Reinforcement Learning on Tetris: From Tabular Q to CNN-based DQN and Beyond

William Zhu [1] **Kevin Wang** [1]

Abstract

Tetris presents a challenging environment for reinforcement learning due to its high-dimensional state space and sparse, delayed rewards. In this project, we explore the viability of three progressively more powerful agents on Tetris. We begin with a tabular Q-learning agent trained on a reduced 6×6 board, which verifies environment learnability. We then scale to the full 10×20 board using Deep Q-Networks (DQN) with both multi-layer perceptron and convolutional neural network architectures. Finally, we introduce a model-based agent "Deep V-Network" that learns state values and performs look-ahead via successor enumeration. Our results show that tabular methods are limited by their inability to generalize, DQNs struggle with sparse rewards, and DVN achieves strong performance by leveraging the deterministic transitions of Tetris. Furthermore, we discuss on how imitation learning could help overcome the remaining credit-assignment challenges observed in DQN agents.

1. Introduction

Tetris is a widely popular video game with enduring appeal. As a reinforcement learning (RL) environment, Tetris requires decision-making under uncertainty due to random piece sequence, partial observability due to limited preview of future pieces, and compounding consequences due to piece placements and build-ups.

In this paper, we investigate the viability of reinforcement learning agents in Tetris through a staged progression of increasing complexity. We begin with a tabular Q-learning agent on a reduced 6×6 board as a proof of concept. We then scale to the full 10×20 board using deep Q-networks (DQN), using a multi-layer perceptron (MLP) and a convolutional neural network (CNN) architecture. Finally, we implement a model-based value learner that predicts the value of successor states through deep learning and selects actions accordingly. We refer to this agent as a "Deep V-Network" (DVN). Our results highlight the strengths and limitations of each approach, and motivate imitation learning as a promising direction for overcoming sparse-reward inefficiencies.

[*]Equal contribution [1]University of Chicago, IL, USA. Correspondence to: William Zhu <williamzhu@uchicago.edu>, Kevin Wang <haochuanwang@uchicago.edu>.

Submitted as part of CMSC 31170: Planning, Learning, and Estimation for Robotics and Artificial Intelligence, University of Chicago, Spring 2025.

2. Background and Related Work

2.1. The Rule of Tetris

Tetris is a single-player game played on a board, traditionally 10 columns wide and 20 rows tall. At each time step, the player receives a "tetromino", one of seven fixed shape pieces composed of smaller blocks, and must decide how to rotate and position it before it locks into place. Players score points by completing horizontal lines, which are then cleared from the board; incomplete lines remain. The game ends when the stack of blocks reaches the top of the board.

In our Gym-compatible environment, we train the agent playing Tetris. At each step, the agent observes the current board along with the current and next tetromino, then selects either rotation or horizontal placement. Once the piece locks into place, the board updates and a new tetromino appears; play continues until no valid placements remain. Also we simplify the rules by disabling gravity so pieces do not fall automatically and gaps will be left by previously cleared lines.

2.2. Related Work

Mnih et al. (Mnih et al., 2013) demonstrated the success of using DQN agents with CNN architectures across a suite of Atari 2600 games using pixel inputs and reward signals. And Van Hasselt et al.(Van Hasselt, 2016) further introduced Double DQN to mitigate value overestimation and stabilize training.

Tetris presents a significant challenge for reinforcement learning due to its sparse reward structure: line clears occur infrequently relative to the number of placement decisions, making it difficult for agents to assign credit to the actions that lead to success. Community-driven projects continue to refine Tetris agents. For example, Lee proposed a genetic algorithm that evolves weights over a set of engineered board features, allowing the agent to evaluate placements using a dense linear heuristic (Lee, 2013). Faria's tetris-ai on GitHub provides a collection of such explorations (Faria, 2013). Our work builds on these foundations by applying deep RL techniques to a more flexible Tetris board configuration, analyzing their limitations, and proposing a model-based alternative that sidesteps action credit assignment through successor state evaluation.

