s

PHENIKAA UNIVERSITY

FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING

FINAL PROJECT

Topic : Tetris Game Solving using Deep Reinforcement Learning

Student implementation:

Anh Tuan Nguyen ID: 22014593

Class: K16 AI&RB

Instructors: ThS. Hoang Dieu Vu

Faculty of Electrical and Electronics Engineering

Ha Noi, Semester III, Academic year 2024-2025

TABLE OF CONTENTS

ABSTRACT 4

CHAPTER 1: INTRODUCTION 5

CHAPTER 2: ENVIRONMENT 6

2.1 State 6

2.2 Actions 6

2.3 Reward 7

2.4 User Interface (UI) 7

CHAPTER 3: DEEP REINFORCEMENT LEARNING ALGORITHMS 9

3.1 DQN Algorithm 9

3.2 Double DQN Algorithm 10

3.3 Dueling DQN Algorithm 11

3.4 Advantage Actor–Critic Algorithm 12

CHAPTER 4: EVALUATING 14

4.1 DQN 14

4.2 Double DQN 16

4.3 Dueling DQN 18

4.4 Advantage Actor–Critic (A2C) 20

CHAPTER 5: CONCLUSION 22

REFERENCES 23

TABLE

Table 1: Actions for Tetris environment 7

Table 2: Reward structure for Tetris environment. 7

TABLE OF FIGURE

Figure 1: UI for Tetris Game 8

Figure 2: Reward for DQN with learning rate LR=0.0001 14

Figure 3: Reward for DQN with learning rate LR=0.0005 15

Figure 4: Reward for DQN with learning rate LR=0.0001 and 0.005....................15

Figure 5: Reward for Double DQN with learning rate LR=0.0005....................16

Figure 6: Reward for Double DQN with learning rate LR=0.0001....................17

Figure 7: Reward for Double DQN with learning rate LR=0.0001 and 0.005....................17

Figure 8: Reward for Dueling DQN with learning rate LR=0.0001 18

Figure 9: Reward for Dueling DQN with learning rate LR=0.0005 19

Figure 10: Reward for Dueling DQN with learning rate LR=0.0001 and 0.005....................19

Figure 11: Reward for A2C with learning rate LR=0.0001 20

Figure 12: Reward for A2C with learning rate LR=0.0005 20

Figure 13: Reward for A2C with learning rate LR=0.0001 and 0.005....................21

ABSTRACT

Tetris, a widely recognized puzzle game, poses a significant challenge for Deep Reinforcement Learning (DRL) due to its combinatorial complexity, high-dimensional state space, and the necessity for long-term strategic planning [1]. Despite its seemingly simple mechanics, the game requires an agent to effectively manage resources, optimize spatial arrangements, and dynamically adapt strategies to prolong gameplay and maximize the cumulative score [2]. DRL algorithms enable artificial agents to autonomously learn and improve strategies by interacting with the environment and optimizing reward signals over time [3]. In this study, we specifically examine the application of two advanced variants of Deep Q-Network (DQN): Double DQN [4] and Dueling DQN [5], to address the complexities of the Tetris game. We constructed a custom Tetris environment tailored for evaluating the performance of these algorithms under realistic and varying game scenarios, including randomized piece generation and variable speeds. The evaluation involved measuring cumulative rewards per episode and assessing convergence behavior. Experimental results demonstrate that Double DQN effectively mitigates the overestimation bias inherent in traditional DQN, leading to more stable learning and policy improvements [6], while Dueling DQN shows superior performance by separately estimating state-values and advantage functions, enhancing decision-making quality, particularly in nuanced game situations [7]. The outcomes indicate both algorithms' robustness and effectiveness, with Dueling DQN exhibiting particular strengths in state-value estimation and rapid convergence towards optimal strategies. The findings underscore the suitability of advanced DQN variants for solving Tetris and suggest promising directions for further research, including integration with other DRL techniques such as Prioritized Experience Replay and Rainbow architectures [8].

CHAPTER 1: INTRODUCTION

Tetris, a widely recognized puzzle game, has captivated players with its intuitive yet challenging gameplay for decades. Players must strategically position falling geometric shapes (Tetrominoes) to complete and clear horizontal lines without gaps. Despite its seemingly straightforward mechanics, Tetris presents a complex environment that demands real-time decision-making, spatial planning, and adaptability to evolving circumstances, making it an ideal testbed for exploring Deep Reinforcement Learning (DRL) algorithms [1].

Deep Reinforcement Learning, a subfield of machine learning, involves training intelligent agents to make sequential decisions to maximize cumulative rewards through continuous interaction with the environment, leveraging deep neural networks for function approximation [2]. Within the context of Tetris, DRL offers promising avenues for developing autonomous agents capable of discovering optimal strategies to manage limited space, anticipate future placements, and handle the stochastic nature of Tetromino sequences [3]. The combinatorial complexity and the high-dimensional, dynamic state space of Tetris introduce significant challenges for DRL methodologies, emphasizing the need for effective exploration-exploitation trade-offs and accurate state-value estimations [4].

This study aims to evaluate the effectiveness of two advanced variants of the Deep Q-Network (DQN)—Double DQN and Dueling DQN—in addressing the challenges posed by the Tetris game [5,6]. These algorithms were selected due to their proven ability to mitigate common issues encountered in standard DQN, such as value overestimation and insufficient differentiation between states and actions [7]. Double DQN separates action selection and evaluation to reduce bias [8], while Dueling DQN independently assesses state values and advantages, leading to more precise decision-making in nuanced scenarios [9].

This paper is structured as follows. Section 2 introduces our customized Tetris environment, detailing the definitions of states, actions, and rewards. Section 3 presents an overview of the Double DQN and Dueling DQN algorithms utilized in this research. Section 4 discusses experimental results, analyzing each algorithm's performance, highlighting their strengths and identifying limitations. Finally, Section 5 summarizes the study's insights and outlines future research directions aimed at enhancing DRL approaches for complex, dynamic gaming environments.

CHAPTER 2: ENVIRONMENT

2.1 State

In our Tetris DRL agent, the state at each time step is represented as a stack of four consecutive 84×84 grayscale frames. Each frame is obtained by rendering the current board (10×20 cells) to an off-screen 300×600 pixel surface, converting to a single-channel image, and resizing to 84×84 via area interpolation. Stacking four frames captures both the static layout of locked Tetrominoes and the temporal motion of the falling piece, enabling the CNN to infer piece velocity and direction without an explicit velocity input.

Formally, the observation $s_t \in R^{4\times 84\times 84}$ is constructed by

- **Rendering** the board and active piece to a 300×600 RGB surface.
- **Grayscaling** via luminance conversion to a single channel.
- **Resizing** to 84×84.
- **Frame stacking**: appending the latest frame to the preceding three, dropping the oldest.

This representation balances informativeness and compactness:

- **Spatio-temporal features**: The CNN can detect both the current spatial configuration and recent piece movement.
- **Dimensionality reduction**: Downsampling to 84×84 keeps the input size manageable for real-time DRL training.
- **Stationarity**: By normalizing pixel values to [0,1][0,1][0,1], we maintain consistent input distributions that aid stable network convergence.

Using this 4×84×84 tensor, our Q-network's convolutional layers extract hierarchical features—such as column heights, holes, and piece orientations—upon which the fully connected layers build action-value estimates. This design proved effective for both Double DQN and Dueling DQN, enabling the agent to learn high-quality placement strategies directly from raw visual inputs.

