Designing a Tetris Controller with CEM

Cailin Winston, Peter Michael, Ronak Mehta
December 28, 2020

1 Introduction

Designing a high-performing Tetris player is a fundamental benchmark problem in artificial intelligence (AI), due to the the difficulty of the problem. First and foremost, as in all reinforcement learning settings, actions taken by the agent affect the environment. The state space is large and discrete, and the reward signal does not lend itself to typical gradient-based optimization. Computationally, success in Tetris is inherently tied to executing a long game, which implies a training time that grows with performance. In this project, we collect hand-designed features from the literature to handle the unruliness of the state space, and use a variant of Cross Entropy Method optimization for our non-differentiable reward function. We adjust the method to combat some of the statistical and computational pitfalls that agents run into when using this algorithm. In doing so, we achieve a strong-performing Tetris player with a relatively simple parametrization. Section 2 organizes the literature on this problem from various perspectives. Section 3 details our overall problem formulation, algorithms and hyperparameters, and hardware on which we trained. Section 4 highlights and analyzes our main results, while Section 5 reviews the work and discusses possible extensions.

2 Related Work

Approaches to designing Tetris controllers can broadly be characterized by how state features are designed, how policies are parametrized, and how the parameters of the policy are optimized. While the most raw representation of the standard 20×10 Tetris board is a binary matrix in $\{0,1\}^{20\times 10}$ that indicates filled and unfilled squares, we can provide inductive bias by choosing a hand-designed feature function $f : \{0,1\}^{20\times 10} \times \mathbb{A} \to \mathbb{R}^d$ that summarize salient aspects of the board to improve learning (where $\mathbb{A}$ is the action space) (15). These features can represent the status of the game, such as landing height of the next piece or number of holes, which are invariant to the size of the board (15), or can be board-specific, which scale with the size of the board (2). (3) presents a review of about 20 features in each category that are used throughout the literature. Given, the feature vector, most approaches evaluate the feature vector using a linear function $\theta^\top f(x,a)$ which determines the fitness of an action $a \in \mathbb{A}$ for state x. Langenhoven et al. (10) use a neural network $h_\theta : \mathbb{R}^d \to \mathbb{R}$ to parametrize this fitness function. Some approaches construct feature maps whose range is small enough the value iteration and policy iteration-type dynamic programming algorithms can be applied in this reduced feature space (16; 2; 6; 4; 9), but these approaches have not remained competitive.

While feature design and policy parametrization have seen many standard themes, the choice of optimization algorithm is an active area of research and experimentation. Because the reward function for this problem (the number of lines cleared) is not in general continuous (much less differentiable), gradient-based optimization is typically out of the question. Thus, approaches fall into three major categories: evolutionary and genetic algorithms (1; 8; 11; 7), particle swarm

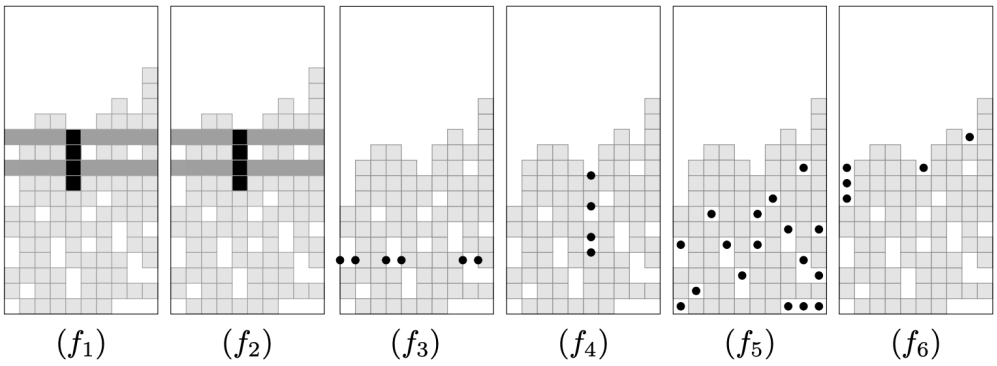


Figure 1: Visual depiction of feature maps $f = (f_1, f_2, ..., f_6)$. Source: Boumaza (3).

optimization (PSO) (10), and a noisy cross entropy (NCE) method (12; 13), which we adapt for our procedure.

