REINFORCEMENT LEARNING IN TOADS AND FROGS

LOGAN GARWOOD

1. Introduction

Artificial Intelligence is a field of Computer Science in which an agent can "learn" to simulate human like intelligence through the use of data. AI is a broad term for intelligent computer agents that are trained to learn a task although in practice this is often done with the use of machine learning. Reinforcement learning is a further subset of machine learning in which an agent learns to perform a task by repeated exploration of an environment and subsequent rewards and punishments according to how well the agent does its job. Reinforcement learning has been used to train agents to play many combinatorial games such as Tic-Tac-Toe, Checkers, Connect Four, Chess, Go and many others. The goal of this project is to explore how reinforcement learning can be used to play and understand combinatorial games using the game Toads and Frogs. In this paper I will discuss the prior analysis of Toads and Frogs as a combinatorial game, then introduce the reinforcement learning framework with which I trained the agents. The goal of this project is to demonstrate that an agent can learn to play Toads and Frogs and discuss the pros and cons of solving games using reinforcement learning.

2. Toads and Frogs

Toads and Frogs is a simple combinatorial game invented by Richard Guy [5]. I have chosen to use this game for reinforcement learning due to its simplicity and novelty. As far as I can tell no one has tried to train an AI to play Toads and Frogs before; this is probably because the game was invented to study the theory of combinatorial game theory and is not popular to play in real life.

Rules. Toads and Frogs is played on a $1 \times n$ rectangular board. Each square contains either a toad, a frog, or nothing. Left can move the toads to the right and Right can move the frogs to the left. Each amphibian can either "slide" one square forward if that square is empty or, if the square is not empty but contains an amphibian of the opposite side, they can "hop" over the opposing amphibian and end up two squares advanced. The game follows normal play convention. Typically a board initially consists of some consecutive toads followed by blank spaces and then some consecutive frogs although this is not necessary. We will use the typical notation for a game by representing toads with a T, frogs with a F, and blank squares with a $\square$. For example a starting position may be $T\square\square F$. Furthermore we will denote repeated sequences of squares with exponents so for example $(T\square)^2F^3 = T\square T\square FFF$.

Analysis. Now we quickly discuss the analysis that has already been done [2], [3] on the game Toads and Frogs. Often in combinatorial game theory we want to simplify a position to a simpler position so that we can more easily determine the outcome and best move. In Toads and Frogs, the key to simplifying positions is to identify *dead pieces*. For example the position $T\square\square TTFTFF\square F$ looks complicated at first but because the middle contiguous sequence of amphibians begins with two toads and ends with two frogs, none of these pieces will have any moves for the rest of the game. We call these pieces dead. So the position simplifies to $T\square\square + \square F = 2 - 1 = 1$. Any section of the board that is not a contiguous sequence of toads or frogs beginning with TT and ending with FF will be called *alive*.

Now let's look at one of the most basic results from [1] which has been generalized by [2]. We call a square isolated if both of its neighbors contain amphibians.

Proposition 2.1 ([2] The Death Leap Principle)**.** *Any position in which the only legal moves are jumps into isolated spaces has value 0.*

Proof. Suppose it is Left's turn to play. If she has no moves then Right wins. Otherwise she must jump a toad into an isolated space. Because this move jumped over a frog, Right now has a move which is to slide that frog forward one space. Now Left must move again but again every legal move is a jump into an isolated space so by induction this is a $\mathcal{P}$ position meaning that Right will win the game. Similarly if Right plays first. $\square$

Corollary 2.2 ([2])**.** *Any position that does not contain any of the four subpositions $TF\square\square$, $\square\square TF$, $T\square$, or $\square F$ has value zero.*

Erickson proves in their paper [2] that for any dyadic rational, there is a Toads and Frogs position with that number as its value. They also showed there are Toads and Frogs games with arbitrarily high temperatures. Lets use the notion of dead pieces and the Death Leap Principle to analyze some board positions and prove the first of these claims.

