Quadris - A Deep Q-Learning Tetris Bot

Grant Chau

***Abstract*— In competitive video games, a player's playstyle can be represented as a set of biases towards actions that link each successive game state to the previous one. This brings me to Q-Learning, where an agent is trained from scratch, with its main goal being to maximize its total reward. Thus, if a reward system is specifically engineered to promote bias towards certain actions in certain states, a novel playstyle is created. Since reinforcement learning is well-suited to train an agent to play games, I implemented Quadris - a Deep Q-Learning bot that plays Tetris via *deep* Q-Learning. This way, given a vector of board properties, the bot can output an estimated Q-Value and execute the action that maximizes the estimated Q-Value. The primary goal of this project was to produce at least one bot that can clear 500 lines at a time, and playstyle simulation would only be possible once that first goal was met. After I had one functional bot, I created more and gave each bot different rewards values for successfully executing a Tetris. I then ran some tests to determine whether or not engineering different reward systems actually impacted an agent's bias towards a certain actions that are characteristic of their respective playstyles. I found that reward systems can be designed to promote certain behaviors, but I had to be careful not to reward an agent too much for any behavior, as this results in a playstyle that is solely defined on that action. However, all three bots are capable of clearing over 500 lines, which was the main goal of this project.**

I. Introduction

I'm a huge fan of competitive games, as I've been playing fighting games nearly my whole life. In particular, I'm interested in how one's playstyle in-game is reflective of their personality and overall philosophy towards their game of choice. In general, one's playstyle is a pool of biases that a player considers (either voluntarily or involuntarily) when evaluating the game state and choosing their next course of action. Aggressive play is characterized by biases towards moves that are more likely to leave the opponent in a disadvantageous state, and patient play is characterized by biases towards moves that are more likely to maintain the relative value of their current game state in hopes of generating a poor reaction out of their opponent and capitalizing off of their mistakes.

In this project, I wanted to explore this idea of representing a playstyle discretely. That is, I wanted to see if I could engineer two different reward systems, one which rewarded aggressive play and one which rewarded patient play, and see if these playstyles were apparent in the choices made by the two resulting models that were trained off of each respective reward. Since Tetris is traditionally not a zero-sum game (usually, there is no winner or loser), I had to consider an engine such as Puyo-Puyo Tetris', which sends over $\lfloor lines\ cleared \rfloor$ to the other player each turn. My first goal, now, is to train an AI to play Tetris.

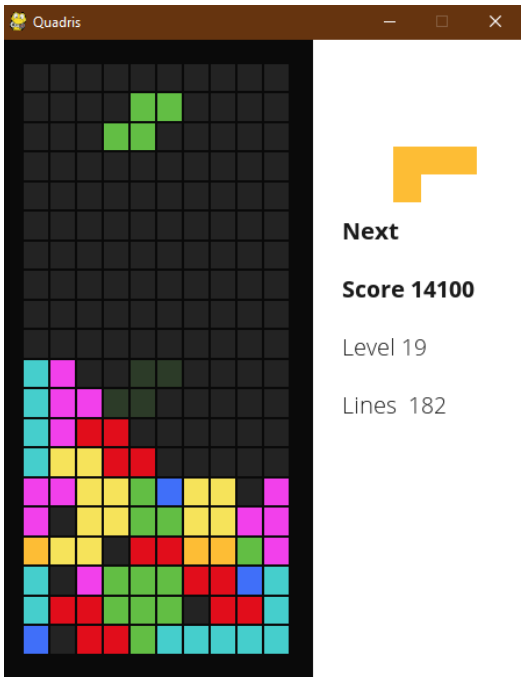

Fig. 1: My UI was made in pygame and it displays the board, the next block, the current score (not to be confused with reward), the level number, and the total number of lines cleared. I had no prior pygame experience, and I used [2] to learn the engine and used [3] for reference when coding the game's rules. Simply put, the goal of Tetris is to place blocks to create rows of colorful tiles, which causes them to disappear so you can place more blocks. The game ends when the board does not have enough space for the next block.

For training, I took inspiration from the DeepMind Atari paper [1] that we discussed in class. In their paper, they used a version of deep reinforcement learning called Deep Q Networks (DQNs). In particular, Q-Learning is well suited to the game "Breakout" and Tetris. This is because future rewards are generally preferred over immediate rewards in both games (clearing one brick vs. clearing the whole back row of bricks and clearing one line vs. clearing four lines at once).

