Poker with Self-Play Deep Recurrent Q-Learning

Lê Gia Khang - 21522189 Lê Duy Khang - 21522188

Abstract

The field of artificial intelligence has seen remarkable advancements in recent years, with deep reinforcement learning proving to be a powerful technique for training intelligent agents. This report explores the application of self-play deep recurrent Q-learning in the domain of poker. Poker, a complex and strategic game, poses unique challenges for AI agents due to its inherent uncertainty and the need to make decisions based on incomplete information.The report discusses the design and implementation of the self-play deep recurrent Q-learning framework, including the representation of the game state, the neural network architecture, and the training process.

1. Introduction

The game of poker has long been regarded as a benchmark for human intelligence due to its complex decision-making, strategic thinking, and inherent uncertainty. In recent years, the field of artificial intelligence has made significant strides in developing intelligent agents capable of playing games at a high level. Deep reinforcement learning, in particular, has emerged as a powerful technique for training agents to make optimal decisions in dynamic and challenging environments. This report explores the application of self-play deep recurrent Q-learning in the domain of poker, specifically focusing on Limit Texas Hold'em.

Poker poses unique challenges for AI agents due to the need to reason under uncertainty and make decisions based on incomplete information. Unlike games such as chess or Go, where the entire state of the game is observable, poker requires players to make strategic choices while accounting for hidden information, such as their own private cards and the intentions of their opponents. This aspect of poker introduces an element of bluffing, risk assessment, and adaptation that makes it an intriguing testbed for AI research.

The objective of this study is to develop an intelligent agent capable of playing Limit Texas Hold'em poker through the use of self-play deep recurrent Q-learning. This approach combines the power of deep reinforcement learning with recurrent neural networks to capture the temporal dependencies and long-term strategies involved in poker. By leveraging historical actions, the agent learns to make informed decisions over multiple betting rounds, taking into account its own hidden cards, the community cards, and the actions of other players.

The report discusses the design and implementation of the self-play deep recurrent Q-learning framework, detailing the representation of the game state, the architecture of the neural network, and the training process. We also developed a baseline for our deep learning method in this two-player imperfect information game by trying to learn a Q-network for policy in Poker.

To evaluate the performance of the developed agent, we conduct experiments on Random Agent of RLCard. The results demonstrate the agent's ability to adapt its strategy, learn from opponents, and achieve competitive performance in the challenging domain of Limit Texas Hold'em poker.

2. Related Work

2.1 Playing Atari with Deep Reinforcement Learning

In this paper, DeepMind formulates Deep Q-Networks to play Atari games. DQN is a neural approximation of Q-learning in which a network is trained to represent a (state, action) to value function. The agent is trained using a memory buffer to avoid catastrophic forgetting, which occurs when an agent starts to forget what it learned from early transitions in the training period. These methods yielded state-of-the-art results on 6 of the 7 single player Atari games it was played on.

2.2 Deep Recurrent Q-Learning for Partially Observable MDPs

In this paper, they experiments with the use recurrent Q-networks in agents in order to integrate information across timesteps. In the Atari games played by the agent, states are fed in as flickering screens at discrete timesteps. The agent is able to successfully learn to integrate information across timesteps in order to learn sequentially dependent patterns such as velocity and direction of objects moving in the game. We seek to implement similar models to allow our agents to make sequentially dependent valuations of their own hand based on the order in which the community cards are revealed in poker.

3. Our Work

3.1 State Representation, Action Space and Rules.

We highly recommend you to read Poker's rules first, especially **Limit Hold-em** which is the version we used in this report, in order to understand the rest of our report.

Winnings or Rewards are measure in terms of big blinds, which is the amount of money one has to blindly put into the game prior to seeing their hand to incentivize betting. The reward is calculated based on big blinds per hand. For example, a reward of 0.5 (-0.5) means that the player wins (loses) 0.5 times of the amount of big blind. In Limit Hold-Em poker, the betting in each round is limited to four big blinds, which is represented by the five indices of the state vector corresponding to each betting round.

For this project, we rely extensively on RLCard, a toolkit that has environments for several highly popular card games. It also has a convenient infrastructure for training and testing models. We referenced the RLCard example agents when writing our own for the infrastructure.

The state representation of Limit Hold-Em poker in the RLCard toolkit consists of a size-72 vector representing the cards and bets that have been placed so far, as well as a list of actions which can be taken from the current state. The first 52 elements of the state vector represent the cards: a 1 is placed in the slots of cards that are either in our hand or on the board as a community card. The next 20 indices represent the betting amounts over the course of the hand, where 5 indices are allocated for each betting round.

