Playing Tetris using Reinforcement Learning, Homework B

Course: Advanced machine learning using neural networks.
Contact: Henrik Klein Moberg, henrik.kleinmoberg@chalmers.se

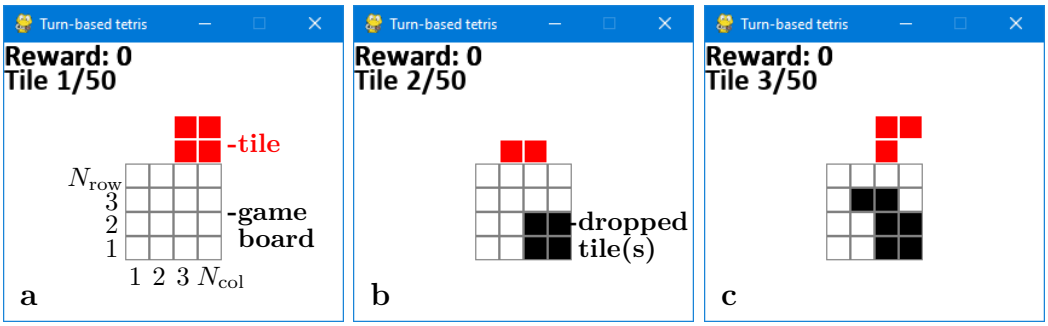
---

1 Background

In reinforcement learning [1, 2, 3], an agent interacts with an environment. The environment provides the agent with rewards in response to what actions it takes in a given state. By trial and error, the agent learns a strategy of how to take actions to optimize the long-term future reward. This setup has many applications in decision making and control theory. Some example applications are robot control [4], self-driving vehicles [5], natural language processing [6], reduced energy consumption in data centers [7], or to understand observed behaviour in nature, for eample schooling of fishes [8], soaring of birds [9], or navigation strategies of swimming plankton [10]. Reinforcement learning is also suitable for developing efficient strategies in many types of games. It has for example been successfully implemented to play old Atari computer games on the level of humans [11], beating the world champion in the game of Go [12], and to beat top class players in Starcraft [13]. In this task you will use reinforcement learning (Q-learning and deep Q-networks) to construct an artificial player that learns to play a simplified version of the computer game Tetris.

2 Description of the game

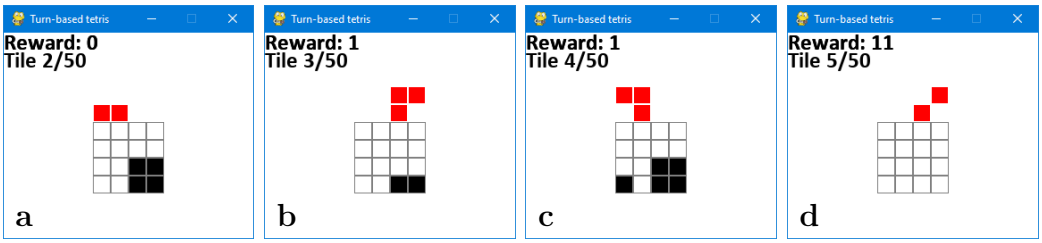
The game consists of a rectangular game board with $N_{\text{row}} \times N_{\text{col}}$ positions, where N_{row} and N_{col} denote the number of rows and columns. Initially, the game board is empty (Fig. 1**a**). At each turn a tile is placed on the game board (red squares in Fig. 1). The tile can be rotated in steps of 90° and moved horizontally before it is dropped from the top. It lands on top of the first occupied position encountered on the game board (Fig. 1**b**–**c**). If



Figur 1: Game board at the first three tiles of a game.

there is no place on the game board to put the next tile, the game is over and the player obtains a negative reward, $r = -100$. If all positions in a row become occupied (Fig. 2**a**), the row is removed and all rows above fall down one step (Fig. 2**b**). Completing a row in

this way gives a reward $r = 1$. If several rows are completed at the same turn, you get a much higher reward $r = 10^{N_{\text{completed}}-1}$, where $N_{\text{completed}}$ is the number of rows that were completed (Fig. 2**c**,**d**). The aim is to build up the game board such that you have a high



Figur 2: Game board configurations filling up one (**a**,**b**) or two (**c**,**d**) rows.

probability to complete several rows in one turn, while minimizing the risk of filling the game board up so that you cannot place any tile. The game ends after a fixed number ('`max_tile_count`') of turns (no penalty is given if the game ends in this way).

