Glenhaven President Neural Network Report

A final project report for COMP 3106

Dawson Theroux (SN:101106602)

Due: December 9th 2022

Introduction

Training a neural network to a card game is not a new idea by any means, which is why I decided to create a neural network to play a very specific card game which the exact version is only played with my friends and I on Glenhaven road. After this project is done, I am hoping to be able to use this model to be able to analyze game positions to help improve my play with my friends.

Prior to this project, I had a vast wealth of knowledge about playing this version of the President card game, but no knowledge about neural networks. As the semester went on I gained knowledge about neural networks through a neural networks class which helped me debug some of the issues I was having regarding my model.

How to play President

The game starts with the all of the cards in the deck (including Jokers) are dealt to all 6 players. The goal of the game is to finish playing all your cards as fast as possible. The first person to finish their hand becomes the President, second becomes Vice President, third becomes Neutral1, fourth becomes, Neutral 2, fifth becomes Vice Ass, and sixth becomes Ass. The game loop works as follows. A player leads one of a kinds, two of a kind, three of a kind, or four of a kind, all subsequent players have the opportunity to play an equal or higher value play on top, i.e. if the first person plays two 4s, the next player must play two cards that are equal or higher, such as: two 4s, two 5s, or two 6s and so on. If a player plays the same cards on top of the cards on the table a BURN occurs, which means that the player who played the same value card on top can clear the table and lead whatever they so desire. Card Strengths can be broken down as the following:

“A” < “4” < “5” < “6” < ,…, < “J” < “Q” < “K”

As you can see, “2”, “3”, and “Joker” are missing from the above ranking. This is because “2”s, “3”s, and “Jokers” are power cards. Power cards have the following special properties.

Any Single < “2” < “3” < “Joker” < “Bomb”

Any Double < “2” < “3” < “Joker” < “Bomb”

Any Triple < “2,2” < “3” < “Joker” < “Bomb”

Power cards are special in that a single power card (in some cases) can be multiple regular cards. The main glaring drawback of power cards is that if you play a power card as your last card you AUTOMATICALLY get last place. In the above strength chart, one can notice “Bomb” is apart of that list. Bombs are four of a kind and beat everything. Bombs only be beat by another bomb of a higher value.

“4,4,4,4” < “5,5,5,5” < ,…, “K,K,K,K” < “2,2,2,2” < “3,3,3,3”

The last thing to note is that playing bombs of power cards will still lead to automatic ass. i.e. playing “2,2,2,2” as your last play will net you Ass, but “4,4,4,4” will not.

Prior work in the topic area: Prior to this project I have had no experience in this topic. The only experience I had was with the game President which in the long run help me identify overfitting without fitness because I was able to see if the model was learning the wrong things against itself. I also had almost parallel experience because I started this project before learning neural networks in COMP 3105, but by the end we covered them extensively.

* Significance of the project: This project is significant because it can help identify good play among my friends because we have recently changerd the rules in how we play president and where unsure of the best way to play 2s. This project is also significant because
* The objective of the project: To train a model to play a good strategy of Glenhaven president.

Methods

In order to train a neural network to play the card game President, the input data must first be acquired from source. To acquire the original input data and to be able to test the model that was trained, I first had to implement President in python. This was achieved through five classes, the GameClass, the the PlayerModule class, and three Card Interface classes. The GameClass holds 6 PlayerModules objects and handles all the logic of the game. The PlayerModules objects each hold player information like ID and a Card Interface object. Finally the Card Interfaces classes handle prompting the different kinds of players for cards. The three kinds of Card Interface classes are: CommandLineInterface, RandomCardInterface, AIModelInterface. These three interfaces are necessary for the game to be played by a player through the command line, an script that randomly selects possible plays, and a pytorch model.

Now that a president game has been implemented, I was able to use reinforcement learning to train a neural network to play it. The first thing to determine is how each game state will be encoded so that the neural network is able determine a play. This is done through 1 hot encoding where the value in the list representing a certain state is set to 1 if it is true and false otherwise. Here is how one input row, or one play, is encoded where the number in the parenthesis is the number of values required to encode the information:

The number of players still in the game (6)

The number of players still in the game represented by 6 values represents the number of players still in the game. i.e. at the start of the game the values are all 1s and at the end of the game the values are all 0s.

The players hand encoded (54)

This represents all the cards in the players hand where each value is a 1 if the player has the card and a 0 if the player does not. So indexes 0-3 are the 1 cards, indexes 4-7 are the 2s and so on. There are 54 values because president is played with Jokers.

The cards on the table encoded (54)

The cards on table enoded represents the play that the player must play on. These values are represented as follows: single 1 is index 0, double 1 is index 1, triple 1 is index 3.

All the cards discarded encoded (54)

These values are encoded the same way as the cards in the players hand where each value corresponds to a card.

Player Previously passed (1)

Previously passed represents if the player has passed on this set of cards previously. This value is reset to 0 once the play is cleared and another player leads.

All this data means that the input vector for a game state into the neural network is 168. The target for the neural network is the play chosen by the player during data acquisition, which is then encoded with the score they got in the rankings where:

- President: 3

- Vice President: 2

- Neutral 1: 1

- Neutral 2: -1

- Vice Ass: -2

- Ass: -3

With the input shape determined, a basic neural network was constructed with the following layers:

\*\*ADD DIAGRAM\*\*

The strategy used to train this neural network will be a simple reinforcement learning process where each generation starts by taking in the last generations game data for training. Then once training for that generation is complete, new training data is created using the next iteration of the model. The only problem with this is that we do not have the initial input data to train the model. This issue was solved by implementing a random player through the RandomCardInterface which selects a play randomly out of the possible plays. This method yields around 130000 (excluding plays where the only possible play was to pass) plays over 2000 games.

Now that the first iteration of training is complete, how do we use the model to play 2000 more games to use as input data for the next generation. The AICardInterface is can take in a model and use it to prompt the model for a play after encoding the game state. The play that is chosen is the model output with the highest value and is also a legal move.

With the random player generating the initial input data, and the intermittent models being able to generate the next generations input data, the model must tuned in order to create the best results. When I began this process I had not yet implemented a fitness function. Instead, I simply printing out all plays of one game after the model is done training. This let me observe the decisions made by the model which was adequate in judging the performance of the model because the best move tends to be evident to an experienced player. \*\*TALK ABOUT IMPLEMENTING A FITNESS FUNCTION\*\*

\*\*Missing Loss function\*\* \*\*Missing all optimizations: Optimizer, epochs, generation, fitness value, dropout rate\*\* \*\*Missing talking about generating data with bots in the game vs without\*\*

- Benefits/drawbacks of methods:

* + There was a while that I did not have fitness values which made it hard to determine how the model was doing.
* Evolution/validation strategy:
  + Fitness value.
  + Played this game a lot so just looking at one of the game outputs I am able to objectively determine if the bot made good moves or bad moves.

Results

* Qualitative results:
  + Me looking at the games
  + They play correctly and save power cards till the end
  + Reassures my question about 2s being played early.
* Quantitative results:
  + Fitness score of the bot (against prior model?) against random?

Discussion

* Limittation and biases
* Implications of the work??
* Analysis of results/outcome with respect to objectives
* Potential Improvements/ future work.