3 Approach

3.1 Reward Function

Our chosen reward function $r : \mathbb{X} \times \mathbb{A} \to \mathbb{R}$ consumes the current state $x \in \mathbb{X}$ and action $a \in \mathbb{A}$ and produces the number of lines cleared by virtue of a. Thus, our total reward over an episode $\tau = (x_1, a_1), ..., (x_T, a_T)$ satisfies

$$\sum_{t=1}^{T} r(x_t, a_t) = \text{ total lines cleared in episode } \tau,$$

where T is given by the number of actions until the game terminates. The state space is large and highly-structured (see Section 3.2), so we apply a parametrized policy $\pi_\theta : \mathbb{X} \to \mathbb{A}$. The total reward with respect to this θ is the expected sum over all possibly episodes when applying this policy. That is,

$$J(\theta) = \mathbb{E}\left[\sum_{t=1}^{T} r(x_t, \pi_\theta(x_t))\right].$$

Note that T is also random in the expression. This reward is a direct measure of performance in the Tetris game, but is not differentiable and involves an average over an combinatorially-many number of possible trajectories. As a result, we took a black-box optimization approach to find a high-performing θ.

3.2 State, Action, and Policy Representation

For our state representation, we choose $d = 6$ features from (5), which until 2008, produced the best performing artificial Tetris player (3). The six features are given by:

- f_1 = **Landing height:** the height at which the current piece fell (8 in Figure 1).

- f_2 = **Eroded pieces:** the contribution of the last piece to the cleared lines time the number of cleared lines (2×2 in Figure 1).
- f_3 = **Row transitions:** number of filled cells adjacent to empty cells summed over all rows (58 in Figure 1).
- f_4 = **Column transition:** same as f_3, summed over all columns note that borders count as filled cells (45 in Figure 1).
- f_5 = **Number of holes:** the number of empty cells with at least one filled cell above.
- f_6 = **Cumulative wells:** the sum of the accumulated depths of the wells ((1 + 2 + 3) + 1 + 1 on in Figure 1).

The first two features depend on the action a, while the rest only depend on the state. Because of the difficulty of black box optimization, we chose a representation that had a few number of interpretable parameters that were well-justified in the literature. Note that the physics of the falling piece is not included in the environment, although evaluating an action is fast enough that it usually does not make a noticeable impact (3).

The actions are defined by an orientation and column placement of the current piece, although the number of legal actions given a state can be fewer than that (depending on placement of the board). We can let $\mathbb{A} = \{\texttt{UP}, \texttt{DOWN}, \texttt{LEFT}, \texttt{RIGHT}, \} \times \{0, 1, 2, ..., 9\}$, and have $r(x, a) = -\infty$ for any illegal action.

Finally, the policy maximizes a linear function of the state vector, the parameters of which is chosen by the optimization algorithm to optimize $J(\theta)$.

$$\pi_\theta(x) = \arg\max_{a \in \mathbb{A}} \ \theta^\top f(x, a) = \arg\max_{a \in \mathbb{A}} \ \sum_{j=1}^{6} \theta_j f_j(x, a)$$

3.3 Optimization Algorithm

We chose to use the cross-entropy method (CEM), a black-box policy optimization method, to directly optimize the objective $J(\theta)$. Using a gradient-free approach here is beneficial because of the non-differentiability of our cost function. CEM is an algorithm in which a distribution over weights θ is learned iteratively. At each iteration, we sample a set of weights from the current distribution, evaluate them, and select an elite set of high performing weights from which to re-estimate the parameters of the same distribution. This iteratively "collapses" the distribution on a region of the parameter space $\mathbb{R}^d$ with high values of $j(\theta)$.

The namesake of the method can be seen by considering the following setup. Let q_{ϕ_t} be the distribution of the parameters θ, parametrized by iterate ϕ_t. We can set some threshold $\gamma \in \mathbb{R}$, and let

$$\phi_{t+1} = \min_{\phi} H(q_{\phi_t}(\cdot \mid J(\theta) > \gamma), q_\phi)$$

$q_{\phi_t}(\cdot \mid J(\theta) > \gamma)$ is the distribution of θ parametrized by the previous iterate, but conditioned on having "good" performance. We can estimate this distribution from samples of θ that achieve high values of $J(\theta)$, after which cross entropy minimization becomes maximum likelihood estimation of

the parameters ϕ. Rather than setting a hyperparameter γ, we instead take the best performing values of θ from which to estimate this empirical distribution. To be more concrete, we can let $\phi = (\mu, \Sigma) \in \mathbb{R}^d \times \mathbb{R}^{d \times d}$ and q_ϕ be the Gaussian distribution.

