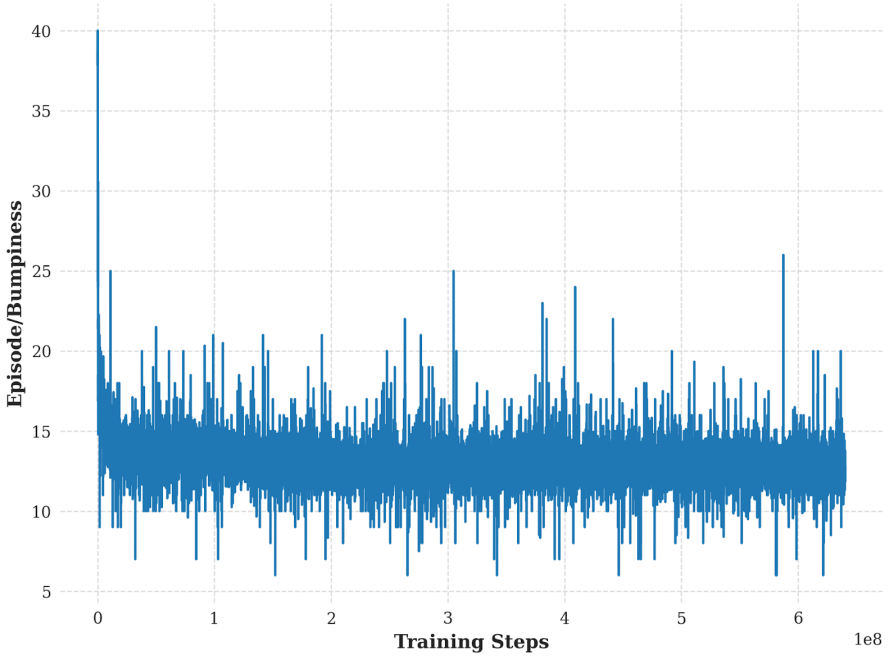


Figure 32 - End of episode bumpiness

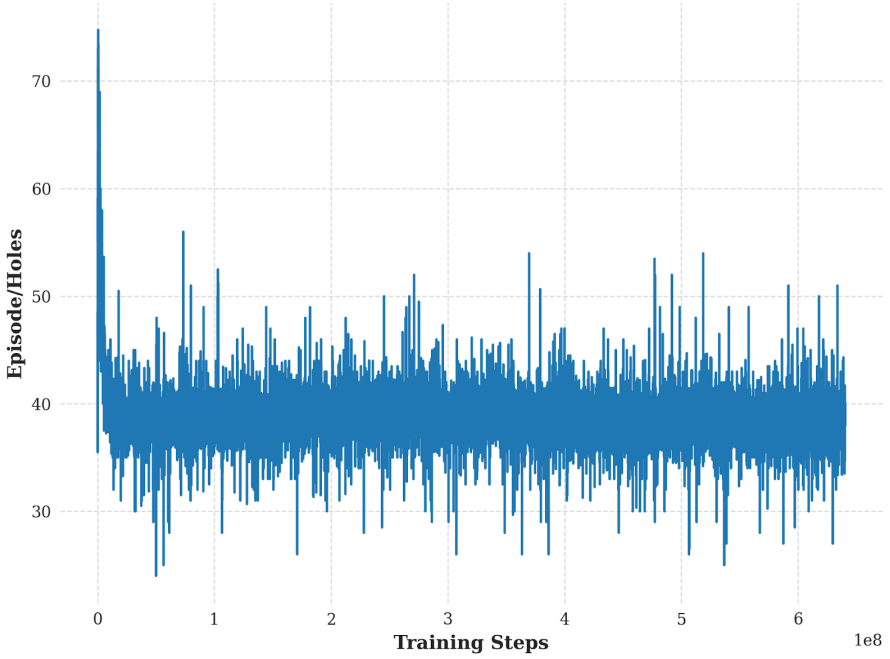


Figure 33 - End of episode holes

The mean reward curve (Figure 34) presents an interesting contrast to the other metrics. Despite showing overall improvement, it exhibits more pronounced volatility than metrics like piece count or lines cleared. This volatility likely reflects the complex relationship between immediate rewards and long-term strategic play in Tetris, where actions that temporarily reduce reward might be necessary for better board management.

