16-899C ACRL
Tetris Reinforcement Learner

Alex Grubb Stephane Ross Felix Duvallet

February 3, 2009

1 Introduction

Tetris is a game in which blocks (tetraminos) are placed one at a time. Completed lines are removed, allowing the game to continue. This report outlines our approach to using reinforcement learning on this problem.

2 Approach

Our approach to this problem was to use reinforcement learning with a function approximator to approximate the state value function [RSS98]. In our case, a +1 reward was given for every completed line, so that the value function would encode the long-term number of lines that is going to be completed by the algorithm. In order to achieve this, we extract features from the game state, and use gradient descent to update the parameters of our function approximator.

Hence our approach is similar to Q-learning with a function approximator [RSS98], however contrary to Q-learning we learn only the state value function rather than the state-action value function. This is possible in our case since we can easily compute the exact immediate rewards and the exact next state of the game given the current state and the action executed by the agent. Learning the state value function is more practical since it can be represented with fewer parameters than the state-action value function, allowing our approach to learn faster.

2.1 Value Function Update

For our tetris agent, we used a simple policy based on the estimated value for each state and a gradient descent rule for updating this estimate. We seek to estimate the value function $V : S \to \mathbb{R}$ mapping states to an expected time-discounted reward, and use this to determine the policy for our agent.

Given an estimate on the value function V, our optimal policy is simply the action maximizing expected value:

$$\pi(s) = \underset{a}{\text{argmax}}[R(s,a) + \gamma V(T(s,a))] \tag{1}$$

where γ is the discount factor and T is the deterministic transition function that gives us the next state.

Since tetris is deterministic and we try to learn the value function of the optimal policy, we must have that, for each state s, the value function satisfies:

$$V^*(s) = \max_a [R(s,a) + \gamma V^*(T(s,a))]. \tag{2}$$

Thus if we have an estimate V, we can compute the Bellman error δ_s of this estimate at state s as:

$$\delta_s = V(s) - \max_a [R(s,a) + \gamma V(T(s,a))] \tag{3}$$

If V is represented by a set of parameters $\{w_i\}_{i=1}^n$, we can update w_i using gradient descent [RSS98]:

$$w_i' = w_i - \alpha \delta_s \frac{\partial V}{\partial w_i}; \tag{4}$$

for some small constant α, where α represents the learning rate. Our approach consists in updating the weights whenever the agent acts in state s. The action executed by the agent in that state is determined by the agent's exploration strategy. We describe the different exploration policies we have considered later. We now present the different features we used to describe a state and the several function approximators we have tried to represent V.

2.2 Features

The state in the game of tetris is described as a 20×10 matrix of binary variables indicating whether or not a block is present at each of these positions on the board. Since these binary features are not very informative by themselves, it would be very hard to learn a value function straight from these features. Instead we transform these state features into more informative features that are better at representing the value function.

The features we used are listed below, most of these features were found in the litterature. Features with a * are vectors.

- **Column Heights***: The heights of all the columns. [DPB96]
- **Max Height**: The tallest column. [DPB96,NB05]
- **Min Height**: The shortest column (or empty).
- **Max-Min**: Difference of max and min. [NB05]
- **Holes**: Number of holes (empty space with at least a block above it). [DPB96,NB05]

- **Blocks**: Total number of blocks on the board. [NB05]
- **Column Differences***: Absolute value of difference between neighboring columns. [DPB96]
- **Maximum Well Depth**: Deepest "well". A well is a column of blocks which is between two higher columns of blocks. The depth of the well is measured as the height of the smallest of these two columns minus the height of the short column. [NB05]
- **Sum of Wells**: Sum of all well depths. [NB05]
- **Row transitions**: All transitions (empty to full or full to empty) across the rows. The outer region is considered full. [NB05]
- **Column transitions**: Same as above, across all columns. [NB05]
- **Constant bias**: A constant feature always equal to 1.

All of these features were normalized so they would all take real values in the interval $[0, 1]$. Once these features were extracted from the state. We represented the value function as a function of these features using a function approximator.

2.3 Function Approximator

Since it is hard to know in advance the shape of the value function as a function of the features, we have tried several function approximators: a polynomial approximator, a RBF approximator, a neural net approximator and a linear-exponent-displacement approximator. These are described in detail below.

2.4 Polynomial Approximator

Given the vector of features f representing a state s, the polynomial approximator approximates the value function as a polynomial of those features. Here we do not consider cross-terms, so that if we decide to use a polynomial of order k then the value function is approximated as:

$$V(f) = \sum_{i=1}^{n} \sum_{j=1}^{k} w_{i,j} \left(f_i\right)^j \tag{5}$$

where $w_{i,j}$ are the weights that we need to learn and f_i represents the i^{th} feature in vector f.

