Learning to Play Tetris using Reinforcement Learning

Karol Kuna, 410446, karolkuna@mail.muni.cz

Introduction

The goal of this project was to create an AI agent that learns to play the game Tetris using reinforcement learning, where the agent has to find its playing strategy based on reward signals from the game.

Existing Solutions

Tetris was proven to be NP-complete in its offline version [1], but there exist many feature-based, hand-coded, heuristic or evolutionary methods which reach or exceed human-level performance. There are however fewer attempts at learning Tetris end-to-end using reinforcement learning and they usually don't reach human level [2] [3] without feature engineering.

Tetris

Tetris is a single-player game where a piece called tetromino is falling from the top of the screen. Player can rotate the tetromino, move it (left, right, down), or do nothing. Tetromino falls down by one level every 5 steps, giving player an opportunity to either think or move it as desired.
The objective is to arrange tetrominos at the bottom of the screen so that a horizontal line is created. Once that happens, this line is cleared and player scores a point. The game ends when the stack of tetrominos reaches top of the screen and no new tetromino can appear.

Tetrominos come in 7 different shapes looking like letters I, J, L, S, T, O, Z. The exact sequence of tetrominos is not known in advance. Only the current and next tetromino are known, which makes it an imperfect information game. The playing field is 10 blocks wide and 22 blocks high.

Reinforcement Learning

An artificially intelligent agent is trained to play Tetris using reinforcement learning (RL). Formally, a RL agent observes in each time step t the current state of the environment s_t, chooses action a_t according to its policy, and receives reward r_t. The task is to find such policy that maximises the cumulative reward discounted by factor γ [4].

$$R = \sum_{t=0}^{\infty} \gamma^t r_{t+1}$$

Q-learning

To find the reward maximising policy, Q-learning is used [5]. Let Q be a function which assigns to every state-action pair a quantity $Q(s_t, a_t)$. This quantity represents the cumulative reward agent expects to receive by taking action a_t in state s_t and after that following current policy.
If such function was known, the optimal policy would simply choose action that maximises value of Q in current state — $\arg\max_a Q(s_t, a)$

Q-function can be learned iteratively using temporal difference learning and approximated with a neural network trained by backpropagation.

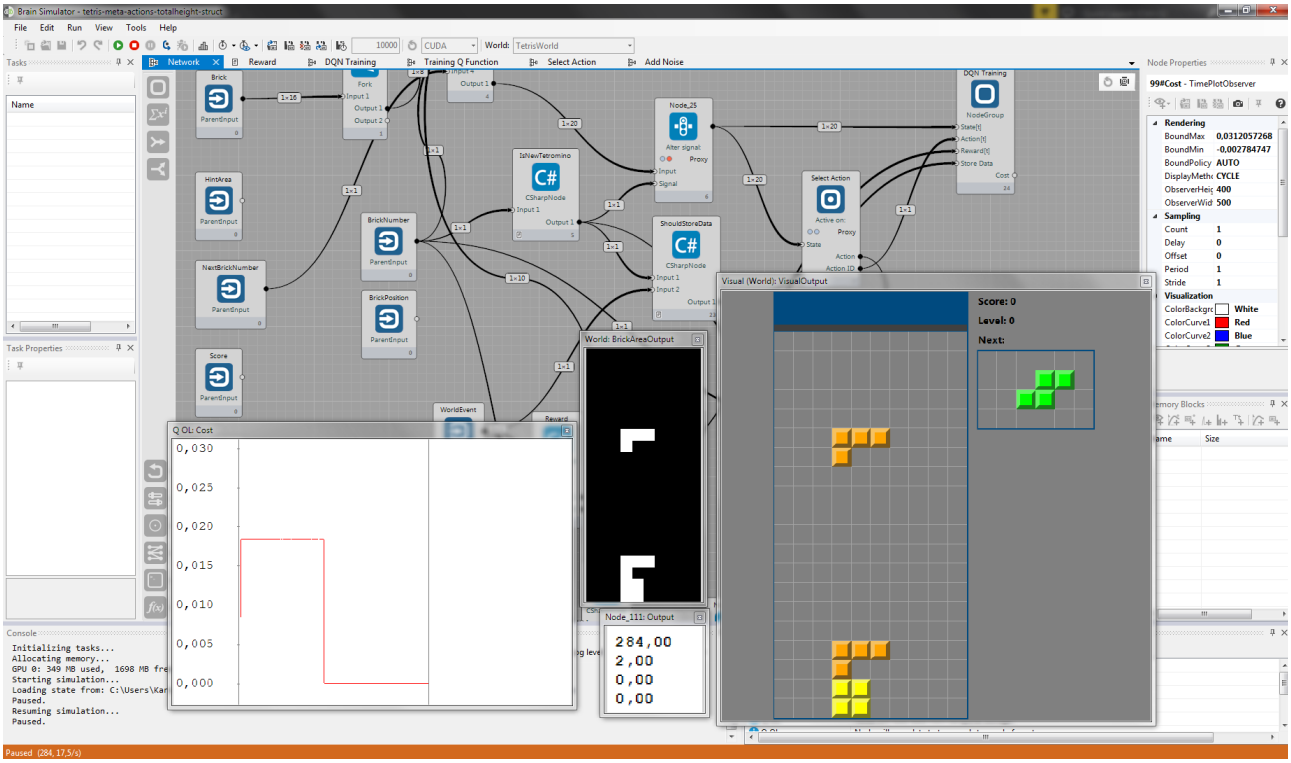
$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

