Kyle Fleskes

Professor R. Mitchell Parry

CS 4800-101

6 May 2021

**Abstract**

The goal of this capstone project is to create a reinforcement learning AI that plays the game known as Exploding Kittens. Exploding Kittens is a Russian roulette-style card game where the players draw from a shared deck of cards while trying to not draw the titular exploding kitten. The rest of the cards in the deck are designed to help either make your opponent draw the losing card or help prevent you from doing so. The environment and AI analysis were handled by Andrew Coates while the AI was created by myself, Kyle Fleskes. Two versions of the AI were created and while both used a Monte Carlo Search Tree, one also used a Neural Network. The AI was modeled after Deepmind’s Alphazero, a powerful reinforcement AI that can beat the best Shogi players in the world. To make the AI easier to work with we reduced the ruleset of the game, limited the number of players to just two, reduced the number of card types, and card combinations that can be played. Some cards can be played at any time regardless of the player’s turn, so to reduce the complication of AI gameflow we omitted these cards.

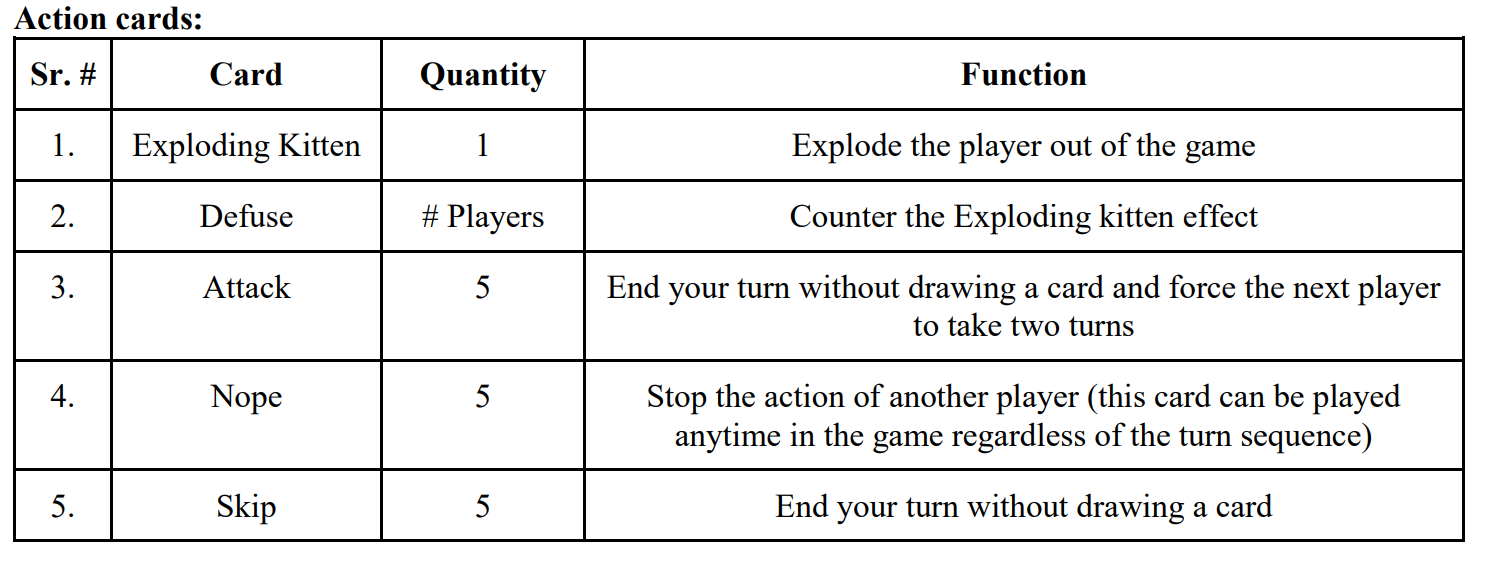
**Project Overview and Design:**

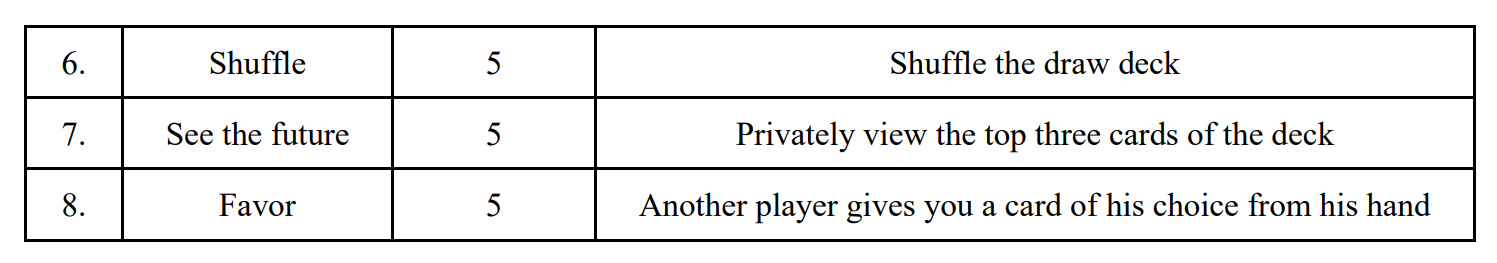