3. Tabular Q-Learning

3.1. Motivation

We first implemented a tabular Q-learning agent as a proof of concept. Tabular Q-learning provides a straightforward approach to RL. It maintains a table of action-values $Q(s, a)$ and updates them using the Bellman equation. This transparency makes it an ideal baseline for verifying our Tetris environment. If a tabular agent succeeds in a simplified environment, then we gain confidence that the environment dynamics and reward signal are learnable.

However, applying tabular Q-learning to the full Tetris environment is computationally expensive. In standard 10×20 boards, the total number of distinct board states is bounded only by $2^{10\times20}$. Even if data storage is possible, many states are unlikely to be reached in reasonable training time. Hence, we start by testing the agent on a reduced 6×6 board. This smaller configuration drastically limits the number of reachable states, making it possible to represent the Q-function explicitly. While such a board is not representative of full Tetris play, it retains the core dynamics: piece placement, line clearing, and game-over conditions.

3.2. Implementation

Our tabular agent operates on a flattened representation of the game state. Each observation consists of the current board state and the indices of the current and next tetromino. The board is flattened into a binary vector of size $6 \times 6 = 36$, and each piece index is converted to a one-hot vector of length 7. The final state vector is of dimension 50, formed by concatenating the flattened board, current piece, and next piece encodings. The agent uses the standard Q-learning update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right] \tag{1}$$

where α is the learning rate, and γ is the discount factor. Exploration is handled via an epsilon-greedy policy, with epsilon ϵ decayed after each episode.

The training algorithm is outlined in Algorithm 1.

Algorithm 1 Tabular Q-Learning

```
1: Initialize Q-table $Q(s,a) \leftarrow 0$
2: for episode = 1 to $N$ do
3:   Reset environment; observe $s$ and valid action $\mathcal{A}$
4:   for step = 1 to max_steps do
5:     With probability $\epsilon$ select random action $a \in \mathcal{A}$
6:     Otherwise select $a = \arg\max_{a' \in \mathcal{A}} Q(s, a')$
7:     Execute action $a$, observe reward $r$, next state $s'$,
       and next state valid actions $\mathcal{A}'$
8:     Learn by updating $Q(s,a)$ using Eq. (1)
9:     $s \leftarrow s'$, $\mathcal{A} \leftarrow \mathcal{A}'$
10:    if game over or max_steps reached then
11:      break
12:    end if
13:  end for
14:  Decay $\epsilon$: $\epsilon \leftarrow \max(\epsilon \cdot \epsilon_{\text{decay}}, \epsilon_{\min})$
15: end for
```

The action space includes all valid (rotation, x-position) placements for the current tetromino. Importantly, invalid actions are filtered to obtain $\mathcal{A}$ for each step, ensuring that all chosen actions are executable.

The reward is designed to encourage line clearing and penalize game termination. Specifically, the agent receives a survival reward, plus a Tetris line-clear reward, scaled by board width w and height h:

$$r = 10/h + (\text{lines\_cleared})^2 \cdot w \tag{2}$$

This formulation incentivize the agent to clear multiple lines simultaneously to get larger reward. And if the game ends, the agent receives a game-over penalty of -10, which aim to discourages agent from doing unsafe stacking.

We used the following hyper-parameters: learning rate $\alpha = 0.05$ for faster training due to time consideration, discount factor $\gamma = 0.9$ to encourage future planning, initial exploration rate $\epsilon = 1.0$ with exponential decay rate 0.99995 to a minimum of 0.1 to maintain some exploration.

