Sir Tet Tetris AI Playground and Deep Q Learning agents for Tetris

May 11, 2022

The Sir Tet Playground is a simulation environment for Tetris AI agents designed for easy extensibility and modularity, where agents can be easily built, tested and benchmarked. Included are a few attempts at using Deep Q learning to play Tetris. The code for this project can be found at https://github.com/murtaza64/sirtet.

1 Introduction

Tetris is a video game played on a 10x20 grid where a player must rotate and translate falling tetrominoes (contiguous pieces of four square blocks) to pack them efficiently, filling up horizontal lines of blocks to clear them. The objective of the game is to score as many points as possible before being unable to fit anymore blocks on the screen, where points are awarded for clearing lines, and clearing multiple lines with one block awards more points than separately clearing each of those lines.

The simplified version of Tetris most of the existing AI literature examines allows the agent to select only a column and orientation of the tetromino to place, which entails that sliding falling pieces under a gap or performing 'T-spins' is not possible. However, this reduced version still captures a large portion of the game's difficulty: strategically placing tetrominoes in a way that enables future space-efficiency and clearing.

2 Related work

Many approaches to Tetris agents have been taken over the past few decades. As the state space of Tetris is $2^{200} \cdot 7 \cdot 7 \approx 10^{62}$, much of the work in Tetris is about extracting features from states and/or state-action pairs, and selecting a good feature to reduce the state space, and then optimize over this reduced space. Dellacherie's hand-coded agent [1] uses a fixed set of six features of state-action pairs, and the agent simply chooses the action which maximizes the following formula:

$$\text{eroded piece cells} - \text{col transitions} - \text{row transitions} - \text{landing height} - \text{cumulative wells} - 4 \cdot \text{holes}$$

An explanation of these features will appear in section 1.4.

This hand-coded agent performs remarkably well, and most more sophisticated agents still struggle to match its performance. For learned optimizations, various methods have been attempted: successful genetic/evolutionary approaches [2, 3], some of which can beat Dellacherie's agent; several reinforcement learning approaches, including Least Square Policy Iteration [4] and Ant Colony Optimization [5], which generally fall short of the Dellacherie benchmark; and some novel approaches such as a feed-forward neural network with Particle Swarm Optimization [6] which achieves performance about half as good as Dellacherie.

There are several existing literature reviews of the various approaches to Tetris collating performance results [7, 8]. However, one shortfall of these reviews and indeed the existing literature as a whole is that there doesn't seem to be a standard Tetris simulation envrionment, so slight differences in implementation, for example in approach to randomizing the next tetromino (grab-bag vs uniformly random), might be a source of slight inaccuracies in head-to-head comparisons of approaches. This is one of the motivations of introducing the Sir Tet playground, where agents with all manner of approach can be benchmarked against one another.

There seems to be a lack of successful Q-learning approaches to Tetris in the literature, which was one of the motivations for trying linear and deep Q-learning approaches here. Additionally, Q-learning seemed like a good fit for a Tetris agent to be able to discover strategies for maximizing score without too much hand-coding.

3 The Sir Tet simulator and AI playground

The Sir Tet AI playground is implemented in Python with curses, so it can run in almost any terminal with color support, without the need for GUI functionality. It is designed for modularity and extensibility, with plug-and-play addition of agents which must only implement a `get_best_move` method, and the ability to watch agents play in real time as they train. In addition, you can play Tetris in your terminal.

3.1 Agent features

Agents are implemented as python classes (inheriting from `BaseTetrisAgent`) and can be installed with a simple addition to `AIRunner` in `ai_runner.py`:

```
class AIRunner(Runner):

    installed_agents : 'list[tuple[BaseTetrisAgent, set]]' = [
        (TetrisQLearningAgent,  {Options.TRAINABLE}),
        (DellacherieAgent,      {}),
        (DeepQAgent,            {Options.TRAINABLE, Options.MONITOR, Options.SAVABLE}),
        (DeepQExpReplayAgent,   {Options.TRAINABLE, Options.MONITOR, Options.SAVABLE})
    ]
```

Agent initiailizers must take one positional argument which is a `logging.Logger` object.

