Temporal Difference Learning in the Tetris Game

Hans Pirnay, Slava Arabagi

February 6, 2009

1 Introduction

Learning to play the game Tetris has been a common challenge on a few past machine learning competitions. Most have achieved tremendous success by using a heuristic cost function during the game and optimizing each move based on those heuristics. This approach dates back to 1990's when it has been shown to eliminate successfully a couple hundred lines. However, we decided to completely depart from defining heuristics for the game and implement a "Know-Nothingism" approach, thereby excluding any human factor and therefore error/bias/pride in teaching the computer to play efficiently. Instead we use a combination of value iteration based on the Temporal Difference (TD) methodology in combination with a Neural Network to have the computer learn to play Tetris with minimal human interaction.

2 Theory

2.1 Temporal Difference Basis

The bulk theory is largely based on the general TD methodology and its implementation in the game of Backgammon [1]. We define an experience tuple $< s, a, r, s' >$ at each move representing the current state (s) of the field configuration, the action that is taken in the state (a), the reward obtained for the move (r) and the next state (s'). The state is defined as the configuration of the playing field. The function $V(s)$, representing the value function of a state s, is defined as the discounted value of all rewards attained from a given state onward if the optimal strategy is chosen. Intuitively, the value function represents how strategically favorable is for the game to achieve that state. Note, that we are not trying to minimize or maximize the function $V(s)$ but merely to estimate it, implying that the game itself possesses a true value function $V_{true}(s)$ intrinsic for the game, the goal for the learning algorithm is to converge to that function.

The methodology of convergence to that function is based on the state update step. Suppose the playing field is in a configuration and a Tetris piece is generated for a new move, for each possible action of placing the new piece a, the value function $V(s')$ is evaluated based on the configuration of the playing field following that action. The optimal move is then chosen as $\underset{a}{\operatorname{argmax}}\,(r + V(s'))$, denote it $V^*(s')$. Once the best next state has been chosen the value function of the current state,

$V'(s)$ is updated based on the Sutton TD formula [2]:

$$V'(s) = V(s) + \alpha(r + \gamma V^*(s') - V(s)), \tag{1}$$

where the expression $r + \gamma V(s')$ represents the estimated value function of the next state, and α is the learning rate. The $\gamma < 1$ parameter expresses that there is uncertainty in the evaluation of the next state and the algorithm shouldn't fully trust it. The reward is defined to be high when a row is eliminated, and negative when the game is lost. The value function is only updated based on the most optimal next action. In such a manner, the estimated $V(s)$ should increase for states that can result in an eliminated line and greatly decrease for the actions that result in loss.

2.2 Mapping States to Values via a Neural Network

The value function role of mapping states to a value representing their progress toward eliminating a row is done via a Neural Network, similar to the one implemented in [1]. The details of the neural network are presented in below sections. The general algorithm of 1 is implemented in the neural network in the following manner. At each move of the game, after the optimal state $V^*(s')$ has been calculated, the neural network weights are updated by the equation:

$$w_{m+1} - w_m = \eta \sum_{k=0}^{m} \lambda^{k-m} \nabla net_w^k, \tag{2}$$

where η is the network learning rate and ∇net_w^m is the gradient of the network output with respect to the neuron weights at move m. This step is a modified back-propagation routine for training neural networks. The improvement lies in modifying the direction of the gradient step, essentially instead of only considering the the gradient of the latest step ∇net_w^m, the algorithm keeps track of all the previous descent directions and scaling them by λ^{k-m} incorporates them in the descent/ascent direction. This combinatorial scaling of the descent directions is accomplished by a recursive function e of the following form:

$$e_{m+1} = e_m \lambda + \nabla net_w^m. \tag{3}$$

In the above equation, λ is the effective "momentum" of the computation. The value of e_m is maintained together with the weights w_m of each neuron and updated every time during the learning state in a recursive fashion. The network is updated after every move and hence it is trained at the same rate. The back-propagation step is performed for one iteration only when a new input-output state is provided.

