Poker Project - Team 47: Training a Counterfactual Regret Minimisation (CFR) Agent using a Deep-Q Network (DQN)

AUYOK Sean, CAI Jiaxiu, CHIK Cheng Yao, Joyce YEO Shuhui, ZHUANG Yihui
National University of Singapore
sean.auyok,e0201975,chikchengyao,joyceyeo,yihui@u.nus.edu

1 Introduction

In the quest to understand the human mind, scientists have delved into designing Artificial Intelligence (AI) agents in more structured environments like games. AI agents have long established their dominance in games, perhaps most famously in chess with IBM DeepBlue's win over world champion Garry Kasparov in 1996. The capabilities of AI extend to imperfect information games, where AI agents have beaten top human experts.

An interesting example would be Texas Hold'em Poker, a complex game including elements of chance and bluff. More interestingly, it is not enough to merely win the opponent, but also to value-bet and win by as large a margin as possible. This paper focuses on our team's development of a poker AI for Heads-Up Limit Texas Hold'em played under strict time controls.

When designing our agent, the main functionality was for the agent to recognise strong hands to invest in. Given the limited time for our poker agent to decide on the next action, we employed various optimisation strategies to reduce the game complexity. For instance, we taught our agent to cluster poker hands together (Section 2), and play them with similar strategies.

Upon evaluating the strength of a hand, our agent has to make the decision to fold, call or raise. We trained our agent to make such decisions using Counterfactual Regret Minimisation (CFR), which aims to play an approximate Nash Equilibrium strategy against all opponents. Our agent traverses the game tree based on the betting sequence observed and the strength of its hand. At each decision node, our agent takes each action with a certain probability, depending on how much it regrets not having taken the action (Section 3).

Unfortunately, training a CFR agent is an extremely inefficient process. The CFR agent's limited implementation also makes it slow in making real-time decisions during games. We discuss these constraints (Section 3.4) and how we circumvented them using a Deep-Q Network (DQN) (Section 4). The DQN is a neural network optimised for games like poker, where we want to evaluate the value of making different action. We trained our DQN against our loosely-trained CFR agent.

We conclude by reflecting on the process of designing and training poker agents, and how we could have better approached the training process.

2 Modelling Game State

Poker has a plethora of game states an AI agent can tap on, such as the hand strength, pot size and player stack. Given the complexity of poker, it would be unfeasible to use all the game states in designing our agent. The primary heuristic in a rational poker agent would be the hand strength, because that is how poker's end state is evaluated - the player with the higher hand strength clears the pot. In this section, we focus on how we abstracted hand strengths for our agent.

2.1 Hand Strength

There is extensive research on methods to evaluate hand strength, such as CactusKev, TwoPlusTwo and 7-card. These evaluators aim to assign numerical values to hands, and subsequently group cards with similar strengths together. These buckets of cards tend to have similar play styles.

Since hand strengths change across streets as community cards are revealed, the classification key is typically a function of the current hand strength, and the potential hand strength. We use the expected win rate of a hand, which takes into account different combinations of the community cards and opponent hands given a set of hole cards.

Card Isomorphism

To effectively evaluate hand strengths, it is imperative to simplify the hands using abstraction techniques like card isomorphism. Card isomorphism exploits the fact that a card's suit has no inherent value. It is only useful to know the values of the hole cards and whether they are suited (have the same suit). For example, A♣ A♣ has the exact same winning chance as A♠ A♠, and these hole cards would be played with identical strategies.

Card isomorphism can reduce the number of game states in the pre-flop street from $\binom{52}{2} = 1326$ sets of hole cards to 169 states (Appendix A). Each state is identified using the value of the cards, and a boolean indicator if they are the suited.

Percentile Bucketing

However, card isomorphism does not simplify the game complexity enough. The 169 game states in the pre-flop state will exponentially increase as we introduce player actions and more community cards.

To further reduce the branching factor of our game tree, we employ a second layer of abstraction, card bucketing, to

group cards of similar strengths together. For instance, two sets of hole cards like K♡ Q♢ and Q♢ J♡ can be handled with a similar strategy as they are offsuit highcards with a possibility of completing a straight.

