Checkers Reinforcement Learning Agent Using Deep Q-Learning

Arnav Chittibabu, Leigh Anne LeMoine, Wenhan Lu

March 16, 2024

Abstract

The following project will explore both the process and challenges of building a checkers game-playing agent using reinforcement learning. This project implements a checkers-playing agent utilizing deep Q-learning, a fundamental reinforcement learning algorithm. Through a process of self-play, the agent iteratively refines its decision-making strategy. It learns to evaluate the game state and prioritize actions that maximize its expected future reward. This reward system is designed to incentivize capturing all of the opponent's pieces and win the game. Performance analysis demonstrates the agent's increasing competency as it accumulates game experience. Over time, the agent learns to exploit weaknesses in its own earlier strategies and prioritize favorable actions within the constraints of the game's rules.

1 Project Objectives

1.1 Why Reinforcement Learning?

Our group chose to create a reinforcement learning agent for our final project because we all had prior experience with clustering, CNNs, and RNNs. Reinforcement learning is one of the most studied and applied subfields of artificial intelligence, and we wanted to challenge ourselves with a topic none of us had ever worked with before. Perhaps the most appealing part of working with reinforcement learning is the fact that it doesn't require obscene amounts of data to train and test on like most other deep learning techniques. Often times, the sheer gigabytes of data serve as the bottleneck for most machine learning endeavors, as collecting, cleaning, storing, and (most of all) labeling are extremely difficult tasks. Reinforcement learning agents break this mold by learning through trial and error, fascinatingly mirroring the process in which humans learn skills—receiving information from the environment, taking an action according to prior belief, and observing the consequences to update said beliefs. By training an agent, it can gain the ability to navigate complicated environments, learn from its interactions, and determine the optimal course of action to achieve a certain outcome. Reinforcement learning already finds many uses in robotics (automating assembly lines, service and personal care robots to improve quality of life), transportation (self-driving cars would forever change infrastructures), finance (trading, optimizing portfolios, risk management), healthcare (personalized treatment among other things), and strategy games (to push the limits of modern technologies and our understanding of them). The ability to adapt to dynamic environments means constant improvement and evolution, which gives serious potential to revolutionize industries and shape the future of the world.

1.2 Why Checkers?

When deciding on a game for our agent to train on, we faced an interesting dilemma on choosing the appropriate game style and complexity level. We initially considered games such as Snake and Pong, but quickly realized time should not be a gameplay factor, as it would be too difficult for a reinforcement learning agent to play. From there, we decided to look at turn-based games since they featured discrete, deterministic action spaces. We looked into card games such as Go Fish and Blackjack, but we wanted to have a playing environment for ease of representing game states visually. From there, we considered a multitude of turn-based strategy board games. Tic-Tac-Toe was deemed too simple, so in the end, we arbitrarily decided on checkers over Connect 4. We thought checkers would pose an interesting challenge to a reinforcement learning agent because it requires players to plan ahead and make careful decisions. In addition, the ruleset is not too complex to implement, but

it still features an incredibly large number of game states (around 5×10^{20} possible configurations!), making it a formidable challenge to learn the policy for. By observing how the agent improves said policy as training goes on, we can observe the principles of reinforcement learning applied to a fun and ubiquitous real-world game.

1.3 Other Approaches besides Deep Q-Learning

Other methods of solving this problem of finding an optimal policy for checkers include the minimax algorithm, alpha-beta pruning, and Monte Carlo tree search. Although it has never been trained on checkers, the most successful approach to this type of problem is probably Google DeepMind's AlphaZero for chess, go, and shogi, which are games that are much, much more complex than checkers, so there are no doubts that AlphaZero could find an optimal policy. Perhaps most interestingly, checkers was actually weakly solved in 2007, meaning an algorithm exists to guarantee perfect play from the beginning to the end of a match (although notably not from any state). This means that if both sides were to play perfectly, a draw is inevitable. With 5×10^{20} possible game states, this took around 10^{14} total calculations spread over 18 years.

1.4 What We Hope to Accomplish

Obviously, we are not expecting to find the optimal policy in the given timespan using just deep Q-Learning, nor do we have 18 years at our disposal. The end goal is to have an agent that poses a fair challenge to the average checkers player, which is far more realistic. Perhaps more importantly, this project is an opportunity to apply theoretical knowledge to a real-world game. By spending implementing step-by-step algorithms from scratch, tuning hyperparameters, and fixing the seemingly endless bugs, we gain a much deeper understanding of what's going on underneath the hood of reinforcement learning algorithms. In addition, designing a board representation and accounting for unexpected states and edge cases, we learn much about how to model environments for future reinforcement learning endeavors.

