15381 Project Report: Artificial Intelligence for Breakout and Tetris

Rishub Jain
CS
rishub@cmu.edu

Fanyu Kong
CS&Math
fkong@andrew.cmu.edu

Varun Sharma
CS
varunsha@andrew.cmu.edu

Project Goals

The primary goal of our project is to develop AI for simple games. We would try to train AI for breakout and tetris first and analyze the result we get. Hopefully we can generalize an effective systematic method to build AI for other games of similar complexity and apply this technique in a broader use. We got motivated from the online AI block battle challenge at theaigame.com. It seems like a lot of fun but when we started we realized that it is a bit too complicated to start with, both in terms of the complexity of the game and the difficulty of training. So we decided to start with something simple and turned to tetris and breakout. We thought that would give us a basic understanding of how algorithms could perform in this sort of scenarios and the possibility of future generalization. Starting with simpler games also gives a more direct refection of the performance and is also easier to train and tune. We hope to carry those experience onto future projects and applications. Lastly, we decided to do this because we would be excited and feel accomplished if our AI turns out to be rather successful at the games.

Project Approach

Our original plan was to create an AI for the game AI Block Battle from The AI Games. However, after looking more into it we realized there would be a lot of issues with doing this. The way the moves are handled does not correspond exactly to a Markov Decision Process, since we have to take many actions at a time of variable length. In addition, the simulation of the game seemed slow, and the complexity of the game was very high as there are many subtle rules that our model would have to learn. Therefore, we opted to first create an AI for the simpler games Tetris and Breakout.

Our approach includes three steps mainly. We first find existing implementation of tetris and breakout, and adapt the code to the interface of our machine learning algorithm. This is where we abstract the representation of state information and measure of rewards. At the same time, we implement the training algorithm themselves. In this case, we decided to train the AIs both by DQN and policy gradient to make a comparison and hopefully figure out the more suitable algorithm for this kind of task. The last thing we do is to just train the AIs and let them play and observe the results. We decide based on that if more training would be helpful and how much if so, or if we should consider adjusting our algorithm, or tuning parameters like the reward. We divided the task into these steps so that we could easily adapt the methodology to building AIs for other games without too much work.

Heuristically we think RL should be a suitable algorithm here over other options since the concepts of state space, action and reward really seem to capture the essence of tetris, breakout and really any game of similar complexity, so it should be pretty powerful. It is also because all of us had some prior experience with RL algorithms so it felt natural when it came to mind. Specifically, we chose to use the DQN algorithm because it is known to achieve nearly state of the art performance, and we also had some experience with it. We chose the REINFORCE algorithm because its Monte-Carlo Policy Gradient variant was simple to understand, and we wanted to compare our DQN performance with something much simpler.

Project Progress

We had pivoted from our original idea of creating an AI for AI Block Battle because it seemed too challenging.

After we found code online to simulate Tetris and Breakout, we had to modify it such that it would have two functions, reset() and step(action), in addition to on-demand visualization capabilities.

Then, the next challenge was to find a good measure of the reward of the game. This could be tricky in the sense that it will have a huge impact on the result of the training. Originally our reward function was simply the change in score after each time step. After running this, exploring different hyper parameters, we could not achieve good results on either Tetris or Breakout, on either the DQN or the REINFORCE algorithms. We realized the way our state and reward were being represented were the main reasons why it wasn't working.

For both Tetris and Breakout, we changed the reward function to be a combination of different heuristics. We had went through many iterations, adding and removing expressions depending on how our trained breakout model was doing. In addition, we modified our original state space to be more discretized.

After making these changes (explained in our Implementation Overview), we found that Breakout did well on the DQN network, but Tetris was still not performing well. The main problem with Tetris was that it was only placing blocks one on top of the other because it had not learned that clearing a line was much better, because taking its random actions did not lead to it clearing the lines. This was because our original set of actions were just left, right, rotate piece, and drop. We changed this to have 40 actions, corresponding to placing a piece in one of the 10 columns in one of the 4 orientations. This performed much better, and it was able to learn much more quickly that spreading out the pieces on the board actually increased this reward function.

The most important hyper parameter of the DQN network we explored with was the network architecture. Originally, we only had 1 hidden layer with 8 units for both games, and it seemed to perform poorly. Then, we increased it drastically to two layers. For Breakout, we had the first layer with 64 hidden units and the second layer with 32 hidden units. For Tetris, we had the first layer with 128 hidden units and the second layer with 64 hidden units. This change lead to much better performance.

Implementation Overview

For the implementations of the two games, we added step(action) and reset() functions to their interface and also changed the internal representation of the game state so that it could be output and passed to our training algorithm as a whole. We have defined all legal actions to be single integers. The step(action) function takes an action as an argument and simulates the game state after that action is executed. Finally we modify the main routine of the games to not interact with humans so that our AI could be trained and actually play the game. The implementations of the games use Pygame.

We also defined the reward function for each game. The reward function for Breakout was:

$$R(t) = (Score_{t-1} - Score_t) + 2 * ph_t + 10 * (Lives_t - Lives_{t-1}$$

where $Score_t$ is the score at time t for a given episode, ph_t is the indicator where its value is 1 if the ball hit the paddle at time t and 0 otherwise, and $Lives_t$ is the lives the game has at time t.

