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. Before starting this project, I had no prior knowledge of Machine Learning, but I was taking a Machine Learning course this semester which helped me in the ladder half of the semester when we started learning about neural networks. This was my first time creating my own neural network from scratch including choosing the network shape and how the input and output data would be encoded. This project meant a lot to me because my roommates and I love to play president, and we have evolved the ruleset to be as competitive as possible over time. I also believe that we have developed an optimal strategy, so I was hoping that training a neural network to play would help us refine the meta even more. So, with this project, I was hoping to be able to play and practice president against a high level whenever I pleased. Also training a neural network means that I could also input various game states into the neural network and be able to analyze the output produced to determine what the true best strategy is as defined by a neural network. When this project is complete, I plan on running experiments with it. For example, I would like to determine how much the hand delt to you impacts your placement in rounds as well as the probability of moving up or down in positions after each round. Overall, the goal of this project was to train a model to play president as optimally as possible to be used to research and pleasure purposes.

How to play President

The game starts with all 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 either one of a kind, two of a kind, three of a kind, or four of a kind, then 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, 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 “Joker”s 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 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 can 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.

# Methods

Compare model 7000 and 8000. They are both after the fix, but 8000 uses the new training method and 7000 does not.

### President Implementation

Since training a Neural Network to play president requires a way to play president, this project started off with a President card game implementation in Python. The president game was programmed in python with five underlying classes: *Game, PlayerModule, CommandLineInterface, RandomInterface, AIModelInterface.* The game class handles all the game logic like dealing the cards and validating plays. A *Game* object holds 6 *PlayerModule* objects. *PlayerModule* objects hold arrays which contains the hand of the player a prompt card function. The prompt card function prompts the card interface object held by the player for their play given the hand and the cards on the table. The *CommandLineInterface* class allows players to play the president game through the command line interface, the *RandomInterface* is a simple reflex agent which picks a random legal play, and the *AIModelInterface* is an interface which takes in a *PresidentNeuralNetwork* object model and picks plays based off its output.

### Neural Network Input Data

With a working President card game implemented in Python, I was then able to start training a model to play President. For a Neural Network to play games and learn, data must be encoded into vectors so that they are compatible with the input layer of the model. For President, I was able to encode the entire game state into an input vector of shape length 168. The following is a breakdown of the input data shape where the number in the parenthesis is the number of 1 hot encoded digit are required to represent it. For example, if there are 4 players in the game, the 1-hot encoding would be [1,1,1,1,-1,-1].

The number of players still in the game (6)

The 6 values represent the number of players still in the game. i.e. at the start of the game the values are all 1 and at the end of the game the values are all -1.

The players hand (54)

This represents all the cards in the players hand where each value is a 1 if the player has the card and a -1 if the player does not. 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 in the deck, so there are 54 cards.

The cards on the table (54)

The cards on table 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, and so on up to double joker at position 54.

All the cards discarded (54)

These values are encoded the same way as the cards in the players hand where each value corresponds to a card. If the card has been played previously in the game, then the value is one, otherwise the value is -1.

Player Previously passed (1)

Previously passed represents if the player has passed since the table was last empty. So, if the first play is a 2 and the player passed, then another player plays a 3 and it comes back to the given player, this value will be set to one. The goal of this value is to give the model more information on what they have done in the past to possibly encourage bluffing.

### Neural Network output

The output of the neural network is encoded the same way as the cards on the table are encoded in the input data. The output is of length 55 where each index out of the 55 represents a possible play. Index 0 is the pass option where the player decides to play nothing on top of the cards on the table. Indexes 1-55 are the encoded plays where index one is the play [1] (Ace) and the index two is the play [1,1] (Pair of aces) and so on, all the way up to index 55 which is [Joker, Joker]. The output also differs from the input in that it is not encoded with one hot values, instead the value stored in the position of the play selected is the resulting position from playing that play with the following mapping:

- President: 3

- Vice President: 2

- Neutral 1: 1

- Neutral 2: -1

- Vice Ass: -2

- Ass: -3

This essentially means that the neural network is trying to predict what position playing a certain play will net.

