OpenSpiel: A Framework for Reinforcement Learning in Games

Marc Lanctot[1,**], Edward Lockhart[1,**], Jean-Baptiste Lespiau[1,**], Vinicius Zambaldi[1,**], Satyaki Upadhyay[2], Julien Pérolat[1], Sriram Srinivasan[2], Finbarr Timbers[1], Karl Tuyls[1], Shayegan Omidshafiei[1], Daniel Hennes[1], Dustin Morrill[1,3], Paul Muller[1], Timo Ewalds[1], Ryan Faulkner[1], János Kramár[1], Bart De Vylder[1], Brennan Saeta[2], James Bradbury[2], David Ding[1], Sebastian Borgeaud[1], Matthew Lai[1], Julian Schrittwieser[1], Thomas Anthony[1], Edward Hughes[1], Ivo Danihelka[1] and Jonah Ryan-Davis[2]
[1]DeepMind, [2]Google, [3]University of Alberta, [**]These authors contributed equally

OpenSpiel is a collection of environments and algorithms for research in general reinforcement learning and search/planning in games. OpenSpiel supports *n*-player (single- and multi- agent) zero-sum, cooperative and general-sum, one-shot and sequential, strictly turn-taking and simultaneous-move, perfect and imperfect information games, as well as traditional multiagent environments such as (partially- and fully-observable) grid worlds and social dilemmas. OpenSpiel also includes tools to analyze learning dynamics and other common evaluation metrics. This document serves both as an overview of the code base and an introduction to the terminology, core concepts, and algorithms across the fields of reinforcement learning, computational game theory, and search.

Contents

1. OpenSpiel Overview

1.1. Acknowledgments

OpenSpiel has been possible due to a team of contributors. For a full list of all the contributors, please see the list of authors on github.

We would also like to thank the following people, who helped and supported the development of OpenSpiel:

- Remi Munos
- Michael Bowling
- Thore Graepel
- Shibl Mourad
- Nathalie Beauguerlange
- Ellen Clancy
- Louise Deason
- Andreas Fidjeland
- Michelle Bawn
- Yoram Bachrach
- Dan Zheng
- Martin Schmid
- Neil Burch
- Damien Boudot
- Adam Cain

1.2. OpenSpiel At a Glance

We provide an intentionally brief overview here. For details, please see Section 3.

OpenSpiel provides a framework for writing games and algorithms and evaluating them on a variety of benchmark games. OpenSpiel contains implementations of over 20 different games of various sorts (perfect information, simultaneous move, imperfect information, gridworld games, an auction game, and several normal-form / matrix games). Game implementations are in C++ and wrapped in Python. Algorithms are implemented in C++ and/or Python. The API is almost identical in the two languages, so code can easily be translated if needed. A subset of the library has also been ported to Swift. Most of the learning algorithms written in Python use Tensorflow [1], though we are actively seeking examples and other support for PyTorch [61] and JAX[1].

OpenSpiel has been tested on Linux and MacOS. There is also limited support on Windows.

Components of OpenSpiel are listed in Tables 1 and 2. As of October 2019, these tables will no longer be updated: Please refer to the Overview of Implemented Games or the Overview of Implemented Algorithms pages on the web site for most current information.

[1] https://github.com/google/jax

Game	Reference(s)
Backgammon	Wikipedia
Breakthrough	Wikipedia
Bridge bidding	Wikipedia
Catch	[51] and [59, Appendix A]
Coin Game	[64]
Connect Four	Wikipedia
Cooperative Box-Pushing	[70]
Chess	Wikipedia
First-price Sealed-bid Auction	Wikipedia
Go	Wikipedia
Goofspiel	Wikipedia
Hanabi (via HLE)	Wikipedia, [7]
Havannah	Wikipedia
Hex	Wikipedia
Kuhn poker	Wikipedia, [38]
Laser Tag	[42, 41]
Leduc poker	[73]
Liar's Dice	Wikipedia
Markov Soccer	[45, 28]
Matching Pennies (three-player)	[33]
Matrix Games	[71]
Negotiation	[43, 20]
Oshi-Zumo	[19, 10, 62]
Oware	Wikipedia
Pentago	Wikipedia
Phantom Tic-Tac-Toe	[3, 40, 44]
Pig	[56]
Quoridor	Wikipedia
Tic-Tac-Toe	Wikipedia
Tiny Bridge	
Tiny Hanabi	[23]
Y	Wikipedia
Cliff-Walking (Python-only)	[75, Chapter 6]

Table 1 | Game Implementations in OpenSpiel as of October 2019. Please see Overview of Implemented Games for an up-to-date list.

Algorithm	Category	Reference(s)
Minimax (and Alpha-Beta) Search	Search	Wikipedia, Wikipedia, [34]
Monte Carlo tree search	Search	Wikipedia, [35, 21, 18]
Lemke-Howson (via `nashpy`)	Opt.	[71]
Sequence-form linear programming	Opt.	[36, 71]
Counterfactual Regret Minimization (CFR)	Tabular	[86, 55]
CFR against a best responder (CFR-BR)	Tabular	[32]
Exploitability / Best Response	Tabular	[86]
External sampling Monte Carlo CFR	Tabular	[39, 40]
Outcome sampling Monte Carlo CFR	Tabular	[39, 40]
Q-learning	Tabular	[75]
Value Iteration	Tabular	[75]
Advantage Actor-Critic (A2C)	RL	[50]
Deep Q-networks (DQN)	RL	[52]
Ephemeral Value Adjustments (EVA)	RL	[26]
Deep CFR	MARL	[15]
Exploitability Descent (ED)	MARL	[46]
(Extensive-form) Fictitious Play (XFP)	MARL	[29]
Neural Fictitious Self-Play (NFSP)	MARL	[30]
Neural Replicator Dynamics (NeuRD)	MARL	[57]
Regret Policy Gradients (RPG, RMPG)	MARL	[74]
Policy-Space Response Oracles (PSRO)	MARL	[41]
Q-based “all-action” Policy Gradients (QPG)	MARL	[76, 63, 74]
Regression CFR (RCFR)	MARL	[81, 54]
Rectified Nash Response (PSRO_{rN})	MARL	[4]
α-Rank	Eval / Viz	[58]
Replicator / Evolutionary Dynamics	Eval / Viz	[31, 69]

Table 2 | Algorithms Implementated in OpenSpiel as of October 2019. Please see Overview of Implemented Algorithms for an updated list.

