CS8803 ACRL HW2: Tetris Player

Tianrong Chen* and Yanjie Guo†
Georgia Institute of Technology, Atlanta, GA, 30313, USA

I. Introduction

In this homework, I applied three methodologies to train an agent to play a traditional Tetris game. The first methodology I used is, intuitively, Q learning. I selected three value to represent the weight of three features. Then the value function is the linear combination of the feature times the weight. In this implementation, the reward function plays a significant role. The different choice of reward function would lead to the extremely weird actions even though the action satisfies the requirement of maximizing the accumulate reward. I will talk about it later. Inspired by the form of fitted Q learning, which is similar to the gradient gradient, and the success of the Deep Q Network, I also tried DQN in this problem. Intuitively, instead of select the Engineering feature by ourselves, I tried to use CNN to extract the feature to get high reward. Two architecture of the DQN is appied but both of them failed. I will talk about my thinking about the reasons later. For the third method, we applied the cross-entropy method to find the best parameter which brings the highest reward. We also investigate several variants of this method, such as noisy cross entropy method, and create some features.

Note, we develop all this work in python, since both team members are not familiar with java. PyGame need to be installed before running our codes.

II. Trials

A. Fitted Q method

In this method, I selected three feature to represent for the current state in this problem. The baseline comes from the Akingabramson's implementation of his tetris AI. Please refer to his homepage:$https://github.com/akingabramson/tetris\_reinforcement\_learning\_ai$. Here, the feature I tried is as following:First one is the differences between the height of the current total blocks and the height of the block after do the specific action, the second is the difference between the holes of the current state and the holes of the next state after doing the specific action, the final one is the differences between how many counters and the counters of the next state. The counters here means how many ups and downs of the surface of the blocks. 1.1Play with fitted V method: I played around his code and did some interesting experiment on that. First, I implement the fitted V method. The update rule of the fitted V as following:
$\theta \leftarrow \theta + \alpha[(r + \gamma Q(x', a)) - V(x, a)]\nabla Q_\theta(x, a)$
Since the $V_{theta}(x)$ is the linear combination of the feature weight, so $\nabla V_\theta(x)$ becomes the specific feature value.

The result is shown in Fig. 1:

As we can see, the result is not satisfied. Meanwhile, it costs more time for each selection of decision.

1.2 Analysis

Why it does not work? The first idea, I think, is because of the characteristic of Bellman equation. As we can see from the update equation above, the term we are only interested in is the current state value function and the following state's Q function. Our policy is only depend on the former updated weight. This characteristic push us to update every one step and it will leads to high variance.

2.1 Play with reward function: It is clear that the reward function plays an important role in this game. Accordingly, I tried several reward function. First one is the numbers of cleared lines. Since the agent can hardly obtain the reward at the beginning, The update procedure does not bootstrap the Q value in the most of case. Since this weight is used for the following decision selection, the weight will become worse and worse, the decision will be getting worse as well. (please refer to the function $update1$ in tetris.py) Second one is the differences between the height. The reward now is plentiful, in the worst case, it will at least a number be a value which is smaller than zero. It leads to the result shown in the figure above. (In terms of code, please refer to the function $update_new$ in tetris.py) Third one is the differences

*Master Student, Graduate research assistant, Electrical and Computer Engineering.
†PhD Student, Graduate teaching assistant, Daniel Guggenheim School of Aerospace Engineering.

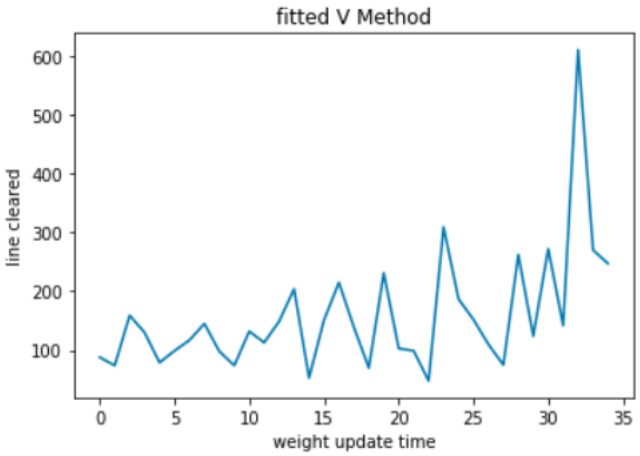

Fig. 1 Fitted V method.

between the height plus the differences between the holes. It leads to a very interesting result. This tetris agent trapped itself in a sub optimal point by a weird way as shown in Fig. 2:

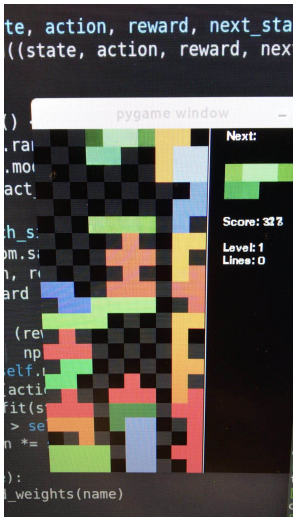

Fig. 2 Weird behavior.