2 Methodologies

2.1 Q-Learning Algorithm

The core of our reinforcement learning algorithm is Q-learning, which allows the agent to learn an optimal policy that finds the best action given any state. The process begins by initializing an arbitrary action-value function $Q(s, a)$ for all possible state-action pairs. Then for each time step t, the agent takes 1 of 2 moves depending on the exploration rate. The agent either takes a random move at whatever percent of the time the exploration rate is set to. The rest of the time the games observes state s_t, takes action a_t using an ϵ-greedy policy derived from Q, receives reward r_t, and transitions to state s_{t+1}. By iteratively updating the action value function values until a terminal state is reached for each episode, we can return the expected reward for each state-action pair in a Q-table. We can visualize the algorithm as such:

Initialize Q-table: $Q(s, a) = 0 \; \forall \; s \in S, a \in A$
For each episode:
  Initialize starting state, s
  Repeat (for each step of episode):
    Choose action, a, from state, s, using an exploration strategy
    Take action, a, observe reward, r, and next state, s'
    Update Q-value:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

    $s \leftarrow s'$
  Until terminal state is reached

Note that gamma takes a value between 0 and 1 and controls how much importance the agent places on future rewards.

While we initially attempted to use Q-learning to train our agent, we quickly realized that the checkers environment has way too many possible state-action pairs for Q-learning to handle. To mitigate this, we decided to switch to a deep Q-learning approach, which uses a neural network to approximate the Q-values and their relative importance to other values.

2.2 Deep Q-Learning

Deep Q-Learning is another reinforcement learning algorithm that performs better in more complex and higher-dimensional state representations. Rather than utilizing a Q-table and iterative updates, Deep Q-learning uses a deep neural network to learn that action value function give the state representation as an input.

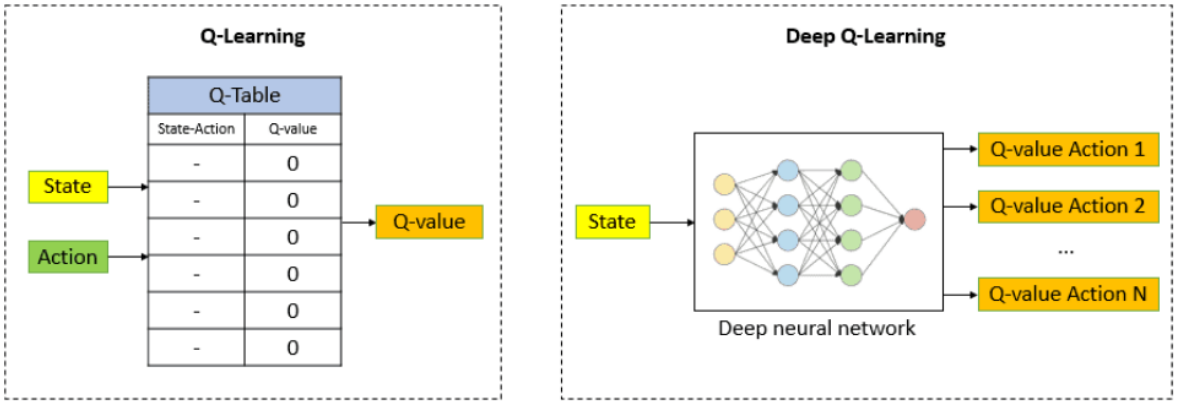

Figure 1: The difference between Q-learning and deep Q-learning

The algorithm is structured as such:

Initialize replay memory $\mathcal{D}$ **with capacity** N
Initialize action-value function Q **with random weights** θ
Initialize target action-value function $\hat{Q}$ **with weights** $\theta^- = \theta$
For each episode:
  Initialize starting state s
  Repeat (for each step of episode):
    Choose action a using ϵ-greedy based on $Q(s, a; \theta)$
    Take action a, observe reward r, and next state s'
    Store transition (s, a, r, s') in replay memory $\mathcal{D}$
    Sample random minibatch of transitions (s_j, a_j, r_j, s'_j) from $\mathcal{D}$
    Calculate target values:

$$y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ is terminal} \\ r_j + \gamma \max_{a'} \hat{Q}(s'_j, a'; \theta^-) & \text{otherwise} \end{cases}$$

    Perform gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. θ
    Every C steps, reset $\hat{Q} = Q$
  Until terminal state is reached

2.2.1 Implementation

While in theory the Q-value updates are as above, in our Deep Q-Learning model we utilize the Keras Sequential API to build each layer of the neural network that learns the Q-values. The layers are structured as follows:

- The first layer is a dense layer with 32 neurons. Relu is the activation function that introduces non-linearity allowing the DNN to learn the complex patterns within this same. It takes in an input of 64 dimensions (8x8 board) which is the state representation.
- The second layer is also a dense layer with 16 neurons. It uses the relu activation function as well. In this layer we apply L2 regularization with a coefficient of 0.1. This penalizes large weights in the model which we hope would prevent overfitting.
- The output layer has a single neuron and uses a relu activation function. L2 regularization is applied again. This output represented the estimated Q-value for a given action.
- Lastly the compilation step utilizes an optimization algorithm called 'nadam' responsible for updating weights. Nadam (Nexterov-accelerated Adaptive Moment Estimation) is a variant of adam. We then define the loss function to binary cross-entropy. This is a common loss function for problems with two classes. While Q-values themselves are continuous, many DQN implementations frame the loss calculation as a binary classification problem: "Should the Q-value be updated higher or lower?"