This project had a few inspirations, namely the overall design was influenced by a paper called “Reinforcement Learning for Exploding Kittens” by a collection of Stanford University students. Furthermore, the Exploding Kittens game simulation was adapted to Python from Javascript from this public github:<https://github.com/yisheng90/the-exploding-kitten>. The Monte Carlo Search Tree was heavily adapted from this github by Deepmind here: <https://github.com/int8/monte-carlo-tree-search>.

**What is Exploding Kittens?**

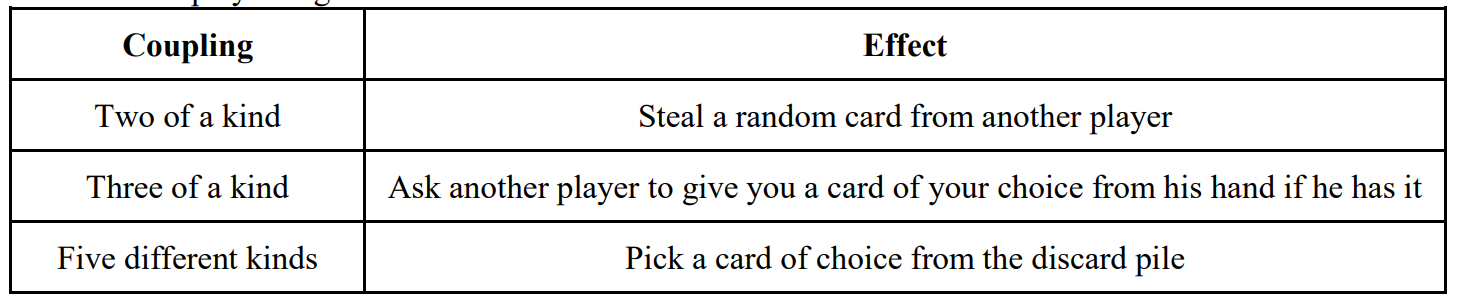
Exploding Kittens is a 2-6 player card game where the idea is to not draw the exploding kitten card from the deck which thus eliminates you from the game. The idea is to be the last player “alive” by having not drawn the exploding cards. Players must draw from the deck to end their turn. Therefore, the other cards from the deck are designed to protect yourself, manipulate the deck, and/or meddle with other player’s hands so you may have the smallest chance of drawing the exploding kitten card.

There are 13 different card types, 8 of which are action cards and 5 that are non-cards. See the following chart to see the different types of cards available.