Theorem 2.3 ([2])**.** $(TF)^mT\square(TF)^n = \frac{1}{2^n}$ *for all m and n.*

Proof. When $n = 0$, we have $(TF)^mT\square = T\square$ which clearly equals 1. Now we induct on n. Suppose $n > 0$. Left's only move is to $(TF)^m\square T(TF)^n = 0$ by the Death Leap Principle. Right's only move is to $(TF)^mTFT\square(TF)^{n-1} = (TF)^{m+1}T\square(TF)^{n-1} = \frac{1}{2^{n-1}}$ by induction. So $(TF)^mT\square(TF)^n = \left\{0 \mid \frac{1}{2^{n-1}}\right\} = \frac{1}{2^n}$ by the number simplicity theorem. $\square$

Corollary 2.4 ([2])**.** *For any dyadic rational number $\frac{k}{2^n}$ there is a Toads and Frogs position with that value.*

Proof. Using Theorem 2.3 and the Death Leap Principle, we see that the game $(T\square(TF)^nTTFF)^k$ has value $\frac{k}{2^n}$. $\square$

3. Reinforcement Learning Basics

We now give a brief introduction to the theory of reinforcement learning.

3.1. **Markov Decision Process.**

We will represent the game Toads and Frogs as a Markov Decision Process as is typical in reinforcement learning. A typical Markov Decision Process (MDP) consists of four things: a set of environment states $\mathcal{S}$, a set of agent actions $\mathcal{A}$, a probability transition function $\mathcal{T}$, and a reward function $\mathcal{R}$. We will not consider the probability transition function as the outcome of an action in a combinatorial game is deterministic.

In our Toads and Frogs MDP, the states are any board position reachable from the initial board position. The actions available to an agent are to choose which amphibian of theirs to move (note that each amphibian can either slide or hop but not both) for example in the position $TT\square TF\square F\square F$, Left can choose to play the action 1 or 2 representing moving their first or second toad while Right has 2 and 3 as possible actions.

The reward function is a way of giving points to the agent based on the result of its actions. We will discuss the details of this later but essentially a good reward function will give positive points for moves that lead to winning positions and give negative points for moves that lead to losing positions.

3.2. **Training.**

The goal of the agent is to learn a function $\pi : S \to A$ called a **policy** that returns the best action to take at any state. To do this we will use **Q-learning** which is a popular algorithm for finding the optimal policy of a Markov Decision Process. This requires first finding a function $\mathcal{Q} : S \times A \to \mathbb{R}$ where $\mathcal{Q}(s, a)$ approximates the reward of taking action a from state s. Once a quality $\mathcal{Q}$ function is achieved, extracting a policy is trivial: at every state s, play the action a such that $\mathcal{Q}(s, a)$ is maximized. In most practical applications of reinforcement learning, this Q function is hard to find exactly due to memory constraints so we typically use function approximations. For this project we will choose to use a Deep Q-Neural Network (DQN) that takes in a board state as input and returns an estimated Q value for each action from that state.

The main training loop consists of the reinforcement learning agent playing many games, sometimes called episodes, against another opponent. At each time step t in an episode, the agent will make an action a_t in state s_t to receive reward r_{t+1} and move to state s_{t+1}. We use the Bellman equation [4] to optimize the Q function Neural Network which is a common technique in reinforcement learning that updates

$$\mathcal{Q}^{new}(s_t, a_t) := (1 - \alpha) \cdot \mathcal{Q}(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) \right)$$

where α is the learning rate and γ is a discount factor that controls how much the reward from future states should trickle down to prior states. Essentially what this update algorithm says is that we will set the new Q-value to be some linear combination of the old Q-value and the sum of the amount of reward we received with the discounted expected value for the state we end up at. This algorithm is simple to implement if we model our Q function with a hash table but requires some adaptation when using DQN.

