# Learning Grid Node-Based Games

Akahay Gowrishankar, Daniel Nee, and Devin Chotzen-Hartzell

June 8, 2021

## 1 Introduction

Node-based two-player adversarial games are played on a graph, where each vertex has an owner, a weight, and a growth rate. The weight represents the owner's strength in a node, while the growth rate increases the weight each term. To play, players move weight along one edge of the graph at a time, with the goal of controlling a certain set of vertices. The resulting weight at a node is the sum (if the players are the same) or difference (if the players are different) of the previous weight at the node and the weights moved into the node during that turn.

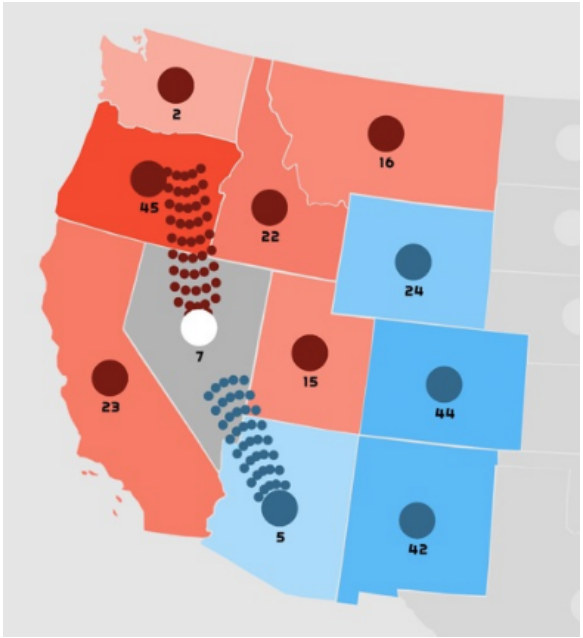



Figure 1: The mobile game *states.io* is an example of a fully-connected node-based game played on a map of the Western US.

This schematic is common in many multiplayer online strategy games, including the *Europa Universalis* and *Civilization* franchises. The goal of this project is to apply state-of-the art techniques in reinforcement learning to this kind of game, while also attempting to keep the agent as general as possible to potentially play different instances of node-based games.

## 2 Related Work

Using RL algorithms to optimally play games has been an area of intense research for decades. In recent years, deep learning, Q-learning, and self-play have come together to produce agents capable of playing increasingly complicated games at superhuman levels. Examples include Deepmind taking on the Atari 2600[1], Go, and Chess[2], among others. OpenAI has found success extending Deepmind's self-play approach into the realm of multiplayer imperfect information games in their work on DOTA 2, which trained an agent to control what is normally 5 different actors[3].

This project also refers to the OpenAI gym, which is a standardized reinforcement learning framework [6].Deep Q-learning is one of the most common methods in Reinforcement Learning, though is usually applied to situations with small, discrete action spaces [7]. There have been attempts at modifying Deep Q-learning to larger action spaces, including the Wolpertinger Algorithm, which relates actions by learning an embedding for them in $n$-dimensional Euclidian space. [8]

## 3 Game Implementation

To leverage our ability to use existing work, we decided to use a N by N grid as an example of a node-based game. This allows the potential use of existing convolutional neural net architectures in training an agent. Given our limited computational resources, we default to a 5x5 grid as the state space is exponential in board size. Players start in opposite corners take turns moving units from an owned square to an adjacent square (not diagonal), with non-neutral squares gaining one unit per turn. We also cap the maximum game length at 200 turns, with the player owning more tiles being declared the timeout winner.

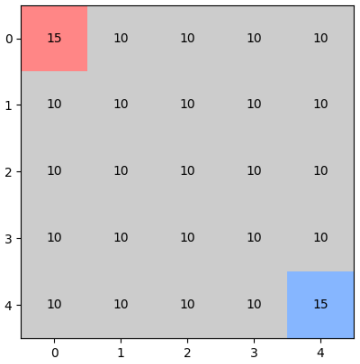



Figure 2: An example of turn 1 - red can move their 15 units either down or right

# 4 Methods

## 4.1 Deep Q-Learning with OpenAI Gym

We initially leveraged OpenAI Gym's reinforcement learning infrastructure to create a modular environment representing different variations of the node-based game problem. We represented the state of the game (the observation space) at each node as a tuple of the player controlling it and their strength at that node.

