Building a Tetris-Playing Learner

Sandeep Reddy Baddam
Department of Mechanical Engineering
sbaddam@uw.edu

D J Rama Krishna
Department of Mechanical Engineering
djrk@uw.edu

Tyler Han
Department of Computer Science and Engineering
than123@uw.edu

1 Introduction

In the game of Tetris, Tetriminos fall down through a 21 x 10 grid one at a time, each time stacking upon any pieces which have already fallen completely. The player is allowed to rotate a piece, and move it to any of the allowable ten columns as it falls. Each game piece is picked randomly from a set of seven shapes and a new piece is picked each time the previous one has finished falling. The game ends if the height of the structure reaches the height of the board, but rows of the structure can be cleared if the entire row is composed of squares from fallen pieces. The objective of the game is to clear as many lines as possible before the game ends.

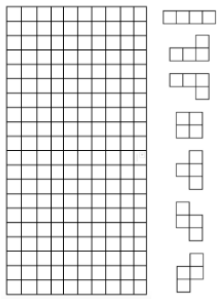

Figure 1: Layout of 21 x 10 Tetris and seven Tetriminos

In general, the state-space is large and discrete. Suppose we naively take the entirety of the Tetris game environment as the state of the Markov Decision Process we aim to solve. The board itself has 210 occupiable squares resulting in 2^{210} different states. As for the actions, each piece has about 4 orientations x 10 columns (there may be fewer depending on the piece) resulting in 40 potential actions to be taken at each round. The state-action space is thus non-differentiable and quite large at a size of about 40×2^{210}, making solving the game of Tetris to be considerably challenging.

2 Related Works

Due to the discrete nature of the game, gradient-free methods are often the primary approach to reward optimization [1]. This includes methods such as the Cross Entropy Method and CMA-ES.

These methods also perform better when reducing the state-space to a set of hand-crafted features [3] and learning a set of weights on these features through reinforcement learning episodes. These features capture higher-level properties of the board such as: (1) landing height (height at which the current piece fell), (2) piece contribution (number of lines cleared by the previous piece times the number of contributed squares), (3) row transitions, (4) column transitions, (5) number of holes, and (6) accumulated well-depth.

More recent work on Tetris has used deep learning methods. In [5], a convolutional layer is first applied over the entire game board, feeding into networks for column-wise and then row-wise feature extraction. While this approach does not improve performance over explicitly featurized agents, the formulation of the game also included the real Tetris dynamics (e.g. incorporating timing and piece-slotting).

3 Reward Function

We tried two types of reward functions. They are i) Lines cleared in current turn and ii) Fitness function. It came to a conclusion that the first type is very effective when using Dellacherie's hand-crafted features that capture good state representation (more in State Representation part). And, the second type uses an explicitly featurized fitness function [7]. This is useful when the state definition is naive, like taking only the field and next piece into consideration. Our finalized algorithm used the first type and showed some impressive results over iterations.

Here, every time the agent clears lines, the reward is the number of lines cleared in that turn and we would be adding this over the episode to attain the cumulative reward. Note that we are not using any discounted factor as the game focuses on the cumulative lines cleared over an episode rather than immediate early rewards. Later for each player, we would be computing the expectation of the cumulative lines cleared over some episodes and then comparing it with other players.

$$r(s, a) = \text{Number of lines cleared in current turn}$$

$$\sum_{i=0}^{end} r(s_i, a_i) = \text{Cumulative lines cleared over an episode}$$

We know policy parameters θ i.e., weights to find the best action given current state information. The main motivation is that this reward function helps us to track if the model has attained the prescribed goal (here, 10000 lines) directly without further requiring any other computation. It can save some time as the model improves. Also, we can do early-stopping by comparison.