In deep Q-neural networks, the goal is the same: to learn a good Q function that can be used to extract the optimal policy. To do this, the agent plays many games and at each move, saves the state, action, reward, and resulting state in its memory. This data is then used to calculate the expected Q-values from the Bellman update algorithm. Once this dataset is large enough, the agent will optimize its neural network using the difference between the current Q-values and the expected Q-values as the loss to backpropagate through the network. However, in order to maintain a differentiable computational graph through which to backpropagate, the current Q-values cannot be calculated using the same network as the expected Q-values. This is why I implemented a common technique in which two neural networks with the same architecture are used, one called the main network and the other called the target network. The main network calculates the current Q-values and the target network is used to calculate the expected Q-values. Every twenty games, the target network is updated to inherit the same parameters as the main network. In this way the main network will hopefully converge to an accurate Q function that looks at a state and gives a score to each action based on how likely it is to lead to a win.

3.3. **Exploration vs. Exploitation.** While training, the agent is learning about its environment and optimizing its neural network, but how should the agent choose which move to play at any given time? Always choosing its expected best move may not be optimal, especially early on when we know that the agent's view of the environment is still lacking. This is a concept known as *exploration vs. exploitation.* Essentially, our agent starts out knowing nothing about its environment and every time it makes a move, it gets feedback about how to make choices in the future. So in the beginning, we want the agent to explore moves that may be suboptimal in order to gain experience, but after the model has been trained for a while, it can start trying to chase big rewards by exploiting its known strategy. Imagine you are playing a game in which there are three slot machines with some unknown distribution of payout and at every time step, you choose one of the machines to play. Your objective is to maximize the payout. Because the distribution of each machine is originally unknown, you will want to try each one a couple of times until you start to make some guesses about which slot machine is better to play. After several time steps you will be more confident about which one is best and then you can exploit that machine. This is the dilemma any reinforcement learning agent faces.

To implement this exploration then exploitation strategy, I used the ε-greedy algorithm. According to this algorithm, the agent chooses a random legal move with probability ε and chooses what it believes to be the current best move with probability $1 - \varepsilon$. We set ε to be some positive number less than 1 and then decay it throughout training so that the agent tends to exploit later on.

4. Code Implementation

I programmed the game Toads and Frogs and coded the reinforcement learning algorithms from scratch in Python (https://github.com/garwoodl/combinatorial-game-theory-final-project.git). To do this I made classes for states and agents that interact with each other to successfully play and display the game. I also implemented some trivial bots in their own Agent subclass such as a random bot and an 'end' bot that always plays their amphibian that is furthest up the board (or furthest back). There are also several miscellaneous functions for simulating many games between agents, displaying plots, and getting human interaction.

The bulk of the energy spent on this project was in programming the reinforcement learning algorithm. As mentioned above, I used a deep Q-learning neural network to represent the agent's Q function. After experimenting with several architectures, I settled on a fully connected perceptron with three hidden layers of size 100, 150, and 100 separated by ReLU activation functions. This model takes in a state s represented as a vector of 0s, 1s, and -1s for blank spaces, toads, and frogs respectively. It then outputs a vector with length corresponding to the number of possible actions where the a^{th} entry of the vector is the estimated Q-value $\mathcal{Q}(s, a)$.

I ran several training experiments using different initial board configurations to find a good set of training hyperparamters such as the learning rate and reward function. The model was optimized using the Adam algorithm which is typical for most deep learning neural networks. The main training loop looks something like this:

```python
for episode in range(num_episodes):
        episode_done = False
        # decay epsilon from start to end using inverse sqrt
        epsilon = end_epsilon + (start_epsilon - end_epsilon) / (episode + 1) ** 0.5
        while not episode_done:
            # make a move and get feedback
            action = self.choose_move(state, epsilon)
            next_state, reward, done = self.step(state, action)

            # the opponent responds
            if not done:
                opp_action = opponent.choose_move(next_state)
                # this reward is still with respect to the agent being trained
                next_state, opp_reward, done = self.step(next_state, opp_action)
                reward += opp_reward  # counteract reward by the success of the opponent

            self.buffer.push(state, action, reward, next_state, done)
            state = next_state
            episode_done = done

            # make sure there are enough move samples to optimize
            if len(self.buffer) >= self.batch_size:
                loss = self.optimize_model()  # performs a step of backpropagation in DQN
                losses.append(loss)

        if episode % self.target_update_freq == 0:
            self.update_target_network()
```

