Building an Efficient Poker Agent Using RL

Alex Kashi
Harvard
alexkashi@g.harvard.edu

Vedang Lad
MIT
vedlad@mit.edu

Håkon Grini
Harvard
hgrini@g.harvard.edu

Abstract

Poker is one of the most widely played card games worldwide, particularly due to its unique blend of skill and luck. The psychology of a successful poker player, how a player demonstrates confidence through bets and bluffs combined with imperfect information makes poker very "human". Our group aims to create a deep reinforcement learning agent that is capable of outperforming existing poker bots to demonstrate RL's ability to "solve" non-deterministic problems. We plan to build on the existing framework Neuron_poker[1]. This framework provides an OpenAI gym-style environment for training and evaluating Poker agents. It also allows easy creation and integration of new poker "players", which we create in this project. Using this framework, we create a novel PPO and modified DQN agent that outperforms the existing agents that Neuron Poker has to offer. We also experiment with other variations of DQN, such as dueling DQN and Double DQN to enhance performance. We present a deep reinforcement learning alternative to playing poker, one that does not rely on probability tables or continuous outcome trees, purely on Reinforcement Learning.

1 Introduction

We create deep reinforcement learning poker agents based off of the state of the art algorithms that can run in real-time, without explicit calculation of probabilistic trees. Our algorithm intends to outperform tree-based methods, as well as poker enthusiasts by building intelligence through "self-play". To gauge the performance of our agents we use the return of the agent across multiple rounds of Texas Hold'em. Following this metric, we can have different agents playing against one another to show that our model is capable of outperforming opponents. We introduce some poker variations, such as Limit Texas Hold'em and various assumptions in order to realistically solve the problem with the resources available.

2 Related work

Poker has long been a hot research topic for the field of artificial intelligence, as it is a popular example of a game dealing with imperfect information, which makes decision-making more complicated and challenging. In this section we outline some previous works in the field of computational poker, specifically into Heads-Up Texas Hold'em, with an emphasis on efforts using Deep Q-Learning and Policy Gradient Methods, as these are most closely related to our experiments. We also review some of the defining literature on the methods used.

[1]https://github.com/dickreuter/neuron_poker

Preprint. Under review.

2.1 Deep Q-Learning

Reinforcement learning using Deep Q-Networks (DQN) was popularized by Mnih *et al.*, who were able to teach an agent to play Atari games by training it to predict the best actions based on the previous sequence of observations and actions [MKS$^+$13]. As the observations are represented by the pixels, the authors used a neural network to learn high level features of the observations, thereby reducing the dimensionality of each observation. With this approach they were able to teach the agent to play 7 different Atari games, surpassing a human expert in 3 of them, without any prior knowledge using only the reward from the end state to guide the training. One of their main innovations was to utilize a replay memory to store the agents experience to be able to continue learning from them when training. Sampling from this buffer randomly allows the agent to be exposed to less correlated datapoints.

Several variations of DQN have been introduced in the last few years. An example is Double DQN, first introduced by van Hasselt *et al.*. The original DQN approach is based on training using a target network consisting of a previous iteration of the network parameters, to estimate the value of the next state after an action as seen in the following equation.

$$Y_t^{DQN} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-) \tag{1}$$

Here Y_t^{DQN} represents the target value, R_{t+1} represents the reward from taking the action, and $\gamma \max_a Q(S_{t+1}, a; \theta_t^-)$ represents the discounted value of taking the action of the highest value in the next state S_{t+1} with respect to the old network weights θ_t^-. The use of the old weights is meant to separate the target from the network itself to avoid using a moving target which changes with the weights. Using the max operator to find the value of the next state does, however, tend to lead to overestimations when valuing the future states, as an overestimated action is likely to be picked, which could lead to a suboptimal policy. To avoid this issue the idea of Double DQN is to have two sets of weights where one set of weights is used to select the best action whereas the other set of weights is used to evaluate that action, according to the following equation.

$$Y_t^{DQN} = R_{t+1} + \gamma Q(S_{t+1,a} Q(S_{t+1}, a; \theta), \theta_t^-) \tag{2}$$

To avoid having to train several networks the authors suggest using the online weights to select the best action for the next state but using the target network consisting of an older iteration of the weights to evaluate the action. This approach helps reduce overestimation as it is less likely that an action that would be overvalued by the target network is chosen based on the online network than if it had been greedily chosen by the target network.