2.2 Actions

In our Tetris environment, the agent's decision at each time step is selected from a discrete set of six possible actions that fully control the falling Tetromino. These actions are encoded as integers 0 through 5 and correspond to the following controls:

'0'	Move left
'1'	Move right
'2'	Rotate
'3'	Soft drop
'4'	Hard drop

‘5’	Hold

Table 1: Actions for Tetris environment

Together, these six actions give the agent full control over piece positioning, orientation, and timing, enabling both precision placements and strategic adjustments to optimize line clears and minimize board irregularities.

2.3 Reward

The reward function in our Tetris environment is carefully shaped to guide the agent toward efficient line clears, a smooth board surface, and prolonged survival. At each piece lock-in, the agent receives:

EVENT	REWARD
Clear n line (n = 1, 2, 3,…)	LINE_SCORE[n] (100, 300, 500 or 800)
Piece “lock” (each placement)	+ SURVIVAL_BONUS (currently +1)
Light hole (above mid-line)	– ALPHA × (# light holes) (ALPHA = 0.2)
Dead hole (below mid-line)	– BETA × (# dead holes) (BETA = 1.0)
Game over	– GAME_OVER_PENALTY (currently –10)

Table 2: Reward structure for Tetris environment.

Our reward structure is designed to encourage high line-clear scores while also penalizing poor board states and discouraging premature game termination. Each time the agent locks a piece, it receives the standard Tetris line-clear points—100 for a single line, 300 for a double, 500 for a triple, and 800 for a “Tetris”—plus a small survival bonus of +1 to reward continued play. To prevent the agent from creating unfillable gaps, we impose graded hole penalties: each “light hole” above the board’s midline carries a penalty of 0.2 ($\alpha = 0.2$), and each “dead hole” below the midline carries a heavier penalty of 1.0 ($\beta = 1.0$). These weights reflect the intuition that holes deeper in the well are far harder to recover, so $\beta \gg \alpha$ steers the agent to avoid deep holes especially. Finally, if the board fills and the episode ends, the agent incurs a −10 game-over penalty. By combining line-clear bonuses, a survival incentive, and asymmetric hole penalties controlled by α and β, plus a game-over deterrent, we provide dense, informative feedback that pushes the agent toward both high scores and robust, clean play.

2.4 User Interface (UI)

Figure 1 illustrates the Pygame-based UI we developed for our Tetris environment. The main playfield is rendered at a resolution of 10 columns by 20 rows, with each cell drawn as a 30 × 30-pixel square. Gray blocks represent the settled Tetrominoes on the board, while the currently falling piece is highlighted in green. As the piece moves or rotates, the display is updated in real time at a fixed frame rate, allowing both human observers and the DRL agent to "see" the exact state used for decision making.

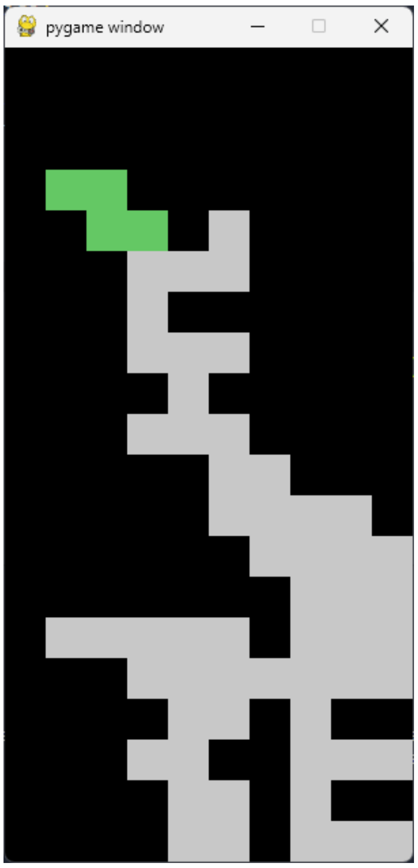

Figure 1: UI for Tetris Game

This simple yet effective visualization supports debugging, performance monitoring, and qualitative assessment of the agent's learned behavior, as one can observe how quickly and precisely it responds to changing board configurations.

CHAPTER 3: DEEP REINFORCEMENT LEARNING ALGORITHMS

In this project, we implement and evaluate four deep-reinforcement-learning algorithms—Vanilla DQN, Double DQN, Dueling DQN, and Advantage Actor-Critic (A2C)—on a custom Tetris environment. Vanilla DQN serves as our baseline, learning action-value estimates via an ε-greedy policy but often suffering from overestimation and instability. Double DQN mitigates this bias by decoupling action selection from evaluation, while Dueling DQN further decomposes the Q-function into separate value and advantage streams to accelerate convergence and improve state-value estimation. Finally, A2C adopts an actor-critic architecture, jointly learning a policy (actor) and state-value function (critic) to reduce variance and enhance sample efficiency. We train all four under the same regime and compare their stability, convergence speed, and overall gameplay performance.

3.1 DQN Algorithm

Algorithm 1: Deep Q-Learning with Experience Replay

To train our DQN agents, we employ the classic Deep Q-Learning algorithm augmented by experience replay and a target network. The procedure unfolds as follows:

3.1.1 Initialization

- We first create a replay memory **D** of fixed capacity *N* to store past experience tuples.
- Two neural networks are defined: the **online Q-network** $Q(s, a; \theta)$ whose parameters θ are learned via gradient descent, and the **target Q-network** $\hat{Q}(s, a; \theta^{-})$, whose parameters θ^{-} are hard-updated from θ every *C* training steps to stabilize the bootstrapping targets.

3.1.2 Episode Loop

For each episode i=1i=1i=1 to MMM:

a. **Reset the environment** and obtain the initial raw observation x_1

b. **Pre-process** it into a stacked state representation $\phi_1 = \phi(x_1)$ (e.g. four grayscaled, resized frames).

3.1.3 Time-step Loop

For each time step $t = 1$ to T until the episode terminates:

a. **Action selection (ε-greedy):**

- With probability ε, choose a random action a_t to encourage exploration.
- Otherwise, select the action maximizing the online network's estimate:

$$a_t = \arg\max_a Q\,(\phi(s_t), a; \theta)$$

b. **Environment interaction:**

- Execute a_t in the Tetris environment, observe the immediate reward r_t and the next raw frame $x_{(t+1)}$
- Form the next state $\phi_{t+1} = \phi(x_{t+1})$ and determine whether the episode has terminated.

c. **Store transition:**

- Append the tuple $(\phi(s_t), a_t, r_t, \phi_{t+1})$ to the replay buffer **D**, possibly discarding the oldest entry when full.

d. **Experience replay & network update:**

- Sample a random minibatch of B transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from **D**
- For each sampled transition, compute the **TD-target**

$$y_j = \begin{cases} r_j, & if the\ next\ state\ is\ terminal \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right), & otherwise \end{cases}$$

- Perform a gradient descent step on the mean-squared error loss

$$L(\theta) = \frac{1}{B} \sum_j \left(y_j - Q(\phi_j, a_j; \theta)\right)^2$$

with respect to the online network parameters θ

- Every C steps, reset the target network by copying $\theta^- \leftarrow \theta$, ensuring that the bootstrap targets remain stable over several updates.

3.1.4 Repeat

- Continue until all M episodes are completed, allowing the online network to approximate the optimal action-value function $Q^*(s, a)$.

