Reinforcement Learning for Exploding Kittens

Radhika Patil RADHIKAP@STANFORD.EDU
Stanford University

Ryan Silva RDSILVA@STANFORD.EDU
Stanford university

Atharva Parulekar ATHARVA@STANFORD.EDU
Stanford university

Abstract

A two player Exploding Kittens game is an interesting environment to explore, learn and play. Neural networks and model-free reinforcement learning algorithms together provides an opportunity to characterize strategic games involving human thinking and intuition by learning optimal policies over a large set of possibilities. We use two player self-play reinforcement learning to learn strategies for two agents playing the Exploding Kittens card game formulated as a Markov Decision Process. We implement a Monte Carlo Tree Search over the states and actions to calculate Q values and feed it to a neural network to train for learning optimal policy to maximize the Q value over the allowed actions.

Keywords: Reinforcement Learning, Neural Networks, Monte Carlo tree Search, Markov Decision Process, Exploding Kittens

1 Introduction

Exploding kittens is a popular 2-6 player strategic card game [2] incorporating the idea of Russian Roulette. While drawing cards from the deck, the mission of the game is to stay in the game and not explode. Similar to Russian Roulette where a participant puts one bullet in the gun cylinder and randomly rotates it before inserting it back in the gun after which he puts the gun to his head and fires with a chance of being dead, this game involves a card called 'Exploding Kitten' which results in the player getting out of the game and players strategize to avoid getting the card.

1.1 Rules of the game

There are 13 different cards – 8 action and 5 non-action, in a regular game each having specific functions. Following is a table of the cards with their quantities and functions [2]

Action cards:

Sr. #	Card	Quantity	Function
1.	Exploding Kitten	1	Explode the player out of the game
2.	Defuse	# Players	Counter the Exploding kitten effect
3.	Attack	5	End your turn without drawing a card and force the next player to take two turns
4.	Nope	5	Stop the action of another player (this card can be played anytime in the game regardless of the turn sequence)
5.	Skip	5	End your turn without drawing a card

©2018 Radhika Patil, Ryan Silva, Atharva Parulekar

6.	Shuffle	5	Shuffle the draw deck
7.	See the future	5	Privately view the top three cards of the deck
8.	Favor	5	Another player gives you a card of his choice from his hand

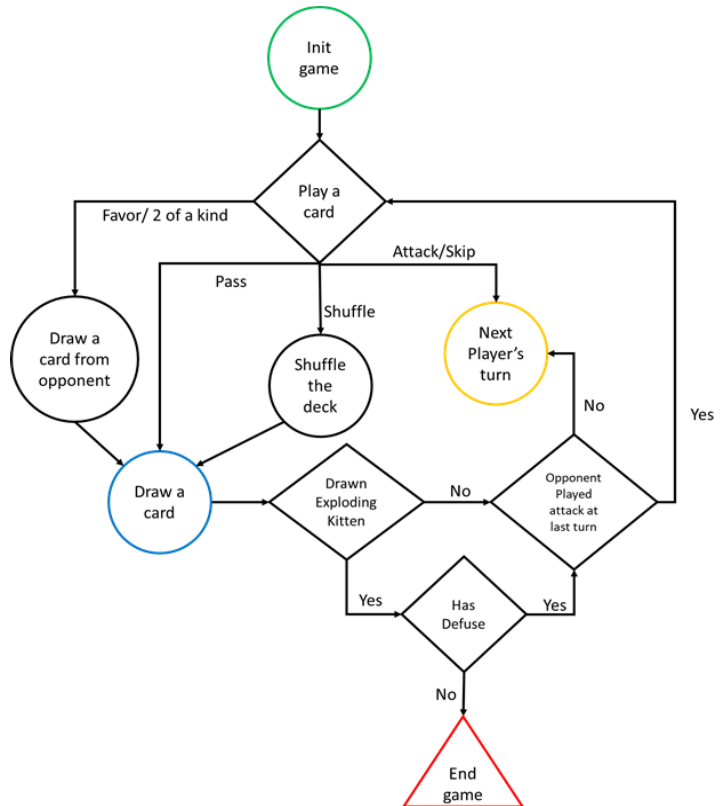
Non-action cards:
In addition to the action cards, there are 5 different kinds of non-action cards (called cat cards) which do not perform any action individually, but they can be coupled together to make up new actions when played together.

Coupling	Effect
Two of a kind	Steal a random card from another player
Three of a kind	Ask another player to give you a card of your choice from his hand if he has it
Five different kinds	Pick a card of choice from the discard pile

In the initial state of the game each player has 5 cards including one defuse, and the remaining cards are shuffled along with the Exploding kitten into the deck.

1.2 Simplification of the rules

We limit the game to a two-player scenario where players alternate the turns. As the game is quite complicated with a lot of possible strategies to manipulate the opponent, we simplify the rules and the environment of the game by including excluding some of the actions – Nope, See the Future, Three of a Kind, Five of a Kind. This leaves 11 cards in our game. The 'two of a kind' action is also simplified by allowing it only on the five cat cards and not the action cards. The flowchart in above figure shows the simplified logic of game play.



1.3 Self-play reinforcement learning

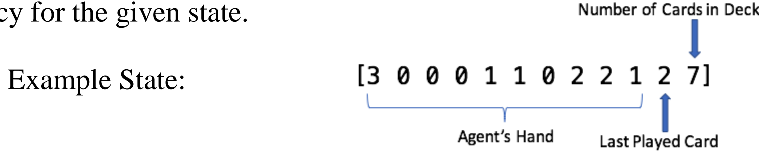
An important component of any reinforcement learning algorithm is the rewards received after the actions. In this game, the agent gets a reward only at the end of the game knowing whether a player exploded or not. Therefore, we require to play multiple games to learn strategies for avoiding the exploding kitten. For this purpose, we use a self-play reinforcement learning library written by David Foster and Applied Data Science Partners [4]. The high-level idea of the self-play algorithm is to play two copies of the same agent against each other, and have them learn the game from experience. This experience is recorded and after some time the agent's global approximator (in this case a deep neural network) is trained using

the experience gathered. The two agents are then evaluated against each other in a tournament, and another iteration is started by duplicating the best agent and playing the two copies against each other again to gain more experience.

.

2 Representation of Exploding kittens

We use a Markov Decision Process to represent the process of playing the game. The estimated Q and U values of this MDP are used to train a neural network to learn a policy for the given state.



2.1 Markov Decision Process

A state in our MDP represents the information that one player has at any point of time during the game. We sequentially number our card types providing a unique ID to each type. The state is represented as a vector of length 11, where first 9 numbers represent the number of cards of each type in the player's hand, the 10th value represents the ID associated with the last played card and the 11th value is the number of cards left in the deck. A rough calculation of this state space shows that it is on the order of 10^8.

An action in our MDP is playing a certain card. All the action cards as well as any two of a kind cat cards in the hand provide an action for the MDP. We also add a 'Null' action, where the player does not play any card. Unless the action is an Attack or Skip card, every action also includes drawing a new card from the deck before passing the turn to the opponent. Since there is no upper limit on the number of cards in the hand, the possible action space is variable at each step of the MDP and can be completely derived from the state.

2.2 Monte Carlo Tree Search and Neural Network

We implement a Monte Carlo Tree Search to find an estimated optimal action for a player at any given state. A node in MCTS represents the state of the player and the edges represent the actions (the card which can be played). Associated with each edge are statistics detailing the current estimated Q(s, a) value, as well as the number of times the edge has been taken. As the action space is variable, at every state the MCTS checks for possible actions it can take from the current state before simulating. While evaluating a state, the agent chooses an action that maximizes the following:

$$Q(s,a) + c\left((1-\varepsilon)P + \varepsilon\nu\sqrt{\frac{N(s)}{1+N(s,a)}}\right)$$

If the action leads to a new state not in the tree, the algorithm evaluates the leaf node by passing the new state through the neural network to get initial Q values for the edges. The algorithm then uses an eligibility trace to propagate the estimated value of the new state through the path it took to the leaf node.

The neural network we are using is a densely connected network with 4 hidden layers. There are two

Algorithm 1 Monte Carlo Tree Search

```
1: function SIMULATE(s)
2:     loop
3:         path = ∅
4:         leaf, r ← MOVETOLEAF(s, path)
5:         r ← EVALUATELEAF(leaf, r)
6:         BACKFILL(s, r, path)
7: function MOVETOLEAF(s, path)
8:     while s ∈ T and ¬ ENDSTATE(s) do
9:         a ← argmax_a(Q(s,a) + c ∗ Q_0(s,a) ∗ √(N(s)/(1 + N(s,a))) ▷ c is the exploration param
10:        path ← path ∪ (s, a)
11:        (s′, r) ← G(s, a)
12:        s ← s′
       return (s, r)
13: function EVALUATELEAF(s, r)
14:    T ← T ∪ s
15:    if ¬ ENDSTATE(s) then
16:        r ← PREDICT(s)    ▷ Use the value head of the NN to predict the value of this state
17:        for a ∈ A(s) do
18:            Q_0(s,a) ← PREDICT(s,a)    ▷ Use the policy head of the NN to predict the initial Q value
19:            W(s,a) ← 0
20:            Q(s,a) ← 0
       return r
21: function BACKFILL(s, r, path)
22:    currentPlayer = PLAYERTURN(s)
23:    for (s,a) ∈ path do
24:        N(s,a) ← N(s,a) + 1
           ▷ Need to account for whether this state was played by the agent or the opponent and adjust the value accordingly
25:        if PLAYERTURN(s) ≠ currentPlayer then
26:            direction = -1
27:        else
28:            direction = 1
29:        W(s,a) ← W(s,a) + r∗direction
30:        Q(s,a) ← W(s,a)/N(s,a)
31:
```

output layers: one objective is to predict the utility of a state, and the other objective is to predict the utility of the actions, or Q(s, a) values. The structure of the neural network is taken from a paper published on training an agent to play the game Go at a superhuman level [1]. We have modified the network to the needs of our state space and hardware constraints.

3 Setting up the game environment

We set up our own game environment by implementing the states and actions as described above in the MDP section. The simulator takes into account the possible actions at any given state. At every point in time, the agent simulates the possible actions using MCTS and chooses the action which locally optimizes the utility before ending the current turn. Each time the agent takes an action during training, it simulates between 50 and 100 moves into the future to estimate the Q values of the current state. Every iteration, there are 8 to 15 full games played between the two agents. The agent trains it's neural network when it has stored between 2500 and 5000 memories. These numbers were saved as hyperparameters to the algorithm, and generally larger numbers means a better agent, at the cost of more training time.

4 Training

Training an agent consists of two phases. First, the agent gathers estimates about the values of states in the game environment and stores these states and estimates as memories. After sufficient time has been spent exploring the state space, the agent then replays its memories by training its neural network on batches of random samples from the memories. For the neural network training we minimize the 'softmax cross entropy loss' to learn a Q value function and 'mean squared error' to learn a function for state values.

The exploration parameter turned out to be an important factor for training an agent to play Exploding Kittens. Since the agent always has the option to not play a card, and sometimes that is the only allowed action for the agent, it is more likely to spend time exploring the value of the 'Null' action. If the exploration parameter in MCTS was not high enough, the memories generated were dominated by samples weighted towards the 'pass' action, which in turn biased the neural network to favor the pass action. This then influenced subsequent Q value estimates in the MCTS algorithm, and vice-versa, until the only action ever picked by the agent was to not play a card. Thus, good exploration of the state-action space is crucial to allow iterations between the agent's RL algorithm and neural network to start converging to correct values.

5 Results

To test an agent, we play it against different baselines. During testing time, and agent always chooses the the best action, which it estimates by running MCTS from the current state, using its trained network to inform the initial Q values of future states.

We compare our best agent learnt so far with a random agent obtained at initialization. Since the neural network uses Xavier initialization, the values given out by the policy head before training starts is a uniform random softmax vector. We have two ways to compare policies. The first one would be comparing the loss of each policy and the second one would be to actually play the policies against each other.

The best policy learned so far although beats other agent versions in matches. The following are the results for the games between a random agent and our best agent.

	Player 1 wins	Player 2 wins
Best vs Random	100	0
Random Vs Random	54	46

Lastly, we draw the tournament head map which describes how each policy fared with other policies during tournaments.

