PokerPal AI

Andrew Dettmer (apd67), Shivali Halabe (sbh94), Ian Paul (ijp9)

GitHub Repository and Video Demo

1 Introduction

In July 2019, an artificial intelligence (AI) program defeated multi-time World Poker Tour winners in No Limit Texas Hold'em poker. The first AI of its kind, it was especially important in solving real-world issues, due to the applications of poker as a game of incomplete information. As summarized by Facebook CEO Mark Zuckerberg, poker involves "multiple adversaries at once in environments where we have imperfect information about their resources or what they're doing," which is emulative of more drastic situations in which AI could play a pivotal role in the future.

This paper will explore different reinforcement learning (RL) strategies for training such poker bots. The AI mentioned above, created for two-player No Limit Texas Hold'em by Carnegie Mellon University in collaboration with Facebook AI, utilized self-play algorithms based on Counterfactual Regret Minimization (CFR) combined with search procedures for imperfect information games. However, CFR is an incredibly well-known and widespread strategy for solving complex games, so we decided to investigate two novel RL algorithms that incorporate self-play in ways different from CFR.

2 Algorithm Overview

As a brief summary, reinforcement learning is a type of machine learning where an algorithm possesses no training data at its onset. The goal is for some type of agent trained with this algorithm to evolve and learn from its own experiences in an environment– a state which is reached based on actions taken thus far. For an RL algorithm to work, the environment must be computable and maintain some sort of reward function that evaluates how

well an agent is performing. Learning like this can be easily applied to games, because states are clearly defined by the program, rules can be modeled by preconditions or constraints, and there is a known task to accomplish (a score or award metric).

2.1 Deep Q Network

Learning with Deep Q networks (DQN) builds off the Q learning algorithm, whose main element is its Q-table: a 2-dimensional matrix mapping a state with a corresponding Q-value for each action taken. The algorithm aims to iteratively find the optimal Q-table by maximizing the Bellman equation, which states that the Q-value for a state and action is equal to the immediate reward plus a discounted future reward. From this, the algorithm obtains a Q learning function, which is then used to update the Q-table.

Because the Q learning algorithm is a greedy algorithm, it always chooses the action that yields the best Q-value for the current network; but different Q-values are randomly generated at the algorithm's start, which can cause issues for learning. For instance, the algorithm might think that bad actions will get the most reward. Such cases are avoided in practice by integrating a random parameter that forces the agent to randomly choose its action for a certain percentage of the time. This percentage can be decremented over time to make the algorithm converge to the optimal Q-table.

In games like poker, there can be an extremely large number of states, making the table-mapping aspect of Q learning impractical. Deep Q Learning was introduced in 2013 as a solution to this problem. Essentially, this strategy extends the traditional Q learning algorithm by replacing the Q-table with a Q-network– a neural network used to approximate the Q-value for a given state. The Q-network takes the state of the environment as input and outputs the predicted Q-value of each action. DQN enables scaling of the algorithm to more complex environments, like those required for poker, since the number of states does not have to be predefined.

2.2 Neural Fictitious Self-Play

Fictitious Play is an iterative method that finds Nash equilibria in two-player zero-sum games. However, it is only applicable to normal form games– games that are tabular and do not capture the concept of time. With this in mind, researchers developed Fictitious Self-Play, an extension of Ficti-

tious Play that utilizes sequential extensive form games as its basis. Using reinforcement learning in conjunction with game theoretical principles, Fictitious Self-Play finds an approximation of the best response to the other player's strategy based on player beliefs; then it uses supervised learning to average several best responses into a strategy. Fictitious Self-Play is proven to converge to a Nash equilibrium.

Neural Fictitious Self-Play (NFSP) incorporates a neural network into Fictitious Self-Play. It uses two independent networks, one of which is actually a Deep Q network, and two memory buffers assigned to each of the networks. The buffer assigned to the DQN uses reservoir sampling– this involves sampling a stream of data such that the probability of choosing a data item is proportional to the length of the stream, even though the length is unknown and the memory size is limited to less than the length, so not all stream data can be buffered. This reservoir stores the best response in terms of states and actions at specific time steps. The other buffer stores agent experience.

