May the Flop be with you - Deep RL based No-Limit Texas Hold'em Pokerbot

Srivatsan Srinivasan SRIVATSANSRINIVASAN@G.HARVARD.EDU
Sebastien Baur SEBASTIENBAUR@G.HARVARD.EDU
Donghun Lee DONGHUNLEE@G.HARVARD.EDU

Abstract

Imperfect information games such as Poker have always proven challenging to master for an AI agent. In this work, we introduce a self-sufficient end-to-end approach to learning approximate Nash equilibria in such games without prior domain knowledge. For the learning agent, we have utilized the architecture of Neural Fictitious Self Play(NFSP) where each playing agent stores separate experience buffers and trains two neural networks that learns best action values and average strategy using off-policy deep Reinforcement Learning and supervised learning respectively. We also develop subnetworks that learn abstracted states and employ frozen hidden layer weights to represent hand and board in a reduced dimension space. We then demonstrate few aspects of the learning model's performance through structured experiments of the agent's game play against baseline agents and its own clones.

1. Introduction

Poker is an archetypal game of imperfect information. The challenges involved in poker include sophisticated decision sequences, large state-action space, imperfect information and adversarial behavior, presenting a perfect template for probabilistic inference and deep reinforcement learning to sequentially learn from the game states via non-linear models. Heads-up no-limit Texas Hold'em (HUNL) is a two-player version of poker in which two cards are initially dealt face down to each player(pre-flop), and additional cards are dealt face up in three subsequent rounds(flop(3),turn(1) and river(1)). No limit is placed on the size of the bets although there is an overall limit to the total amount wagered in each game.

Imperfect information games demand way more complex reasoning than perfect information games in the same state space. The main complication is due to recursive reasoning, where an appropriate decision at any point in the game is a function of the probability distribution over latent(private) information of your opponent(cards, strategy) and opponents' mechanism of revealing that information depends on the players' actions. This makes it impossible to reason about a game state in isolation, the backbone of usual heuristic search methods in perfect information games. Combination of imperfect information with the humongous state space of poker also renders classic supervised learning through simulation of all possible scenarios prohibitively large. Hence an off-policy deep Reinforcement Learning framework where the agent learns through self-play(learning by playing against its own clones) is a natural direction that this work adopted in the training procedure.

2. Background

In this section, we provide a brief overview of deep reinforcement learning, extensive form games and Fictitious Self-Play. We would suggest the readers to refer (Sutton & Barto, 1998) for fundamentals of Reinforcement Learning, (Schaul et al., 2015) and (Silver et al., 2016) for Experience Replay and for Double DQNs and (Heinrich et al., 2015) for extensive form games and fictitious self-play to assimilate a more in-depth background.

2.1. Extensive Form Games

Extensive form games(Wikipedia) allow explicit representation of pivotal game aspects, like players' move sequences, choices made, the information each player has about the other players' decision mechanism and the utilities for each resulting action. A perfect in-

Submitted as part of final project requirements in *CS281:Advanced Machine Learning*, Advisor: Prof. Alexander S. Rush, Fall-2017 SEAS, Harvard University..

formation game is characterized by a player knowing exactly the history of happenings in the game and the information set is the same singleton for both players. Games that fall under the purview of this include tic-tac-toe, checkers and Go. Most card games such as Poker are imperfect information games. In **imperfect-information games**, each player only observes his own information states(only his own cards in poker) and generates a strategy profile that maps information states to a distribution over possible actions. A **strategy profile** (Heinrich & Silver, 2016) is a collection of strategies of all players and a best response is one that obtains optimal payoff against the other strategies within the strategy profile. In this context, a **Nash equilibrium** is a strategy profile in which each player's strategy is a best-response to the other strategies in the profile - essentially a deadlock where no single player can gain by deviating from his strategy.

2.2. Reinforcement Learning

Reinforcement learning(RL) agents typically try to maximize the expected reward when reacting with its given environment (Sutton & Barto, 1998). Usually, the environment is modeled under the MDP framework, where transition from a state does not depend on history, and overall the agent tries to find a trajectory that maximizes its rewards. Many reinforcement learning algorithms learn from sequential experience in the form of transition tuples $(s_t; a_t; r_{t+1}; s_{t+1})$. The problem is setup to learn the action-value function Q(s,a) which is the value of an action taken from a state followed by adherence to the base policy from the next state onward. Q-values can be setup to learn off-policy, i.e. learn all Q-values while the agent enacts a policy different from the one suggested by Q-values. **Q-Learning**(Watkins & Dayan, 1992) is an algorithm mostly to this effect where we learn from a batch of transitions. **Double-DQN** with experience replay(Silver et al., 2016) is a recent advancement in Q-learning where the agent learns Q-values from experience data by fitting neural networks in the state-action space.