Ganzfried and Sandholm, the team behind a world-class poker AI (Tartanian), found that using too many buckets meant that the training algorithm could not converge sufficiently fast. Eventually, they settled on a $8 - 12 - 4 - 4$ bucket size for the four different streets [Ganzfried and Sandholm, 2012]. We simplified their model and chose to use the same bucket size of 5 in all streets for our agent. The branching factor is large enough for our agent to differentiate hands, but not too large that any training would be intractable.

Our heuristic for bucketing is the expected win rate of a hand. Each bucket is effectively identified by a minimum and maximum expected win rate. When community cards are generated, our agent estimates the win rate of the hand using a Monte-Carlo simulation, then assigns the hand into one of the five buckets. Unfortunately, the number of hand combinations is very large. Instead of calculating the win rates for every hand, we only calculated the win rates of 1000 hands to estimate the win rates needed to demarcate the buckets.

In each street (pre-flop, flop, turn, river), 1000 different hands were generated. The hands would have two pairs of hole cards and 0, 3, 4 or 5 community cards depending on the street. For each of these 1000 hands, 500 Monte-Carlo simulations were run to determine the win rate of the hands. We used a percentile-win-rate clustering method, so the first bucket is marked by the win rate of the hand at the 20^{th} percentile. We stored the bucket margins for our CFR agent across different streets (Appendix B). Each margin demarcates the two adjacent buckets, so there are 4 margins given 5 buckets.

Our results corroborate with the effect of imperfect information in hand evaluation. In the pre-flop stage, most cards had a fairly similar win rate. As the uncertainty in the community cards was resolved, the only unknown factor was the opponent's hand. Our agent would then be better able to differentiate the expected win rates of different hands.

3 Counterfactual Regret Minimisation (CFR) Agent

Our agent is primarily trained using the Counterfactual Regret Minimisation (CFR) Algorithm, which is an extended version of Regret Matching. Regret Matching is a modern technique developed, theoretically achieving an approximate Nash Equilibrium play for poker [Martin Zinkevich and Piccione, 2007].

3.1 Regret Matching

Regret is a concept of reward relative to an action, quantifying how much the agent retrospectively wants to take the action given the past observations. Regret is then proportional to the loss observed, when the agent did not play the action. Effectively, the agent favours actions that score the highest in the regret heuristic. Every action x has a regret value that can be formalised:

$$Regret(x) = ActionUtility(x) - CurrentUtility()$$

The regret of an action is how much more the agent expects to win by always playing the action (ActionUtility) vis-a-vis the amount the agent expects to win with its current strategy (CurrentUtility). An action with positive regret can increase the agent's utility.

The strategy changes based on the updated regrets of the actions. The regret-based agent engages in reinforcement learning to update its beliefs about each action's regret. Later on, we discuss the convergence of this iterative self-play towards an approximate Nash Equilibrium.

The agent plays the actions with a probability proportional to their positive regrets. Our agent will only take actions that have a positive regret. Actions with negative regret are not played, as they theoretical decrease the agent's expected utility. Our agent generates a 3-tuple for the optimal probability of playing each action (fold, call, raise). For example, one of the tuples generated could be $(0.2, 0.3, 0.5)$.

CFR - Extending Regret Matching

By its nature, poker has a large game tree with chance nodes in the generation of community cards, making it problematic to accurately define the utility at each terminal node. The CFR technique extends Regret Matching to games where multiple actions are sequentially taken, and there are chance nodes between these actions. The agent is counterfactual as it takes into account the chance of each decision being made, and the expected utility from that node.

Poker's game tree is extremely extensive (Appendix C). At each chance node, community cards are generated and the CFR agent assigns the hand into one of 5 buckets, so the tree branches by a factor of 5. Without bucketing, the flop chance node would have a very large branching factor of $\binom{48}{3} = 17296$ combinations of community cards, which is not feasible to work with. Clearly, our bucketing strategy was key in controlling the size of our CFR agent's game tree.

3.2 Randomising Strategy

The exploitability of opponent behaviour is symmetric, so our agent has to acknowledge that the opponent can similarly extract patterns from our play style. It would then be beneficial to mask our agent's biases.

