Playing Tetris with Cross-Entropy Method and V-Learning

Author: Wenhui Ma

V-Number: V01062702

ECE 556: Deep Reinforcement Learning (Summer 2025)

Department of Electrical and Computer Engineering

University of Victoria

August 15, 2025

Abstract

Tetris is deceptively simple for humans yet algorithmically challenging, making it a classic benchmark for reinforcement learning research. This project compares two reinforcement learning approaches for Tetris: the Cross-Entropy Method (CEM) and V-Learning. We formulate Tetris as a Markov Decision Process with a compact state representation and evaluate both methods in a shared experimental setup. CEM performs policy search by optimizing linear weights over handcrafted board features, selecting the highest-scoring final placement. V-Learning uses a convolutional neural network to approximate the state value function V(s), enabling bootstrapped updates and richer board evaluation.

Experiments show that V-Learning outperforms CEM, averaging 792 lines cleared per game versus CEM's 319.8. The improvement comes from better generalization to unseen board states and the ability to refine estimates through bootstrapping. Our contributions are: (1) a unified MDP formulation for Tetris; (2) reproducible implementations of CEM and V-Learning; and (3) a comparative analysis of policy search versus value-based learning.

We recommend V-Learning when computational resources allow, as it yields stronger and more stable play. CEM offers a simple and interpretable baseline, making it useful for educational purposes and for testing feature designs. However, its computational inefficiency and slow convergence make it less suitable for practical training compared to modern value-based methods.

1 Introduction

Tetris, created in 1984, has become not only one of the most popular video games but also a significant benchmark problem in artificial intelligence and machine learning research. The game's simple rules yet complex decision space make it an ideal testbed for studying sequential decision-making under uncertainty. Despite its simple mechanics, finding optimal piece placements in Tetris is NP-complete, even when the sequence of pieces is known in advance [4].

The development of artificial intelligence approaches to Tetris reflects the broader evolution of reinforcement learning (RL). Early theoretical foundations were established by Bertsekas and Tsitsiklis's work on neuro-dynamic programming [1], which achieved around 2,800 lines, followed by Lagoudakis and Parr's least-squares policy iteration (LSPI) [2], reaching 1,000–3,000 lines. A significant non-RL breakthrough came from Dellacherie's carefully designed heuristic evaluation functions [3], demonstrating the power of feature engineering and reaching 630,000 lines. Moving back to learning, Szita and Lőrincz optimized feature weights with the noisy cross-entropy method (CEM) [4]. Although

it achieved 348,895 lines—below the handcrafted approach—it established a robust learning-based baseline. Building on this, Thiery and Scherrer [5] showed that enhanced feature engineering combined with cross-entropy optimization could push performance further, reporting up to 35 million lines. Later, Gabillon et al. [6] demonstrated that approximate dynamic programming via classification-based policy iteration (CAPI) can perform remarkably well, reaching 51 million lines.

As comprehensively reviewed by Algorta and Şimşek [8], traditional approaches emphasizing feature engineering and strong evaluation functions often outperform more complex end-to-end methods on the original game. Recent attempts to apply deep RL directly to Tetris face significant challenges: student projects at Stanford [7] and CUHK [9] show that modern deep RL struggles with the original game's large state–action space, although simplified variants with pruning can be solved effectively.

This picture aligns with open-source practice. While several implementations explore deep Q-learning [10, 11], the strongest systems—such as StackRabbit [12]—rely on non-RL heuristic search and feature-rich evaluation, augmented by offline value-iteration ranks and efficient C++ search. Community resources like CodeBullet's implementation walkthrough [14] and educational blogs [13] document the V-learning method, while supervised learning from human data [15] sits outside our RL scope.

Overall, the consistent success of feature-based approaches—culminating in StackRabbit's record of 102 million points and reaching level 237 [16]—suggests that for the original Tetris, sophisticated end-to-end value learning may be unnecessary. Instead, careful feature engineering with greedy or shallow-search decision rules can suffice.

Research Problem and Objectives As a course project in reinforcement learning, our goal is to gain hands-on experience with different RL approaches by implementing and comparing two well-established methods for Tetris.

Our research objectives are straightforward:

1. **Implementation Experience**: Implement two different RL methods - Cross-Entropy Method (CEM) [4] for policy search and V-Learning [13] for value function approximation - to understand their practical implementation challenges and requirements.
2. **Method Comparison**: Compare how these two approaches perform on the same Tetris environment, analyzing their strengths, weaknesses, and learning characteristics.

