Homework 3: Tetris

Andrew Wei
nowei@cs.washington.edu

Eric Chan
yee96@cs.washington.edu

Romain Camilleri
camilr@cs.washington.edu

1 Introduction

Tetris is a tile-matching video game where players must complete lines by moving differently shaped pieces (tetrominoes), which descend onto the playing field. The completed lines disappear and grant the player points, and the player can proceed to fill the vacated spaces. The game ends when a piece reaches the ceiling. Although the rules of Tetris are very simple, designing artificial players is a great challenge for the artificial intelligence community.

Here, the goal of the assignment is to design an artificial Tetris player, which improves over experience. Note that one simplification is made: the dynamics are ignored, such that the only interactions that the players have are through rotating and shifting the tetrominoes to place them in the columns of choice.

The repository can be found here: `https://github.com/nowei/TetrisBot`.

2 Approach

We inspired ourselves from [1] and [2] and used evolutionary algorithms to build this Tetris player. Specifically, we used the Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES). In the following sections we will describe how we represent the state, how we model the cost function, more about our interpretations of the algorithm, and how we went about training.

2.1 State representation

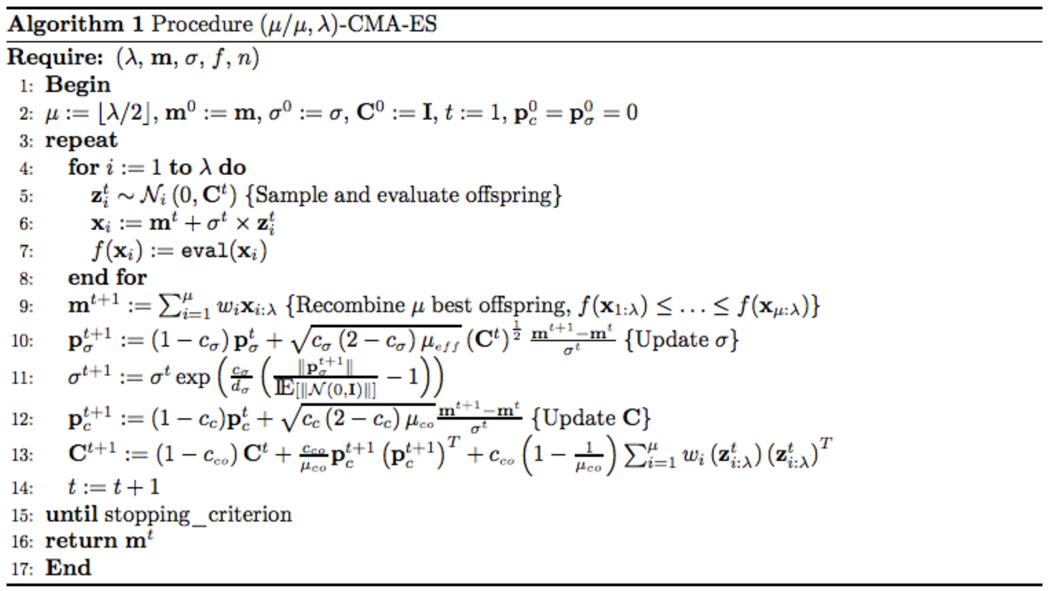
We chose to keep the 6 first features of [1].

The first feature is the landing height. It is computed as the height of the last tetromino added. The intuition behind the choice of this feature is that increasing the pile height generally leads to worse board (which could even sometimes end the game), and thus should be punished.

The second feature chosen is related to erosion of pieces, computed as the product of the number of rows eliminated by adding the last tetromino with the number of bricks eliminated from that last tetromino. This should value completing rows and adding pieces whose bricks will be immediately eliminated.

The third and fourth features are about the homogeneity of the board: these are respectively the number of horizontal full to empty or empty to full transitions between the cells on the board and the number of vertical full to empty or empty to full transitions between the cells on the board. The intention is to keep the board as homogeneous as possible, in the sense that a batch of holes next to each other is preferred to a sprinkle of one cell holes, as it is easier to fill one bigger gap than many small ones. Note that the walls of the board are considered as full cells.

