Poker-RNN: Playing Limit Hold-Em with Recurrent Networks

Mukund Venkateswaran[1] **Saniyah Shaikh**[1] **Artem Kuriksha**[1]

1. Introduction

Deep reinforcement learning currently is one of the most actively developing strands of the machine learning research. The potential benefits are mainly seen to be related to real-world applications like data-centers management (Lazic et al., 2018) or robotics (Gu et al., 2017; Abbeel et al., 2007). However, for a large part of the advancements in the field, the working ground is recreational games. Recent examples have attracted a lot of attention within and outside the machine learning community (Silver et al., 2017; Vinyals et al., 2019; Mnih et al., 2013).

In particular, deep reinforcement learning algorithms now achieve super-human performance in classic games like Chess and Go. These games are perfect-information games, which means that for a given combination of actions the reward is deterministic and calculable for each of the players. Most card games, however, are imperfect information games, meaning that each of the players faces uncertainty about the state of the game while making a move. In poker, for example, each player is uncertain about the hands of their opponents. This uncertainty depends not only on the stochastic component introduced by the initial shuffling randomization but also on the strategies that the opponents use to play.

Consequently, many of the algorithms that demonstrate good results for perfect-information games fail to converge for games of imperfect information; for example, DQN is one such algorithm (Heinrich & Silver, 2016). This issue has motivated several algorithms that are based on the game-theoretic ideas of ”learning” in games, including fictitious play as originally suggested by Brown (1951). These game-theoretic ideas were developed when researchers investigated theoretical properties of Nash equilibria and looked for various justifications to apply Nash equilibrium as a solution concept. These ideas then were applied in computational game theory, and can now be incorporated into the deep reinforcement learning framework. For a detailed coverage of the theory of learning in games, refer to Fudenberg et al. (1998).

In this project, we apply a broad range of deep reinforcement learning techniques to poker, a popular imperfect-information game. Poker provides an interesting and challenging case when many traditional reinforcement learning algorithms can perform badly and do not converge. Importantly, we also can try to impose some inductive biases based on our knowledge of the game structure.

We apply five methods: linear Q-learning, deep Q-learning, deep recurrent Q-learning, neural fictitious self play (NFSP), and recurrent neural fictitious self play. The first method serves as the non-deep learning baseline to compare all other algorithms to. The second method, deep Q-learning, is a standard approach in RL that proved to be efficient for many other problems. Deep recurrent Q-learning exploits the consequential nature of poker to provide an inductive bias and improve efficiency. Finally, the two versions of NFSP rely on the ideas borrowed from learning game theory.

We seek to compare the performance of recurrent and non-recurrent models through our analysis as well as the differences seen between deep Q learning and neural fictitious self play in games of imperfect information. We find that the neural fictitious self play model takes a long time to converge in the Limit Hold-Em game in both its ordinary and recurrent formulation, and that recurrence only seemed to slow the convergence of the reinforcement learning algorithms.

2. Related Work

2.1. Playing Atari with Deep Reinforcement Learning

In Mnih et al. (2013), 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) → 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. We developed a baseline for our deep learning methods in this two-player imperfect information game by trying to learn a Q-network for policy in

[*]Equal contribution [1]University of Pennsylvania, Philadelphia, PA. Correspondence to: Mukund Venkateswaran <mukundv@seas.upenn.edu>, Saniyah Shaikh <saniyah@seas.upenn.edu>, Artem Kuriksha <kuriksha@sas.upenn.edu>.

poker.

2.2. Deep Reinforcement Learning from Self-Play in Imperfect-Information Games

Deep Q Learning has its flaws in the multiplayer imperfect information setting as its learned strategy is quite exploitable depending on the circumstances in which it was trained. In order to learn approximate Nash equilibria in an imperfect information setting, DeepMind introduces Neural Fictitious Self Play (NFSP), a reinforcement learning framework based on Fictitious Self Play (Heinrich & Silver, 2016). The NFSP algorithm works by adding an average policy network to the Deep Q learning framework and training this network to output the average policy learned by the action value network over the training period. They find that the average policy empirically converges to an approximate Nash equilibrium in the game of Leduc Hold-Em, and to near state-of-the-art policies in Limit Hold-Em.

2.3. Deep Recurrent Q-Learning for Partially Observable MDPs

(Hausknecht & Stone, 2015) 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.