Some researchers employ purification techniques to overcome abstraction coarseness. For instance, the 0.1 chance of folding in $(0.1, 0.3, 0.6)$ could be due to insufficient play rounds that the regret of folding has not been minimised. Purified poker agents prefer the higher-probability actions and ignore actions that are unlikely to be played. In particular, Ganzfried and Sandholm found full purification to be the most effective [Ganzfried and Sandholm, 2012]. The full purification technique lets the agent play the most-probable action with probability 1.

However, we argue against purification because noise in our player behaviour can disrupt the opponent's attempt in identifying patterns in our play style. Yakovenko et al's experiments found that when human players were pitched against against fixed-style poker agents, the human players could recognise repeated mistakes made by the agent. After 100 of 500 hands, the human players exploited the patterns in the AI and boosted their win rates [Nikolai Yakovenko and

Fan, 2015]. In particular, the randomness in our CFR play supports our deceit techniques of bluffing and slow-playing. Bluffing, or raising with weak cards, leads our opponent into over-estimating our card strength and possibly folding. Slow-playing, or calling even when we have very strong cards, leads our opponent into under-estimating our card strength. The opponent stays in the game longer, increasing the eventual pot size.

3.3 Approximate Nash Equilibrium

Bowling et al, the pioneers of Regret Matching, assert the convergence of CFR agents towards an approximate Nash Equilibrium [Martin Zinkevich and Piccione, 2007]. In particular, we expect the regret values over the entire tree to be minimised. Intuitively, the minimisation of regret occurs because regret is undesirable.

Our CFR agent has a higher probability of playing actions with high regret. By definition, actions with high regret will lead to an increased expected reward (CurrentUtility). The added benefit (ActionUtility - CurrentUtility) of playing the action is then reduced, in turn minimising the regret. Here we observe that the minimisation of regret is also the maximisation of CurrentUtility. Eventually, the CFR agent converges to an equilibrium where its utility is fairly constant against all other agents.

Training our CFR agent

In training our CFR agent, we initialised all action nodes with zero regret, so our agent starts by playing each action with equal probability. We let two copies of our CFR agent play against itself over 4000 rounds. The two CFR agents share the same game tree, and collaboratively update the regret values at each node. Sharing a game tree can significantly reduce the number of rounds we need to find the optimal regret values. In each round, both CFR agents explore different paths in the tree as the betting sequences are asymmetric to both of them.

We let our CFR agent play against HonestPlayer for 100 games of 500 rounds, and we beat HonestPlayer $(1459140 : 540840)$.

Unfortunately, even after 4000 rounds, the two CFR agents were unable to explore the entire game tree. Some action nodes were still at the initialised zero regret values. Evidently, more training could have been done to overcome this limitation. Nevertheless, given a sufficiently large number of training cycles, the CFR agent is expected to optimise the regret values of all the nodes.

Unexploitability of CFR Agent

The basis of CFR agents is that they are unexploitable in the long term, regardless of the strategy played by the opponent [Johanson, 2007]. The CFR agent has already taken into consideration different opponent play styles (Appendix D) since the training stage minimised the regret across all betting sequences. In fact, part of the unexploitability of CFR agents is their probabilistic play (Section 3.2). Even if an opponent knows the strategy of our CFR agent, they are not aware which path down the game tree we took because of the dice rolls creating probabilistic actions.

While an agent following an optimal strategy may not win the most amount of money from their opponents, they also cannot be exploited in the long run. In theory, a fully-trained CFR agent playing against itself would expect to end the game in a tie, averaging over all hand combinations and betting sequences.

3.4 Limitations of CFR Agent

In practice, it takes nearly 10^8 iterations [Johanson, 2007] of training for the model to converge. Moreover, we found that the extensive form game tree using a bucket size of 5 (Appendix C) takes up around 5 GB of RAM during training. These constraints make the CFR agent computationally expensive to train.

We managed to train our CFR agents for 4000 iterations. Every 200 iterations, we tested its performance against HonestPlayer over 10 games of 1000 rounds each. However, there did not appear to be a stable increase in performance. The 4000 iterations only covered a small portion of information sets in the game tree. Even the traversed portions did not have minimised regret values. Nevertheless, we did observe an overall slight performance boost in our lightly-trained CFR agent (Figure 1), though the flux is too high to make a definitive conclusion.