Due to poker's large state space and high dimensionality, there have been a substantial amount of effort directed into using Deep learning in combination with reinforcement learning for the game. One of the more notable ones is the DeepStack algorithm introduced by Moravčík *et al.*, which trains a model by playing itself, using search to look ahead to see potential outcomes of moves and evaluate those states using the trained DQN. Deepstack uses various heuristics to decrease the search space, and manages to defeat professional poker players with a statistical significance in heads-up no-limit Texas Hold'em.

A more recent example of a DQN based approach to poker is the Recursive Belief-based Learning (ReBeL) [BBLG20a] algorithm introduced by Brown *et al.*which provably converges to a Nash equlibrium in all two-player zero sum games, which includes heads-up no-limit Texas Hold'em [BBLG20a]. This algorithm also combines reinforcement learning with search to evaluate the states different actions would lead to, which is similar to the approach taken in state of the art algorithms for deterministic games such as AlphaZero [SHS$^+$17]. To account for the fact that the next observation cannot deterministically be determined by the current observation and action it incorporates the probabilistic belief distribution of the agents on what state they are in depending on the information they are given.

The Nash equilibrium is defined as the policy profile where no agent can achieve a higher expected value by switching to a different policy, and have been proven to exist for all finite games. The benefit of the Nash equlibrium is that if the player sticks to this policy it is guaranteed to not loose in expectation, independently of what the opponents strategy is.

2.2 Policy Gradient Methods

There have been previous efforts into using policy gradient methods for zero-sum imperfect information games, and Texas Hold'em specifically. An example of this can be found in the paper by Srinivasan *et al.*which demonstrates that a model-free agent that uses gradient-ascent to functions that approximates the discounted return of decisions, can generate policies that approximate the Nash equilibrium [SLZ+18]. In their approach they maximize the advantage from doing an action in a state compared to the mean return among all available actions, weighted by their likelyhood under the policy. This mean return is used to evaluate the state, which makes the approach a Mean Actor-Critic based one.

A recent innovation within the Policy Gradient field is the Proximate Policy Opimization (PPO) family of methods, first introduced by Schulman *et al.*[SWD+17]. This approach is intended to be scalable, data efficient and robust and is proposed as an alternative to other policy gradient methods and DQN. Their main novelty is that they propose a novel objective which uses clipped probability ratios, to form a lower bound of the policy performance. This also builds on the idea of Trust Region Policy Optimization (TRPO) introduced by Schulman *et al.*, which maximizes the expected advantage subject to a constraint stating that the weighted Kullback-Liebler divergence between the old and updated policy must be below a certain threshold [SLM+15]. This constraint is intended to ensure that policies don't change too much between updates, but makes the algorithm more complicated and harder to compute. To circumvent this issue, and also avoid adding a penalty term for the KL divergence to the optimized function, the authors instead proposes to set a bound on the ratio between the old ratio and the new one for every optimization step, ensuring that the gradient is also capped when making a step in its direction. This method requires the setting of a hyperparameter to decide how big the relative change between policies can be before clipping.

2.3 Other State Of The Art Poker Algorithms

For games with three or more players, the concept of a Nash equlibrium becomes more complicated than for the two player version discussed so far, as even if the players independently computes a Nash equilibrium the resulting list of strategies may not necessarily be a Nash equlibrium. This gives players incentive to deviate from their computed strategy. For this reason, the Pluribus algorithm introduced by Brown *et al.*does not guarantee convergence to a Nash equilibrium when playing against multiple opponents, but still manages to consistently defeat elite human players in six-player, no-limit Texas Hold'em, and was the first algorithm to do so.

Similarly to ReBeL [BBLG20a], the Pluribus algorithm is trained by playing against itself and gradually improving, by searching for better strategies in real time. To shrink the large state and action space it buckets different decision points together, and also removes some possible actions. This significantly different from ReBel [BBLG20a] which does not abstract the game at all, making it more applicable to other games than just poker. Another major difference is that Pluribus does not use deep learning but relies solely on reinforcement learning.

