Cubibs, a reinforcement learning approach to Tetris

Lorenzo Tribuiani
lorenzo.tribuiani@studio.unibo.it

June 2023

Abstract

The main aim of this project is to develop an automatic solver for a simplified version of the game *Tetris* using *Reinforcment Learning* Techniques. The project has been partially developed for a previous project involving *Constraint Programming* and, since *Tetris* is a complex game to handle only with CP, a simplified version is proposed where all the block are resticted to be cube or rectangle with a minimum shape of 1x1 and a maximum one of 3x3

Introduction

The following pages reports two different approaches for solving the *Cubics* problem using reinforcement learning. In particular, the following paradigms are used: *Q-Learning* and *Deep Q-Learning*
Altough the *Cubics* game is a simplified version of Tetris, a huge dimensionality of the possible exploration space (in case of constraint programming) is still preserved since the possible moves are not reduced by the simplification applied. For each state there's 180 possible new states (9 possible blocks and 20 possible move, at maximum, for block) leading to $O(180^n)$ possible states for the first n blocks. For this reasons a *Reinforced Learning* approach could improve the performance of the Agent.

1 Q-Learning

This section describes the first approach to the solution of the *Cubics* problem trough the use of **Q-Learning**. In particular, for this specific paradigm, a *q-table* is used. The following sections describes the encoding of the observation and actions, the reward system, train and results.

1.1 Observation Encodings

As described before, the problem has an enormous dimensionality and encoding all the possible observation in a complete way would be impossible. Anyway, using partially complete observation, is still possible to manage the problem using a q-table, in particular we have:

- **Actions**. Given the current state each action could be modelled in two different ways:
  - **player actions**. This representation only consider the possible action a player can make, in this way we only have 3 possible action (*move right*, *move left*, *rotate*)
  - **final position**. This representation consider the final position in witch the block will be placed, with this representation there are twenty different actions, one for each final position with and without rotation

even though the *player actions* seem to be the smaller one, this is, in fact, the biggest. Due to the fact that many observation of the same state needs to be considered (at least one for each step down the blocks does), the number of element the final table needs to store increase drastically. With the *final position*, each state is considered only once, and so this is the representation chosen. Since the possible position in which a block can be placed depends directly from the shape of the block itself, some actions represented in the table are not necessary (i.e., a block with shape $(3, 2)$ can only be placed in the first seven slot of the field with a total of 14 possible final position considering rotation). Removing those actions from the table would lead to an irregular table shape with a more complex exploration system, so, those values are kept in the table and simply marked with a negative reward in case of exploration or exploitation (Sec. ()).

- **Observation**. Each state is basically represented by two different elements:
  - The block to place on the board
  - The board

  Blocks are encoded in a similar way w.r.t the actions. Each block is represented with and integer with a total number of 9 elements (shapes from $(1, 1)$ to $(3, 3)$).
  The board represents the more challenging encoding. At game level, the board is represented as a binary matrix with shape $(20, 10)$ where each element represents a slot of the board. This representation can not be simply mapped with a dictionary as done with the previous ones (to many possible states). In order to have a more efficient indexing system the following representation is used:
  For each column of the board the number of occupied slots is counted and the number is appended into a list. The final result is a 10-digit number with base 21 (the height of the column plus the zero element).
  This representation is not complete (only the number of occupied slot is considered and not the position) leading to a reduction of the number of total possible states. Still, this number is unpractical, since the total number of possible states are 21^{10}. To further reduce the total dimensionality, only the first 3 rows are considered leading to a total of 4^{10} elements

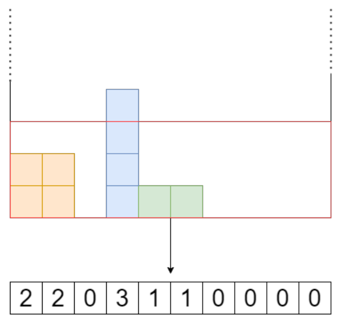

Figure 1: Observation encoding

1.2 Indexing and Q-Table

with the current encoding of states and action the following elements need to be stored into a table:

- 4^{10} different states
- 9 possible blocks
- 20 different actions

to simplify the indexing problem a $3-D$ *numpy* array with shape $(1.048.576, 9, 20)$ is used with a total of 188.743.680 different elements.