The non-DQN is a neural network used to predict action values from agent experience in its corresponding buffer using reinforcement learning. It approximates the best response with an ϵ-greedy strategy. The DQN maps states to action probabilities and defines the agent's average strategy; essentially, it attempts to reproduce the best response using its history of best response behavior in its buffer. Thus, the algorithm maintains two neural networks to predict actions: the non-DQN by using past experience and the DQN by using best response history.

The agent must choose which of these networks to use in selecting an action at every turn. It does this by using a parameter which enables qualitative changes in learning stability by introducing an anticipated correction to the action. The correction parameter determines the probabilities with which the agent uses each network. In particular, this anticipated correction is the advantage of NFSP which supposedly allows for convergence to Nash equilibria in imperfect information games.

2.3 Effective Hypothesis

Comparing these two algorithms, we hypothesized that NFSP would yield better results in small poker matches than would DQN. The reasoning for this was that NFSP is based on the concept of a DQN but incorporates an additional neural network and correction parameters for its DQN's predictions,

ultimately making it seem more reliable in convergence to a Nash equilibrium. Its ability to stabilize its choices and average over its best responses seemed to give it a theoretical edge. In practice, we coded and trained working models of both algorithms for two-player Limit Texas Hold'em. Then we allowed our models to compete in tournaments against each other, as well as other random agents, for direct evaluation of results.

3 Modeling with RLCard

For training and evaluating the performance of our NFSP and DQN models, we utilized the Python package RLCard. The following excerpt is taken from the source documentation:

> RLCard is a toolkit for Reinforcement Learning (RL) in card games. It supports multiple card environments with easy-to-use interfaces. The goal of RLCard is to bridge reinforcement learning and imperfect information games and push forward the research of reinforcement learning in domains with multiple agents, large state and action space, and sparse reward. RLCard is developed by DATA Lab at Texas A&M University.

RLCard includes many built-in tools for developing game-playing algorithms, including agents for DQN and NFSP. We also leveraged the TensorFlow package, as instances of DQN and NFSP agents are built upon neural networks and require TensorFlow for training and program run-time execution.

3.1 Limit Texas Hold'em Representation

In Texas Hold'em Poker, each player is dealt two face-down "hole" cards. Then five "community" cards are dealt in three stages. Each player seeks the five best cards among the hole cards and community cards. There are four betting rounds, and in each round, players can choose an action to pursue. In the RLCard package, these actions are limited to $\in$ {`call, raise, fold, check`}.

For Limit Texas Hold'em Poker, each player can only choose a fixed amount of raise. In each round, the number of raises is limited to four. Within the source package, a game state is encoded as a vector of length 72. The first 52 elements represent cards, where each element corresponds to one card.

The hand is represented as the two hole cards, plus the observed community cards that have been dealt thus far. The remaining 20 elements are the betting history.

Given that a "big blind" in poker is a certain amount a player must deposit into the pot before seeing any cards when dealt a specific chip, the reward in RLCard is calculated based on big blinds per hand. For example, a reward of 0.5 means that the player wins 0.5 times of the amount of the big blind, and a reward of -0.5 means that the player loses 0.5 times the amount of the big blind.

4 Training and Evaluating Models

Training of our models is performed by `dqn.py` and `nfsp.py`. For each algorithm, we trained three separate models under various constraints. The primary constraint is the `episode_num`, which specifies the total epochs of training. For each model, we trained a heavily constrained model with 1,000 epochs, moderately constrained with 10,000 epochs, and a relatively unconstrained model with 30,000 epochs.

Throughout training, we also maintain a secondary environment for tracking improvements in performance in our bot. At regular intervals, a tournament is run against a random agent, and we record the average reward. In general, we should expect training to improve the average reward of the model before converging. A png file is produced that gives a visual indicator of the trend in training rate by plotting the average reward over the environment time-steps.

In order to integrate our trained models with the RLCard framework, several steps needed to be taken. First, we migrated the folders containing raw model data from a local directory we made into the RLCard package directory. This allowed us to then register our trained model within the context of the package, which is necessary in order to load the model for use in tournaments involving the agents we train. Model registration occurs in the `__init__.py` file of the `rlcard/models` directory. In order to properly register a model, this `__init__.py` file must call the `register()` method with two arguments: the "model identifier", which is a descriptive name we give to our model, and the "entry-point" of the model, which is a string that indicates the location of the model class, and is given by the path to the

model folder concatenated with the name of the model's wrapper class.