The current state of the art algorithm along with ReBel [BBLG20a] comes from Deepmind's "Player of Games" [SMB+21]. It uses uses growing-tree counterfactual regret minimization to recursively explore sub trees, and update parameters accordingly. The algorithm combines self-play, guided search, and game-theoretic reasoning in order to take on many games of imperfect information well. While it employs a different method as to Facebook's ReBeL and methods taken in the paper, POG tackles the problem of imperfect information with an ensemble of methods.

3 Problem Formulation

3.1 State Space

One of the inherent challenges of poker is the large state space, as there are a very large number of configurations. The state-space naturally also depends on the number of players at the table, as several components of the state are specific to each state. Examples of this are the amount of money each player has left, the position each player has in the turn, and the history of player actions. This is effectively determining how aggressive each player is, just a few techniques that poker players employ. Furthermore, one would need to keep track of the shared information about the table, such as the size of the pot, which players have folded, how much has been raised, and naturally the community cards. All of this, along with the cards that are in the current player's hand, would be part of the state.

Naturally, all these variables lead to a very large state space which makes it impossible to explore all feasible states. This motivates the possible use of a model-based state evaluator. This model would survey the current state of the game and result in a state evaluation, which would eventually be used as an input into the larger network (DQN/PPO). This is similar to the approach taken by DeepMind in their work when solving chess. A limitation of this method is extensive(and expensive) computing time, which we may not currently have access to.

For the time being, we propose limiting the state space to the following three categories and their respective states. While a model-based evaluator would be able to capture "hidden features" of poker more effectively, we only use the following states to train our model. We use "hidden features" here as a description of other subtleties of poker, for instance, a model-based evaluator may be to recognize when a player is bluffing more accurately than if there was just a normal state input.

State		
Community State	Stage State	Player State
Current position Stage Community pot Current round pot Active players Big blind Small blind Legal moves	Raises Min Call at Action Contribution Stack at action Pot at action Calls	Position Equity to river alive Equity to river 2plr Equity to river 3plr Stack

It is important to note that while the state is very large, many of the state variables only have finite values i.e the current position, the value of the blind, etc. We use one hot encoding where appropriate which helps overall convergence of the algorithm.

3.2 Action Space

The action space for poker is also very large, as one can not only fold, call and check but also raise by an arbitrary amount. To simplify this large action space, a common approach is to only allow agents to raise by discrete amounts. In our case, the amount an agent can raise is defined by the amount of money in the pot. It is possible to go all-in when the amount of cash on hand is limiting one's ability to raise these discrete amounts.

For efficient training and reducing the size of the action space, we limit the raised actions to the following. This way one can characterize the "boldness/intensity" of a given move better. A future work here would be to perhaps use a regression between the 5 different raises to characterize the "intensity" of a raise in continuous action space.

Actions		
Blind Actions	Raise Actions	Card Actions
Big Blind Small Blind	Raise 3x Big Blind Raise Half Pot Raise Pot Raise 2x Pot All In	Fold Check Call

The raise actions are ordered in the terms of the "intensity" of the move. 3x Raise of the big blind is a conventional raise while going "All In" is a move that exudes confidence (whether real or bluffed).

3.3 Objective Function

Poker as a game has an inherent benefit when it comes to objective functions, as one will get a clear indication of how good one's policy was at the end of each round by seeing how much money one lost or won that individual round. This allows for a dense reward function which makes evaluating states and actions simpler. The reward function will likely need some experimentation to find the one that works best in practice. As an alternative to simply using the money lost or won as a reward, one could scale each reward by subtracting the average reward over all possible actions for the given state.

3.4 Base Learning Algorithm

We will be investigating the effectiveness of two DQN and PPO in learning Poker, the architecture of each agent is as follows.

DQN is implemented using keras-rl2. The network contains 4 dense layers with relu activations, with a final linear layer. It is trained locally before play with the ability to control the duration of the game (Limit Texas Hold'em), with the opportunity to find the optimal batch size. We explore other DQN structures such as dueling DQN and Double DQN to measuring performance.

The second implementation utilizes a three layer PPO followed by the Tanh layer. This is implemented using the pytorch framework, as keras-rl currently does not have a PPO implementation. We train both PPO and DQN against the random agent and the equity agent, which we describe below.

3.5 What simulator are you using

For our experiments, we are using the Neuron Poker RL environment, which extends upon OpenAI's gym package allowing users to test and compare agents against each other in Texas Hold'em. This package includes a few pre-made agents such as a random agent and a Deep Q-learning agent made with a relatively shallow, dense network.