3.3. Results

The agent was trained for $1{,}000{,}000$ episodes with a per-episode step cap of 100 on 6x6 board. Figure 1 shows the average reward obtained in training where each data point is the average reward over a $10{,}000$-episode window. The average episode reward improves steadily during early training and plateaus around 5.3. This indicates that the agent successfully learned a policy capable of surviving and somewhat capable of line clearing. The reward ceiling arises from two key factors. First, tabular methods cannot generalize across similar states. With a large and sparsely explored

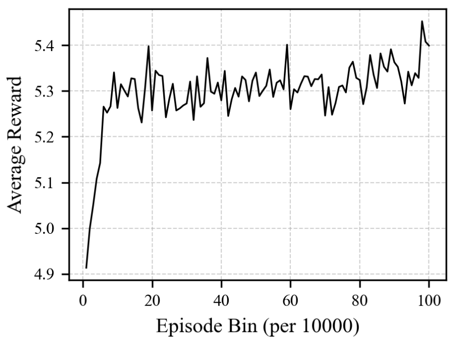

Figure 1. Average episode reward for the Tabular Q-learning agent

state space, some useful (s, a) pairs remain unvisited. Second, performance in small-board Tetris is highly sensitive to the stochastic sequence of incoming tetromino pieces, limiting the achievable reward even under optimal play.

4. Deep Q-Networks

4.1. Motivation

While tabular Q-learning provides a useful proof of concept, it fails to scale to realistic versions of Tetris. To overcome these limitations, we transition to function approximation. DQN replaces the tabular Q-function with a neural network $Q_\theta(s, a)$ parameterized by weights θ. By generalizing across similar states, the DQN agent can learn meaningful patterns in high-dimensional environments even with limited visitation.

The network is trained to minimize the Bellman error using observed transitions:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(Q_\theta(s,a) - y\right)^2\right], \quad (3)$$

$$y = r + \gamma \max_{a'} Q_{\tilde{\theta}}(s', a') \quad (4)$$

Here, $\tilde{\theta}$ denotes the parameters of a target network, which is periodically synchronized with the main network to stabilize training. Experience replay is used to randomly sample mini-batches from a replay buffer $\mathcal{D}$ to break the temporal correlations between consecutive transitions and improving data efficiency (Mnih et al., 2013). Finally, to further stabilize training and mitigate overestimation bias, we adopt a "Double DQN" structure, where the target is computed using the target network to evaluate the action selected by the online network (van Hasselt et al., 2015).

In our implementation, we explore two network architectures for Q_θ: a multi-layer perceptron (MLP) operating on a flattened board and engineered features, and a convolutional neural network (CNN) designed to exploit the spatial structure of the board layout.

4.2. Implementation

Both our MLP and CNN agents implement the Double DQN algorithm with experience replay and a target network. The high-level training loop is identical across agents, and is outlined in Algorithm 2. Reward is the same as Eq. (2).

Algorithm 2 Double Deep Q-Network with Replay

```
1: Initialize Q-network $Q_\theta$ and target network $Q_{\theta^-} \leftarrow Q_\theta$
2: Initialize empty replay buffer $\mathcal{D}$
3: for episode = 1 to $N$ do
4:     Reset environment; observe state $s$, valid action $\mathcal{A}$
5:     for step = 1 to max_steps do
6:         With probability $\epsilon$ select random action $a \in \mathcal{A}$
7:         Otherwise select $a = \arg\max_{a' \in \mathcal{A}} Q_\theta(s, a')$
8:         Execute $a$, observe reward $r$, next state $s'$, terminal
           state indicator $d$, and valid action $\mathcal{A}'$
9:         Store $(s, a, r, s', d)$ in $\mathcal{D}$
10:        Sample a mini-batch $\{(s_i, a_i, r_i, s'_i, d_i)\}_{i=1}^{B}$ from
           replay buffer $\mathcal{D}$
11:        For each $i$, compute:
               $a_i^* = \arg\max_{a'} Q_\theta(s'_i, a')$   (online network)
               $y_i = r_i + \gamma \cdot Q_{\tilde{\theta}}(s'_i, a_i^*) \cdot (1 - d_i)$
12:        Update $\theta$ by gradient descent on $\mathcal{L}(\theta)$ from Eq. (3)
13:        $s \leftarrow s'$, $\mathcal{A} \leftarrow \mathcal{A}'$
14:        If episode is done, break
15:    end for
16:    Every $K$ episodes: update $Q_{\tilde{\theta}} \leftarrow Q_\theta$
17:    Decay $\epsilon$: $\epsilon \leftarrow \max(\epsilon \cdot \epsilon_{\text{decay}}, \epsilon_{\text{min}})$
18: end for
```

4.2.1. MLP Agent

The MLP-based agent operates on a flattened vector representation of the state. Specifically, we concatenate the binary 10×20 board (flattened to a 200-dimensional vector), one-hot encodings of the current and next tetromino (each of length 7), and six handcrafted features capturing key structural properties of the board. The features are inspired by Lee (Lee, 2013).