2.3. Fictitious Self-Play

Fictitious Play is a game-theory framework where an agent chooses its best-response to the expected(average) opponent's response profile. It has been proven that the average strategy of the fictitious players converge to Nash equilibrium in two-player zero sum games(both for perfect and imperfect information games)(Leslie & Collins, 2006). Advances in machine learning and reinforcement learning have helped develop approximations to these two profiles using game events. Fictitious self-play, introduced by (Heinrich et al., 2015), replaces best response and average response computation with Q and π(policy) values learned from reinforcement learning and supervised learning respectively from the game transitions observed by the agent.

3. Related Work

Studying games has drawn great interest from the academic community because the dynamics of several real-world phenomena such as network security (Tam, 2011), traffic control, financial and energy markets (Nemivaka et al., 2006) could be abstracted to a miniature version of game-play. (Brown, 1951) introduced fictitious play as a popular method for learning Nash equilibrium in extensive form-games. (Heinrich et al., 2015) introduced the idea of Fictitious Self-Play(FSP) where agents learn the best response and average policy by playing against its own clones and leverages reinforcement learning and supervised learning to develop these approximations. Neural Fictitious Self Play(NFSP) (Heinrich & Silver, 2016) is the more recent advancement to FSP where the authors leveraged DQN for learning the best response policy and another neural classifier to learn the average policy.

Initial versions of Q-learning always leveraged linear function approximations as it was shown that combining model-free reinforcement learning algorithms such as Q-Learning with non-linear function approximators (Tsitsiklis & Roy, 1997) or indeed with off-policy learning could cause the Q-network to diverge. Subsequently, the majority of work in reinforcement learning focused on linear function approximators with better convergence guarantees. The advent of deep learning, triggered an active interest in research that combined deep learning with reinforcement learning. Deep neural networks have then been used to estimate the environment and (B. Sallans, 2004) used Restricted Boltzmann Machines to estimate the value function. Deep Learning was first applied to non-linear control by (Mnih. et al., 2013) while introducing Deep-Q networks(DQN) to play Atari games only using the visual frames of the games as inputs. Following this, Double deep-Q networks(DDQN) were introduced by (Silver et al., 2016) as an improvement over the DQN idea, in which the authors introduced a target-Q network to counter the over-optimism biases prevalent in Q-learning. At the same time, prioritized experience replay (Schaul et al., 2015) was introduced as an improvement over the experience replay buffer used by (Mnih. et al., 2013) as it prioritizes experience sam-

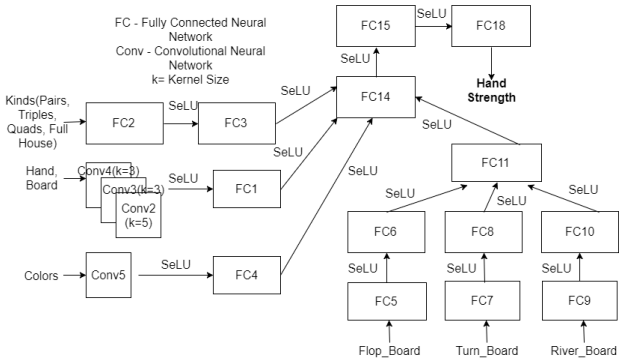

Figure 1. Neural Network architecture for the state space abstraction - Neural Net that predicts implicit hand strength from the hand and the board

pling using the temporal difference error in Q-learning for each experience.

Our work, while it draws inspiration for the NFSP algorithm from the works of (Heinrich et al., 2015) and (Heinrich & Silver, 2016), develops its own approximate best response network(DDQN) and average policy(π) network architectures. Besides, this work also includes indigenous implementations of experience storage, MDP definition and state-abstraction through neural networks(card featurizer), experiments monitoring framework and an end-to-end poker game simulator.

4. Models

We have setup the problem as a reinforcement learning problem with an underlying Markov Decision Process. Poker as MDP is plagued by the size of the state space. For instance, the river cards could potentially be anything among the $\binom{52}{7} * \binom{7}{2} = 2,809,475,760$ possible card combinations and thus enumerating each card combination is a futile exercise computationally. Hence, we have setup a neural network which abstracts the different hand board combinations and their predicted hand strength into a frozen set of 500 weights (not trained further), thus having a lower-dimensional yet accurate representation of cards. Finally, we implement the NFSP algorithm, which is constituted by two neural networks, one for each of Q and π. The following subsections will outline each of this modeling ideas concisely. For the purposes of this work, we are going to refer pre-flop, flop, turn and river as T_1,T_2,T_3 and T_4 respectively.