2.4. Non-DL Work on Poker

Poker was an object of interest in artificial intelligence research for a long time. Among other algorithms that do not utilize neural networks and rely on the prior knowledge of the game, several projects managed to achieve some success or a super-human performance for a particular type of poker in the past. Bowling et al. (2015) and Brown et al. (2017) both are based on a Monte Carlo Tree Search (MCTS) with counterfactual regret minimization. Heinrich & Silver (2015) combine MCTS with self-play.

3. Methods

3.1. State Representation, Action Space, and Explaining the Game

The objective of poker is to have the best 5 card hand between your own two cards and the 5 cards on the board. There exist some complex patterns between cards that determine the value of hands, i.e. pair of a number, three of a kind, four of a kind, straight (five cards forming a sequence), and flush (five of one suit) to name a few. Betting consists of four rounds: 1 round prior to any community cards being shown, 1 round after the first 3 community cards have been shown, 1 round after the fourth community card has been shown, and 1 round after the final community card has been shown. Winnings are measured 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. 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 (Zha et al., 2019). 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). This information is provided to us in the 'possible_actions' attribute of the environment.

An interesting point to note, and the basis for our methodology here, is that the state space of the game is represented to the agent as a bag-of-cards. This is similar to the bag-of-words model in the field of Natural Language Processing (NLP) where a sentence is represented simply as the counts of words inside of it, disregarding the order in which the words appeared in the sentence. Though this was useful in a few NLP and Information Retrieval tasks, models that take word order into account have become the state of that art in many NLP tasks (even simple models like n-grams up to RNNs and Transformers). Similarly, the order in which cards are presented to the agent is very important in the game of poker.

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.

3.2. Non-deep learning baseline: Linear Q-Learning Approximation (LQA)

Our first trained model to baseline performance of our models was training a simple [72 x 4] linear layer to estimate Q-values from linear combinations of the state vector. This learning algorithm is the same as the Q-Learning algorithm outlined below, but a single linear layer without activation is used to estimate state-action values, rather than the Q-networks. After experimenting with the learning rate and buffer size, we proceeded with the following hyperparameter configuration:

Component	Value
learning rate	.0001
batch size	64
update learner parameters every	10 steps
memory buffer size	100000
copy to target every	300 steps
ϵ	.9 up to 1 over training
γ	.97

Figure 1. LQA Agent model configuration

3.3. Deep Q Network Agent

Our first attempted agent is the Deep Q-Learning agent, a neural approximation to the Q-learning algorithm which will serve as our deep learning baseline (Mnih et al., 2013). In order to stabilize learning, we performed parameter updates on a learner network and produced actions as well as next state value estimations using a target network. The target network was refitted with the learner networks parameters every $\{300, 1000\}$ transitions, and we found better results on the lower end of this range. The pseudocode 1 of our Q-Learning Algorithm can be found in the appendix.

In order to simulate play against another player in the environment and to train in a fashion similar to that of Neural Fictitious Self Play, we train against another DQN agent.

As Q-learning is typically suited for a one player or perfect information environment, we expected the policy produced by the DQN to be quite exploitable, especially when training against another DQN agent. We hypothesized that the Q-learning agents will continually try to learn state-action values relative to what the other player is doing during the training process, and in doing so the policies will diverge from Nash equilibria. As we plan on plotting learning curves as reward against a random agent, we believe that the reward will oscillate, representing the two DQN agents continually trying to learn best responses to one another.

We experimented with the number and size of layers, learning rate, number of steps between updates, how often to copy target parameters, and other parameters. We found that the model was quite sensitive to learning rate, requiring a very low learning rate as compared to the rest of the models, especially when performing updates more frequently. We finally decided on the following architectural configuration and hyperparameters as it yielded the most stable learning curve when being trained against the random agent.

Component	Value
Feed-forward hidden layers	$[512, 512, 512]$
learning rate	.0001
batch size	64
update learner parameters every	10 steps
memory buffer size	100000
copy to target every	300 steps
ϵ	.9 up to 1 over training
γ	.97

Figure 2. DQN Agent model configuration

3.4. Deep Recurrent Q Network Agent (DRQN)

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. As previously stated, cards are represented to the agent as a bag-of-cards, which we may be able to better represent to the model through recurrence over states. 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. In the bag-of-cards state representation, the model is not able to distinguish between when which community card was revealed, which we attempted to solve through recurrence.

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 by algorithm 2.

Similarly to the previous DQN agent, since we train recurrent Q-learning agents against both random agents and other recurrent Q-learning agents, we still expected the model to be exploitable in both cases, but still learn some general strategies as they are not able to properly model the two player game space.