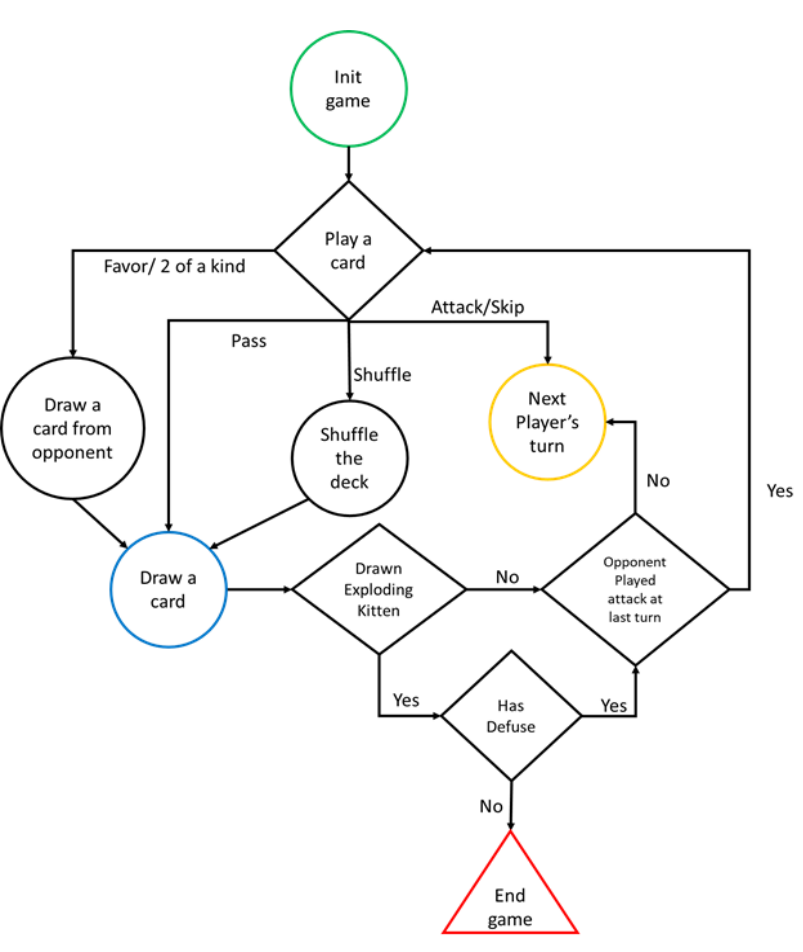


There are 5 types of non-action cards that cannot be played by themselves. They can only be played with a matching copy of that non-action card. However, playing different amounts at the same time has different effects. Our project can be found on github at <https://github.com/KyleFleskes/Reinforcement_Learning_Capstone>.



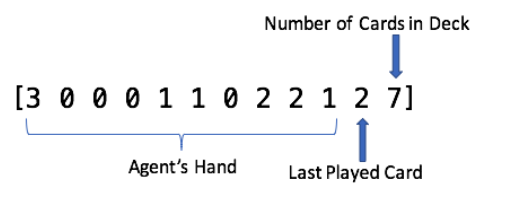
**Simplification of the Game**

Having an MCTS AI play the unabridged version of the game would be very difficult to implement, but it is possible. To start with, we simplified the game by reducing the number of players to just 2. We also remove the Nope, See the Future, three of a kind, and five of a kind from the game. This is because, in the case of the Nope card, having an action that could be played at any time would significantly increase the complexity of the action economy of the game. In the case of the card See the Future, it would be hard to implement a card counting ability and somehow incorporate that into the AI game state. As for allowing for three and five of a kind, it makes a lot of plays too conditional. Moreover, two of a kind can only be played with the 5 types of non-action cards as opposed to every card type like in the full ruleset. This allows us to have a relatively simple flow of play as shown by the chart below.



**Representation of Exploding Kittens to the AI**

The state representation of the game at any given point is shown in the diagram just below to the AI:

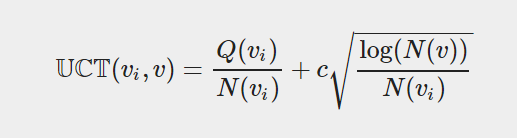


The first 9 numbers are the amount of each card type in the AI’s hand. The 10th card is the index of the card type last played in the game, and the last number is the number of cards still in the deck.

The actions the AI can do are playing a card in their hand, or a draw action where the player must end their turn by drawing a card; unless the AI played a skip card during their turn or ended their turn with an attack card.

**Monte Carlo Tree Search(MCTS)**

The idea is to create a Monte Carlo Search Tree that explores the gamespace stemming from the current board state using a Q function:



V\_i is a child node of node V, where Q(V\_i) is the win rate for that node. N(V\_i) is the number of times that node was visited and C is an exploration parameter.