We represented the action space as a tuple of the originating node and its neighbors, representing movement of as much strength as possible from the node to the chosen neighbor. OpenAI Gym does not provide a straightforward way of allowing only subsets of actions to be available to an agent at a given time, or for an agent to a priori validate whether an action is possible. This is necessary for a sensible implementation of this game because players can only shift their strength to nodes neighboring those they control, so only several of the many potential actions will actually change the state of the game. Therefore, agents had all $4n^2$ possible actions to choose from, not just those that were actually possible. We attempted to pass in a vector representation as input to the network to see if it would learn the valid moves, but this proved infeasible. Any such vector could be passed as a mask on the output vector regardless.

We tried several ways of defining the reward function. The most basic reward function gives a reward when the game is won, and a penalty when the game is lost. However, models trained on this reward function were not able to differentiate between valid and invalid moves, and thus did nothing. To encourage learning of which moves were valid, the reward function can ascribe a heavy penalty to invalid moves, and a reward to valid moves. It is also known that a beneficial strategy is to expand the player's territory, so we experimented with rewards proportional to territory controlled, in addition to rewards gained only upon conquest of new nodes.

### 4.1.1 Model Architecture

We used the keras-rl package to train several DQN models on the OpenAI gym environment, using both a 3x3 and 5x5 grid with the model-agent starting in one corner and the random adversary in the opposite corner. We experimented with allowing the random adversary to validate its moves, but this made it too difficult for the DQN to learn anything to begin with. The DQN performed better against a random adversary which was similarly blind to move validity, yet never was able to learn the valid move space, and never won a match against a random opponent which only took valid actions.

We define $\text{Valid}(a)$ to be 1 if an action is valid and -1 otherwise, $\text{Conquered}(a)$ if an agent's action leads it to control a square it previously didn't control, and $\text{Territory}(a)$ to be the proportion of squares an agent controls. For example, define the following reward functions:

$$R_1(s,a) = \text{Valid}(a) + \text{Territory}(a)$$
$$R_2(s,a) = 10 \cdot \text{Valid}(a) + \text{Territory}(a)$$

Now, we train the following model using these reward functions for 40 episodes using the keras-rl *DQNAgent* and the *LinearAnnealedPolicy*, which gradually trades random moves and exploring the action space for following the learned moves. We trained the model for $4 \times 10^6$ moves, using epochs of $10^5$. The neural network design is analogous to an example we were given for a neural network which learned *Pac-Man*. We found that varying the parameters of the network and convolutional layers did not affect the results.

```
D = 200
model = Sequential()
model.add(Dense(D, input_shape=(args.n, args.n, 3)))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(D))
model.add(Activation('relu'))
model.add(Dense(D))
model.add(Activation('relu'))
model.add(Dense(env.action_space.n))
model.add(Activation('linear'))
```

## 4.2 AlphaZero Self-Play

Following our attempts with reward engineering and standalone deep Q networks, we adapted an implementation[5] of the algorithm seen in the AlphaZero paper[2]. The algorithm relies solely on self play, with the only provided knowledge of the game being the rules in the form of allowed actions and end states. Policy improvement is through a Monte-Carlo Search Tree (MCTS), which leverages the neural net to provide an improved policy. The process is iterative: the neural net plays itself using the MCTS, generating outcomes and policies as examples which are used to train the network after a certain number of games. Each part of the system is described below.

### 4.2.1 Policy and Value Neural Network

The model takes in as input a state of the game, which consists of an 3 channel $n \times n$ array with channels representing which player controls each node, how many units are on the node, and the time elapsed. The model outputs $v_\theta(s) \in [-1, 1]$, which describes the value of the state of the board for the given player, with $-v$ being the value of the board for the opposing player, as well as a policy $p_\theta(s)$ that represents a probability vector over the action space. We then seek to minimize the value of

$$\Sigma_t((v_\theta(s_t) - z_t)^2 - \pi_t \cdot \log(p_\theta(s_t))) \tag{1}$$

where $z_t$ describes the final outcome of the subgame starting at state $s_t$ and $\pi_t$ is a better estimate of the policy from state $s_t$ given by the Monte-Carlo Search Tree (see below). Note the equation does not include regularization terms.

The network trained was a convolutional neural net with four 512-feature convolutional layers and two fully connected feedforward layers. We used the Adam optimizer with a batch size of 64, dropout .2, and batch normalization under PyTorch's implementation.