2. Getting Started

2.1. Getting and Building OpenSpiel

The following commands will clone the repository and build OpenSpiel on Ubuntu or Debian Linux, or MacOS. There is also limited support for Windows. We now show the *fastest* way to install OpenSpiel. Please see the recommended installation instructions using `virtualenv` for more detail.

```
apt-get install git
git clone https://github.com/deepmind/open_spiel.git
cd open_spiel
./install.sh
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
# Install pip deps as your user. Do not use the system's pip.
python3 get-pip.py --user
# If using Linux:
  export PATH="$PATH:$HOME/.local/bin"
  ln -s $HOME/.local/bin/pip3 $HOME/.local/bin/mypip3
# Else (MacOS):
  export PATH="$PATH:$HOME/Library/Python/3.7/bin"
  ln -s $HOME/Library/Python/3.7/bin/pip3 $HOME/Library/Python/3.7/bin/mypip3
mypip3 install --upgrade pip --user
mypip3 install --upgrade setuptools testresources --user
mypip3 install --upgrade -r requirements.txt --user
mkdir build
cd build
CXX=clang++
cmake -DPython_TARGET_VERSION=3.6 -DCMAKE_CXX_COMPILER=clang++ ../open_spiel
make -j12   # The 12 here is the number of parallel processes used to build
ctest -j12  # Run the tests to verify that the installation succeeded
```

Note that we have tested OpenSpiel Linux and MacOS, and there is limited support on Windows. Also, for the case of Linux, some of the scripts and instructions currently assume Debian-based distributions (i.e. Debian, Ubuntu, etc.). All of the dependencies exist on other distributions, but may have different names, and package managers differ. Please see `install.sh` for necessary dependencies.

2.1.1. Setting PATH and PYTHONPATH

To be able to import the Python code (both the C++ binding pyspiel and the rest) from any location, you will need to add to your PYTHONPATH the root directory and the `open_spiel` directory. Add the following in your `.bashrc` or `.profile`:

```
# For your user's pip installation.
# Note! On MacOS, replace $HOME/.local/bin with $HOME/Library/Python/3.7/bin
export PATH=$PATH:$HOME/.local/bin
# For the Python modules in open_spiel.
export PYTHONPATH=$PYTHONPATH:/<path_to_open_spiel>
# For the Python bindings of Pyspiel
export PYTHONPATH=$PYTHONPATH:/<path_to_open_spiel>/build/python
```

2.2. Running the First Example

After having built OpenSpiel following Sec 2.1, run the example from the `build` directory without any arguments:

```
examples/example
```

This prints out a list of registered games and the usage. Now, let's play a game of Tic-Tac-Toe with uniform random players:

```
examples/example --game=tic_tac_toe
```

Wow – how exhilarating! Now, why not try one of your favorite games?

Note that the structure in the build directory mirrors that of the source, so the example is found in `open_spiel/examples/example.cc`. At this stage you can run one of many binaries created, such as `games/backgammon_test` or `algorithms/external_sampling_mccfr_test`.

Once you have set your PYTHONPATH as explained in Sec 2.1.1, you can similarly run the python examples:

```
cd ../open_spiel
python3 python/examples/example.py --game=breakthrough
python3 python/examples/matrix_game_example.py
```

Nice!

2.3. Adding a New Game

We describe here only the simplest and fastest way to add a new game. It is ideal to first be aware of the general API, which is described on a high level in Section 3, on github, and via comments in `spiel.h`.

1. Choose a game to copy from in `games/`. Suggested games: Tic-Tac-Toe and Breakthrough for perfect information without chance events, Backgammon or Pig for perfect information games with chance events, Goofspiel and Oshi-Zumo for simultaneous move games, and Leduc poker and Liar's dice for imperfect information games. For the rest of these steps, we assume Tic-Tac-Toe.
2. Copy the header and source: `tic_tac_toe.h`, `tic_tac_toe.cc`, and `tic_tac_toe_test.cc` to `new_game.h`, `new_game.cc`, and `new_game_test.cc`.
3. Add the new game's source files to `games/CMakeLists.txt`.
4. Add the new game's test target to `games/CMakeLists.txt`
5. In `new_game.h`, rename the header guard at the the top and bottom of the file.
6. In the new files, rename the inner-most namespace from `tic_tac_toe` to `new_game`
7. In the new files, rename `TicTacToeGame` and `TicTacToeState` to `NewGameGame` and `NewGameState`
8. At the top of `new_game.cc`, change the short name to `new_game` and include the new game's header.
9. Add the short name to the list of expected games in `python/tests/pyspiel_test.py`.
10. You should now have a duplicate game of Tic-Tac-Toe under a different name. It should build and the test should run, and can be verified by rebuilding and running the example from Section 2.2.
11. Now, change the implementations of the functions in `NewGameGame` and `NewGameState` to reflect your new game's logic. Most API functions should be clear from the game you copied from. If not,

each API function that is overridden will be fully documented in superclasses in `spiel.h`. See also the description of extensive-form games in Section 3.1 which closely matches the API.