2.3 Game State Representation

In order to apply deep Q-learning, the checkerboard state must be represented in a format that can be understood by the algorithm. It is important that this representation captures all details of the state while also being computationally efficient due to the number of iterations that will be run. To this end, we chose to use a hash map as the data structure of choice to represent each unique state with a key, which corresponds to the Q-values approximations of every possible action in the state. This allows for quick accessing and updating, which is essential for the sheer number of possible action-state pairs in checkers encountered during training.

2.4 Action Space

Since checkers is an ancient game (over 5000 years old!), there are a plethora of variations when it comes to the rules. The general gist is similar though: a two-player strategy game involving a checkered board (usually 8x8, but can vary from 10x10 to even 12x12) and pieces arranged on alternating squares. The goal of the game is to capture all of the opponent's pieces by jumping over them. The variations mostly arise with the rules regarding moving and capturing exist: Pieces can only move forward one square at a time in a diagonal fashion. One can only capture an adjacent piece if the square behind that piece is empty. If a piece reaches the opponent's back rank, it is crowned as a king and is allowed to move both forward and backward for the rest of the match. These rules are all widely accepted, but there are a few ambiguous ones as well. One rule states that if, after capturing, there is another capture available, that move can be taken immediately in the same turn (leading to double jumps, triple jumps, and so on). Another rule stipulates that if there is an available piece to capture, then the player must make that capture. In order to simplify things as much as possible for our agent, we decided to not allow multiple jumps and not to require forced captures. This is very important to remain consistent on, as the rules of the game determine the set of allowed actions the agent can take. Examples are given below as it may be difficult to visualize these rules from a written description.

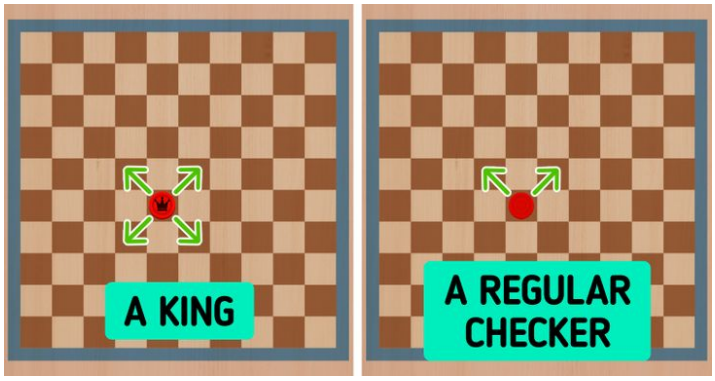

Figure 2: Standard rules for moving pieces

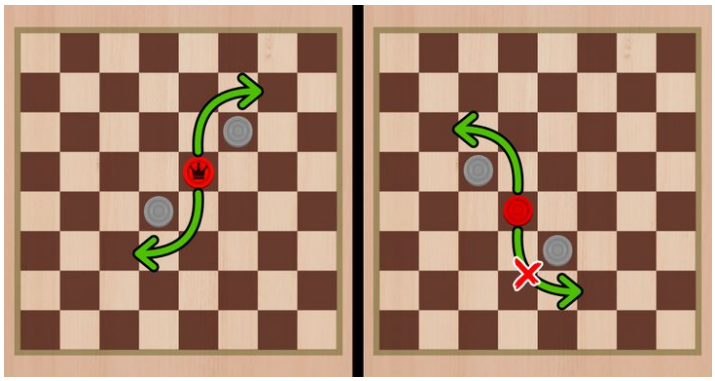

Figure 3: Our chosen rules for capturing pieces (no multiple jumps or forced captures)

2.5 Allocating Rewards

We mentioned in class that rewards are usually given only for terminal states (1 for winning and -1 for losing), with all non-terminal states receiving a reward of 0. This is called a sparse reward, which, as the name implies, features very sparse rewarding with most actions resulting in zero reward. In actuality, there is another style of reward allocation known as dense rewards, which rewards for intermediate actions as well, which "shapes" the reward structure in order to guide the agent to the optimal policy. While dense rewards are usually preferred because sparse rewards can sometimes result in agents learning an unintended policy to achieve the goal (for example, when the British Raj offered money for dead cobras in India, people started breeding cobras instead of killing them). However, shaping an ill-defined dense reward system is dangerous, and since defining the rewards for intermediate actions is often difficult, we decided to implement a sparse reward system instead, which only rewards 10 for winning, -10 for losing, and 0 for ties.