The Neuron Poker environment also has built-in functionality to test agents against real humans, which could be interesting to experiment with, to investigate how a trained agent performs against players with different styles such as players with different risk aversion.

3.6 Success criterion

There is a simple success criterion from game to game, which is simplified by the rules of poker. Whoever walks away with the pot has won the game as they have taken the chips of other players. Therefore, the rewards of all players in a game should realistically decay to zero and the winner will be the only one to make a profit.

The success criterion of our specific model is to outperform existing models that exist in Neuron poker. This is accomplished by running two agents against one another and seeing who has won after a finite number of iterations (or until someone has won).

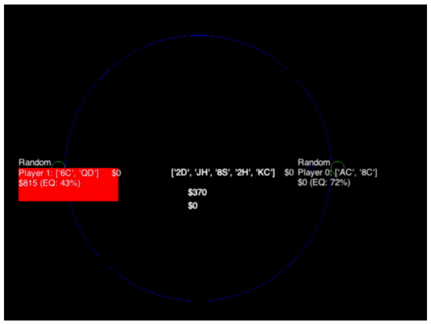

Figure 3: We visualize the Nearon Poker framework by the Poker table seen above. While here we play Heads Up Texas Hold'em with two random agent, we can pick any of the trained agents to play - even through manual keyboard inputs.

4 Results

4.1 Baselines

To get a sense of the performance of the trained models we compare the models to two separate baselines, which will be referred to as the random and the equity agents respectively.

4.1.1 The random agent

The random agent is an agent which takes no information about the current state into account, and makes a random choice among the legal moves. We consider this as the lower bound for the performance of a trained agent, because if not the agent would make worse decisions by taking state information into account. However, during the early training steps it makes sense that performance becomes worse than the random agent as being overoptimistic or overpessimistic is likely to lead to poor strategies.

4.1.2 The equity agent

In addition to the random agent we compare the models against an agent that bases its decisions on the approximated equity, which represents the likelyhood that the agent will win based on the cards on its hand and the shared cards on the table. In the framework this equity is estimated by performing 500 Monte Carlo simulations for which every simulations hands random cards to every other player and to the reimaining slots on the table, and then checks to see if the agent would win. The equity is then the agents win ratio over these simulations.

The action taken by the player is then determined by hyperparameters that sets a minimum equity for every action to be taken. The equity needed for the agent to go all in is naturally the highest, while the equity required to call is the lowest.

Although this agent is better than the random agent it has a set of shortcomings. One problem is that it becomes sensitive to the choice of hyperparameters, which dictates the policy. It is also an issue that the actions taken by the agent reveals a lot of information about the players hand which can be easily exploited. For instance, if the agent needs a 50% chance of winning to raise, but only a 30% chance of winning to call, then the action of calling indicates to the opponent that the player is not very confident in its chances of winning. The equity agent is also unable to take the opponents actions into account which naturally is crucial as these actions reveal something about the opponents confidence in its cards.

The equity agent therefore does not produce a policy according to a Nash Equlibrium, and should be possible for a properly trained agent to beat, but it does represent a policy that is substantially more hard to beat than that of the random agent.

4.2 Training

We train all methods PPO and DQN and its variations on both the random and equity agent which serve as a baseline. The intelligent baseline being equity agent while the unintelligent baseline is the random agent, however explores the space to a greater extent than the equity agent. Figure 1 below illustrates the training for all of the agent, the first column being the equity agent and the second column being the random agent. We see the advantages of the random agent in training, as it is able to explore more of the space and attain a higher reward than the intelligent equity agent. Since the equity agent is more intelligent, we see that the model has a smaller reward than the random agent.

We also notice that regardless of the variation of DQN, as seen in the bottom right, all methods are within variance of one another. This reveals a shortcoming not of the type of algorithm, but a limitation of DQN itself. We also find that, as expected, longer iterations lead to higher episode rewards as limited by our computation abilities.