The learning process begins by exploring the game environment using a random policy. Experience of playing the game is recorded as quadruples (s_t, a_t, r_{t+1}, s_{t+1}) and stored in a buffer. A deep Q-network (DQN) is trained using temporal difference learning by replaying the training samples stored in the buffer.

Implementation

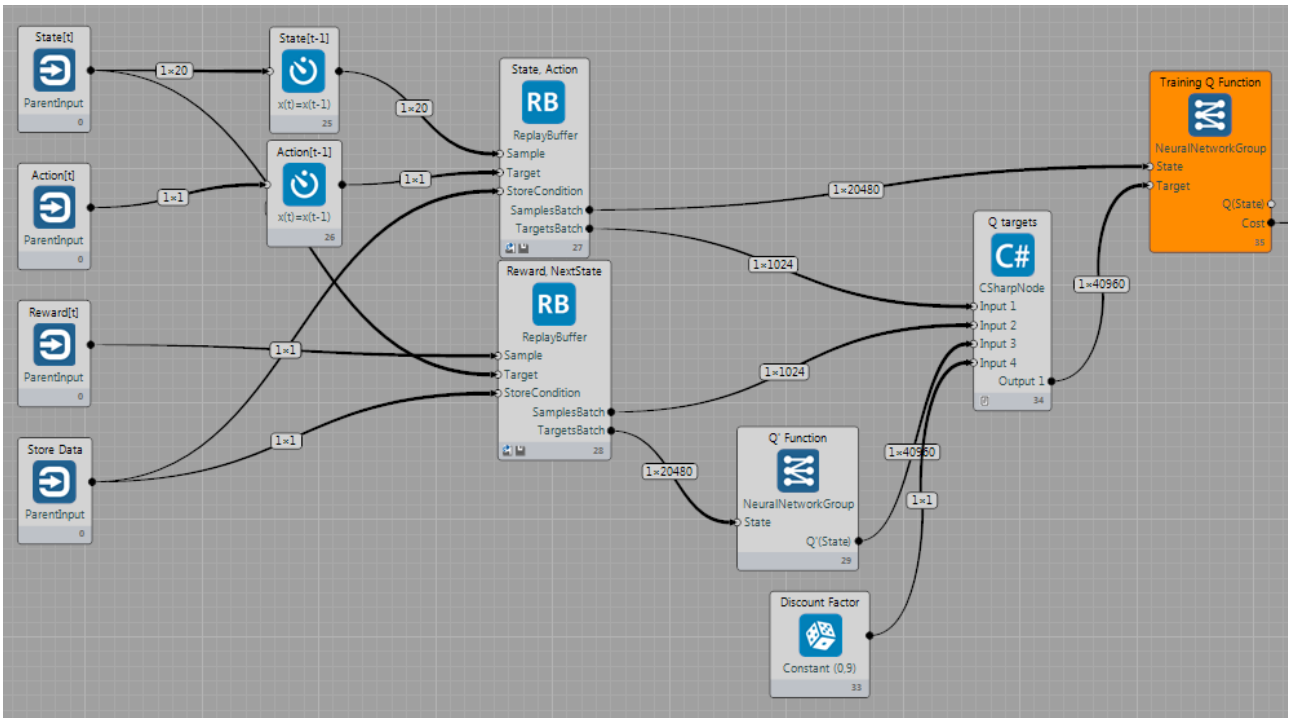
As a framework to implement the RL agent, I chose BrainSimulator [6] from company GoodAI. In BrainSimulator, computation is represented as a graph of nodes which process data and edges which show flow of data between nodes.
Architecture of the computation graph can be saved as a .brain project file and it can also include trained data (e.g. network weights).