This combination of experience replay (to decorrelate training samples), ε-greedy exploration (to balance exploration and exploitation), and a slowly updated target network (to stabilize targets) forms the foundation upon which both our **Double DQN** and **Dueling DQN** extensions build, yielding more robust and accurate value estimates in the high-dimensional, strategic domain of Tetris.

3.2 Double DQN Algorithm

Double Deep Q-Network (Double DQN) Algorithm

The Double DQN algorithm enhances the stability and performance of the original DQN by mitigating the overestimation bias commonly present in Q-learning updates. Instead of using the same network to both select and evaluate the next action, Double DQN decouples these roles using two networks: the **online network $\boldsymbol{Q(s, a; \theta)}$** and the **target network $\boldsymbol{Q(s, a; \theta^-)}$**

At each training episode, transitions are collected by interacting with the environment using an ε-greedy policy. These transitions are stored in a replay buffer $\mathcal{D}$ of fixed size. Once enough experiences are accumulated, the agent samples a minibatch from the buffer to update the Q-values.

The key distinction of Double DQN lies in how the target value y_j is computed. Instead of directly applying $max_a' Q(s', a')$ from the same network, the algorithm selects the best action using the **online network** but evaluates its value using the **target network**:

$$a^{\max} = \arg\max_{a'} Q\ (s', a'; \theta)$$

$$y_j = \begin{cases} r_j, & if\ s'\ is\ terminal \\ r_j + \gamma Q(s', a^{\max}; \theta^-), & otherwise \end{cases}$$

The loss is defined as the squared difference between the target y_j and the predicted Q-value:

$$L(\theta) = |y_j - Q(s, a; \theta)|^2$$

A gradient descent step is then applied to minimize this loss with respect to the parameters θ. Periodically, the target network parameters θ^- are updated to match the online network θ, ensuring stability during learning.

This mechanism effectively reduces over-optimistic value estimates, resulting in more robust convergence and improved performance across various reinforcement learning tasks.

3.3 Dueling DQN Algorithm

Dueling Deep Q-Network (Dueling DQN)

Dueling DQN is an enhancement of the traditional Deep Q-Network that introduces a novel architecture by decoupling the estimation of the **state-value function** $V(s)$ and the **advantage function$\boldsymbol{A(s, a)}$**. These two components are combined to compute the Q-value using the following equation:

$$Q(s, a; \theta) = V(s; \theta_v) + \left(A(s, a; \theta_a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta_a) \right)$$

In this architecture:

- $\theta = (\theta_v, \theta_a)$ represents the shared parameters of the value and advantage streams.
- $V(s)$ estimates how good it is to be in a state.
- $A(s, a)$ captures the importance of each action in that state.

The learning procedure is similar to that of Double DQN. Two networks are maintained:

- An **online network $\boldsymbol{Q_{(policy)}}$** which is updated every step,
- And a **target network $\boldsymbol{Q_{(target)}}$** which is periodically synchronized from the online network to stabilize training.

The agent selects actions using an ε-greedy exploration strategy. For each transition $(s_t, a_t, r_t, s_{(t+1)})$, the TD target is computed as:

$$y_j = \begin{cases} r_j\,, & if\ s_{(t+1)}\ is\ terminal \\ r_j + \gamma \cdot Q_{\text{target}}\left(s_{t+1}, \arg\max_{a'} Q_{\text{policy}}\,(s_{t+1}, a'; \theta); \theta^-\right), & otherwise \end{cases}$$

The loss is then computed by minimizing the Mean Squared Error between the predicted Q-value and the TD target:

$$L(\theta) = \left(y_j - Q_{\text{policy}}(s_j, a_j; \theta)\right)^2$$

Over time, the target network is updated according to:

$$\theta^- \leftarrow \theta$$

This architecture allows the network to learn which states are valuable, regardless of the specific actions, which is particularly beneficial in environments like Tetris where many actions may result in similar outcomes. The Dueling DQN setup implemented in our project aligns closely with this theoretical framework, confirming its suitability and effectiveness for the task.

3.4 Advantage Actor–Critic Algorithm

The Advantage Actor-Critic (A2C) algorithm combines policy-based and value-based methods by learning both a parameterized policy $\pi_{(\theta)}(a \mid s)$ (the "actor") and a value function $V_{(\theta)}(s)$ (the "critic") with shared parameters θ. At each timestep the actor selects actions according to an ε-greedy version of its policy, while the critic evaluates those actions by estimating the state-value. Gradients from both objectives are accumulated and applied asynchronously to improve data efficiency and stability.

3.4.1 Initialize

- Shared network parameters θ (for both actor and critic), and a target copy $\theta^- \leftarrow \theta$
- Global step counter $T \leftarrow 0$, per-thread step counter $t \leftarrow 0$

3.4.2 Get initial state s_0 by resetting the environment**.**

3.4.3 Repeat until T exceeds a maximum total step count T_{max}:

- **Action selection**
  - Choose action a_t from an ε-greedy policy based on the actor's Q-estimates:

$$a_t \sim \begin{cases} random\ action, & with\ probability\ \epsilon \\ \arg\max_a Q\ (s_t, a; \theta), & \text{otherwise.} \end{cases}$$

- **Environment step**
  - Execute a_t, observe reward r_t and next state s_{t+1}
- **Compute TD-target**

$$\boldsymbol{y_t} = \begin{cases} \boldsymbol{r_t}, & \boldsymbol{if s_{(t+1)} is terminal}, \\ r_t + \gamma\ \max_{a'} Q\ (s_{t+1}, a'; \theta^-), & otherwise \end{cases}$$

- **Accumulate gradients**
  - **Critic loss:** compute $L_c = \left(y_t - Q(s_t, a_t; \theta)\right)^2$
  - **Actor (policy) loss:** compute the advantage $A_t = y_t - V(s_t; \theta)$ then $L_p = -\log \pi_\theta\ (a_t \mid s_t)\ A_t$
  - Accumulate ∇θ L_p and ∇θ L_c into a shared gradient buffer dθ
- **Periodic parameter updates**
  - Increment counters: $T \leftarrow T + 1$, $t \leftarrow t + 1$
  - If T mod T_target = 0, synchronize target network $\theta^- \leftarrow \theta$
  - If t mod I_asyncUpdate = 0 **or** s_{t+1} is terminal:
    - Apply the accumulated gradient dθ to update θ via (asynchronous) gradient descent.
    - Clear the gradient buffer $d\theta \leftarrow 0$, and reset $t \leftarrow 0$

3.4.4 End Repeat

This asynchronous update scheme allows multiple workers (threads or environments) to collect experiences in parallel, aggregate gradients, and update a shared policy–value network, leading to faster, more stable learning compared to purely synchronous methods.

CHAPTER 4: EVALUATING

4.1 DQN

In this project, we trained the Deep Q-Network (DQN) on the custom Tetris environment for approximately 200000 frames, experimenting with two different learning rates: 0.0001 and 0.0005. The reward per episode was tracked to assess the model's learning behavior and stability.