In general, the described approach relies heavily on the ability to construct the value function using the neural network. Because the neural network only learns the states s that the game achieves, in order to converge to $V_{true}(s)$, the algorithm needs to be in every state at least a few times, implying that the game is required to be played many thousand, or even million, times. As we wanted as little human intrusion in the game as possible, no initial strategies were preset into the policy chooser, instead a randomization routine was implemented ensuring that the algorithm explored different moves earlier in the stages of training. Even with this exploration breadth addition, we expected a few million games to be played before the value function for each state converges.

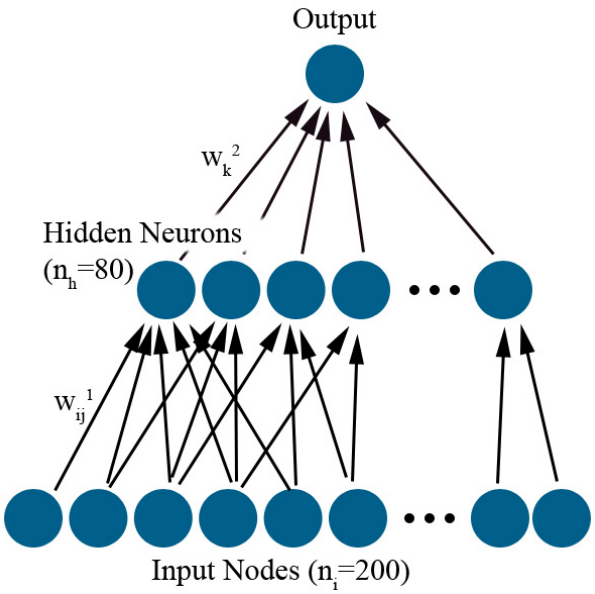

Figure 1: Configuration of neural network used for game learning.

3 Implementation

In our implementation we used the Java Tetris simulator provided by Eric Whitman.

3.1 Implementation of the Neural Network

Since all available free implementations of neural networks are designed for batch learning, we implemented our own feed-forward network. The input to the network is the raw encoding of the state which consists of 200 squares that can either be on (1) or off (0). The neural network, portrayed in Fig. 1, maps this input to a single decimal number as output, representing the value function $V(s)$. Since our input and output structure is very similar to that of TD-Gammon (200 input neurons for Tetris as compared to 198 in TD-Gammon), we used 80 hidden units in one single hidden layer, alike [1]. The next piece was not included in the state information.

For the structure, the neurons are fully interconnected with both their preceding and succeeding layer. The output functions of the neurons for the input and output layer are identity mappings. The hidden layer implements the sigmoid function:

$$s(x) \quad = \quad \frac{1}{1+\exp(-x)}. \tag{4}$$

The network is trained by the on-line modified back-propagation routine after each move as described above.

3.2 Modeling Reward

Modeling the reward was accomplished as portrayed in table 1. The reward for each line is an obvious choice, since lines are the main goal of the game. The reward for each piece set was introduced to get around a very common problem that occurs in the beginning stages of every game, when pieces are practically set by chance. The Tetris game would build up all pieces in the leftmost column until the very top, where the cost of "game-over" forced it to place pieces to fill up the empty space, thereby delaying loosing and eliminating rows. Therefore, the first favorable states that the algorithm experienced were the ones that had a lot of holes at the bottom, and were built up to the very top. The reward for each line was meant to encourage a more even build up of lines right from the start, overall increasing the probability of line elimination.

Event	**Reward**	**Reason**
each move	+1	to learn from the startout
each deleted lin	+5	the ultimate goal
game over	-10	an additional penalty

Table 1: Rewards.