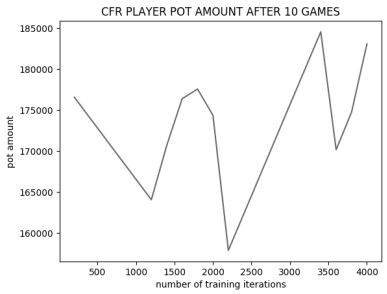

Figure 1: CFR performance every 200 training iterations

Besides the limitation in computational resources required for training, the CFR agent also does not fulfill the in-game time and space constraints given. Given our game tree, we would need a strategy file of 100MB (without compression), far exceeding the 50MB requirement given initially.

Furthermore, when community cards are generated at each chance node, the CFR agent runs a Monte-Carlo simulation to assign the hand a bucket. Running a reasonable simulation size of 1000 (as used by HonestPlayer) would exceed the 100ms limit, and reducing the simulation size would compromise on the accuracy. While an alternative was to store precomputed data in a look-up file, we could not do this because of the file size constraints.

4 Training with a Deep-Q Network (DQN)

Given the practical limitations we experienced with our CFR agent, we sought alternative methods that could improve the training and decision-making efficiency. We wanted to retain the approximate Nash Equilibrium play of our CFR agent, so we kept to an action-based paradigm and used Q-values.

Q-values map actions to their expected reward, similar to how our CFR agent maps actions to their regret value. How-

ever, using reinforcement learning to iteratively learn the Q-values introduces high flux. When our agent updates the value of an action, it also uses the new value in learning about subsequent actions. It would be difficult for the constantly-changing Q-values to converge.

Deep-Q Algorithm

We let our agent learn the Q-values of the actions using a Deep-Q Network (DQN), which aims to make training more stable. We believe that stabilising the training will let our agent converge to an approximate Nash Equilibrium play much faster than a CFR training. After all, a very slow convergence rate was the reason we turned away from the CFR training. The DQN has two key traits:

- Use of two neural networks. The primary training network is constantly updated in every training iteration. DQN introduces an additional target network which occasionally synchronises with the training network. Since the Q-values in the target network are only updated occasionally, they are stable enough to be used for training.
- Experience Replay. A large database (size 5000) of observed data is stored in a replay buffer. Training samples are randomly drawn from this database, mimicking supervised learning with more stable training data sets.

4.1 Training Methodology

As we have mentioned earlier, poker has a multitude of game states that can be used in training our agent. The more game states used, the more parameters in the DQN and we expect the training time to increase.

To minimise our training time, we initially extracted only key features to feed into the DQN: the hole cards and community cards, pot size, player stack and opponent stack. However, we found that after 3000 training cycles, the resulting DQN could not outperform RandomPlayer and HonestPlayer a third of the time.

We then attempted to feed in all the state features that could be retrieved into the DQN (Appendix E). After the same amount of training (3000 cycles), the DQN agent could beat RandomPlayer and HonestPlayer (Appendix F). We extended the training to 5000 cycles, and we observed the DQN agent bankrupting both players.

Given the DQN with all features extracted, we tuned the hyperparameters (Appendix G). We experimented by letting the DQN agents play against RandomPlayer and HonestPlayer as a baseline score. Agents that beat both baseline players would be pitched against each other over 10 games. Similar to the practical limitations in training our CFR agent, we did not have sufficient time to fully experiment with different combinations of the hyperparameters.

4.2 Reinforcement Learning Process

Once we have defined the parameters, we began the formal training for our DQN agent. The DQN agent plays against an opponent, and learns from the opponent. We divide the training into 3 phases, training against RandomPlayer, HonestPlayer and our CFR agent in that order, with the aim of letting the DQN agent learn from these opponents.