The reason is as following. The second term of reward, differences between holes, is much more easier to get a positive reward than the height. The agent can get positive reward from the different between height only form clearing lines. Otherwise, the reward can only be zero or negative value. On the other hand, for the differences between holes, for the next state, the agent can get positive reward simply by filling the empty place. Meanwhile, it is a Q learning method, the agent only care about the successor state and the current state, instead of the total reward, so the agent will not care about whether I clear lines for the higher total reward. It only cares, for the next step, how can I get higher reward.

B. DQN method

In this method, I implement two architecture of the RNN. In the first architecture, the input of the neural network is the image of the board. Then it goes throw two CNN network, then the extracted features captured by CNN are fed into one dense layer, and the output is a 40 dimension tensor, each element of one dimension represents for the grade of each action. Then I do a softmax to select the action which result in the highest grade. As usual, I put a dropout in the neural

network to avoid overfit. There are a 2000 buffer, which restore the history. When we train the neural network, 24 histories memory will be extracted to train the neural network. The only differences of the second architecture has is that, the output has only 1 output, which represent for the value of current state, instead of the value of the action under current state. So we are approximating the value function instead of Q function. Both of them failed. By analysis the reasons, I think the biggest problem is the reward. As a Q-learning class algorithm, we are using the policy which is trained by former state, meanwhile, the reward bootstraps this policy as well. In terms of the analysis of the reward, you can refer to the former part–policy gradient. Meanwhile, another big problem I think, comes from the CNN. Without engineering selected feature, we can hardly say the feature extracted by CNN is reasonable to determine the value of a specific action, or state. Or in other word, it will be hard to converge. I think, for the following work, we can add some hand craft feature in the end layer of CNN. This will probably improve the stability of the neural network.

III. Methodology

After some trials and failures, we gradually realized that maybe the response surface of the total reward with respect to our design parameters are not smooth at all. Especially when we have randomness in the system and don't have too much time to estimate a good expectation. Even if the surface is some what smooth over the expectation, it is very likely to be multi-modal. From this insight, we decide not to use any gradient based method such as policy gradient or any gradient descent optimization algorithm.

Also, we note that the next block is random and the optimal policy should be robust under this randomness. It doesn't make any sense to build a model to anticipate future blocks. Thus, any model-based method is not suitable for this problem.

The last insight is that, at least for tetris, human knowledge outperforms insufficiently trained AI feature extractors. Since this game seems so trivial, we can hand designed some very good features. Whereas AI, such as CNN or DQN, will need some delicately designed objective function or reward, as well as tons of training data, to generate only comparably good features. The easy way to solve this problem must exploit our human experience. If we really want to use some neural net for feature selection, we might also augment some hand designed features to improve the performance of that algorithm.

From these insights, we decide to explore mainly model-free methods and some black box optimization methods, using human designed features. From [1] and [2], we learn that CEM with or without noise can perform very good with suitable features. Moreover, considering that the response surface may be very noisy, in order to avoid multi modalities and not to be trapped in a local optimum, we augment it with simulated annealing as our black box optimization algorithm.

A. Cross Entropy method (CEM)

CEM is a zeroth order optimization method, i.e. it doesn't access the gradient information of the objective function. The idea is very simple. We first sample 100 points in the design space from a Gaussian distribution with zeros mean and 100 identity covariance matrix. For later iterations, we perform policy evaluation for each point, i.e. compute its total reward; select the top 5-15% designs and use their mean and covariance to generate next 50 sample and repeat. The main problem for this method is that the first few iterations work fairly fast since those policies reach gameover very soon. But at later iterations, our policies are becoming really good thus it takes a really long time for them to finish one game, which greatly limited our sampling size. This can cause problems when we use too few points to generate a distribution.

Another interesting point we found is that, in later iterations where we have several equivalently good policies, which can clear around 4 thousand. However, their distance in the design space are very large, indicating that they are very different policies. This totally make since because while the best policies may look similar, the sub optimal policies can be different. In order to possibly fix this issue, one way is to add noise in the covariance. And we come up with another one, a modified simulated annealing method with half CEM inside.

B. Simulated Annealing (SA) with CEM

Simulated annealing is best known by its ability to find the global optimum, which is ideal to deal with the locality issue. The algorithm is simple. We first sample 50 points from a Gaussian distribution as mentioned above and compute the total lines cleared. For later iterations, we first generate another 50 samples from the previous 50 distribution, and then randomly search the neighborhood for each point and update it if the new point is better or accept with the

exponential probability $e^{\Delta R/T}$, where ΔR is the difference between the reward and is negative if it is worse, and T is our temperature schedule. We then keep the top 50 points among the new 100 population. We can see this algorithm as half SA half CEM with 50% elite rate. From another point of view, this is a genetic algorithm where we select the top half as parents. The CEM samples are their offspring and the SA samples are mutations. The comparison will be shown in the result.