2.6 Training Loop

After the code for the Q-function, updating the board representation, finding legal actions at each state, determining a winner, and giving rewards have been implemented, we can qualitatively describe our training process. We begin by initializing two arbitrary policies to play against each other. Simply put, for each episode, the agents take turns observing the legal moves at each state, choosing an action according to the approximate Q-value function, and updating the board state until a terminal state is reached. This check is done once after every player's move, which looks at the board state to see if either side has no pieces left or no legal moves to make. Once this happens, the appropriate rewards are given to each agent, and the next episode begins. Every 10 episodes, the policy is updated once with the accumulated rewards. Notably, we decided to try two training methods—one where the agent was trained against a random policy, and the other where the agent trained against itself.

To learn the Q function, the agent first observes its current state and selects an action. By executing this action, the agent receives a reward from the environment and transitions to a new state. This experience—the state, action, reward, and next state—can potentially be stored in an experience replay buffer. DQN then calculates a target Q-value. The difference between this target Q-value and the current Q-value predicted by the main Q-network determines the loss. Using backpropagation, the DQN updates its network weights to reduce this loss. Through repeated interaction and updates, the DQN gradually learns to predict Q-values that guide it towards actions that maximize its long-term rewards in the environment.

2.7 Exploration-Exploitation Tradeoff

In reinforcement learning, there is a core principle is known as the exploration-exploitation tradeoff. In short, because the algorithm has very limited knowledge at the beginning, we must find a balance between picking known actions (which may very well be suboptimal) to maximize the immediate rewards and exploring other moves and their effects, which may lead to greater long-term rewards. If we do not explore new moves, we risk sticking with a suboptimal policy when there are much better ones to be discovered, but if we explore too much, our model won't maximize reward as intended. This delicate balance is why it's known as a tradeoff. In checkers, each side has twelve pieces, thus there are many possible actions at any given state. Given this information, we decided on an initial exploration rate of 0.3, meaning 70% of the time, the agent will choose the greedy action, and 30% of the time, a random action will be taken instead.

2.8 Performance Metrics

To evaluate the strength of the policy the agent has learned, we must decide on several metrics to quantify its performance. Benchmarks are a popular evaluation method for tasks like image classification, but unfortunately, nothing of the sort exists for checkers. However, there are still ways to measure how successfully our agent converged to a policy as well as the quality of that policy itself. These are the following:

- Winrate over time
  - By plotting our agent's winrate vs a policy that only makes random moves, we hope to see the winrate approach near 100 as training continues, since a random agent should not stand a chance against a trained agent
- Cumulative reward over time
  - Similarly, by plotting cumulative reward over time, we expect to see a constant increase. This is because as the agent learns, its policy becomes stronger and should aim to maximize rewards by winning games
- Average game length over time
  - Finally, we expect to see a steady decrease in average game length (measured by number of moves) over time because although the agent starts off by making random moves, it will gradually learn a strategy to take the opponent's pieces and win the game much faster
- Q-values at select states
  - Although not a quantified metric like the other three, we are able to display the Q-values for all possible moves in certain states of the game. This allows us to manually inspect how our agent is choosing moves, and whether they are reasonable or not.

3 Theoretical Justifications

While the most of the rationale for our model and design choices were explained in the methodology section, there are still a few more subtleties left to explain.

3.1 Checkers as a Markov Decision Process

A Markov decision process is a mathematical representation of a decision-making process, where at each time step in state s, the agent chooses an available action a. The state changes from s to s' according to the corresponding transition probability. At first glance, checkers may not appear to be a MDP because each action in a given state can result in only one new state without any randomness (thus being deterministic). However, since the opponent's moves are unknown, the state changes occur with uncertainty after they move, thus adding the element of randomness to the state changes. Markov decision processes are the foundational concepts behind reinforcement learning, as they provide a framework for the agent to iteratively explore and exploit actions to update its policy. Since we can model checkers using MDP, it is more than appropriate to apply reinforcement learning to this task.

3.2 Nature of Action-State Spaces

As previously mentioned, checkers has around 5×10^{20} possible states. However, due to the starting positions and the nature of how the pieces move, a large proportion of these states are never encountered. Additionally, each piece has at most two moves (four if the piece in question is a king), but almost all of these moves are blocked at any given state thanks to the board size or other pieces. These two facts mean the possible moves at any state—as well as the possible states in general—are much less than we initially expected, although there are still a staggering amount of possible state-action pairs. However, this has dramatically improved our situation, and it should be enough for deep Q-learning to handle, as neural networks are more than capable of learning relationships in massive quantities of data.