Our final representation of the state space for Breakout included the following values:

- Position of the paddle
- Position and velocity of the ball
- A boolean grid of which bricks were still left
- Lives left

Breakout's actions were just 0 and 1, corresponding to moving left or right respectively.

For Tetris, we chose the following reward function:

$$R(t) = 2 * (Lines_t)^2 + (Density_t - Density_{t-1})/10 + GO_t$$

where $Lines_t$ number of lines cleared at time t, $Density_t$ is the density of the rows at time t which is equal to the amount of blocks filled in the grid divided by the number of rows being occupied, and GO_t is the indicator with a value of -0.3 if the game has finished. This was to encourage prolonging the game, since this value would be lower later in the game with diminishing rewards.

Our final representation of the state space for Breakout included the following values:

- Position of the Tetris piece
- Type of the current Tetris piece represented by a 1 hot vector
- The rotation the piece is at
- Type of the next Tetris piece represented by a 1 hot vector
- A boolean grid of which blocks were being occupied

We made Tetris had the 40 actions described in Project Progress.

For the implementation of our DQN, we used the framework of the code that Rishub had written in another class. We had adapted it heavily for our use in these two games, as the code was tailored for the OpenAI gym use. For our implementation of REINFORCE, we used Adriel Martinez's code that was tailored for a different environment and changed it to work for our games.

Our DQN implementation used TensorFlow and Keras to create, train, and use the Deep NN. Our REINFORCE implementation also used TensorFlow. Both of our algorithms, as well as the changes we made in our game environments, used numpy to perform calculations.

Results

The total reward for each game at the beginning of training, when each move is mostly random.

Breakout Baseline: -7.62 ± 18.955

Tetris Baseline: 0.12 ± 0.26

The total reward for each game, with different network configurations after reaching stable performance after 10 hours of training.

DQN - 2 Layer NN for Breakout: 33.57 ± 42.56

REINFORCE - Breakout: -4.31 ± 23.28

DQN - 1 Layer NN Tetris: 2.43 ± 1.83

DQN - 2 Layer NN Tetris: 2.41 ± 2.19

The DQN implementation of Breakout did very well, playing the game almost perfectly. However, the DQN implementation of Tetris did not do as well, though it did do better than the baseline, because of the complexity of the game. The performance did increase with the more complexity we added to out network structure.

The REINFORCE algorithm did poorly on Breakout. This is probably because the algorithm itself was not able to exp

Analysis

Based upon our result, it is fair to say that our AI for breakout was rather successful after enough iterations of training. Although it does not necessarily exceed average human performance in the game, it is getting very close and is definitely intentionally catching the ball instead of doing something random hoping to hit a luck shot. Although the breakout AI performed well, its performance stagnated due to the particular network architecture. We initially tried a single layer architecture, which had performance, which leveled off sooner. AI for Tetris on the other hand is not performing as well. It is trying some basic approaches to score but it did not turn out to be very effective. It is still performing far worse than an average human player. We think that the main issue lies with the reward function and if we were to tune it to better reflect the situation of the game state the AI should be a lot smarter. The main problem with Tetris is that there are very many actions that can be taken and the reward for those actions is very delayed. The effectiveness could be improved by making a better reward function, which takes into account predicted future performance with better heuristics. The only heuristic that we used was the density of blocks. This heuristic improve the performance greatly, but performance could be improved even more by using more targeted heuristics to improve training. Different network architectures could also have improved performance significantly.

The success in breakout should prove the effectiveness of our approach of training AI for simple games. We think that the algorithms we used should be a solid choice in these applications in general. The true performance of the AI still depends greatly on how well the game states are represented and how effective the reward function is, but those are very situational dependent.

Role of Each Teammate

Risub mostly worked on adapting the code for the DQN and REINFORCE algorithms. Fanyu worked mostly on obtaining and modifying the code for the game simulations. Varun worked on both adapting the RL algorithms and modifying the game code, and computing the results and final output. We all equally worked on the presentation and final report.

External Resources Used

The main goal of this project was to use machine learning techniques to create AI's for games. Because of this focus, we used open source implementations of the games and then modified them extensively to be able to use them for our purposes by extracting the game state and input controls, changing the games to not be locked at a particular frame rate, and changing the visualization to be able to visualize only when needed to improve the performance of training.

The tetris implementation used was from silvasur on GitHub:
`https://gist.github.com/silvasur/565419`

The breakout implementation used was from a pygame tutorial on CodeTronix:
`http://codentronix.com/2011/04/14/game-programming-with-python-and-pygame-making-breakout/`

For the implementation of our DQN, we used the framework of the code that Rishub had written in another class. We had adapted it heavily for our use in these two games, as the code was tailored for the OpenAI gym use. For our implementation of REINFORCE, we used Adriel Martinez's code (found at `https://gist.github.com/Adriel-M/ca8a536983ec6df45dff96fd18599b74#file-cartpole-reinforce-mcmc-py`) that was tailored for a different environment