3 Getting started

You do not need to program the game itself, you can download python scripts for the game at the course page under [files/Homeworks/HW B Kristian]. To get started, you can run the program to verify that it works and to familiarize yourself with the game mechanics. The program consists of three files: `tetris.py`, `gameboardClass.py` and `agentClass.py`, with content as described below

- `tetris.py` is the runnable script. It contains initializations and the main loop over game turns. The only thing you need to change in this file is '`param_set`' that you can set to different values for the different tasks (`PARAM_TASK1a`, `PARAM_TASK1b`, and so on) and '`human_player`' that you can set to either 0 or 1. Initially, '`human_player`' is set to 1, giving you a graphical user interface of the game, allowing you to familiarize yourself with the game. You can later use this mode to test the different configurations in '`param_set`', or to test if you can get a higher score than your artificial player. Pressing left or right on the keyboard moves the tile horizontally and by pressing up you rotate the tile. When you have moved the tile to the desired position you press down or space to drop it on the game board. Following the rules above you get rewards until the game finishes.

  You also have the possibility to set `strategy_file` to the path of data file containing the Q-table or Q-network of a previously trained agent. In this mode, the agent puts the tile where it wants it and you can drop it by pressing or holding space to evaluate how the agent plays the game.

- `gameboardClass.py` contains a class TGameBoard that takes care of the gameboard and the game mechanics. You do not need to make changes to this file, but it may help to occasionally look through it, since you will need to call functions and use attributes from this class.

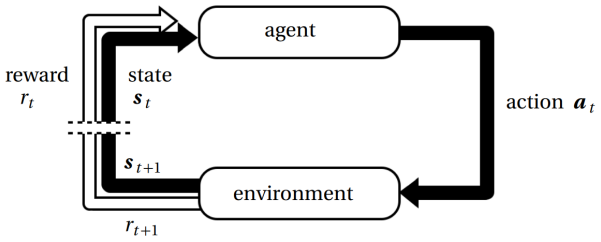
- `agentClass.py` contains three classes: TQAgent, TDQNAgent and THumanAgent. THumanAgent is simply an interface to allow a human to play the game via the

GUI. TQAgent and TDQNAgent contain skeleton structures for agents that can place tiles on the gameboard. Your task is to write the code in these classes, allowing the agents to use Q-learning (TQAgent) and Deep Q-Networks (TDQNAgent) to learn to play the game.

The files above import packages (pygame, h5py, etc) that you may need to install before being able to run the script.

4 Q-learning

Q-learning is a general framework for reinforcement learning [1, 3]. It is based around an agent that learns to solve a problem using trial and error in the process illustrated in Fig. 3. Given a state s_t of the environment at time t, the agent chooses an action a_t.



Figur 3: Scheme illustrating the process in reinforcement learning. Taken from Ref. [3].

Using this action the agent interacts with the environment until a new state s_{t+1} occurs. At this point the agent is also given a reward r_{t+1} that quantifies how well it performed. Starting from the new state s_{t+1}, the agent repeats the process of choosing an action to get to a new state and obtaining a new reward. The process gives rise to a sequence of visited states, taken actions and received rewards on the form:

$$s_0\,,\ a_0\,,\ r_1\,,\ s_1\,,\ a_1\,,\ldots,s_t\,,\ a_t\,,\ r_{t+1}\,,\ s_{t+1}\,,\ a_{t+1}\,,\ldots$$

The action is chosen for a given state according to the current policy of the agent. The policy could be any way of choosing the action, for example to stubbornly choose the same action irrespective of the state, or to always choose the action randomly. Reinforcement learning aims at finding the policy that optimizes the expectation value of the sum of future rewards

$$R_t = \sum_{\tau=t}^{T-1} r_{\tau+1} \tag{1}$$

where T is the terminal time of the process. For any state-action pair, we collect the expected future reward in a table with elements $Q(s,a)$. The optimal policy is then to, for a given state s, choose the action a with the maximal entry in the Q-table: $a = \text{argmax}_{a'} Q(s,a')$. It follows that $Q(s,a)$ can be defined recursively as

$$Q(s_t,a_t) = \langle r_{t+1}\rangle + \max_a Q(s_{t+1},a)\,. \tag{2}$$

In Q-learning we use experience to improve an approximate Q-table, $Q_t(s,a)$, using the update rule

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha\left(r_{t+1} + \max_a Q_t(s_{t+1},a) - Q_t(s_t,a_t)\right) . \tag{3}$$

Here α is a learning rate, $0 < \alpha \leq 1$. Using Eq. (2), we see that the true Q-table, $Q(s,a)$, is a fixed point to this equation if r_{t+1} is replaced by its average. For most realistic applications the process will not converge quick enough to the true Q-table, but the rule (3) nevertheless tends to give close to optimal policies.