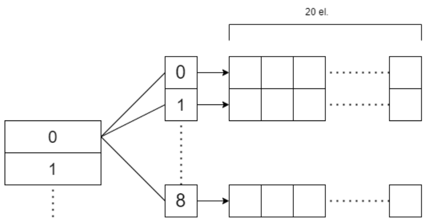

Figure 2: Q-Table Structure

To access an element given the state representation the following process is applied:

1. the 10-digit base 4 number representing the board is converted into a base 10 number.
2. this number is used to index the first dimension of the table
3. the block code is used to access the second dimension of the table
4. the *q-value* or action is indexed accordingly

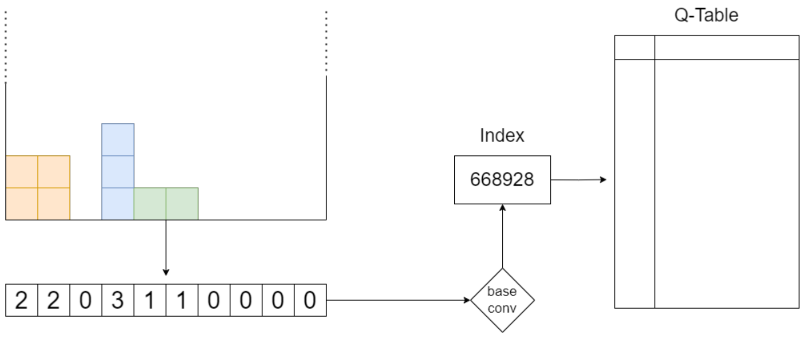

Figure 3: Indexing system

1.3 Reward

The reward system is based on 4 main element:

- $+100$ for each completed line
- -10 for each increment in height, in this way the action is punished only if the height of the new state is greater than the one of the previous
- $+10$ for each new cell contiguously added to an existing one. In this way the agent is pushed to move blocks one next to another avoiding holes and increasing the probability of deleting a line
- -10 for each *unreachable* hole. Unreachable holes are empty slots of the board which cannot be filled because of another block laying on top of it.

finally, if the block can not be placed in the spot selected, the agent chose one of the actions discussed in sec. 1.1, a total reward of -200 is given avoiding the agent to chose impossible moves.

Following the paper *Playing tetris with deep reinforcment learning* another parameters is added to the reward: *Bumpiness.* Bumpiness is the average different in height between the columns of the borad, the final reward function is described as follows:

$$\begin{aligned}100 \cdot del\_lines_t+\\ 10 \cdot (longest\_row_{t-1}-\\ longest\_row_t)-\\ 10 \cdot (height_{t-1}-\\ height_t-\\ 10 \cdot unreachable - bumpiness\end{aligned}$$

1.4 Train

The agent has been trained on different game with a *Epsilon greedy* exploration strategy.

1. Initially the table is initialized with all zeros
2. each time a new state is created the agent access the table and extract the q values for 20 possible actions.
3. those values are summed with random sample from a normal distribution (Numpy randn) multiplied by a factor. In this way, in the initial steps, actions are taken randomly, but, as the table gets updated, the normal distributions becomes smaller and less influent.
4. The action is made and the reward is computed
5. values in the table are updated according to the Bellman Equation

1.4.1 Simulated Training

Given the high dimension of the table, the convergence of the training by *greedy search* to an optimal agent resulted to be extremely slow. In order to have an idea of the performances of the agent a simulated training have been run.
For each element of the table the relative state is created (starting from the index) and each action is then executed and evaluated with the reward system. Finally the q-values are updated. This process, with multiprocessing on 8 different threads required about 4 and a half hours.

1.5 Results

The agent has been evaluated over a series of matches with different levels. Each level differs in how fast the blocks drop down, at level 1 blocks falls at two slot per second, while at level 2 at 4 slot per second.