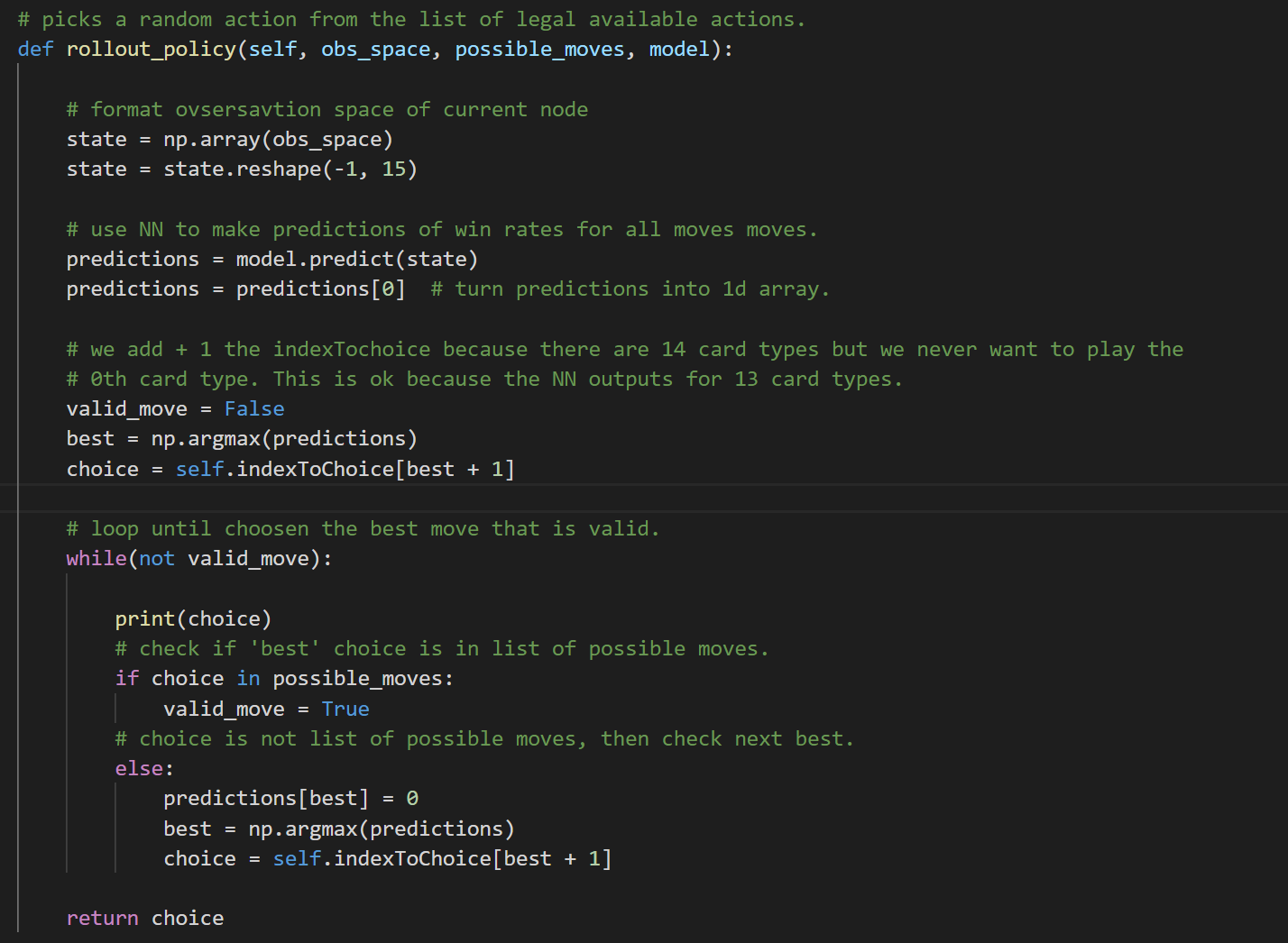
The Neural Network will be using 1 input layer, and 3 hidden layers with 1 output layer. The input layer takes in the described gamestate and the output layer predicts the estimated win rates for each possible move. The NN will help the MCTS simulate games instead of just picking random moves until a win or loss.