		Opposing Agent Version									
		1	2	3	4	5	6	7	8	9	Loss on test sample
Agent	1		0.4	0.4	0.5	0.5	0.4	0.7	0.5	0.4	1.2784
	2	0.5		0.6	0.5	0.3	0.5	0.4	0.6	0.6	0.9969
	3	0.6	0.4		0.2	0.4	0.6	0.2	0.5	0.5	1.1938
	4	0.5	0.1	0.8		0.3	0.5	0.5	0.4	0.7	1.1349
	5	0.5	0.7	0.6	0.7		0.7	0.7	0.7	0.6	0.9987
	6	0.6	0.5	0.4	0.5	0.3		0.4	0.2	0.4	1.2163
	7	0.3	0.6	0.8	0.5	0.3	0.6		0.5	0.6	1.0916
	8	0.5	0.4	0.5	0.6	0.3	0.8	0.5		0.6	1.0643
	9	0.6	0.4	0.5	0.3	0.4	0.6	0.4	0.4		0.9962

The agents are numbered according to training time spent, so agent 1 is the earliest version of the network whereas agent 9 has trained for the most amount of time. The heat map values correspond to the percentage of games an agent won against an opposing agent in a 10 game match where each agent went first 5 times. On the far right is the loss of the agent's network on a batch of testing data.

Examining row 5, this agent has won or tied the majority of games in all matches. This network also has one of the better loss values among the agents on the testing data, hinting at a very slight correlation that a well trained network performs better on the actual game. However, the rest of the plot is quite noisy- this shows the high amount of variance in the outcome of a game of Exploding Kittens.

6 Discussion

We modelled the game as an MDP where we assume we know the state exactly at any point of time. Another interesting and probably more effective way to model the game would be to use a POMDP. Using a belief state, the player could account for what could be the state of the deck as well as the opponent's hand. A full belief state for this game would be intractable to solve, but perhaps modeling just the belief of the position of the Exploding Kitten could be solvable. An important aspect of the game is keeping track of how many cards of each type were played until the present time. This offers players insight into what could be a the probability of the next card being an exploding kitten and whether one should strategize to force the opponent to draw a card. As a trivial example, if there's only one card left in the deck and both the players are alive, then it implies that the last card is the exploding kitten. We tried to incorporate this but including a definitive variable of 'number of cards in the deck' in our state, but having a belief state over other possible outcomes will improve the strategization learning.

We have excluded some of the actions to decrease the complexity of the game. For example, playing a 'Nope' card to force stop an opponent's action is allowed at any point of time in the game, whether it's the player's turn or not. This action is a very powerful tool to strategize for winning. Similar is the case with other actions that we have removed for simplification and including them will highly improve the strategizing.

As there is a really large number of possible accessible states, we may not encounter most of them in the Monte Carlo Tree Simulations for the players. So, it is very important to be able to predict their utility using the experience if required in future. This makes training a neural network important in this scenario.

Since our neural network has two heads, a value head and a policy head, during each step of the optimizer, we try to propagate gradients through both heads. During training, we observe that although the value loss keeps on getting smaller the policy loss stops decreasing after a certain number of iterations and keeps fluctuating. This behavior may be caused as a result of there being a lot of policies which perform similarly on the game. Since Exploding Kittens is not a deterministic game like Chess or Go it might not have an optimal policy which is dominant in all scenarios and there may be a vast array of policies which are similar. The policy learned by this method although beats other policies, it also has its drawbacks namely the policy head loss. Hence these policies are not guaranteed to win.

7 Work Breakdown

All the team members are taking the class for 3 units. Ryan searched for the self-play library, aided game environment development, helped modify MCTS algorithm to account for uncertainty in environment, tuned hyper parameters of algorithm, trained and tested different agents. Radhika helped with game ideation and formulation, assisted with debugging and training the code and outlined the report. Atharva modified the neural network, trained and tested the network and analyzed the results.

8 References

[1] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T. & Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, 550, 354--.

[2] Exploding Kittens, How to play. (2018, 12 6). Retrieved from Exploding Kittens: https://explodingkittens.com/

[3] Kochenderfer, M. (2015). *Decision Making under Uncertainty.* Cambridge, Massachusetts: The MIT Press.

[4] Foster, D. (2018, January 26). How to build your own AlphaZero AI using Python and Keras [Blog Post]. Retrieved from Medium: https://medium.com/