4.1. MDP Setup

We represent the hand(H) using a 13 * 4 matrix in with each value being an indicator. Similarly the board(B) is represented via a 3[T_2,T_3 and T_4] * 13 * 4 matrix, with again the cards represented by an indicator. Let PH represent the play histories(actions in different turns). With pot, dealer, big blind, player and opponent stack being represented by P,D, BB, PS_1 and PS_2, any state at a point of time s_t can be represented as $[H, B, P, D, BB, PS_1, PS_2]$. The actions are represented by a 16-dimensional vector as - [Fold,Check,Call,All-in,Raise-Bins(12)]. The rewards are only awarded at the end of each episode as the net change in player stack for that episode. Every round is treated as an episode that ends with either a showdown or a fold. A game is said to end when one player's stack becomes empty.

4.2. Card Featurizer

In spite of being a huge state space ($\approx 2 \cdot 10^9$), poker provides scenarios that could be considered the same by virtue of symmetry - (Kh, Qd) is the same as (Qh, Kd). On top of the suit symmetry properties, the rules of the game helps us treat several seemingly dissimilar cards in the same fashion. For instance, the hands [2s, 3c] and [4s, 7d] with a board [Kh, Qh, Th] should be played almost exactly the same because they have about the same strength and the same possible game evolutions. We intend to capture this intuition of taking similar action in similar scenarios through hand strength which would help us reduce the dimensionality drastically.

(M. Johanson et al., 2013) suggest that a way of reducing the state space is to cluster together situations that have a similar hand strength. **Hand Strength**(HS) as a function of the hand and board is the probability of the player's hand beating a uniformly sampled hand in a potential showdown(with remaining board cards sampled uniformly too). Our idea was to use a neural network as a featurizer to cluster situations with similar HS. From the hand and the board, the neural network is trained to predict HS. Once trained, we can use the output of the last hidden layer as a hand-board encoder. We generated this dataset by running 10^6 Monte-Carlo Simulations for pre-flop, flop , turn and river scenarios and generated a dataset of [Hand, Board, Win Probability]. Once we generated the dataset we employed a neural network as shown in Figure 1. This model has two inputs: the hand and the board (a 13*4 and a 3*13*4 tensor). The board has 3 as first dimension because we intend to produce 3 outputs, one per round, that will then be processed

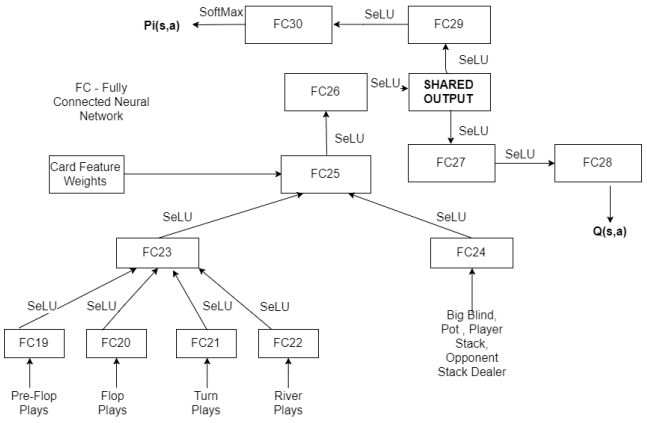

Figure 2. Neural Network architecture for Q and π networks. They share common set of weights until they branch out into corresponding hidden layers.

by the q network together with other game inputs.

Predicting the hand strength calls for detecting the combinations formed by the hand and those formed by the board alone. From the board and hand, we therefore compute two tensors, containing the count of suits and ranks. In order to detect possible straights, we process the latter with convolutional layers to capture the spatial information. As the problem is symmetrical with respect to suits, the tensor with suit counts is pre-processed by a 1x1 convolution before being processed by fully connected layers. The rest of the inputs is processed with fully connected layers (to detect pairs, trips, quads, full houses). Concisely, the network could be simplified as follows with FC and conv denoting fully connected and convoluted layers respectively.

$$HS \sim FC\Big[Conv(Ranks) + Conv(Suits) + FC(FlattenedHand, Board)\Big]$$

This neural network is pre-trained before game-play and its weights are frozen and used per se in a forward pass while training the best response and average policy networks in NFSP. We use its output, as well as the outputs of some of its hidden layers, as inputs of the Q network.

