Algorithms for playing 2048

Related document: dev backlog

Writeup TODO list:

- Re-run distributed Ray PPO experiment on EC2 for ~12hr. Compute expected cost first :) -- against fast c++ bitvector board

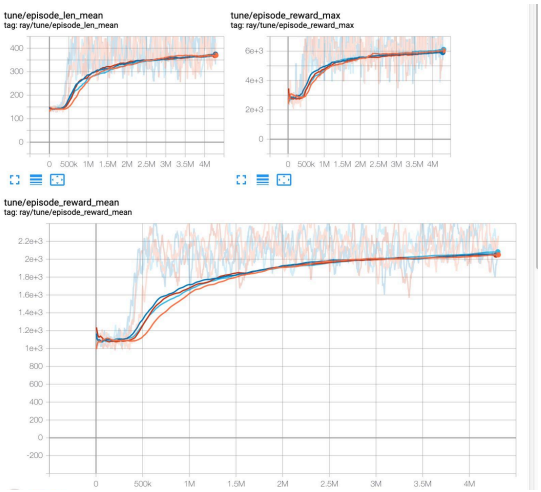
Draft writeup

Goals of the work:

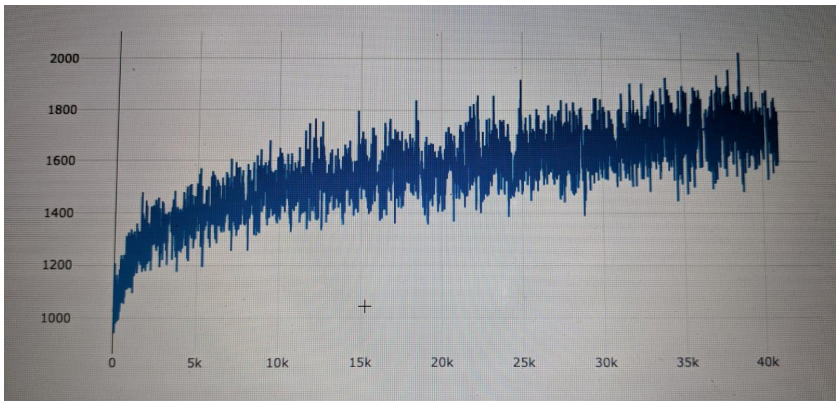
- Build agents that can achieve human-level game scores playing 2048
- Use known learning and AI techniques such as Reinforcement Learning, Dynamic Programming, and Deep Learning
- Analyze and understand dynamics of 2048 compared to other learning envs (e.g. state/action space comparison)

Experiments / Results

- Learning/planning per-step techniques
    - Take single sample per every leaf of depth 4 action tree, max of those + switch to random rollouts when board is space constrained (30k avg, 70k max) (Nick) (what Sutton & Barto call “Heuristic Search”)
    - Run MCTS on each step (15k avg score, 30k max) (Andy)
    - Run Ray on each step (1k avg score, 2k max) (Andy)
- Algos in Ray

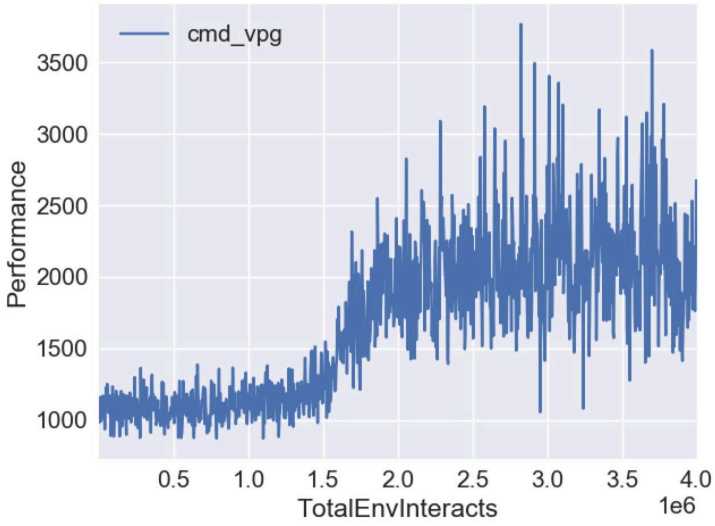
- 

  PPO on 4 GPUs + 40 CPUs (for, I think??, a couple of hours) -- experiment [cluster setup]() and [exec script (line 46)]()