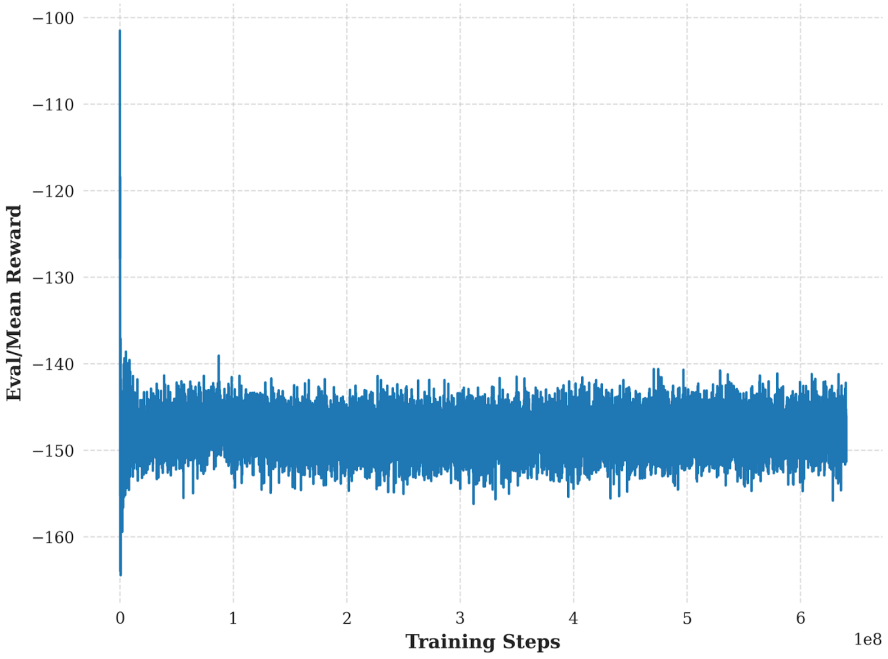


Figure 34 - Mean Reward

Episode length (Figure 35) correlates strongly with the pieces placed metric, showing similar improvement patterns. The final training loss curve (Figure 36) indicates stable learning, with initial high variance settling into a more consistent pattern as training progressed. This stabilization suggests successful convergence of the Q-learning process, though the persistent fluctuations align with the known challenges of applying deep reinforcement learning to Tetris (Algorta & Şimşek, 2019).

