Learning Tetris

Matt Zucker Andrew Maas

February 3, 2009

1 Tetris

The Tetris game has been used as a benchmark for Machine Learning tasks because its large state space (over 2^{200} cell configurations are possible) and sizable branching factor (a typical piece may be placed in over 30 configurations) necessitate the use of effective heuristic approximations in order to produce viable policies.

The standard Tetris board consists of a 10 column by 20 row playing field. Tetrominoes, or connected shapes of four cells each, are dropped into the playing field from the top and stacked upon the bottom of the field or already-occupied cells. There are seven different tetrominoes; after every move, the next piece to be dropped is selected among them with uniform probability. The game state x therefore consists of the occupancy of each of the 200 cells, along with the next tetromino to be dropped. A player can rotate (but not flip) the current piece befre dropping it. Dropping the piece in any column is allowed, as long as it stays within the 10×20 field. Hence, the action $u = (p, r)$ consists of the desired position and rotation of the current piece. The set $U(x)$ is the set of all valid actions given the state x.

When a full 10-column line of cells is occupied, the line is removed and all cells above it are moved down by one cell. Multiple lines may be removed at a time. Each line removed adds one to the player's score. The game is over when any cell in the top row becomes occupied. The goal of the game is to maximize the score. Denote the successor of the state x after applying action u by x'. To generate the successor, the piece is dropped, any full lines are removed to generate the one-step reward $r(x, u)$, and a new piece to drop is selected at

random. Therefore we draw x' from the distribution

$$x' \sim p(x'|x, u)$$

2 Quality Estimation Function

Many heuristic approaches are possible; we chose to optimize a parameterized function which approximates the quality of a state based on a low-dimensional set of features extracted from the state. Let $f(x) \in \mathbb{R}^n$ denote the feature vector extracted from the state x. Note that we do not include any information about the current piece in the feature vector. Therefore we can write a deterministic function $f(x, u)$, which is shorthand for the features extracted from the successor state x', because the only non-deterministic aspect of x' is the current piece to be dropped.

The 22 features initially used to do learning consist of the following:

- The height of the topmost occupied cell h_i in each column i (10 features)
- The absolute difference $|h_{i+1} - h_i|$ between adjacent columns (nine features)
- The maximum height of any column (one feature)
- The number of holes on the board – unfilled cells with one or more filled cells above them (one feature)
- The previous one-step reward $r(x, u)$ indicating the number of lines just removed (one feature)

Let the function $Q(\theta, f(x))$ denote the estimated quality of state x based on a parameter vector $\theta \in \mathbb{R}^m$. Such a quality function can implicity impose a policy: for a given state x, we can choose which action $u \in U(x)$ to apply by maximizing Q over the set of possible successor states:

$$\pi(x) = \arg\max_{u \in U(x)} Q\big(\theta, f(x, u)\big)$$

The simplest such estimation function is the linear function

$$Q\big(\theta, f(x, u)\big) = \theta^T f(x, u)$$

More complicated functions might entail artificial neural networks or other non-linear components.

3 Stochastic Policy Gradient Descent

Instead of this deterministic policy, we may also use a stochastic policy which selects the next action from a multinomial distribution $q(\theta, x, u)$:

$$u \sim q(\theta, x, u)$$

By turning the hard max above into a soft max, we get the familiar Boltzmann distribution:

$$\begin{aligned} \ell(\theta, x, u) &= \exp\big(Q(\theta, f(x, u))\big) \\ Z(\theta, x) &= \sum_{u_i \in U(x)} \ell(\theta, x, u_i) \\ q(\theta, x, u) &= \frac{\ell(\theta, x, u)}{Z(\theta, x)} \end{aligned}$$

The REINFORCE algorithm [2] allows us to optimize the parameters of a stochastic policy by taking the gradient of expected reward. In order to do REINFORCE, we need to be able to compute the *score ratio*

$$\frac{\nabla q(\theta, x, u)}{q(\theta, x, u)}$$

We observe:

$$\begin{aligned} \nabla \ell(\theta, x, u) &= \nabla Q\big(\theta, f(x, u)\big)\, \ell(\theta, x, u) \\ \nabla Z(\theta, x) &= \sum_{u_i \in U(x)} \nabla \ell(\theta, x, u_i) \end{aligned}$$