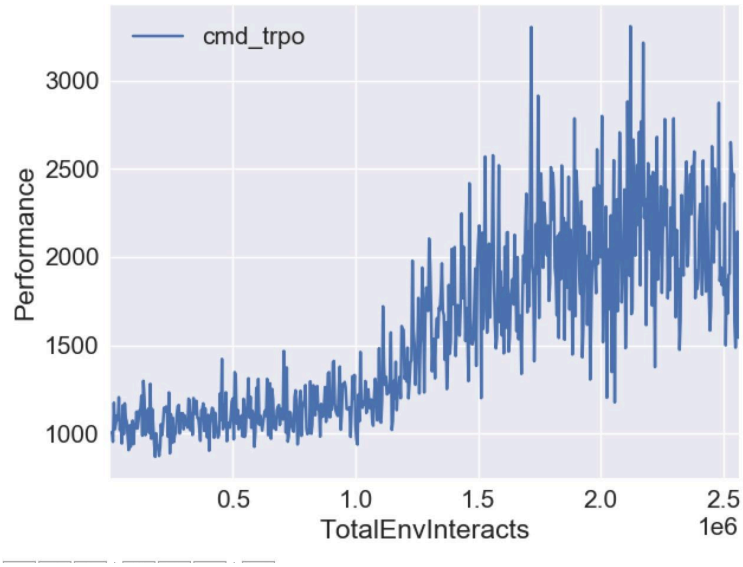
- 

  Ray PPO algo running on a single node for ~12 hours

  - A bunch of other Ray algos failed to run on our 2048 Env out of the box (e.g. threw exceptions)
- Tried using algos in OpenAI's SpinningUp
  -



- VPG

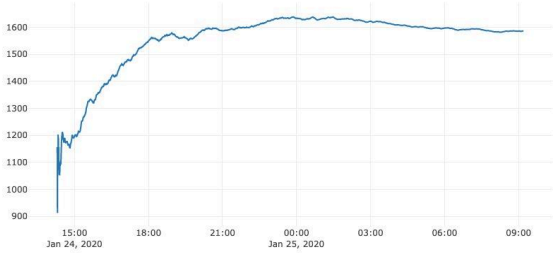


- TRPO

PPO

- Hand-rolled Algorithms
  - Q-Learning on canonical afterstates (Nick)
  - Hand-rolled DQN w/ TensorFlow (and also DQN in Ray) (Andy)

- Hand-rolled online Deep Q Learning (Andy) (avg score 1.6k after 24K episodes)

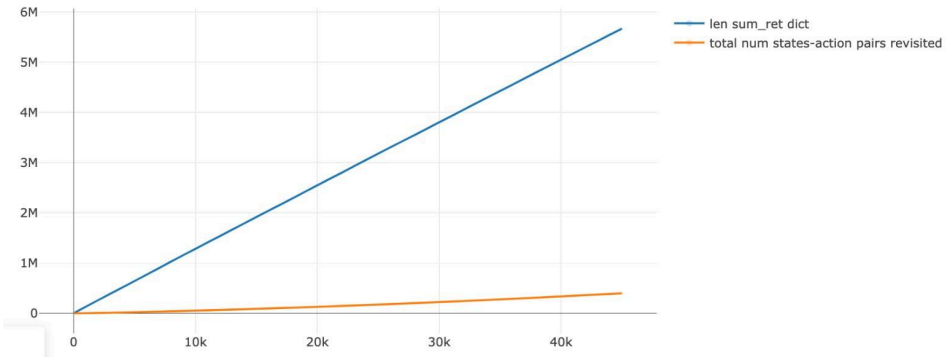


cumulative average; after running 30 hours on my macbook air

- Vanilla MCTS (Andy)
- Hand-rolled online Deep V Learning (using canonical afterstates) (Andy)
- Vanilla Policy Gradient?? (Andy)

Analysis of 2048

- # of initial possible board states:
- # of possible 2nd board states:
- How fast does state space explode?
  - TODO: generate plot with y axis = total # reachable board states; x axis = max tile of game (or max depth of game?)
  - TODO: given the size of the MDP and some reasonable assumptions around compute--how long to run DP/bellman algos to get the optimal policy (see also Chess/Go papers)
- How does it compare, e.g. to Chess, Go?
- What is the min score for each maxtile (e.g. min possible score to get the 2048 tile), max score for each maxtile, max score for a game
- plot of the hit/miss rates in tabular based MCTS algo (not using canonical states or after-states)



Performance of some simple strategies (i.e. policies):

- Only left (or right/up/down) gets avg of 13 points (median 4) per game. Median max tile = 4.