Why These Methods: We chose CEM and V-Learning because:

- **Proven Effectiveness**: Both methods have demonstrated good performance on Tetris in previous work
- **Implementation Feasibility**: They are well-documented and suitable for this project scope
- **Paradigm Diversity**: They represent different RL approaches (policy search vs. value function approximation), providing valuable learning experience

Our Approach: We focus on practical implementation and systematic comparison rather than achieving record-breaking performance. By working with the same environment and evaluation protocols, we can directly compare how different RL paradigms learn to play Tetris and what insights this provides about their relative strengths.

2 MDP Problem Formulation

2.1 Problem Goal

The goal of Tetris is to maximize the number of lines cleared while surviving as long as possible. Players must strategically place falling tetrominoes to complete horizontal rows, which are then cleared from the board. The game ends when the stack of pieces reaches the top of the playing field.

2.2 Agent and Environment

The **agent** is the player (either a human or an AI controller) that selects actions at each time step. The **environment** is the Tetris game engine, which:

- Displays the current board and falling tetromino,
- Applies gravity and collision detection,
- Clears filled rows,
- Generates the next tetromino.

2.3 State Space

2.3.1 CEM State Representation

For the Cross-Entropy Method, we use a **compact feature vector** inspired by Dellacherie's work [3]:

1. **Number of Holes**: Empty cells below filled cells that cannot be filled by future pieces
2. **Landing Height**: The lowest possible landing position for the current piece
3. **Row Transitions**: Number of transitions between filled/empty cells in each row
4. **Column Transitions**: Number of transitions between filled/empty cells in each column
5. **Cumulative Wells**: Sum of well depths, where wells are deep vertical gaps
6. **Eroded Cells**: Holes in cleared lines, serving as a reward signal

Note: The feature vector contains only these 6 board-related features. The piece-related information are **not encoded in the feature vector** but are available in the environment state and used during action generation and state transitions.

2.3.2 V-Learning State Representation

We adopt a two-branch state encoding:

- **Grid branch**: a binary 20×10 tensor representing the board occupancy, used as CNN input to capture local spatial patterns.
- **Feature branch (1D, 63 dims)**: 10 column heights, 10 hole counts, 1 binary hold-allowed flag, and 7×6 one-hot vectors indicating the current piece, hold piece, and the next four pieces. When generating s', the fourth next piece is unknown and set to zeros.

This hybrid design allows the network to combine low-level board geometry with high-level piece information.

2.4 Action Space

The action space is **dynamic**, depending on the current tetromino type and board state:

2.4.1 CEM Action Space

- **Direct placement**: (position, rotation) pairs
- **position**: Horizontal position (0 to width-piece_width)
- **rotation**: Rotation count (0, 1, 2, 3)
- **Implementation**: One-step placement without intermediate actions

2.4.2 V-Learning Action Space

Similar to CEM, each action corresponds to a **final placement**:

- All legal $(\text{rotation}, \text{column}, \text{hold} \in \{0, 1\})$ combinations for the current piece.
- Each action places the piece directly into its final position (or swaps with hold then places), skipping intermediate moves.

2.5 Transition Dynamics

Both CEM and V-Learning operate on the same fundamental principle: evaluating all possible final board states after placing the current piece. The key difference lies in how they find the best final board state.

CEM Transition Dynamics

For each decision step:

1. Enumerate all legal $(\text{rotation}, \text{column})$ placements of the current tetromino.
2. For each placement, simulate an **instant drop** of the piece to its final landing position, applying line clears if possible, to obtain the next state s'.

3. Extract the engineered feature vector of s' (e.g., holes, bumpiness, well depths).
4. Compute the placement score $f(s') = w^\top \phi(s')$ using the current weight vector w.
5. Select the action a that maximizes $f(s')$; with a small probability, choose a random legal placement for exploration.

V-Learning Transition Dynamics

For each candidate placement:

1. Simulate landing and line clearing in one step, producing next state s'.
2. Compute immediate reward $r(s \to s')$ from the number of cleared lines.
3. If s' is terminal, set target $y = \gamma\, r(s \to s')$; otherwise set $y = \gamma\,[\, r(s \to s') + V(s')\,]$.
4. Select the action a with $\arg\max_{s' \in \mathcal{N}(s)} \big(r(s \to s') + V(s')\big)$, with ϵ-greedy exploration.