3.3 Exploration

Exploration is a key ingredient for unsupervised learning. On the other hand, building the value function is based on the assumption that the best possible moves are made. Thus the two goals of approximating the value function correctly and exploring the state space have to be weighted against each other. This is done by sampling the next move according to the quality assigned to it by the value function. Define S in the following manner:

$$S(l) \equiv \sum_{k=0}^{l} \left[V(s(a_k)) - \min_{r \in A} V(s(a_r)) \right], \tag{5}$$

where $s(a_k)$ is the state that is reached after taking action a_k, and A is the set of all possible actions. Then the exploration rule is: For a random number $R \in [0, 1]$, select action j, if

$$\frac{S(j-1)}{S(n)} \leq R < \frac{S(j)}{S(n)}. \tag{6}$$

Intuitively the equation portrays that the move is sampled from the list of all possible moves according to their improvement over the (conceived) worst possible move.

4 Results

The achieved results with this algorithm were somewhat unfavorable. We do, however, still believe strongly in the philosophy of Know-Nothingism and discuss the encountered computational and modeling-related issues below.

4.1 Numerical Results

The results stated here were achieved using the parameters listed in table 2.

Parameter	Value	Use
η	0.01	step-size / learning rate
λ	0.6	TD-parameter
γ	0.9	Discount factor of future states

Table 2: Parameters used in the algorithm.

Table 3 shows the results for our network after a selected number of training games. N_G is the number of games, N_L is the average number of lines removed per game, and N_E is the number of times a sub-optimal move is chosen for exploration purposes divided by the overall number of moves. The last number, which reflects the S function (eqn. 5, is very interesting because it provides insight about the change of the value function. The decreasing value implies that the function steadily grows more confident about the best move to make.

$N_G[1/1000]$	N_L	N_E
10	0.21	0.86
300	0.45	0.81
700	0.77	0.76

Table 3: Results achieved by the playing algorithm.

4.2 Computational Problems

The major problem, created because of slow implementation of the algorithm in Java and the natural complexity of the Tetris game, was presenting as many training examples to the game as necessary. While TD-Gammon needed 4.5 million games to converge, we only were able to play only 700,000.

4.3 Modeling Errors and Complexity Discussion

Our conceived modeling errors are mostly bound to our commitment to follow the very successful approach of TD-Gammon. Indeed, at first glance, the similarities (table 4) seem very striking.

Property	TD-Gammon	TD-Tetris
Bare state descriptors	198	200
output	1	1
random branching each move	6	7
Problem Dimensionality	10^{20}	$2^{200} \approx 10^{60}$

Table 4: Comparison of TD-Gammon and TD-Tetris.

There are, however, some differences, that make the application of the TD algorithm to Tetris

more difficult than to Backgammon. For one, there is the obvious problem of the much higher dimensionality of Tetris as compared to Backgammon. This problem shall be illustrated by the following numbers: The original TD-Gammon needed 4.5 million games for the neural network to converge. A normal game among humans takes about 50 to 60 moves, but TD-Gammon's initial games took much longer due to the lack of experience. From these numbers, one can estimate the number of moves approximately to 100 moves per game. Because TD-Gammon plays against itself during training, the neural network learns on a total of 900 million half-moves.

Suppose now the most favorable case, that the necessity for training examples or TD-updates scales with $\log_{10}$ of the number of possible states. This implies that Tetris would need 30 times as many updates as TD-Gammon, amounting to $900,000,000 \cdot 30 = 27,000,000,000$ updates. Given that Tetris updates at every move and there are 270 moves per game (which is much more than is reached in the beginning stages), Tetris would need 100,000,000 training games to achieve the same success levels as TD-Gammon. Considering that we only managed to get through 700,000 games, our algorithm has not even completed one percent of the training necessary, and should thus not be judged in unfair comparison.

References

[1] Taken from: http://www.research.ibm.com/massive/tdl.html

[2] Richard S. Sutton, "Learning to predict by the method of temporal differences," *Machine Learning*, 3:9-44, 1988.