#### 4.2.2 Monte-Carlo Search Tree with NN Estimation

To improve the policy and simulate games, we used a Monte Carlo Tree Search, which focuses the search for moves into useful ones[3].

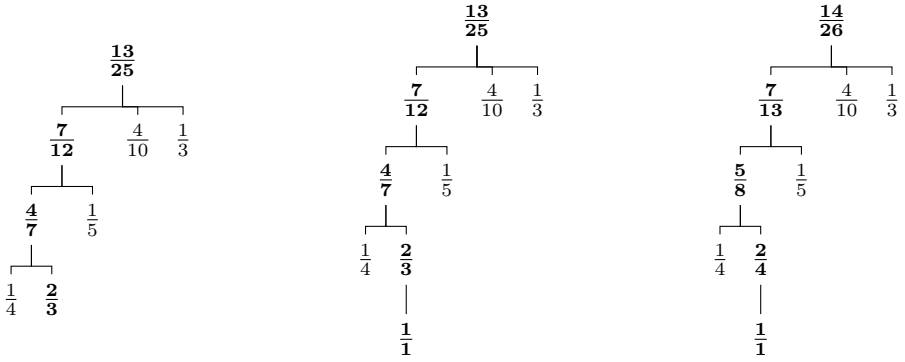


Figure 3: An example of a normal MCTS. First, the tree is traversed by recursively evaluating subgames (following the bolded path) until the leaf is reached. Simulation from this leaf results in a loss, which is backpropagated in an alternating fashion to match the preferences of the two players.

To pick an action in our search, we recursively maximize the Upper Confidence Bound (UCB):

$$Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\Sigma_b N(s,b)}}{1 + N(s,a)}$$

where $s$ is the root of the subtree, $Q(s,a)$ is the expected reward after action $a$ is taken, and $N(s,a)$ is the number of times the simulations took action $a$ from state $s$, with $P(s,a)$ be the neural network's policy's estimate for taking action $a$ from state $s$. $c_{puct}$ is a hyperparameter that balances the exploitation of high $Q$ with the benefit of exploring actions that have been visited few times.

If an action leads to a state not in the tree, no simulation occurs. Instead, the new leaf node is initialized with $v$ and $P(s,\cdot)$ according to the output of the neural network. $Q$ and $N$ for the child are set to 0 for all actions and the value of $v$ is then backpropogated to all parent $Q$ functions. Given, the UCB equation, enough simulations starting from a given state $s$ will yield an improved policy $\pi_s$ equivalent to the normalized counts of $N$ over the available actions.

#### 4.2.3 Self-Play Policy Improvement

We now describe the training process, beginning with a neural network with randomly initialized weights. We play a number of games (100) from the initial state. At every state $s_t$, we perform 50 MCTS simulations to get an improved policy for the state $\pi_t$. We then pick from the distribution $\pi_t$ our action to take from $s_t$, repeating the process. When the game is over, we have examples $(s_t, \pi_t, z_t)$ for all states we took to the termination state.

After collecting examples from all 100 games, we augment the dataset by performing a 180 degree rotation on the board and action vector, as these situations are equivalent to the player making a move. The network is then trained on these examples and pitted against the previous network. If it wins more than a certain threshold (60%), we accept it as a new and better iteration, otherwise we collect another 100 games of data for training.

# 5 Results

## 5.1 Deep Q-Learning

We did not notice a consistent trend in the average reward for each episode, suggesting that the model's occasional success was purely stochastic. In this example, unhelpful moves were heavily penalized several orders of magnitude beyond other game attributes, so if the model had learnt the rules of the game we would expect to see a clear increase in the reward.