A basic AI agent should win against the RandomPlayer and HonestPlayer, which are provided in the engine. Our agent played 5000 games of 1000 rounds each with both players, which took about 1 day to complete. Training against RandomPlayer teaches to make bets on strong hands and fold on weaker ones. HonestPlayer is the rational version of RandomPlayer, keeping strong hands and folding weaker ones. Training against HonestPlayer lets our agent recognise that the opponents are also rational. The DQN agent learns that the opponent will also pick the best action for themselves at each decison node. The trained DQN agent is very competitive against RandomPlayer and HonestPlayer, consistently sweeping their stacks within 500 rounds.

Before we trained the DQN agent against our CFR agent, we extensively compared their performance. After 100 games of 500, the DQN agent defeated the poorly-trained CFR agent $(1704060 : 295280)$. This poor performance of our CFR agent reflects its lack of training. We then trained this DQN agent against our CFR player to produce our final agent.

5 Conclusion

Theoretically, CFR agents play an approximate Nash Equilibrium strategy that is unexploitable. However, due to limitations in the training process, our CFR agent was not able to converge to near to a Nash Equilibrium strategy. After we realised that a DQN agent was more suitable in the training process given these constraints, we decided to change our poker agent to be a DQN agent. However, our work with the CFR agent did not go to waste. Since the CFR agent outperforms RandomPlayer and HonestPlayer, the CFR agent is a good player for the DQN agent to play against.

In hindsight, our group did not expect the CFR agent to be faced with such training limitations. We initially believed that with card isomorphism and good bucketing, we would be able to optimise the training for the CFR agent. Needless to say, we *regretted* using the CFR agent.

After learning more about DQN agents, we realised that DQN agents could not only model the games better, but also be trained faster. DQN agents are also able to more finely differentiate the hands as no card bucketing is involved. The neural network is able to represent the hands and betting actions efficiently in matrices. Furthermore, based on our testing, a DQN could be trained to a decent level with a fewer number of rounds.

We realised that we should have *folded* our efforts in CFR early and instead went *all in* on DQN. That way, we could have spent more time tuning our agent's hyperparameters, and putting our agent through more rounds of training. Having more rounds of training would give us the space to experiment playing against a better-trained CFR agent, and even engaging in iterative self-play. Nevertheless, our group had a good *learning* experience implementing both agents and observing the effects of their play.

References

[Ganzfried and Sandholm, 2012] Sam Ganzfried and Tuomas Sandholm. Tartanian5: A heads-up no-limit texas hold'em poker-playing program. *Association for the Advancement of Artificial Intelligence*, 2012.

[Johanson, 2007] Michael Bradley Johanson. Robust strategies and counter-strategies: Building a champion level computer poker player. *University of Alberta Library*, 2007.

[Martin Zinkevich and Piccione, 2007] Michael Bowling Martin Zinkevich, Michael Johanson and Carmelo Piccione. Regret minimization in games with incomplete information. *Advances in Neural Information Processing Systems 20*, 2007.

[Nikolai Yakovenko and Fan, 2015] Colin Raffel Nikolai Yakovenko, Liangliang Cao and James Fan. Poker-cnn: A pattern learning strategy for making draws and bets in poker games. *Association for the Advancement of Artificial Intelligence*, September 2015.

A States in Pre-Flop Street

AAp	AKs	AQs	AJs	ATs	A9s	A8s	A7s	A6s	A5s	A4s	A3s	A2s
KAo	KKp	KQs	KJs	KTs	K9s	K8s	K7s	K6s	K5s	K4s	K3s	K2s
QAo	QKo	QQp	QJs	QTs	Q9s	Q8s	Q7s	Q6s	Q5s	Q4s	Q3s	Q2s
JAo	JKo	JQo	JJp	JTs	J9s	J8s	J7s	J6s	J5s	J4s	J3s	J2s
TAo	TKo	TQo	TJo	TTp	T9s	T8s	T7s	T6s	T5s	T4s	T3s	T2s
9Ao	9Ko	9Qo	9Jo	9To	99p	98s	97s	96s	95s	94s	93s	92s
8Ao	8Ko	8Qo	8Jo	8To	89o	88p	87s	86s	85s	84s	83s	82s
7Ao	7Ko	7Qo	7Jo	7To	79o	78o	77p	76s	75s	74s	73s	72s
6Ao	6Ko	6Qo	6Jo	6To	69o	68o	67o	66p	65s	64s	63s	62s
5Ao	5Ko	5Qo	5Jo	5To	59o	58o	57o	56o	55p	54s	53s	52s
4Ao	4Ko	4Qo	4Jo	4To	49o	48o	47o	46o	45o	44p	43s	42s
3Ao	3Ko	3Qo	3Jo	3To	39o	38o	37o	36o	35o	34o	33p	32s
2Ao	2Ko	2Qo	2Jo	2To	29o	28o	27o	26o	25o	24o	23o	22p