The framework includes Tetris game in an environment called TetrisWorld, which interacts with other nodes through its inputs and outputs. I slightly modified the Tetris game code to provide extra information about game state.

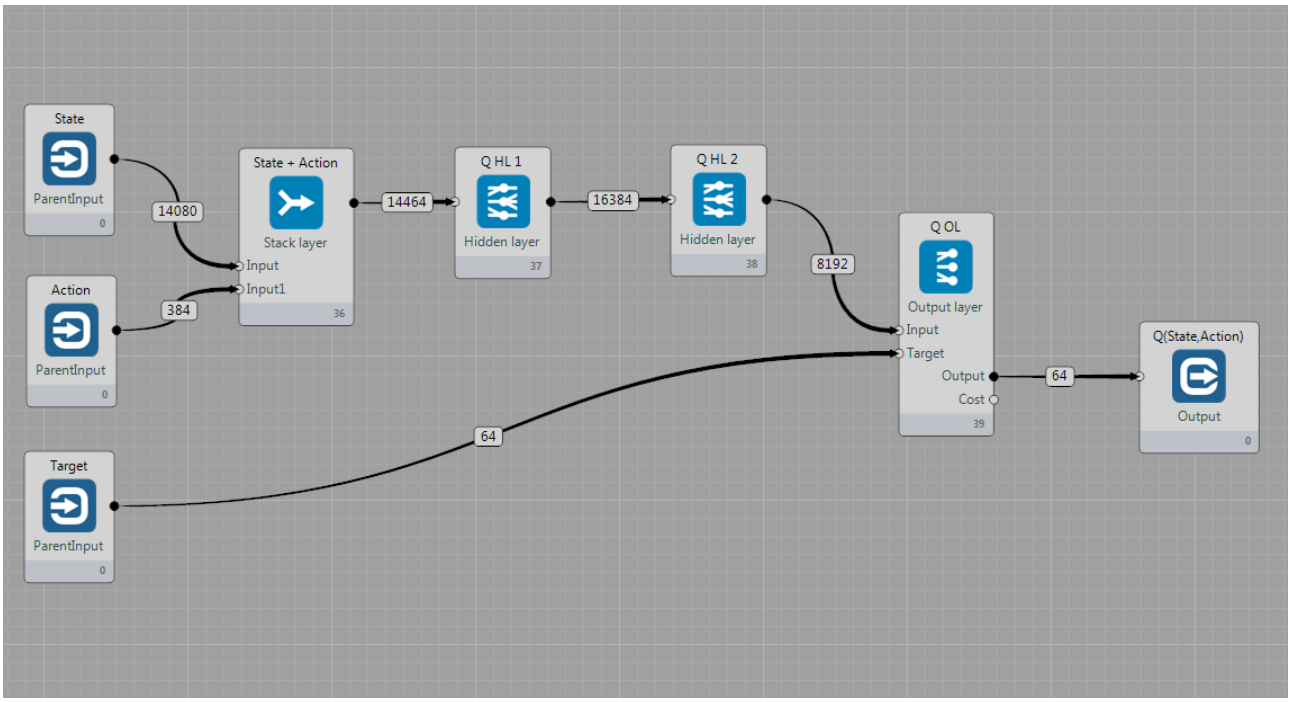


Neural networks, as well as other common data transformation nodes are also available. Other data manipulation tasks were implemented as C# scripts inside scripting nodes.

