Theory of Computer Games 2022 – Project 2

Overview: **Train a player to play *Threes!* with high win rates**.

1. Implement the n-tuple network.
2. Implement the TD(0) afterstate learning framework.
3. **Train a player based on TD learning and n-tuple network.**

Specification:

1. The player should be trained by **the TD(0) afterstate learning framework** as in [1].
   a. An episode is defined as $s_0 \cdots \dashrightarrow s_t \xrightarrow[r_t]{a_t} s'_t \dashrightarrow s_{t+1} \xrightarrow[r_{t+1}]{a_{t+1}} s'_{t+1} \dashrightarrow \cdots s_T$
   b. The TD(0) afterstate learning framework adjusts the afterstate values, $V(s'_t)$, by $\boldsymbol{V(s'_t) \leftarrow V(s'_t) + \alpha\big(r_{t+1} + V(s'_{t+1}) - V(s'_t)\big)}$. See Methodology for more details.
      i. You can train the player with more sophisticated TD variants if needed.
      ii. The total number of episodes for training is not limited.
2. The n-tuple network for storing afterstate values **can only use 1GB of memory at most**.
   a. The structure of the n-tuple network (e.g., 8×4, 4×5, 4×6) is not limited.
   b. The network should be serializable, i.e., be able to save to a file and load from a file.
3. The player **takes actions based on the rewards and the afterstate values,** $\boldsymbol{r_t} + \boldsymbol{V(s'_t)}$:
   a. Not required to perform extra searching, i.e., just expand 4 afterstates.
   b. The program speed should be **at least 50000 actions per second**.
      (an approximate value, see Scoring Criteria for details)
4. The environment and the rules are the same as those in Project 1.
5. The statistic is required, and the requirements are the same as those in Project 1.
6. The program arguments and record format are the same as those in Project 1.
7. The program should be able to execute in the Linux environment.
   a. C++ is highly recommended for TCG projects since the methods involved are sensitive to CPU speed.
   b. For other programming languages (e.g., Python), contact TAs for more details.
   c. A makefile for the program should be provided.
      i. Provide `make` for compiling the program.
      ii. Provide `make stats` for executing the program, generating statistics of 1000 games and saving the results into a file named `stats.txt`.

Methodology:

1. The player should expand all afterstates, **evaluate the afterstate values by the n-tuple network**, then select an action according to the immediate rewards and the afterstate values. To achieve this, you need to
   a. Design and implement an n-tuple network.
   b. Implement the TD(0) afterstate learning algorithm.
   c. Train an n-tuple network with TD(0) afterstate learning algorithm.
   d. Use immediate rewards and afterstate values to take action.

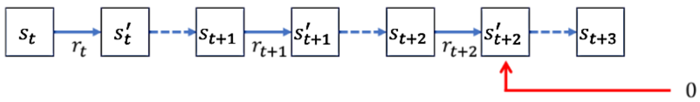
2. Precautions for implementing an n-tuple network:
   a. **Try the simplest 8×4-tuple network first** (4 vertical lines and 4 horizontal lines). It has 8 weight tables $\Theta_1, \dots, \Theta_8$, corresponding to 8 tuples, $\phi_1, \dots \phi_8$:
      iii. $\phi_1(s'_t)$ maps the first row to an index for accessing feature weights in Θ_1, e.g., 