12. Once done, rebuild and rerun the tests from Sec 2.1 to ensure everything passes (including your new game's test!)

2.4. Adding a New Algorithm

Adding a new algorithm is fairly straight-forward. Like adding a game, it is easiest to copy and start from one of the existing algorithms. If adding a C++ algorithm, choose one from `algorithms/`. If adding a Python algorithm, choose one from `python/algorithms/`. For appropriate matches, see Table 2.

Unlike games, there is no specific structure or API that must be followed for an algorithm. If the algorithm is one in a class of existing algorithms, then we advise keeping the style and design similar to the ones in the same class, re-using function or modules where possible.

The algorithms themselves are not binaries, but classes or functions that can be used externally. The best way to show an example of an algorithm's use is via a test. However, there are also binary executables in `examples/` and `python/examples/`.

3. Design and API

The purpose of OpenSpiel is to promote *general* multiagent reinforcement learning across many different game types, in a similar way as general game-playing [25] but with a heavy emphasis on learning and not in competition form. We hope that OpenSpiel could have a similar effect on general RL in games as the Atari Learning Environment [8, 47] has had on single-agent RL.

OpenSpiel provides a general API with a C++ foundation, which is exposed through Python bindings (via `pybind11`). Games are written in C++. This allows for fast or memory-efficient implementations of basic algorithms that might need the efficiency. Some custom RL environments are also implemented in Python. Most algorithms that require machine learning are implemented in Python.

Above all, OpenSpiel is designed to be easy to install and use, easy to understand, easy to extend ("hackable"), and general/broad. OpenSpiel is built around two major important design criteria:

1. **Keep it simple**. Simple choices are preferred to more complex ones. The code should be readable, usable, extendable by non-experts in the programming language(s), and especially to researchers from potentially different fields. OpenSpiel provides reference implementations that are used to learn from and prototype with, rather than fully-optimized / high-performance code that would require additional assumptions (narrowing the scope / breadth) or advanced (or lower-level) language features.
2. **Keep it light**. Dependencies can be problematic for long-term compatibility, maintenance, and ease-of-use. Unless there is strong justification, we tend to avoid introducing dependencies to keep things portable and easy to install.

3.1. Extensive-Form Games

There are several formalisms and corresponding research communities for representing multiagent interactions. It is beyond the scope of this paper to survey the various formalisms, so we describe the ones most relevant to our implementations. There have been recent efforts to harmonize the terminology and make useful associations among algorithms between computational game theory and reinforcement learning [74, 46, 37], so we base our terminology on classical concepts and these recent papers.

Games in OpenSpiel are represented as procedural extensive-form games [60, 71], though in some cases can also be cyclic such as in Markov Decision Processes [75] and Markov games [45]. We first give the classical definitions, then describe some extensions, and explain some equivalent notions between the fields of reinforcement learning and games.

An **extensive-form game** is a tuple $\langle \mathcal{N}, \mathcal{A}, \mathcal{H}, \mathcal{Z}, u, \tau, \mathcal{S} \rangle$, where

- $\mathcal{N} = \{1, 2, \ldots n\}$ is a finite set of n **players**[2]. There is also a special player c, called **chance**.
- $\mathcal{A}$ is a finite set of **actions** that players can take. This is a global set of state-independent actions; generally, only a subset of *legal* actions are available when agents decide.
- $\mathcal{H}$ is a finite set of **histories**. Each history is a sequence of actions that were taken from the start of the game.
- $\mathcal{Z} \subseteq \mathcal{H}$ is a subset of **terminal histories** that represents a completely played game.
- $u : \mathcal{Z} \to \Delta_u^n \subseteq \Re^n$, where $\Delta_u = [u_{\min}, u_{\max}]$, is the utility function assigning each player a utility at terminal states, and $u_{\min}, u_{\max}$ are constants representing the minimum and maximum utility.
- $\tau : \mathcal{H} \to \mathcal{N}$ is a **player identity** function; $\tau(h)$ identifies which player acts at h.
- $\mathcal{S}$ is a set of **states**. In general, $\mathcal{S}$ is a partition of $\mathcal{H}$ such that each state $s \in \mathcal{S}$ contains histories $h \in s$ that cannot be distinguished by $\tau(s) = \tau(h)$ where $h \in s$. Decisions are made by players at these states. There are several ways to precisely define $\mathcal{S}$ as described below.

We denote the legal actions available at state s as $\mathcal{A}(s) \subseteq \mathcal{A}$. Importantly, a history represents the true ground/world state: when agents act, they change this history, but depending on how the partition is chosen, some actions (including chance's) may be private and not revealed to some players.

We will extend this formalism further on to more easily describe how games are represented in OpenSpiel. However, we can already state some important categories of games:

- A **constant-sum** (k-sum) game is one where $\forall z \in \mathcal{Z}, \sum_{i \in \mathcal{N}} u_i(z) = k$.
- A **zero-sum** game is a constant-sum game with $k = 0$.
- An **identical interest** game is one where $\forall z \in \mathcal{Z}, \forall i, j \in \mathcal{N}, u_i(z) = u_j(z)$.
- A **general-sum game** is one without any constraint on the sum of the utilities.

In other words: k-sum games are strictly competitive, identical interest games are strictly cooperative, and general-sum games are neither or somewhere in between. Also,

- A **perfect information** game is one where there is only one history per state: $\forall s \in \mathcal{S}, |s| = 1$.
- A **imperfect information** game is one where there is generally more than one history per state, $\exists s \in \mathcal{S} : |s| > 1$.