CEM: Uses handcrafted linear features to score all final placements and selects greedily (*no bootstrapping*); **V-Learning**: Also enumerates final placements, but evaluates using $r + \gamma V(s')$ (*with bootstrapping*) via a CNN+MLP value approximator.

2.6 Reward Function

CEM Reward Function: $+1$ for each cleared line, 0 if no lines are cleared, **No explicit game over penalty** - only line clearing rewards.

V-Learning Reward Function: $+1$ for each cleared line, -500 penalty when the game ends. Discount factor $\gamma = 0.95$.

3 Problem Classification

We classify the above MDP along standard RL dimensions (Table 1), with brief justifications aligned to the formulation and prior analyses.

Table 1: Tetris MDP classification and rationale.

Dimension	**Classification**	**Rationale**
Episodic vs. Continuing	Episodic	Each game ends when the stack reaches the top; episode length is variable but finite.
Deterministic vs. Stochastic	Stochastic	Board update is deterministic given an action; but next-piece generation is random and materially impacts returns.
State space	Finite (very large)	Raw grid has 2^{200} configurations.
Action space	Dynamic discrete	All legal (position, rotation) pairs for the current piece; cardinality varies by piece and surface profile.
Observability	Fully observable	Agent observes full board, current piece, and next piece (no hidden state).
Reward structure	Sparse	Rewards only on line clears plus terminal penalty; no complicating credit assignment.

Implications: Tetris has a vast yet structured state space, dynamic discrete actions, stochastic inputs (next piece), and sparse rewards. This makes it better to use function approximation and look-ahead evaluation, and feature engineering and search augmentation can improve performance.

4 Solution Methods

We implement and compare two distinct reinforcement learning approaches for Tetris: the Cross-Entropy Method (CEM) and V-Learning. Each method represents a different paradigm in RL and offers unique advantages for this domain.

4.1 Cross-Entropy Method (CEM)

4.1.1 Algorithm Overview

Following Szita & Lőrincz (2006) [4], we implement the Cross-Entropy Method for Tetris, a policy search method that learns optimal weights for the Dellacherie feature set [3].

Methodological Foundation: Our approach combines handcrafted feature engineering with automated weight optimization. We use the proven 6-feature Dellacherie strategy that achieved 630,000 lines through manual tuning, then apply CEM to automatically learn optimal feature weights instead of manual tuning.

Feature Set Selection: While Szita & Lőrincz used 22 features, we chose the 6-feature set for computational efficiency, proven effectiveness, interpretability, and direct comparison with known optimal handcrafted weights.

4.1.2 Weight Learning

The algorithm learns weights for the linear evaluation function:

$$\begin{aligned} score(s) = w_1 \times holes + w_2 \times landing\_height + w_3 \times row\_transitions \\ + w_4 \times column\_transitions + w_5 \times wells + w_6 \times eroded\_cells \end{aligned} \tag{1}$$

where w_i are learned weights. We compare these with the handcrafted baseline:

$$\mathbf{w}_{handcrafted} = [-4.0, -1.0, -1.0, -1.0, -1.0, +1.0] \tag{2}$$

to assess learning effectiveness and optimization quality.

4.1.3 Core Principle

CEM maintains a probability distribution over the solution space and iteratively updates it based on elite performance. At each iteration, the algorithm:

1. Samples candidate solutions from current distribution
2. Evaluates their performance using the policy
3. Selects the elite subset based on fitness ranking
4. Updates distribution parameters to concentrate around elite solutions

This process gradually shifts the distribution toward high-performance regions.

Distribution Updates: We use statistical updates rather than gradient methods. The new mean and variance are computed directly from elite samples, with controlled noise injection to prevent premature convergence and maintain exploration.

4.1.4 Algorithm Pseudocode

Algorithm 1 Cross-Entropy Method

Initialize weight distribution $\mathcal{N}(\mathbf{0}, \sigma_0^2\mathbf{I})$ for 6 features
for generation = 1 to max_generations **do**
    Sample weight vectors from current distribution $\mathcal{N}(\boldsymbol{\mu}_{g-1}, \boldsymbol{\sigma}_{g-1}^2\mathbf{I})$
    for each weight vector $\mathbf{w}_i$ **do**
        Evaluate performance using policy $\pi(s) = \arg\max_a score(s, a, \mathbf{w}_i)$
        Record fitness score f_i
    end for
    Select elite subset based on fitness ranking
    Update distribution: $\boldsymbol{\mu}_g = \text{mean(elite)}$, $\boldsymbol{\sigma}_g = \sqrt{\text{var(elite)} + Z_g}$
    if convergence criteria met **then**
        break
    end if
end for