Now we use the product rule to get the derivative of $q(\theta, x, u)$:

$$\nabla q(\theta, x, u) = \frac{-1}{Z(\theta, x)^2} \nabla Z(\theta, x)\, \ell(\theta, x, u) + \frac{\nabla \ell(\theta, x, u)}{Z(\theta, x)}$$

And we can divide through by $q(\theta, x, u)$ to get the score ratio

$$\begin{aligned}
\frac{\nabla q(\theta,x,u)}{q(\theta,x,u)} &= \frac{-\nabla Z(\theta,x)}{Z(\theta,x)} + \frac{\nabla \ell(\theta,x,u)}{Z(\theta,x)}\frac{Z(\theta,x)}{\ell(\theta,x,u)} \\
&= \frac{-\nabla Z(\theta,x)}{Z(\theta,x)} + \frac{\nabla \ell(\theta,x,u)}{\ell(\theta,x,u)} \\
&= \frac{-\nabla Z(\theta,x)}{Z(\theta,x)} + \nabla Q\big(\theta, f(x,u)\big) \\
&= \nabla Q\big(\theta, f(x,u)\big) - \frac{1}{Z(\theta,x)} \sum_{u_i \in U(x)} \nabla \ell(\theta,x,u_i) \\
&= \nabla Q\big(\theta, f(x,u)\big) - \frac{1}{Z(\theta,x)} \sum_{u_i \in U(x)} \nabla Q\big(\theta, f(x,u_i)\big)\, \ell(\theta,x,u_i) \\
&= \nabla Q\big(\theta, f(x,u)\big) - \sum_{u_i \in U(x)} \nabla Q\big(\theta, f(x,u_i)\big)\, q(\theta,x,u_i) \\
&= \nabla Q\big(\theta, f(x,u)\big) - E_q[\nabla Q(\theta, f)]
\end{aligned}$$

When Q is linear in f:

$$Q\big(\theta, f(x,u)\big) = \theta^T f(x,u)$$

we get the very simple derivation

$$\frac{\nabla q(\theta,x,u)}{q(\theta,x,u)} = f(x,u) - E_q[f]$$

4 REINFORCE Update Rule

To update the weights θ, we run the policy and compute a step Δ which can be added to the weights. This is the "Markov Chain Gradient" algorithm from [2]:

$$\begin{aligned}
z_{t+1} &= \beta\, z_t + \frac{\nabla q(\theta, x_{t+1}, u_{t+1})}{q(\theta, x_{t+1}, u_{t+1})} \\
\Delta_{t+1} &= \Delta_t + \frac{t}{t+1}\Big(r(x_{t+1}, u_{t+1})\, z_{t+1} - \Delta_t\Big)
\end{aligned}$$

We initialize with $z_0, \Delta_0 \in \mathbb{R}^m = \mathbf{0}$. Weights are updated by

$$\theta \leftarrow \theta + \alpha\Delta$$

for some small scalar α. We can get faster convergence by using a covariant gradient method [1], which tracks the covariance G of the state transition distribution:

$$G_{t+1} = G_t + \frac{t}{t+1}\left(z_{t+1}\, z_{t+1}^T - G_t\right)$$

where G_0 is initialized a matrix with small positive values on the main diagonal in order to ensure numerical stability. The update rule is changed to

$$\theta \leftarrow \theta + \alpha G^{-1}\Delta$$

This chooses a modification of weights which is small with respect to the current distribution.

4.1 Practical Issues

Two questions remain: when should we reset Δ and G, and how often should we take steps to update θ? Empirically, we found that taking a gradient step after the end of every Tetris game was sufficient. Instead of setting Δ and t to zero after every update, we found that decaying both by a factor of two added "momentum" which sped learning. In practice, we update G by keeping a moving average over the last $10,000$ or so steps. G is never reset.

5 Initial Results

Initially, we used the 22-element feature vector described in section 2, with the linear function $Q(\theta, f) = \theta^T f$. After running REINFORCE over about 5,000 games of Tetris, the policy was evaluated on 100 games with learning turned off. The mean score over the 100 games was 4,331 lines, and the maximum score was 31,705 lines.