As they described, Q-Learning is based on the concept of reinforcement learning, which can be observed in humans. Reinforcement learning, put simply, is a process of trial-and-error, in which actions with larger positive outcomes are generally rewarded. This makes Q-learning a slow (in the case of this project, *very* slow) but very intuitive method of learning. To emulate this process of trial-and-error, I utilized a technique called *experience replay* that uses a random sample of prior actions instead of the most recent action to proceed in order to smooth changes in

the data distribution. Furthermore, I used deep reinforcement learning rather than supervised learning, since I wanted my models to formulate their own playstyles rather than to simply copy a professional's. For learning, I used a neural network rather than a convolutional neural network (which is what DeepMind used in for their Atari experiments), since neural networks preserve the order of the input data. This is important, since I'll be feeding the network input data in a fixed-order vector of numerical properties that are extracted from the board. For this project, I implemented *(2 points)* a Q-Learner with discretized actions, states, and policies and had it control *(1 point)* a player in a video game. I also made a visual UI [Fig. 1] *(1 point)* to display the game.

II. Methodology

A. Neural Network Architecture

As mentioned in the previous section, I implemented a version of reinforcement learning called a Deep Q-Learning Network. Each agent has a neural network that will take board properties as inputs (more on this later) and output corresponding Q values. Since my goal was to input a list of four board properties and receive a number (Q value) as output, a neural network was an appropriate choice. The neural network used in my DQN is a sequential model with two hidden layers, each with 64 densely connected neurons and ReLU activation functions between the layers, with the exception of a linear activation function between my last hidden layer and output layer. I used two hidden layers, since, due to the various board properties in Tetris, Q value prediction is likely not a linearly separable problem. In order to determine how many layers and nodes to use, I experimented with various architectures, and I found that two densely connected hidden layers with 64 nodes each gave the fastest Q-value convergence times.

B. Deep Q-Learning vs. Vanilla Q-Learning

The Q-Table that corresponds to a complex game such as Tetris would explode in memory. Thus, we use deep Q-Learning to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. In terms of reinforcement learning, all the past experience is stored in an experience buffer. The next action is then determined by the maximum output of the neural network. Finally, the loss function determines the error between the predicted Q-value and the target Q-value. Since we do not know either the target or the actual value, we must go back to the Q-update equation derived from the Bellman equation

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\mathbf{R_{t+1}} + \gamma \mathbf{max_a Q(S_{t+1}, a)} - Q(S_t, A_t))$$

where the bolded represents the target Q-value.

C. Rewards

In order to produce a model that exhibits the desired behavior, we must engineer a reward system that would promote tendencies towards certain behaviors. For example, we would prefer to clear four lines to one line at a time, so we use a relatively large discount factor ($\gamma = 0.95$) and reward the number of lines cleared quadratically. In order to simulate biases towards certain actions, I slightly tweaked the reward for getting a Tetris in the models that were supposed to represent aggressive play.

Algorithm 1 Deep Q-Learning Algorithm Overview Using an Experience Buffer

```
Initialize experience buffer B to capacity N
Initialize action-value function Q with random weights
Initialize target network Q'
for game = 1, M do
  Initialize ε with a ε decay variable for exploration
  Choose an action a, observe reward r, next state s', and
  termination boolean terminal
  Store transition (s, a, r, s', terminal) in the experience
  buffer B
  if enough experiences in B then
    Sample a random batch of n transitions from B
    for every transition (s_i, a_i, r_i, s'_i, terminal_i) in B do
      if terminal_i then
      y_i = r_i
      else
        y_i = r_i + γ max_{a'∈A} Q'(s'_i, a')
      end if
    end for
    Use MSE to calculate the loss
    Update Q using the adam optimizer
    Every C steps reset Q' to Q
  end if
end for
```

Playstyles and Rewards			
Lines Cleared	Balanced	Aggressive	Hyper Aggressive
0	1	1	1
1	10	10	10
2	40	40	40
3	90	90	90
4	160	200	250

D. Epsilon and Exploration

In order to help the network continue to search the game state and avoid getting stuck in local optima, I used an exploration variable to determine whether or not the agent should select the action that leads to the highest Q-value. After some experimentation, I found that Q-Values tended to converge after around 1700 games. Thus, I used the formula

$$\epsilon = max[episode_{stop} - episode_{current}, 0](\epsilon_0 - \epsilon_{final})/episode_{stop}$$

where $episode_{stop}$ is 1700, $\epsilon_0 = 1.0$, and $\epsilon_{final} = 0$ to compute epsilon. Clearly, ϵ decays after every episode. I then sampled a float distributed uniformly in the interval [0, 1) and explored if the sampled float was less than or equal to ϵ.

E. Inputs to the Neural Network

Since I'm essentially dealing with a classification problem, I needed a way to describe the board to the agent in order to get an appropriate output. I originally considered 12 board features, and this greatly increased training time. In order to determine which were the most significant, I consulted a friend who is a top 500 Tetris player in America. Between my twelve properties, he said the most important three were *easily* (1) lines cleared after a piece is placed, (2) holes in the board, and (3) how bumpy the "surface" of the board is. He also added that, in versions of Tetris where you play against other people such as the aforementioned Puyo Puyo Tetris, (4) the sum of all the row values of the surface block in each column is an important consideration.