where the most important functions that needed to be implemented were `choose_move()`, `step()`, and `optimize_model()`. I tried to give a reward to the agent when it entered a $\mathcal{P}$ position according to the death leap principle but this did not make any significant improvement to training, likely because the majority of positions arrived at from the starting board $T^a\square^b F^a$ contain one of the sub-positions given in Corollary 2.2 when b is large. Therefore, the reward function I chose for most of the project gave 30 points for a win, -10 points for a loss, and -15 points for an illegal move.

5. Results

All training and experiments were done on my old laptop so these results are mostly proof of concept and it should be understood that with the same methods but more computing resources and time, the quality of the agents could have been improved. We will mostly analyze games with starting position $T^a\square^b F^a$ which [3] conjectured was equal to 0 or $*$ for any $b \geq a > 0$. To evaluate an agent's play, we will look at simulations of 1,000 games played against a random opponent and also explore the Q-values given for particular sequences of moves in these games. All of the training and simulation was done against random bots although I tried other training strategies as well. I tried many times to no avail to train a toad agent and frog agent separately against random bots and then further train them against each other. However, this produced unsatisfactory results such as one agent becoming much better than the other while the skill gap between them continued to grow throughout training.

5.1. **Small Board.**

First I began by showing that all the code worked as desired and the agent's neural network was learning to produce Q-values that are "good".

I trained a reinforcement learning agent to play as toads against a random bot on the board $TTT\square\square\square FFF$ (which [3] showed is equal to the 0 game). Note that the number of reachable positions on this board is upper bounded by $\frac{9!}{3!3!3!} = 1,680$ so this game should be rather simple to solve even without the use of reinforcement learning.

After training the model for 6,000 episodes or approximately 9 minutes, it produced the loss plot [Figure 5.1] showing the decrease in loss throughout training.

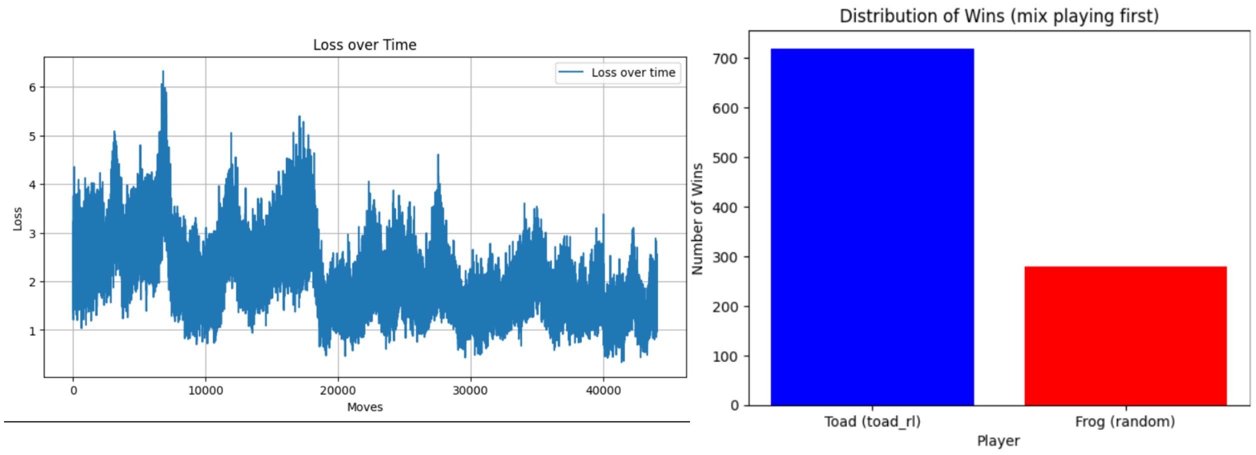

FIGURE 1. (Left) Training Loss and (Right) Simulation results for $T^3\square^3F^3$

After training this model, I had it play 1,000 games against a random opponent, swapping who goes first each game, in which it won 763 of them [Figure 5.1]. This is a significant winning margin and shows that the agent was successfully able to learn the game Toads and Frogs.