We experimented with the number of layers, hidden state size, architecture of the fully connected output layers, and other parameters. Below is our final hyperparameter configuration. In particular, we similarly found this model to be quite sensitive to the learning rate especially with the frequent updates. Less frequent updates allowed the model to gather more variation in the memory buffer between updates of the model.

Component	Value
Number of recurrent layers	1
Hidden size	128
Feed-forward hidden layers	$[256, 512]$
learning rate	.0001
batch size	64
memory buffer size	100000
update learner parameters every	10 steps
copy to target every	300 steps
ϵ	.9 up to 1 over training
γ	.97

Figure 3. DRQN Agent model configuration

3.5. Neural Fictitious Self-Play Agent (NFSP)

We implemented a Neural Fictitious Self Play model to better learn strategies in a two-player imperfect information environment. This model is comprised of an action value to represent best response policy at a given time during the training period, as well as an average policy network to represent the average of these best response strategies. The policy of the action value network is expected to converge to an approximate Nash equilibrium over the training period, thus we expected this agent to be less exploitable than the previous DQN agent. This agent is trained by playing another NFSP agent, hence the name self-play. The training loop for this model is outlined below, and we similarly make use of a learner and target network to estimate state-action values. We proceeded as described by algorithm 3.

The hyperparameter configuration and architecture for this model are as follows:

Component	Value
Average policy net layers	[512, 5124, 512, 512]
Action value net layers	[512, 512, 512, 512]
Average policy net lr	.005
Action value net lr	.06
Batch size	256
RL memory size	30000
SL memory size	1000000
Copy to target every	1000 steps
ϵ	.92 up to .99 over training
η	.2
γ	.99

Figure 4. NFSP Agent model configuration

3.6. Recurrent Neural Fictitious Self Play Agent (RNFSP)

We formulated a recurrent version of the Neural Fictitious Self Play algorithm to induce our bias of sequentially dependent gameplay in Poker. We replaced the previous fully connected action value network and average policy network with recurrent networks which take in one state at each timestep, with hidden states being reset after each hand. We proceeded as described by algorithm 4.

The hyperparameter configuration and architecture for this model are as follows:

Component	Value
Number of recurrent layers	1
Hidden size	512
Average policy FF layers	[1024, 512, 512]
Action value FF layers	[1024, 512, 512]
Average policy lr	.002
Action value lr	.03
Batch size	64
RL memory size	40000
SL memory size	500000
Copy to target every	1000 steps
ϵ	.92 up to .99 over training
η	.2
γ	.99

Figure 5. RNFSP Agent model configuration

4. Analysis

In the below analyses of our implemented model, we plotted a learning curve in the form of the reward obtained against play against a random agent, as it is more interpretable than a loss plot over the training period especially when playing against another deep learning agent. Reward in this game is measured as average number of big blinds won per hand over an average of 100 hands per plotting point. We believe that higher rewards against the random agent represents knowledge of the game at a simply probabilistic level, without taking strategies into account. We then made comparisons between models by having them play each other in a high number of hands, and provided some intuition for the results we yielded.

4.1. Linear Q-Learning Approximation (LQA)

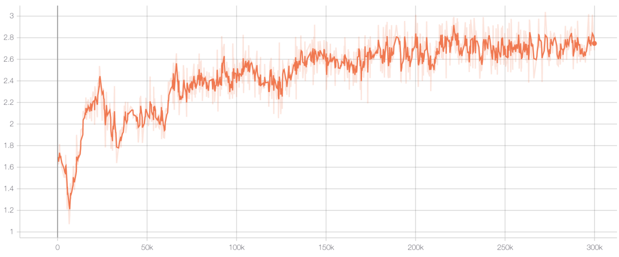

Figure 6. Reward of Linear Agent against a random agent over the training period

We see that the linear Q-learning approximation agent is able to learn quite well when playing against another LQA agent. Rather than having a noisy rewards curve like the DQN and DRQN agent below, we actually see quite a steadily inclining curve similar to that of our NFSP models.

4.2. Deep Q-Learning (DQN)

Below is a plot of the reward of the DQN agent against the random agent while training against another DQN agent.