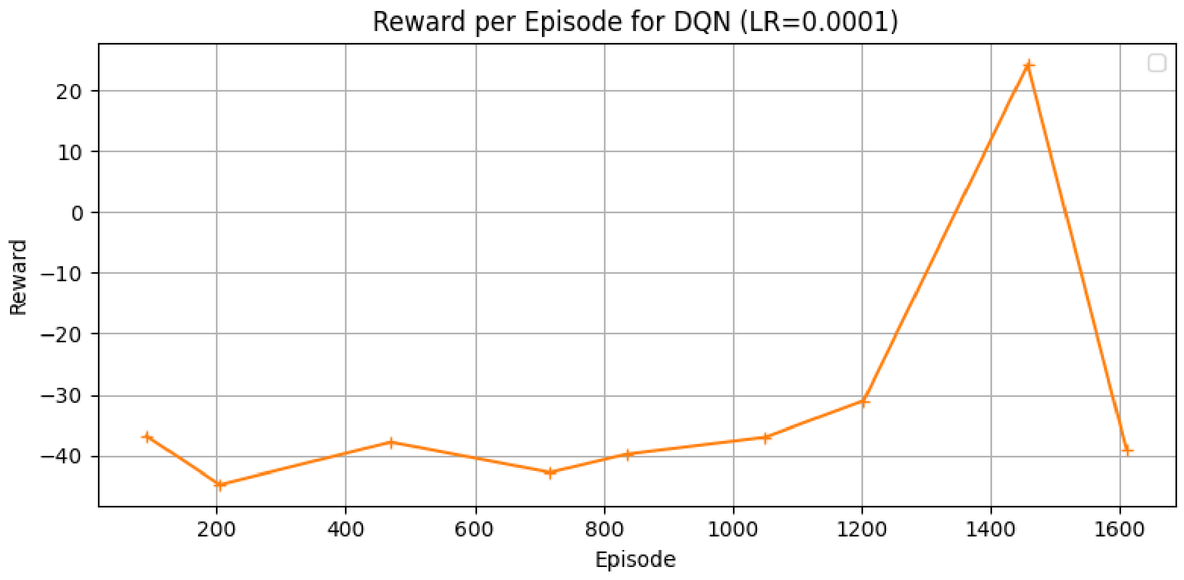

Figure 2: Reward for DQN with learning rate LR=0.0001

At LR = 0.0001, DQN's rewards rise slowly from about –45 to –30 over 1 600 episodes, reflecting more consistent line clears and fewer hole penalties. A single spike to +25 around episode 1 450 shows the agent can occasionally discover high-value sequences, but the gains aren't maintained—rewards fall back to –40 by episode 1 600. This suggests the small learning rate yields stable but slow improvements.

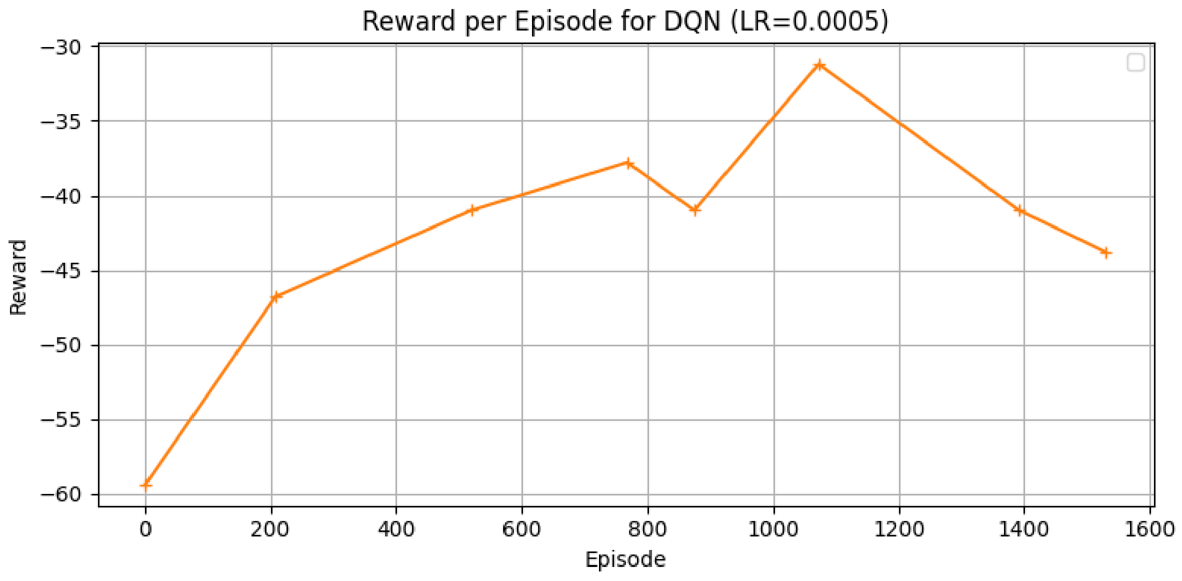

Figure 3: Reward for DQN with learning rate LR=0.0005

With LR = 0.0005, the loss curve drops sharply from ~40 down to the 2–5 range within the first 200 episodes, indicating rapid initial learning. After that, loss fluctuates periodically but remains relatively low (around 2–6), suggesting stable value updates despite ongoing exploration. Those recurring spikes align with episodes where large reward changes occur (e.g. occasional line clears), but the overall low plateau shows the network has largely converged its Q-estimates. In combination with the reward plot, this tells us that while the agent quickly fits its value function, the higher learning rate still induces some instability in policy quality.

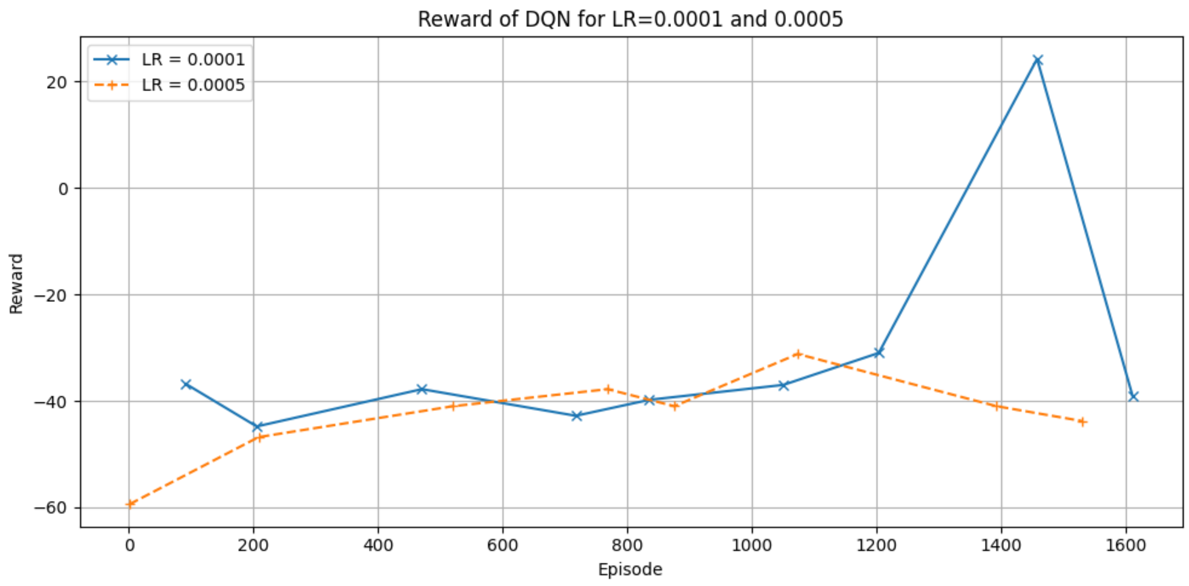

Figure 4: Reward for DQN with learning rate LR=0.0001 and 0.005

Comparing the two curves, we see that the smaller learning rate (LR = 0.0001, blue) yields more pronounced improvements later in training—most notably the jump above zero around episode 1500—

whereas the higher rate (LR = 0.0005, orange) produces a smoother but shallower upward trend, peaking only around –30. Early on, both settings start with heavily negative rewards, but LR = 0.0005 climbs out of the worst losses slightly faster, reaching around –30 by episode 1100. In contrast, LR = 0.0001 remains more conservative initially, dipping only to about –45, then makes a sharper late gain that briefly surpasses positive territory before settling back near –40. This suggests that the smaller learning rate trades slower initial progress for the potential of larger late-stage breakthroughs, while the larger rate encourages steadier but ultimately more modest improvement.

