CS 383: Final Project

Aaron Giera

May 5, 2016

gitlab: https://gitlab.com/agiera/BlockbattleBot

For my final project I chose to develop an AI to play battle-blocks. I played this game frequently in high school, so I am very familiar with this game. I learned a lot from this project and I think it will help me later with my personal projects and I hope to have a career in which this knowledge is relevant. This was also a very interesting assignment with a lot of possible approaches. In order to communicate mine I will first give an analysis of the game moves, states, competitive play, and how humans play the game.

Because I have played this game before I was able to narrow down the possible moves to a reasonable number. Note that by never competitive I mean never optimal and therefore do not need to be considered. Whenever using down it is only competitive to go all the way down, so I condense these moves into a project down move so that there aren't twenty moves for every move that projects down at some point. It is never competitive to turn more than thrice. It is never competitive to move more than five times. Using this information I was able to narrow down to 81 moves. This is also lowered because if a move result in an invalid board I discount this move (for all resulting positions of a piece there exists a move that does not knock into blocks at any step). Other considered approaches include choosing possible positions of the piece and finding the optimal resulting board. I discounted this because it there are at best twenty resulting boards (this is already pretty close to the 81) and it is a complex task to find the move for one.

There are many approximately optimal ways to encode the states of the game play. Encoding the game in full is not one of them. Meaning recording the entire field and the pieces as digits. First there are states in this space that are not possible to achieve given the mechanics of the game and secondly there are states that are never competitive to be in. This resulted in the following encoders:

1. Flat encoder: TETbot/Bot/Strategies/Encoders/FlatEncoder.py

   This encoder has the benefit of containing all relevant information. But with this comes $2^{10*20} * 7^2 * 32$ states just considering the players board, the pieces the player has next and if they have a combo. I am assuming that the player will never be able to attain more than a 31 combo. In this paper, `http://www.mcgovern-fagg.org/amy/courses/cs5033_fall2007/Lundgaard_McKee.pdf` (given to me by professor Taylor (thanks)), by Nicholas Lundgaard and Brian McKee, we see taking the approach of using a straight forward neural network cannot work because it learns from each state independently and so cannot recognize similarities. The neural network would have to encounter all relevant states many times in order to learn and because there are so many it would take an unreasonable amount of time. An implementation of Q-learning with a sparse matrix for Q would not work as there would need to be approximately $2^{10*20} * 7^2 * 32 * 81 = 2.0409399e + 65$ entries in this table. The matrix would be either too sparse to be effective or too large to store.

2. Top heights encoder: TETbot/Bot/Strategies/Encoders/TopHeightsEncoder.py

   This encoder finds the highest block on the board and finds the heights of each column normalizing the highest column to 3 and using a minimum height of 0. This way each column needs two bits for its height to be accounted for. There are seven pieces so these take up three bits each for a total of $10 * 2 + 6 = 26$ bits of information needed to encode this minimalist state. For a Q-learning implementation only $2^26 * 81 = 5,435,817,984$ entries would need to be recorded. This would probably not take up too much space as not every entry would necessarily be used. Unfortunately I was not able to find an easy way to read tables from a disc or create sparse matrices. Also it is important to note that this representation cannot produce even an approximately optimal solution. In order to

be competitive in this game both combos and t-spins are a necessity. Combos cannot be done in this representation because two columns on the left or right must be left empty in order to clear lines fast. This representation is not able to see down these columns. It is also not able to see holes in its three lines of visibility. This means it cannot rotate a t block into them.

3. I settled on a mix of the two. The first offers detail while the second offers a way to relate states to one another. I think having both in the representation would help a neural network pick up on the game faster and better.

I decided to use reinforcement learning early on. This is a complicated game with many subtleties and giving extra rules for the robot to follow would help hone their skills. Once it understands competitive play these rules can be taken away and the robot can be rewarded purely on the points it receives. This is similar to how musicians can become great by first being disciplined and learn to play well before allowing themselves to break rules and be creative to produce beautiful works of art. A great analysis of the general rules of competitive play and how they can be coldly calculated is presented nicely in this article: `https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/`. Also from my previous explorations I found a guide made for humans to learn how to play extremely competitively written by a professional Tetris coach: `http://harddrop.com/forums/index.php?showtopic=3640`. My rough policy is written here: TETbot/Bot/Strategies/Gratifications/GameplayReward.py.

During this project I had the naive tendency to desire for my AI to work like humans as humans are very good at calculations. Although this may not be the immediately best solution I think some day there will be available the proper tools to model a useful AI after the way humans operate. While considering this I thought about the following ideas. Humans see all of the board and are able to relate similar states to each other very easily. Humans are able to calculate the next state given a state, move, and rules. Computers are very good at following rules and this game is very easily predicable. That is because Tetris is a one player game we are able to calculate many moves before knowing the opponents moves. This tells me that searching the decision tree is essential for any AI. From the human perspective we must be able to reason about each state and only search ones that look like nice states the robot has seen before. This lead me to the following algorithm:

```
choose_action(state, time_out):
  let t be a tree with the root being ((state, null), 0)
  let q be a queue containing only the root node
  let k be an integer
  while time != time_out:
    deepen_search(t, q, k)
  return max(t.children)

deepen_search(t, q, k):
  let q' be an empty queue
  for kid in q:
    for a in Q.choose_k_best(kid[0], k):
      sp = simulate_action(a)
      r = get_reward(sp)
      q'.enqueue(((sp, a), r))
            Q.learn(kid[0], a, sp, r)
      propagate_reward(kid, r)
  q <- q'

max(node):
  best = ((null, null), -infty)
  for kid in node:
    if kid[1] > best[1]:
      best <- kid
  return best[0][1]
```

```
propagate_reward(kid, r):
  if kid == null: return
  kid <- (kid[0], kid[1]+r)
  propagate_reward(kid.parent, r)
```

k can be chosen by the believed quality of the neural network Q. It is the branching factor of the decision tree. The moves with the best reward in the future will be rated highest and chosen in max. If Q always calculates the best move to be in the top k, this algorithm will be successful. Using searching as humans do helps funnel the actions of the neural network so it doesn't have to completely guess. It also lowers the brute work of the search by pruning using a neural network. Also each imaginary though of the robot teaches the neural network as it does in humans. This algorithm is very general and can be applied to any situation that is easily calculable.

My implementation didn't go so well. I got an error for importing numpy. Note that I also edited the engine to tell the program (via stdout) if it won or lost at the end of the game.

I was able to output games run locally in TETbot/games. The following are some final boards of games:

1.
```
000     0000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
 000000  00
  00000  00
0    0    00
           00
       0
          0 0
0 0   0 0 0
      0 0 0
 00         0
    0   0
 0   0      0
```

2.
```
00      0000
  00  0000  00
  00  0000    0
  0   0000    0
  0   00000   0
  0   0   0   0
          0  00
        0 0   0
       00  00  0
  0    00  0
  0  000  0
       000  0   0
       000  0
  0    00  0
  0    00       0
  0      0 0
  0      0 0
  00     0       0
  0    000     00
```

I plan on continuing the research on my deep Q pruning algorithm using FANN as a neural network. I have already started modifying code which is included in the git lab. I might hook it up to this game engine or another similar one. Over the summer I will do some benchmarking to see if this is a viable algorithm.