We formally define the board features as follows. Let $\mathcal{B} \in \{0,1\}^{20\times10}$ denote a binary matrix representing the Tetris board, where $\mathcal{B}_{i,j} = 1$ indicates that cell (i, j) is filled. Rows are indexed top to bottom, so smaller values of i correspond to higher rows, with $i = 0$ denoting the topmost row. We define the column height vector $\vec{h} \in \mathbb{R}^{10}$ as the height of each column of the Tetris board:

$$h_j = 20 - \min\{i \mid \mathcal{B}_{i,j} = 1\}, \quad (5)$$

and bumpiness vector $\vec{\delta} \in \mathbb{R}^9$ as the column's difference in height with its right-adjacent column:

$$\delta_j = |h_j - h_{j+1}|. \quad (6)$$

Then, we have the following features:

- **Maximum height:** $\max_j h_j$
- **Minimum height:** $\min_j h_j$
- **Total height:** $\sum_{j=1}^{10} h_j$
- **Maximum bumpiness:** $\max_j \delta_j$
- **Total bumpiness:** $\sum_{j=1}^{9} \delta_j$
- **Total number of holes:** A hole is an empty cell such that there is a filled cell on top of it. i.e., A hole is an empty cell $\mathcal{B}_{i,j}$ such that there exists $i' < i$ with $\mathcal{B}_{i',j} = 1$.

The Q-network is a two-layer multilayer perceptron:

- Fully connected layer with 128 units + ReLU
- Fully connected layer with 64 units + ReLU
- Output layer with $|\mathcal{A}_{\text{full}}|$ units

where $|\mathcal{A}_{\text{full}}|$ denotes the total number of unique (rotation, placement) action pairs possible across all states. For any given state, only a subset of these actions are valid (i.e., $\mathcal{A}$) and used for action selection.

We use Adam optimizer with a learning rate $\alpha = 0.001$. We set discount factor $\gamma = 0.99$, batch size of 32, and target network update frequency of 10 episodes per update. The replay buffer has a capacity of 10,000 transitions. Exploration follows an epsilon-greedy policy with initial exploration rate $\epsilon = 1.0$, exponential decay rate of 0.9995, and minimum value $\epsilon_{\min} = 0.1$.

4.2.2. CNN Agent

The CNN-based agent uses a tensorized state representation with shape $(15, 20, 10)$. The first channel encodes the binary board (1 if filled, 0 if empty), which is the only input processed by the convolutional layers. The remaining 14 channels represent one-hot encodings of the current and next tetromino types; these are constant planes and are not passed through the convolutional stack. Instead, they are extracted as a 14-dimensional vector and concatenated with the flattened convolutional features before entering the fully connected layers. This design allows the CNN to extract local spatial patterns from the board, while injecting piece identity as global context during action-value prediction.

The Q-network architecture is as follows:

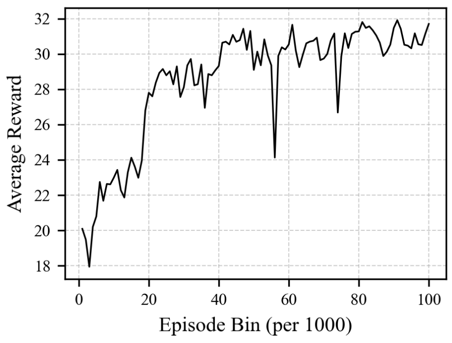

Figure 2. Average episode reward for the MLP Q-learning agent

- Convolutional layer: $1 \rightarrow 32$ channels, kernel size 3×3, padding 1, followed by ReLU
- Convolutional layer: $32 \rightarrow 32$ channels, kernel size 3×3, padding 1, followed by ReLU
- Max pooling layer: 2×2, reducing spatial size from 20×10 to 10×5
- Flattened feature vector generated by the CNN layers concatenated with a 14-dimensional piece identity one-hot encoding vector
- Fully connected layer with 128 units + ReLU
- Fully connected layer with 64 units + ReLU
- Output layer with $|\mathcal{A}_{\text{full}}|$ units (one per action)

Section A.1 shows the CNN network structure.