5.2. **Medium Board.** After seeing that the agent is indeed learning the way it should, I increased the board size and training time by looking at $T^4\square^6F^4$ (which [3] showed is equal to the 0 game).

After training for 20,000 episodes the results are shown in [Figure 5.2].

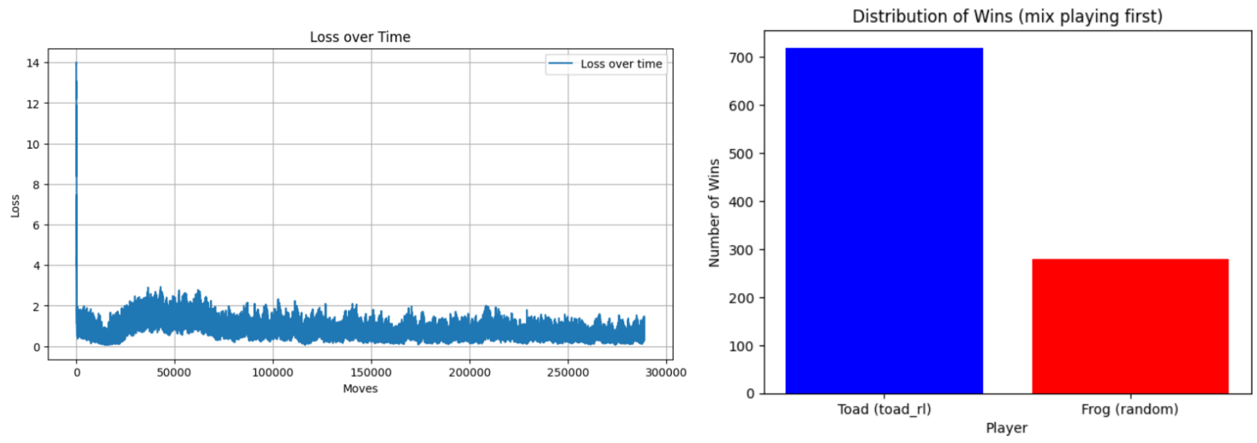

FIGURE 2. (Left) Training Loss and (Right) Simulation results for $T^4\square^6F^4$

This showed a win rate of 720 games and a decreasing training loss. I also analyzed several games move by move to see what the model was thinking while it was playing and learning.

Here is one such end of a game:

random	$\rightarrow T\square\square T\square FT\square TFFF\square\square$	$[-4.3, 22, 10.3, -6.4]$
rl	$\rightarrow T\square\square T\square F\square TTFFF\square\square$	
random	$\rightarrow T\square\square TF\square\square TTFFF\square\square$	$[-8.6, -11.4, 20.6, 24.3]$
rl	$\rightarrow \square T\square TF\square\square TTFFF\square\square$	
random	$\rightarrow \square TFT\square\square\square TTFFF\square\square$	$[-11, -14, 27.1, 22.6]$
rl	$\rightarrow \square TF\square T\square\square TTFFF\square\square$	
random	$\rightarrow FT\square\square T\square\square TTFFF\square\square$	$[-12.6, -14.7, 30.1, 29.8]$
rl	$\rightarrow FT\square\square\square T\square TTFFF\square\square$	rl wins!