4.1.5 Mathematical Foundation

Following Szita & Lőrincz (2006), we implement the exact distribution update equations:

Mean Update:

$$\mu_{t+1} = \frac{\sum_{i \in I} w_i}{|I|} \tag{3}$$

Variance Update with Noise:

$$\sigma_{t+1}^2 = \frac{\sum_{i \in I} (w_i - \mu_{t+1})^2}{|I|} + Z_{t+1} \tag{4}$$

where σ_{t+1}^2 is the variance vector at iteration $t+1$, $(w_i - \mu_{t+1})$ represents the deviation of each elite weight from the new mean, Z denotes the noise, and $|I|$ is the number of elite samples ($|I| = \rho \cdot n = 10$

in this example).

4.2 V-Learning

4.2.1 Algorithm Overview

Following Rex L's implementation [13], V-Learning is a value function approximation method tailored for Tetris, designed to avoid the inefficiency of learning trivial movement actions. The method separates decision-making into two conceptual levels:

- **Learning level**: Chooses the best final board configuration (placement) by evaluating $V(s')$ over all legal placements of the current piece.
- **Execution level**: Derives the action sequence (e.g., rotate, shift, drop) required to reach the chosen placement from the current state.

The key modifications compared to standard Q-learning are:

- **Eliminating low-level action learning**: The environment enumerates all valid $(\text{rotation}, \text{column})$ placements, avoiding the need to learn primitive movement policies.
- **Direct state evaluation**: The agent selects the placement with the maximum $r + \gamma V(s')$ without explicitly representing $Q(s, a)$.
- **Reduced training complexity**: Dataset size is reduced to roughly $1/7$ of that for primitive-action Q-learning, where 7 is the average number of primitive steps per placement.

This approach significantly accelerates learning and stabilizes training, since the agent focuses only on the strategic aspect — *where* to place pieces — and not *how* to move them.

Implementation Source: Our implementation closely follows Rex L's open-source code, with minor adaptations for environment compatibility and additional logging for evaluation.

4.2.2 Algorithm Pseudocode

Algorithm 2 V-Learning Implementation

Initialize neural network V_θ randomly
for outer_iteration $= 1$ to max_outer_iterations **do**
    Collect samples (s, r, s') using current policy: $\pi(s) = \arg\max_{s' \in \mathcal{P}(s)} [r(s \to s') + \gamma V_\theta(s')]$, where $\mathcal{P}(s)$ is the set of all legal placements.
    Append samples to replay buffer
    for inner_iteration $= 1$ to K **do**
        Sample a mini-batch from the replay buffer
        Compute targets: $y = \gamma[r + V_\theta(s')]$ for non-terminal states, $y = \gamma r$ for terminal states
        Update V_θ by minimizing $\mathcal{L} = \|V_\theta(s) - y\|^2$
    end for
end for

4.2.3 Key Features

- **V-learning approach**: Uses V(s') evaluation instead of Q(s,a) for more efficient learning
- **CNN-based value network**: Uses convolutional layers to process the raw board grid, concatenated with engineered features (heights, holes, piece encodings).
- **Environment-assisted action set**: The environment generates legal placements, drastically reducing exploration noise.

4.3 Method Comparison

Table 2: Comparison of CEM and V-Learning Approaches

Aspect	**CEM**	**V-Learning**
Learning Paradigm	Policy search in **weight space** (linear evaluator)	Value function approximation in **state space** (CNN)
State Representation	6 Dellacherie-style board features	Raw grid (20×10) + engineered features (heights, holes, piece encodings)
Training Signal	Episodic returns from sampled rollouts	Temporal-Difference targets $r + \gamma V(s')$
Architecture	Fixed linear weights	Convolutional neural network
Search Scope	Enumerates placements, scores via linear feature weights	Enumerates placements, scores via learned $V(s')$
Advantages	Fast inference, low compute cost, interpretable weights	Reduces dataset size, focuses on strategic placement, avoids learning trivial moves
Limitations	Slow training, limited expressiveness; sensitive to hand-crafted features	Requires CNN training; performance depends on value network accuracy

5 Implementation and Results

5.1 Implementation Overview

Our implementation follows a systematic approach with both methods implemented in Python, using the same Tetris environment for fair comparison.

