LSINF2275 : Project II
Markov Decision Processes meet

Vianney Coppé – 49461400 (INFO2MS)
Group 7

I. Introduction

A. The context of the study

In this study, we are interested in finding good policies for playing the very popular game *Tetris*. Indeed, it is a really interesting game because its rules are really simple yet the number of possible configurations are huge.

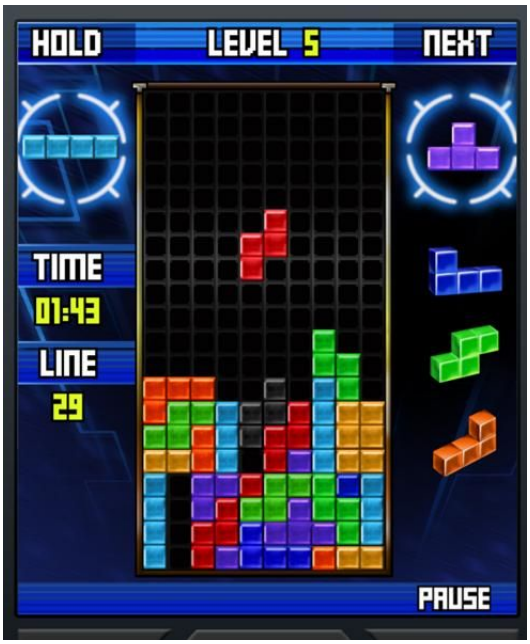

Fig. 1. The famous *Tetris* game.

Tetris is a tile-matching played on a grid of 20 rows and 10 columns. The player has to choose where to place the tiles (which are called the *tetrominoes*, see Figure 2), falling from the top of the screen, by shifting them to the left or right and rotating them. When a row is filled, it is cleared and all the tiles above it go one row down. The goal is to remove the most rows before the given tile cannot be placed on the board.

Here, we are interested in the one tile look-ahead version of the game, which means that given the current board, we only know the tile we have to place next. In some variants of the game, this number can go up to three. However, even in the case that the sequence of pieces is known in advance, finding the optimal strategy is an NP hard problem [1].

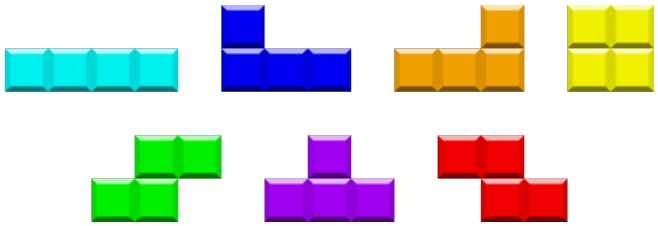

Fig. 2. The seven *tetrominoes* of the game.

B. Objectives

The classical *Tetris* game contains a huge number of configurations. Indeed, the board has 20 rows and 10 columns which means that we can have up to 2^{200} different states. In the Markovian formulation of the problem, a state is defined by the grid and the current tile to place, which means that this huge number is multiplied by a factor 7.

Obviously, the value-iteration algorithm is not tractable in this context because it builds a table containing all the states and their estimated expectation of the score so we will have to investigate other techniques. However, we will first use this algorithm for small instances and try to overcome the state space size problem

by finding other representations of the board. With this knowledge of the problem, we will then apply value-function approximation in order to generalize our results for higher dimensions of the grid and see how they perform comparatively.

II. Theory and modeling

We will first discuss the formulation of the problem and then the techniques used to optimize the decision making.

A. Modeling the problem

As said earlier, each state is identified by a grid and the tile to place. The transitions between the states are defined by all the positions where we can place the tile, we then compute the resulting grid (if some rows are deleted) and for this new grid, we can have any of the 7 tiles to be placed next.