maps to 0x6241. See Appendix for more details.
      iv. **Be careful to implement feature extraction and index encoding**.
      v. Once the simplest network works, you may refer to [2] for a more powerful n-tuple network design or even design it yourself.
   b. Isomorphic patterns with shared weights can speed up the training process.
      i. Be careful with the order of accessing indexes. You should use the same order when accessing isomorphic patterns.
      ii. Note that the simplest 8×4-tuple network does not consider weight sharing. It has 8 lines, each corresponding to a unique weight table. When weight sharing is applied, Only 2 weight tables are used to store 2 sets of tuples: the outer lines, and the inner lines.
      iii. When you are using a larger network, it is highly recommended to use isomorphic patterns. In addition to speed up the training, it also significantly reduces the amount of required memory.
3. Precautions for implementing the TD(0) algorithm:
   An episode is defined as $s_0 \cdots \dashrightarrow s_t \xrightarrow[r_t]{a_t} s'_t \dashrightarrow s_{t+1} \xrightarrow[r_{t+1}]{a_{t+1}} s'_{t+1} \dashrightarrow \cdots s_T$
   a. Take the simplest 8×4-tuple network as an example, there are 8 weight tables $\Theta_1, \dots, \Theta_8$, corresponding to 8 tuples, $\phi_1, \dots \phi_8$:
      i. The value function $\boldsymbol{V(s'_t)}$ **is calculated as** $\boldsymbol{\Theta_1[\phi_1(s'_t)] + \cdots + \Theta_8[\phi_8(s'_t)]}$.
      ii. For a value function $V(s'_t)$, only its corresponding 8 feature weights are adjusted: $\Theta_1[\phi_1(s'_t)], \dots, \Theta_8[\phi_8(s'_t)]$. These 8 feature weights **are adjusted with the same TD error:** $\boldsymbol{\Theta[\phi(s'_t)] \leftarrow \Theta[\phi(s'_t)] + \alpha\big(r_{t+1} + V(s'_{t+1}) - V(s'_t)\big)}$.
   b. Do not **forget to train the last afterstate** s'_{T-1}. Its TD target should be 0.
   c. Do not **confuse the immediate rewards**. The TD target for $V(s'_t)$ is $r'_{t+1} + V(s'_{t+1})$.
      i. Threes! has no official immediate reward. Instead, a value function $F(s)$ of the whole puzzle is defined: $F(s) = \sum_v 3^{1+\log_2(v/3)}$ for all $v \geq 3 \wedge v \in s$
      ii. You may use the difference of F as the reward, i.e., $F(s'_t) - F(s_t)$.
   d. Do not **confuse the feature weight and the value function** when calculating TD errors. The TD error for $\Theta[\phi(s'_t)]$ is NOT $(r_{t+1} + \Theta[\phi(s'_{t+1})] - \Theta[\phi(s'_t)])$.
   e. Do not **confuse the states and the afterstates**.

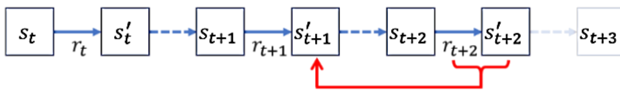
f. The backward method and the forward method are both common implementations:

   i. The backward method updates the afterstates from the end to the beginning.

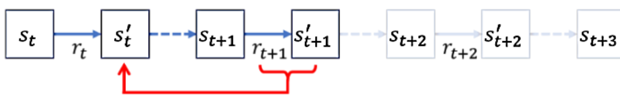
      Step 1: after game over (s_{t+3}), update the last state (s'_{t+2})



      Step 2: update the previous *afterstate* (s'_{t+1})

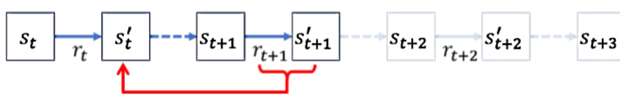


      Step 3: update the previous *afterstate* (s'_t)

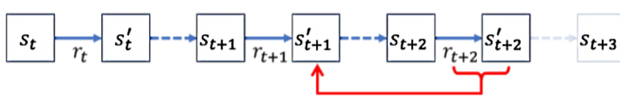


   ii. The forward method updates the afterstates from the beginning to the end.

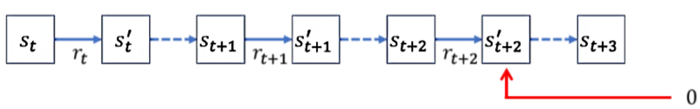
      Step 1: apply an action (s'_{t+1}), update previous state (s'_t).



      Step 2: apply an action (s'_{t+2}), update previous state (s'_{t+1}).