### Network shape

The *PresidentNeuralNetowrk* is implemented in PyTorch and is a Neural Network comprised of 168 input nodes connected to four fully connected hidden layers with layers of size 1024, 1024, 512, and 256 respectively. Finally, the output layer is of shape 55 for reasons described previously. Every layer is activated with ReLU activation except for the output layer which is activated with tanh activation. The reason for the final tanh activation is because the model is trying to predict values between [-3,3] so it must be able to produce negative value. The loss function used prior to back propagation scales the label being predicted on back by 1/3 so that it turns from [-3,3] to [-1,1] to match the prediction.

Text

Description automatically generated

Figure : Model Shape

### Training

Since this is a new implementation that has never been done before, there is no input data that exists to train on. Instead, the *RandomInterface* was implemented for this purpose. Training begins by calling a function that spawns 15 threads to run 1000 games each with the 6-player using the *RandomInterface*. This typically generates about 1.5 million rows of input data, or plays, to initially train the model. With the initial set of random data, the *PresidentNet* object can be created along with its Adam optimizer to perform backwards propagation.

The training loop starts by first running a train function which gets a prediction from the model and calculates the loss using a masked squared error loss function. Masked squared error means that the loss is only calculated on non-zero values as to not train the model on plays that never happened since all the placement values are {-3, -2, -1, 1, 2, 3} and the loss function is. The scale is required in the loss function because the tanh activation of the output only gives values between -1 and 1, so, the label must be scaled back by 1/3 since its values are between -3 and 3. Once the loss is calculated, the optimizer can perform a backwards propagation step for each minibatch of 32 from the data. Now that the model has changed since the last iteration it can be tested with the test function. The test function evaluates the fitness of the model using a fitness function. The fitness function tests the model against the previous best version of the model (random if none exist) and if the current model fitness is above a threshold of 50, the model state dictionary is saved to be tested against during future epochs. A threshold for the fitness was used because the inherent randomness of the game means that two models close in performance will typically perform in the range of -40 to 40. With the model tests, the next epoch can begin repeating the cycle until the model stops improving for the generation. When this happens, a new input dataset is created to train on by running 25 thousand new games with the best model from the previous generation.

The fitness function evaluates the model’s performance by running 1000 games against a competing model where each model controls 3 players. The model fitness starts at a value of 0 and once a game ends values are added to the fitness based on the positions the evaluation model players placed in. The values added are as follows: President: 1, Vice President: 1, Neutral1: 1, Neutral2: -1, Vice Ass: -1, Ass: -1. In the end, if the fitness is positive then we know the model being evaluated is better than the competing model since it placed in one of the top 3 positions on average.

Generating new input data for the next generation is done by running the *AIModelInterface* in training mode. Training mode differs from regular evaluation in that it will purposefully choose the second or third best choice instead of the first to gather more data new strategies that it has not tried. When in training mode it chooses the top prediction with 70% probability, the second best with 15% probability, and third best with 15% probability. This differs from other known strategies of generating input data that I found in other articles because in their examples they simply have the model play itself instead of using a more unique method of adding variation of the data.\*\*LINK AN ARTICLE THAT TALKS ABOUT JUST HAVING THE MODEL PLAY ITSELF AGAIN\*\*

Benefits/drawbacks of methods:

- Not using

* + 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

During training I saved the model state dictionary to google drive whenever a better is model is found. With this, I am able to visualize the model improving by having the best model of each generation play the random bot and add the fitness value to a graph. Here are the results I found.

Since I trained models using both my training method and the normal training method, I am also able to have the best model at each generation play each other to be able to gather information about how the different training data generation effects the resulting models.

The models are also able to be evaluated based on how they react to certain situations. Since I have a lot of experience playing president I have had a lot of time to refine my strategy, so after observing game outputs for all the genereations I am able to highlight the mistakes they make and see how they improve from a play to play prospective instead of just a fitness function prespective.

- Give examples of how the model improves at each step. Right now it looks like at step 1, they are not always playing doubles.

Discussion

Implications of work:

- Talk about how I could use this model to simulate many games and get data to see the probability that a player moves up positions from ass.

- Talk about how specific positions where my roomsates and I don’t know the best play can be identified.

Limitations of the project:

- I am not sure if it is possible to play with a different number of people with this model.

- You could probably use the same model as a starting point.

- The Hyperparameters may not have been tuned as best as possible because of time limitations and debugging.

Analyzing the results:

- Talk about if I actually found a good model that plays how I would expect

- Talk about the fact that I am actually able to play against a model so that part was a success. I atleasaed learned how to make a model and neural network work from a technical standpoint with python

-

Potential improveents:

- Using a less naïve way to choose the

-

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