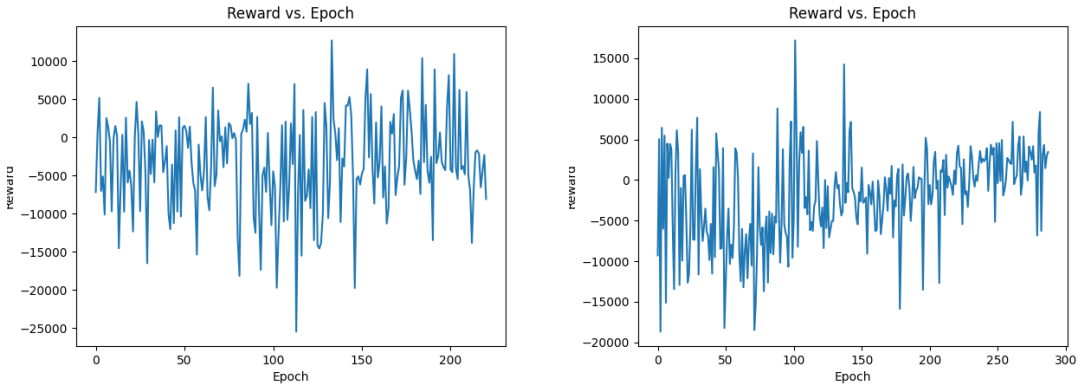



Figure 4: Left, reward vs. epoch for $R_1(s, a)$; Right, reward vs. epoch for $R_2(s, a)$. While the learning for $R_2(s, a)$ is stable, the action space still isn't learned after "emphasizing" the validity of moves. While the reward does increase, the learned agent still loses to random actions.

**Example of Gameplay for** $R_1$: https://gfycat.com/hollowclearcutcoyote
**Example of Gameplay for** $R_2$: https://gfycat.com/lankycharmingibex

## 5.2 AlphaZero Approach

We trained using the AlphaZero inspired approach detailed above. Using 100 games per iteration with 50 simulations per step, we trained for 18 hours on a RTX 3080. Our main baseline was a random player who randomly performs a valid move at every step. Doing so produced 8 iterations of the agent, not including the random 0th iteration.

When testing and pitting the iterations against each other, we kept the process of picking moves from the MCTS as in the training step, opposed to directly picking the highest probability action from the neural net output. This prevents games from being identical while also leveraging the theoretically improved decision similar to how $\pi_t$ is seen as a better approximation of $P$, the neural net's output, during self-play. Games are not identical because when playing a 100-game series, the MCTS is not reset between games so simulations and backpropogation carry over between games. We used 100 simulations per turn for testing compared to 50 when training, which we show likely improves the accuracy of $\pi$.

Along with baselines, we will visually analyze the games played to observe any learned strategies.

### 5.2.1 Baselines and Iteration Comparison

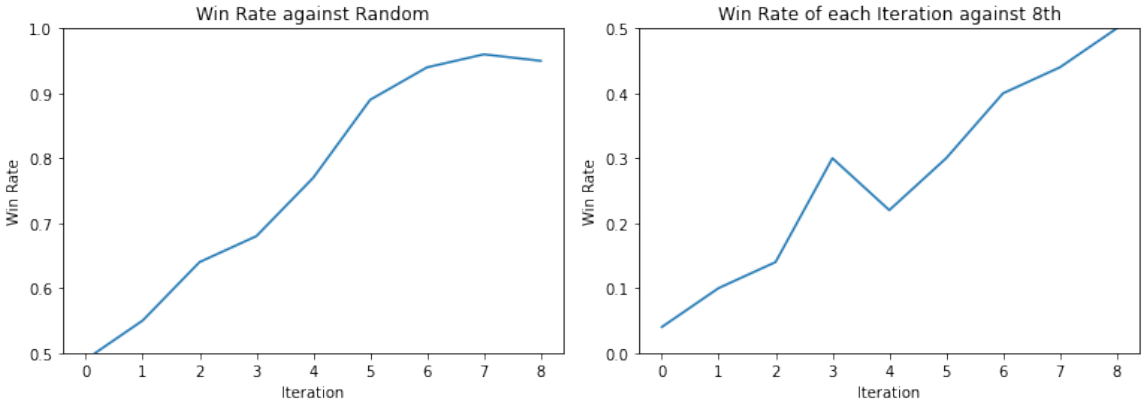
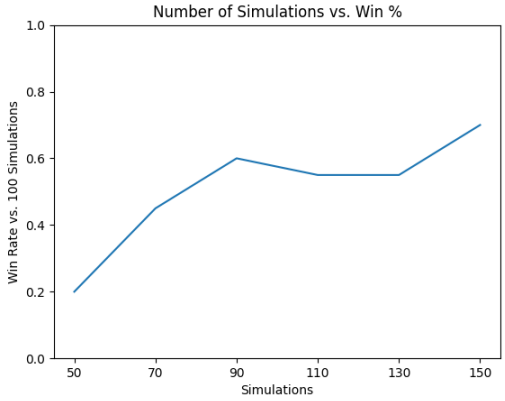



Figure 5: Performance of each iteration against the random baseline, as well the best iteration

We plot the performance of each iteration against random as well as the best iteration and see that our agent quickly dominates the random agent within 6 iterations. Furthermore, we confirm that each iteration will be better than the previous, rather than following any 'rocks-paper-scissors' like cyclical strategy type rotation.