At minimum, an installed agent with a `get_best_move` method will support watching the agent play Tetris. While watching the agent, scores of individual games are printed out and a high score is tracked. The method should return a legal `(orientation, column)` integer pair.

Agents with `Options.TRAINABLE` set must implement a `train(n_iters)` method which runs `n_iters` iterations of a training step. For the Q-learning approaches here, this iteration count corresponds to number of games to play, but other approaches could interpret this parameter in any other reasonable way. The training is done in a separate thread, while a (much slower) foreground thread displays a live view of the agent continuously playing games with its current training state, so the user can watch their agent learn and improve.

Agents with `Options.SAVABLE` must implement `save(filename)` and `load(filename)` methods to save and load their state. For example, Keras-based agents can use `model.save_weights` and `model.load_weights`. When the user chooses to save or load weights, the checkpoint name they provide will be prepended with a directory named after the agent's class name before being sent to the `save` or `load` method (e.g. `DeepQAgent/checkpoint`).

Finally, `Options.MONITOR` indicates that an agent has a `monitor()` method, which can be polled by the controller to retrieve arbitrary context about the current state of the agent for display to the user. The screenshot above shows an example of this, where the left panel contains information about hyperparameters and training statistics provided by the agent's `monitor()` method.

3.2 Recording demonstrations

The Sir Tet playground lets you record demonstrations for agents that might require player data for pretraining steps (for example, Deep Q-Learning from Demonstrations). After entering the demonstration screen, you play Tetris for as long as you want, and when you choose to save them, a text file under `demonstrations/TIMESTAMP.demo` will be created. The format of of this text file consists of state-action pairs separated by newlines. The state is encoded by treating each row of the board as a 10-bit integer, concatenating the base 10 representations of the rows and appending the letter represenations of the current tetromino and next tetromino (from the set `zsjliot`). The action is encoded by an orientation index (for rotation of tetromino) and a column index. Here is an example of the demonstration syntax:

```
96/48/48/48/60/48/16/16/48/32/56/48/56/16/48/24/0/0/0/0|j|l:0,4
```

Helper functions for loading demonstrations from a file into memory are still in progress.

3.3 Sir Tet architecture

3.3.1 Game classes

The `TetrisGameState` consists of a `TetrisBoard`, a current `Tetromino` and the next `Tetromino`.

A `TetrisBoard` is internally represented as a list of lists, but can be queried by indexing it with a coordinate tuple (`board[x, y]`). The return value of this indexing operation is truthy if the cell is occupied and falsy if it is empty. Spaces outside the board are empty.

`Tetromino`es can be indexed with an orientation index to receive an `OrientedTetromino`, which itself can be indexed relative to its bottom left cell with a coordinate tuple to check if that cell is occupied by the tetromino in its current orientation. For example, a `z` piece in orientation `0` will return `1` when indexed at `(1, 0)`, `(2, 0)`, `(0, 1)` or `(0, 2)`, and `0` at any other coordinate pair.

Moves are represented by pairs of integers representing orientation index and column index.

3.3.2 Useful methods

An agent may use the following methods to aid in making its decisions:

`TetrisGameState.generate_move_context(orient, col)`: given a move, return `gameover, dummy, yi, cleared`. `gameover` is a bool which is `True` if the move resulted in a game over, and the other return values will be `None`. `dummy` is an instance of `TetrisBoard` with the move completed. It can be modified arbitrarily by the agent. `yi` is the column index at which the bottom left corner of the tetromino was placed. `cleared` is a list of row indices that were cleared by the move.

`TetrisBoard.test_place_tetromino(tet : Tetromino, orient, col)`: given a tetromino and a move, return `dummy, yi, cleared` or raise `GameOver`. Used by `TetrisGameState.generate_move_context`.

`TetrisBoard.score(cleared)`: given a list of cleared line indices, return the score rewarded.