Training is divided into a number of episodes. Each episode starts at the beginning of a game and lasts until game over (by filling the game board or if `max_tile_count` turns is reached). Starting from, for example, an initially empty Q-table, $Q_0(s,a) = 0$, the Q-table is updated according to Eq. (3) when the reward has been given after a tile placement. The resulting Q-table after one episode is used as the starting Q-table in the next episode and training goes on until a predetermined number of episodes ('`episode_count`') have been played out.

During training, it is often beneficial to mainly choose the action according to the current best estimate (greedy policy), but sometimes choose the action randomly with a small probability ε (ε-greedy policy):

$$a_t = \begin{cases} \text{random valid action} & \text{with probability } \varepsilon \\ \text{argmax}_a Q_t(s_t,a) & \text{otherwise} \end{cases}$$

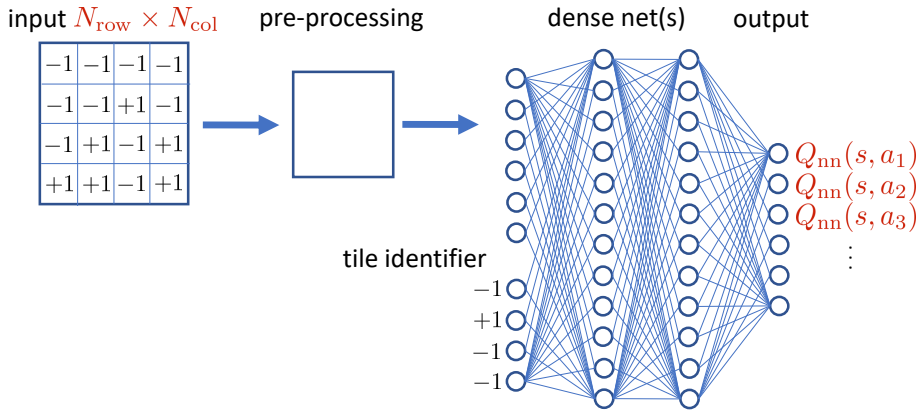
If several actions give the same maximal $Q_t(s_t,a)$, choose one of these at random.

The scheme outlined in Fig. 3 applies to many different optimization problems. The difference between different problems is of course that the environment is different, but also the choice of state, actions and rewards strongly influence which policy is optimal. In our case the agent is the player of the game. Its actions are to place tiles on the game board. Valid actions are horizontal tile placements within the game board and non-degenerate rotations. To find out which actions are valid for a given tile, you can call `fn_move(new_tile_x,new_tile_orientation)` which returns 1 if the action is invalid (and moves the tile according to the action if it is valid). Invalid actions can be avoided by setting their corresponding entries in the Q-table to large negative values. Equivalently, you can use one Q-table per tile type, making each entry of the Q-tables correspond to an allowed action. The environment is the game that gives the player a reward given by the rules in the game description above, i.e. $r = -100$ and game over if no tile can be placed, and $r = 10^{N_{\text{completed}}-1}$ when $N_{\text{completed}}$ rows are completed and removed in one turn.

The states contain the information the agent use to play the game. To improve convergence, one should only include the most relevant information needed to play the game. The state you should use in this task is the identifier of the current tile and a $N_{\text{row}} \times N_{\text{col}}$ matrix with binary elements indicating whether the corresponding position on the game board is occupied (note that if you were to only use the column heights as the state, the agent will have a hard time because it does not know if there are holes in the columns). This binary representation of positions is a good setup for implementing pattern recognition in deep Q-networks, but one could in general use additional information such as the number of placed tiles, or hard-code problem-specific states, such as the sum over column heights on the game board, or the number of holes among the occupied game board positions.

5 Deep Q-networks

If the number of states is very large or if the state is continuous, it is impractical to use the Q-table to approximate the expected future reward. Instead, one can use a function approximator, such as a neural network, that takes the state as an input and gives one output for each possible action. These outputs estimate the expected future rewards for the different actions for any input state. We denote this network by $Q_{\rm nn}(s,a)$ and we use a method called 'deep Q-networks' [2], introduced in Ref. [11], where it was used to develop agents that play Atari games. The function approximator we use is illustrated in Fig. 4. The first input is the state of the game board represented by binary values, where -1 denotes unoccupied positions and $+1$ denotes occupied positions. This is followed by a pre-processing step. In Ref. [11] this step consists of convolutional neural networks that process image data from the Atari game. The output of the convolutional network is, in the current setup, combined with a binary representation of the tile identifier (the node corresponding to the current tile is +1, all other nodes are -1). The combined signal is fed to a fully connected network whose number of output nodes is equal to the number of possible actions. Each hidden layer is followed by a ReLU activation function. It is easiest to implement the network using a deep learning framework such as PyTorch.