4.3. Q and π Networks

Figure 2 shows the neural network architecture of Q and π networks. Here each fully connected layer represents a part of the state-action dimensions. The hand and board are implicitly processed through hand strength via a FC layer, the pre-flop, flop, turn and river action histories(called play histories, P) are passed by FC layers(processed via different NN because of different tensor dimensions) and the other global game variables(G) such as pot, dealer, stacks, blinds are processed in another layer, whose hidden layer weights are combined in another fully connected layer to produce a shared output, from which the RL(Q) and SL(π) networks branch out via another respective hidden layer. With the shared output represented by S, the model could be concisely summarized as follows.

$$S \sim FC(FC(P) + FC(G, HS))$$
$$Q \sim FC'(S)$$
$$\pi \sim FC''(S)$$

While training the Q-network we use a Double Deep Q-Network that reduces over-optimism in off-policy Q-learning by decomposing the max operation in the target network into target selection and target evaluation(Silver et al., 2016) These models are trained during self-play with prioritized experience replay buffers for Q-network and a uniformly sampled reservoir buffer for the supervised π network.

4.4. Experience Buffers

Experience buffers are another key aspect of training DDQNs as it enhances data utility efficiency, removes strong correlation that would have otherwise been part of contiguous samples and avoids locally unwanted feedback loops(Mnih. et al., 2013). In our work, we train two networks in parallel with two different set of experiences - M_{RL} a Reinforcement Learning memory of (s,a,r,s') experienced by the agent in each iteration and a M_{SL} a Supervised Learning memory that stores the agent's best response actions taken by the Q network. For M_{SL}, we employ a reservoir buffer(with exponential decay) of size 10^6 and we sample our experiences uniformly. For the M_{RL} - a simple circular buffer of size 10^6, we use prioritized experience replay (Schaul et al., 2015) where we sample experiences from a priority queue with the TD-error in Q Learning as the priority metric. The sampling from M_{RL} employs a rank-based mechanism, where the probability p(i) of a transition i being sampled is as below.

$$p(i) \propto \frac{1}{rank_i}$$

The algorithm requires updating the priorities of samples at each backward pass of the Q network. To reduce computational burden, we followed the same scheme as in (Schaul et al., 2015) by introducing stratified sampling where each sample gets assigned to a strata and samples in the same strata are uniformly sampled. The prioritization is handled by varying the number of samples to assign to each strata.

4.5. Neural Fictitious Self-Play

The Neural Fictitious Self-Play algorithm is the connecting thread across the models that we described in the previous subsections. NFSP (Heinrich & Silver, 2016) combines FSP with a neural network for approximating the response functions. In Algorithm 4.5, each player is a NFSP instance that learns from simultaneous play against its own clones, i.e. self-play. In these game interactions, the NFSP agent caches its experience of game transitions(experiences) and its own best response behaviour in two memories, M_{RL} and M_{SL}. With these two distinct datasets, NFSP agent trains a neural network, $Q(s, a|\theta^Q)$, to predict action values from data in RL experience memory using Q-Learning(off-policy). The resulting network defines the agent's approximate best response strategy, $\beta = \epsilon-$greedy$Q$. To imitate its own past best response behavior, the agent trains another neural network $\Pi(s, a|\theta^\Pi)$ using supervised classification from data in M_{SL}. As this is a state and action probability map, this defines the agent's approximate(average) stochastic response policy π. We also employ an approximation for anticipatory dynamics derived in (Shamma & Arslan, 2005) where the NFSP agents choose their actions from the mixture policy $\sigma(1-\eta)\cdot\beta + \eta\cdot\pi$. In classic off-policy RL setting, you can play your average policy π while evaluating and maximizing your action values that results in the best-response policy β. But in the NFSP settings where the agents play against each other, if we adopt this idea, the agent will not have any experience of its own best-response behavior which is imperative to train the average policy network $\Pi(s, a|\theta^\Pi)$ (Heinrich & Silver, 2016). To address this problem, we employed the aforementioned mixed sampling policy that creates a richer diversity of experiences.