3.3 Potential Points of Failure

Unfortunately, there's no guarantee that the Q-values will converge in Deep Q-Learning. DQN uses deep neural networks to approximate the Q-values, which are inherently non-linear. While powerful, this non-linearity can introduce instabilities during the training process, preventing convergence. DQN updates Q-values based on targets calculated partly using the same network being updated (the target network within DQN helps mitigate this issue, but it's not foolproof). This "moving target" problem can lead to oscillations with non-convergent behavior. Lastly the choice of the learning rate, network architecture, experience replay size, and other hyperparameters significantly affect the stability and convergence potential of DQN.

4 Results

4.1 Performance against Random Agent

We trained our agent against a random agent for 3000 games (and updated policies every 10 games) to produce a final policy using the deep Q-Learning structure specified above in methodologies. We can evaluate its performance by plotting the following metrics to visualize our agent's training process and improvement over time.

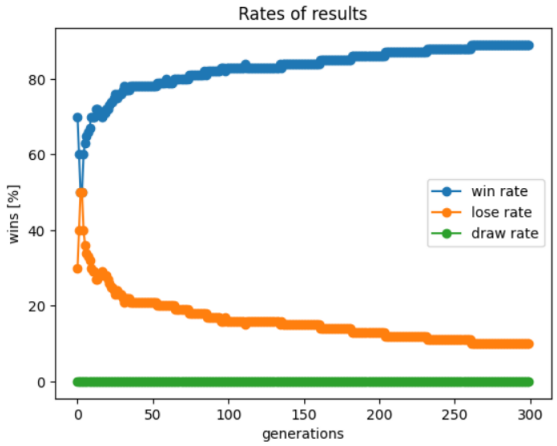

Figure 4: Winrate of our agent against a random policy over time, which stabilizes around 85%

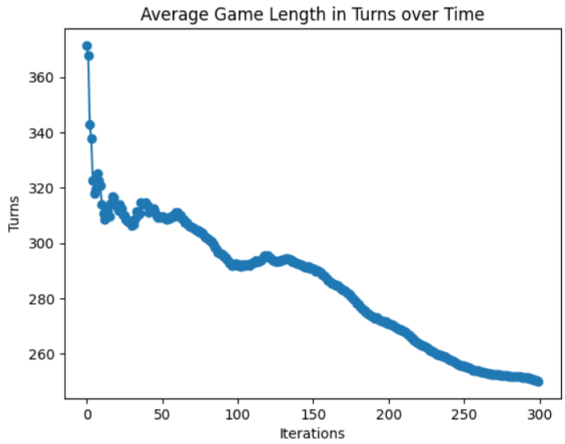

Figure 5: As the policy updates, our agent moves more deliberately and requires fewer turns to win.

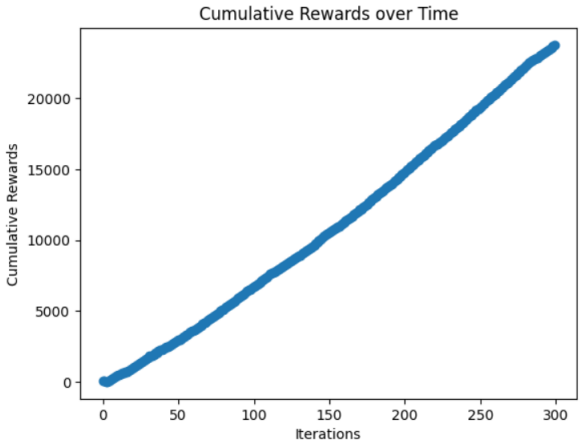

Figure 6: Model is reliably maximizing rewards over time through consistent victories

Through these three figures, we observe that our agent performs extremely well against an opponent that makes completely random moves. While this is not a terribly high bar, the improvement of our agent is immediately evident from the figures above, which signify that through training, our agent has learned a strong strategy to maximize rewards through winning. Additionally, the agent learned to