For each model, we were required to define a separate wrapper class in `pretrained_models.py`, which inherits RLCard's built-in `Model` class in order to perform registration. Since our models all utilized TensorFlow, our calls to `register` only occur if the machine has a version of TensorFlow installed that is compatible with RLCard. With all of these elements in place, we are able to register our model.

The registered models are able to be loaded into our tournament setting in `play.py` by calling `models.load('model_directory')`. Since we trained each model against random agents, we extract the trained agent from the model and insert it into the `agents` field of our game environment. After setting the environment agents to the desired models for the current run, we can perform a tournament of 1000 games and log the average payoff (reward) of each agent in the game for further analysis.

5 High-Level Overview of Code

5.1 `poker-pal-ai/nfsp.py`

This file trains an NFSP Agent to play Limit Texas Holdem. We first create a directory path to save logs and learning curves to. Hyperparameters we set include how frequently to train the agent, how frequently to evaluate the performance of our model, how often we want to plot the reward as the agent learns, the number of games in each tournament the agent plays, and finally how many episodes to train the NFSP agent for. We then set up an environment and do the following on each episode:

- Sample a policy for the episode
- Generate data from the environment
- Feed transitions into the agent memory and train
- Evaluate performance of agent by having it play in a tournament with random agents

We then plot the learning curve and save our trained model so we can restore it for future use i.e. to play in tournaments against other agents.

5.2 `poker-pal-ai/dqn.py`

This file trains a DQN Agent to play Limit Texas Holdem. We create a directory path to save logs and learning curves to, and include the same hyperparameters and perform similar steps to those detailed above to train the DQN agent.

5.3 `poker-pal-ai/play.py`

This file loads pre-trained DQN and NFSP models we have saved off and plays the agents and evaluates how our model performs in a tournament. We set a variable indicating how many games we want our agents to play and log the average payoffs for each agent (player) in the tournament.

5.4 `rlcard/models/__init__.py`

This file registers all of the rule-based or pre-trained models available in the package. Initially, this file contained only the sample models produced by DataLab, such as a rule-based decision-making model for Dou Dizhu. We modified this file to register our pre-trained models, each of which had been imported to the package after training.

5.5 `rlcard/models/pretrained_models.py`

This file contains all of the wrapper class definitions for the pre-trained models in `rlcard/models/pretrained`. In order to be loaded into a game environment, a model must specifically have a wrapper class definition that inherits the `Model` class. For a model that has a supportive representation in TensorFlow, the class definition must initialize a TensorFlow session and restore the TensorFlow data from the raw model data. We updated this file to include wrapper `Model` class definitions for each of our trained models, allowing us to properly load saved models later for experimentation and data analysis.

6 Training Reward Graphs

We trained both the DQN and NFSP agents over 1000, 10000, and 30000 episodes. We plotted the learning curves for each of these for a total of 6 curves. The x-axis is the episode number and the y-axis is the reward the agent achieved playing in a tournament of a specified number of games.

The duration of each tournament was set to 100 games for all runs. The 3 learning curves for our DQN models are pictured below followed by the 3 learning curves for our NFSP models:

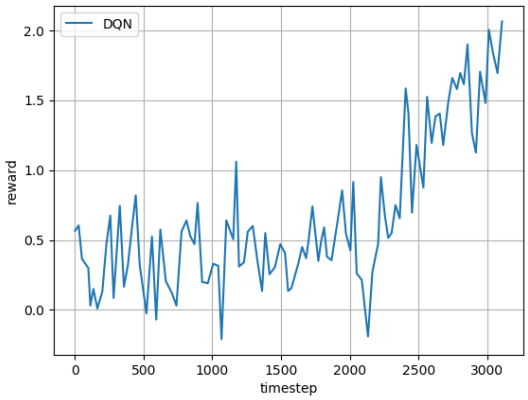

Figure 1: DQN agent v1 learning curve over 1000 episodes

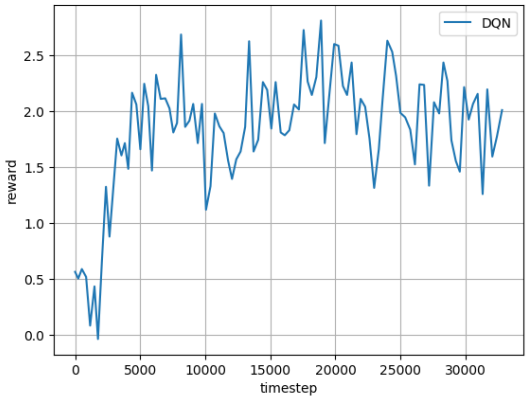

Figure 2: DQN agent v2 learning curve over 10000 episodes

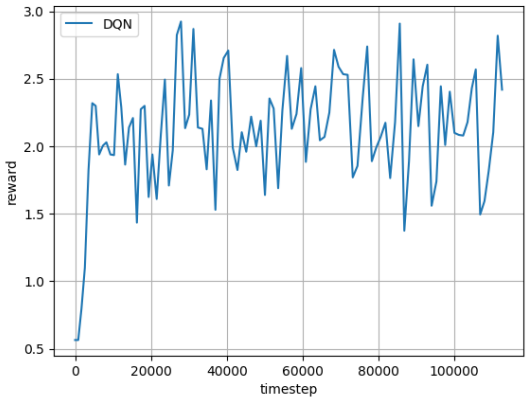

Figure 3: DQN agent v3 learning curve over 30000 episodes

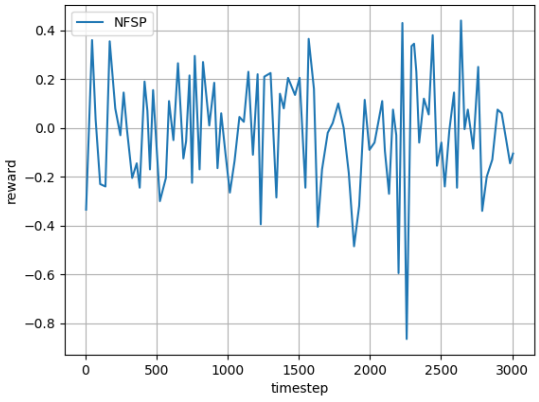

Figure 4: NFSP agent v1 learning curve over 1000 episodes

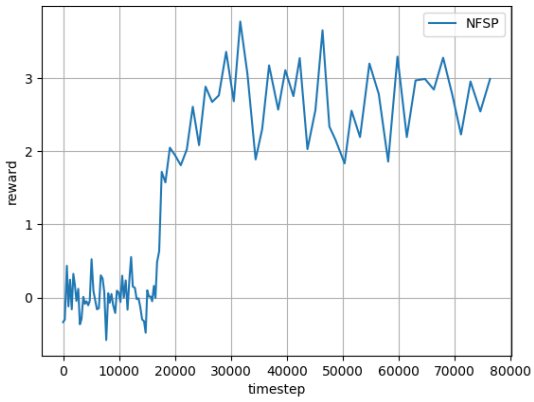

Figure 5: NFSP agent v2 learning curve over 10000 episodes

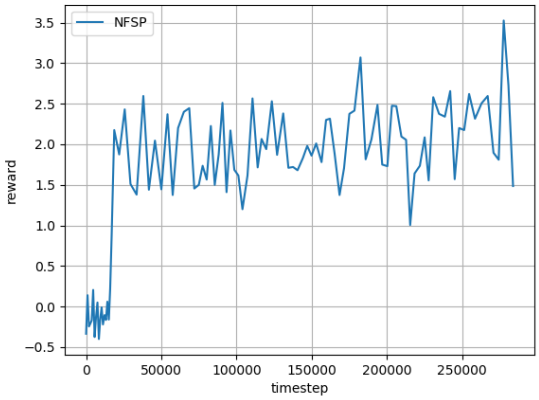

Figure 6: NFSP agent v3 learning curve over 30000 episodes

7 Analysis of Training Differences

The DQN agent does begin to learn a successful strategy to play Limit Texas Holdem within 1000 training episodes, as shown Figure 1. The NFSP agent does not learn a successful strategy at all within 1000 training episodes as shown by the variance in reward and complete lack of trend evident in Figure 4. Thus, we conclude that the DQN model has a faster learning rate than the NFSP model.