Figure 2: 169 pre-flop states with Card Isomorphism

The figure above shows the 169 pre-flop states when card isomorphism is applied. p indicates a pair, s indicates a set of suited hole cards and o indicates a set of offsuit holecards.

B Percentile Bucketing Results

Street	b1-b2	b2-b3	b3-b4	b4-b5
Pre-flop	0.406	0.482	0.534	0.588
Flop	0.308	0.43	0.54	0.684
Turn	0.262	0.412	0.562	0.736
River	0.198	0.416	0.626	0.832

Table 1: Win rates used to demarcate bucket margins

In the pre-flop street, hands with a win rate of ≤ 0.406 would be assigned to bucket 1. Hands with a win rate > 0.406 and ≤ 0.482 would be assigned to bucket 2. In the river, hands with a win rate ≤ 0.198 would be assigned to bucket 1.

In the pre-flop stage, the bucket margins are close and the difference in win rates of the first and last margin was only 0.182. Most cards in the pre-flop stage had a fairly similar win rate because of the uncertainty in the community cards. This difference increased to 0.634 in the river, when all the community cards have been revealed. The only uncertainty that remains is the opponent's hands.

C Extensiveness of Game Tree for CFR Agent

This section attempts to calculate just how big the game tree for a CFR agent with 5 buckets across each street would be.

At each chance node, we bucket the cards into one of 5 branches, so the branching factor is 5. With four streets (pre-flop, flop, turn, river), there are four chance nodes. In total, there would be $5^4 = 625$ subtrees just from the bucketing alone.

Between the chance nodes there are action nodes for players to alternately make actions. There are at most 15 layers of actions across a round, 7 for calling (1 in pre-flop and 2 from the other streets) and 8 for raising (4 from each player). A loose upper-bound would lead us to assign an approximate branching factor of 2 to these action nodes, for calling and raising. We ignore folding when considering the branching of the game tree, as folding leads to a terminal node. A loose estimation of the upper-bound of the number of action nodes would be $2^{15} = 32768$ nodes.

In practice, our CFR agent generated $6.45 * 10^9$ game states, including the effect of chance nodes, action nodes and predictions on the strength of the opponent's hold cards.

D Characterising Player Strategies

We aim to let our CFR agent approximate a Nash Equilibrium solution in the long term, regardless of the opponent's strategy. However, we first have to consider what a strategy is and how we can formalise strategies. A simple way to characterise player strategies is to consider betting behaviour when the player has weak hands and strong hands.

When a player receives a weak hand, they could continue playing or choose to fold. This behaviour can be defined on a tightness-looseness scale. A tight player only plays a small percentage of their hands, while a loose player would choose to take risks and make bets based on the potential of the hand. Loose play is called bluffing, and deceives the opponent into over-estimating the agent's hand strength and folding. Theoretically, a tight play aims to reduce losses in the case of weak hands. However, a very tight play would also mean that the player keeps folding and the chances to observe the opponent's behaviour is diminished.

On the contrary, when a player receives a strong hand, they could call or raise their bets. This behaviour can be defined on a passiveness-aggressiveness scale. A passive player keeps their bets low and stable, often calling. On the other hand, an aggressive player actively makes raises to increase the game stakes. Passive play, or slow-playing, is another form of deceit which leads opponents to under-estimate the hand strength, thereby continuing to place bets and raise the pot amount. An aggressive play style hopes to maximise the winnings when the hands are strong. However, an over-aggressive play will also encourage opponents to fold, reducing the overall winnings.

The interaction with the opponent comes from the betting actions and the showdown. However, not every game ends in a showdown so using card information from the showdown may not be as effective. Hence, our heuristics for player tightness and aggressiveness is derived from the betting actions of folding and raising.