Chess, Go, and Breakthrough are examples of perfect information games without events (no chance player). Backgammon and Pig are examples of perfect information games with chance events. Leduc poker, Kuhn poker, Liar's Dice, and Phantom Tic-Tac-Toe are examples of imperfect information games. Every one of these example games is zero-sum.

Definition 1. *A **chance node** (or chance event) is a history h such that $\tau(h) = c$.*

In zero-sum perfect information games, minimax and alpha-beta search are classical search algorithms for making decisions using heuristic value functions [34]. The analogs for perfect information games with chance events are expectiminimax [49] and *-minimax [6].

[2]Note that the player IDs range from 0 to $n - 1$ in the implementations.

3.1.1. Extension: Simultaneous-Move Games

We can augment the extensive-form game with a special kind of player, the simultaneous move player: $\div$. When $\tau(s) = \div$, each player i has a set of legal actions $\mathcal{A}_i(s)$, and all players act simultaneously choosing a **joint action** $a = (a_i)_{\{i \in \mathcal{N}\}}$. Histories in these games are then sequences of joint actions, and transitions take the form (h, a, h'). The rest of the properties from extensive-form games still hold.

Definition 2. *A **normal-form** (or **one-shot** game) is a simultaneous-move game with a single state,* $|S| = 1$. *A **matrix game** is a normal-form game where* $|\mathcal{N}| = 2$.

Fact 1. *A simultaneous-move game can be represented as a specific type of extensive-form game with imperfect information.*

To see why this is true: consider the game of Rock, Paper, Scissors ($\mathcal{A} = \{\text{R}, \text{P}, \text{S}\}$) where each player chooses a single action, revealing their choice simultaneously. An equivalent turn-based is the following: the first player writes their action on a piece of paper, and places it face down. Then, the second player does the same. Then, the choices are revealed simultaneously. The players acted at separate times, but the second player did not know the choice made by the first player (and hence could be in one of three histories: $h = \text{R}, h = \text{P},$ or $h = \text{S}$), and the game has two states instead of one state. In a game with many states, the same idea can simply be repeated for every state.

Why, then, represent these games differently? There are several reasons:

1. They have historically been treated as separate in the multiagent RL literature.
2. They can sometimes be solved using Bellman-style dynamic programming, unlike general imperfect information games.
3. They are slightly more general. In fact, one can represent a turn-based game using a simultaneous-move game, simply by setting $\mathcal{A}_i(s) = \emptyset$ for $j \neq \tau(s)$ or by adding a special PASS move as the only legal action when it is not a player's turn.

We elaborate on each of these points in the following section, when we relate simultaneous-move games to existing multiagent RL formalisms.

3.1.2. Policies, Objectives, and Multiagent Reinforcement Learning

We now add the last necessary ingredients for designing decision-making and learning algorithms, and bring in the remaining standard RL terms.

Definition 3. *A **policy*** $\pi : \mathcal{S} \to \Delta(\mathcal{A}(s))$, *where* $\Delta(X)$ *represents the set of probability distributions over* X, *describes agent behavior. An agent acts by selecting actions from its policy:* $a \sim \pi$. *A **deterministic** policy is one where at each state the distribution over actions has probability* 1 *on one action and zero on the others. A policy that is not (necessarily) deterministic is called **stochastic**.*

In games, the chance player is special because it *always plays with a fixed (stochastic) policy* π_c.

Definition 4. *A **transition function*** $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ *defines a probability distribution over successor states* s' *when choosing action* a *from state* s.

Fact 2. *A transition function can be equivalently represented using intermediate chance nodes between the histories of the predecessor and successor states* $h \in s$ *and* $h' \in s'$. *The transition function is then determined by* π_c *and* $\Pr(h|s)$.

Definition 5. *A player, or agent, has* ***perfect recall*** *if, the state does not lose the information about the past decisions made by the player. Formally, all histories $h \in s$, contain the same sequence of action of the current player: let $\textsc{SAHist}_i(h)$ be the history of only player i's state-action pairs (s, a) experienced along h. Player i has perfect recall if for all $s \in \{s \mid s \in \mathcal{S}, \tau(s) = i\}$, and all $h, h' \in s$, $\textsc{SAHist}_i(h) = \textsc{SAHist}_i(h')$.*

In Poker, a player acts from an information state, and the histories corresponding to such an information state only differ in the chance event outcomes that correspond to the opponent's private cards. In these partially-observable games, a state is normally called an **information state** to emphasize the fact that the agent's perception of the state (s) is different than the true underlying world state (one of $h \in s$).

The property of perfect recall turns out to be a very important criterion for determining convergence guarantees for exact tabular algorithms, as we show in Section 3.2.

Definition 6. *An observation is a partial view of the information state and contains strictly less information than the information state. To be valid, the sequence of observations and actions of all players should contain at least as much information as the information state. Formally: Let Ω be a finite set of* ***observations****. Let $O_i : \mathcal{S} \rightarrow \Omega$ be an observation function for player i and denote $o_i(s)$ as the observation. As s contains histories h, we will write $o_i(h) = o_i(s)$ if $h \in s$. A valid observation is such that the function $h \rightarrow (o_i(h'))_{h' \sqsubset h}$ defines a partition of the history space $\mathcal{H}$ that is a sub-partition of $\mathcal{S}$.*

In a multiplayer game, we define a per-step **reward** *to player i* for a transition as $r_i(s, a, s')$, with $r(s, a, s')$ representing the vector of returns to all players. In most OpenSpiel games, these $r(s, a, s') = 0$ until s' is terminal, ending the episode, and these values are obtained by `State::Rewards` and `State::PlayerReward` function called on s'. Player interaction over an episode generates a trajectory $\rho = (s_0, a_0, s_1, \cdots)$ whose length is $|\rho|$. We define a **return** *to player i* as $g^{\rho}_{t,i} = \sum_{t' \geq t}^{|\rho|-1} r_i(s_{t'}, a_{t'}, s_{t'+1})$ with g^{ρ}_t representing a vector of rewards to all players as with per-step rewards. In OpenSpiel, the `State::Returns` function provides g^{ρ}_0 and `State::PlayerReturn` provides $g^{\rho}_{0,i}$. Note that we do not use a discount factor when defining rewards here because most games are episodic; learning agents are free to discount rewards however they like, if necessary. Note also that the standard (undiscounted) return is the random variable G_t.

