How to design bad Tetris players

Benjamin Wong

November 2022

1 Introduction

1.1 Related work

This version of tetris is known as the tetris problem in the literature, where the pieces are to be placed to the board without any dynamic effect as opposed to the version played by human. The state space is intractably large, which means even with known probability the problem cannot be solved by simple MDP method. Most method first contract the board state into feature state before learning. [1] claimed to have trained a deep Q model to play the game with dynamics. The owner contracted the board in to features similar to [2], but with only 4: lines cleared, holes, bumpiness, and height. The feature vector is passed trough a 3 layer deep network and outputs the Q value. [3] designed a tetris player using deep convolution network. The entire board is fed into the neural network to output the Q value, this work uses the same 4 heuristic features to create heuristic reward instead of as state to accelerate the learning. [4] also designed a deep Q network with 1000 line cleared.

2 Methodology

2.1 Approach

The approach is to use Q learning given a handful of successful example from the review. On the other hand, Policy-based learning would be tricky with the state dependant action space. The Q-function in my approach would be partially approximated by a neural network. The overall approach can be seen as a 1-step tree search followed by the Q function of the next state as discussed in the below section.

2.2 Formulation

In principle, every state in the tetris game can be fully defined by the tuple (X_b, X_{np}). Where X_b is the entire configuration of the 20x10 board with a binary occupancy value on each cell. X_{np} is the id of one of the seven terimino to be

placed at the current round. The action space is the legal place where the next piece can be placed to the board.

With this the Q-function can be written as

$$Q(X_b, X_{np}, a) = \mathbb{E}[r(X_b, X_{np}, a) + \gamma \max_{a'} Q((X'_b, X'_{np}), a')] \tag{1}$$

one important fact, as many articles have pointed out, is that the action can be simplified into choosing the next board state. Moreover, this board state transition is completely deterministic since the current piece is already known and there are no noise associated. The Q function can written as such

$$Q(X_b, X_{np}, a) = r(X_b, X'_b(a)) + \gamma \mathbb{E}[\max_{a'} Q((X'_b(a), X'_{np}), a')] \tag{2}$$

This suggests that the reward is completely know at any given state and can be maximized by simulating all available action, this does not need to be learned by the Q-learning and greatly reduces the complexity. Now, focusing on the reward-to-go, the only stochastic part of the state is the X'_{np} and thus the associated allowable actions are also unknown. The goal here is to fit a neural network to the reward-to-go. Since the $X'_b(a)$ is deterministic, this will be the input to the neural net instead of the current board state X_b.

$$Q(X_b, X'_b) = r(X_b, X'_b) + \gamma h(X'_b) \tag{3}$$

$$h(X'_b(a)) = \mathbb{E}[\max_{a'} Q((X'_b(a), X'_{np}), a')] \tag{4}$$

In other word, the neural net is only learning the expected value function of the net board configuration instead of the entire Q function of the state-action pair.

The parameter update law is straight forward. According to the lecture, the target given data (x, a, x')

$$y = r + \gamma \max_{a'} Q(x', a') = r + \gamma h(X'_b) = r + \gamma(r' + \gamma h(X''_b)) \tag{5}$$

$$\nabla_\theta L = (y - Q(x, a))[\nabla_\theta y - \nabla_\theta Q(x, a)] \tag{6}$$

$$\nabla_\theta L = \gamma^2(r' + \gamma h(X''_b) - h(X'_b))[\nabla_\theta(r' + \gamma h(X''_b)) - \nabla_\theta h(X'_b)] \tag{7}$$

$$\text{Let } y' = r' + \gamma h(X''_b) \tag{8}$$

$$\nabla_\theta L = \gamma^2(y' - h(X'_b))[\nabla_\theta y' - \nabla_\theta h(X'_b)] \tag{9}$$

This form is exactly the same as the ordinary gradient decent of Q-learning but one step ahead and scaled by γ^2, which can be merged with the learning rate.

2.3 Reward

The ultimate goal of tetris is to not lose. Every other reward including clearing row can be seen as heuristic. Losing also act as the only terminating state. With this, naturally the reward of losing should be crafted first and guide the other reward type. In this project i have assigned -9000 for losing. Clearing rows is obviously the only possible action to stay away from losing indefinitely, therefore should be assigned a high reward. However, no amount of row cleared in the future should be able to balance out the cost of losing. The maximum row one can clear in one turn is 4. The maximum expected reward is bounded by

$$Q \leq \frac{4 * r_{clear}}{1 - \gamma} < |r_{losing}| \tag{10}$$

in this project, the reward for clearing row is chosen to be 100 and γ is 0.9. Other heuristic reward such as number of holes, bumpiness, maximum height were added similar to [3], but the performance was not good, for instance the policy converged to stacking blocks to avoid making holes instead of clearing lines. All heuristics were taken away for the final version. A reward of 1 is granted for each turn staying alive in the game.