There are four possible actions to take in the game, encoded as follows:

- 0. Call (accepting another player's previous bet)
- 1. Raise (raise the amount of the bet on this round)
- 2. Fold (give up the hand)
- 3. Check (this is a "pass" when no other player has placed a bet)

Note that there are states from which we are not able to take certain actions (e.g., we cannot check when a previous player has bet).

3.2 Deep Recurrent Q Network.

In this project, we present recurrent formulations of popular reinforcement learning models to play poker in order induce the bias that card order is important in the game of poker. This will hopefully allow our models to integrate information across states from different timesteps in their decision making.

In order to induce our bias of poker having sequentially dependent (in other words, time-dependent) gameplay, we implemented a recurrent approximation of Q-learning so that the agent is able to integrate information across different timesteps in a given hand. In particular, we believed that the agent may be able to better model state action values when integrating the order in which bets are placed and the order in which community cards are revealed. For example, having when the opponent bets big on when a certain community card is revealed, the model may be able to leverage this information when making a move.

To isolate the effects of recurrency, we modified the architecture of traditional DQN. The DRQN consists of an LSTM with hidden layer of size 150 to encode historical moves and current state, then we feed forward to a MLP layer of size [128,256]. A linear layer outputs a Q-Value for each action.

Updating a recurrent network requires each backward pass to contain many time-steps of game states and target values. Additionally, the LSTM's initial hidden state may either be zeroed or carried forward from its previous values. We consider **Bootstrapped Sequential Update**:

Episodes are selected randomly from the replay memory and updates begin at the beginning of the episode and proceed forward through time to the conclusion of the episode. The targets at each timestep are generated from the target Q-network, Q^. The RNN's hidden state is carried forward throughout the episode.
Sequential updates have the advantage of carrying the LSTM's hidden state forward from the beginning of the episode. However, by sampling experiences sequentially for a full episode, they violate DQN's random sampling policy.

In order to train the model on a series of transitions, we altered the prior Q learning algorithm to train from batches of episodes as is described below. We train our agent against it self, that is why we call it Self-Play Deep Recurrent Q-Learning.

Algorithm: Deep Recurrent Q-Learning

Initialize a circular memory buffer *M*
Initialize a learner recurrent value network *L*
Initialize a target recurrent value network *T*
for episode 1...n **do**
    Reset hidden of *T* and *L*
    Get initial state s_1 of hand
    Initialize list for sequence of transitions *A*
    while state st is not terminal **do**
        Act according to *T* with probability , otherwise
        choose random action. Let at be the selected action.
        Take action at in the environment and observe reward r, next state s_{t+1} after other player takes their action as well.
        Store transition $\{s_t; a_t; r_t; s_{t+1}\}$ in *A*
    end while
    Store sequence of transitions *A* in *M*
    Sample episodes *E* from *M*
    Reset *T* and *L* hidden
    for $\{s_j; a_j; r_j; s_{j+1}\}$ in *E* **do**
        $q = max_{a'} T(s_{t+1}; a')$
    if s_{j+1} is terminal **then**
        $y_j = r_j$
    else
        $y_j = r_j + \gamma q$
    end if
end for
Input sequence of states to *L* to get output at each
timestep of *E*
Update *L* using MSE Loss on the computed labels

from T
Periodically update $T \leftarrow L$

3.3 Details about training with self play.

And now that we have an agent that can learn how to play poker, how do we teach it to play poker? and obviously, we cannot teach it by ourself!

One way is to let it play against a known-to-play-well agent (e.g. a pretrained DQN agent,...) and hope that it can infer a great and general playing strategy from its plays against that target agent. However, this method has a downside: our agent can only be as good as (or a little bit better than) the target agent or most of the time only learn to play well against that specific agent. And the only way to fix this is to increase both the quantity (types) and quality of the target agents… And, you can see how this cannot be done indefinitely.

But, what if we consider that our agent is the best agent (a bit pretentious… but it's fine). Then we can just let it play against another instance of itself and have both of them learn from each other's moves and hope that it, the best agent we have, can teach itself how to play better. And, like before, this arrangement has a problem: it is extremely insufficient both in performance and the quality of learning. First of all, to train both our agent and the target agent means that we have to train 2 agents at the same time which is going to double our GPU renting bill. And secondly, in a hypothetical situation where agent A and agent B (the target agent) are set in the above environment, if agent B learns a good move then obviously agent A will learn to counteract it and then agent B will stop using that move which is rational. But the problem is what A learns from this interaction as the only thing it learns is to counteract B's move but not the move itself. What happens here is that the learning is divided between two agents and the best agent we can have from this process is only at half of its true potential. If only there is a way to merge two agents into the best possible agent…