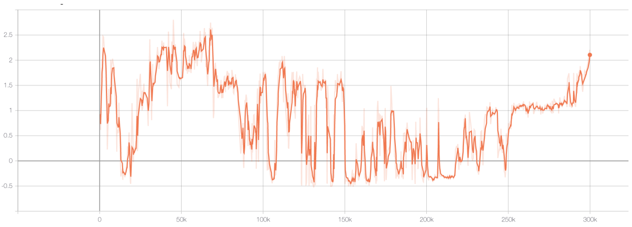

Figure 7. Reward of DQN Agent against a random agent over the training period

As predicted, the plot is quite noisy over the training period since each of the agents is trying to learn best responses towards each other. The best response to the other agent at any given time is the policy being played against the random agent, which may not reflect the general probabilistic rules of the game (which we would expect to do well against the random agent). It is interesting that, as also shown with the deep recurrent network below, that there does seam to be some steady increase in the reward prior to the oscillations occur in the loss plot between high and low rewards against the random agent. This is likely a result of both DQN agents learning the rules of the game first (which cards are high and low valued) before trying to exploit one another at the expense of diverging from these general probabilistic rules.

4.3. Deep Recurrent Q-Learning (DRQN)

Below is a plot of reward of the Deep Recurrent Q-Learning agent against the random agent while training against another DRQN agent.

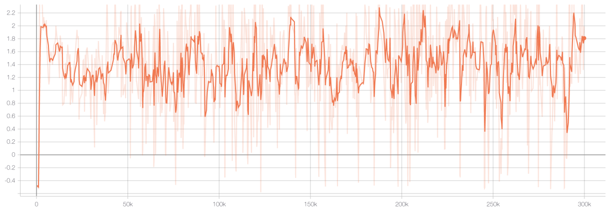

Figure 8. Reward of DRQN Agent against a random agent over the training period

We note similarities to the deep q-learning curve in the oscillations of reward. As stated previously, we believe these oscillations to be the result of continually trying to learn best responses to the other agent.

4.4. Neural Fictitious Self-Play (NFSP)

We see that the Neural Fictitious Self-Play agent's average policy over the training period is much more stable and seems to converge to a value of 2 big blind per hand over the training period against the random agent. A plot of the reward against the random agent over the training period is shown below. Note that the NFSP agent uses its average policy network rather than its current best response (from the action value network) during evaluation.

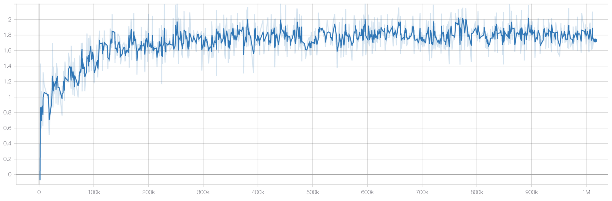

Figure 9. Reward of NFSP Agent against a random agent over the training period

We notice a much smoother inclining learning curve, reflecting the average policy network being able to slowly incorporate new best responses learned during the training period. In making comparison to the deep Q network, the deep Q network shows what a learning curve of the best response network of the NFSP agent may look like in the training period when not taking the average policy into account. The average policy network allows the agent to incorporate the evolving best responses over time to be able to learn properly in the multi-player setting. We note that (Heinrich & Silver, 2016) train the model for durations in the order of millions of updates, whereas we were only able to feasibly train for 1 million episodes. Our model showed signs of convergence, however probably nowhere near the duration required to converge to Nash equilibria. Training of this model took quite a bit longer than the DQN agent, which makes sense as we have to train two networks simultaneously, four in total including the agent we are training against!

4.5. Recurrent Neural Fictitious Self-Play (RNFSP)

Below is a plot of the reward of our recurrent Neural Fictitious Self Play model against the random agent over the training period. Note that the RNFSP agent uses its average policy network rather than its current best response (action value network) during evaluation. We notice similarities to the NFSP model in that the learning curve is much more stable compared to the Q-learning models, which again makes sense as the average policy network is able to learn the average of the best response behavior over the training period, as opposed to reflecting the current best response to the opposing agent.

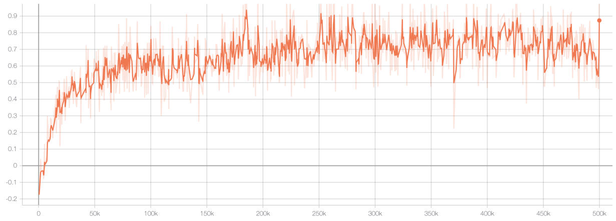

Figure 10. Reward of RNFSP Agent against a random agent over the training period

The recurrent NFSP model took much longer than any of the other models we tried, due to the updates being so computationally expensive. Additionally, we have to update both the average policy network and action value network for both agents, which adds to the overhead. This model likely requires an even longer time to converge than NFSP due to the shared recurrent parameters. Though the learning curve of our agent was inclining throughout the training period, it increased at a much slower rate than that of NFSP.