4.2 Double DQN

In this project, we trained the Double Deep Q-Network (Double DQN) on the custom Tetris environment, using two learning rate settings: 0.0001 and 0.0005. The agent was trained for approximately 200000 frames. Episode-wise rewards were tracked to evaluate learning efficiency, convergence behavior, and policy stability.

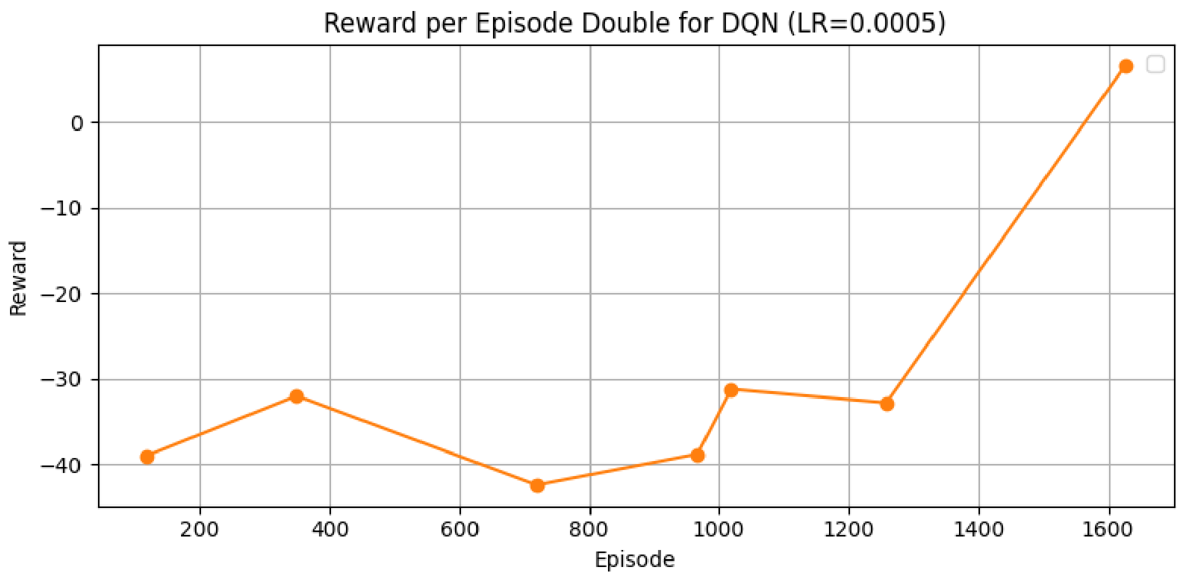

Figure 5: Reward for Double DQN with learning rate LR=0.0005

For Double DQN with a learning rate of 0.0005 starts off with deeply negative episode rewards (around –40), gradually improving to approximately –28 by mid-training. It then plateaus for several hundred episodes before exhibiting a dramatic late-stage breakthrough, finally reaching a positive reward just above zero near episode 1650. This pattern shows that, although overestimation bias is mitigated, a higher learning rate can still delay meaningful policy improvements, resulting in slow but eventual convergence.

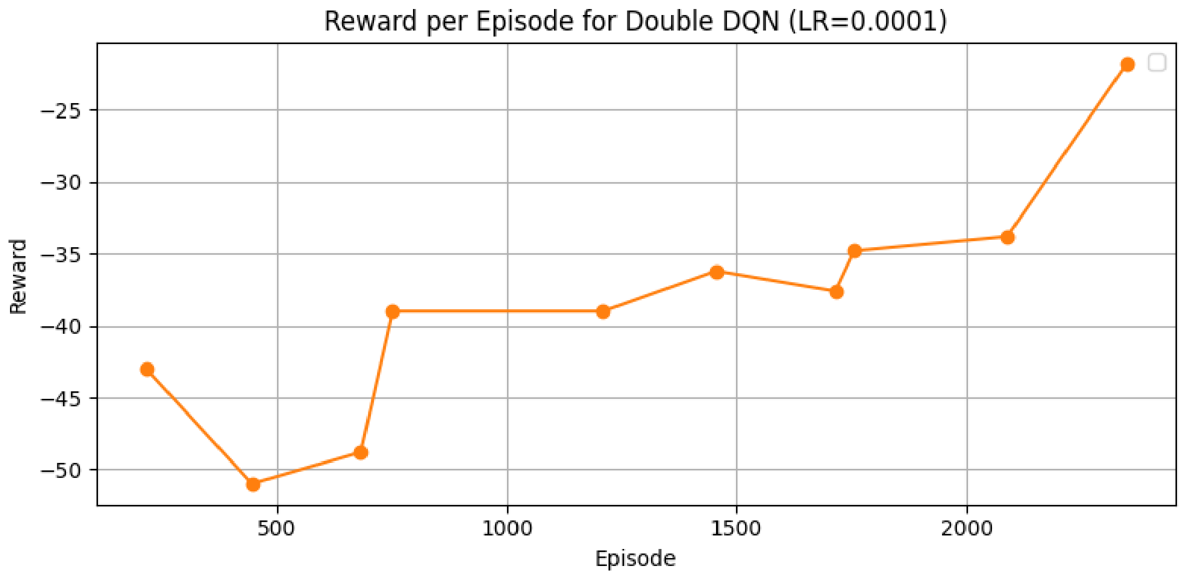

Figure 6: Reward for Double DQN with learning rate LR=0.0001

With a learning rate of 0.0001 exhibits a steady, if slow, improvement over roughly 2,300 episodes. Early on, episode rewards hover between –50 and –40, reflecting poor line clears and heavy hole penalties. Around episode 700, the agent makes its first consistent gain, jumping to about –39, then gradually climbs through the –35 to –33 range by mid-training. In the final phase (episodes 2,000–2,300), rewards rise more sharply, reaching approximately –22, suggesting the policy has begun to exploit line clears and minimize dead holes. This slow but stable ascent underscores the benefit of a small learning rate for reliable, incremental learning in Double DQN.