CEM is realized as an evolutionary policy search over linear weights on Dellacherie-style features, while V-Learning is a value-based method that enumerates legal final placements and approximates $V(s)$ with a CNN.

5.2 General Implementation Flow

1. **Environment setup**: Implemented Tetris game engine with both methods
2. **Algorithm implementation**: CEM using evolutionary search, V-Learning using neural networks
3. **Training process**: Systematic training with progress monitoring
4. **Evaluation**: Comprehensive testing with multiple metrics

5.3 Cross-Entropy Method (CEM)

5.3.1 Algorithm Implementation

We implemented a parallel Tetris AI training system based on the Cross-Entropy Method (CEM). The core architecture includes:

- **Parallel CEM Architecture**: Utilizing multi-process parallel evaluation of individuals, significantly improving training efficiency
- **Feature Extractor**: Implementing Pierre Dellacherie's 6 classic features
- **Action Evaluator**: State evaluation and action selection strategy based on feature weights

The training algorithm adopts the following key parameters:

- **Population size**: 100 individuals
- **Elite selection**: Top 10% (10 individuals) used for distribution updates
- **Evaluation stability**: 30 games per weight vector for reliable performance assessment
- **Noise strategy**: Implements $Z_g = \max(5 - g/10, 0)$ to prevent premature convergence
- **Adaptive stopping**: Training stops when reaching convergence threshold (1000+ lines) or after 20 generations without improvement
- **Parallel processing**: 8 parallel processes for efficient population evaluation

Training budget & stopping. Although CEM typically continues improving for tens of generations in prior work, its wall-clock cost grows quickly because each generation requires evaluating many full episodes. In our runs, we stopped after **4 generations** due to runtime constraints. For context, the original noisy cross-entropy study for Tetris reported more than a month of CPU time on a 1 GHz machine using Matlab to reach peak performance [4].

5.3.2 Training Results

Results. Figure 1 shows the learning process: the best score reached **400.1** lines (Generation 4), while the mean improved to **179.5** lines. This indicates CEM can discover effective weightings of the six features, but remains below our V-Learning agent under the same environment and budget.