6 Additional Lookahead

We can increase performance of our Q function by using an "expected-max" lookahead function Q^L which we will define recursively. Let the base function

Q^0 be defined as $Q^0(\theta, x, u) = Q(\theta, f(x, u))$. Then define

$$Q^L(\theta, x, u) = \sum_{x'} p(x'|x, u) \left(\max_{u' \in U(x')} Q^{L-1}(\theta, x', u') \right)$$

Since computation of Q^L calls Q^{L-1} once for every possible action in every possible successor state, computing it is order $O((SB)^L)$ where S is the number of successor states (always 7 in Tetris), and B is the expected branching factor, or mean number of moves available in a given state. What is the expected branching factor of Tetris?

- The square is symmetric under rotation, and it can be placed in 9 positions.
- The line has two distinct rotations (horizontal and vertical). In the horizontal position, it can be placed in 7 positions, and in the vertical it can be placed in 10 positions.
- The S, Z, L, and J pieces all have two distinct rotations. In their horizontal rotation, they can be placed in 8 positions, and in their vertical rotation, they can be placed in 9 positions.
- The T piece has four distinct rotations. In either horizontal rotation, it can be placed in 8 positions. In either vertical rotation, it can be placed in 9 positions.

So, the mean branching factor is

$$\frac{1}{7}\left(9 + (7 + 10) + 4 * (8 + 9) + (8 + 8 + 9 + 9)\right) = 18$$

Hence, we expect that computing Q^1 incurs an additional factor of $7 \cdot 18 = 126$ board evaluations over computing Q^0. Computing more lookahead is exponentially more expensive: to compute Q^3, we increase computation by a factor of $126^3 = 2,000,376$!

By simply using Q^1 instead of Q^0, we saw much better average performance. Averaging over 10 games with the weights learned in section 5, we observed a mean score of 83,222 lines with a maximum of 161,223. Unfortunately, since evaluating the lookahead policy takes a very long time, we decided not to run more trials.

7 Max-Margin Learning from Demonstration

If we observe a game being played by an expert, we can learn a policy based on the expert's actions. Let u^+ be the action that an expert chose for state x. We wish that the expert's action appears to be the best in a maximum-margin sense under our estimator Q. That is, for all actions $u \in U(x)/\{u^+\}$, we want the expert's action to be the best by a margin c:

$$Q(\theta, x, u^+) > Q(\theta, x, u) + c$$

This constraint is obviously still met if we beat the maximizer u^-:

$$u^- = \underset{u \in U(x)/\{u^+\}}{\arg\max} \; Q(\theta, x, u)$$

And so the constraint is

$$Q(\theta, x, u^+) > Q(\theta, x, u^-) + c$$

Learning θ can be accomplished via a stochastic online subgradient update. On every step of a game observed, we can verify if the constraint above is met. If not, we compute the margin loss ϵ for the current step:

$$\epsilon = Q(\theta, x, u^-) - Q(\theta, x, u^+) + c$$

Then we update the weights as follows:

$$\begin{aligned} \Delta &= \epsilon\Big(\nabla Q\big(\theta, x, u^+\big) - \nabla Q\big(\theta, x, u^-\big)\Big) \\ \theta &\leftarrow \theta + \gamma\Delta \end{aligned}$$

Where $\gamma \in (0, 1]$ is a step size.

8 Max-Margin Lookahead Policy Optimization

By combining the max-margin learning from demonstration along with a lookahead policy we can learn better policies.

- *Conjecture 1:* given a reasonable estimator Q^0, the policy imposed by Q^1 is better than the original Q^0 policy.

- *Conjecture 2:* learning a new $Q^{0'}$ by treating Q^1 as an expert gives better performance than the original Q^0 policy.

If these conjectures are true, then the following algorithm should work: Start with a reasonable policy Q^0. Play one game (or even a fraction of a game - we use a maximum of 10,000 moves) under Q^1. Train a new policy $Q^{0'}$ with the max-margin learning from demonstration algorithm by treating Q^1 as an expert. Repeat until performance is good.

Starting with the policy learned in section 5, we did this for four iterations. The resulting policy significantly outperforms the original policy: on 100 games, the maximum score observed was 92,248 lines with an average score of 16,208 lines.