For identifying the states, each tile is numbered from 0 to 6 but for the grid, we can think of several representations :

- a two-dimensional array of *booleans* indicating whether the cell is empty, this is the only exact representation because we may want to know if our action will remove one or more rows at the same time and also, there are some moves which need knowledge of the holes (see example on Figure 3).
- if we take one step back, we can observe that the decisions are done mostly considering the maximum height of each column. As a result, we can approximate the whole grid by a single dimension array containing the height of each column.
- if we pursue this reasoning, we can suppose that the global height of the columns does not impact so much the decisions but only the upper contour of the tile wall is important. The contour is the difference of the height of adjacent columns, which is illustrated on Figure 4.

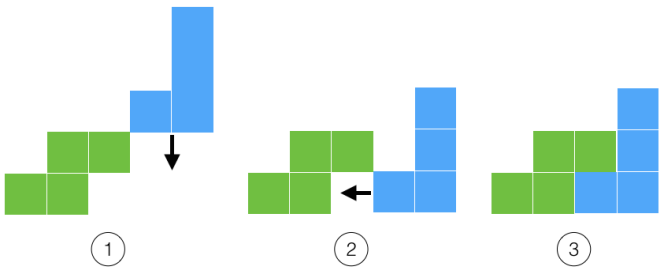

Fig. 3. Shifting the *tetromino* at the last moment.

The ideal grid representation is of course the first one. Unfortunately, since the number of states is a concern, we will have to deal with the latter ones which offer less accuracy but hopefully a decent sight of the game.

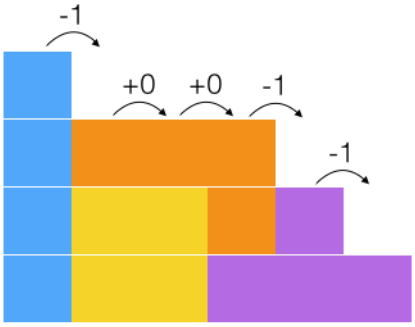

Fig. 4. A game grid and its contour.

B. Value-iteration algorithm

For small boards and/or with the state representations described previously, we can still try to find the optimal policy with the value-iteration algorithm, which is the following :

$$\hat{v}(S_k) = \max_{a \in U(S_k)} c(a|S_k) + \sum_{S_{k'} \in S} p(k'|k,a)\hat{v}(S_{k'})$$

with $\hat{v}(S_d) = 0$, S_d being the destination state. We reformulated a little the original recurrence since we are facing a maximization problem and also to keep the same notations for both algorithms. We apply the recurrence for each state until global convergence.

C. Value-function approximation

With this experience, we can now design a value-function by selecting meaningful features of the game. Here, we chose :

- the contour of the board as explained in II-A.
- the number of holes in the tile wall.
- the maximum height of a column.
- the minimum height of a column.
- the average height of the columns.

We combine these features in a unique linear function which is simply the inner product between the feature vector and a weight vector $\mathbf{w}$ with the same number of components.

On-policy learning implies the use of a reward function which appeared to be decisive for the quality of the value-function approximation. We considered a simple reward function, a weighted sum of two features :

- the difference of the average height of the columns in order to encourage the global height of the tile wall to decrease.
- the difference of the number of holes : again, we try to prevent our policy from creating a lot of holes.

We can now use an algorithm to find the weight vector which best represents the value of a given state. The n-step semi-gradient TD [2] seemed appropriate because actions can have large consequences several turns after. In addition, we can simply set the time difference to zero in order to get a more greedy approach.

This algorithm uses simulations of the game in order to learn the policy. We thus fix an amount of games to be played and then for each game, starting from an empty grid at time $t \leftarrow 0$ with $T \leftarrow \infty$:

1) Randomly select a tile.
2) Take an action according to the current policy and store the resulting state S_{t+1} with its reward R_{t+1}.
3) $\tau \leftarrow t - n + 1$
4) If $\tau \geq 0$, update the weight vector according to :
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(G - \hat{v}(S_\tau, \mathbf{w}))\nabla\hat{v}(S_\tau, \mathbf{w})$$
where α is the learning rate,
$G = \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$,
γ a decay factor, $\hat{v}(\cdot)$ is value-function and $\nabla\hat{v}(\cdot)$ its gradient.
5) $t \leftarrow t + 1$
6) Go to 1) if $\tau < T - 1$

The crucial step is the fourth one, where we build an estimate G of the value of the state S_τ given the next n rewards and the state reached $S_{\tau+n}$. We then update the weight vector $\mathbf{w}$ based on the difference between this estimate and the current value of the state S_τ multiplied by the gradient.