This method is known to have a few common pitfalls. The iterates can converge quickly to a suboptimal solution (12), and the large number of θ samples can be computationally taxing. In the case of Tetris, $J(\theta)$ must be estimated for particular θ by rolling out many episodes of Tetris games. When the value of $J(\theta)$ becomes large, then episodes take longer, as the reward function is proportional to how long a player lasts in the game. If the data estimates an approximately low-rank Σ, then future iterates will continue to lie on the subspace spanned by the principle eigenvectors on Σ, instead of searching all directions of $\mathbb{R}^d$. To ensure that the algorithm searches expansively, (12) adds a noise term to the estimate covariance matrix, given by

$$\Sigma := \Sigma + \max\left\{\left(5 - \frac{t}{10}\right), 0\right\} \cdot I_d$$

This also improves the conditioning of the estimated covariance matrix.

We adaptively change the number of players (number of θ samples) and the number of episodes used to estimate $J(\theta)$ as a function of the average number of lines cleared by the elite set of players. We initialize the number of players to be large in order to explore the parameter space more. As the trace of the covariance matrix decreases, indicating convergence, and as the average number of lines cleared by the elite set increases, we increase the number of episodes used to evaluate each player, to be more statistically confident about the ranking of players. In order to balance out the increasing number of episodes and in order to fit within a computational budget, we decrease the number of players over iterations. We applied this Reduced Noisy Cross Entropy method to the Tetris problem. The classic Noisy Cross Entropy outlined in detail in Algorithm 1, and our computational adjustments are outlined in Section 3.4.

Algorithm 1: Noisy Cross Entropy (NCE) Method

Input:
(μ_0, Σ_0): mean and variance of initial Gaussian distribution
n_players: number of weight vectors sampled from distribution at each iteration
n_episodes: number of episodes each player plays in order to evaluate each weight vector
evaluate(w): a function that runs *n_episodes* games using weights w
elite_percent: fraction of best-performing weight vectors selected at each iteration
reg_iters: number of iterations to regularize variance
λ: how much to scale the covariance regularizer
inter: controls how much interpolation between previous mean and variance
$it \longleftarrow 1$;
$elite\_players \longleftarrow []$;
$(\mu, \Sigma) \longleftarrow (\mu_0, \Sigma_0)$;
while *not converged* **do**
  Sample *n_players* weight vectors $w_1, w_2, ...w_{n\_players}$ from $\mathcal{N}(\mu, \Sigma)$;
  $rewards \longleftarrow []$;
  for $p = 0$ *to n_players* **do**
    $episodes\_rewards \longleftarrow []$;
    for $e = 0$ *to n_episodes* **do**
      $episodes\_rewards[e] = \text{evaluate}(w_p)$;
    end
    $rewards[p] = \text{mean}(episodes\_rewards)$;
  end
  $elite\_players \longleftarrow$ Select w_j for $elite\_percent \cdot n$ with highest evaluation;
  $\mu \longleftarrow (1 - inter) \cdot \mu + inter \cdot \text{mean}(elite\_players)$;
  $\Sigma \longleftarrow (1 - inter) \cdot \Sigma + inter \cdot (\text{Cov}(elite\_players) + \lambda \cdot \max\{reg\_iters - it, 0)\} \cdot I$;
  $it++$;
end

3.4 Training

We ran two phases of training, because after 50 iterations (number of iterations in each phase), the algorithm converged and regularization stopped.

In Phase I, we set the initial $\mu_0 = 0$ and $\Sigma_0 = 100 \cdot I$, to search as expansively as possible. We let $elite\_percent = 20\%$, but lower bounded the size of the elite set by the dimension of w to ensure positive definiteness of the estimated covariance matrix with probability 1. We let $reg\_iter = 50$ and $\lambda = \frac{1}{10}$, as in (12). In order to augment the method to fit into our computational budget, we adjusted *n_episodes* and *n_players* over the iterations. Specifically, we let *n_players* interpolate linearly between 40 and 15 and *n_episodes* interpolate linearly between 5 and 25. In Phase II, we set $\mu_0 = \mu$ and $\Sigma_0 = \Sigma + 5 \cdot I$, where (μ, Σ) were estimated from the previous phase. We let *n_players* interpolate linearly between 25 and 20 and *n_episodes* interpolate linearly between 20 and 25, over 50 iterations.