4.6. Policy Visualizations

Below is a heatmap showing the DQN valuation of different starting hands in terms of Q-values. The correspondence between axis labels on the plot and cards are shown below. Note that it is impossible for the agent to get two of the same exact card, so we set the diagonal equal to the minimum q-value for ease in visualization.

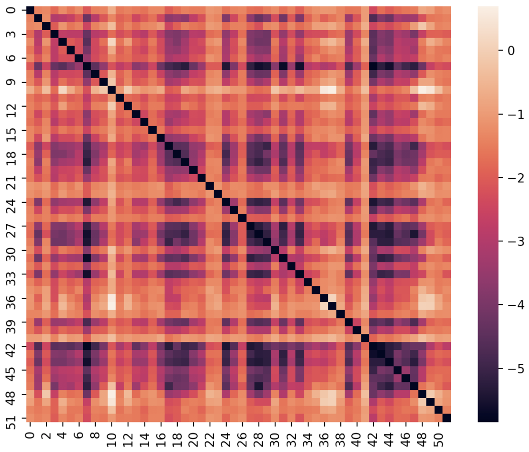

Figure 11. Heatmap visualization of DQN q-values

Analyzing the heatmap above, we notice white blocks (denoting high q-values) near pocket pairs, signifying that the agent was able to pick up on the fact that these were of high value. In particular, we see high values near (10, 49) and (10, 32) denoting a pair of Jacks being of high value to the agent. Moreover, the agent has high values with hands like Jack and Queen or Jack and 10 together, denoting that the agent was able to learn more complex patters like straight potential (five card values in a row). It also seems that the agent has learned to like high valued cards in general, as the orange stripes throughout the plot tend to happen at high valued cards.

In analyzing its failures, though, we see that the model had difficulty learning that a hand with the same suit was valuable, but this is likely due to the fact that flushes (five of the same suit) are quite rare compared to other patterns. We show the learned policy of the NFSP agent in the heatmap below.

It seems that the NFSP agent may have been starting to pick up on some similar patterns seen in the DQN plot, but no clear patterns at all. This is evidence that our NFSP model requires a much longer training period to converge to Nash equilibria, or any useful policy at all.

Similarly, we provide a policy visualization for our RNFSP agent below.

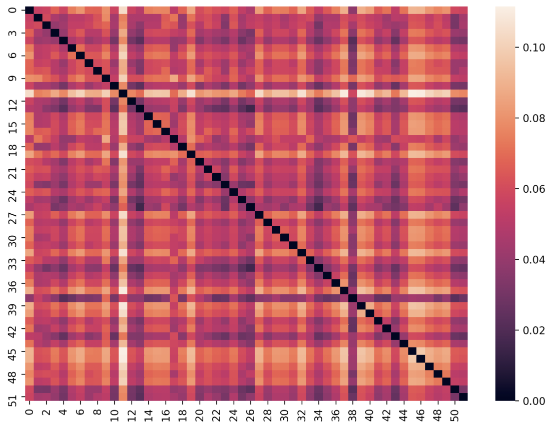

Figure 12. Heatmap visualization of NFSP q-values

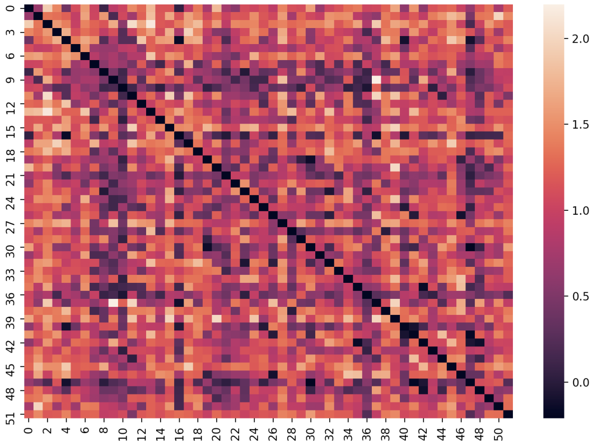

Figure 13. Heatmap visualization of RNFSP q-values

Again, the policy of this network seems quite random. We do notice some better patterns than in the case of NFSP, though. In particular, we notice lighter blocks and stripes on some of the higher valued cards similar to the orange stripes we saw in the DQN visualization. We do not notice too many patterns corresponding to pairs or suits, though.