There are a lot of parameters to play with in this algorithm, namely :

- the learning rate α : it controls the weight of the adjustments made to the weight vector $\mathbf{w}$, we chose $\alpha(k) = e^{-k/C}$ where k is the number of episodes already ran and C a constant.
- the time difference n : the amount of time we will look ahead.
- the decaying factor γ : it determines how the future rewards will affect the estimate of the value.

and we also decided to add a parameter ϵ which is used at step 2) of the algorithm. We take a random number between 0 and 1, if it is greater than ϵ then we use the best action according to the current policy, otherwise we pick a random action. It helps to explore a bit more of the state space. As for the learning rate, we want ϵ to decrease w.r.t. the number of episodes already completed and we chose $\epsilon(k) = \frac{1}{1+C\ln k}$.

D. Implementation

The game logic is contained exclusively in the classes `Tile` and `Field` whereas the learning algorithm are implemented in the class `Tetris` and `State`.

The class `Field` has several important functions which the algorithms have to call :

- `positions` returns the list containing the positions where a tile can be placed in the grid with the corresponding orientation.
- `successor` returns the updated grid, the reward and the real game gain given a tile, its orientation and the position where it has to be placed.
- `utility` returns the value of the grid given the weight vector $\mathbf{w}$.
- `utility_update` returns the updated weight vector given $\mathbf{w}$, α and G (see step 4 of the value-function approximation algorithm).

The class `State` contains key procedures for both algorithms :

- `mdp_update` updates the estimation of the expectation for the value-iteration algorithm.
- `mdp_move` returns the best action according to the value-iteration result.
- `vf_move` returns the best action according to the current policy given the weight vector.
- `vf_train_move` calls `vf_move` with probability $1 - \epsilon$, otherwise it returns a random action.

With all these help functions, the class `Tetris` can implement learning algorithms :

- `update` runs through all the states in memory and updates their estimated expectation.
- `optimize` performs the value-iteration algorithm by calling `update` until the expectations have converged with a total tolerance of 10^{-6}.
- `episode` implements the n-step semi-gradient TD algorithm for one episode (i.e. until the game is lost in the simulation).
- `learn` calls `episode` a fixed number of times while updating the values of the parameters α and ϵ.

Last but not least, after having trained a policy, it is possible to watch it play with the function `play` in the class `Tetris`.

III. Experiments and simulations

We will now evaluate the performances of our algorithms with two experiments :

1) we will compare the value-iteration (MDP) result with the value-function approximation (VFA) for small grids.
2) since the value-iteration is not tractable for large grids, we will compare the value-function approximation policy with three baseline approaches.
3) we will draw an experimental learning curve of the value-function approximation algorithm.

A. Baseline strategies

The first baseline approach (RND) is to take a random action and should perform really bad since *Tetris* is a puzzle game. The second one (LOW) is already more convincing : we will take the action which places the tile the lowest. Finally, the strategy (HOL) is to take the action that leads to the smallest number of holes.

B. Experimental setting

For our first experiment, we will use a 4x5 board without the I tile (the first one on Figure 2) because it is almost free score in such a grid, and for the second, the board of original size : 20x10. As the game has a big random factor, we will compare the algorithms in parallel (with the same tiles falling) and will repeat the process a thousand times for the first one and for the second a hundred times since the games last longer.

For the second experiment, we will also repeat the whole test 15 times in order to run the VFA learning process each time and compare with the baseline strategies according to the average on these 15 runs, because the learning phase is impacted by the random simulations it uses.

We will assess the performance of the strategies based on a custom game score : the player earns points whenever he removes lines following the formula $100 \cdot l^2$ with l the number of rows cleared at the same time.