Each agent's **objective** is to maximize its own return, $G_{0,i}$ or an *expected return* $\mathbb{E}_{z \sim \pi}[G_0, i]$. However, note that the trajectory sampled depends not just on player i's policy but on *every other player's policies*! So, an agent cannot maximize its return in isolation: it *must* consider the other agents as part of its optimization problem. This is fundamentally different from traditional (single-agent) reinforcement learning, and the main challenge of multiagent RL.

3.2. Algorithms and Results

3.2.1. Basic Algorithms

Suppose players are playing with a joint policy π. The expected returns algorithm computes $\mathbb{E}_{\pi}[G_{0,i}]$ for all players $i \in \mathcal{N}$ exactly, by doing a tree traversal over the game and querying the policy at each state s. Similarly, for small enough games, one can get all the states ($\mathcal{S}$) in a game by doing a tree traversal and indexing each state by its information state string description.

The trajectories algorithms run a batch of episodes by following a joint policy π, collecting various data such as the states visited, state policies, actions sampled, returns, episode lengths, etc., which could form the basis of the data collection for various RL algorithms.

There is a simple implementation of value iteration. In single-agent games, it is identical to the standard algorithm [75]. In two-player turn-taking zero-sum games, the values for state s, i.e. $V(s)$, is stored in view of the player to play at s, i.e. $V_{\tau(s)}(s)$. This can be solved by applying the identities $V_1(s) = -V_2(s)$ and $r_1(s, a, s') = -r_2(s, a, s')$.

3.2.2. Search Algorithms

There are two classical search algorithms for zero-sum turn-taking games of perfect information: minimax (and alpha-beta) search [34, 67], and Monte Carlo tree search (MCTS) [21, 35, 18].

Suppose one wants to choose at some root state s_{root} : given a heuristic value function for $v_{0,i}(s)$ (representing the value of state s to player i) and some depth d, minimax search computes a policy $\pi(s)$ that assigns 1 to an action that maximizes the following depth-limited adversarial multistep value backup:

$$v_d(s) = \begin{cases} v_{0,\tau s_{root}}(s) & \text{if } d = 0; \\ \max_{a \in \mathcal{A}(s)} v_{d-1}(\mathcal{T}(s, a)) & \text{if } \tau(s) = i; \\ \min_{a \in \mathcal{A}(s)} v_{d-1}(\mathcal{T}(s, a)) & \text{if } \tau(s) \neq i, \end{cases}$$

where here we treat $\mathcal{T}(s, a) = s'$ as a deterministic map for the successor state reached from taking action a in state s.

The Python implementation of minimax includes expectiminimax [49] as well, which also backs up expected values at chance nodes. Alpha-beta style cut-offs could also be applied using $*$-minimax [6], but it is not currently implemented.

The implementations of MCTS are vanilla UCT with random playouts. Chance node are supported and represented explicitly in the tree: at chance nodes, the tree policy is always to sample according to the chance node's probability distibution.

3.2.3. Optimization Algorithms

OpenSpiel includes some basic optimization algorithms applied to games, such as solving zero-sum matrix games ([71, Section 4], [45]) and sequence-form linear programming for two-player zero-sum extensive-form games ([36] and [71, Section 5]), and an algorithm to check whether an action is dominated by a mixture of other strategies in a normal-form [71, Sec 4.5.2].

3.2.4. Traditional Single-Agent RL Algorithms

We currently have three algorithms usable for traditional (single-agent) RL: Deep Q-Networks (DQN) [52], Advantage Actor-Critic (A2C) [50], and Ephemeral Value Adjustments (EVA) [26]. Each algorithm will operate as the standard one in single-agent environments.

Each of these algorithms can also be run in the multiagent setting, in various ways. The default is that each player is independently running a copy of the algorithm with states and observations that include what other players did. The other way to use these algorithms is to compute an approximate best response to a fixed set of other players' policies, described in Section 3.2.5.

The main difference between the implementations of these algorithms and other standard ones is that these are aware that only a subset of actions are legal / illegal. So, for example, in Q-learning the value update for a transition (s, a, s') and policy updates are:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s', a') - Q(s, a)), \tag{1}$$

$$\pi(s, a) = \begin{cases} 0 & \text{if } a \notin \mathcal{A}(s); \\ 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = \text{argmax}_{a' \in \mathcal{A}(s)} Q(s, a'); \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{otherwise.} \end{cases} \tag{2}$$

Note that the actions are in the set of *legal actions* $\mathcal{A}(s)$ and $\mathcal{A}(s')$ rather than assuming that every action is legal at every state. For policy gradient methods, a masked softmax is used to set the logits of the illegal actions to $-\infty$ to force the policy to sets probability zero to illegal actions.

3.2.5. Partially-Observable (Imperfect Information) Games

There are many algorithms for reinforcement learning in partially-observable (zero-sum) games, as this is the focus of the core team's research interests.