When training our agents over 30000 episodes as depicted in Figures 2 and 5 for the DQN and NFSP agent, respectively, we can see that the DQN and NFSP agent has learned a successful Limit Texas Holdem strategy as indicated by the positive reward. The reward the DQN Agent and NFSP agents achieve is around 2.0 on average. However, the DQN agent's reward has a relatively large variance in the reward as compared to the NFSP agent. Specifically, after the agent's learn a strategy, the reward range is between 1.5 and 3.0 for the DQN agent and is between 2.0 and 3.0 for the NFSP agent.

So why does the DQN agent perform worse than the NFSP agent? The problem is that DQN agents exclusively generate self-play experience according to their ϵ-greedy strategies. Theory states that DQN average behavior does not achieve a Nash equilibrium. NFSP agents use an ever more slowly changing (anticipated) average policy to generate self-play experience. Thus, their experience varies more smoothly, resulting in a more stable data distribution, and therefore more stable neural networks (Heinrich and Silver, 2015). Over 100,000 episodes, we see a similar pattern in both the DQN and NFSP agents as we do over 300000 episodes, with the DQN agent having a larger variance in reward than NFSP, but both having an average reward of around 2.0 to 2.5. Thus, our experimental results agree with theory.

8 Strategy Comparison

The primary form of evaluation of each reinforcement learning strategy was a tournament organized between each trained model. Each DQN model played 1,000 games against each NFSP model, with version 1 (v1) of each model being the one trained with the least number of episodes and version 3 (v3) being the one trained with the greatest number of episodes. We provided results in terms of the reward for the NFSP bot; the more positive the number, the better the NFSP bot fared in the face-off, and the more

negative the number, the worse it fared. The results of these tournaments can be seen in the following payoff matrix:

Model	DQN v1	DQN v2	DQN v3
NFSP v1	-1.8245	-2.0665	-2.1935
NFSP v2	0.8835	0.1195	-0.297
NFSP v3	0.447	-0.2215	0.284

As can be seen from the table, NFSP v1 performed poorly against DQN v1; and this discrepancy grew larger as the DQN models trained for longer periods of time. Similarly, the DQN v1 model fared better against NFSP v1 than NFSP v2, to which it lost on average. As both models received more training, NFSP emerged as the dominant strategy between equivalently trained models. These results agreed with the theoretical discussion in the analysis section above.

One anomaly in the results is the performance of NFSP v3 against DQN v1 and DQN v2. While NFSP v2 beats both DQN v1 and DQN v2 on average, NFSP v3 beats DQN v1 by a smaller margin and performs worse than DQN v2. We speculated that this was due to overfitting the NFSP v3 model and training on a number of episodes that was too large. However, this model still beat DQN v3, corroborating our hypothesis.

We also surmised that there was not a major difference in reward and that neither model had an extremely definitive edge over another as training increased due to the similarities between the two strategies. As described in the Algorithmic Overview, NFSP uses two neural networks– a DQN and a separate neural network. It chooses between the highest-reward actions found by each of these networks using an anticipated correction parameter. Given this information, it makes sense that our limited amount of training did not result in a significant discrepancy between the rewards of the two algorithms.

9 Further Considerations

Had we had more time to explore and experiment with NFSP and DQN, we would have tested how the model architectures of perform when playing different variations of poker, such as Limit Hold'em, Leduc Hold'em, as well as entirely different games like Blackjack. Measuring the winning rate of

these same model architectures in different game environments, where rules are completely different but there exists a common theme of imperfect information and multiple adversaries, would further highlight the usefulness of reinforcement learning models to solve a wide variety of problems that extend well beyond the world of casino games.

As mentioned in our introduction, CFR is already well-known to be a state of the art algorithm to solve complex games, which is why we decided to investigate the NFSP and DQN reinforcement learning algorithms. However, as the goal of this project was learn about the architecture, strengths, and weaknesses of these algorithms, we shifted our focus away from the absolute reward aspect used to assess the models. Further research should be done into the complexity and model training time trade-off that occurs to achieve the state of the art results with CFR, especially contrasted with the NFSP and DQN algorithms' worse but still competitive performance (as measured by a percentage of the average reward CFR achieves).

10 References

Deep Reinforcement Learning from Self-Play in Imperfect-Information Games
Applying Deep Reinforcement Learning to Poker
Facebook AI Beats Poker Pros
Neural Fictitious Self-Play
RLCard Documentation
RLCard GitHub