I selected this game snippet because it demonstrates what the reinforcement learning agent is thinking at each step and how we can interpret the Q-values. The vectors on the right hand side show the models outputted Q-values for each action available from the state that the random bot played into. The relative magnitude of each entry in a vector shows how good the model predicts that move to be for the agent. In this game, ε was set to 0 so the agent will always play the action with the highest Q-value. Note that in the first move shown, the agent takes advantage of the concept of dead pieces in Toads and Frogs by playing action 2 which creates a $TTFF$ sequence, thus killing that section of the board. This is advantageous for Left because it reduces the position to the game $T\square\square T\square F\square$ which is clearly a positive game. Also, remember that the model is technically allowed to select illegal moves but this will result in an instant loss. The Q-values from game play show that it has a good understanding of when a move is illegal by assigning it a negative number.

5.3. **Large Board.**

Toads and Frogs is a relatively simple game to train a reinforcement learning agent on because, for small boards it is not hard to expand the game tree and find the winning move from any position definitively. The advantage of using reinforcement learning to play combinatorial games lies in their ability to quickly extract the best move from a position based off of learned patterns in the board without resorting to memory. For this reason, we will now look at the agent's ability to make sense of large Toads and Frogs boards.

I trained a reinforcement learning agent on the starting position $T^6\square^9F^6$ because this is one of the smallest games for which [3] was not able to calculate the canonical form (although it is believed to be either 0 or $*$). Note that the upper bound for this game's reachable positions skyrockets to $\frac{21!}{6!9!6!}$ which is over 270 million, so expanding a game tree for this position would be extremely memory intensive. However, my reinforcement learning agent is able to pick an informed move almost instantaneously. This is because the time it takes to select a move is determined by feeding a state vector through the agent's fully connected neural network, which is akin to matrix multiplication, a computationally efficient operation. However, the question remains: how informed is this move?

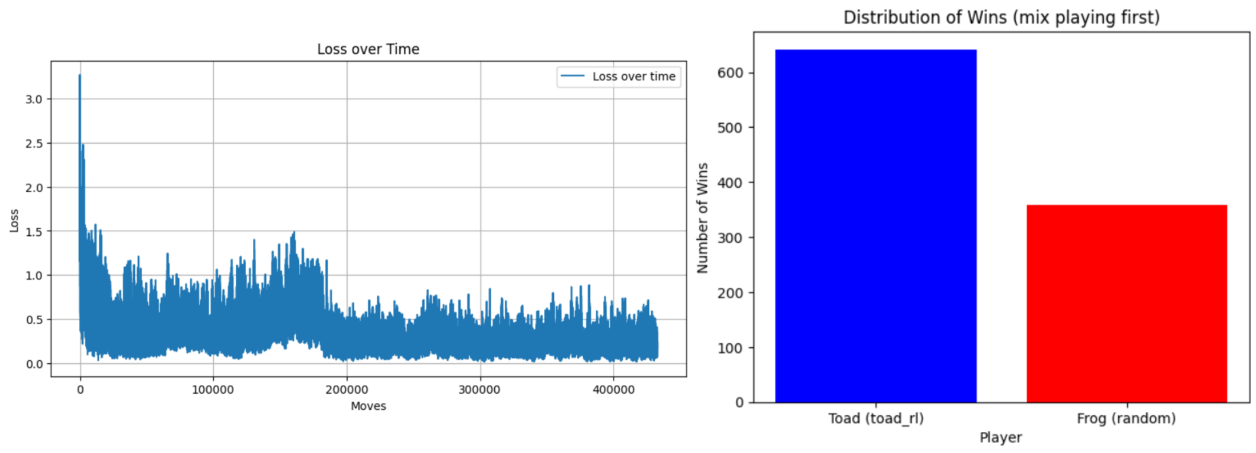

FIGURE 3. (Left) Training Loss and (Right) Simulation results for $T^6\square^9F^6$

After training for 20,000 episodes or approximately 2 hours, we can see that even on the large board the network was able to decrease its loss and the agent was able to outperform the random bot by winning 642 games [Figure 5.3]. As the loss plots show, this training resulted in the agent seeing over 400,000 not necessarily distinct board positions which, even if they were distinct, is on the order of 0.1% of the 270 million possible board positions. This highlights the immense power of AI to scale in size and understand the general properties of a board that help determine the best course of action.