We use the Adam optimizer with learning rate $\alpha = 0.001$. We set discount factor $\gamma = 0.99$, batch size to be 32, and target update frequency to be 10 episodes per update. The replay buffer has a capacity of 10,000 transitions. Exploration follows an epsilon-greedy policy with initial exploration rate $\epsilon = 1.0$, exponential decay rate 0.9995, and minimum value $\epsilon_{\min} = 0.1$.

4.3. Results

Both the MLP DQN agent and the CNN DQN agent are trained for $100{,}000$ episodes. Figure 2 shows the per $1{,}000$-episode average training reward curve of the MLP DQN agent. The agent demonstrates rapid early improvement, with the average reward increasing in the first 20,000 episodes. This is followed by a plateau around a reward of 30, suggesting that the agent converges somewhat to a locally optimal policy. Occasional dips in performance indicate some instability, likely due to overfitting to frequently seen board configurations or limited generalization from the handcrafted features.

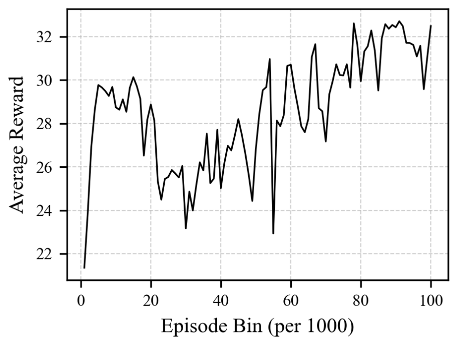

Figure 3. Average episode reward for the CNN Q-learning agent

Figure 3 shows the per 1,000-episode average training reward curve for the CNN DQN agent. The agent achieves rapid early gains, with average reward rising to nearly 30 within the first 10,000 episodes. However, the learning process becomes notably unstable afterwards, with sharp oscillations and a performance dip. This instability may be attributed to overfitting to local board patterns or the difficulty of optimizing convolutional features from sparse reward signals. Nonetheless, the agent recovers in later stages to a performance level comparable to the MLP agent, suggesting that spatial features extracted by the CNN can eventually support competent play, albeit with a longer and noisier learning trajectory.

Our experiments with DQN agents revealed instability and slow convergence in learning reliable action-value estimates. As seen in the reward curves for both the MLP and CNN agents, performance gains are inconsistent, with extended plateaus and sharp fluctuations. In DQN, the agent attempts to learn the value of each action given the current state. This requires the network to implicitly learn the dynamics of the environment — that is, what each action does and whether the resulting state will eventually lead to reward. In Tetris, the action space is large and context-dependent: the same action can be optimal in one board configuration and disastrous in another that is identical except for a few cells. Learning a single unified function over all such (s, a) pairs requires extensive exploration, which is unrealistic to our project's scope.

5. "Deep V-Network"

5.1. Motivation

Tetris has fully deterministic transitions: given the current piece and board, the result of an action is uniquely determined. We can exploit this by directly enumerating all possible next states s' from the current state s, and selecting the action that leads to the highest predicted state value $V(s')$. This removes the burden of modeling the action's consequence encountered in Section 4 by our DQN agents. The agent in this case only needs to evaluate the value of outcomes, not learn how actions transform states. In short, we adopt a model-based approach.

Formally, we aim to learn a function

$$V(s) \approx \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right], \tag{7}$$

which assigns to each board state an estimate of its long-term return. At each timestep, the agent exhaustively simulates all valid successor states s' and selects the action that leads to the highest predicted $V(s')$. Since the transitions are deterministic and the successor states are known exactly, this amounts to a form of value-based planning using a learned value function. We refer to this approach as a "Deep V-Network (DVN)", reflecting its role as a deep neural approximation of the state-value function.