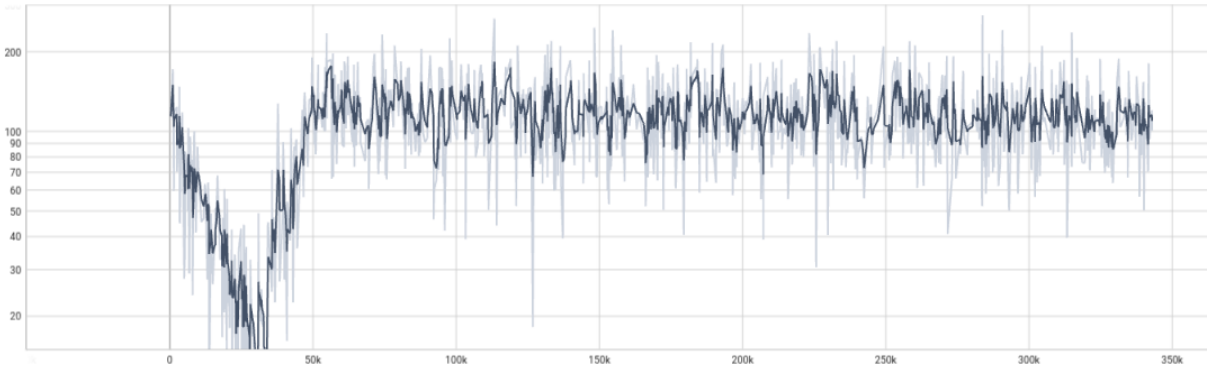
And, I couldn't think of any method that can achieve that and settle with a simple solution: only let one agent learn and periodically clone that agent to the target agent. And in theory, this modification should fix both of the above problems as now, the training is only done on one agent and secondly, to continue with the original hypothetical situation, when A learns a new move, so does B, hence, it is rational for A to learn how to counteract that move. You can see how our agent is forced to learn both of the skills instead of only one like before.

3.4 Details about the training environment and settings.
The environment is initialized with the following parameter (as none of us are experienced in picking parameters, it is very likely that the following values is not the optimal one):

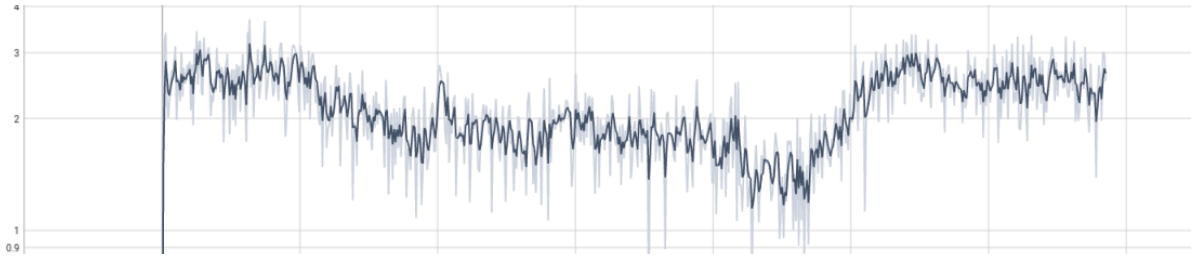
name	value
target update frequency	700
epsilon	[0, 1]
epsilon decay steps	20000
learning rate	0.0001
relay memory size	100000
batch_size	32
mlp layers	[256, 512]
lstm hidden size	128
clone agent parameters to the target agent	every 200 step

3.5 Result.

The agent was trained on a g2-standard-8 (8 vCPU, 1x L4) instance on google compute platform for a duration of 6 hours. The performance of our agent over the training period is measured using its performance against a random agent as our agent does not learn from a random agent for the whole training period.



Loss every 50 update



Reward vs. random agent

We can see how, despite never interacting with the random agent, our agent achieved an outstanding win rate and rewards against the random agent.

On further testing with the output agent, we found that when playing against a random agent, our agent manages to win 96% (9578 out of 10000 plays) of the time. When matching against a Rule-based model (from rlcard), it still manages to win 92% (9173 out of 10000 plays) of the time. However, this result is quite questionable as when we match the random agent against the Rule-based model, the random agent was able to win 65% of the time.

4. Conclusion

In conclusion, the approach of "Poker with Self-Play Deep Recurrent Q-learning" presents a novel and promising technique for training intelligent agents to play poker. By combining deep reinforcement learning with recurrent neural networks and self-play, this approach demonstrates significant advancements in autonomous decision-making within the complex and strategic domain of poker.