The fifth feature is the number of holes. This is computed as the number of empty cells with at least one filled cell above (considering a deep hole to be a single hole). This leads to preventing holes as



```
Algorithm 1 Procedure $(\mu/\mu, \lambda)$-CMA-ES
Require: $(\lambda, \mathbf{m}, \sigma, f, n)$
 1: Begin
 2: $\mu := \lfloor \lambda/2 \rfloor$, $\mathbf{m}^0 := \mathbf{m}$, $\sigma^0 := \sigma$, $\mathbf{C}^0 := \mathbf{I}$, $t := 1$, $\mathbf{p}_c^0 = \mathbf{p}_\sigma^0 = 0$
 3: repeat
 4:   for $i := 1$ to $\lambda$ do
 5:     $\mathbf{z}_i^t \sim \mathcal{N}_i(0, \mathbf{C}^t)$ {Sample and evaluate offspring}
 6:     $\mathbf{x}_i := \mathbf{m}^t + \sigma^t \times \mathbf{z}_i^t$
 7:     $f(\mathbf{x}_i) := \mathtt{eval}(\mathbf{x}_i)$
 8:   end for
 9:   $\mathbf{m}^{t+1} := \sum_{i=1}^{\mu} w_i \mathbf{x}_{i:\lambda}$ {Recombine $\mu$ best offspring, $f(\mathbf{x}_{1:\lambda}) \leq \ldots \leq f(\mathbf{x}_{\mu:\lambda})$}
10:   $\mathbf{p}_\sigma^{t+1} := (1 - c_\sigma)\mathbf{p}_\sigma^t + \sqrt{c_\sigma(2 - c_\sigma)\mu_{eff}}\,(\mathbf{C}^t)^{\frac{1}{2}} \frac{\mathbf{m}^{t+1} - \mathbf{m}^t}{\sigma^t}$ {Update $\sigma$}
11:   $\sigma^{t+1} := \sigma^t \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|\mathbf{p}_\sigma^{t+1}\|}{\mathbb{E}[\|\mathcal{N}(0,\mathbf{I})\|]} - 1\right)\right)$
12:   $\mathbf{p}_c^{t+1} := (1 - c_c)\mathbf{p}_c^t + \sqrt{c_c(2 - c_c)\mu_{co}}\frac{\mathbf{m}^{t+1} - \mathbf{m}^t}{\sigma^t}$ {Update $\mathbf{C}$}
13:   $\mathbf{C}^{t+1} := (1 - c_{co})\mathbf{C}^t + \frac{c_{co}}{\mu_{co}}\mathbf{p}_c^{t+1}(\mathbf{p}_c^{t+1})^T + c_{co}\left(1 - \frac{1}{\mu_{co}}\right)\sum_{i=1}^{\mu} w_i (\mathbf{z}_{i:\lambda}^t)(\mathbf{z}_{i:\lambda}^t)^T$
14:   $t := t + 1$
15: until stopping_criterion
16: return $\mathbf{m}^t$
17: End
```

Figure 1: CMA-ES

much as possible and ignores how deep the hole is, as it is much less annoying (easier to clear) to have one deep hole than many one cell holes.

The sixth feature is the related to the board wells: it is computed as the sum of the accumulated depth over the wells. Preventing wells, especially the deeper ones is a good idea because they force to wait for a vertical bar.

We obtain the first two features by forward-simulating the action on the state and then calculate the remaining features from the resulting state.

2.2 Cost function

We set the reward of each action (that is, adding a tetromino) as the number of rows cleared by adding it. We initially chose this as the reward function because we felt like it would be the simplest way to represent the reward of (state, action) pair. We also don't have much use for the reward function much because the algorithm we chose to implement does not rely on the feedback of a reward function, but only the weighting of select features of the state and action.

2.3 Algorithm

We implemented an evolution strategy to learn the weights of the evaluation function that uses the aforementioned features. In the literature this approach is known as the covariance matrix adaptation evolution strategy (CMA-ES). Refer to Figure 1, from [1], for the description the general procedure of CMA-ES.

Basically, the CMA-ES is a recursive process during which points are sampled at each iteration. These samples (also called "offsprings") come from a distribution that is iteratively updated, by computing a mean ("centroid") and a covariance matrix from the best observed offspring. We followed the recomended values of [2] for the parameters c_σ, d_σ, c_c, μ_{eff} , μ_{eff}, c_{co}.

2.4 Training methods

We spent 2 days initially writing the code for CMA-ES and the features. Then we spent 7 days training the model, fixing bugs we found/restarting the training from scratch, and testing out the weights our algorithm found. We trained the children in parallel using separate instances of the environment to speed up training.

We also adopted the method of training on "harder" games from [1], which essentially changes the probability of getting "S" and "Z" pieces to 3/11 each with a 1/11 probability of getting any of the remaining 5 pieces, instead of the original 1/7 probability of getting any piece. We also added an option to train using a game with a 2/9 probability for "S" or "Z" pieces, as there was initially a concern about how well the weights obtained from the games with 3/11 probabilities for "S" and "Z" will perform when playing a normal game with 1/7 probability for all the pieces.

Some notable bugs we encountered were incorrectly computing features, incorrectly typing out the math for CMA-ES, accidentally hard-coding the training script to only train on hard games, and accidentally not using the updated sigma when computing new offsets for children.

There were some slight discrepancies within [1] and [2] for what the algorithm of CMA-ES was, so we attempted to train different instances using both. The discrepancies come up when trying to update the conjugate evolutionary path and covariance evolutionary path, where there is an σ factor difference in computing the second term. Wikipedia agrees with [2] [1]. The authors of [1] take an average of 100 games per evaluation per child to reduce the width of the confidence interval. We set a static number of episodes per evaluation per child and took an average of the scores to decide their ranking, trying both 5, 20, or 100. By an episode, we mean running the child until it has lost the game. We also tried varying the number of children, as we were concerned that we might have been unlucky in terms of how the children were sampled.