To train the value network $V_\theta(s)$, we adopt a standard $TD(0)$ learning setup using bootstrapped targets. The network is optimized to minimize the Bellman error over sampled transitions:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,r,s',d)\sim\mathcal{D}}\left[\left[V_\theta(s) - (r + \gamma V_\theta(s'))\right]^2\right]. \tag{8}$$

Transitions are stored in a replay buffer $\mathcal{D}$, and mini-batches are sampled uniformly to break temporal correlations and stabilize training.

This design has multiple advantages. First, it reduces the learning problem to a lower-dimensional input space, avoiding the combinatorial explosion of (s, a) pairs. Second, it leverages full knowledge of the environment's transition model via enumeration, allowing the agent to focus purely on predicting the value of board configurations. Finally, it still uses gradient-based deep learning for scalability of the Tetris board.

5.2. Implementation

The DVN agent approximates the state-value function $V(s)$ using a two-layer MLP trained via $TD(0)$ learning. At each step, the agent enumerates all valid next states s' resulting from legal placements of the current tetromino, evaluates $V(s')$ for each s', and selects the action that leads to the highest predicted value. The overall training loop is shown in Algorithm 3.

The input to V_θ (i.e., the state) is a handcrafted feature vector extracted from the raw board state. It includes the same seven features described in Section 4.2.1. Note that these features are computed deterministically from the board and

Algorithm 3 Deep V-Network with Successor Enumeration

```
1: Initialize value network $V_\theta$ and empty replay buffer $\mathcal{D}$
2: for episode = 1 to $N$ do
3:     Reset environment; observe initial state $s$, valid action $\mathcal{A}$
4:     for step = 1 to max_steps do
5:         Find all valid successor states: $\mathcal{S} = \{(a_i, s'_i) \mid a_i \in \mathcal{A}(s)\}$
6:         With probability $\epsilon$, select random action $a \in \mathcal{A}$
7:         Otherwise, select $a = \arg\max_{(a',s')\in\mathcal{S}} V_\theta(s')$
8:         Execute $a$; observe reward $r$, next state $s'$, terminal state indicator $d$, and valid action $\mathcal{A}'$
9:         Store transition $(s, r, s', d)$ in $\mathcal{D}$
10:        Sample mini-batch $\{(s_i, r_i, s'_i, d_i)\}_{i=1}^{B}$ from $\mathcal{D}$
11:        Compute TD targets:
               $y_i = r_i + \gamma V_\theta(s'_i) \cdot (1 - d_i)$
12:        Update $\theta$ by gradient descent on $\mathcal{L}(\theta)$ from Eq. (8)
13:        $s \leftarrow s'$, $\mathcal{A} \leftarrow \mathcal{A}'$
14:        If episode is done, break
15:    end for
16:    Decay $\epsilon$: $\epsilon \leftarrow \max(\epsilon \cdot \epsilon_{\text{decay}}, \epsilon_{\min})$
17: end for
```

reduce the state space to a low-dimensional representation. We use the same reward-shaping as outlined in Eq. (2).

The value network V_θ is a two-layer MLP:

- Fully connected layer with 128 units + ReLU
- Fully connected layer with 64 units + ReLU
- Output layer with 1 unit

The final output is a scalar prediction of the expected cumulative reward from the input state.

We train the network using the Adam optimizer with a learning rate $\alpha = 0.0001$. The discount factor is $\gamma = 0.99$. We use a replay buffer of size 10,000 and sample mini-batches of 64 transitions for each update. Exploration is governed by an epsilon-greedy policy, with exploration rate ϵ decayed exponentially from 1.0 to 0.1 at a rate of 0.9995. Each episode runs up to 2,000 steps.