- Random strategy gets avg of 1.1K points per game (median 1.06K). Median max tile = 128.
- Good human players with some practice can get up to the 8192 tile, or a score of 100K+ per game.
    - NOTE: We aren't sure what a good human "average" is, just the current Jakey max tile.

Canonicalization and After-states

Simplified 2048 (aka "Easier 2048"?)

- To get a sense of what makes 2048 hard, we implemented two types of simplifications to the game dynamics: (1) deterministic state transitions (as function of (curr_state, curr_action), (2) max_depth games.
    - Another way to simplify the game is to reduce board size. See also "Blog about 2048 MDP" in related work below.
- We calculated tables (see here) of max scores and max tiles given different depths and different random seeds (a given random seed makes the state transition deterministic). We also computed the distributions of max scores and max tiles for a given depth across 100 different random seeds to get a sense of what the best score and max tile is for an average game when it's restricted to a given depth.
    -

Results by Andy, generated with `contrib/max_score_calculator.py`, using 100 different random seeds and showing stats per each depth over the 100 runs, using pruning for performance speedup:

```
Depth: 1
Max max tile: 3.08 mean, 2.0 med, 1.3090454537562863 std, 8 max
Max score: 1.72 mean, 0.0 med, 2.209434316742635 std, 8 max
total_state_action_pairs: 4.0 mean, 4.0 med, 0.0 std, 4 max

Depth: 2
Max max tile: 4.44 mean, 4.0 med, 1.251559027772962 std, 8 max
Max score: 4.44 mean, 4.0 med, 1.251559027772962 std, 8 max
total_state_action_pairs: 14.72 mean, 16.0 med, 1.8659046063504956 std, 16 max
```

- Results generated by Nick for single random seed: 42

```
$ python contrib/depth_limited_search.py

DFS with Depth Limit 20 and random seed 42

Depth: 1:
        Max Tile: 4
        Max Score: 4
        Total State Action Pairs: 3
        Depth Time: 0.0 sec
Depth: 2:
        Max Tile: 4
        Max Score: 4
        Total State Action Pairs: 13
        Depth Time: 0.0 sec
```

Related work

- Expectimax + very efficient board (C++ bit representation)
- Temporal Difference learning with n-tuple + canonical after-states
- StackOverflow discussion by a number of AI/heuristic developers
- John Lees-Miller blog series:
  - Minimum Moves to Win with Markov Chains
  - Counting States with Combinatorics
  - Counting States by Exhaustive Enumeration
  - Optimal Play with Markov Decision Processes
  -
- Reddit discussion of max tile possible

Lessons we learned

- 2048 has a lot of randomness (e.g. n-tuple paper doesn't use epsilon greedy because already sufficient randomness in 2048 env dynamics)
- Most algorithms we tried don't achieve more than 2x after training for 48hr
- The most successful algorithms (ours and related work) do planning from scratch after each move of the game.
  - This seems to suggest that next 5-10 moves are much more significant determinants of game outcome (score/max tile) than larger depth simulations
  - This seemed true in Dynamic MCTS which didn't do better when it ran much longer (10hr/game) at deeper depth (max depth 10 or 20 vs 4 or 5)

- State-of-the-art RL algos (in this case, PPO in Ray) doesn't do as well as you might expect when you only have a small budget for compute resources, e.g. relatively new multi core laptops and < $100 dollars of EC2 resources.
- Humans quickly learn (or, at least, hypothesize) heuristic higher-level strategies (e.g. keep the larger tiles in a corner). Most standard RL algorithms start with a blank slate and can only "learn" what to do by revisiting a state multiple times. Humans are able to strategize in never-before-seen states through (presumably) analogy to other, similar states + heuristic strategies. This suggests that 1) more training time and 2) collapsing the state space (e.g. through canonicalization and/or heuristics) improve the results of RL algorithms
- Debugging is difficult, for example: An algo is running and generating unimpressive results. Is the algorithm implemented correctly? Is it learning at all? Does it just need $1M more of compute resources? Where are the bottlenecks? What kind of function approximator should you use? How to choose hyperparameters -- hyperparameter search is itself an extremely large space to explore.