### 5.2.2 Simulation Hyperparameter Analysis




Intuitively, the number of MCTS simulations from a given state should be around the square of the average branch factor. This is because it allows the a typical state to be explored to a depth of 2. For reference, DeepMind in chess used 1600 simulations per turn with 25000 games per iteration. Unfortunately, the linear increase in simulation time proved prohibitive for us to perform over 500 simulations per turn. Despite this, we pitted our best iteration against itself, varying the number of simulations at each turn and plotting the outcome. We see that more simulations do produce a better policy. We also validate our choice to use the MCTS instead of an argmax over the policy vector output by the neural net when testing the iterations.

Given our small simulation count relative to the branch factor, investigation of $c_{puct}$ was impractical.

### 5.2.3 Selected Games

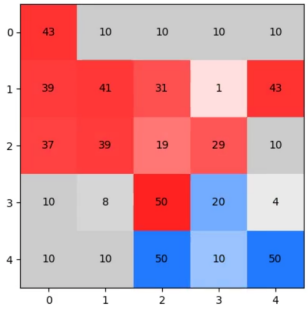



Figure 6: The 8th iteration denying space (top area) to its opponent

**8 vs. 8:** https://gfycat.com/anguishedficklehorsemouse

This example is behavior typical of the 8th iteration, which expands horizontally across the board, then vertically towards the other player's base.

**3 vs. 8:** https://gfycat.com/grippingachingflicker
**4 vs. 8:** https://gfycat.com/carefreewelldocumentedkinglet
**8 vs. random:** https://gfycat.com/offbeatimpoliteirishdraughthorse

From these examples, we see that the model has learned the importance of fighting for space, going as far as to 'wall' areas off for later taking while fighting over more central tiles, as seen in Figure 6.

# 6 Conclusions and Future Work

We conclude that it not immediately feasible to learn grid node-based games with Deep Q-Learning. Despite the positive upward trend in the reward function, granular gameplay still loses to random agents. Further investigation should vary the parameters of the neural network, explore convolutional methods in more detail, and apply more computing power to see if the upward trend continues to the point where the network exhibits an actual strategy.

Furthermore, it is possible to learn how to play a grid node-based game with the AlphaZero approach, and that it is difficult to train a RL agent without disallowing invalid actions with an output mask. The agent does not require any human knowledge beyond the rules of the game and convincingly beats both previous iterations as well as a random baseline. We also confirm that the AlphaZero approach can be extended to smaller problems with smaller networks when using consumer hardware.

Given more time, we'd try to approach more general node-based games, with tile specific growth rates and different levels of connectivity (i.e. hex grids, non-adjacent movement), exploring how to adapt the CNN architecture critical to our current approach

The code used for this project can be found here, along with the trained models:

OpenAI Gym/DQN Code: https://drive.google.com/file/d/165CdMaNAjV4n_ZbvtPEG_IHpw3k41D66

AlphaZero Code: https://drive.google.com/file/d/17Ubnh72d1e4LkoDiR2d9jWcIXbpRDPQA

# References


[1] Mnih, Vlad and Kavukcuoglu, Koray, *Human Level Control Through Deep Reinforcement Learning, Nature, 2015.*. Publishing Company, 1994. https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning

[2] Silver, David, Hubert, Thomas, et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.* Arxiv, 3017.

[3] Brown, Catherine, et al. *A Survey of Monte Carlo Tree Search Methods* IEEE Transactions on Computational Intelligence and AI in Games, 2012, https://ieeexplore.ieee.org/document/6145622

[4] Berner, Christopher, Brockman, Greg, et al. *Dota 2 with Large Scale Deep Reinforcement Learning.* OpenAI, 2019.

[5] Nair, Surag, Thakoor, Shantanu, et al, *Alpha Zero General* https://github.com/suragnair/alpha-zero-general

[6] https://gym.openai.com/

[7] https://keras.io/examples/rl/deep_q_network_breakout/

[8] Dulac-Arnold, Gabriel, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. "Deep Reinforcement Learning in Large Discrete Action Spaces." ArXiv:1512.07679 [Cs, Stat], April 4, 2016. http://arxiv.org/abs/1513.07679.