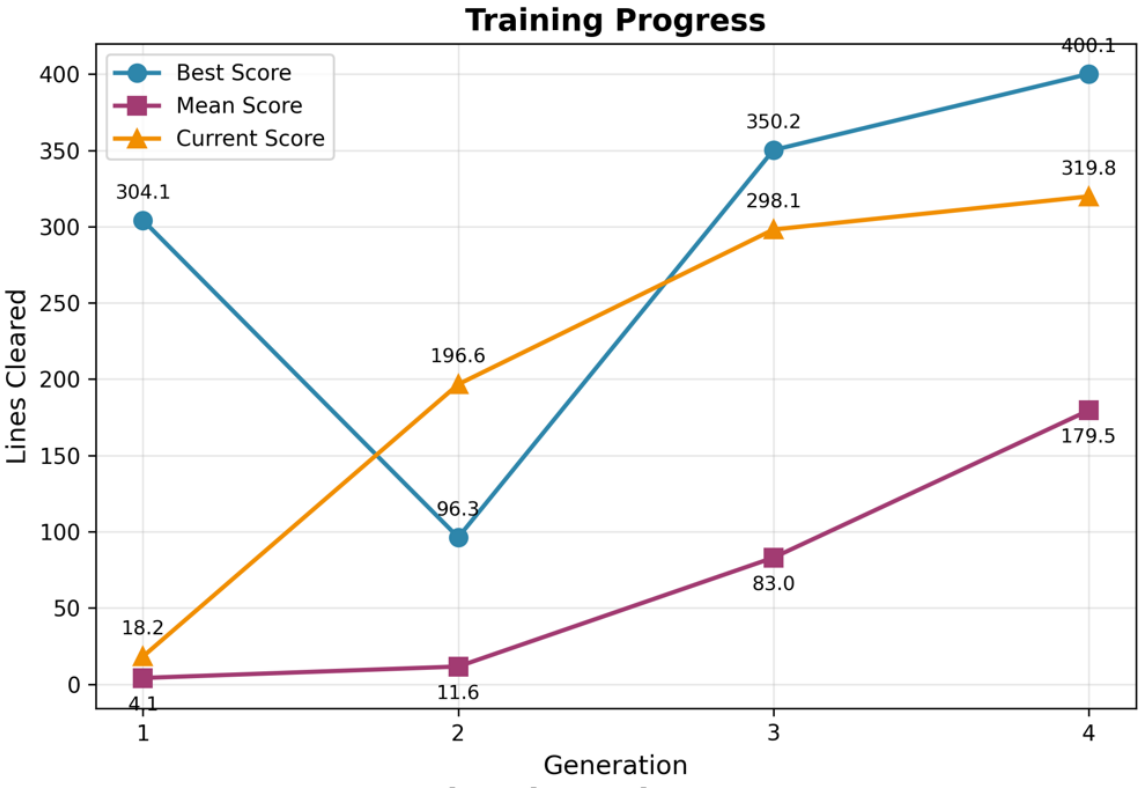

Figure 1: CEM training progress

It is worth noting that the "current" metric—defined as the average performance over 30 games using the weights learned in the given generation—provides a fair estimate of the deployable policy rather than isolated best cases. The steady improvement of this metric across four generations (from 18.2 to 1319.8 lines) demonstrates that the CEM algorithm is indeed learning effectively, despite the limited number of training generations.

To establish a performance baseline, we evaluated Pierre Dellacherie's hand-tuned weights within our

framework. Over 100 games, the handcrafted weights achieved an average of 157.1 lines with a best of 420 lines. In comparison, our 4-generation CEM training achieved a similar level (179.5 average, 400.1 best), demonstrating that the algorithm is indeed effective in recovering strategies comparable to human-designed heuristics. This validates the correctness of our CEM implementation, even though the absolute numbers are far from the original report of 630,000 lines due to differences in environment and optimization details. More importantly, this baseline highlights the clear advantage of V-Learning, which surpasses both handcrafted and CEM-trained weights by a large margin within only five outer iterations.

5.4 V-Learning

5.4.1 Implementation Details

Our implementation follows Rex L's design [13] with two input branches and a scalar value head:

- **Grid branch**: binary $20 \times 10 \times 1$ board fed to two CNN branches:
  – Conv2D (4×4, 128 filters) $\rightarrow$ max pooling $\rightarrow$ avg pooling
  – Conv2D (6×6, 64 filters) $\rightarrow$ max pooling $\rightarrow$ avg pooling
- **Feature branch (1D, 63 dims)**: 10 column heights, 10 hole indicators/counts, 1 hold-allowed flag, and 7×6 one-hot vectors for current/hold/next1..4 pieces.
- **Fusion & head**: concatenate [pooled CNN features, 1D features] $\rightarrow$ Dense 128 $\rightarrow$ Dense 64 $\rightarrow$ Dense 1.
- **Output**: a single scalar $V(s)$.

5.4.2 Training Parameters

Unless otherwise noted, we adopt the original settings:

- **Replay buffer**: 50,000 transitions
- **Inner loop**: 5 gradient steps per outer iteration
- **Epochs**: 5 per inner update

- **Batch size**: 512
- **Optimizer**: Adam (learning rate 0.001)
- **Loss**: Huber loss; **Metric**: mean squared error
- **Discount**: $\gamma = 0.95$ **Exploration**: $\varepsilon \approx 0.05$ (epsilon-greedy)

5.4.3 V-Learning Training Results

As shown in Figure 2, the agent surpassed CEM within **5 outer iterations**, reaching an average of **792.0** lines and a best of **1470.1**.