A game snippet from this agent shows similar strategy as on the medium board:

$$\text{random} \rightarrow TTTT\square\square\square\square\square T\square F\square TFFF\square FF\square \qquad [-13, -2.5, 0.1, -9.7, -15.3, -17.6]$$

$$\text{rl} \rightarrow TTT\square T\square\square\square\square T\square F\square TFFF\square FF\square$$

$$\text{random} \rightarrow TTT\square T\square\square\square\square T\square F\square TFFFF\square F\square \qquad [-9, 0.9, 4.1, -2, -11.4, -14.1]$$

$$\text{rl} \rightarrow TTT\square\square T\square\square\square T\square F\square TFFFF\square F\square$$

$$\text{random} \rightarrow TTT\square\square T\square\square\square TF\square\square TFFFF\square F\square \qquad [-13, 8.1, -0.8, -2.5, -12.4, -19]$$

$$\text{rl} \rightarrow TTT\square\square T\square\square\square\square FT\square TFFFF\square F\square$$

$$\text{random} \rightarrow TTT\square\square T\square\square\square\square FT\square TFFFFF\square\square \qquad [-14.8, 19.4, -1.3, -1.3, -12.4, -18.8]$$

$$\text{rl} \rightarrow TTT\square\square T\square\square\square\square F\square TTFFFFF\square\square \qquad \text{...rl wins eventually!}$$

Just like in the medium sized board, the reinforcement learning agent recognizes that it has many moves in reserve so if it can make a blockade in the middle of the board, then it will easily win the game. The first couple moves set the scene until the random bot decides to play its forward most frog. This gives the reinforcement learning agent a tempo with which to leap its toad closer to a blockade position. Then when the random bot did not address the threat, the agent completed the blockade and killed almost all the frogs resulting in an easy win. This may not be that impressive because the random bot is not a challenge. However, even I lost to this trained agent a few times before I figured out how to exploit it. Although Toads and Frogs is a simple game, when it comes to large boards, every move feels somewhat random anyways until you get close to a situation with dead pieces like shown above.

Also note that in the larger board, the Q-values are all less than we saw in the medium sized board. This is likely because each game can last longer on the larger board and, since the only way to achieve a positive reward is to reach the end of a game and get a win, it takes longer for the rewards from a win to propagate to moves earlier in the game. Similarly, the longer the game goes on, the more chances there are for an agent to make an illegal move and get a big negative reward. Therefore, it would require much longer training before we see Q-values that are as intuitive as we saw in the medium sized board (where illegal and losing moves are negative while winning moves are positive).

5.4. **Conclusion.** In this project, I explored the application of reinforcement learning to combinatorial games by analyzing Toads and Frogs. By training a deep Q-learning neural network, I demonstrated that an AI agent can learn to play the game effectively. The results showed that the agent was able to significantly outperform a random player but also to understand certain game mechanics such as illegal moves and dead pieces. Although the win percentages against random opponents left room for improvement across the board, I attribute a large proportion of this effect to the resources at my disposal. I would be interested in further training and experimentation on a virtual machine so that I can let it train for several hours uninterrupted and with much higher computational power. Overall, I am pleased with my algorithm's ability to learn strategy, make sense of a position, and improve its understanding through training.

References

[1] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning ways for your mathematical plays. Vol. 1.* A K Peters, Ltd., Natick, MA, second edition, 2001.

[2] Jeff Erickson. New toads and frogs results. In *Games of No Chance, Proc. MSRI Workshop on Combinatorial Games*, pages 299–310, 1994.

[3] Thotsaporn"Aek" Thanatipanonda. Further hopping with toads and frogs. *arXiv preprint arXiv:0804.0640*, 2008.

[4] Wikipedia contributors. Bellman equation — Wikipedia, the free encyclopedia, 2023. [Online; accessed 3-June-2024].

[5] Wikipedia contributors. Toads and frogs — Wikipedia, the free encyclopedia, 2023. [Online; accessed 15-May-2024].