Reinforcement learning is implemented as a graph of neural networks, several transformation nodes and two ReplayBuffer nodes, which store experience from playing the game and then create training data from it. Two buffers are necessary solely because of technical reasons, as one buffer can only store pairs of data, but quadruples are needed.



The neural network of choice has 2 fully connected hidden layers with activation function RelU and output layer with no activation function.



Preparing and preprocessing of data, scripts, as well as action execution can be found in attached projects if deemed interesting.

Evaluation and Testing

RL agents were compared against each other as well as against a human in 3 metrics:
- average duration of the game (in time steps)
- average score at the end of the game (number of cleared lines)
- average number of placed tetrominos

The averages were calculated from playing 25 independent games.

Agents were tested with two different representations of game state:
- **raw** downscaled playing field of size 10x22 with removed colors
- **structured** information such as height of every column, current and next tetromino

There are 6 **primitive actions** available to agents. As an attempt to eliminate long delay between actions and rewards, **meta actions** were also tested. Instead of moving tetromino one action at a time, meta action represents final position and rotation of the tetromino, which is then unfolded into sequence of primitive actions. There are 40 meta actions in total — 4 different rotations per 10 horizontal positions. After moving tetromino to the right position, the meta action script moves it straight down.

Since performance of RL agents depends heavily on reward function used to train it, following game information was used to create rewards:
- **score** change as provided by game. Line elimination is rewarded with 1 point and game over with -1 point
- **time**: small constant reward for every step the agent survives, which should motivate it to play longer
- **lowest** stack of tetrominos. Agent should minimise maximum height of tetromino stack
- **total height** of tetromino stack. Minimise sum of heights of all columns in stack
- **bumpiness** of tetromino stack. Minimise absolute differences in heights of neighboring columns in tetromino stack
- **holes:** minimise number of holes in tetromino stack. Hole is an empty field covered by a tetromino from above

The exact formulas for calculating combined rewards can be found in attached project files inside a C# script node called Reward.

When primitive actions were used, DQN consisted of 256 and 128 neurons in hidden layers respectively and was trained with mini-batch size 64.
Because of the increase in number of available actions (from 6 to 40) when meta actions were tested, DQN was larger - 512 and 256 neurons in hidden layers respectively and batch size 1024 was used.

Neural network weights were optimised using RMSprop with training rate 10^{-4}, smoothing factor 0.9 and momentum 0.9. Training rate was manually decreased down to 10^{-6}. The agents were trained for at least 200 000 steps until learning stalled. Longer training of even several million steps hasn't improved performance in measurable way.

Agents perceived the game in epochs instead of continuous play. An epoch begins when the first tetromino appears and ends after game over signal from TetrisWorld is sent.

Results

Results of my experiments can be found in following table. Parameters of the agents are shortened to an acronym. The first letter - P or M signifies whether primitive or meta actions were used. Next item is the reward function:

- LTB stands for combination of lowest height, total height and bumpiness
- STBH stands for score, total height, bumpiness and holes
- SLTBH stands for score, lowest height, total height, bumpiness and holes

Following item describes whether raw screen or structured data were used as state of the agent. Last item shows the discount factor.