2.4 Implementation

2.5 Neural Network Inputs

Instead of fully relies on feeding the entire board to the CNN or fully condense everything to feature vector, this approach uses a couple obviously impactful features such as maximum height of the entire board, bumpiness (measured in standard deviation), and total number of holes (total number of empty cells under occupied cell for each column) to aid the learning. As using the convolutions network to learning every important features on it's own would requires a vastly more complex network and thus a lot more training would be required. Other more sophisticated features from [2] are omitted. A shallow convolutional network is implemented to let the network learn other important features not captured by the heuristics. The results are concatenated and pass through several fully connected network to generate the final value function. The detailed architecture is shown in figure 1.

2.6 Player

The player training policy is primarily choosing the action that maximizing the Q-function. With a modified ϵ-greedy policy, where with a 0.01 chance, the player would randomly choose 1 action from the top 3 actions for very light exploration. At each round within the game, the tuple (state, action, reward, next state, legal next actions) is stored in a batch. After the game ended, the batch is transferred to the player's short-term memory. If the number of batches in the short-term memory exceeds 10, the player will select one random episode

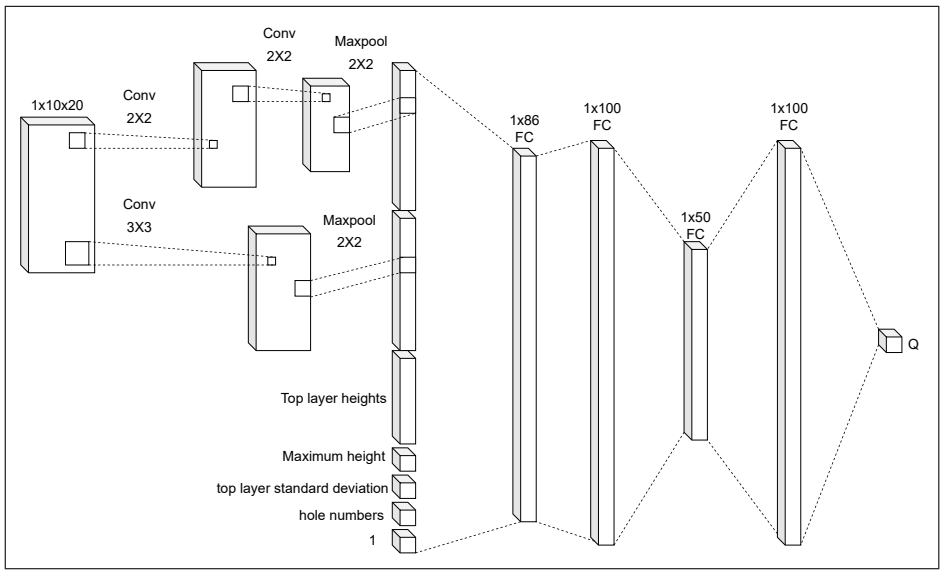

Figure 1: Convolutions Neural Network Architecture

from the long-term memory and compiled with the short-term memory into a training batch, the training batch is randomly shuffled and split into mini-batch of size 500 and passed into the neural network for training. The 10 batches in the short-term memory is then transferred to the long term memory. After it is transferred the short-term memory will be erased. The maximum length of the long-term memory is kept at 1000 where older memory will be erased first.

3 Evaluation

This model was trained for roughly 5 days non-stop, over 16000 episodes. The training process can be seen at figure 2, which shows the number of row cleared at each episode and the moving average with a window size of 100 episodes. The learning was initially very rapid around 1000th episode. The row cleared rapidly dropped after that and started slowly increasing. The running average shows a steady roughly line increase, and the maximum row cleared seems to be growing exponentially and peaked at 600 lines, suggesting that when the condition is right the model can take advantage of it but not doing as well on average which peaked at about 110. The corresponding MSE loss of the neural net is shown at figure 3. Both the row cleared and the MSE loss shows that the learning process is not yet finished at the point of termination. It is unknown at which point the model would plateau but it is expected it can at least reach middle player level given enough time. The back-propagation and one-step look-a-head simulation takes a long time to calculate on a lab-top GPU causing ineffective learning.