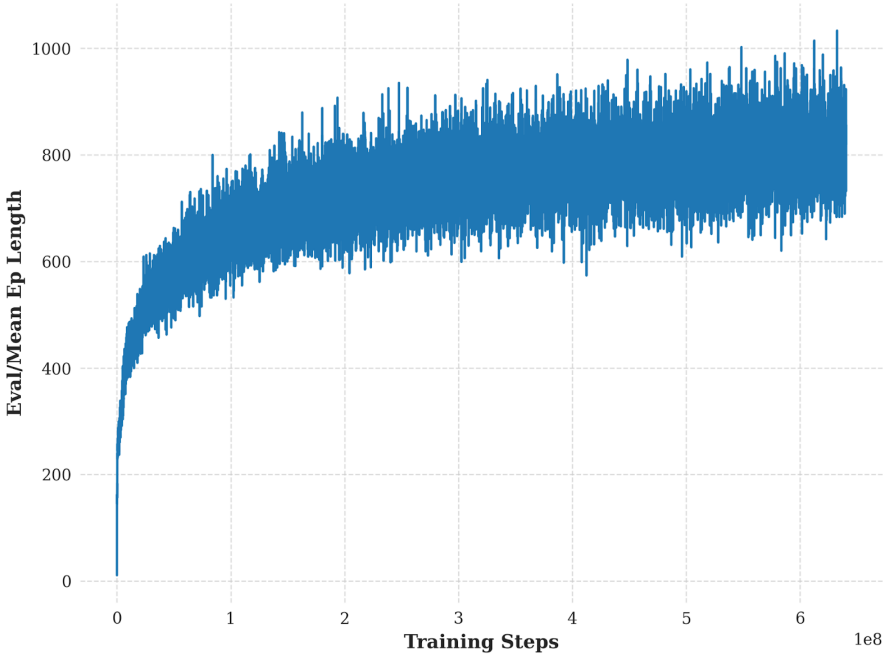


Figure 35 - Episode Length

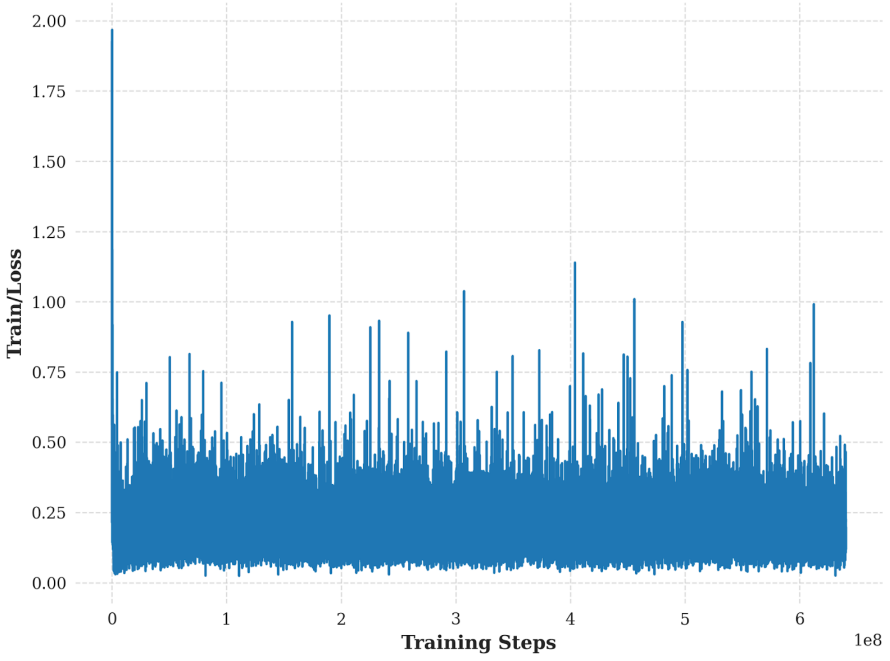


Figure 36 - Train loss

These results demonstrate that while the agent successfully learned effective strategies for Tetris gameplay, the learning process remains characterized by significant variability - a common challenge in deep reinforcement learning applications to complex games.

5.4 Conclusions

The evaluation demonstrates that our Deep Q-Learning agent successfully achieved its primary objective of outperforming random gameplay across all key metrics. Analysis of 500 evaluation episodes showed the agent consistently surpassing random play with an average score of 677.31 (±195.40) versus 93.21 (±16.37), while dramatically extending survival time from 44.97 to 796.86 timesteps. This 1672% improvement in episode duration clearly demonstrates the agent's acquisition of fundamental board management strategies.

The analysis of board state metrics revealed valuable insights, though our collection methodology could be enhanced. Currently, metrics like holes and bumpiness are only recorded at episode termination; tracking these throughout gameplay would provide richer data about strategy development. Similarly, the apparent contradiction between improved gameplay and decreased cumulative reward suggests opportunities for reward function refinement.

While our implementation successfully demonstrated deep reinforcement learning's potential for Tetris, there remains substantial room for improvement in both performance measurement and reward system design. These findings align with previous research suggesting that Tetris continues to present unique challenges for learning-based approaches (Algorta & Şimşek, 2019), while highlighting specific areas for future enhancement.