Initially, we decided that while it would be more consistent to evaluate each child on the same randomly generated sequence of pieces, we chose not to do this, as we believed that a child needs to be able to perform relatively well at any game it is given. Although later on, we realized that it could be the case that bad children get easier games and better children get harder games, making it hard to fairly compare how the children performed against each other, so we changed our training methods to evaluate the children on the same games. That is, on each generation of children, we generated a number of seeds equal to the number of episodes we wanted to run for each child, so that every child was evaluated using the same seeds. We also noticed that the `tetris.py` script was not properly using the random seeds that were being set.

Lastly, we also attempted to normalize the weights of the children, as described in [1], which bounds the weights to live on the unit sphere. We did not see significant improvement in performance from doing so, but we did see that the convergence did happen quicker and the step size did not become unbounded.

3 Results

We obtain the following results:

The best average performance we achieve while running 20 evaluations over random games is 5728.5, while achieving a maximum of 21253 in one of its 20 games. It was trained using the algorithm in [1], 5 episodes, using the same games, on the easy game (normal Tetris), with 9 children, and for 298 generations. We only include our best-performing weights in the repository and they can be tested using `python player.py`.

We trained many agents over the course of the last week. We will describe their performance on an average of twenty runs as well as their maximum lines cleared, and the parameters we used when training them. We initialized all the agents with a mean (m) of all 0s, a covariance evolution path (p_c) of all 0s, a conjugate evolution path (p_σ) of all 0s, and a step size (σ) of 0.5. We tried different variations of the following parameters: whether it was trained using the algorithm in [1] or in [2]; the number of episodes we trained with and whether it was using random or the same games; whether we trained on the normal (easy), hard, or harder game; whether it was normalized; and the number of children we tried. We report our results in the table. We may have agents with the same settings repeated, as we trained multiple models asynchronously and did not do a great job in communicating which models were being trained.

Something curious that we saw while training was instances in which the children of the weighted average of well-performing agents performed significantly worse than the their parents. This leads us to believe that it may be possible for the children of well-performing parents to move away from

[1]https://en.wikipedia.org/wiki/CMA-ES#Algorithm

well-performing regions within the fitness landscape, either due to their size of the region being small or our step size being too large.

Alg	Episodes, seeded	Normalized	Type	Children	Generation	Average	Max
[1]	5, yes	no	easy	9	298	5728.5	21253
[1]	5, yes	no	easy	9	756	4934.35	13368
[1]	5, yes	no	easy	9	111	2460.8	9412
[1]	50, yes	no	harder	18	44	2872.05	8650
[1]	100, no	yes	hard	9	67	4243.85	14395
[1]	100, no	yes	harder	9	54	2742.3	7163
[1]	100, yes	yes	harder	18	65	2680.0	11347
[2]	5, no	no	hard	9	1453	1590.1	6499
[2]	5, no	no	easy	9	963	2513.6	9063
[2]	5, no	no	easy	9	163	4704.3	32960
[2]	5, no	no	hard	9	963	3411.0	8415
[2]	5, no	no	easy	9	904	3702.8	10028
[2]	20, yes	no	hard	20	28	4863.1	12402

We also saw instances where less than μ children performed well, but the algorithm suggests taking a weighted average of μ children, so even if only one child performed well, we would have to take into account the contribution of bad children in the creation of the new mean for the next generation. Although one could argue that if this continues to happen, it's possible that the fitness landscape was not very good to begin with, as it was unlikely for the children to perform well.

These observations give some possible insights into possible extensions and modifications that can be made to the CMA-ES algorithm.

After talking to the TAs, we have reason to believe that there may be a small bug in our implementation of CMA-ES or the features or in our choice of parameters, as we do not achieve a similar performance to what was described in [1] or by the TAs.

The randomness of the performance seems to indicate that there is a lot of variability in Tetris games, which was also mentioned in the [1]. This makes it one of the main challenges of the assignment. That is, we must find an agent that can perform well despite the randomness of the games. We regret not having performed more exploration into tuning the hyperparameters for the CMA-ES algorithm, as we spent more time tuning the algorithm and things like adding normalization rather than tuning the algorithm's hyperparameters.

While we achieved a modest performance of 5728.5 average lines cleared, we believe that the endeavor was worthwhile and gave us a better understanding of what reinforcement learning environments and tasks are like in a more practical sense.

References

[1] A. Boumaza (2014). How to design good Tetris players

[2] N. Hansen, A. S.P. Niederberger, L. Guzzella, and P. Koumoutsakos (2008). A Method for Handling Uncertainty in Evolutionary Optimization with an Application to Feedback Control of Combustion