The agent has been evaluated taking actions in two different way:

- **standard**. The agent has only the observation of the first three rows at the bottom of the board
- **moving window**. The agent evaluates the entire board three rows a time and then chosing the one with the higher value

The following table reports the best result obtained over 10 matches for each level
As we can see, the Agent is less efficient w.r.t. the CP solver, but is more stable over the different levels. This is due to minimum time required to take an action. *Standard* and *Moving window* system didn't seem to have an interesting affect on the overall results

Level	1	2	3	4	5	8	10
CP	10112	6820	2620	140	0	0	0
Standard	200	150	120	210	140	120	150
Moving	300	180	200	70	100	80	160

Table 1: maximum score for CP solver and Q-table

2 Deep Q-Learning

This section describes the second approach for solving the *Cubics* problem using a **Deep q-learning** approach. This code is highly based on the paper Solving Atari games with Deep Reinforcement Learning by *Deep Mind.*

2.1 Network Architecture

The main network used with this paradigm is a *Convolutional Network* with a series of fully connected layers a the end.

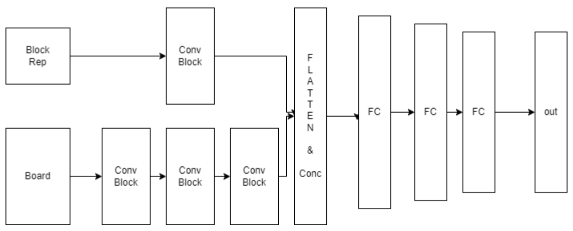

Figure 4: Depp-q Network structure

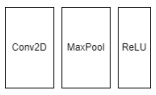

Figure 5: Convolutional Block

- the *Board* is simply passed as a 3D tensor of shape (1,10,20) representing the binary matrix of the board itself
- the *Block* is encoded in a 3D tensor again of shape (1,9,9) representing the binary representation of the block itself

2.2 Reward

The reward system used is the same described in Sec. (1.3) but, as eescribed in Scaling Reward Values for Improved Deep Reinforcement Learning, reducing the magnitude of the reward could bring benefits and improvement to the training of the networks itself. Following this idea the whole reward has been scaled by a factor of 10

2.3 Training and Hyperparameters

The *Deep-q-learning* algorithm has many different parameters to tune and could be modified to maximize the performances of the training itself. the following set of hyperparameters has been used:

- γ : 0.9

- batch size : 216
- learning rate : 5^{-6}
- epsilon decay : 30.000
  This parameters regulate the decay of the epsilon for the epsilon greedy algorithm

Moreover, according to the paper, a *replay buffer* has been used with the following parameters

- buffer size : 50000
- minimum replay size: 1000.
  This parameters indicates the minimum number of element required inside the replay buffer to start the training

2.4 Positive buffer and Repetitions

The base algorithm seemed to be unable to learn efficiently the approximation function with a reward stable around -400. This could be a direct consequence of the way the game works. Tetris require a set of specific move before reaching a positive reward and, using random moves to populate the buffer leads to an extremely high number of negative examples. to mitigate the effect, in the initial phase of the training, the replay buffer is populated only with rewards over a certain *treshold* and thos element are copied into a second buffer called *positive buffer*. During the training itself, if the new state has a positive reward, this is appended in both the replay and positive buffers. if the new state has a negative reward this is append into the replay buffer together with an example from the positive buffer. In this way the distribution of reward inside the buffer is kept constant.
Moreover, since the game has different repetition inside a single match, to eliminate states which are unwanted, we keep each match played during the training up to a certain number of blocks and, as the agent become smarted, we increase the total number of blocks.

2.5 Results

The agent has been trained with an initial number of 5 blocks, increased every 10000 steps and a maximum number of blocks of 30. The new system managed to reach a general reward of 5, able to place the first blocks but still unable to delete any lines. The training seems to have reached a plateau around this point, a possible cause could be the reward algorithm itself.

Level	1	2	3	4	5	8	10
CP	10112	6820	2620	140	0	0	0
Standard	200	150	120	210	140	120	150
Moving	300	180	200	70	100	80	160
Deep-Q	0	10	0	0	0	0	0

Table 2: Comparison over different paradigms