Our agent incorporates these heuristics into its modelling of the opponent by saving the opponent's action history each round. The betting sequence is a meaningful heuristic as it encapsulates when players make certain decisions (e.g. call then raise vs. raise then call), while being short enough to track.

E Features Extracted for DQN

```
# Extracted game features for DQN
street = round_state['street']
hole_card_probability = self.card_pairs_prob[(hole_card[0], hole_card[1])]
community_cards = round_state['community_card']
pot_size = round_state['pot']['main']['amount']
player_stack =
        [s['stack'] for s in round_state['seats'] if s['uuid'] == self.uuid][0]
opponent_stack =
        [s['stack'] for s in round_state['seats'] if s['uuid'] != self.uuid]
dealer_btn = round_state['dealer_btn']
small_blind_pos = round_state['small_blind_pos']
big_blind_pos = round_state['big_blind_pos']
next_player = round_state['next_player']
round_count = round_state['round_count']
```

Figure 3: Game state features extracted for DQN

The image above shows the final features used in our DQN agent. Some states do not seem to have any real effect on performance, such as the dealer_btn. Nevertheless, we did observe an improved performance including these states.

F Training DQN Agent with RandomPlayer and HonestPlayer

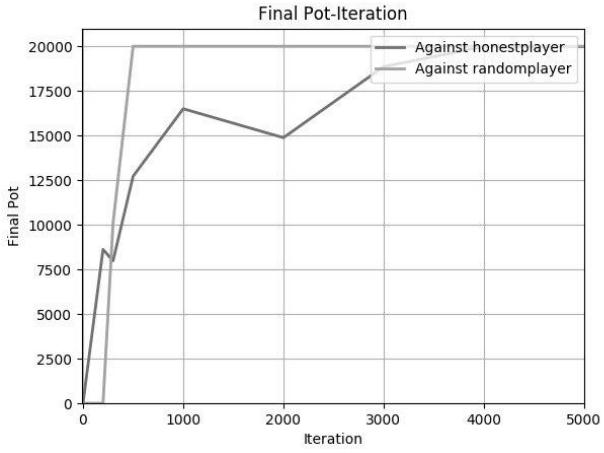

Figure 4: Performance of DQN Agent with training

At about 3000 training cycles, we could conclusively posit that the DQN was defeating RandomPlayer and HonestPlayer. We extended the training to 5000 cycles, and we observe our DQN agent bankrupting both players.

It is interesting that HonestPlayer performed worse than RandomPlayer. We hypothesised some reasons why this could have occurred:

1. The machine we tested on (AWS) could not carry out the Monte-Carlo calculations fast enough. HonestPlayer ending up timing out and folding a large number of times.
2. HonestPlayer had no form of deceit involved, so whenever HonestPlayer raised our DQN knew that their hands were very strong and we should just fold. When HonestPlayer would win, they would only win a small amount of money. On the contrary, RandomPlayer may have folded good hands by chance but unintended bluffing meant that RandomPlayer actually had a chance of winning a significant amount from the pot.

G Tuning of DQN Hyperparameters

We tuned the training batch size, hidden layer size and learning rate of our DQN. Given our limited time, we could only experiment with select combinations of parameters.

We followed intuitive combinations, increasing learning rate as we increased the batch size. Our tuning process involved the following sets of hyperparameters:

No.	Batch Size	Hidden Layer Size	Learning Rate
M1	64	64	0.0001
M2	128	128	0.0001
M3	128	128	0.001
M4	128	256	0.001

Table 2: Tuning of DQN Hyperparameters

We let DQN agents trained with these parameters play with each other over 10 games, and recorded the results below.

M1	**M1** vs. M2	
M2	132600 : 67360	M1 vs. **M4**
M3	M3 vs. **M4**	69600 : 130340
M4	47420 : 152560	

Table 3: Results of Tuning DQN Hyperparameters

Given 5000 training cycles., the model with a batch size of 128, hidden layer size of 256 and learning rate of 0.001 performed the best.

H Agent Implementations

Our implementation of the CFR and DQN agents, as well as our training algorithms: `https://github.com/pikulet/poker.git`