We provide a comparison between all our attempted models in the section below.

4.7. Model Comparison

Figure 14 shows a table of the performance of our models against each other and the random agent.

Overall, we find the neural fictitious self play models to perform worse than the rest in head to head matchups, as well as against the random agent. We believe this could be the case for a number of reasons. Predominantly, we believe that the NFSP models were undertrained. Since NFSP agents are training against one another, it is possible that there was not enough noise in the agents buffers to learn how to play against many other strategies in the game. NFSP models take a long time to converge, likely for this reason, which would explain performance against the various other agents. On the other hand, the Q-Learning agents learn against another agent trying to for best responses to their own actions. This may allow the Q-learning agents to learn against various best response strategies produced by the other throughout the training period, rather than being trained against an average policy like NFSP. We then hypothesize that NFSP learns the game very slowly, which explains the low peaking, though steadily inclining, learning curves and that it is not able to properly defend against the more complex strategy learned by the Q-Learning agents through a noisier training.

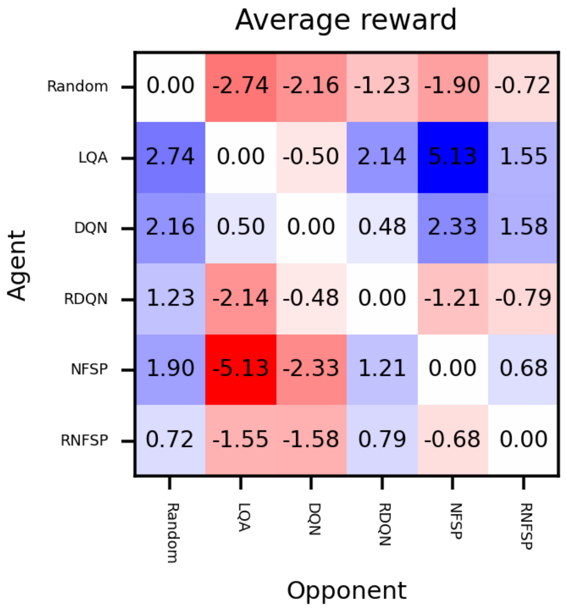

Figure 14. Model comparison - average reward per hand over 10,000 hands. For agents playing against themselves, we impose the theoretical value, which is 0 in zero-sum games

Moreover, it seems that our RNFSP agent learns even slower than the NFSP agent, with a much less steeply inclining learning curve and a lower final peak. We see that the RNFSP agent might be able to generalize a bit better than the NFSP agent though, as it performs better against the other agents and is close to NFSP in a head to head matchup.

Our linear model performed better than expected, and was able to even beat out the NFSP models that we trained. Since the linear model is essentially a simplified Q-network, this can similarly be explained by the difference in the NFSP and Q-Learning process especially over shorter training periods. Additionally, it is possible that due to the underparameterization in this model, it had a more clear learning objective than that of NFSP and RNFSP with more convoluted learning procedures that take longer to converge.

4.8. Q-Learning vs. Neural Fictitious Self-Play

As stated previously, the addition of the average policy network to the DQN architecture essentially describes the architecture of the NFSP model. We can see that the addition of this network more properly models learning in the two player environment, by averaging best responses over time rather than always acting according to the best response as is DQN. In this way, NFSP is a sort of extension of DQN to the multi-player setting as it adds another layer of abstraction to manage the various learned responses, rather than the single learned best response by Q-value maximization in the single player perfect information setting.

4.9. Recurrent vs. Fully connected models

Our recurrent models seem to perform quite a bit worse than their non-recurrent counterparts. It is possible that the recurrence caused too much focus on hidden states rather than the input states at a given time, convoluting the learning objective of trying to learn the value of states. Moreover, it is likely that the recurrence causes the models to take even longer to converge, which explains the low peaking, slowly inclining graph seen in NFSP.

We see that the RNFSP agent is able to perform better against some of the other models than the NFSP agent, which we believe might be some evidence showing that RNFSP is able to learn general strategy better than NFSP. The policy visualizations seem to confirm this as we find more sensical patterns in the RNFSP policy than the NFSP policy, even though NFSP was able to win. With proper training for a long period of time and tuning, we believe the RNFSP model would converge similar to how the NFSP model would. The only difficulty is that NFSP already requires a large number of steps to converge, and the recurrence only adds to it.