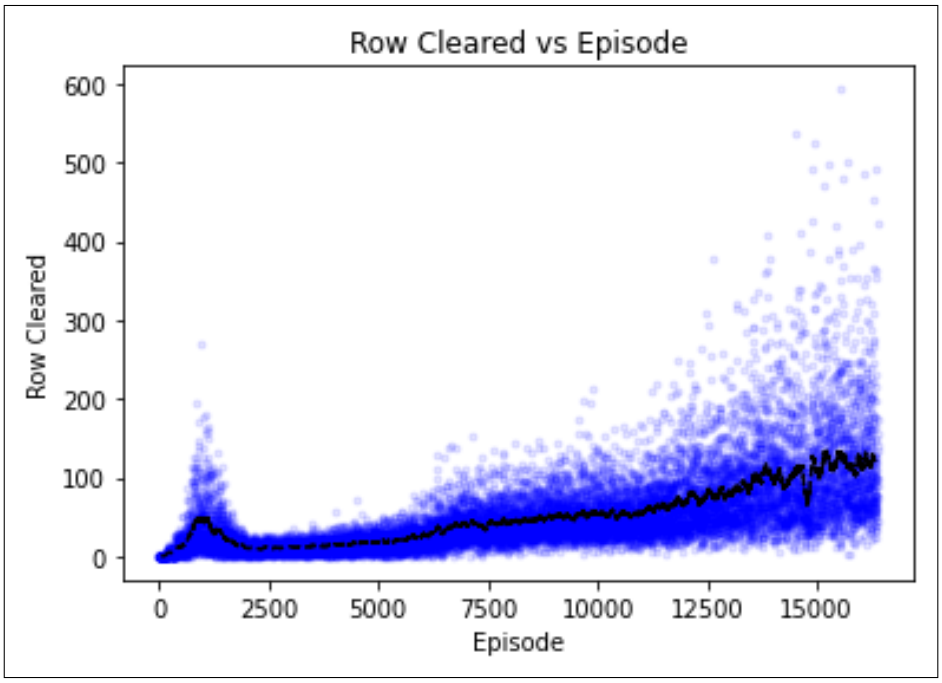

Figure 2: Row cleared during training

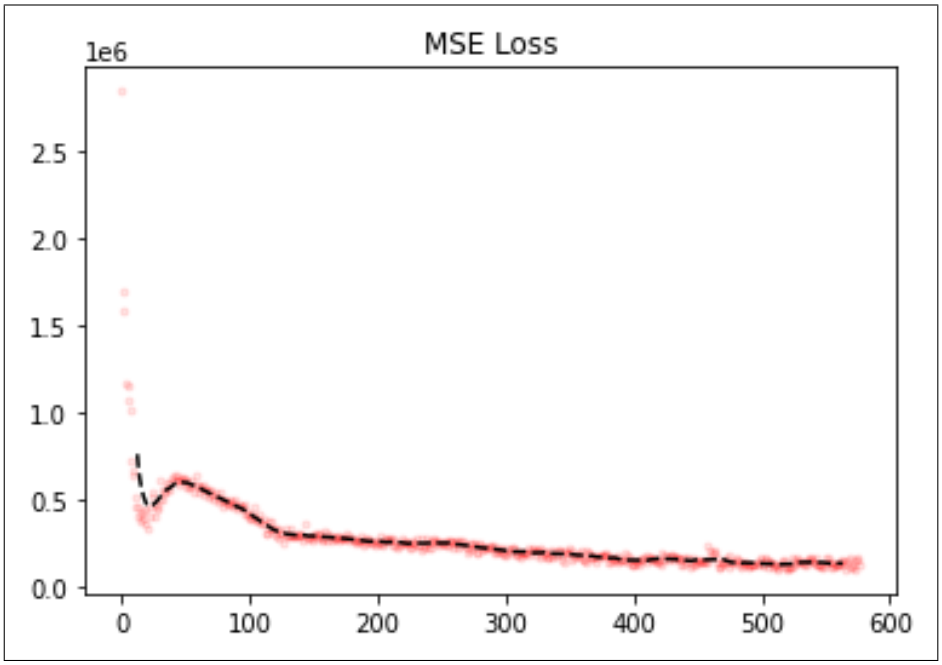

Figure 3: MSE loss during training

For the 20 test trial, the final model results in clearing 134.6 $\pm$ 73.87 lines on average. The maximum number of line cleared was 341. which makes it a poor player.

4 Potential Improvements

As [3] suggested, the convolution kernel could be column and row wise, instead of square, since most features of tetris game is row or column dependent. The current small kernel may not capture the hole distribution for a single column which might be an important feature.

Another way is to train the fully connected network first using all the features suggested by [2], and then appended the convolution network once the fully connected network is well trained such that the convolution network is only there to capture any missing features.

Another hindsight is that the "top layer height" feature vector should have been passed through a 1d convolution before appending to the vector to extract top layer features instead of using every single bit of information to reduce complexity.

References

[1] V. Nguyen, Tetris-deep-q-learning-pytorch, `https://github.com/uvipen/Tetris-deep-Q-learning-pytorch` (2020).

[2] A. Boumaza, How to design good tetris players. (2013).
URL `https://hal.inria.fr/ hal-00926213/document`

[3] M. Stevens, S. Pradhan, Playing tetris with deep reinforcement learning (2016).
URL `http://cs231n.stanford.edu/reports/2016/pdfs/121`$_R$*eport.pdf*

[4] H. Liu, L. Liu, Learn to play tetris with deep reinforcement learning.
URL `https://openreview.net/pdf?id=8TLyqLGQ7Tg`