make efficient and deliberate moves to win games quickly, avoiding wasting actions on random moves that don't contribute to a winning position. Our expectations for an increasing winrate, decreasing game length, and increasing cumulative rewards have all been verified. Interestingly, the agent never ties once with the random agent. This is likely because ties mostly occur when each player has one piece left, where it is usually impossible to force a capture. However, an opponent that makes random moves will eventually move in front of the agent's piece, allowing for a free capture and victory.

```
GENERATION 30
   0  1  2  3  4  5  6  7
0 [ ,  ,  , b,  , b,  , b]
1 [b,  , b,  , b,  , b,  ]
2 [ , b,  , b,  , b,  , b]
3 [ ,  ,  ,  , r,  ,  ,  ]
4 [ , b,  ,  ,  ,  ,  , r]
5 [r,  , r,  , r,  ,  ,  ]
6 [ , r,  , r,  ,  ,  , r]
7 [r,  , r,  , r,  , r,  ]
Moves      Q-Values
(4, 7, 3, 6) [3.4649403]
(5, 0, 3, 2) [3.1207814]
(5, 2, 3, 0) [2.907558]
(5, 2, 4, 3) [3.1295598]
(5, 4, 4, 5) [3.0904548]
(5, 4, 4, 3) [3.1175513]
(6, 7, 5, 6) [2.9882672]
(7, 4, 6, 5) [2.8124106]
Agent chooses move: (6, 7, 5, 6)
```

```
GENERATION 150
   0  1  2  3  4  5  6  7
0 [ , b,  , b,  , b,  , b]
1 [b,  , b,  , b,  , b,  ]
2 [ ,  ,  ,  ,  ,  ,  , b]
3 [ ,  , b,  ,  ,  , b,  ]
4 [ ,  ,  ,  ,  , r,  , r]
5 [r,  , r,  , b,  ,  ,  ]
6 [ , r,  , r,  ,  ,  , r]
7 [r,  , r,  , r,  , r,  ]
Moves      Q-Values
(4, 5, 3, 4) [4.0385838]
(4, 7, 2, 5) [4.0418334]
(5, 0, 4, 1) [3.8793628]
(5, 2, 4, 3) [3.988312]
(5, 2, 4, 1) [4.239991]
(6, 7, 5, 6) [3.9811606]
(7, 4, 6, 5) [4.095106]
(7, 6, 6, 5) [4.1203127]
Agent chooses move: (7, 4, 6, 5)
```

```
GENERATION 270
   0  1  2  3  4  5  6  7
0 [ , b,  , b,  ,  ,  , b]
1 [b,  , b,  , b,  , b,  ]
2 [ , b,  , b,  , b,  ,  ]
3 [ ,  , b,  , r,  , b,  ]
4 [ ,  ,  ,  ,  ,  ,  , r]
5 [r,  , r,  , r,  ,  ,  ]
6 [ , r,  ,  ,  , r,  , r]
7 [r,  , r,  , r,  , r,  ]
Moves      Q-Values
(5, 0, 4, 1) [4.053127]
(5, 2, 4, 3) [4.10713]
(5, 2, 4, 1) [4.2762203]
(5, 4, 4, 5) [4.214704]
(5, 4, 4, 3) [4.5505004]
(6, 5, 5, 6) [4.1346655]
(6, 7, 5, 6) [3.8903244]
(7, 2, 6, 3) [4.460596]
(7, 4, 6, 3) [4.3483872]
Agent chooses move: (5, 4, 4, 3)
```

Figure 7: Example of the agent choosing moves in varying states according to Q-value approximations

Finally, we can observe the mechanisms of our agent's decision process a bit closer using various board state snapshots at different stages of training. For this figure, three board states along with their Q-values were randomly pulled from the model's training process. Although our exploration rate was set to 0.3, it just so happened that the agent chose to explore in all three of these board states, choosing a random action rather than the one with the highest Q-value. Using this method greatly helps us understand our agent's learning process by manually inspecting whether or not the moves it chooses are reasonable at different stages of training.

4.2 Performance against Itself

Additionally, we wanted to see if the training process would differ if we trained the agent against itself for 3000 games (again updating policies every 10 games) to produce another final policy using the same deep Q-learning technique. Self-training may seem pointless, since one would naturally expect the winrate to sit at around 50%. However, it is important to note that the model was only trained on one side of the board, that being the red pieces. Thus, we should expect to see decent results from playing the agent against itself. We can again evaluate its performance by plotting the following metrics to visualize our agent's training process and improvement over time.

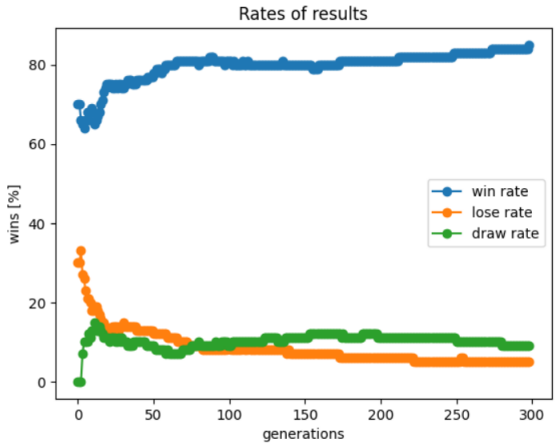

Figure 8: Winrate of our agent against itself over time, which again stabilizes around 85%