C. Experimental results

TABLE I
PERFORMANCES ON A 4X5 BOARD.

	Average score
MDP	577.946
VFA	380.253

In Table I, we can clearly observe that the MDP policy outperforms the VFA. The first one removes almost 6 lines on average whereas the latter generally removes only almost 4.

TABLE II
PERFORMANCES ON THE ORIGINAL 20X10 BOARD.

	Average score
VFA	6052.44
LOW	1175.11
HOL	780.89
RND	13.78

In Table II, we can see that VFA outperforms all baseline approaches. Special mention to LOW, which obtains good results despite a very simple strategy.

On Figure 5, we can observe that the value-function approximation performance increases a lot during the first 20 episodes and then the scores vary between 5000 and 8000 (we did not try for larger number of episodes due to the computation time).

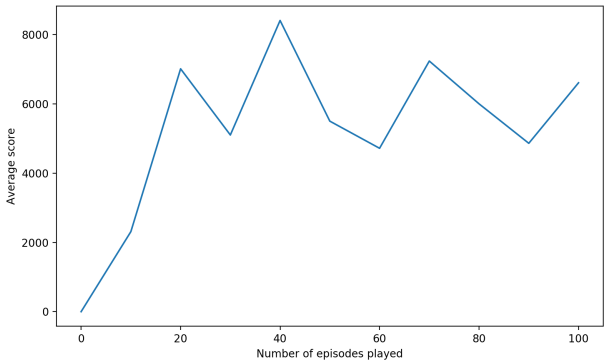

Fig. 5. Learning curve of the VFA.

D. Discussion

From the first experiment, we know that MDP outperforms VFA which is perfectly normal. Indeed, MDP computes the optimal policy based on an exact representation of the game. On the contrary, VFA uses a partial representation (see II-C) and with knowledge gained from a very small experience of the game (only 50 episodes played). Thus, it is quite impressive that the difference is only of 2 lines.

The second experiment illustrates well the performances which can be achieved by value-function approximation algorithms. It is able to clear more than 60 lines on average while it has been trained on only 50 games. Further tuning of the parameters could greatly improve these results as they were very sensitive to changes.

The empirical learning curve demonstrates that the linear value-function approximation quickly reaches satisfying performance. However, it does not seem to converge to a unique value or it should be computed on way more episodes.

IV. CONCLUSIONS AND FURTHER WORK

Compared to other implementations and techniques applied to *Tetris*, the scores reached are not stunning (see [3] where they remove 51,000,000 lines). Still, the goal was to generalize policies to the 20x10 board which was a success, reaching hundreds of lines cleared, with large room for improvement in the current implementation.

However, since it was a first experience in reinforcement learning, it raised a lot of questions. For instance, should the features used in the value-function approximation be scaled between 0 and 1 : divide the heights by the height of the board, divide the number of holes by the total number of cells, etc. Similarly, should the weight vector be normalized after each update. Furthermore, there were a lot of parameters to tune but, without much experience, it is difficult to chose an appropriate learning rate and to fix all the other parameters (n, γ, ϵ, the reward function and also the number of episodes).

Anyway, it has been a thrilling work because we chose the game *Tetris* without knowing neither its difficulty nor the popularity it has in the approximate dynamic programming world. In addition, the results were already really satisfying and we lost a lot of time watching our agent play.

With more time, it would have been interesting to investigate the methods described in [3]. For instance, Cross Entropy and Classification-based Modified Policy Iteration which, according to the paper, are very promising because they search directly in the policy search space.

REFERENCES

[1] S. H. E. Demaine and D. Liben-Nowell, "Tetris is hard, even to approximate," in *Proceedings of the Ninth International Computing and Combinatorics Conference*, 2003, pp. 351–363.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning : An Introduction*. MIT Press, 1998.

[3] V. Gabillon, M. Ghavamzadeh, and B. Scherrer, "Approximate dynamic programming finally performs well in the game of tetris," in *NIPS*, 2013.