One issue that we faced in early trials was fast convergence to a local optima, as indicated by the trace of the covariance approaching 0. The hyperparameters above combatted this, especially

injecting a lot of variance into the parameter distribution (initializing at $\Sigma = 100 \cdot I$) in early stages of the optimization. Adding a linearly decreasing regularization term (12) that started at 5 worked well. Another strategy that we employed was carrying over the elite set into the next iteration. This helps to prevent oscillations and always ensures that the performance of the elite set doesn't decrease. To smoothen the change in the distribution, we interpolated the previous parameters with the new estimates for the current iteration.

We ran the algorithm on an Amazon Web Services (AWS) EC2 instance, of instance type `c5a.16xlarge` with 64 vCPUs and 131072 MiB RAM. The algorithm ran for about 48 hours. We parallelized the evaluations over both players and episodes. At each iteration we tracked the following metrics to assess performance:

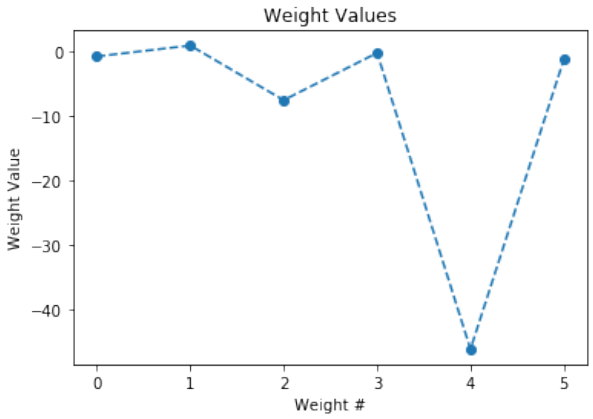
- Maximum lines cleared by any player (averaged over episodes).
- Average lines cleared by all players.
- Trace of the covariance matrix (to assess convergence)
- Condition number of the covariance matrix (to assess expansiveness of the optimization trajectory)

4 Results

After training for 85 iterations, we found that the following weights resulted in 12551 lines being cleared on average over 20 episodes, with a 95% confidence interval of (6161.31, 18940.68).

$$[-0.74239601, 0.95166667, -7.53798525, -0.19433414, -46.23768896, -1.17153955]$$

Below is a plot of our optimal weights.



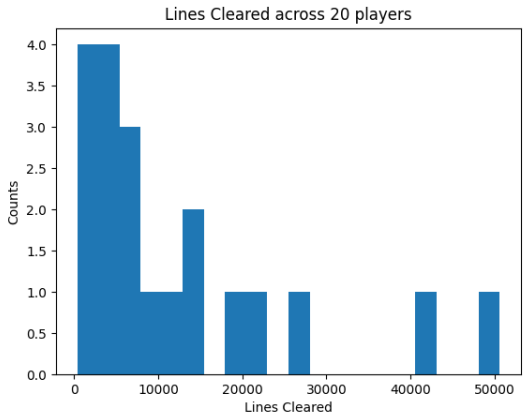
The signs on these weights make sense, when we consider what features of the game board they represent. w_0 is negative since we want a lower landing height, as that represents being in a better position in the game (further away from game end). w_1 is positive since we want to maximize the number of eroded pieces, as it represents the number of cleared lines times the contribution of the last piece to those cleared lines. w_2, w_3, w_4 are negative since we want to minimize the number of

holes (which is wasted space). w_5 is negative since we want to have a less number of wells (these are less easily accessible to pieces

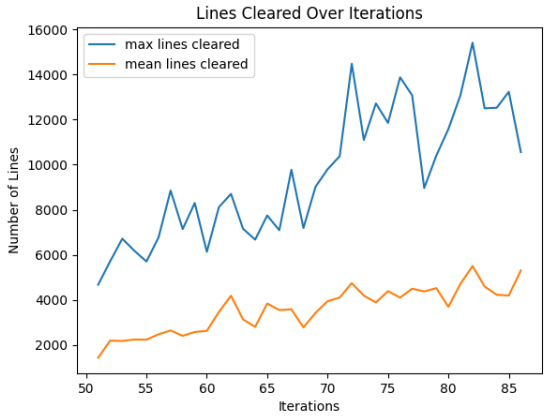
Looking at the magnitude of these weights, we see that w_4 has the most weight, indicating that the number of holes is the most important feature in determining a successful player.