`TetrisGameState.get_moves()`: generator that yields all valid `(orient, col)` pairs. Note that moves that can cause a `GameOver` are still considered valid.

Other methods of a `TetrisGameState` or its `TetrisBoard` should not be used by agents as they may mutate the controller's state.

3.3.3 Writing features

Feature extractors of a state or state action pair may use the methods discussed above. Many features are implemented in `features.py` and `features2.py` (the latter of which has cleaner code). Here is an example feature:

```
def holes(state : TetrisGameState, orient, col):
    '''
    count number of holes in board resulting from action (normalized to [0, 1])
    '''

    #this preamble is standard for move-agnostic features
```

```
gameover, board, _, _ = state.generate_move_context(orient, col)
if gameover:
    return 0

h = 0
for x, y in board.coords():
    if not board[x, y]:
        for ty in range(y + 1, HEIGHT):
            if board[x, ty]:
                h += 1
                break
return h/200
```

3.3.4 `Runners` and `UserInterface`

The Sir Tet playground consists of three `Runners` found in `*_runner.py` which control the `UserInterface` object responsible for rendering the screen to the user. The latter has methods that make extending the platform more tractable, allowing easy displaying of text, prompting for strings, and manipulating the visible Tetris game.

4 Tetris Agents

After the poor performance of the linear Q-learning agent from a few months ago, where there was difficulty in selecting a good feature set and tuning hyperparameters, I wanted to try the approach of taking the proven feature set of Dellacherie (that still outperforms many more complex approaches) and seeing if a non-linear, neural optimization function would achieve better results than the linear function.

4.1 Dellacherie's agent

Dellacherie's agent, as described earlier, simply picks the move that optimizes the following formula:

$$\text{eroded piece cells} - \text{col transitions} - \text{row transitions} - \text{landing height} - \text{cumulative wells} - 4 \cdot \text{holes}$$

Some of these features are move-agnostic (i.e. features of the resulting board state only) while others are dependent on the move.

`eroded_piece_cells`: number of lines cleared by move * number of bricks of placed tetromino cleared (e.g. if four lines are cleared by an `i`, this equals 16)

`col_transitions`: number of vertical pairs of cells which contain both an empty cell and a block, i.e. number of vertical transitions from block to gap

`row_transitions`: similar to `col_transitions`

`landing_height`: height at which placed piece lands (=`yi` from `generate_move_context`)

`cumulative_wells`: $\sum_{w \in wells}(1 + 2 + ... + depth(w))$ where a well is defined as a vertical sequence of empty cells with blocks on the left and right.

`holes`: number of holes where a hole is defined as an empty cell with at least one block above it in the same column.

The implementation of this agent was simple once the features were implemented:

```
class DellacherieAgent(BaseTetrisAgent):

    agent_name = 'Dellacherie\'s legendary hand coded agent'

    weights : 'dict[Callable[[TetrisGameState, int, int], float], float]' = {
        eroded_piece_cells: 1.0,
        col_transitions: -1.0,
        row_transitions: -1.0,
        landing_height: -1.0,
        cumulative_wells: -1.0,
        holes: -4.0
    }

    def __init__(self, logger):
        self.logger = logger

    def get_best_move(self, state: TetrisGameState) -> 'tuple[int, int]':
        moves = list(state.get_moves())
        move_scores = {(orient, col): sum(w*f(state, orient, col) for f, w in self.weights.ite
            for orient, col in moves}
        return max(moves, key=lambda m: move_scores[m])
```

4.2 Deep Q-Learning (naive)

This first approach to Deep Q-Learning uses a Q-value estimator network which takes the extracted features of a state-action pair as inputs and outputs a single Q value. The feature set was kept the same as Dellacherie's. The model consists of two fully connected hidden layers with 16 hidden units. There is also a target network to which the weights of the estimator are copied every 100 training steps. On each training step, the estimator chooses an action using the ϵ-greedy approach, and then calculates the loss

$$l = (r_{s,a,s'} + max_{a'}\hat{Q}(s', a') - Q(s, a))^2$$

where Q is the estimator model, $\hat{Q}$ is the target model, and r is the reward obtained in the transition resulting from the chosen action. This loss is used to update the weights of the estimator network using the `keras.optimizers.Adam` optimizer.

