ACRL Homework 1: Tetris

Eleanor Avrunin, Humphrey Hu, Kumar Shaurya Shankar

February 27, 2014

1 Choice of Algorithm and Motivation

We approached the Tetris task by using Policy Gradient. Our main motivation was that we wanted to exploit the knowledge of the structure of the system due to the availability of cheap, inexpensive forward simulation available via the game simulator. In implementation, we tested out REINFORCE, G(PO)MDP, and Natural Policy Gradient as presented in class, with additional reference from Peters and Schaal [1]. Natural Policy Gradient performed the best of the three, though this may also be due to our implementation being less numerically sensitive than the other two implementations.

2 Algorithm

We implemented a variant general Policy Gradient Theorem inspired by the REINFORCE algorithm. We incorporate causality by only calculating rewards that are obtained after the current timestep, since the actions that are chosen at a particular timestep cannot influence the rewards in the past. The reward function to go is thus just the sum of individual rewards over the length of the trajectory till death. The algorithm is as follows:

Algorithm 1 Natural Policy Gradient

for $i = 1 \dots n_{\text{fitting iterations}}$ **do**
  Run simulator with π_θ to collect $\xi^{1 \dots N}$
  for $j = 1 \dots N$ **do**
    $z_1 \leftarrow \vec{0},\ \Delta_1 \leftarrow \vec{0}$
    for $t = 1 \dots T-1$ **do**
      $z_{t+1} \leftarrow \gamma z_t + \nabla_\theta \log \pi \left(a_t^i | s_t^i\right)$
      $\Delta_{t+1} \leftarrow \Delta_t + \frac{1}{t+1} \left(z_{t+1} \sum_{\tau=t+1}^{T} r_\tau - \Delta_t\right)$
    end for
    $\delta(j) \leftarrow \Delta_T$
  end for
  $\hat{\delta} \leftarrow \frac{1}{N} \sum_{i=1}^{N} \delta(j)$
  $\theta_{new} \leftarrow \theta_{old} + \alpha \hat{\delta}$
end for

3 Features

The features we use are all based on the state of the board after taking an action. We assert that the path taken to a board configuration is unimportant, hence our choice of state being only the board itself. This might not be the case for a robot, however, where different motion or sensing actions could have different associated costs. This single step lookahead can be interpreted as a very crude state-action value estimate. A better estimate would perform many steps of lookahead, though this is computationally intractable. Utilizing a state-action value estimator in an actor-critic framework could address this problem, though we did not investigate it due to time constraints.

In looking at the board after taking an action, we consider the features used by human Tetris players. These features are enumerated and expanded upon below.

1. **Column heights (10)** - The height of the tallest block in each column
2. **Column height differences (9)** - The absolute value of the difference in height between adjacent columns
3. **Max column height** - The maximum of the column heights
4. **Number of holes (10)** - The sum of the column heights minus the number of filled tiles

In general, these are the height of the board, how full the space is, and how many rows can be cleared, which we then separate into a number of individual features. The height of the board is the primary factor in how likely the player is to die soon, and the density of the board determines both how likely it is that more rows can be cleared (higher reward) and how likely it is that the height will increase.

For the height of the board, our features are based on the individual column heights, accessed using the `getTop()` method provided. We include as features the height of the tallest and shortest columns, and the average height of the columns.

We describe the density of the board using the number of filled squares in the rows. As with the columns, we have features for the size of the largest and smallest rows. We also pay particular attention to the empty squares below where the agent has placed pieces. We compute the total number of holes (empty squares with a filled square higher in the same column) in the board. As another measure of density and available space, we also compute the number of empty squares below the highest filled square in the board, whether or not there is a filled square above them. This distinguishes boards that are fairly evenly filled from boards of the same height that have a tower of pieces in one small area.

Our final feature is the number of rows cleared by making the given move.

We also tried a number of other features. One such feature was the number of filled squares in the top four rows of the board, which was meant to strongly discourage the agent getting close to the end of the game. We also tried including the height and number of holes for each column individually, and the number of filled squares in each row. We tried several different measures of density, such as the number of filled squares in the highest row with a piece and the average number of filled squares in a row (over both the whole board and the region with pieces). Our results with these additional features were slightly worse than with the feature set described above, so we discarded them. In addition, we considered including the number of overhangs (filled squared with an empty square below them, the counterpart to holes) as a feature, but in the end decided it

was not likely to produce a significant improvement when we already had the number of holes as a feature.

We sped up the computation of the features by making a binary copy of the board, where filled squares have value 1 instead of the ID of the piece at that location. This allows us to compute the number of filled squares in a row or column by summation, and find the gaps by subtraction from the total number of squares in a row or the height of the column.

4 Implementation Details

For the simulation we use the Java based simulator provided in the assignment. Our system architecture consists of generic interfaces that are implemented as class variants for modularity. We generate random trajectories and calculate gradient terms by concurrently forward simulating using multiple threads. Interestingly, this approach led us to discover a bug in the simulator implementation, where concurrent read accesses to a shared array of legal moves was causing incorrect values to be returned, effectively adding stochasticity into the system transition model. The stochasticity was particularly detrimental when starting training.

The policy distribution function is designed to be a Boltzmann distribution that operates on a linear function of features, i.e.,

$$\pi(a|s) = \frac{\exp\left(\theta^T f\left(s'_a\right)\right)}{\sum_{a'} \exp\left(\theta^T f\left(s'_{a'}\right)\right)}$$

Since the policy is stochastic, we pick an action to take by first generating the probability distribution of picking an action given a state, and then pick the action by sampling from this distribution. We normalize our probabilities when generating a probability mass function (PMF) by subtracting the largest exponent (log-likelihood) from each exponent. This guarantees that the exponential will produce values between 0 and 1. An alternative normalization is to normalize the feature vector to have unit norm, but this results in a smoothed PMF. We sample from the PMF by picking a random number from a uniform distribution and finding the inverse of the cumulative distribution function.

5 Training Procedure

We initialize our weight parameter $\theta^{(0)}$ to zeros, and used a reward function that gave 10 for each row cleared, 1 for each turn alive, and -100 upon death. Each training iteration is then as follows:

1. Generate 64 trajectories and their respective gradient terms with $\beta^{(t)}$ random moves.
2. Calculate the natural gradient step from the trajectories
3. Apply the update $\theta^{(t+1)} = \theta^{(t)} * \alpha^{(t)}$
4. Repeat

We experimented with various step and exploration series. Poor chocies of step size would result in the learning process diverging as the parameter vector grew unbounded, and poor choices in exploration resulted in very slow progress. The learning process appeared extremely sensitive

to these hyperparameters and choice of features. This suggests that our attempts to numerically stabilize the learning process and policy may be insufficient. We also experimented with different exploration methods, such as using a temperature quotient term in the exponents, analagous to simulated annealing. We also tested starting with high discount factors to quickly learn short-term effects, and then decreasing the discount factor over time.

6 Results and Discussion

For the implemented method, we obtain an average of 1323 rows cleared with a standard deviation of 975 over 12 runs. The high variance in the results suggests that the learned policy is not very robust and could use continued improvement. It is unclear at this point despite extensive debugging and testing whether there are issues in our implementation with numerical stability, or an incorrect implementation of the policy gradient algorithm.

References

[1] J. Peters and S. Schaal. Policy gradient methods for robotics. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2219–2225, Oct 2006.