Best Response and NashConv

Suppose π is a joint policy. A **best response** policy for player i is a policy that maximized player i's return against the other players' policies (π_{-i}). There may be many best responses, and we denote the set of such best responses,

$$BR(\pi_{-i}) = \{\pi_i' \mid \pi_i' = \operatorname*{argmax}_{\pi_i} u_i(\pi_i, \pi_{-i})\}.$$

Let $\delta_i(\pi)$ be the incentive for player i to deviate to one of its best responses: $\delta_i(\pi) = u_i(\pi_i^b, \pi_{-i}) - u_i(\pi)$, where $\pi_i^b \in BR(\pi_{-i})$. An approximate **ϵ-Nash equilibrium** is a joint policy such that $\delta_i(\pi) \leq \epsilon$ for all $i \in \mathcal{N}$, where a Nash equilibrium is obtained at $\epsilon = 0$.

A common metric for determining the rates of convergence (to equilibria) of algorithms in practice is:

$$\textsc{NashConv}(\pi) = \sum_{i \in \mathcal{N}} \delta_i(\pi).$$

In two-player constant-sum (i.e. k-sum) games, a similar metric has been used:

$$\textsc{Exploitability}(\pi) = \frac{\textsc{NashConv}(\pi)}{|\mathcal{N}|} = \frac{\sum_{i \in \mathcal{N}} \delta_i(\pi)}{n} = \frac{u_1(\pi_1^b, \pi_2) + u_2(\pi_1, \pi_2^b) - k}{2},$$

where $\pi_i^b \in BR(\pi_{-i})$. Nash equilibria are often considered optimal in two-player zero-sum games, because they guarantee maximal worst-case returns against any other opponent policy. This is also true for approximate equilibria, so convergence to equilibra has been a focus in this class of games.

Fictitious Play and Best Response-Based Iterative Algorithms

Fictitious play (FP) is a classic iterative procedure for computing policies in (normal-form) games [14, 65]. Starting with a uniform random policy at time $t = 0$. Then, for $t \in \{1, 2, \cdots\}$, do:

1. Each player computes a best response to the opponents' average policy: $\pi_i^t \in BR(\bar{\pi}_{-i}^{t-1})$.
2. Each player updates their average policy: $\bar{\pi}_i^t = \frac{(t-1)\bar{\pi}_i^{t-1} + \pi_i^t}{t}$.

OpenSpiel has an implementation of extensive-form fictitious play (XFP) [29], which is equivalent to the classical fictitious play. To run it on normal-form games, the game needs to be transformed into a turn-based game using `TurnBasedSimultaneousGame` in `game_transforms/`. Fictitious Self-Play is a sampled-based RL version of XFP that uses supervised learning to learn the average policy and reinforcement learning to compute approximate best responses. Neural Fictitious Self-Play (NFSP) scales these ideas using neural networks and a reservoir-sampled buffer to maintain a uniform sample of experience to train the average policy [30].

The average policy in fictitious play can be described equivalently as a meta-policy that assigns uniform weight over all the previous best response policies, and each iteration computes a best response to the opponents' meta-policies. Policy-Space Response Oracles (PSRO) generalizes fictitious play and the double-oracle algorithm [41, 48] by analyzing this meta-game using empirical game-theoretic analysis [82]. Exploitabiliy Descent replaces the second step of fictitious play with a policy gradient ascent against the state-action values given the opponents play their best responses [46]. This one change allows convergence of the policies themselves rather than having to maintain an average policy; in addition, it makes the optimization of the polices amenable to RL-style general function approximation.

A convergence curve for XFP and ED are shown in Figure 1. A convergence curve for NFSP in 2-player Leduc is found below (Figure 3), included with the policy gradient methods.

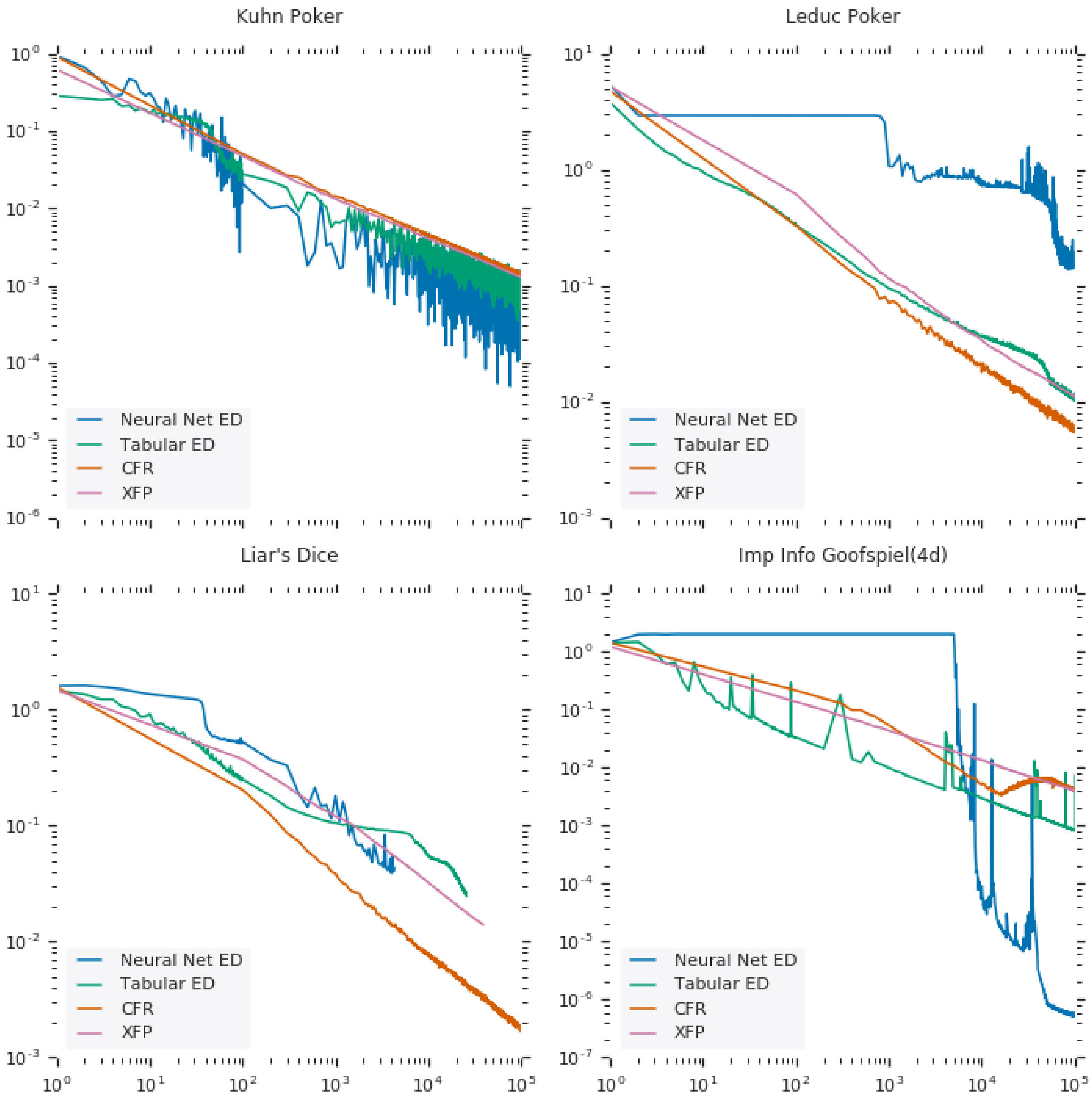


Figure 1 | Convergence rates of XFP and ED algorithms on various partially-observable games in OpenSpiel. The units of the x-axis is iterations and the units of the y-axis is NashConv. Figure taken from [46].

Counterfactual Regret Minimization

Counterfactual regret (CFR) minimization is a policy iteration algorithm for computing approximate equilibra in two-player zero-sum games [86]. It has revolutionized Poker AI research [66, 68], leading to the largest variants of poker being solved and competitive polices that have beat top human professionals [12, 53, 16, 17].

CFR does two main things: (a) define a new notion of state-action value, the counterfactual value, and (b) define a decomposed regret minimization procedure (based on these values) at every information state that, together, leads to minimization of overall average regret. This means that the average policy of two CFR players approaches an approximate equilibrium.

Define $\mathcal{Z}(s)$ as the set of terminal histories that pass through s, paired with the prefix of each terminal $h \sqsubset z$. Define a reach probability $\eta^{\pi}(h)$ to be the product of all players' probabilities of state-action pairs along h (including chance's), which can be decomposed into player i's contribution and their opponents' contributions: $\eta^{\pi}(h) = \eta^{\pi}_i(h)\eta^{\pi}_{-i}(h)$. Similarly define $\eta^{\pi}(h, z)$ similarly from h to z and ha as the history h appended with action a. The counterfactual state-action value for $i = \tau(s)$ is:

$$q^c_{\pi,i}(s,a) = \sum_{(h,z)\in\mathcal{Z}(s)} \eta^{\pi}_{-i}(h)\eta^{\pi}(ha,z)u_i(z).$$

The state value is then $v^c_{\pi,i}(s) = \sum_{h\in s} \pi(s,a) q^c_{\pi,i}(s,a)$.

CFR starts with a uniform random policy π^0 and proceeds by applying regret minimization at every information state independently. Define $r^t(s,a) = q^c_{\pi^t,i}(s,a) - v^c_{\pi^t,i}(s)$ to be the instantaneous **counterfactual regret**. CFR proceeds by minimizing this regret, typically using regret-matching [27]. A table of cumulative regret is maintained $R^t(s,a) = \sum_t r^t(s,a)$, and the policy at each state is updated using:

$$\pi^{t+1}(s,a) = \begin{cases} \frac{R^{t,+}(s,a)}{\sum_{a\in\mathcal{A}(s)} R^{t,+}(s,a)} & \text{if the denominator is positive;} \\ \\ \frac{1}{|\mathcal{A}(s)|} & \text{otherwise,} \end{cases}$$

where $x^+ = \max(x, 0)$.

In addition to basic CFR, OpenSpiel contains a few variants of Monte Carlo CFR [39] such as outcome sampling and external sampling, and CFR+ [77].

Regression CFR

Regression CFR (RCFR) was the first variant to combine RL-style function approximation with CFR techniques [81, 54]. The main idea is to train a regressor to predict the cumulative or average counterfactual regrets, $\hat{R}^t(s,a) \approx R^t(s,a)$ or $\bar{r}'^t(s,a) \approx {}^{R^t(s,a)}/_t$, instead of reading them from a table. The original paper used domain-specific features and regression trees. The implementation in OpenSpiel uses neural networks with raw inputs obtained by each game's `InformationSetAsNormalizedVector` bit string.

Figure 2 shows the convergence rate of RCFR compared to a tabular CFR.

Deep CFR [15] applies these ideas to a significantly larger game using convolutional networks, external sampling Monte Carlo CFR, and–like NFSP–a reservoir-sampled buffer.