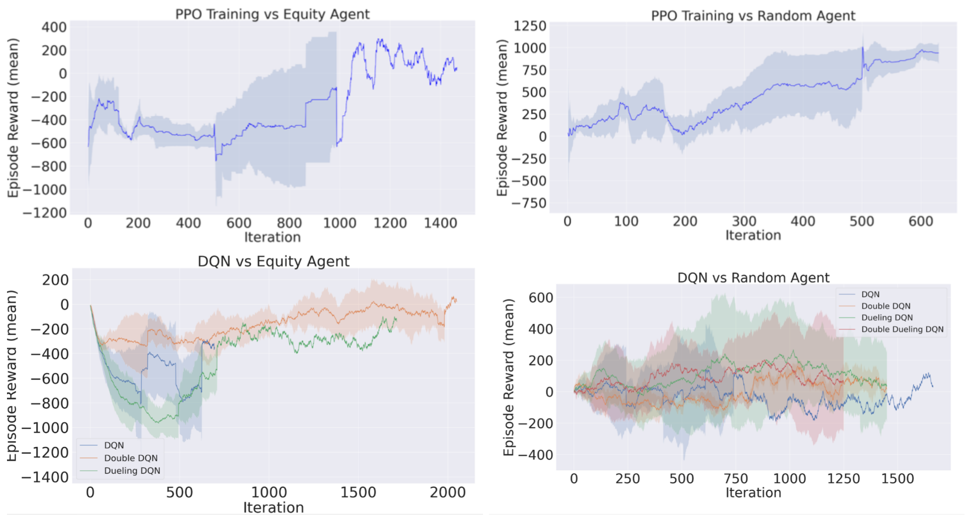

Figure 1: Top left: We have a PPO agent trained against an equity agent, an agent that acts more intelligently than the random agent, as seen by the Top Right. As seen over multiple runs, the random agent achieves a higher reward than the more intelligent equity agent as it is able to explore the large state and action space more rigorously. This is further confirmed in inter-play plots depicted in the following section. On average, we find that even the DQN trained against the Random agent (Bottom Right) achieves a higher reward for similar reasons. Regardless of the variation of DQN, all algorithms achieve similar performance which suggests limitations inherent to DQN.

4.3 Testing

We test trained agents in the most reasonable way possible: we allow inter-agent play. In particular, we take the best DQN agent and the Best PPO agent to play Heads Up Texas Hold'em. Here the best agent is determined through inter-agent play, also illustrated by the episode reward. More specifically, we play a total of 100 poker games, in which the PPO agent beats the Double DQN agent, around 60-40, as seen in the plot below. In order to ensure that the "best model" truly has learned, we play another 100 games against the random agent, where the PPO agent wins handedly, around 80-20.

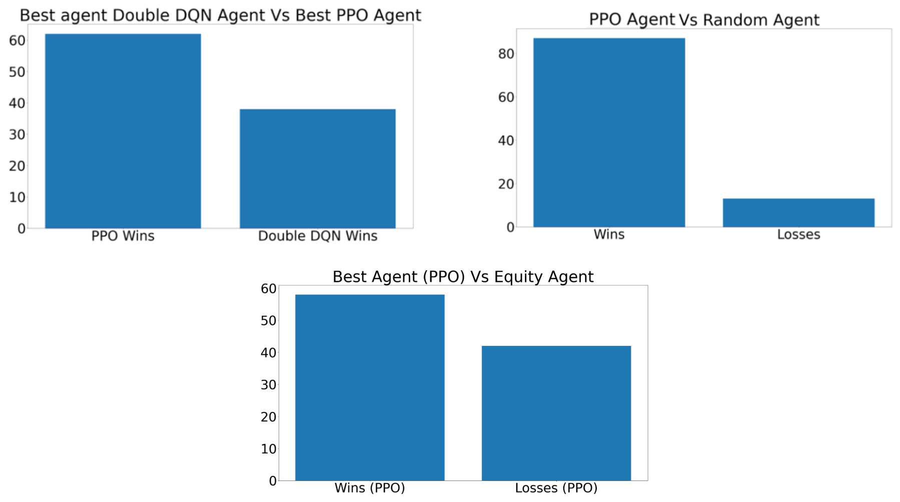

Figure 2: As played over 100 games, we see that the best PPO agent outperforms the best DQN agent, the double DQN. As a baseline comparison, we see that the PPO wins hand idly over the random agent, but not as well over the Equity Agent. While this is certainly not the state of the art, it represents a computationally limited model with a small network that is able of learning a policy to play poker well.