Figur 4: Illustration of network architecture for an example state s ($N_{\rm row} \times N_{\rm col}$ binary input representing the current game board and identifier of the current tile). Each output node represents one action a_i and the values of the output nodes are $Q_{\rm nn}(s,a_i)$ for the current state s. The neural network is fully connected but not all connections are shown.

As before, the action is chosen using an ε-greedy choice

$$a_t = \begin{cases} \text{random valid action} & \text{with probability } \varepsilon_E \\ \text{argmax}_a Q_{\rm nn}(s_t,a) & \text{otherwise} \end{cases}$$

where $\text{argmax}_a Q_{\rm nn}(s_t,a)$ denotes the identifier of the output node with maximal output from the network fed with the state s_t. To facilitate exploration, we reduce ε_E from unity initially to a final value ε as a function of the episode number E by $\varepsilon_E = \max(\varepsilon, 1-E/E_0)$, where E_0 is the scale of decay (`epsilon_scale` in the program). To avoid invalid actions, one possibility is to choose the action with the largest output among the valid actions. However, this turns out to impede convergence in some cases. If so, an alternative is to allow invalid actions, but give them a large negative reward.

To stabilize training, two networks are used. One network $Q_{\text{nn}}(s, a)$ is used to calculate the next action and is updated at each change of state. The second network $\hat{Q}_{\text{nn}}(s, a)$ is a target network, it is kept constant for long periods of time and is used to evaluate network performance. Initially $\hat{Q}_{\text{nn}}(s, a) = Q_{\text{nn}}(s, a)$ and every `sync_target_episode_count` episode, $\hat{Q}_{\text{nn}}(s, a)$ is synchronized by copying the weights from $Q_{\text{nn}}(s, a)$.

The network is trained to minimize the loss function

$$L = (Q(s_t, a_t) - y)^2 \,, \tag{4}$$

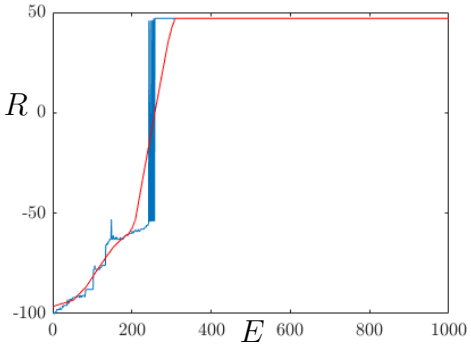
where $y = r_{t+1}$ if the episode has ended and $y = r_{t+1} + \max_a \hat{Q}(s_{t+1}, a)$ (evaluated using the target network) otherwise.

Instead of using the instantaneous state and action as in Eq. (3), the Q-network is updated using an experience replay buffer. This is a data storage of a predefined number (`replay_buffer_size`) of quadruplets on the form $\{s_t\,,\ a_t\,,\ r_{t+1}\,,\ s_{t+1}\}$. The first `replay_buffer_size` time steps of the training (possibly over several episodes) is used to fill up the replay buffer and no update of the Q network is done. At each state change after the replay buffer is full, a new quadruple is appended to the replay buffer and the oldest quadruple is removed. At the same time, a small sample (mini-batch) of `batch_size` randomly chosen quadruplets from the replay buffer is chosen. For each quadruple in the mini-batch, the loss function (4) is calculated and $Q_{\text{nn}}(s, a)$ is updated using a built-in optimization scheme (for example 'Adam') in your deep learning framework.

6 Assignment

Implement Q-learning by completing the class TQAgent according to the instructions above. You can take the learning rate $\alpha = 0.2$. In each of the first three subtasks below a training scenario is listed. For each scenario, train using Q-learning for the given episode count. Store the return (the sum over all rewards) for each episode during training and make a plot of the return against the episode number. Plot the return for each episode and plot the moving average of the return over 100 episodes. It can also be a good idea so save the final Q-table if you want to analyze the found strategy, or if you want to evaluate the behavior of the agent later (using `strategy_file`, see above).

It is recommended to start with task 1.a) below. It is easiest and when it works, tasks 1.b) and 1.c) should be straightforward. An example solution of tasks 1.a) is given below:



If you do not get the Q-learning to work, there are several ways to simplify the system to facilitate debugging. For example, you can reduce the gameboard size to 2×2 and use only 1×1 blocks. Or use the 4×4 gameboard with only the 2×2 tiles, or not allow rotation of the tiles.