Algorithm 1 Neural Fictitious Self-Play

```
Initialize game G with 2 players.
Execute each player via runAgent.
function runAgent(G):
    Initialize replay memories M_RL, M_SL
    Initialize action-value network Q(s, a|θ^Q)
    Initialize average-policy network Π(s, a|θ^π)
    Initialize anticipatory hyper-parameter η
    Define L_SL as cross entropy loss on π
    Define L_RL as TD Error loss on Q value.
    for each episode do
      σ: P(σ = ε greedy Q ) = η, else σ = Π
      Observe s_1, r_1
      for t=1,T do
          Sample action at from policy σ
          Execute a_t and Observe r_{t+1}, s_{t+1}
          Store (s_t, a_t, r_{t+1}, s_{t+1} in M_RL)
          if σ = ε greedy Q then
              Store (s_t, a_t) in M_SL
          end if
          Update θ^Π with SGD on loss L_SL
          Update θ^Q with SGD on loss L_RL
      end for
    end for
```

5. Training

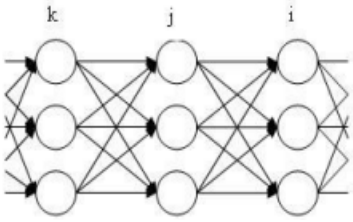

Figure 3. A simple multi-layer feed-forward neural network. Source : www.images.google.com

Backpropagation is a generalization of the gradient descent learning rule to multi-layered feed-forward networks, using the chain rule to iteratively to compute gradients for each layer. To understand backpropagation, let us consider a simplified version of a neural net shown in Figure 3. At each step, for the weights connecting a node in layer k to a node in layer j, the change in weight is given by:

$$\triangle w_{kj}(n) = \alpha \cdot \delta_j y_k + \eta \cdot \triangle(w_{kj}(n-1)$$

where α is the learning rate, η is the momentum, n is the epoch number, and y_k activation of the presynaptic node and δ_j is the error-term associated with the post synaptic node. The error term for a node in a hidden layer requires the error term from nodes in the subsequent (i.e. downstream) layer, and so on until the output layer error terms are calculated. Thus, computation of the error terms must proceed backwards through the network, beginning with the output layer and terminating with the first hidden layer, giving it the name backpropagation. The idea of backpropagation. Backpropagation works under the assumptions that the loss function can be written as function of the NN outputs so as to compute gradients and the overall loss function can be averaged over losses of the individual training examples(backpropagation works on smaller batches) (M.Nielsen)

We have three neural networks that we train as part of this work - Card Featurizer (Figure 1), Q-Network and π network(Figure 2. All of them are feed-forward neu-

ral networks and the gradient updates happen through backpropagation. We built the neural networks as dynamic computational graphs in PyTorch, with the forward passes computing the losses and the backward passes computing gradients and adjusting weights. The training happened for all the networks in batch sizes of 256 on a GPU with several iterations of learning rate varying between 0.001 and 0.1. We also used batch normalization (Ioffe & Szegedy, 2015) in order to account for internal co-variate shifts and normalize input layers, so that the model is regularized and becomes less sensitive to model initialization issues.

In the card featurizer network, our input dataset comprises cards represented as matrices and dependent variables in the form of probabilities in [0,1]. It would be inappropriate to use either the standard MSE or CrossEntropyLoss on this kind of dataset. So, before training, we used an inverse sigmoid transformation on the hand strength, then scaled and normalized the data before it was processed in the neural network. Now that we have transformed the dependent variable on to the continuous space, we used the MSE loss as our loss metric to train the network. Because Q-values are continuous and unbounded(not strictly true as our rewards are in [-100,100]), we again leveraged MSE loss as the forward loss function. For the π network, our final loss function was a cross-entropy loss across the action bucket classes since it is a supervised classification problem.

6. Methods

6.1. Hyper-parameter Choices

This problem contained a massive number of hyperparameters to optimize for and we focused on a few of them which we felt added maximum utility in our learning process. We chose a γ of 1 in our MDP setup since the episodes are fairly short and that there is no tangible advantage in preferring shorter episodes over longer ones. The anticipatory parameter η has been set to 0.1 for standard NFSP agents following the results of (Heinrich & Silver, 2016). Q-learning algo has been chosen with an initial ϵ of 0.1(to promote exploration) with a decaying ϵ schedule as the number of episodes increases. The hyperparameters of the experience buffers were chosen mostly keeping computational memory constraints in mind. In the buffers, we set the number of partitions to 2^{11}, overall buffer size to 2^{18} and initial learning is set at 2^{10} episodes(Refer (Schaul et al., 2015)). Backpropagation was done once every 2^6 transitions.

6.2. Model and Software Development

As part of this work, the entire MDP setup, neural network architectures and implementations(using PyTorch) have been developed by the authors. We also built a completely tested end-to-end poker simulator mainly because we could not find good open-source simulators that we could integrate with our MDP definitions. Drawing from the works of (Heinrich & Silver, 2016), we implemented our own version of the NFSP algorithm. We inherited open-source code for Prioritized Experience Replay and developed wrappers to connect our simulator's outputs with the code's experience storage mechanism. Then, we developed an experiment framework that allows us to tune the aforementioned hyperparameters and run any form of structured experiments across the agents. As part of the experiments monitoring, we also created a real-time performance monitor local server using TensorBoard for monitoring several performance variables live while the simulations happen, such as hand strengths, actions, rewards, Q and π network losses. The framework is nimble enough to add any new monitor variables in the future.