      Step 3: after game over (s_{t+3}), update the last state (s'_{t+2})



4. Precautions for training the n-tuple with TD(0) algorithm learning algorithm:

   a. The initial learning rate α should be calculated based on the number of features. For example, **$\boldsymbol{\alpha}$ can be $\mathbf{0.1 \div 8 = 0.0125}$ for the simplest 8×4-tuple network**.

      i. For an isomorphic 4×6-tuple network, $\alpha = 0.1 \div 32$ since it has 32 features.

      ii. Reduce the learning rate α only if the network is converged. In addition, do not reduce the learning rate too fast.

   b. When using the provided framework for training, **always set `--total`, `--block`, and `--limit`**, e.g, `--total=100000 --block=1000 --limit=1000`.

   c. Always keep the statistic files and log files. They are useful for further analysis.

      i. To record the training log, use the `tee` command.

      ii. It is recommended to use `tmux` or `screen` command to launch your program.

d. Remember to regularly save network snapshots.
   i. Even if all the code is bug-free, the network may sometimes become worse during the training, especially when the learning rate α is low. Snapshots help you recover from such a situation.
   ii. You can refer to the README.md in the sample code for an example of taking snapshots during the long training.

Scoring Criteria:

1. **Performance (100 points)**: Calculated by $\text{round}\left(\left(\frac{\text{WinRate}_{384}}{10}\right)^2\right)$.
   a. WinRate_{384} is the reaching rate of the 384-tile in 1000 games.
   b. A **judge program** is provided to assess the grade.
      i. First, play 1000 games and write the statistics to stats.txt by your program:
         `$ ./threes --total 1000 --save stats.txt --OTHER_ARGS`
         You may change `OTHER_ARGS` to other arguments required by the program.
      ii. To judge the statistics, load it by the judge:
         `$ ./threes-judge --load stats.txt --judge version=2`
2. **Report (10 points, optional)**: Graded according to the completeness of the report.
   a. Summarize the network design, the method used, the training process, and so on.
3. **Demo (requirement)**: Demo the project in person.
   a. Demo your program and answer a question about implementation.
   b. To be announced.
4. Penalties:
   a. **Time limit exceeded (–30%)**: If the program speed does not meet the minimum speed expected by the judge program.
   b. **Memory limit exceeded (–30%)**: If the program uses more than the memory limit.
   c. **Late submission (–30%)**: If the project requires any modifications after the deadline.
   d. **No version control (–30%)**: If there is no version control.
   e. **Failed demo (–30%~100%)**: If you cannot demo the program or cannot answer the asked question.
5. The final grade is the sum of the indicators minus the penalties.
   a. **The maximum grade is limited to 100 points**.
   b. **The grade is not counted if the demo is not passed**.

Submission:

1. The submission **should be archived as a ZIP file** and **named `ID.zip`**, where **`ID`** is your student ID, e.g., `0356168.zip`.
   a. Pack the **source files**, **makefile**, **report**, and other required files.
   b. Submit the archive **through the E3 platform**.
   c. **Do not upload the network weights to the E3 platform**.
      i. We will announce another location for placing weight files.
   d. Do not upload the version control hidden folder, e.g., the `.git` folder.

2. The program **should be able to run under the provided Linux workstations**.
    a. Available hosts: tcglinux1.cs.nycu.edu.tw, …, tcglinux10.cs.nycu.edu.tw
        i. Use the NYCU CSIT account to log in via SSH.
        ii. Place project files in `/tcgdisk/ID`, where ID is your student ID. Note that you need to create the folder first. For example, suppose that your ID is 0356168:
        `$ mkdir /tcgdisk/0356168 && chmod 700 /tcgdisk/0356168`
    b. The projects will be graded on the provided workstations.
        i. You may use your machine for development. The judge program should work on most Linux platforms.
    c. Do not occupy the workstations. Contact TAs if the workstations are crowded.
3. Version control (e.g., GitHub or Bitbucket) is required during the development.

References:

[1] M. Szubert and W. Jaśkowski, **Temporal difference learning of N-tuple networks for the game 2048**, CIG 2014.

[2] K. Matsuzaki, **Systematic selection of N-tuple networks with consideration of interinfluence for game 2048**, TAAI 2016.

[3] K.-H. Yeh, I-C. Wu et al., Multi-Stage Temporal Difference Learning for 2048-like Games, IEEE TCIAIG 2016.

[4] Threes JS, http://threesjs.com/.

Appendix:

1. Average scores during the training of a million episodes: