Winning Uno With Reinforcement Learning

Olivia Brown
Stanford University
olivia21@stanford.edu

Diego Jasson
Stanford University
djasson@stanford.edu

Ankush Swarnakar
Stanford University
ankushsw@stanford.edu

Abstract

The card game Uno is a complex environment to explore with reinforcement learning, with several sources of uncertainty, including uncertainty over the next card drawn, uncertainty over others' hands, etc. Here, we introduce two approaches to train an agent that optimally plays Uno. Because of the extensive complexity of the state space, we pursue model-free approaches that estimate an action-value function Q, rather than storing a Q value for all state-action pairs (s, a). Specifically, we train agents with Q-learning (DQN) and SARSA (DeepSARSA), using a feedforward neural network to approximate Q, using ϵ-greedy exploration in tournaments with games of two random agent competitors. Though DeepSARSA converges faster and more stably than DQN, DQN consistently outperforms DeepSARSA in both direct competition and tournaments with random agents. Both approaches also significantly outperform random agents in individual tournaments against two and three other random agent competitors, demonstrating that our approach effectively optimizes our agent's policy.

1 Introduction

Uno is popular card game played with its own unique deck. There are two main versions of the game: official and common. The official game rules prescribe a point value to each action and a player wins by being the first to reach a threshold. The common game is the popularized version where the first player to have no cards remaining is the winner; this paper uses only the common version of the game. Therefore, players strategize to shed the most cards and to prevent the other players from shedding theirs (primarily by using action cards). Here, we aim to train an agent that optimally plays the common version of Uno.

2 Game Description

2.1 Set Up

Uno is a card game with the following set-up:

- $2 - 10$ players
- Each player is dealt seven cards from a deck. You can only see your own cards.
- The remaining cards, which are also not visible, are placed in a draw pile.
- The top card from the draw pile becomes the first card in the discard pile.

To reduce the complexity of our project, we consider games with 3-5 players.

2.2 Deck

The Uno deck contains numbered cards, action cards, and wild cards. The numbered cards are each one of four different colors (blue, green, red, and yellow) and have a number between $0 - 9$. The action cards each have one of the four colors and one of three actions (skip, reverse, and draw 2 cards). The wild cards allow you to change the current color of gameplay and some also include an action to draw 4 cards.

Value	Count
Number 0	1 of each color
Numbers $1-9$	2 of each color
Reverse	2 of each color
Skip	2 of each color
Draw 2	2 of each color
Wild	8

Table 1: Cards of Uno.

Notice that a player can have at most 2 of any card, with the exception of the wild cards. [6]

2.3 Game Play

The game begins with the player immediately left of the dealer, and proceeds in a clockwise direction among the players.

During a player's turn, their goal is to place one of their cards in the discard pile. To do so, the player must place a card from their hand that matches the card on top of the discard pile by number, color, and/or action. If they do not have a card that matches the top of the discard pile, they must draw a card from the draw pile. If they still do not have a card that matches the top of the discard pile, the game moves on to the next player. Alternatively, a player can play a "wild card" which allows them to reset the color on top of the draw pile.

Certain cards can alter the game play. If a player plays a reverse card, the direction of gameplay is reversed. If a player plays a skip card, the next player is skipped. If a player plays a draw card, the next player must draw a card from the draw pile.

The objective of the game is to be the first player to play all of your cards. The game ends once any player has played all their cards and thus has no cards remaining in their hand. [6]

3 Environment

For the environment, we utilized RLCard, an open source library. [15] To represent the state of the game, we need to define what a player knows.

3.1 State

At each player's turn (and therefore decision point), the current player can see the current state, which contains the following elements:

Name	Representation
Own Hand	list of card identifiers that current player has
Played Cards	list of card identifiers that all players have played
Others' Hands	list of card identifiers in all other players' hands
Target	the visible card on top of the discard file

Table 2: State representation of Uno.

The state is encoded as 7 planes, where each plane is 4×15 matrix. The 4 rows correspond to each of the 4 colors available, while the 15 columns correspond to the different values a card can have. Column 0-9 represent their respective number value, and then each of the follow columns correspond to each of the action cards (Wild, Wild Draw 4, Skip, Draw Two, and Reverse). Each of the entries is either 1 or 0.

Each card except for the wild cards has a maximum count of two. For encoding purposes, the wild cards are arbitrarily assigned a color for identification purposes, though they can be played on top of any color. With this encoding, a player can have at most two of any card. Thus, to capture the count of the cards, there are three different planes that represent the count: there is one plane that represents having zero of that card, one plane that represents having one of that card, and one that represents having two of that card. To represent the entire state, there are three planes (for the different counts) for a player's own hand and for the others' hand, and there is one plane that represents the target, resulting in a total of 7 planes.

We represent this as a $7 \times 4 \times 15$ NumPy matrix. Since each of the $(7)(4)(15) = 420$ parameters can be either 0 or 1, our state space $\mathcal{S}$ has size $|\mathcal{S}| = 2^{420}$, which is on the order of 10^{126}.

3.2 Action

Each possible action is given an identifier. In total, there are 61 different actions a player can take, which are playing each of the 60 different cards and drawing a card from the deck. Our action space $\mathcal{A}$ thus has size $|\mathcal{A}| = 61$.

Since our state stores our agent's hand, we can enumerate our legal possible actions with this representation.

3.3 Reward

The reward is defined as $+1$ for winning, -1 for losing, and 0 for all intermediate states.

4 Related Work

Reinforcement learning has often been applied to games, especially card games [5]. Previous work has identified card games' ideal structure for Reinforcement Learning application, including large state spaces, competitive agents with partial information sets, large action sets, and small or delayed returns [15]. As a result, many approaches have been applied to card games. Approaches include Deep Q Networks (DQN), Monte Carlo Tree Search, Advantage-Actor Critic methods, and Counterfactual Regret Minimization (CFR). Jiang et al. 2019 show that computationally expensive methods such as CFR that require tree traversals are unable to efficiently deal with large state space games, such as Uno [7].

Card games can become particularly complex when agents are dynamic and can learn over time [3], but since we train against static agents, we did not investigate the literature further. Zha et al., 2020 create a virtual environment for several card games, including Uno. Their work applying popular Reinforcement Learning algorithms on the game demonstrated the feasibility of RL approaches for Uno [15].

Deep learning approaches to Q-Learning and SARSA allow algorithms to approximate the action value function Q with neural networks, rather than storing and update a Q value for all state-action pairs (s, a) [11]. This improvement allows for mitigation of catastrophic forgetting and smoother updates in several multi-agent card games [4]. Additionally, model-free approaches like Q-learning or SARSA, implemented with a neural network, can train offline given sampled data from the environment. As a result, a deep-learning approach requires less interaction with the environment to converge, and as previously mentioned, will result in lower variance in learning updates [2].

5 Approach

Given the extensive complexity and size of the state space for Uno, we find it unwise to model or estimate a transition function $T(s, a)$ or a reward function $R(s, a)$ for all state action pairs (s, a) explicitly. Thus, in our approach, we pursue two model-free approaches: Q-learning and SARSA.

5.1 Q Estimator

Our agent must determine its policy from an action value function Q defined over all state-action pairs (s, a). In traditional Q-learning, the algorithm updates its estimate of Q by sampling a trajectory (s, a, r, s') from the environment and following the rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s,a)\right)$$

Here, γ represents our discount factor and α represents our learning rate, which is a hyperparameter [8].

Intuitively, this algorithm accepts a trajectory (s, a, r, s') and increments our estimate of $Q(s, a)$ with the reward r, summed with the maximum discounted Q across all actions a' available at state s' subtracted from the current estimate of $Q(s, a)$, all scaled by our learning rate α. The algorithm updates Q towards the objective that $Q(s, a) \approx Q^*(s, a)$, where Q^* is the true action value function.

Though traditional Q-learning allows us to train an agent without explicitly modeling a transition function T, it still requires that we represent Q for all state-action pairs (s, a). This is difficult for the game of Uno given the vast complexity and size of our state and action space. Thus, we instead use deep learning to train a function approximator Q.

Here, our approximator is called a Q-network and is parameterized by θ. The network is trained to fulfill the objective that $Q(s, a; \theta) \approx Q^*(s, a)$.

In each step i of our algorithm, we wish to minimize the loss term $\mathcal{L}$:

$$\mathcal{L}(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \rho}[(y_i - Q(s,a;\theta_i))^2]$$

where y_i is called the temporal difference target, defined as $y_i = r + \max_{a'} Q(s', a'; \theta_{i-1})$. Here, γ again represents our discount factor and ρ represents a behavior distribution over all trajectories [11].

To minimize this loss, we leverage the gradient:

$$\nabla_{\theta_i} \mathcal{L}(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \rho} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s,a;\theta_i)\right) \nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

Instead of computing this expectation directly over ρ, we perform stochastic gradient descent (SGD) with a sampled trajectory (s, a, r, s') and use the argument of the expectation to calculate our gradient $\nabla_{\theta_i} \mathcal{L}(\theta_i)$ [11].

To approximate our Q function, we use a feedforward neural network with two dense hidden layers a and b, each with size 512. For input state s in its $7 \times 15 \times 4$ matrix representation, we flatten the matrix into an array x of length $(7)(15)(4) = 420$. This input x is fed into a fully-connected hidden layer of size 420×512, the output of which is fed into another fully-connected hidden layer of size 512×512. The output of the second layer is fed into an output layer of size 512×61, which outputs a vector y that gives us the estimates for Q for each of the 61 actions. Our Q-learning algorithm optimizes the $420 \times 512 + 512 \times 512 + 512 \times 61 \approx 500$K parameters (denoted as θ) of our network.

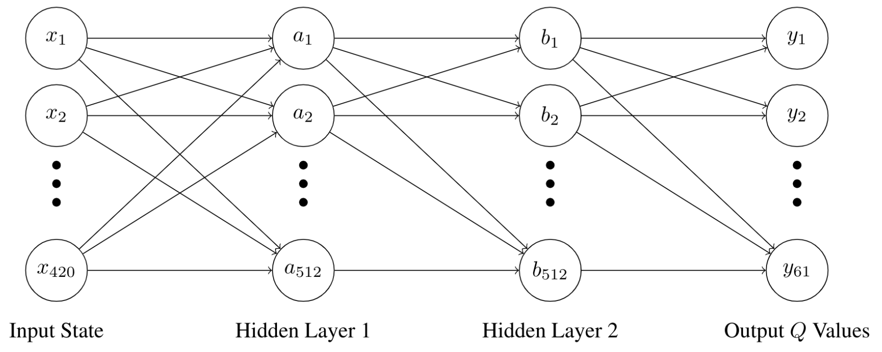

Figure 1: Network architecture to approximate our action value function Q.

We used NumPy for matrix representations and TensorFlow to train our neural network [1].

5.2 Epsilon-Greedy Exploration

To generate trajectories from which we can update our estimate of Q, we use ϵ-greedy exploration. Given a current state s, ϵ-greedy exploration chooses a random legal action with probability ϵ and a greedy legal action with probability $1 - \epsilon$ [8]. We then take that action a, and receive a reward r and next state s'. Our exploration function returns the tuple (s, a, r, s') to our algorithm to be incorporated into the estimate of Q.

ϵ-greedy exploration allows us to dually train our estimate of Q on sequences of states and actions that are presumptively optimal as well as a diverse selection of random states and actions, enabling us to cover the entire state space. This is especially crucial in the first few iterations of our algorithm, since our estimate of Q is based on very sparse data, so it is critical that we randomly sample more state-action pairs to obtain a more generalizable estimate of Q.

With ϵ-greedy exploration, as our algorithm completes more iterations, it will have seen more of the entire state-action space. Thus, as our algorithm continues to run, it becomes relatively less important to sample and train off of random actions as compared to presumptively optimal ones. In each iteration, we thus decay our probability ϵ by some factor κ. Concretely, we have $\epsilon \leftarrow \kappa\epsilon$ [9]. This enables us to sample fewer random actions as our algorithm converges, allowing us to finetune the most important parameters relevant for an optimal policy or estimate of Q.

5.3 Deep Q-Network (DQN)

In a given iteration i of our Q-learning algorithm, we simulate our agent in a tournament T_i where they play $n = 100$ games of Uno with two other agents, who play with random policies. Our agent uses ϵ-greedy exploration in gameplay, and returns a set of trajectories (s, a, r, s'). Our algorithm incorporates those trajectories into our estimate of Q with the SGD update rule:

$$\theta_{i+1} \leftarrow \theta_i - \alpha \left(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i)$$

Here, α represents our learning rate, a hyperparameter that we experimentally finetune [11].

5.4 DeepSARSA

Like Q-learning, SARSA incrementally updates our estimate of Q with trajectories sampled from our environment. However, Q-learning is considered an off-policy algorithm because it updates Q based on the next state s' and the greedy action from that state $\max_{a'}(s', a')$. SARSA, in contrast, is an on-policy algorithm that updates Q based on the next state s' and the actual action taken from the next state a' [8]. Thus, our algorithm instead uses the SGD update rule:

$$\theta_{i+1} \leftarrow \theta_i - \alpha \left(r + \gamma Q(s', a'; \theta_i) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i)$$

where a' represents the actual action taken from next state s', which may or may not be greedy, since our exploration policy is not strictly optimal [13].

Like with Q-learning, in an iteration i of our algorithm, we simulate our agent in a tournament T_i of 100 games against random agents, where they use ϵ-greedy exploration. Our exploration function returns extended trajectories (s, a, r, s', a') to be incorporated into our SARSA update.

6 Results & Discussion

All training was done on a VM instance hosted on Google Cloud Platform. To speed up the computation of the backward pass of our network, we used an NVIDIA Tesla K80 GPU.

6.1 Hyperparameter Validation

We optimized three hyperparameters: the discount factor γ, the learning rate α, the our ϵ-decay factor κ. We trained our DQN and DeepSARSA agents for $n = 40000$ iterations on all possible combinations of hyperparameters from the sets $\gamma \in \{0.75, 0.825, 0.95\}$, $\alpha \in \{0.0001, 0.001, 0.01\}$, $\epsilon \in \{0.8, 0.95\}$. On the basis of highest average win rate in a tournament of $n = 100$ games with 2 random agents, we found that the hyperparameters $\alpha = 0.0001, \gamma = 0.95, \epsilon = 0.95$ worked best.

6.2 Training Comparisons

We trained our DQN agent for $n = 350000$ iterations, until we felt that it had converged via visual observation of the reward trend. Our final agent had an average reward of 0.244 and an average winning rate of 0.622 in a tournament of $n = 100$ games with two random agent competitors. Our estimate of Q improves most within the first 50,000 iterations of the algorithm, but still steadily improves until around 240,000 iterations.

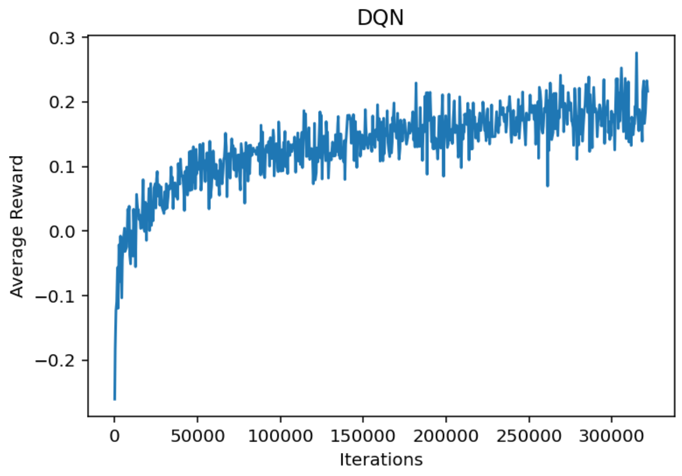

Figure 2: Average reward of DQN agent over iterations of training.

We trained our DeepSARSA agent for $n = 125000$ iterations, until we felt that it had converged via visual observation of reward trend. Our final agent had an average reward of 0.210 and an average winning rate of 0.605 in a tournament of $n = 100$ games with two random agent competitors. Like the DQN agent, most of the improvements in our Q estimator take place within the first 50,000 iterations. Still, steady improvements are made until roughly 100,000 iterations.

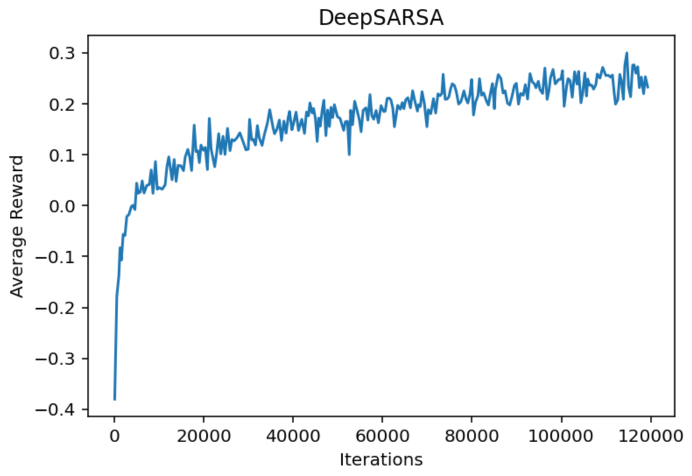

Figure 3: Average reward of DeepSARSA agent over iterations of training.

In comparing the training curves, we see that DeepSARSA took significantly fewer iterations to converge than DQN, despite having the same final performance on the final training tournament. This suggests that an update rule that incorporates the actual action a' taken from the next state s' in a trajectory converges quicker than incorporating a greedy action. Additionally, since we decay our probability ϵ of choosing a random action from a given state, as our agents complete more iterations, they more likely to choose a greedy action as the next action a', which makes the SARSA update equivalent to the Q-learning update. This may partially explain why DQN and DeepSARSA have strikingly similar training curves, particularly in later iterations.

Moreover, from the trends in average reward (Figs. 2 and 3), we also see that the average reward for DQN is much more noisy than the average reward for DeepSARSA. This indicates that though both algorithms converge, DQN tends to make more extreme updates in parameters in later iterations. This can be attributed to its greedy update rule, which does not always update model parameters based on the actual next action taken during exploration. Our results suggest that incorporating updates from actual actions taken may result in a more stable path towards convergence.

6.3 Performance Comparisons

Our primary performance metrics are average win rate (proportion of games won in a tournament) and average reward (average of all rewards earned in a tournament).

We begin by individually simulating our agents in three tournaments of $n = 1000$ rounds: a 3 Agent Game with two random agent competitors, a 4 Agent Game with three random agent competitors, and a 5 Agent Game with four random agent competitors. Though we trained our model on tournaments with exclusively games of size 3, we wanted to test how well our approaches could generalize to accommodate more agents. As a baseline, we also simulate a random agent player for the three tournaments.

	Average Win Rate		
Agent	3 Agent Game	4 Agent Game	5 Agent Game
DQN	0.635	0.455	0.231
DeepSARSA	0.591	0.401	0.249
Random Agent	0.335	0.248	0.206

Table 3: Average win rates in tournament of varying game sizes with random agent competitors.

DQN and DeepSARSA outperform our random agent across all three tournaments, which indicates that our algorithm is able to accurately optimize our policy, even in larger games. We also observe that DQN consistently outperforms DeepSARSA in the 3 and 4 Agent Games by a margin of at least 0.4, which corroborates with its slightly better final average reward during training. The margin between DQN and DeepSARSA is larger in the 4 Agent Game than the 3 Agent Game, which indicates that DQN may generalize better than DeepSARSA. However, DeepSARSA's margin on the 5 Agent Game is slightly larger. Still, neither agent significantly outperforms the Random Agent in the 5 Agent Game, which indicates that the policy is not as optimized for larger game sizes.

To get a better sense for comparison, we also simulated our agents in gameplay against one another. We ran a tournament of $n = 1000$ games of three agents: a DQN agent parameterized by our model, a DeepSARSA agent parameterized by our model, and a random agent.

Agent	Average Reward	Average Winning Rate
Random	-0.506	0.247
DQN	-0.220	0.390
DeepSARSA	-0.274	0.363

Table 4: Average winning rate and reward in tournament of DQN, DeepSARSA, and Random Agent.

Consistent with our earlier results, DQN wins more games than DeepSARSA, by a margin of 0.027 in this tournament. Both agents significantly outperform the random agent, by a margin of at least 0.1. Both agents also outperform the expected performance of a random agent in a tournament of exclusively random agents ($\frac{1}{3} = 0.333$), but not by much; this is likely because this tournament design requires that two-thirds of the agents are optimized, which makes the distribution of outcomes roughly uniform.

7 Conclusion

We sought to apply reinforcement learning techniques to train an agent to optimally play Uno, a popular multi-player card game that utilizes a unique deck. Card games are a powerful framework for decision-making algorithms, since they involve considerable uncertainty (e.g. uncertainty over what card might be drawn, uncertainty over others' hands, etc.) and have clear rules around legal actions and defining rewards.

Given the vast state complexity ($|\mathcal{S}| = 2^{420}$) and delayed rewards ($r = \pm 1$ only at an end state), we primarily pursue model-free approaches that learn an estimate of our action value function Q, rather than explicitly storing a Q value for all state-action pairs. Specifically, we employed Deep Q-Network (DQN) and DeepSARSA algorithms to train on trajectories generated using the RLCard environment [15]. In each iteration of training, we simulate our agent in a tournament of 100 games with two other random agent competitors, where our agent follows ϵ-greedy exploration. We train a feedforward neural network with two dense hidden layers to estimate Q.

We found that both our DQN agent and DeepSARSA agent outperform a random agent in tournaments of 1000 games with three, four, and five players by an average win rate margin of at least 0.1 in the three and four player tournaments. This suggests that our approach tangibly optimizes the policy for an agent playing Uno. Additionally, we see that DQN outperforms DeepSARSA in the tournaments with three and four players, and also beats DeepSARSA more often in direct competition by roughly 2.7 percentage points. This suggests that updating our estimate of Q based on a greedy action taken from next state s', and not the actual action a', optimizes our policy better overall. Neither model significantly outperformed the random agent in the tournament with five players. Each agent's performance also significantly dropped as more competitors were introduced, indicating that training our agent on exclusively tournaments with three players does not generalize well to more agents.

Though DQN consistently outperforms DeepSARSA, DeepSARSA converges in nearly 10^5 fewer iterations and has much more stable path towards convergence. Since SARSA only updates its parameters of Q based on actual actions taken, it may make less extreme updates in future iterations, leading to faster convergence.

Ultimately, we demonstrate that both DQN and DeepSARSA are promising approaches for training an agent to optimally play the card game Uno. For future work, we plan to test other neural network architectures for our estimate of Q. Since our state representation is split into 7 planes, we could try 7 independent input layers of size 4×15 each that learn a separate set of parameters for each plane, which are concatenated together downstream in the network. We could also try convolutional, residual, and recurrent neural network architectures [14]. We also plan to improve our training procedure by simulating tournaments, where each game has a randomly-assigned number of players, to improve the generalization of our agent. Lastly, we might also try other learning approaches like Monte Carlo Tree Search [10] or exploration strategies like curiosity-based exploration [12].

8 Contributions

Olivia worked on explicitly outlining the game rules and defining our state, action, and reward representations. Diego analyzed related literature, including approaches on other card games, to identify strategies for training. Ankush integrated our training and testing code with the RLCard environment, and focused on hyperparameter optimization. We collectively contributed to the development of our training algorithms (DQN and DeepSARSA), testing of our agents, analysis, and writing of the report.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017.

[3] Pablo Barros, Ana Tanevska, and Alessandra Sciutti. Learning from learners: Adapting reinforcement learning agents to be competitive in a card game, 2020.

[4] Henry Charlesworth. Application of self-play reinforcement learning to a four-player game of imperfect information. *CoRR*, abs/1808.10442, 2018.

[5] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *CoRR*, abs/1603.01121, 2016.

[6] Mattel Inc. Uno Instructional Sheet, 2001.

[7] Qiqi Jiang, Kuangzheng Li, Boyao Du, and Hao Chen amd Hai Fang. Deltadou: Expert-level doudizhu ai through self-play, 2019.

[8] Mykel Kochenderfer, Tim Wheeler, and Kyle Wray. *Algorithms for Decision Making*. GitHub, 2020.

[9] Yuxi Li. Deep reinforcement learning. *CoRR*, abs/1810.06339, 2018.

[10] Xiaobai Ma, Katherine Driggs-Campbell, Zongzhang Zhang, and Mykel J. Kochenderfer. Monte-carlo tree search for policy optimization, 2019.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015.

[12] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. *CoRR*, abs/1705.05363, 2017.

[13] Andrei Claudiu Roibu. Design of artificial intelligence agents for games using deep reinforcement learning. *CoRR*, abs/1905.04127, 2019.

[14] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

[15] Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, Songyi Huang, Ruzhe Wei, Junyu Guo, and Xia Hu. Rlcard: A toolkit for reinforcement learning in card games, 2020.