5. Discussion

5.1. Future Work

One component of this project which we explored early on is incorporating Monte Carlo Tree Search (MCTS). MCTS is a heuristic search algorithm which selects the best action out of a set of legal actions by playing out games to completion from the current game state and backpropagating the reward of the terminal game state to nodes to determine the value of different sequences of actions. It has been used with great success in imperfect information board games. We completed an implementation of the basic MCTS algorithm, but since the RLCard environment has many different pieces of state which are all necessary to step the environment and to calculate rewards, we were unable to copy all of these states correctly to be able to do the random playouts. Therefore, further work for this project might include completing this MCTS agent so that our existing models could be evaluated against a better player than the Random Agent which was used, and also attempting to combine the reward calculations of MCTS playouts with the Q values from a DQN or DRQN model to make more informed selections of actions.

Additionally, we observed through the policy visualizations that our self-play models performed worse than their q-learning counterparts, likely due to undertraining and low noise in the memory buffers during self-play. Therefore, additional future work could include training the NFSP and RNFSP models for longer after determining how to add more noise to the agent's memory buffers, and then performing policy visualization on the final models to determine how well they are able to pick up the importance of the high valued cards as compared the DQN and DRQN models.

5.2. Conclusions

This paper introduces recurrence to popular reinforcement learning frameworks in order to induce the bias of sequential gameplay in Limit Hold-Em poker. We find that the addition of recurrence to our models causes convergence to take longer and was not very fruitful in our experimentation. We also note that the Neural Fictitious Self Play models have difficulty learning quickly in this setting, which is expected when applying the algorithm to games with larger state spaces. Recurrence only added to these convergence issues, and we see a flatter learning curve in this case.

Through our policy visualizations, we are able to reflect on what exactly our models were able to learn during the training period. We find that our Q-learning agent was able to pick up on some patterns like high cards and high pairs, whereas our NFSP models did not show clear patterns in their policies.

In our analysis, we provide intuition as to why we thought our Q-Learning models performed the best. We reflect on the fact that Q-learning experiences more noise during the training process by playing against another Q-learning agent, and how this may have helped it learn in this setting. Though our recurrent methodologies did not yield particularly good results in this case, we still believe it may be a fruitful area of future work in sequentially dependent games, especially with longer training periods and potentially larger architectures. By incorporating it with a learning framework like MCTS, we may be able to more efficiently search over the sequential dependencies in the game space, similar to a beam search in NLP methods.

6. Acknowledgements

We would like to thank our course instructors Dr. Konrad Kording, Dr. David Rolnick, and Sadat Shaik for their guidance throughout the semester, as well as our project TA Vatsal Chanana for his help on our project. Additionally, we would like to thank the developers of the RLCard platform for creating a great environment for learning imperfect information card games (Zha et al., 2019).

References

Abbeel, P., Coates, A., Quigley, M., and Ng, A. Y. An application of reinforcement learning to aerobatic helicopter flight. In *Advances in neural information processing systems*, pp. 1–8, 2007.

Bowling, M., Burch, N., Johanson, M., and Tammelin, O. Heads-up limit hold'em poker is solved. *Science*, 347 (6218):145–149, 2015.

Brown, G. W. Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 13(1):374–376, 1951.

Brown, N., Sandholm, T., and Machine, S. Libratus: The superhuman ai for no-limit poker. In *IJCAI*, pp. 5226–5228, 2017.

Fudenberg, D., Drew, F., Levine, D. K., and Levine, D. K. *The theory of learning in games*, volume 2. MIT press, 1998.

Gu, S., Holly, E., Lillicrap, T., and Levine, S. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pp. 3389–3396. IEEE, 2017.

Hausknecht, M. and Stone, P. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*, 2015.

Heinrich, J. and Silver, D. Smooth uct search in computer poker. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

Heinrich, J. and Silver, D. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016.

Lazic, N., Boutilier, C., Lu, T., Wong, E., Roy, B., Ryu, M., and Imwalle, G. Data center cooling using model-predictive control. In *Advances in Neural Information Processing Systems*, pp. 3814–3823, 2018.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575 (7782):350–354, 2019.

Zha, D., Lai, K.-H., Cao, Y., Huang, S., Wei, R., Guo, J., and Hu, X. Rlcard: A toolkit for reinforcement learning in card games. *arXiv preprint arXiv:1910.04376*, 2019.