4 State Representation

Our state representation will follow the Dellacherie features [3] which have a proven track record demonstrating their utility in reinforcement learning efforts in Tetris as discussed in the related work section. We use the six-feature formulation:

$$\text{Landing height} = top[action[`col']]$$

$$\text{Piece contribution} = previousTop[action[\text{`}col'\text{]}] + pieceHeight - top[action[\text{`}col'\text{]}]$$

$$\text{Row transitions} = \sum_{i \in rows \notin empty} transitions[i]$$

$$\text{Column transitions} = \sum_{i \in cols \notin empty} transitions[j]$$

$$\text{Number of holes} = \sum_{j \in cols} \sum_{i \in top[j]}^{0} (value[i,j] == 0)$$

$$\text{Well sum} = \sum_{j \in cols} Arithsum\{min(\mathbf{top[j-1]}(top[j+1] \text{ if j = 0}), \mathbf{top[j+1]}(top[j-1] \text{if j = cols-1})\}$$

5 Approach

We used a variant of the cross-entropy method i.e., noisy cross-entropy method [8] which is a gradient-free optimization technique. In this method, we have two repeating phases [9]. One, we draw a sample from a probability distribution. Two, minimize the cross-entropy between this distribution and a target distribution to produce a better sample in the next iteration. We implemented this technique to learn the weights for the six chosen Dellacherie's features. In RL literature, the cross-entropy method is also called black-box policy optimization as the algorithm tries to find the weights that map the states to best action. It is a linear model (weights multiplied by features) that gives value for each possible action given the state information. Then we choose the action that has maximum value. Here, the algorithm analyzes many players over a set of episodes and selects weights as the mean of elite samples for the next iteration. Elite samples are the ones that clear the maximum number of lines on an average over episodes. Because of this fact, it is not a greedy approach.

Cross − entropy without noise At iteration t, the weights are drawn from an independent multivariate gaussian distribution $N(\mu_t, \Sigma_t, n)$. The number of rows and columns in the weight matrix equals $\#players(n)$ and $\#features$ respectively. Let the best performing samples is given by I $\subseteq [0, 1, 2, \ldots, n-1]$, the mean is updated as

$$\mu_t = \frac{\sum_{i \in I} w_i}{|I|}$$

After some initial training experiments trying out CE without any noise to covariance, it showed that the learned weights reach sub-optimal values very early (also termed as local optima). This indicates that CE application to RL problems lead to the learned distribution concentrating to a single point too fast. To prevent this early convergence, [8] introduced a trick to add some extra noise(Z) to the distribution.

$$\Sigma_t = \Sigma_t + Z$$

Moreover to make the agent perform even more better, [8] mentions that decreasing noise turns out to be better than just adding constant noise as shown in figure 2(a).

$$\Sigma_t = \Sigma_t + Z_t$$

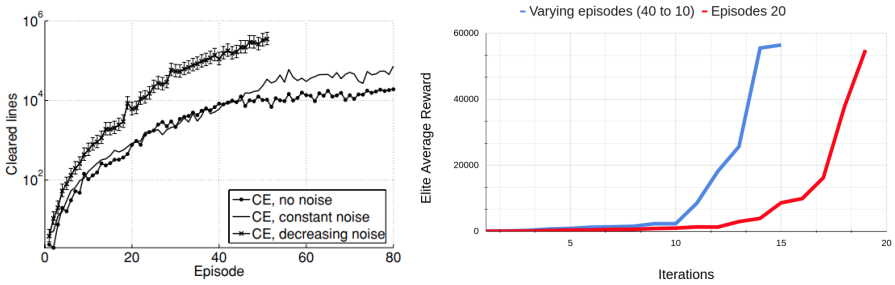

Figure 2: (a) Performance for different variants of cross-entropy: Source[8] (b) Performance of our model over iterations with two different techniques

where Z_t decreasing amount of noise, e

$$Z_t = max(5 - \frac{t}{10}, 0)$$

More information can be found in the pseudo-code at Appendix.

6 Evaluation

The agent has been trained for 26 hours of CPU time on a 3.50GHz × 24 machine. Incorporating a technique of varying episodes helped in reducing the training time as the model improved (figure 2(b)).