6.3. Experiments

To test our models, we tested our agent against a few baseline agents and their own clones. The baseline agents were **Random** and **Mirror** agents.The random agent chooses its actions with uniform probability from allowed actions. The Mirror agent is an agent that mimics player's actions. For instance, if NFSP agent raises, Mirror will call. If NFSP agent checks, it will check too and if Mirror agent goes first, it always checks. It is very important to note that playing against a Mirror agent is not the same as NFSP playing against its own clone as the mirror agent reciprocates NFSP agent's action irrespective of its own cards and board. After inferring performance from these two agents, we ran experiments between NFSP agent and its own clone. A DDQN agent could also be considered a NFSP agent with the anticipatory parameter $\eta = 1$. We monitored the network training losses, the rewards earned by the agents and also tried to reason out certain actions taken by the agent in specific scenarios.

7. Results

7.1. Card Featurizer

We collected a total of 1710396 samples(MC simulations) as the target hand strength data. Subsequently we split them into training, validation(10%),

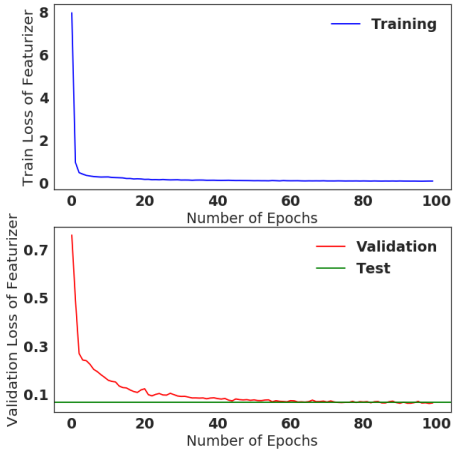

Figure 4. Training, Validation and Test Losses of card featurizer model.

Hand	Board	True HS	Pred. HS
2S, 3H	NA	0.37	0.2614
KS, KS	NA	0.65	0.767
2H,7D	9c,TD,TD	0.24	0.22
7C,JD	5C,JH,QC	0.6764	0.6706
6C,TS	2S,8D,9H,KD	0.30	0.33
AC,AD	9H,KD,AH,AS	1	0.9862
4S,8H	8C,9C,TH,JH,AH	0.299	0.3144
7H,QD	6H,6D,7D,QS,QC	0.999	1

Table 1. Examples of hand strength predicted by card featurizer. HS - Hand Strength. TD indicates dice of Ten.

and test(10%). We tuned hyper-parameters (the learning rate and and weight decay rate of Adam optimizer) and selected a model with the best validation error averaged over 100 epochs. Subsequently, we tested the final model against the test data. In Figure 4, we can see that the training and validation MSE losses converge very effectively while yielding a low test loss. To give a flavor of the network's prediction, Table 7.1 presents the real hand strength learned through Monte Carlo Simulations and the hand strength predicted by our card featurizer network.

7.2. Q and Π networks

Once we have a good card featurizer whose weights have been frozen during the training of the Q and π networks, we see that the MSE loss on Q and the cross-entropy loss on π both converge as we learn from more number of games/episodes.Figure 5 shows the case of converging losses when the games were simulated between NFSP agent and its own clone.

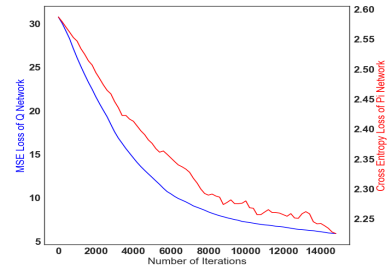

Figure 5. Plots showing Q and π network losses with time.

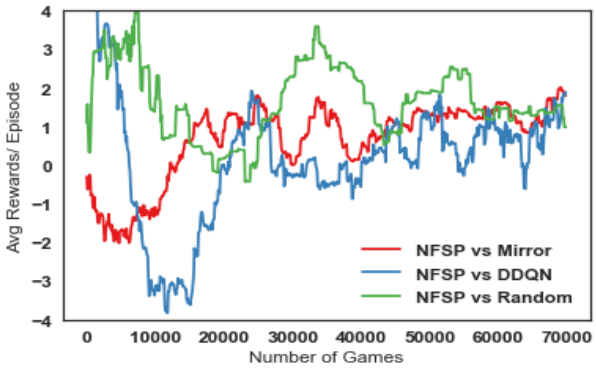

Figure 6. Average rewards per episode won by NFSP agent over DDQN, Random and Mirror Agents.