Agent	Time Steps	Bricks Placed	Lines Cleared	Training Steps
Random	923.53	18.83	0	0
P, Score, Raw, 0.9	741.6	16.7	0	300,000
P, Lowest, Raw, 0.9	507.4	13.6	0	250,000
P, SLTBH, Struct, 0.9	734.1	13.1	0	250,000
P, STBH, Struct, 0.9	878.2	16.1	0	270,000
P, TotalHeight, Struct, 0.9	1001.0	18.9	0	250,000
P, Holes, Struct, 0.9	1118.5	18.6	0	210,000
P, Score, Struct, 0.9	1170.6	20.4	0	210,000
P, Lowest, Struct, 0.9	1219.7	19.9	0	260,000
M, Score, Struct, 0.9	1300.2	18.9	0	300,000
M, Score+time, Struct, 0.9	1400.1	21.5	0	200,000
M, LTB, Struct, 0.9	1092.1	15.7	0	200,000
M, Lowest, Struct, 0	1142.0	16.4	0	100,000
M, TotalHeight, Struct, 0	1502.4	20.7	0	100,000
M, Holes, Struct, 0	1672.6	23.1	0	100,000
Human	965	76	11	0

Since random policy was used for exploration, it is suitable as a benchmark for comparison with RL agents.

Naive application of DQN to raw state with score as reward showed performance worse than random policy. It was soon discovered that the random exploration policy is so terrible that it never clears even a single line. That makes it very difficult to learn a good policy, since the only reward received is punishment when game ends.

This issue was partially solved by using heuristics for more frequent rewards, but raw state proved to be challenging nonetheless. Convolutional neural network may improve the performance, particularly when convolution happens over columns as described here [3]. In following experiments, only structured state was tested.

The best result with primitive actions was achieved by minimising maximum height of tetromino stack, which played for 32% more time steps than random policy and placed 5% more tetrominos.

The pitfall of primitive action is the long time delay between action and reward. This issue can be solved with meta actions. Experiments with them show improved performance.

The best overall policy played on average 81% longer games and placed 22% more tetrominos than random policy. Interestingly, this policy was trained with zero discount factor, which means the policy plans only one step ahead according to the heuristics. My hypothesis is that since sequence of next tetrominos is random, approximating future rewards is difficult and can be only done stochastically.

It should also be noted that some agents failed to converge and their results are not included in the table. Another interesting finding was that by heavily punishing agent for reaching undesirable state (e.g. too many holes in tetromino stack), the learnt policy committed “suicide” by quickly placing tetrominos on top of each other to end the game quickly to get rid of its misery.

Humans play on average shorter games, because they prefer to move tetrominos down quickly. However, humans with no prior training can place many more tetrominos and successfully eliminate lines. An expert could perhaps play the game almost indefinitely.

Conclusion

While most of the found policies were statistically better than their random exploration policy, they are not particularly appealing to human eye. No trained policy was able to to reach novice human level and eliminate lines. The issue of no training examples showing how lines are cleared was not resolved, as I was not able to steer the agent towards doing it by reward heuristics alone. As a solution, one could perhaps create such examples manually or by another AI agent. Tetris remains a challenging game to train end-to-end by game rewards alone.

Installation

My modified version of BrainSimulator can be either downloaded from GitHub or found in attached folder *BrainSimulator*. Only supported platform is 64-bit Windows and CUDA-enabled graphics card is required.
Visual Studio project can be opened from *BrainSimulator/Sources/BrainSimulator.sln*
Alternatively, BrainSimulator application can be launched from *BrainSimulator/Sources/Platform/BrainSimulator/bin/Release/BrainSimulator.exe*

All trained agents together with a .gif showing an example of gameplay can be found in folder *Agents.* BrainSimulator projects prepared for training are located in folder *Projects.* Projects can be opened in BrainSimulator using the File menu.

References

[1] https://en.wikipedia.org/wiki/Tetris
[2] http://www.cs.unm.edu/~pdevineni/papers/Saad.pdf
[3] http://cs231n.stanford.edu/reports2016/121_Report.pdf
[4] https://en.wikipedia.org/wiki/Reinforcement_learning
[5] https://en.wikipedia.org/wiki/Q-learning
[6] https://github.com/GoodAI/BrainSimulator