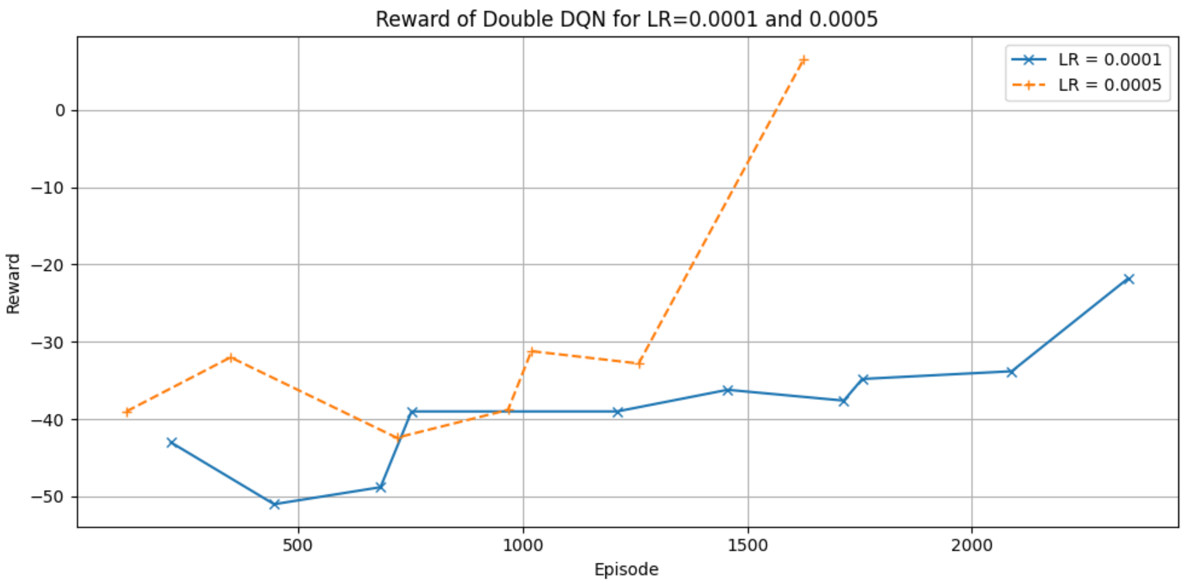

Figure 7: Reward for Double DQN with learning rate LR=0.0001 and 0.005

When comparing the two learning rates for Double DQN, we see that the higher rate (0.0005) yields faster but more erratic gains: by episode ~1,600 it already jumps into positive territory, indicating rapid early learning. In contrast, the lower rate (0.0001) produces a smoother, more gradual ascent—climbing steadily from around –40 up to –22 by episode ~2,300. Thus, while 0.0005 accelerates initial progress, 0.0001 offers more consistent stability over the full training run.

4.3 Dueling DQN

In the stage of our project, we trained the Dueling DQN algorithm on the custom Tetris environment for approximately 200000 frames using two learning rates: 0.0001 and 0.0005. The rewards per episode were recorded and analyzed to evaluate the performance and learning behavior of the agent.

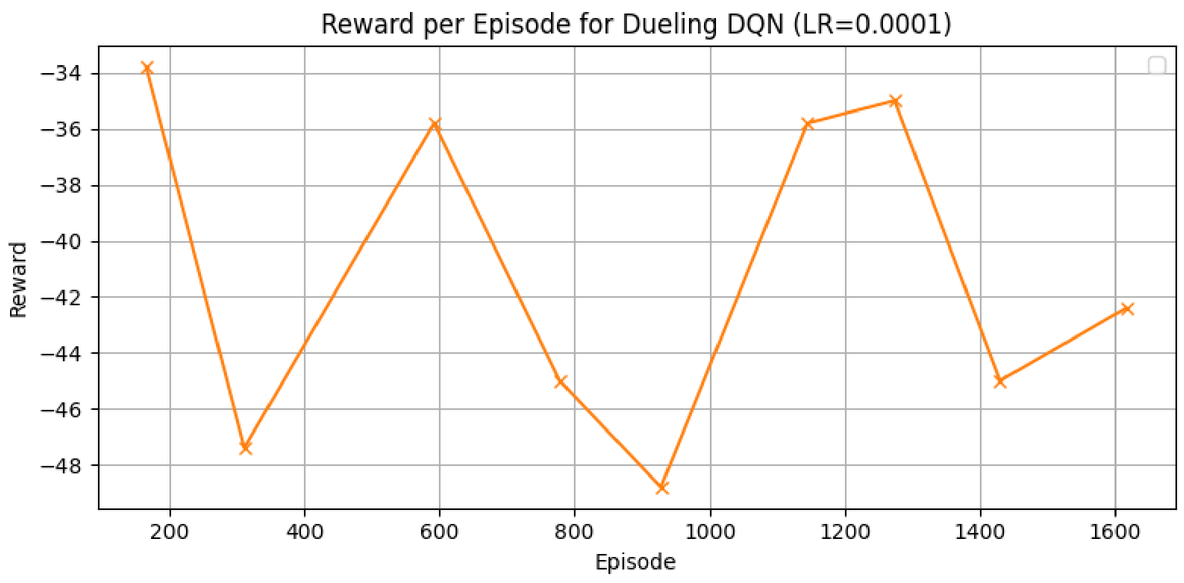

Figure 8: Reward for Dueling DQN with learning rate LR=0.0001

With a learning rate of 0.0001, Dueling DQN shows more muted but steady improvement over time. Early rewards start around –35, dip to –49 by episode 1,000, then recover to roughly –37– –35 by episode 1,200. Despite some variability, the model gradually climbs to –43 by episode 1,600. This suggests that splitting the value and advantage streams provides stable credit assignment, but at this low LR the agent's progress remains modest and slow.

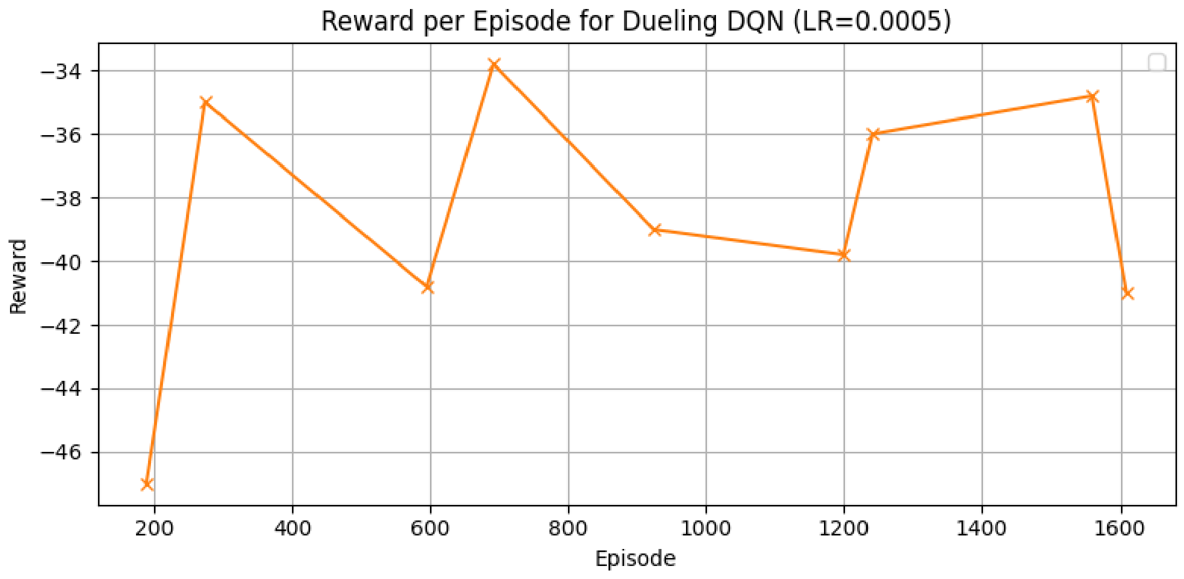

Figure 9: Reward for Dueling DQN with learning rate LR=0.0005

For the learning rate 0.0005, Dueling DQN converges faster to higher rewards compared to LR=0.0001. After just a few hundred episodes, rewards jump from about –47 to –35, peaking around –34 by episode 600. Although there's a slight dip back to –39 near episode 900, the model quickly recovers and stabilizes around –36 to –35 for the remainder of training. This suggests that the higher learning rate helps the dueling architecture adjust value and advantage streams more aggressively, yielding quicker gains and a steadier plateau in performance.