Result :

Maximum number of lines cleared among 20 episodes: $\mathbf{1,69,557}$
Average lines cleared over 20 episodes : $\mathbf{31,348}$

We did early-stopping after 13 iterations as the average reward crossed 30000 lines. For the given problem setup and the algorithm we have, the evaluation time of the action-value function scales linearly with the reward. There is a large scope of parallelization over all players and their episodes which can reduce both training and evaluation time drastically.

References

[1] Sutton RS, Barto AG. Reinforcement Learning: An Introduction. Cambridge, Mass: A Bradford Book; 1998. 322 p.
[2] C. Winston, P. Michael, and R. Mehta, “Designing a Tetris Controller with CEM,” p. 9.
[3] A. Boumaza, “How to design good Tetris players,” p. 20.
[4] C. P. Fahey. Tetris AI, Computer plays Tetris, 2003
[5] C. Thiery and B. Scherrer. Construction d’un joueur artificiel pour tetris. Revue d’Intelligence Artificielle, 23:387–407, 2009.
[6] M. Stevens and S. Pradhan, “Playing Tetris with Deep Reinforcement Learning,” p. 7.
[7] https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/
[8] I. Szita and A. L¨orincz. Learning tetris using the noisy cross-entropy method. Neural Computation, 18(12):2936–2941, 2006. doi: 10.1162/neco.2006.18.12.2936.
[9] https://en.wikipedia.org/wiki/Cross-entropy_method

Appendix

Algorithm 1 Decreasing Noise Cross-Entropy Method

Initialize parameters

$Current\ sampling\ distribution\ is\ Gaussian\ w \sim \mathcal{N}(\mu,\ \Sigma,\ batch\_size)$
$mean,\ \mu = 0$
$cov\_scale = 100$
$\Sigma\ \leftarrow cov\_scale * eye(feature\_count)$
$maxits = 100$
$iter\_count = 0$
$goal\_reward = 10000$
$average\_reward = 0$
$batch\ size \leftarrow number\ of\ players$
$episodes \leftarrow number\ of\ episodes\ each\ player\ plays$
$elite\ percentile \leftarrow percentile\ of\ players\ from\ batch\ that\ are\ considered\ as\ elite$
$retain\ percentile \leftarrow percentile\ of\ players\ from\ elite\ batch\ that\ are\ retained\ for\ next\ weight\ update\ step$
$elite\ retained \leftarrow [\,]$
$elite\ weights \leftarrow [\,]$
$elite\ rewards \leftarrow [\,]$

while $iter\_count < maxits$ **and** $average\_reward\ < goal\_reward$ **do**

// Obtain #batch_size weight samples from the current sampling distribution
$w \sim \mathcal{N}(\mu, \Sigma, batch\_size)$
$w = \mathbf{sample\_multivariate\_gaussian}(\mu,\ \Sigma,\ batch\_size)$
$w\ \leftarrow\ \mathbf{concat}(w,\ elite\_retained)$

// Evaluate objective function(find reward) at the sampled points
$reward \leftarrow \mathbf{find\_reward}(weights, episodes)$ // average reward for each player over episodes

// Sort weights by the reward function values in descending order
$w\ \leftarrow\ \mathbf{sort}(w,\ reward)$

// Take out elite weights and elite rewards
$elite\_weights\ \leftarrow\ w(0\ :\ elite\_percentile)$
$elite\_rewards\ \leftarrow\ sorted\_rewards(0\ :\ elite\_percentile)$

// Update parameters of sampling distribution
$\mu\ \leftarrow\ \mathbf{mean}(elite\_weights)$
$\Sigma\ \leftarrow\ \mathbf{cov}(elite\_weights)$

// Adding decreasing noise to covariance to avoid sub-optimal solution
$\Sigma\ +=\ \mathbf{decreasing\ noise}$

// Retain few elite samples for next update
$elite\_retained\ \leftarrow elite\_weights[0\ :\ retain\ percentile]$

// calculate average reward for elite samples
$average\_reward =\ \mathbf{mean}(elite\_rewards)$

$iter\_count+ = 1$

end while