**Development and Testing**

Much of the first ⅓ of the semester was Andrew creating the AI environment and I getting an understanding of what a Neural Network is and how it works. For the midterm Andrew Coates finished the overall simulation and I created a proof of concept that I knew how to create a Neural Network. See midterm presentation as evidence. After the midterm, Andrew and I both worked together to fit my Monte Carlo Search Tree to work with his simulation; a difficult task as there were many unforeseen bugs and complications. See just some of the github commits as evidence of this:



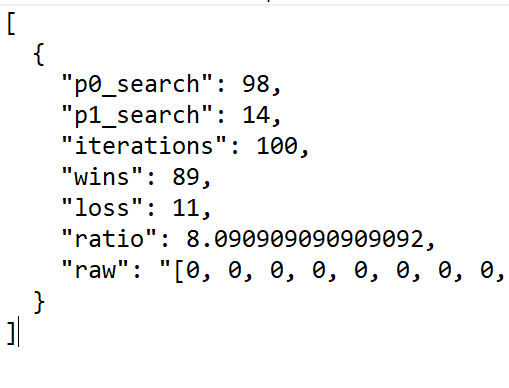
After the code was functional I added in the Neural Network to the MCTS. For evidence of this please view this screenshot:



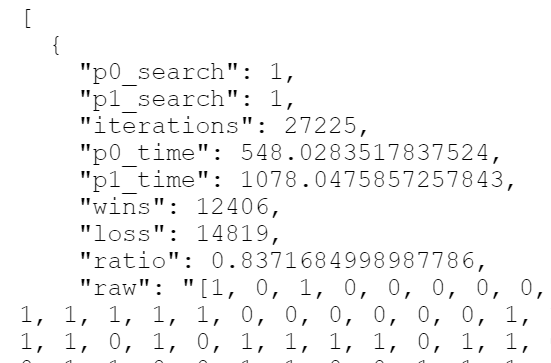
After that, all that was left to do was to generate data to train on and to use that data to train the Neural Net. For evidence of both please check out Generate.py and training.py respectively. Training was a relative easy task to complete but generating data had its issues. The idea was to create a huge MCTS and track the win rates for a statistically representative number of simulations. In my case, I generated data with 1,000,000 simulations and only used game states with a multitude of card options available. This was done due to the fact that the data would have otherwise trend too much towards 0, as unavailable moves had a 0% win rate. When you do not omit these data points, there are too many zeros and the NN just makes every prediction as 0.

**Results**

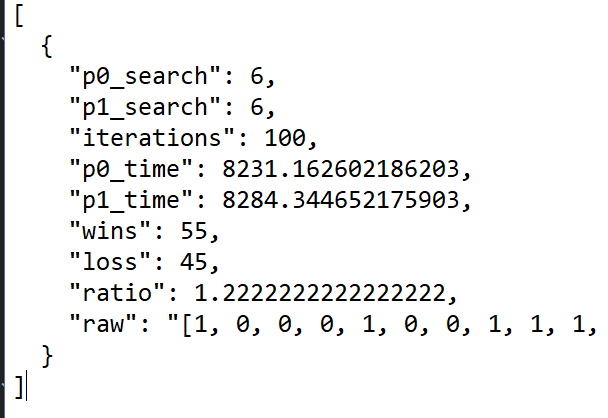
The results showed that both the Monte Carlo Search Tree with and without the Neural Network performed significantly better than random. P0 is the vanilla MCTS, and p1 is just choosing random actions.



According to Andrews analysis, the first player has a disadvantage. Both p0 and p1 are picking random cards, however the second player, p1, wins about 2000 more times than p0.



After Andrew’s analysis, even with the first player disadvantage, it was shown that the Neural Network played slightly better than the vanilla tree. See evidence below:



**Conclusion and Future Work**

It was an interesting experience to dive into a topic with limited knowledge and to attempt to work with it in a freeform setting. We had to organize a project, set learning objectives and adapt to the complications that arose. In the field of computer science, it's important to be adaptable as new technologies are constantly being developed.

Adapting this project to use the full ruleset would be something that could be a very interesting challenge as the omitted cards heavily disrupt the gameflow since those cards could be played at any time. Additionally, expanding the game to be played with more than 2 players could pose an even more interesting problem when in tandem with the full ruleset.