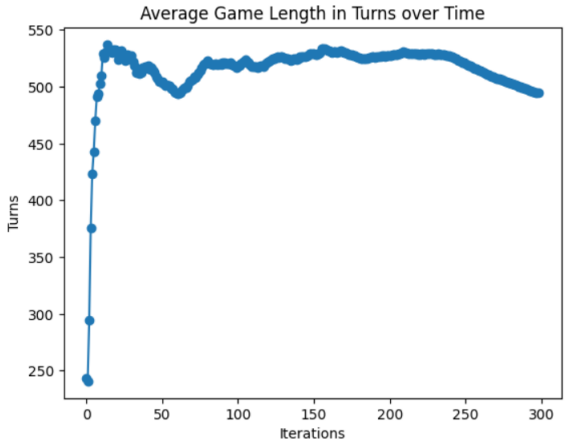

Figure 9: This time, game length doesn't decrease as dramatically

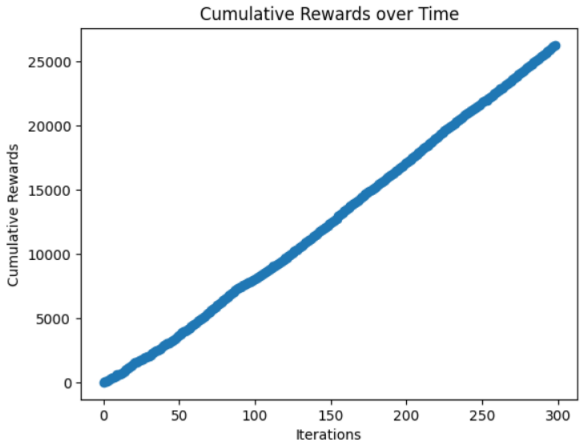

Figure 10: Cumulative reward reaches around 26000 this time compared to 24000 before

We observe similar, although not identical findings with our three figures for the agent training against itself. The most notable difference comes from the average game length over time, which remains relatively constant and doesn't drop off like it did when training against the random agent. This is

likely a result of all of the ties, which account for around 15% of all games. We chose to declare a tie when games went on for too long, and considering our last explanation for why the agent never tied with the random policy, it makes sense why there are much more ties in this situation. The agents must have been playing intelligently enough to not make a mistake when both sides were down to one piece, which means many games were terminated and declared ties. Also notable is the slightly higher cumulative reward, which probably results from ties not being penalized, while previously losses were heavily penalized at -10. It is quite interesting to see the variations in performance, and we would have liked to test our two agents against each other if we had more time.

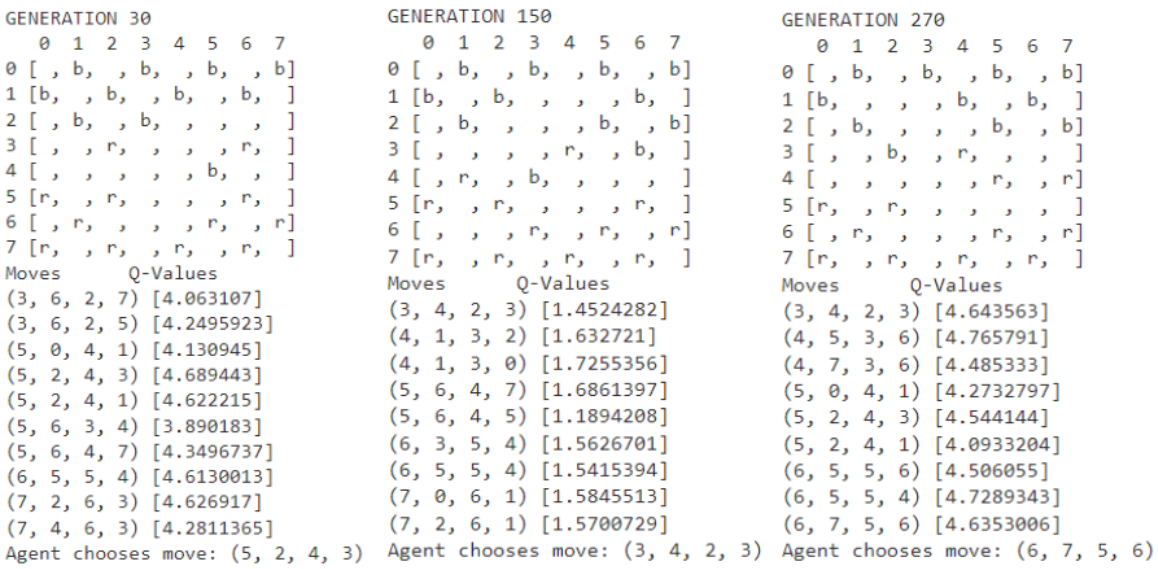

Figure 11: Example of the agent choosing moves in varying states according to Q-value approximations

There is not much to add for this figure. Observe again that at generations 30 and 270, the agent chose the move with the highest Q-value, while in generation 150, it chose to explore another move. This is consistent with Figure 7 and serves a similar purpose for troubleshooting and insights.

