Project Report

-Vladimir Necula, Connor Brown, Aidan Mathieu-

Problem Description

At face value, Connect 4 is a very simple game. Players take turns placing pieces into columns, with the first player to make 4 pieces in a row horizontally, vertically, or diagonally winning. However, with over 4.5 trillion board states, this game is more complex than it seems, and allows for the development of many strategies to force your opponent into a losing position. This makes Connect 4 an ideal game for players of all ages. Unfortunately, this does not mean that Connect 4 is accessible to all individuals. Individuals who are blind or have difficulty moving the pieces may find it challenging to play with either the physical board or existing online versions of the game. For this reason, we have decided to develop a program for individuals to be able to play Connect 4 without touching or seeing the board.

Implementation, Challenges, and Decisions

Implementing our solution to this problem came in two major parts: developing a reinforcement learning model to play against, and developing a UI with the desired accessibility features. Both of these came with their own challenges.

1. *Developing the RL Model*

Our first attempt to tackle the idea of training a RL model was by using a Deep Q-Network (DQN). Despite the simplicity of the game board and rules, any DQN model trained struggled to grasp more than the very basic strategy of the game. To try to improve the model, we attempted many different combinations of features, rewards, and training strategies.

Originally, the model was trained as a single agent intended to play both sides of the game. The idea behind this approach was that by experiencing both perspectives, the model would gain a deeper understanding of the game's dynamics and better evaluate the quality of moves. However, this strategy proved ineffective in practice. Given that an ideal game of Connect4 heavily depends on which player goes first, we shifted to a two-model setup.

This is where you train two models, the first being the model that starts the game, and the other being the model that plays second. This model was able to grasp the basic concepts of

blocking a vertical three in a row, but rarely blocked any horizontal or diagonal threats. This led to a series of reward tweaks where we attempted to get it to identify three in a row more effectively. This was done with greater penalties for missing a block if a threat was present or not making a winning move if it existed.

Additional considerations for improving the model's performance included adjusting the complexity of the neural network, modifying the experience replay buffer, and fine-tuning the balance between exploration and exploitation. The latter was controlled by tweaking the epsilon decay value, which determines how frequently the model selects a random move during training. An epsilon value of 1 means that it will make a random move 100% of the time. Our system used epsilon decay, where the epsilon value started at 1, and then that value would decrease every episode of training. By tweaking that epsilon decay value, we could determine how much it tried new moves before settling into where it believed the best moves were.

Increasing the complexity of the model and changing the buffer were difficult to implement. Making these more complex to be able to store more information also meant that it would take longer to train or require more computing power, which we do not have access to. We did consider other changes, such as including dropouts from the neural network, but as it seemed to be overfitting to their individual play style, that did not appear to support the challenges we were facing. A prioritized buffer could also be made where it keeps more important information towards the top of the buffer, however, the process to sort this information was too costly as well. Despite the many efforts made to tweak the model, we could not seem to get a DQN to play the game very well, and it would normally revert to simply stacking the pieces in columns. Doing some research online allowed us to see that we were not the only ones with this issue, and it appears that a DQN is simply unable to play Connect 4 well. This led us to attempt some other strategies to make an improved model.

The first alternative we explored was a simplified AlphaZero model developed by DeepNet, which uses Monte Carlo Simulation to project future moves and evaluate their quality. However, this approach proved too computationally intensive to train effectively within our available timeframe.

We then moved on to the strategy of using a DQN but with some hard-coded moves. This forced the model to make a blocking move or a winning move if one exists. The upside to this is that it allowed the model to perform better as it was now making moves that it should clearly be making, and then the model's training could focus purely on the strategy of setting up winning positions. The model still ended up making moves that seemed less than strategic, with some stacking of pieces.

As a last-ditch effort, we managed to find a model that plays the game perfectly, as it is a solved game. This model allowed us to make API calls to it that would evaluate every move made by the model. This took out the guessing game for us of whether the move that the reward function that we were using was good, and it allowed the model to learn from each and every move that it made.

We ended up using the model that was being trained on the perfect model API. This model ended up playing the game well, and we felt it was more in the spirit of a machine learning project to not hard code blocking moves into our model.

We believe that our models often failed to work due to a phenomenon known as self-play overfitting. This is described by OpenAI here: https://openai.com/index/competitive-self-play/ The overfitting occurs due to the models purely being trained on their own actions. This causes them to develop strategies that maximize rewards when they are facing each other, but when facing a different opponent it's strategy will not work. When training these models, you want it to learn general strategies to be able to play effectively against any opponents. Unfortunately, we were unable to develop a model with a reward system that was able to do this effectively on its own with assistance from hard coding moves or outside API calls.

2. *Designing the UI*

Making the UI also came with some considerations. We are trying to make an environment that is usable by individuals with disabilities who may struggle to interact with a physical board. The final version of our program would be an environment where individuals could simply speak to their device in order to give commands based on which column they want to drop the piece. This would allow the individual to constantly speak to the program, ideally without having to press any buttons at all after it is set up. There would also be a command to have the board be read out to the player. Those who cannot see the board would now be able to easily get updates on the state of the board in order to make a more informed decision. These players would then be playing against a machine learning model that could be set to different levels of difficulty based on how much of a challenge is desired. In this way we hope to bring more accessibility to this game.

However, when developing this UI, we came across some difficulties. Mainly due to the limitations of the Colab notebook that we were using. Google Colab did not allow for any audio processing, which caused us to resort to testing our features in a separate Jupyter notebook using Visual Studio Code. Through this, we had to have it work by importing audio files, which are

then processed to perform a command. While this is not ideal, it shows proof of the ideal concept of the game.

Methodology

We explained how we went through all implementations, showcasing our decision-making process while navigating from one to another. However, here, we want to explain in more detail how we evaluated each of the models that we dealt with.

Basically, after training each of them, we first looked at the metrics that we recorded:

- Loss over time
- Game Outcomes
- Rewards over time
- Win rate (for each existing player)
- Epsilon decay

Then, if they looked in acceptable ranges, we started evaluating the model against either some tests (to better understand its preferences) or against random players. This step is not present in the notebook, as it was mainly for our better assessment of the performance.

In the end, the final assessment was done by us, playing against the agents through the use of the UI. Usually, this is the step that ruled out most of the models, as this is where we have seen the stacking behaviour, or where we could assess the performance against the “human baseline”.

Future Considerations

Technically, if we wanted to make the model be able to play the game perfectly, we could use a Mini-Max algorithm, which could search through all possible future moves and recursively determine which move is ideal. This is doable as Connect4 is a solved game. However, it comes at a huge computational expense. Thus, future iterations of improving the agent would ideally combine the model setup that we have now with a Minimax approach.

With a model that performs excellently, we would then have to worry about making the game “fun” (after all, players need to also win sometimes). Consequently, later UI and Gameplay designs should integrate a slider that determines how often the model randomly chooses a move other than the one it knows that it should make (or just offer difficulty level options).

We would also want to improve on the UI by having the game run in an environment where it could receive a consistent stream of audio commands. In an ideal future, models such as this would be used for all sorts of games to help bring accessibility to these activities.