IV. Results

We fisrt use 3 features, number of holes, height of the board, and the contour, for CEM and compare the performance with and without noise. The statistic is shown in Fig. 3. The top line is the max, the bottom line is the min, and the solid line is the mean. We can see that given noise, the variance will be higher. The worse policies keep dying immediately, which lower the mean. Without noise, the algorithm improves faster but seems to be trapped by the multi modality and doesn't converge.

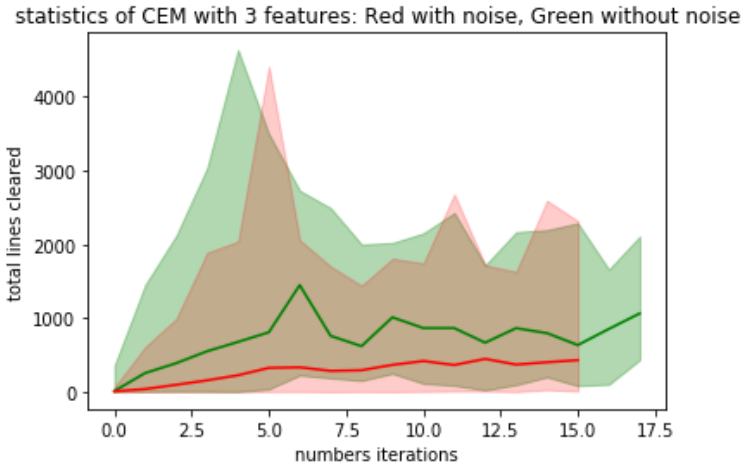

Fig. 3 Statistics of CEM with 3 features: Red with noise, Green without noise .

Then we add two more features, the current number of lines clears and the number of rows with holes, hoping our player can exploit the new information. As shown in Fig. 4, the max now increase to around 4k very fast compared with 3 features. CEM gives a really high score around 9.4k lines cleared. But this is considered as the randomness in the sampling, i.e. we are just lucky. We can see the next iteration comes back to around 4k. SA/CEM, on the other hand, is a little bit slower than CEM because the elite rate is probably too high. We stopped the algorithm since we don't have time. But it seems from the tendency that it can go higher if we keep running it.

All codes for this part are implement by ourselves from the scratch. But we do use the PyGame tetris code to run the simulation. The best results can be run in corresponding python notebook. All the statistics are saved in text files.

V. Conclusion and Future Work

We can see that more features is better. Gradient based or model based algorithm is not suitable for this problem. The best results comes from black box optimization algorithms, since the simulation is relatively cheap. And still we need to fight the locality problem.

Something that can be done in the future is to test more features. For example, we don't use any information about the next block in the policy. This might ease the problem a lot. Also, we can keep fine tune the hyper-parameters and run the algorithm for a longer time.

References

[1] Szita, I., and Lörincz, A., "Learning Tetris Using the Noisy Cross-Entropy Method," *Neural Computation*, Vol. 18, No. 12, 2006, p. 2936–2941. doi:10.1162/neco.2006.18.12.2936.

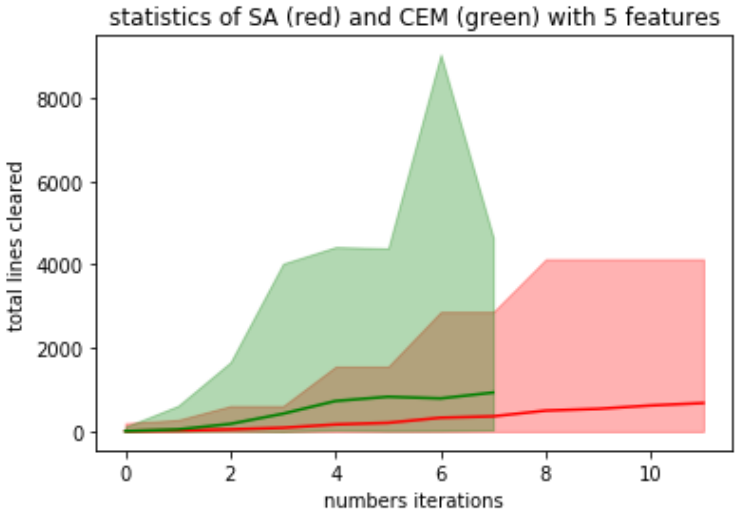

Fig. 4 Statistics of SA (red) and CEM (Green) with 5 features.

[2] Thiery, C., and Scherrer, B., "Improvements on Learning Tetris with Cross Entropy," *ICGA Journal*, Vol. 32, No. 1, 2009, p. 23–33. doi:10.3233/icg-2009-32104.