7.3. Experiment Results

Figure 6 shows the performance of the NFSP agent against three agents - random, mirror(Subsection 6.3) and the DDQN agent. We observe that the agent *continuously adopts an increasingly optimal strategy as it learns from playing more number of games* against both the DDQN and the mirror agents. Against random agent, there is no clear learning trend seen even though the agent earns sizable positive rewards. It is because while playing against a random agent, NFSP folds very often(since random agent raises very frequently) and does not learn many useful experiences in each episode. Figure 7 shows the reward structure when two NFSP agents train to play against each other. Here, we see that there is no clear outperformance of one agent over the other, indicating an equivalence of learning performance. Few of the interesting aspects of the game learned by these agents can be found in Section 8.

8. Discussion

As part of the experiment results, we devoted our attention to understanding a few mechanics of our agent through our TensorBoard metrics monitoring framework. First, we validated the card featurizer's im-

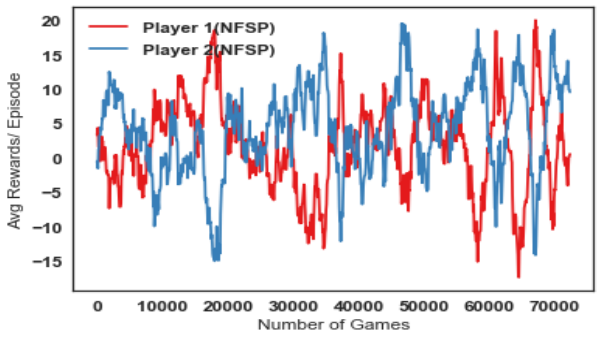

Figure 7. Average rewards/episode won by battle between NFSP agents. There is no clear winner after 50k games in contrast to NFSP's performance vs. baseline agents.

pact to the model by asserting that both networks levy non-trivial weights on the card featurizer inputs and the gradients for those weight updates do not vanish. Then, we manually investigated certain states under which we were expected to take similar actions. For instance, a.) Ks, 9s(Hand)/3d, 5s, As, 4s(Board) and b.) Ah, 9h/2d,3h,Jh,6h, with similar game variables(pot,stack etc.) were dealt with the same set of optimal actions[Raise above $30 at least all the way up to all-in bucket] by our networks. Similar actions were also suggested with 2s,3c/ 8d,5d,As and 2s,3h/Td,Jd,As - outwardly different states which warrant similar actions under the lens of poker. We also observed exact same hands being dealt with different actions when the player stacks were different. While we observed such intuitive behavior in certain cases, we also found a few instances where the agent behaved sub-optimally - for instance, raising high with a weak hand and losing the hand eventually. We could not attribute this specifically to an issue with the model or to limited training experience data.

We also witnessed certain interesting shifts in action patterns in these simulations. For instance, when playing against mirror agents, the average number of checks per episode remained at 0.8 around the first 5000 games, and the number increased to 2.45 in 50000 games. Check is a sensible route to take in these games as the mirror agent is a conservative copy-cat. The agent also learned to fold less frequently in the earlier rounds of the games(Prefers to check in case of low HS or raise with higher HS). While playing against a DQN agent, the NFSP agent learned to fold more often, starting from about 24% in 5000 games to about 57% in 50000 games and and on limited manual inspection, there was a correlation with agent folding and weaker hands. Also, folding happened during pre-flop and flop plays predominantly(average episode lengths of 1.6 on folds). DQN player too learned certain similar patterns, albeit much less cogently.

In the games between two NFSP agents, we observed that both the player agents learned similar actions in certain scenarios. For instance, after 50000 games, agent 1 had transcended to a strategy which led to approximately 0.56 raises per episode while agent 2 learned about 0.59 raises per episode. Similarly, the number of folds per episode was approximately 0.51 and 0.47 respectively for each agent. While we saw such tighter convergence in the actions, we observed that the similarity between the weights of the neural networks learned by both the agents was growing very slowly, this leaving us with no decisive conclusions about the proximity of the weights learned.