4.3 Challenges and Takeaways

Throughout this project, we faced many more challenges than expected. Firstly, implementing the board representation and determining all legal moves at every state was fairly time consuming, especially since special rules for kings added a whole new level of complexity to the mix. After that was finally finished, we successfully implemented Q-learning, or so we thought. Although the code ran, we noticed the Q-table never converged no matter how long we trained it, and our agent's performance was poor at best. As previously mentioned, the problem was that Q-learning is inadequate for such a large-scale and complex game such as checkers. This took quite a while for us to realize, and we were left without much time to find a solution. However, we were able to successfully implement deep Q-learning, which solved many of these problems, but not without a whole lot of bugfixing at every step of the way.

This project has given us a newfound level of respect for reinforcement learning, as implementing everything was an incredible challenge. The fact that agents can learn without any data at all is nothing short of incredible, and we walk away from this project being much more experienced in both the theoretical background and technical implementations of reinforcement learning.

5 Future Extensions and Questions

5.1 Reward Shaping with Dense Reward Allocation

As previously mentioned in our methodologies, implementing a dense reward system would be difficult due to defining which intermediate actions during the game should merit rewards (controlling the center, capturing pieces, kinging pieces, etc), but doing so could potentially expedite the training process since there exists a more defined course of action to maximize rewards rather than simply "win games". We are curious as to how much this would improve our model's overall learning and performance, given the dense reward structure is appropriately and thoughtfully defined.

5.2 Experience Replay

Consecutive samples in the environment are often highly correlated (e.g., consecutive states in a game). Training on such data can bias the learning process towards these correlations and hinder generalization to unseen transitions. A potential solution would be using Experience Replay. Experience Replay stores past experiences (state, action, reward, next state) from the agent's interactions with the environment in a replay buffer. During training, the DQN samples random minibatches from this buffer, breaking the correlations between consecutive samples. This provides a more diverse and representative dataset for training, promoting better generalization and potentially aiding convergence.

5.3 Benchmarking

Something interesting for the future would be to test our agent's winrate against popular online checkers bots. While there is no standard ELO system like in chess, many websites feature the option to play against a computer opponent, which usually come in easy, medium, hard, and sometimes even extreme difficulties. While not a traditional standardized benchmark by any means, it would still be worth creating some sort of program or mechanism to input our agent's moves against the online opponent and the opponent's moves into our agent's gameplay interface. By automating this process, we could get a good idea of our agent's performance against opponents of different skill levels. Additionally, we could add a human gameplay interface to have humans of varying degrees of skill play our agent, which would be a really fun extension to this project.

5.4 Robustness

As previously mentioned, the dimensions of the checkerboard can vary, with 8x8, 10x10, and 12x12 all being common. Since our agent was trained on the 8x8 board, it would be very interesting to see how the agent generalizes to different board dimensions. Obviously, the set of legal moves at each state would have to be updated, but beyond that, would the neural network used to approximate the Q-table be able to generalize the relationship between values to a board with larger or perhaps even smaller dimensions?

5.5 Transfer Learning for Different Games

The concept of transfer learning is quite important in other areas of deep learning. Simply put, it is extremely expensive to train models from scratch every time. Transfer learning alleviates this by leveraging pre-trained models for their recognition of general, low-level patterns and fine tuning them to handle finer details for a more specific task. While quite a stretch, we are interested if this concept could be applied to reinforcement learning as well. In particular, would it be possible to train an agent on another board game and have it utilize the knowledge from that game to accelerate its learning process for checkers? Or in the other case, could we take our agent and apply it to another game environment somehow?

5.6 Implement Additional Rules

We previously chose not to include multiple jumps or forced captures in our ruleset for the sake of simplicity. However, it would be worth implementing these in the future to make the game more interesting and give the agent an additional challenge. We actually hypothesize that requiring pieces

to make a capture if the opportunity is available would aid our agent's training. This is because the overall game duration would decrease since the rate at which pieces are captured would dramatically increase. This then leads to faster training and hopefully a stronger learned policy. However, this is only an educated guess, so we would be interested to see how this actually plays out.

6 Code

https://github.com/leighannelem/checkersRL_math156_24

7 References

- "Reinforcement Learning: An Introduction" by Sutton and Barto
- https://towardsdatascience.com/reinforcement-learning-implement-tictactoe-189582bea542
- https://www.science.org/doi/10.1126/science.1144079
- https://www.chessnboards.com/blogs/tips-tricks/checkers-game-rules
- https://forns.lmu.build/classes/spring-2020/cmsi-432/lecture-13-2.html
- https://github.com/theohern/Checkers/blob/main/report.pdf