Appendix

Appendix A: Algorithms

Algorithm 1 Deep Q-Learning

```
Initialize a circular memory buffer $M$
Initialize a learner value network $L$
Initialize a target value network $T$
for episode 1...n do
    Get initial state $s_1$ of hand
    while state $s_t$ is not terminal do
        Act according to $T$ with probability $\epsilon$, otherwise choose random action. Let $a_t$ be the selected action.
        Take action $a_t$ 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 $M$
        Sample transitions $\{s_j, a_j, r_j, s_{j+1}\}$ from $M$
        if $s_{j+1}$ is terminal then
            $y_j = r_j$
        else
            $y_j = r_j + \gamma \max_{a'} T(s_{t+1}, a')$
        end if
        Update $L$ using MSE Loss on the computed labels from target net
    end while
    Periodically update $T \leftarrow L$
end for
```

Algorithm 2 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 $s_t$ is not terminal do
        Act according to $T$ with probability $\epsilon$, otherwise choose random action. Let $a_t$ be the selected action.
        Take action $a_t$ 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 $T \leftarrow L$
end for
```

Algorithm 3 Neural Fictitious Self-Play

```
Initialize a circular memory buffer $M$
Initialize a memory reservoir $R$
Initialize a learner action value network $L$
Initialize a target action value network $T$
Initialize an average policy network $P$
for episode 1...n do
  Set the policy regime $\pi$:
  $\mathbb{P}\Big(\pi = \text{avg}\Big) = \eta, \mathbb{P}\Big(\pi = \text{grd}\Big) = 1 - \eta$
  Get initial state $s_1$ of hand
  while state $s_t$ is not terminal do
    if $\pi == \text{avg}$ then
      Draw a random action $a_t$ from the distribution generated by $P$.
    else if $\pi == \text{grd}$ then
      Act according to $T$ with probability $\epsilon$, otherwise choose random action. Let $a_t$ be the selected action.
    end if
    Take action $a_t$ 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 $M$
    if $\pi == \text{grd}$ then
      Store response $\{s_t, a_t\}$ in $R$
    end if
    Sample transitions $\{s_j, a_j, r_j, s_{j+1}\}$ from $M$
    if $s_{j+1}$ is terminal then
      $y_j = r_j$
    else
      $y_j = r_j + \gamma \max_{a'} T(s_{t+1}, a')$
    end if
    Update $L$ using MSE Loss on the computed labels from target net
    Sample responses $\{s_t, a_t\}$ from $R$
    Update $P$ using CrossValidationLoss on the sampled responses
  end while
  Periodically update $T \leftarrow L$
end for
```

Algorithm 4 Recurrent Neural Fictitious Self-Play

```
Initialize a circular memory buffer $M$
Initialize a memory reservoir $R$
Initialize a learner recurrent action value network $L$
Initialize a target recurrent action value network $T$
Initialize a recurrent average policy network $P$
for episode 1...n do
  Set the policy regime $\pi$:
  $\mathbb{P}\Big(\pi = \text{avg}\Big) = \eta, \mathbb{P}\Big(\pi = \text{grd}\Big) = 1 - \eta$
  Reset hidden of $T$, $L$, and $P$
  Get initial state $s_1$ of hand
  Initialize list for sequence of transitions A
  while state $s_t$ is not terminal do
    if $\pi == \text{avg}$ then
      Draw a random action $a_t$ from the distribution generated by $P$.
    else if $\pi == \text{grd}$ then
      Act according to $T$ with probability $\epsilon$, otherwise choose random action. Let $a_t$ be the selected action.
    end if
    Take action $a_t$ 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 $A$
    Sample responses $\{s_t, a_t\}$ from $R$
    Update $P$ using CrossValidationLoss on the sampled responses
  end while
  Store sequence of transitions $A$ in $M$
  if $\pi == \text{grd}$ then
    Store responses from $A$ in $R$
  end if
  Sample episodes $E$ from $M$
  Reset $T$ and $L$ hidden
  for $\{s_j, a_j, r_j, s_{j+1}\}$ in $E$ do
    if $s_{j+1}$ is terminal then
      $y_j = r_j$
    else
      $y_j = r_j + \gamma \max_{a'} T(s_{t+1}, a')$
    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 $T \leftarrow L$
  Sample episodes $E'$ from $R$
  Reset $P$ hidden
  Input sequence of states to $P$ to get output at each timestep of $E'$
  Update $P$ using CrossValidationLoss on responses from $E'$
end for
```