F. Training Loop Summary

I will now describe my main training loop, which will largely be a summary of what I presented in this section. I start with a large number of episodes (2000), each episode representing an entire game of Tetris. After determining whether or not it should explore, the agent chooses a state, with each state being represented as a tuple ((x translation, n rotation), [lines that can be cleared, number of holes, bumpiness, sum of the "peak" values]) from the available states by inputting the vector of board properties and choosing the action that maximizes the predicted Q-value. The agent then plays this action on an emulated controller, and the resulting reward and termination status of the game is collected. These features are then appended to the experience buffer in the form of the tuple $(s_i, a, s_{i+1}, r, terminal)$. The current state is then updated, and this continues until the game ends. After 512 new experiences have been collected, the agent then samples a batch of experiences to train its neural network by fitting each state in the batch to the desired output.

III. Experiments and Results

A. Playstyle Experimentation and Versus Mode

As I described in the previous section, I trained three distinct models in an effort to simulate differing playstyles. To experiment and test functionality, I ran each of them separately until they were able to clear 100 lines. Out of 10 trials each, all three models successfully cleared 100 lines in every trial, although the hyper aggressive model only made it to line 87 in an unrecorded trial. The hyper aggressive model has notably poor survivability, as its average game length in 10 trials was approximately 263 lines (in contrast to the other two models, which are capable of clearing 1000+ lines and did so in 10 separate trials).

When testing the models in versus mode, the aggressive model is a clear victor, as the balanced model does not send enough lines to its opponent and the hyper aggressive model spends a few moves at a time setting up Tetrises (although in many cases, the agent doesn't end up clearing all lines due to holes). While this is an optimal strategy in classic single-player mode, this agent's tendency to set up large structures near the edges of the board caused it to die rather quickly, as versus mode requires considerations of defense and survivability and not just line output. When put this way, the aggressive agent seems to be the obvious winner in a versus setting, as it sets up Tetrises when it can yet it also maintains a flexible board state, increasing its survivability over time. While the balanced agent certainly performs well in both modes, its focus on survivability over clearing many lines at once certainly hurt it in a versus setting that requires line output.

B. Training Results

While training, I found that Q-Values took many iterations to converge. In order to determine what episode I should end training and exploration on, I collected two data points for each model: the length of their games and their scores. The results are depicted in Figures 2 and 3.

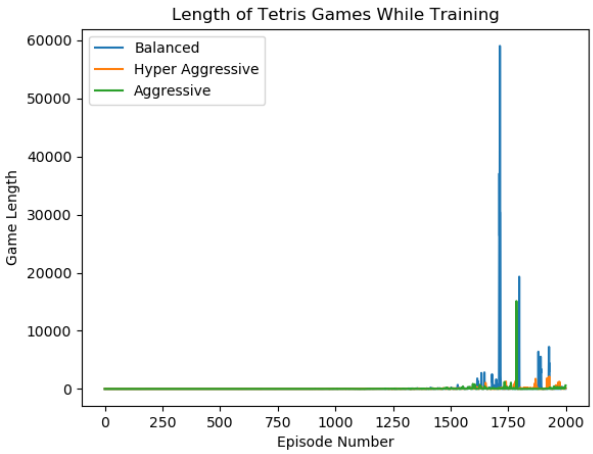

Fig. 2: Lengths of Tetris games with respect to the game number.

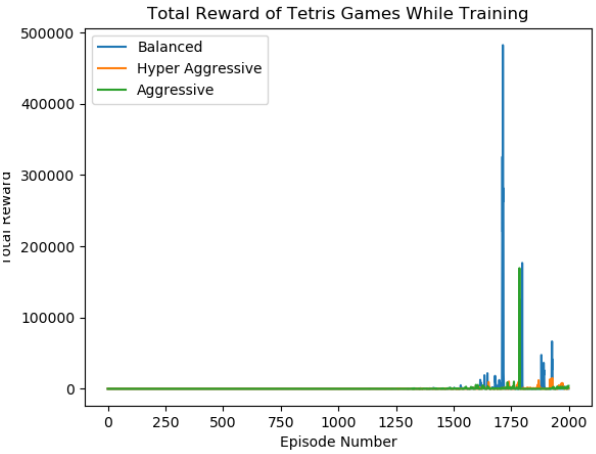

Fig. 3: Rewards of Tetris games with respect to the game number.