5.3. Results

The DVN agent is trained for 5,000 episodes. [1] Figure 4 shows the average reward obtained in training where each data point is the average reward over a 50-episode window. It demonstrates clear improvement throughout training, with

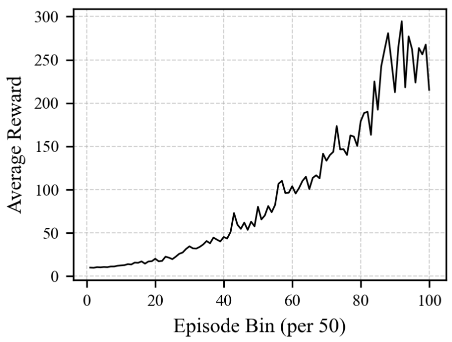

Figure 4. Average episode reward for the DVN agent

accelerating gains as episode increases. This suggests that the learned value function $V(s)$ enables increasingly effective action selection. Unlike our earlier agents, the DVN model appears to learn policies that generalize across board configurations, as evidenced by its ability to consistently achieve rewards exceeding 200 by the end of training.

6. Discussion and Conclusion

6.1. Evaluation Metrics Analysis

To assess the quality of learned policies, we evaluate each agent by running it in a greedy mode (i.e., $\epsilon = 0$) for 1,000 episodes and recording total reward, raw Tetris (i.e., lines cleared) score, and episode length.

Section A.2 showcases the evaluation metrics. Across all metrics, our agents consistently outperform their respective random baselines, confirming that each has learned non-trivial behavior. On the 6×6 board, the Tabular Q-learning agent shows only modest improvements in reward and survival compared to the random agent, but nearly doubles the average score. This suggests that while the agent does not significantly extend lifespan, it has learned to clear lines more efficiently — likely by memorizing effective placements in the limited state space.

On the standard 20×10 board, the gap between trained agents and the random baseline is much larger. Both the MLP and CNN-based DQN agents achieve substantially higher average reward and episode length. However, their average scores remain low: despite surviving over 30 steps on average, the MLP agent clears fewer than one line per episode on average. This indicates that the agents have primarily learned to avoid game-over states rather than to pursue line-clearing strategies. This behavior likely stems from the credit assignment problem: line-clearing often results from a sequence of moves, making it difficult for agents to connect delayed reward to specific actions. As a result, DQN agents often default to learning survival-oriented behaviors, since avoiding game-over states yields

[1]We use a short training schedule because we see promising results early on.

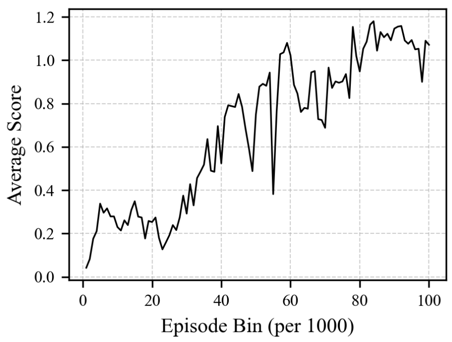

Figure 5. Average episode score for the CNN agent

more immediate and reliable feedback.

In contrast, the DVN agent demonstrates a dramatic improvement across all metrics. It survives nearly six times longer than other agents and scores over 220 per game on average, with a maximum of $1{,}451$ score in a single run. This reflects the advantage of model-based planning: rather than learning $Q(s, a)$ mappings directly, the DVN agent learns to predict the value of board states and selects actions by exhaustively simulating their outcomes. This decouples learning from delayed action effects and enables more globally coherent behavior.

6.2. Learning Dynamics and Future Directions

The CNN agent's average raw Tetris score is shown in Figure 5. While the average reward fluctuates noticeably during training, the score increases relatively steadily. This discrepancy suggests that although the agent's overall episode performance remains volatile, it is gradually learning to clear lines more consistently. This is a promising signal: it indicates that the agent is beginning to internalize scoring strategies. However, because line-clearing rewards are sparse and difficult to encounter through unguided trial-and-error, the agent receives limited feedback during early training, making learning inefficient.

One promising direction for addressing the delayed and sparse reward problem is Imitation learning (IL). IL aims to train policies that mimics expert behavior using supervised learning, rather than relying solely on reinforcement feedback from trial-and-error.