6 Conclusions

6.1 Introduction

Our implementation of a deep reinforcement learning agent for Tetris has provided valuable insights into both the capabilities and limitations of modern AI techniques when applied to classic games. While our DQN-based approach demonstrated significant improvements over random play, achieving consistent high-level performance proved challenging. This chapter examines our results in detail, discussing their implications for both Tetris AI development and the broader field of deep reinforcement learning, while identifying promising directions for future research.

6.2 Discussion

Our journey in developing a deep reinforcement learning agent for Tetris has revealed both the power and limitations of current AI approaches to classic games. Through careful consideration of the requirements established in Chapter 2, followed by systematic design and implementation, we have created a system that demonstrates meaningful learning while highlighting several interesting challenges in applying deep reinforcement learning to Tetris.

The implementation successfully met many of our core requirements. Our agent achieved consistent improvement over random play across all key metrics, with particularly strong results in survival time - extending gameplay duration by over 1600% compared to random actions. The system's ability to clear lines, averaging 15.89 lines per game, demonstrates that it learned fundamental Tetris strategies, though this falls short of expert human performance. This aligns with Algorta and Şimşek's (2019) observation that Tetris remains a challenging domain for AI, despite its seemingly simple rules.

Our architectural decisions proved largely successful. The choice to build upon Stable-Baselines3 for the DQN implementation, following Raffin et al.'s (2021) framework, provided a robust foundation for experimentation. The modular design, separating core game mechanics from learning components, enabled rapid iteration during development. The decision to use Gymnasium for environment standardization, as advocated by Towers et al. (2024), facilitated easy integration with existing reinforcement learning tools while maintaining clean separation between components.

The state representation and reward structure choices revealed interesting insights. Our initial attempts at developing a custom reward function were ultimately superseded by the fitness-based reward structure demonstrated by Stevens and Pradhan (2016). Their approach, which considers multiple board state characteristics including holes, bumpiness, and aggregate height, proved more effective in practice. Our image-based state representation, inspired by Mnih et al.'s (2013) work on Atari games, proved effective at capturing essential game state information. However, the reward structure presented several fundamental challenges that influenced our design decisions.

During our initial experiments with a simplified neural network architecture - consisting of a single convolutional layer followed by two fully connected layers of 128 nodes each - we observed an intriguing emergent strategy. The agent developed a distinct pattern of piece placement, systematically arranging O pieces on the left side of the board followed by vertical I pieces, creating a central column of T pieces, and alternating L and J pieces on the

right side. This spatial organization proved moderately successful, though ultimately unsustainable when execution errors accumulated. This emergent behavior raises important considerations about AI systems' capacity to develop unexpected strategies through pure reward optimization. While demonstrating impressive creative problem-solving capabilities, such emergent behaviors also highlight challenges in AI transparency and explainability. The agent's development of this strategy, driven solely by reward optimization without clear reasoning, emphasizes the importance of careful monitoring and evaluation of AI systems, particularly when similar approaches might be applied to real-world domains where unintended behaviors could have significant consequences.

Resource constraints significantly shaped our development approach, particularly regarding GPU access limitations due to budget constraints. This led to the development of a sophisticated experiment management system that could efficiently utilize available computational resources. The system was specifically designed to run multiple training processes concurrently, optimizing GPU utilization by running parallel experiments since individual training instances didn't fully utilize the GPU capacity. This approach enabled us to maintain continuous progress despite limited resources, though it influenced several key implementation decisions. The time and computational constraints nudged us toward adopting Stevens and Pradhan's (2016) proven reward scheme rather than extensively developing our own, and similarly limited our ability to experiment with alternative network architectures. Our infrastructure enabled systematic exploration of hyperparameters while maintaining experimental reproducibility, allowing us to efficiently identify promising configurations for further optimization despite these constraints.

Our evaluation methodology provided clear insights into the agent's capabilities while highlighting areas for improvement. Initial attempts at parallelization through SubprocVecEnv showed that the overhead of inter-process communication outweighed potential benefits for our computationally lightweight Tetris environment. This led to the adoption of DummyVecEnv, which proved more efficient for environments where state processing is relatively simple. The comprehensive metrics collection system proved invaluable for understanding the agent's behavior, though real-time analysis capabilities could be enhanced.