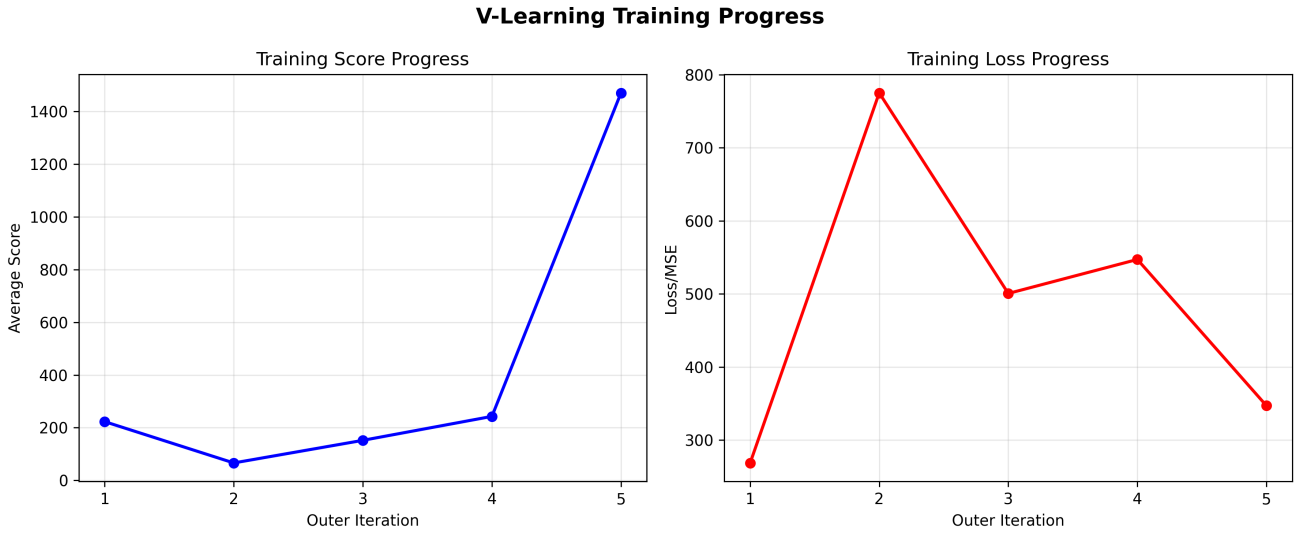

Figure 2: V-Learning training progress.

5.5 Comparative Results

Table 3 summarizes the latest results. V-Learning achieves substantially higher average and best scores within a modest number of outer iterations, while CEM shows steady gains but is far more sample- and time-intensive per generation.

Table 3: Performance comparison of CEM and V-Learning (updated).

Metric	**CEM**	**V-Learning**
Avg. lines cleared	319.8	792.0
Best lines cleared	400.1	1470.1
Generations / outer iters	4	5
Convergence pattern	Steady; needs many gens to peak [4]	Rapid gains with mild instability
Policy behavior	Greedy on linear features (survival-oriented)	CNN-guided; effective hold and T-spin usage
Training time	205.6 min	29.4 min

Training Time Comparison

For CEM, the latest implementation required 205.6 minutes ($\sim$3.4 hours) to finish 4 generations, with per-generation times increasing superlinearly: 8.5, 21.0, 51.8, and 124.2 minutes respectively, corresponding to a $\sim$2.4$\times$ growth factor. This exponential growth makes long-run training prohibitively expensive (the original study [4] reported one month of CPU time on a 1 GHz machine). By contrast, V-Learning reached its peak performance within 5 outer iterations in just 29.4 minutes, highlighting its much higher time efficiency.

CEM represents a class of *black-box policy search* methods. While such approaches are simple and gradient-free, they are inherently sample-inefficient because they require evaluating a large number of episodes per update. This limitation is amplified in Tetris due to its long episodes and sparse rewards, which makes value-based methods such as V-Learning more practical in comparison.

5.6 Learned Policy Behavior

The two methods exhibit distinct gameplay patterns:

- **CEM**: Minimizes holes and surface irregularities; plays conservatively for survival with limited long-horizon shaping.
- **V-Learning**: Learns advanced tactics such as T-spins and strategic hold usage; maintains smooth

surfaces and future flexibility, yielding much higher average and peak performance.

5.7 On Convergence in Tetris RL

Unlike small RL benchmarks, Tetris has a vast state space, sparse rewards, and no fixed horizon; learning curves rarely "flatten." In practice, we evaluate *trend and sample-efficiency* rather than strict convergence. CEM often improves over many generations but is expensive per generation (population $\times$ episodes), whereas V-Learning exploits representation learning and bootstrapping to achieve *clear upward trends* in a few outer iterations. Under equal time budgets, the value-based approach proved more effective in our setting, while CEM could continue to improve given substantially larger compute as reported in prior work [4].

6 Conclusion

This project studied two reinforcement learning approaches for Tetris: the Cross-Entropy Method (CEM) and V-Learning. We formalized the game as a Markov Decision Process and implemented both algorithms under the same experimental framework for fair comparison. Our results show that CEM can steadily improve through evolutionary search, achieving performance comparable to hand-crafted feature weights, but its training process is slow and sample-inefficient. In contrast, V-Learning demonstrates faster convergence, higher performance, and the ability to capture advanced strategies such as board management and T-spins, highlighting the advantage of value function approximation with neural networks.

Overall, the study illustrates the trade-offs between policy search and value-based methods in sequential decision-making problems. CEM provides a simple and interpretable baseline but struggles with scalability, while V-Learning achieves superior performance with moderate computational resources.