8.1. Future Work

Code profiling results(Appendix section 10.2) indicate that different elements combine to cause a single episode to run for $\sim$ 200ms. Simpler model, revamped buffer architecture and better software could bring down game latency by a few orders of magnitude and make it feasible to run $O(10^8)$ games(similar to (Heinrich & Silver, 2016)) in a simulation to better train the network. Owing to this limitation, we have not been able to freeze our model to play against standardized commercial agents, which would be plausible after more simulations and the ensuing consistent convergence. Another shortcoming that we noticed is that our rank-based ER buffer did not sample extreme scenarios frequently enough and hence the network did not learn from showdown experiences more often. We believe that converting the buffer to a proportional ER buffer would help allay this issue. To expedite convergence, we could also incorporate an opponent style knowledge to the network, akin to the work in (Heiberg, 2013).

9. Conclusion

With this work, we developed a neural network that predicts hand strength given player's hand and the board. Building from there, we developed a NFSP bot, an end-to-end deep RL agent that learns from its experiences via self-play. Through baseline experiments and analysis, we have been able to demonstrate that the agent learns to eschew ridiculous actions and plays strategies that lead towards more optimal rewards. While our results are still some distance away from decisively concluding about the expert level performance of this agent, they definitely provide strong encouragement about the efficacy of our approach that employs NFSP to imperfect-information games.

References

Security and game theory: algorithms, deployed systems, lessons learned, 2011.

B. Sallans, G. E. Hinton. Reinforcement learning with factored states and actions. *Journal of Machine Learning Research*, 2004.

Brown, G. W. Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 1951.

Heiberg, A. Using bayesian networks to model a poker player. In *Proceedings of the 2013 AAAI Conference*, 2013.

Heinrich, J. and Silver, D. Deep reinforcement learning from self-play in imperfect-information games. *arXiv: 1603.0112*, 2016.

Heinrich, J., Lanctot, M., and Silver, D. Fictitious self-play in extensive-form games. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv: 1502.03167*, 2015.

Leslie, D. S. and Collins, E.J. Generalised weakened fictitious play. *Games and Economic Behavior*, 2006.

M. Johanson, N. Burch, Valenzano, R., and Bowling, M. Evaluating state-space abstractions in extensive-form games. In *Proceedings of the 12th International AAMAS Conference*, 2013.

M.Nielsen. Backpropagation. http://neuralnetworksanddeeplearning.com/chap2.html.

Mnih., V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., and Ostrovski, G. Playing atari with deep reinforcement learning. *arXiv: 1312.5602*, 2013.

Nemivaka, Y., Feng, Y., and M.Kearns. Reinforcement learning for optimized trade execution. In *Proceedings of the 23rd International Conference on Machine Learning*, pp. 673—-680., 2006.

Schaul, T., Quann, J., Antanouglou, I., and Silver, D. Prioritized experience replay. *arXiv: 1511.05952*, 2015.

Shamma, J.S. and Arslan, G. Dynamic fictitious play, dynamic gradient play, and distributed convergence to nash equilibria. In *IEEE Transactions on Automatic Control*, volume 50(3), pp. 312–327, 2005.

Silver, D., Guez, A., and van Hasselt, H. Deep reinforcement learning with double-q learning. *arXiv: 1509.06461*, 2016.

Sutton, R. S. and Barto, A. G. (eds.). *Reinforcement Learning - An Introduction, Volume-1*. MIT Printing Press, 1998.

Tsitsiklis, J.N. and Roy, B. V. An analysis of temporal-difference learning with function approximation. In *IEEE Transactions on Automatic Control*, volume 5, pp. 674—-690, 1997.

Watkins and Dayan. Q-learning. In *Machine Learning*, volume 8, pp. 279–292, 1992.

Wikipedia. Extensive-form games. https://en.wikipedia.org/wiki/Extensive-form_game.

10. Appendix

10.1. Acknowledgements

We would sincerely like to thank Prof.Alexander S. Rush for many of his timely and useful inputs in shaping the direction of this project. We would like to thank the Professor and SEAS, Harvard University for kindly offering sponsoring cloud computing facilities in order to train our models. We also thank the teaching fellows of the CS281 course for their continued mentorship and insightful feedback throughout the course of this project.

10.2. Profiling Results

Software Segment	Time Taken(ms)
Single Forward Pass	20
Single Backward Pass	50
Experience Replay Sampling	50
Single Episode Simulation	200
Single Game Simulation	2300

Table 2. Code Profiling Results

10.3. Code Repository

All the code that we wrote as part of this project can be found in the master branch of https://github.com/Srivatsan-Srinivasan/CS281-Final-Project. Key script that would serve as entry point into our simulations would be perf_eval_experiments.py that could be found in the Programs directory. We have tried our level best to add comments in several pieces of the code and would strive to add more in the future.