These experiences suggest that while deep reinforcement learning can successfully learn Tetris strategies, significant challenges remain in achieving human-level performance. The gap between our agent's performance and expert human play indicates that there may be fundamental limitations in our current approach to modeling strategic thinking in reinforcement learning systems.

6.3 Future Work

This research has laid a foundation for deep reinforcement learning in Tetris, but there remain several promising avenues for future investigation and improvement. Our plans for future work span multiple areas, from algorithmic improvements to technical optimizations.

A primary focus will be on systematic hyperparameter optimization using the Optuna framework. While our current implementation achieves reasonable results, we believe significant performance gains are possible through careful tuning of learning rates, network architectures, exploration strategies, and buffer parameters. This optimization process would

include comprehensive ablation studies to better understand parameter sensitivity and interactions.

Beyond our current DQN implementation, we plan to explore alternative algorithms available in the Stable Baselines 3 framework. Modern policy gradient methods such as Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC), and Advantage Actor-Critic (A2C) have shown promising results in other domains and may be particularly well-suited to the Tetris environment. We are especially interested in algorithms designed for sparse reward scenarios, as this remains a challenging aspect of the Tetris learning problem.

The state representation of our environment could be enhanced through several approaches. While we currently use raw board states, implementing frame stacking (Mnih et al, 2015) could better capture the temporal dynamics of the game. We also plan to evaluate alternative state representations such as height maps, transition maps, and explicit hole detection features. These representations might help the agent learn more efficient strategies by making critical game state information more directly accessible.

Imitation learning presents another promising direction. By recording and learning from expert human gameplay, we could implement behaviour cloning as a pre-training step. This could be combined with reinforcement learning in a hybrid approach, potentially using techniques like Generative Adversarial Imitation Learning (GAIL) to better capture expert strategies while maintaining the ability to discover novel approaches.

Our Tetris implementation itself could be enhanced to more closely match modern Tetris games. Implementing the Super Rotation System (SRS), hold piece functionality, and next piece preview would bring our environment closer to standard Tetris implementations. Additionally, we plan to implement advanced mechanics such as wall kicks and T-spins, which could enable more sophisticated playing strategies.

To better evaluate agent performance, we intend to implement additional metrics and heuristics. These would include well depth and count, column and row transitions, and more sophisticated hole metrics including depth and connectivity. These metrics could be combined into composite scoring functions that better capture the nuances of strong Tetris play.

On the technical side, we see several opportunities for optimization. A key improvement would be implementing prioritised experience replay, where transitions with higher temporal-difference errors are sampled more frequently. This approach, proven effective in the DQN architecture, could lead to more efficient learning by focusing on the most informative experiences. The current replay buffer implementation could also be optimised in terms of memory usage and sampling efficiency. Additionally, we could improve GPU utilisation through careful tuning of batch sizes and model parallelism. Profiling the training loop could reveal additional bottlenecks that could be optimised, particularly in the interaction between experience collection and network updates.

Finally, we plan to experiment with modern neural network architectures. This includes investigating attention mechanisms (DRQN) and transformer-based architectures that might be better suited to capturing the sequential nature of Tetris gameplay. We're also interested in exploring multi-task learning approaches that could simultaneously optimise for different game objectives.

The immediate priority will be hyperparameter optimization and exploration of alternative algorithms, as these are likely to yield the most significant improvements in agent performance. Environment improvements, while valuable for closer alignment with real Tetris implementations, will be treated as secondary objectives.

These proposed extensions maintain our focus on reinforcement learning while incorporating modern deep learning advances and established game-playing techniques. Each enhancement builds naturally on our existing implementation and is grounded in current literature, providing a clear path forward for improving our understanding of both Tetris-specific challenges and broader reinforcement learning applications.