5 Unexpected Issues that Stymied Progress

When working on this project we ran into several issues. We struggled quite a bit with the framework, in particular when it came to the training of a PPO agent. The reason for this is that the Neuron Poker framework uses Keras-RL, which currently does not support PPO. We tried to make adjustments to the Keras-RL code to make it work, this was unfortunately to no avail. In the end we decided to try integrating another RL framework called EasyRL to train the agent with PPO. This required a substantial amount of tweaking before we were finally able to get it to train.

Another issue with the Keras-RL framework that it is not easily compatible with Apples newest ARM-based chips, which was a problem when the computers of two of the teams members had this type of chip. We tried to solve this issue by moving the training of the models to Google Colab, but as training requires a substantial amount of time, Colab was not a feasible solution as the sessions would expire before the agents had achieved reasonable performance. The long training time was also problematic in general, as it limited the quality of the resulting agents.

6 Conclusion/Discussion

One of our main takeaways from this project is that the PPO algorithm was able to learn poker strategies to the extent that it was able to consistently beat the random agent, and outperform the relatively intelligent baseline of the Equity agent. It also outperformed the best of the trained DQN agents, namely the Double DQN agent. This shows the PPO algorithm's potential for an imperfect information game like poker. However, further experimentation would be needed to thoroughly assess its capabilities against more advanced agents.

An alternative could be to apply the self-training concepts seen with AlphaZero, ReBeL and Deepstack to the PPO algorithm, using older instantiations. Such an agent could help find a balance between the randomness of the random agent and the calculated approach of the equity agent and instead behave more like human players by estimating the most likely outcomes, but with a certain degree of unpredictably. This would further help capture the nuances of the game, as neither of the agents it was trained against take in the entire state when making their moves. So the opponents behavior is simply ignored.

The potential seen in this proof of concept is clear, however, and it gives a strong case that PPO is a suitable alternative to DQN for imperfect information games such as poker. Our DQN algorithms clearly struggled with training, and often ended in loops of illegal moves. We find that while DQN may not be the obvious solution, PPO offers a promising direction forward for imperfect information games.

7 Contributions

7.1 Alex Kashi

Alex took responsibility for making the PPO agent train properly which was no small feat considering all the struggles with using Keras-RL before transitioning to Easy-RL. He also did most of the training of the models once we had them up and running.

7.2 Vedang Lad

When it became clear that Apple's silicon was not going to work with the selected frameworks and environments, Vedang took the main responsibility for transitioning the training to the cloud, allowing all team members to train and experiment with DQN agents. Furthermore, he helped visualize the results of the trained agents and contributed a substantial part to the mid- and final report.

7.3 Håkon Grini

In addtion to training and experimenting with DQN agents, as well as the training environments. Håkon did most of the research into previous approaches and similar works. He also wrote a large portion of both reports.

References

[BBLG20a] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games, 2020.

[BBLG20b] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. *CoRR*, abs/2007.13544, 2020.

[BS18] Noam Brown and Tuomas Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018.

[BS19] Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890, 2019.

[BSA18] Noam Brown, Tuomas Sandholm, and Brandon Amos. Depth-limited solving for imperfect-information games. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[MKS$^+$13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[MSB$^+$17a] Matej Moravč ík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, may 2017.

[MSB$^+$17b] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[Reu19] Dick Reuter. Neuron poker. `https://github.com/dickreuter/neuron_poker`, 2019.

[SHS$^+$17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[SLM$^+$15] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015.

[SLZ$^+$18] Sriram Srinivasan, Marc Lanctot, Vinicius Zambaldi, Julien Perolat, Karl Tuyls, Remi Munos, and Michael Bowling. Actor-critic policy optimization in partially observable multiagent environments, 2018.

[SMB$^+$21] Martin Schmid, Matej Moravcik, Neil Burch, Rudolf Kadlec, Josh Davidson, Kevin Waugh, Nolan Bard, Finbarr Timbers, Marc Lanctot, Zach Holland, Elnaz Davoodi, Alden Christianson, and Michael Bowling. Player of games, 2021.

[Ste19] Eric Steinberger. Pokerrl. `https://github.com/TinkeringCode/PokerRL`, 2019.

[SWD$^+$17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[vHGS15] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.