As expected, the balanced model has the best survivability, as it had a maximum game length of 58993 moves. On the other hand, the hyper aggressive model had the least maximum game length of 2149 moves. The aggressive model

had decent survivability as well, with a maximum game length of 15121 moves. The total rewards per game are directly proportional to the game's length, as clearing lines gives points, and clearing lines means longer games. Thus, the trends for total reward are the same with the balanced model having the highest maximum points, followed by the aggressive model then finally the hyper aggressive model.

C. n Line Clearing Frequency for n = 1, 2, 3, 4

To see if each respective model produced the desired behavior, I also ran a test where each model played a game of length 300 then measured the frequencies of one line clears, two lines clears, three line clears, and Tetrises. As illustrated in Fig. 4, the hyper aggressive model had quadruple the number of Tetrises as the the balanced model. Furthermore, it had more three line and two clears than the other two models. This graph explains the aggressive model's outstanding performance in versus mode, where survivability and line output are both important; it has a high number of one-line clears but also clears two, three, and four lines relatively frequently.

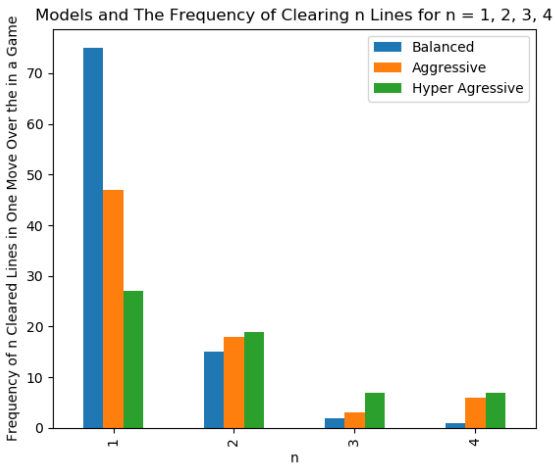

Fig. 4: Frequency at which each model clears n lines.

IV. Conclusion

After completing this project, I learned that, to an extent, playstyles can be partially simulated by carefully engineering reward systems that promote biases towards certain in-game actions. However, much experimentation (adjusting epsilon stop values, neural network architecture, input space, rewards, etc.) must be done in order to produce meaningful results. I also learned that the reward systems that you present the agent with are just as important as the agent's architecture. Without carefully designed rewards, the agent might demonstrate a clear bias towards certain actions, and this may cause a variety of problems such as only setting up for (Tetrises then not having an 'I' block ready in time to execute the Tetris). However, my most stable reward system was able to produce a bot that can clear 50,000+ lines without a game over, and ending exploration at around 1700 episodes produces a well-performing bot with convergent Q-values. Furthermore, a slight increase in reward produced the desired behavior, as my hyper aggressive bot consistently executes more Tetrises than the other two.

This project was extremely satisfying, but generating memories to insert into the experience buffer was time-consuming. This made it especially frustrating when adjusting various parameters for the neural net, and I finally settled on the current neural network architecture after around 20 hours of experimentation. I'm also extremely grateful to my friend who helped me decrease the number of inputs (board properties) from twelve to four, as training and evaluation sped up tremendously after altering my engine and network based off of his recommendations.

V. Appendix

Source code for my implementation is available at https://github.com/gnchau/Quadris. To train your own model, run the `train_learner.py` file, and add a model path called `<model_name>.h5` as a parameter to the `run` function. You should also give the `run` function file names for text files containing the model's score and game length per game. For example, the `run` function should something like `run(Tetris(), QNetwork(), "scores.txt" "length.txt")`. Once ran, the terminal will display trianing information. Training took me around 2 hours for 2000 episodes, since my computer is not CUDA enabled. If you wish to pause and resume training, you can quit out of the terminal then rerun the `run` function with the parameter `epsilon_checkpoint=True`, which simply sets the initial epsilon to the saved epsilon in the text file. After training, you can run `ai_play.py` to watch the balanced model play Tetris. If you want to watch a different model play, change the `model path` parameter to the name of the model you wish to watch. If you want to watch models play in versus mode, run the same file after changing the `single_player` parameter in the `play` function to `False`. If you want to generate graphs similar to the ones in Section III, run `graph.py` in the 'analysis' folder, and feed in the names of the files you wish to graph from as parameters in the `plot_from_file` function. Since the neural network is initialized with random weights, your graphs might look slightly different from mine but choosing an $epsilon_{stop}$ `= 1700` should cause a similar large spike in performance around the 1700th-1800th games. Running `block_frequency.py` construct controllers with which the main three models play separate, 300 line-long games. After the games are played, a graph similar to Fig. 4 will appear. If you wish to alter the length of the game or use different models, you may make these specifications in the `graph_frequencies` function.

References

[1] *https://arxiv.org/pdf/1312.5602.pdf.*
[2] *https://www.youtube.com/watch?v=zfvxp7PgQ6c.*
[3] *https://github.com/Uglemat/MaTris.*