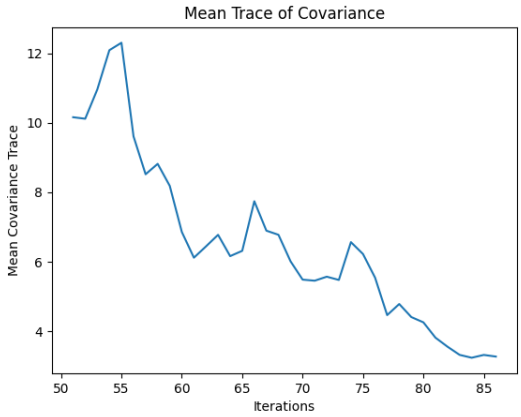
These signs and magnitudes of the weights are backed up by literature as well as (14) discussed a cost function with the weights $[-1, 1, -1, -1, -4, -1]$. Furthermore, we found that initializing our weights to this boosted the performance of our players in the first iteration, by inserting domain knowledge into the policy.

The following histogram shows the distribution of lines cleared.



The following plots show the metrics we tracked over the last iterations. As the mean trace of the covariance matrix decreased over iterations, indicating convergence, the maximum and average lines cleared by all players increased.





5 Discussion

In this project, we built a Tetris controller through feature design and black-box optimization. We parametrized the policy by using six hand-designed features from previous literature, with a linear function determining the ranking of proposed actions. The weights of the linear function were

learned by a version of the Cross Entropy Method, with adjustments made to avoid common pitfalls and fit within computational constraints. While we found this method to be appealing, especially due to lack of assumptions on the problem, we found it to be a difficult optimization algorithm to tune and execute, both from a computational and statistical perspective. Some extensions for the optimization include training for more phases, and implementing the "racing" ideas from (3) to find the minimum number of episodes necessary to produce reasonable rankings. Other extensions include using a larger/different feature set, or designing a proxy reward function that is differentiable and allows for gradient-based updates. Designing Tetris players remains a challenging and instructive benchmark for the progress of AI.

References

[1] P. J. Angeline and K. E. Kinnear. *Genetically Optimizing The Speed of Programs Evolved to Play Tetris*, pages 279–298. 1996.

[2] Dimitri Bertsekas and John Tsitsiklis. *Neuro-Dynamic Programming*, volume 27. 01 1996. doi: 10.1007/978-0-387-74759-0_440.

[3] Amine Boumaza. How to design good tetris players. 01 2013.

[4] G. Calafiore and Fabrizio Dabbene. *Probabilistic and Randomized Methods for Design under Uncertainty*, volume 49. 01 2007. doi: 10.1007/b138725.

[5] C. P. Fahey. Tetris ai, computer plays tetris. http://colinfahey.com/tetris/tetris_en.html, 2003.

[6] Sham Kakade. A natural policy gradient. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, page 1531–1538, Cambridge, MA, USA, 2001. MIT Press.

[7] G. Kókai and S. Mandl. An evolutionary approach to tetris niko böhm. 2005.

[8] J. Koza. Genetic programming - on the programming of computers by means of natural selection. In *Complex adaptive systems*, 1993.

[9] Michail Lagoudakis, Ronald Parr, and Michael Littman. Least-squares methods in reinforcement learning for control. volume 2308, 03 2002. ISBN 978-3-540-43472-6. doi: 10.1007/3-540-46014-4_23.

[10] L. Langenhoven, W. S. van Heerden, and A. P. Engelbrecht. Swarm tetris: Applying particle swarm optimization to tetris. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010. doi: 10.1109/CEC.2010.5586033.

[11] R. E. Llima. Xtris. http://www.iagora.com/~espel/xtris/README, 2005.

[12] I. Szita and A. Lörincz. Learning tetris using the noisy cross-entropy method. *Neural Computation*, 18(12):2936–2941, 2006. doi: 10.1162/neco.2006.18.12.2936.

[13] Christophe Thiery and Bruno Scherrer. Construction d'un joueur artificiel pour tetris. *Revue d'intelligence artificielle*, 23, 05 2009. doi: 10.3166/ria.23.387-407.

[14] Christophe Thiery and Bruno Scherrer. Improvements on learning tetris with cross entropy. *International Computer Games Association Journal*, 32, 2009. doi: inria-00418930.

[15] Christophe Thiery and Bruno Scherrer. Building controllers for tetris. *ICGA journal*, 32, 03 2009. doi: 10.3233/ICG-2009-32102.

[16] J. N. Tsitsiklis and B. Van Roy. Feature-based methods for large scale dynamic programming. In *Proceedings of 1995 34th IEEE Conference on Decision and Control*, volume 1, pages 565–567 vol.1, 1995. doi: 10.1109/CDC.1995.478954.