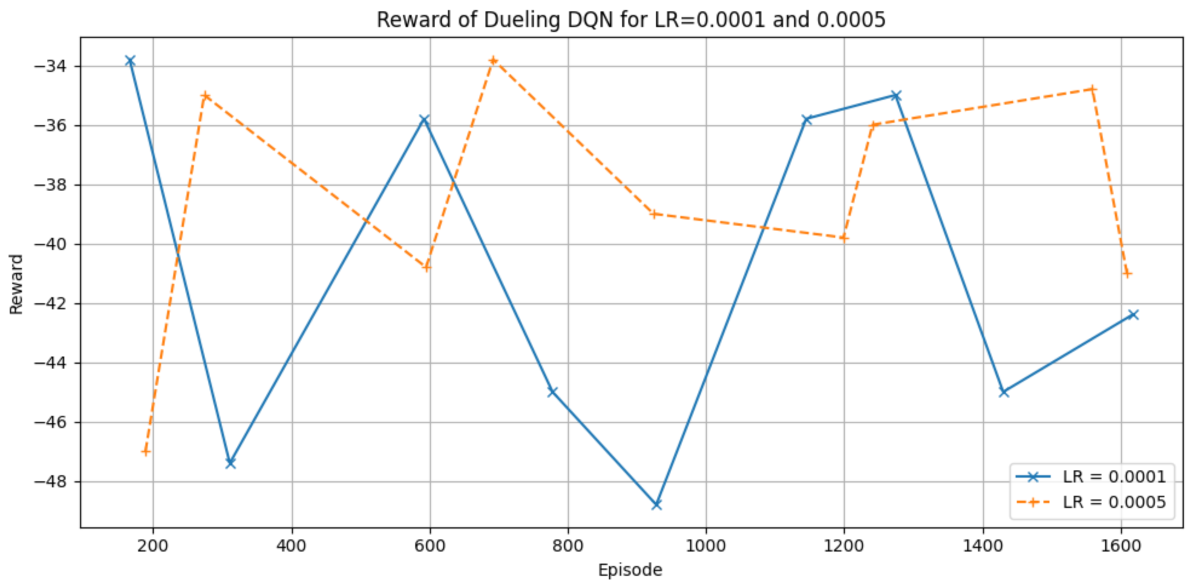

Figure 10: Reward for Dueling DQN with learning rate LR=0.0001 and 0.005

When comparing both curves, LR = 0.0005 achieves its initial jump earlier (by episode 300) and settles around –36 to –35, whereas LR = 0.0001 takes longer to climb (around episode 600) and exhibits more volatility, dipping as low as –49. Overall, the higher learning rate yields faster improvements and a tighter reward band, while the lower rate is slower to converge and more erratic.

4.4 Advantage Actor–Critic (A2C)

In the final phase of our evaluation, we applied the Advantage Actor-Critic (A2C) algorithm to the custom Tetris environment, training over 200 000 frames with two learning rates (0.0001 and 0.0005). We recorded the total reward per episode alongside both the policy and value losses to capture A2C's learning progression.

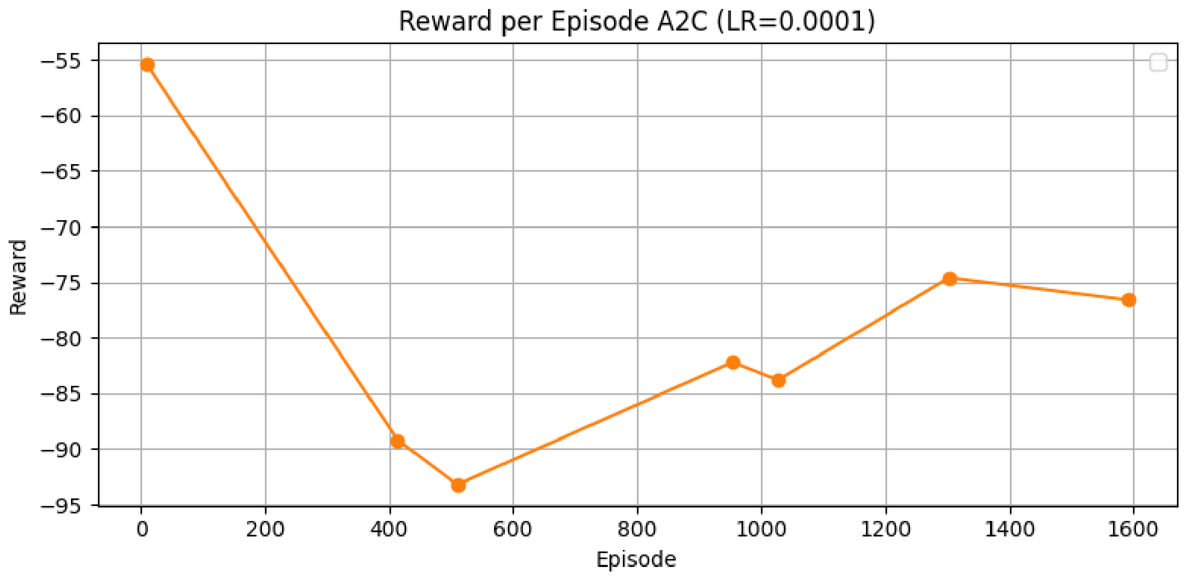

Figure 11: Reward for A2C with learning rate LR=0.0001

With a learning rate of 0.0001, A2C's episode rewards start around –55 and initially drop to about –93 by episode 500, reflecting early exploration and policy instability. After episode 500, rewards steadily improve, climbing to –83 by episode 1000 and reaching –75 by episode 1300. This upward trend suggests that the actor-critic updates are successfully refining both the policy and value estimates over time.

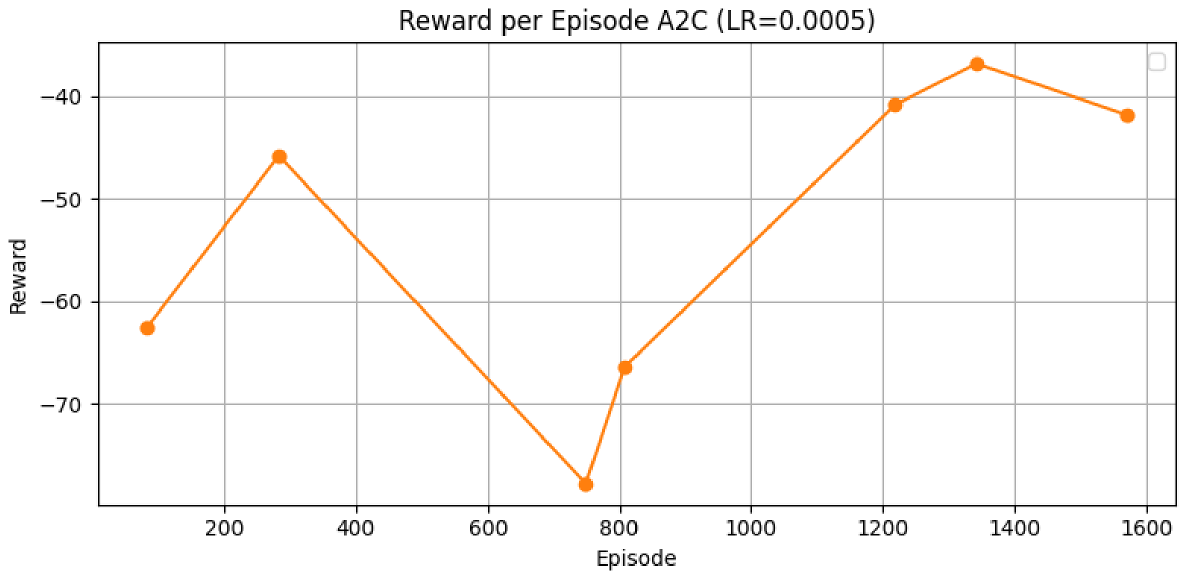

Figure 12: Reward for A2C with learning rate LR=0.0005

With a higher learning rate of 0.0005, A2C exhibits quicker initial improvements but greater volatility. Rewards jump from –63 at episode 100 to –45 by episode 300, then dip sharply to –78 around episode 750. Afterward, performance recovers more rapidly, rising to –41 by episode 1200 and peaking near –36 by episode 1300 before a slight decline to –42. This pattern shows that while a larger LR accelerates early learning, it also introduces instability, making the choice of LR a trade-off between speed and smoothness.