For this particular function approximator, the gradient descent update rule specifies that we should update the weights as:

$$w'_{i,j} = w_{i,j} - \alpha \delta_f \left(f_i\right)^j . \tag{6}$$

Initially the weights $w_{i,j}$ were all initialized to 0.

2.5 RBF Approximator

The RBF approximator [RSS98] instead tries to approximate the value function by a set of radial basis function (RBF) in feature space. Let c be a feature vector and Q be a symmetric and positive definite matrix, then a RBF parametrized by c and Q is defined as $\exp(\frac{-1}{2}(f-c)^T Q^{-1}(f-c))$, where f is the feature vector at which we evaluate the RBF. Hence a RBF can be thought of as a gaussian centered at c with covariance matrix Q. Thus if we use k RBF to represent the value function, the value function is approximated as:

$$V(f) = \sum_{i=1}^{k} w_i \exp(\frac{-1}{2}(f-c_i)^T Q_i^{-1}(f-c_i)). \tag{7}$$

For this approximator, we only tried to learn the weights w_i. c_i and Q_i were initialized randomly at the beginning, such that they would be within our feature space. We also restricted Q_i to be a diagonal matrix so that it would be easy to invert.

Hence, for this particular function approximator, the gradient descent update rule specifies that we should update the weights as:

$$w_i' = w_i - \alpha\delta_f \exp(\frac{-1}{2}(f-c_i)^T Q_i^{-1}(f-c_i)). \tag{8}$$

2.6 Neural Net Approximator

The Neural Net approximator tries to learn a neural net [TM97] which takes the features as input, and outputs the value function. Here we have used sigmoid neurons and train a single hidden layer. If we use k neurons in the hidden layer, this is equivalent to learning a value function of the form:

$$V(f) = \sum_{i=1}^{k} v_i \sigma(\sum_{i=1}^{n} w_i f_i). \tag{9}$$

where σ is the sigmoid function and w_i and v_i are weights that must be learned. The backpropagation algorithm is used to train those weights using the observed errors δ_s [TM97].

2.7 Linear-Exponent-Displacement Approximator

The Linear-Exponent-Displacement (LED) approximator is the combination of the three function approximators used in [NB05]. It tries to learn a value function of the form:

$$V(f) = \sum_{i=1}^{n} u_i f_i + v_i \left(f_i\right)^{a_i} + w_i |f_i - d_i|^{b_i}, \tag{10}$$

where f_i are the features, u_i, v_i, w_i are linear weights, a_i and b_i are exponents, and d_i are displacement factors.

Contrary to [NB05], we use gradient descent to update these parameters, rather than a genetic algorithm. Hence in order to update the parameters, we need to compute various partial derivatives of (10):

$$\begin{aligned}
\frac{\partial V}{\partial u} &= f_i \\
\frac{\partial V}{\partial v} &= (f_i)^{a_i} \\
\frac{\partial V}{\partial w} &= |f_i - d_i|^{b_i} \\
\frac{\partial V}{\partial a} &= v_i (f_i)^{a_i} \ln [f_i] \\
\frac{\partial V}{\partial b} &= w_i |f_i - d_i|^{b_i} \ln (|f_i - d_i|) \\
\frac{\partial V}{\partial d} &= w_i e_i |f_i - d_i|^{e_i - 1} \text{sign} (d_i - f_i)
\end{aligned}$$

where

$$\text{sign} (d_i - f_i) = \begin{cases} 1 & \text{if } d_i > f_i \\ -1 & \text{if } d_i < f_1 \\ 0 & \text{if } d_i = f_i \end{cases} \tag{11}$$

Combining these partial derivatives with the general gradient descent update rule (Equation 4) specifies how these parameters were updated. Initially, u_i, v_i, w_i and d_i were initialized to 0, while the exponents a_i and b_i were initialized to 1.

2.8 Exploration

Our exploration strategy was a modified ϵ-greedy strategy [RSS98]. The best action according to our current estimate V (as computed in Equation 1) is picked with probability $1 - \epsilon$, and a random action is picked with probability ϵ. Our ϵ was a function of time:

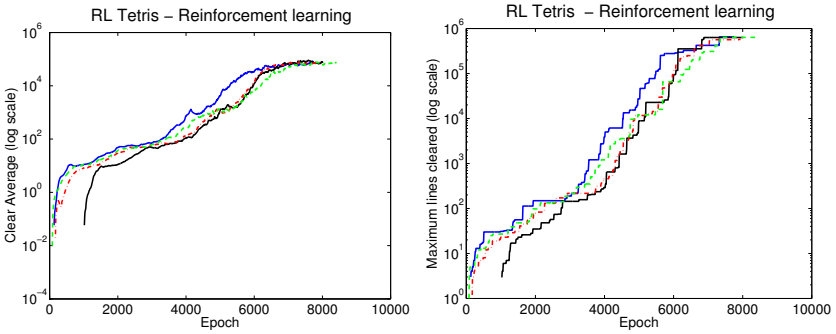
$$\epsilon (t) = \epsilon_0 \beta^t \tag{12}$$

where ϵ_0 is the initial exploration rate in $(0, 1)$ and β was a decay rate in $(0, 1)$. This allowed for more exploration early in the learning, and more exploitation in later epochs.

We also considered Boltzmann exploration [RSS98], but tuning the temperature and decay parameters correctly was a problem, and it did not do the right action when required.

3 Results

Initial trials yielded the best results using the full feature vector (Section 2.2), the LED approximator (Section 2.7), and epsilon-greedy exploration (Sect-



(a) Clear Average (b) Clear Max

Figure 1: Learning Performance for 4 trials

Table 1: RL Performance during learning.

Epoch	Average Score
1000	7.6
3000	60.4
4000	306.4
5000	1947.4
6000	20094.4
7000	61794.7
8000	71821.3

ion 2.8). We have used this configuration to generate the results presented here.

Each epoch plays a game until the agent loses. To evaluate the performance of each algorithm, we looked at both the maximum number of lines cleared (Clear Max) in a single game by each algorithm and a discounted average number of lines cleared by each algorithm (Clear Average) over all games played during learning. The discounted average discounts more performance in the past so that the discounted average at Epoch t reflects more the actual average performance of the current policy at time t. The discounted average was computed as follows:

$$A_t = 0.99A_{t-1} + 0.01c_t \tag{13}$$

where A_t is the discounted average at Epoch t and c_t is the number of lines cleared at Epoch t. This is equivalent to $A_t = 0.01 \sum_{i=1}^{t} 0.99^{t-i} c_i$.

The Clear Average and Clear Max for the LED approximator over 4 different simulations of 8000 epoch are shown respectively in Figures 1a and 1b. For these results, we used a learning rate $\alpha = 0.01$, an initial exploration rate $\epsilon_0 = 0.05$, a decay rate $\beta = 0.9997$ and a discount factor $\gamma = 0.9$, to encourage

the algorithm to clear lines now rather then later. The algorithm reaches a peak Clear Average performance of 88316 lines per game with an overall maximum 675389 lines cleared. The performance of the algorithm, both in terms of average and maximum lines cleared, improves exponentially as a function of the number of epochs and the performance was still improving after 8000 epochs. Hence we believe that given more training time the algorithm could have achieved even better performance. However training becomes very time consuming after so many epochs as the algorithms clears over 80000 lines on average per game. Table 1 shows the average number of lines cleared during learning.

Every 500 epochs, we evaluate the current policy by using it for 50 trials with no learning. Table 2 shows the maximum number of lines cleared by the best current strategy during this evaluation phase. The performance during the evaluation is similar to that of the learning phase. The evaluation performed poorly at epoch 8000, this could be due to a relatively smaller number of games played during evaluation (50 games only).

Videos of the learned policy performing after 1000, 4000, and 8000 games can be found online at `http://www.cs.cmu.edu/~agrubb1/acrl/hw1/`.

Table 2: Maximum lines cleared during evaluation

Epoch	Maximum Score
1000	35
3000	274
4000	4855
5000	21100
6000	58189
7000	79497
8000	37802

4 Conclusion

In this report we have presented a reinforcement learning approach to solving Tetris using function approximators and gradient descent to optimize their parameters. Much more experimentation could be done with other types of function approximators to find a better parametrization of the value function. However the LED approximator seemed to give very decent results. An obvious extension to this approximator would be to add more exponents terms so that it could potentially learn a better fit to the value function. Furthermore, new features could also help the LED approximator to learn a better representation of the value function. However, most features that we found in the litterature were used here, so that new ideas would need to be proposed in this area.

References

DPB96 Neuro-dynamic Programming. D. P. Bertsekas and J. N. Tsitsiklis. Athena Scientific. 1996.

RSS98 Reinforcement Learning: An Introduction. R. S. Sutton and A. G. Barto. The MIT Press. 1998.

NB05 An evolutionary Approach to Tetris. N. Bohm, G. Kokai and S. Mandl. The 6th Metaheuristics International Conference (MIC). 2005.

TM97 Machine Learning. T. Mitchell. McGraw-Hill Education. 1997.