This method seems to confer some promising advantages. Unlike REINFORCE, the policy need not be stochastic. Using a deterministic policy can be faster because the stochastic policy slows convergence – learning only occurs when the policy is rewarded for relatively low-probability actions. This method is also better than running a pure two-step lookahead policy because the lookahead policy only needs to be run for a finite number of moves in each iteration, and the final policy runs much faster than a two-step lookahead.

9 Learning Features

Choosing good features to represent a problem is a central issue in machine learning. To solve this problem, there are several techniques for automatically discovering useful features from data. We hoped to improve the performance of our linear Q-function value estimator by training it with more informative features than those described above. Our approach uses *unsupervised learning* where only data is given, without any sort of label. This learning paradigm excels in areas such as document classification where abundant data is available, but generating labels for data is costly in some way [4]. By recording game states as our basic learner plays, we can quickly generate millions of unlabeled Tetris states. Assigning a label to these states would be difficult, because the true value function over states is not known. By observing our basic Tetris learner, we sample a distribution over states which is typical for a player acheiving thousands of completed lines per game on average.

Given the unlabeled Tetris data, we treat the board as a vector of binary pixels, and seek to discover useful real-valued features. We use a fully connected neural

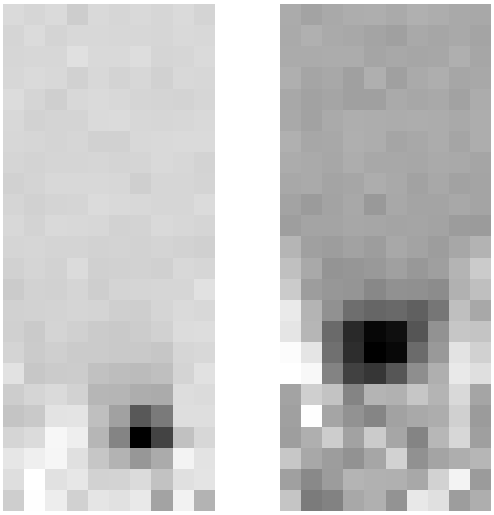

Figure 1: Weights learned by two of the hidden nodes during RBM unsupervised feature learning. The features appear to capture information about holes/wells, as well as overall board height.

network with a single visible hidden layer. Each node in the network is binary and stochastic, where the probability of activation is defined by the Boltzmann distribution:

$$p(s_i = 1) = \frac{1}{1 + exp(-b_i - \Sigma_j s_j w_{ij})}$$

Where w is the symetric weight between node s_i and s_j, and b_i is the bias for s_i. This class of neural network is known as the restricted Boltzmann machine (RBM). The contrastive divergence algorithm (CD) was introduced by Geoff Hinton [3] as a way to learn features in the hidden layer of an RBM with unlabeled data. We applied the CD algorithm to Tetris board states.

Figure 1 shows weights for 2 of the 50 learned features. The features correctly capture information about the overall height of the stack, and holds. However, the 50 learned features do not store additional information beyond what is available by using the features described previously (e.g. row height). After trying several variants of this procedure, we concluded that although the number of possible Tetris states is large, the structure of the game causes states to be represented well by the hand-designed features commonly used.

10 Conclusion

Overall, we tried a number of strategies to learn Tetris policies. Our favorite policy is the one learned in section 8 – the combination of max-margin learning from demonstration with a Q^1 lookahead policy.

Strategy	Max	Average
Covariant REINFORCE	31,705	4,331
Q^1 Lookahead	161,223	83,222
Max-margin + lookahead	94,248	16,208
Learned features	18	4

We would like to further investigate the properties of the algorithm outlined in section 8, and to see if we can prove the conjectures outlined there.

References

[1] J. Andrew (Drew) Bagnell. *Learning Decisions: Robustness, Uncertainty, and Approximation*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 2004.

[2] J. Baxter and PL Bartlett. Direct gradient-based reinforcement learning. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 3, 2000.

[3] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, July 2006.

[4] Kamal Nigam, Andrew Mccallum, Sebastian Thrun, and Tom Mitchell. Text classification from labeled and unlabeled documents using em. In *Machine Learning*, pages 103–134, 2000.