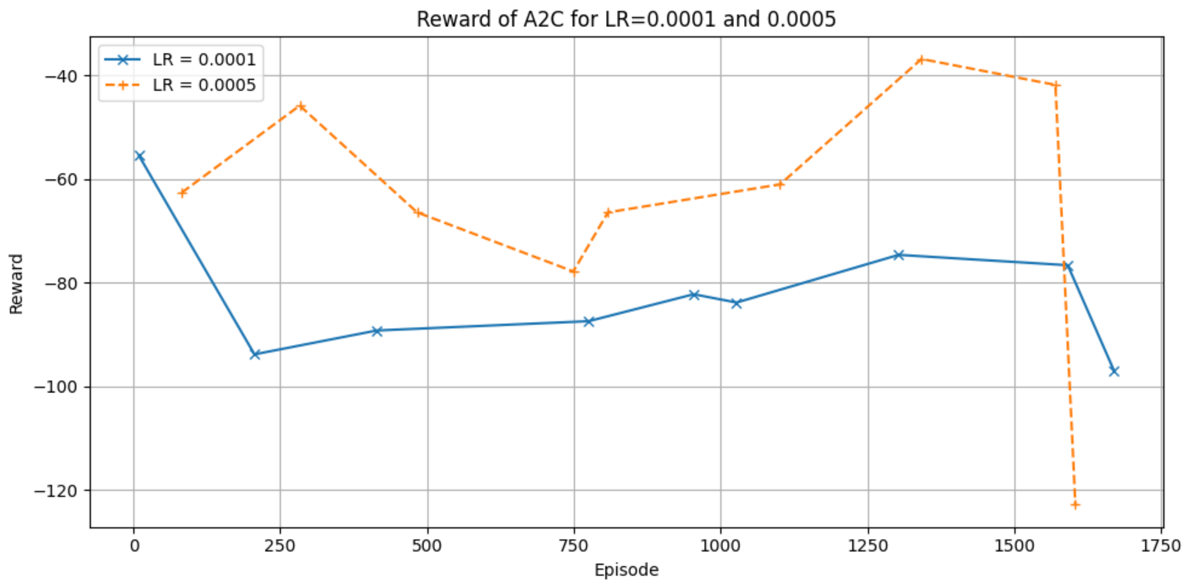

Figure 13: Reward for A2C with learning rate LR=0.0001 and 0.005

Comparing the two learning rates for A2C, we see that LR = 0.0005 (orange) consistently outperforms LR = 0.0001 (blue) across most checkpoints. With LR = 0.0005, the agent achieves higher rewards early—reaching about –45 by episode 300 versus –94 for the smaller rate—and maintains better performance thereafter, peaking near –36 around episode 1300. In contrast, LR = 0.0001 yields a slower, steadier ascent from –95 up to only –75 by episode 1300. However, the higher rate also shows a dramatic collapse to –123 at the final checkpoint, indicating increased risk of instability. Thus, while LR = 0.0005 accelerates learning and achieves stronger mid-training performance, LR = 0.0001 offers greater robustness at the cost of slower improvement.

CHAPTER 5: CONCLUSION

In this work, we built a custom Gym-style Tetris environment featuring stacked grayscale observations, a six-action discrete interface, and a carefully shaped reward that rewards line clears and survival while penalizing holes and uneven surfaces. We then used this environment to compare four deep reinforcement learning algorithms under identical training regimes.

We began with vanilla DQN as our baseline. By training for roughly 4 000 episodes, the DQN agent learned to clear lines but suffered from unstable updates and a tendency to overestimate its action-value estimates. These instabilities manifested as large reward fluctuations and sensitivity to the choice of learning rate, limiting its long-term performance.

To address that overestimation bias, we implemented Double DQN, which decouples action selection (via the online network) from action evaluation (via a periodically updated target network). This change produced noticeably smoother learning curves, faster convergence to high-reward policies, and higher average episode rewards compared to the vanilla DQN baseline.

Building on these gains, we next integrated the dueling architecture. Dueling DQN splits the Q-function into separate value and advantage streams, allowing the agent to better assess the inherent worth of each board state even when immediate action differences are small. In our experiments, Dueling DQN achieved the fastest convergence and the most consistent rewards of the Q-learning variants, demonstrating clear advantages in state-value estimation.

Finally, we incorporated Advantage Actor–Critic (A2C) to explore an on-policy, policy-gradient approach. Although A2C learned more slowly and attained lower absolute scores under our current reward shaping, it exhibited stable, monotonic loss reduction and showed promise for environments where value-based methods struggle with bias or sample efficiency.

Overall, our study confirms that architectural improvements—first through Double DQN's bias mitigation and then through Dueling DQN's value-advantage decomposition—substantially enhance standard DQN's stability and performance. Meanwhile, actor-critic methods like A2C offer complementary trade-offs, trading off raw score for steady optimization. Looking ahead, combining these architectures with techniques such as Prioritized Experience Replay, distributional value estimation, or advanced actor-critic algorithms (e.g., PPO, SAC) presents a promising path toward even stronger performance in high-dimensional, strategic games like Tetris.

REFERENCES

[1] Thiery, C., & Scherrer, B. (2009). "Improvements on learning Tetris with cross-entropy." Neurocomputing, 72(7-9), 1346-1352.

[2] Lewis, M. D., & Vamplew, P. (2005). "Generalisation properties of multiple sequence alignment techniques in Tetris." IEEE Transactions on Computational Intelligence and AI in Games, 1(1), 27-37.

[3] Sutton, R. S., & Barto, A. G. (2018). "Reinforcement Learning: An Introduction". MIT press.

[4] Van Hasselt, H., Guez, A., & Silver, D. (2016). "Deep Reinforcement Learning with Double Q-learning". Proceedings of the AAAI Conference on Artificial Intelligence, 30(1).

[5] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2016). "Dueling Network Architectures for Deep Reinforcement Learning". Proceedings of the 33rd International Conference on Machine Learning (ICML).

[6] Van Hasselt, H. (2010). "Double Q-learning". Advances in Neural Information Processing Systems (NIPS), 23.

[7] Bellemare, M. G., Dabney, W., & Munos, R. (2017). "A Distributional Perspective on Reinforcement Learning". Proceedings of the 34th International Conference on Machine Learning (ICML).

[8] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... & Silver, D. (2018). "Rainbow: Combining Improvements in Deep Reinforcement Learning". Proceedings of the AAAI Conference on Artificial Intelligence, 32(1).