6.4 Conclusions

The development of a deep reinforcement learning agent for Tetris has demonstrated both the potential and current limitations of applying modern AI techniques to classic games. Our DQN-based implementation successfully learned fundamental Tetris strategies, showing significant improvements over random play across all key metrics, while revealing interesting challenges in achieving consistent high-level performance.

The modular architecture developed through this project provides a robust foundation for future research, while our comprehensive evaluation framework offers valuable insights into agent behavior and learning dynamics. However, the gap between our agent's performance and expert human play highlights substantial opportunities for improvement.

Several promising directions for future work emerged from this research, including the exploration of more sophisticated reward mechanisms, the potential integration of imitation learning, and the investigation of modern neural network architectures. While our implementation has advanced our understanding of applying deep reinforcement learning to Tetris, it also highlights the exciting challenges that remain in developing AI systems capable of mastering this deceptively complex game.

7 References

Gym, O. and Sanghi, N., 2021. Deep reinforcement learning with python. Springer: Berlin/Heidelberg, Germany.

Stevens, M. and Pradhan, S. (2016). Playing Tetris with Deep Reinforcement Learning. Course project report for CS231n: Convolutional Neural Networks for Visual Recognition, Stanford University

Brzustowski, J., 1992. Can you win at Tetris? (Masters' Thesis, University of British Columbia).

Demaine, E. D., Hohenberger, S., & Liben-Nowell, D. (2003). Tetris is hard, even to approximate. In Proc. 9th International Computing and Combinatorics Conference (COCOON 2003) (pp. 351–363). Berlin: Springer.

Carr, D., 2005. Applying reinforcement learning to Tetris. Department of Computer Science Rhodes University.

Algorta, S. and Şimşek, Ö., 2019. The game of tetris in machine learning. *arXiv preprint arXiv:1905.01652*.

Temple, M. 2004. Tetris: From Russia with love. [TV Documentary] London: BBC.

Chen, Z., 2021. Playing Tetris with deep reinforcement learning (Master's thesis,University of Illinois at Urbana-Champaign).

Plank-Blasko, D., 2015. 'From russia with Fun!': Tetris, Korobeiniki and the ludic soviet. The Soundtrack, 8(1-2), pp.7-24.

Sutton, R.S. and Barto, A.G., 2018. Reinforcement learning: An introduction. A Bradford Book.

Mnih, V., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M. and Dormann, N., 2021. Stable-baselines3: Reliable reinforcement learning implementations. _Journal of Machine Learning Research_, _22_(268), pp.1-8.

Towers, M., Kwiatkowski, A., Terry, J., Balis, J.U., De Cola, G., Deleu, T., Goulao, M., Kallinteris, A., Krimmel, M., KG, A. and Perez-Vicente, R., 2024. Gymnasium: A standard interface for reinforcement learning environments. _arXiv preprint arXiv:2407.17032_.

Pang, B., Nijkamp, E. and Wu, Y.N., 2020. Deep learning with tensorflow: A review. _Journal of Educational and Behavioral Statistics_, _45_(2), pp.227-248.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L. and Desmaison, A., 2019. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32.

Raschka, S., Patterson, J. and Nolet, C., 2020. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. Information, 11(4), p.193.

Watkins, C.J.C.H., 1989. Learning from delayed rewards.

Watkins, C.J. and Dayan, P., 1992. Q-learning. Machine learning, 8, pp.279-292.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. nature, 518(7540), pp.529-533.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M.G. and Pineau, J., 2018. An introduction to deep reinforcement learning. Foundations and Trends® in Machine Learning, 11(3-4), pp.219-354.

Wu, J., 2017. Introduction to convolutional neural networks. National Key Lab for Novel Software Technology. Nanjing University. China, 5(23), p.495.

Lundgaard, N. and McKee, B., 2006. Reinforcement learning and neural networks for tetris. Technical report, Technical Report, University of Oklahoma.

Fahey, C., 2003. Tetris. http://www.colinfahey.com/tetris/tetris.html. [Online]

8 Appendix

More recent code can be found at https://github.com/atopuzov/tud-drl-project