7 Future Work

While our study demonstrates the feasibility of applying CEM and V-Learning to Tetris, several directions remain open for improvement and extension:

- **Algorithmic extensions**: CEM suffers from poor sample efficiency and long training times due to its reliance on full-episode evaluations. Future work could explore more efficient methods such as policy gradient, actor-critic (e.g., PPO, A3C), or hybrid strategies where CEM provides initial weights that are later fine-tuned by temporal-difference learning.

- **Reward shaping and curriculum design**: Our experiments used a simple reward function (lines cleared and game-over penalty). Incorporating richer signals such as T-spins, combos, or well depth could lead to more sophisticated strategies. Curriculum learning, such as training first on reduced board sizes before scaling to the full game, could further improve stability and convergence.

- **Representation learning and generalization**: V-Learning combined engineered features with CNN-based state evaluation. Future work could study fully end-to-end architectures (CNNs or Transformers) and investigate generalization across different Tetris variants, piece distributions, or board configurations.

These directions would not only improve performance in Tetris but also provide broader insights into designing reinforcement learning agents for large-scale sequential decision-making problems.

References

[1] D. Bertsekas and J. Tsitsiklis, *Neuro-Dynamic Programming*, Belmont, MA: Athena Scientific, 1996.

[2] M. G. Lagoudakis and R. Parr, "Least-Squares Policy Iteration," in *Advances in Neural Information Processing Systems*, vol. 14, 2002, pp. 1547–1554.

[3] C. Fahey, "Tetris AI," June 2003. [Online]. Available: `http://www.colinfahey.com/tetris/tetris.html`

[4] I. Szita and A. Lörincz, "Learning Tetris using the Noisy Cross-Entropy Method," *Neural Computation*, vol. 18, no. 12, pp. 2936–2941, 2006.

[5] C. Thiery and B. Scherrer, "Improvements on Learning Tetris with Cross Entropy," *International Computer Games Association Journal*, vol. 32, no. 1, pp. 23-33, 2009.

[6] V. Gabillon, M. Ghavamzadeh, and B. Scherrer, ”Approximate Dynamic Programming Finally Performs Well in the Game of Tetris,” in *Advances in Neural Information Processing Systems 26 (NIPS)*, 2013, pp. 1754-1762.

[7] M. Stevens and S. Pradhan, ”Playing Tetris with Deep Reinforcement Learning,” Stanford CS231n Course Project Report, 2016. [Online]. Available: https://cs231n.stanford.edu/reports/2016/pdfs/121_Report.pdf

[8] S. Algorta and Ö. Şimşek, ”The Game of Tetris in Machine Learning,” arXiv preprint arXiv:1905.01652, 2019.

[9] H. Liu and L. Liu, ”Learn to Play Tetris with Deep Reinforcement Learning,” *OpenReview*, 2024. [Online]. Available: https://openreview.net/pdf?id=8TLyqLGQ7Tg

[10] N. Faria, *tetris-ai*, GitHub repository, 2017. [Online]. Available: https://github.com/nuno-faria/tetris-ai

[11] vietnh1009, *Tetris-deep-Q-learning-pytorch*, GitHub repository, 2019. [Online]. Available: https://github.com/vietnh1009/Tetris-deep-Q-learning-pytorch

[12] G. Cannon, *StackRabbit*, GitHub repository, 2024. [Online]. Available: https://github.com/GregoryCannon/StackRabbit

[13] Rex-L, ”Reinforcement Learning on Tetris,” *Medium*, 2018. [Online]. Available: https://rex-l.medium.com/reinforcement-learning-on-tetris-707f75716c37

[14] CodeBullet, ”I Created An A.I. to DESTROY Tetris,” *YouTube*, Apr. 2018. [Online]. Available: https://www.youtube.com/watch?v=QOJfyp0KMmM

[15] AskForGameTask, ”AI Plays Tetris with Convolutional Neural Network,” 2020. [Online]. Available: https://www.askforgametask.com/tutorial/machine-learning/ai-plays-tetris-with-cnn/

[16] G. Cannon, ”AI BREAKS NES TETRIS! - 102 MILLION and level 237,” *YouTube*, 2021. [Online]. Available: https://www.youtube.com/watch?v=l_KY_EwZEVA

A Project Repository

The full source code and experimental logs for this project are available on GitHub: `https://github.com/rachel-wenhui-ma/RL_Tetris.git`