For instance, a similar challenge arises in Go, a game with long-term strategic dependencies and sparse rewards. Silver et al. (Silver et al., 2016) addressed delayed and sparse reward by first training a policy via supervised learning on expert demonstrations, before refining it through reinforcement learning. This approach enabled the agent to focus early learning on high-value regions of the state space and greatly accelerated policy improvement.

We can adopt a similar strategy for Tetris. By training on expert demonstrations, from, for example, recordings of Tetris World Championship, the agent can learn to recognize reward-relevant structures even before it receives sufficient feedback through exploration. This could mitigate the survival bias observed in our DQN agents and improve credit assignment in delayed and sparse reward regimes.

Acknowledgements

This project was completed as part of CMSC 31170 at the University of Chicago. We sincerely thank Professor Matthew Walter and TA Tianchong Jiang.

References

Faria, N. tetris-ai, 2013. URL `\url{https://github.com/nuno-faria/tetris-ai}`. Accessed: 2025-05-25.

Lee, Y. Tetris ai – the (near) perfect bot, 2013. URL `\url{https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/}`. Accessed: 2025-05-23.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. ISSN 1476-4687. doi: 10.1038/nature16961. URL `https://doi.org/10.1038/nature16961`.

van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL `http://arxiv.org/abs/1509.06461`.

A. Appendix

A.1. CNN Architecture

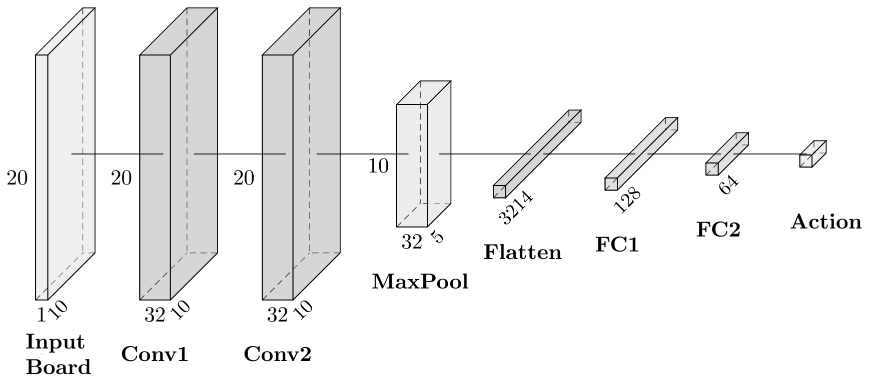

Figure 6. CNN architecture used in our DQN agent.

A.2. Evaluation Metrics

Table 1. Reward statistics over 1,000 evaluation episodes.

	6×6 Board		20×10 Board			
Metric	Random	Tabular Q	Random	MLP DQN	CNN DQN	DVN
Mean	7.30	8.23	10.21	32.74	39.70	1148.59
Std Dev	3.79	4.69	2.25	7.29	6.95	1172.97
Min	3.33	3.33	5.00	13.00	19.00	18.00
Max	45.00	43.33	24.00	89.00	81.00	7721.00

Table 2. Raw Tetris score statistics over 1,000 evaluation episodes.

	6×6 Board		20×10 Board			
Metric	Random	Tabular Q	Random	MLP DQN	CNN DQN	DVN
Mean	0.19	0.43	0.03	0.42	1.88	222.29
Std Dev	0.71	0.94	0.24	0.96	0.98	233.03
Min	0.00	0.00	0.00	0.00	0.00	0.00
Max	7.00	7.00	2.00	7.00	8.00	1451.00

Table 3. Episode length statistics over 1,000 evaluation episodes.

	6×6 Board		20×10 Board			
Metric	Random	Tabular Q	Random	MLP DQN	CNN DQN	DVN
Mean	4.03	4.16	20.14	30.59	30.29	174.81
Std Dev	1.24	1.28	3.45	3.32	3.01	144.81
Min	2.00	2.00	10.00	13.00	17.00	35.00
Max	10.00	10.00	30.00	39.00	41.00	982.00