4.3 Deep Q-learning with Experience Replay

After some reading, I realized that one common approach in Deep Q-learning is to randomly sample from past experiences to mitigate some of the correlation found in a standard sequential learning approach. This agent stores a buffer of observed state transitions (storing the input to the network instead of the actual state action pair), and at each training step samples a minibatch of experiences from the replay buffer, calculates the loss of the entire batch with respect to the current states of the estimator and target model, and performs optimization with these losses.

4.4 Results

The Deep Q learning agents were trained for 8 hours each on approximately 3500 games. The experience replay model was trained with three different sets of hyperparameters, the best one being `copy_iterations=250` and `minibatch_size=64`.

Agent	Average score (last 12 games)	High Score
Dellacherie	41560	194060
Deep Q Learning	365	2940
DQL with ER	423	3180

5 Discussion and future work

The results are surprisingly poor considering the feature set used was the same as Dellacherie's agent. I expected the networks to be able to at least match Dellacherie's performance by approximating a linear function of the features.

One of the reasons for this poor performance might be insufficient training time. Even after eight hours of training, the average losses were in the hundreds. This might indicate that longer training times or larger batches of experiences should be used, or some of the learning rates need to be tweaked to obtain faster convergence. Alternatively, maybe the target network is still moving too fast for the estimator to converge to it. I was unable to test all of these factors due to time constraints.

A possible improvement might be made by implementing Deep Q-Learning from demonstrations. Kick-starting the learning process with a set of user-recorded demonstrations might help the model by starting it off with a reasonable understanding of the strategy and allowing it to catch up to Dellacherie in performance. Unfortunately, because of time constraints, I was unable to implement the usage of recorded demonstrations in this project's timeframe.

Of course, it's possible that the feature sets used in these models are the culprit–maybe Dellacherie's features are not suited to a complex non-linear optimizer. Maybe reducing the complexity of the network or trying different feature sets would have some positive impact on performance.

Lastly, it's possible that Q-learning based approaches are simply infeasible for Tetris. This might be a result of the long sequences of actions without any rewards making it difficult to quickly get accurate Q-value estimates. Perhaps some form of reward shaping might combat this, but perhaps the reason the literature is light on Q-learning for Tetris is that it simply isn't a good solution for the problem.

Hopefully whoever next tackles this problem will find the Sir Tet Playground and the agents I implemented useful.

6 References

[1] Fahey, C. P. (2003). Tetris AI, Computer plays Tetris. http://colinfahey.com/tetris/tetris_en.html

[2] Bohm, N., Kokai, G., and Mandl, S. (2005). An Evolutionary Approach to Tetris. *The Sixth Metaheuristics International Conference (MIC2005).*

[3] Da Silva, R. S., and Parpinelli, R. S. (2017). Playing the Original Game Boy Tetris Using a Real Coded Genetic Algorithm. *2017 Brazilian Conference on Intelligent Systems.*

[4] Lagoudakis, M. G., Parr, R., and Littman, M. L. (2002). Least-squares methods in reinforcement learning for control. *Hellenic Conference on Artificial Intelligence.*

[5] Chen, X., Wang, H., Wang, W., Shi, Y., and Gao, Y. (2009). Apply ant colony optimization to tetris. *Proceedings of the 11th Annual conference on Genetic and evolutionary computation.*

[6] Langenhoven, L., Van Heerden, W. S., and Engelbrecht, A. P. (2010). Swarm tetris: Applying particle swarm optimization to tetris. *Evolutionary Computation (CEC), 2010 IEEE Congress on.*

[7] Thiery, C., Scherrer, B. (2009). Building Controllers for Tetris. *International Computer Games Association Journal*

[8] Carr, D. (2005). Applying reinforcement learning to Tetris. *Rhodes University*