As a final remark, note that both Q-learning and Deep Q-networks are stochastic, meaning that different runs give different results, and some runs may even fail altogether. It may therefore be good to run a few instances of the program in parallel for each setup to see how large fluctuations there are. What we in general are interested in is the best solution, meaning that we can take the solution that gives the largest average long-term return.

1.a) [**3p**] Use `param_set=PARAM_TASK1a` to train on a game with a predefined deterministic tile sequence. Use greedy action and train for 1000 episodes. Make a plot of the return according to the instructions above.

1.b) [**1p**] Same as case a) with ε-greedy action (`param_set=PARAM_TASK1b`). For this problem it turns out that we can set ε quite small and constant, choose $\varepsilon = 0.001$. Train for 10000 episodes and make a plot of the return. Compare the results to the data in subtask a). Explain similarities and differences.

1.c) [**1p**] Use `param_set=PARAM_TASK1c` to train on a random tile sequence instead of the deterministic sequence in subtasks 1.a) and 1.b). Use ε-greedy action with constant $\varepsilon = 0.001$. Train for 200000 episodes and make a plot of the return. How does the average strategy compare to subtasks 1.a) and 1.b)? How does the strategy with the highest score compare to subtasks 1.a) and 1.b)? Explain/discuss the results.

1.d) [**1p**] Set `human_player=1` and `param_set=PARAM_TASK1d` to obtain a larger game board (8×8) with a larger tile set of the original tiles from Tetris. Discuss the possibility to implement the Q-learning algorithm for this setup (you don't need to train your program for this case, but you could try to if you want).

Implement the deep Q-network method by completing the class TDQNAgent according to the instructions above. Use decreasing ε_E with episode number, two separate networks for taking actions and for evaluating performance, and use an experience replay buffer when updating the network.

2.a) [**4p**] Redo task 1.c) using the deep Q-network method. Use parameters from the case `param_set=PARAM_TASK2a`. For simplicity, skip the preprocessing step in Fig. 4 and simply flatten the data to merge it with the tile identifier. Then feed the combined data into the dense, fully connected network. You can use two hidden layers with 64 neurons in each.

2.b) [**2p**] (**Optional bonus question**) Set up a network that can successfully solve the problem with the larger tile set and game board suggested in subtask 1.d). You may need to use a convolutional network in the preprocessing step, or extend the state space. Note that it is very hard to get this setup to work, so you should only try this if you have the interest and time to spare.

7 Examination

In the oral examination you will present your own code and your results to the tasks above. You should be able to explain the basic functionality of your used Q-learning and deep Q-network algorithms and to discuss and explain the results of your solved tasks. As the time is short, make sure that you are well prepared to present your results.

Referenser

[1] Richard S. Sutton, A. G. B. *Reinforcement Learning* (MIT Press Ltd, 2018).

[2] Lapan, M. *Deep reinforcement learning hands-on* (Packt Publishing Ltd, Birmingham, UK, 2020).

[3] Mehlig, B. *Machine learning with neural networks* (2021).

[4] Kober, J., Bagnell, J. A. & Peters, J. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research* **32**, 1238–1274 (2013).

[5] Kiran, B. R. *et al.* Deep reinforcement learning for autonomous driving: A survey (2020).

[6] Hirschberg, J. & Manning, C. D. Advances in natural language processing. *Science* **349**, 261–266 (2015).

[7] Li, Y., Wen, Y., Tao, D. & Guan, K. Transforming cooling optimization for green data center via deep reinforcement learning. *IEEE Transactions on Cybernetics* **50**, 2002–2013 (2020).

[8] Gazzola, M., Tchieu, A. A., Alexeev, D., de Brauer, A. & Koumoutsakos, P. Learning to school in the presence of hydrodynamic interactions. *Journal of Fluid Mechanics* **789**, 726–749 (2016).

[9] Reddy, G., Celani, A., Sejnowski, T. J. & Vergassola, M. Learning to soar in turbulent environments. *Proceedings of the National Academy of Sciences* **113**, E4877–E4884 (2016).

[10] Colabrese, S., Gustavsson, K., Celani, A. & Biferale, L. Flow navigation by smart microswimmers via reinforcement learning. *Physical Review Letters* **118**, 158004 (2017).

[11] Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518**, 529–533 (2015).

[12] Silver, D. *et al.* Mastering the game of go with deep neural networks and tree search. *Nature* **529**, 484–489 (2016).

[13] Vinyals, O. *et al.* Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* **575**, 350–354 (2019).