Regret Policy Gradients

Value-based RL algorithms, such as temporal-difference learning and Q-learning, *evaluate* a policy π by computing or estimating state (or state-action) values that represent the expected return conditioned on having reached state s,

$$v_{\pi}(s_t) = \mathbb{E}_{\pi}[G_t | S_t = s].$$

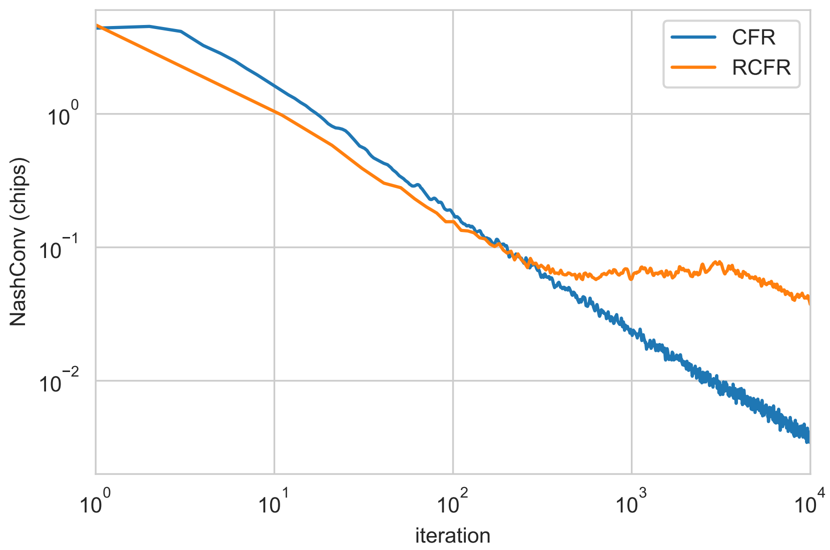


Figure 2 | Convergence rate of RCFR in Leduc poker using a 2-layer network with 400 hidden units in each layer. The average policy is computed exactly (i.e. tabular), and regression targets are the cumulative predicted regrets.

Policies are *improved* by choosing the actions that lead to higher-valued states or higher-valued returns.

In episodic partially-observable games, when agents have perfect recall (Def 5), there is an important connection between traditional values in value-based RL and counterfactual values [74, Section 3.2]:

$$v_{\pi,i}(s) = \frac{v^c_{\pi,i}(s)}{\beta_{-i}(\pi, s)},$$

where $\beta_{-i}(s) = \sum_{h \in s} \eta^{\pi}_{-i}(h)$ is the Bayes normalization term to ensure that $\Pr(h|s)$ is a probability distribution. CFR is then as a (tabular) all-actions policy gradient algorithm with generalized infinitesimal gradient ascent (GIGA) at each state [74], inspiring new RL variants for partially observable games.

These variants: Q-based "all-actions" Policy Gradient (QPG), Regret Policy Gradients (RPG), and Regret-Matching Policy Gradients (RMGP) are included in OpenSpiel, along with classic batched A2C. RPG differs from QPG in that the policy is optimized toward a no-regret region, minimizing the loss based on $r^+(s, a)$, the motivation being that a policy with zero regret is, by definition, an equilibrium policy. Convergence results for these algorithms are show in Figure 3.

Neural Replicator Dynamics

Neural Replicator Dynamics (NeuRD) [57] takes the policy gradient connection to CFR a step further: in [74], the relationship between policy gradients and CFR was possible via GIGA [85]; however, this requires ℓ_2 projections of policies after the gradient step. NeuRD, on the other hand, works directly with the common softmax-based policy representations. Instead of differentiating through the softmax as policy gradient does, NeuRD differentiates only with respect to the logits. This is equivalent to updating the policy of a parameterized replicator dynamics from evolutionary game theory [31, 69] using an Euler discretization. The resulting update reduces to the well-known multiplicative weights update algorithm or Hedge [24], which minimizes regret. Hence, NeuRD in partially-observable games can replace regret-matching in CFR and retain convergence guarantees in the tabular case since that algorithm reduces to CFR with Hedge.

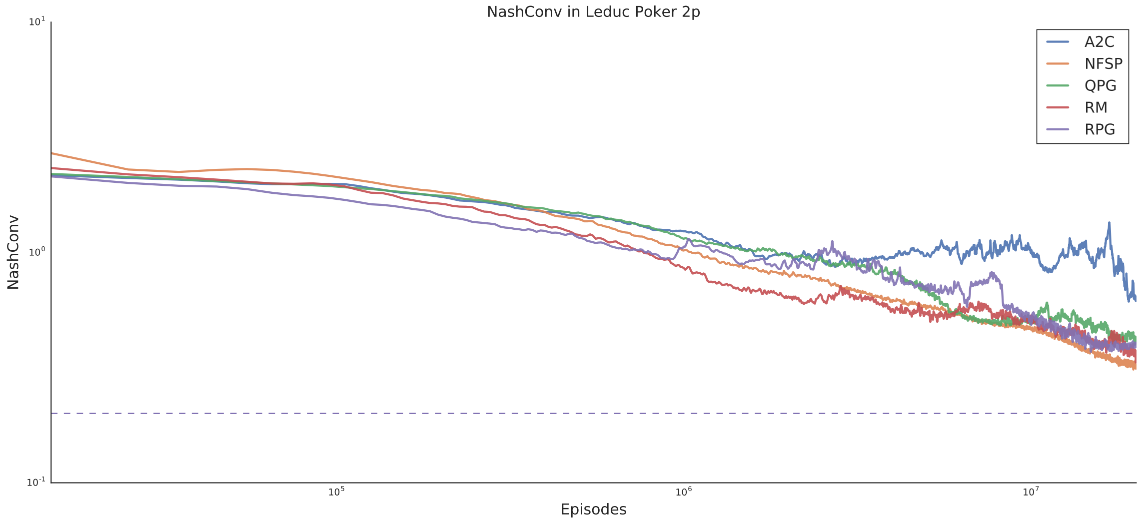


Figure 3 | Convergence rates of NFSP and various (regret-based) policy gradient algorithms in 2-player Leduc poker. Each line is an average over the top five seeds and hyperparemeter settings for each algorithm. The lowest (around 0.2) NashConv value reached by any individual run is depicted by a dashed line.

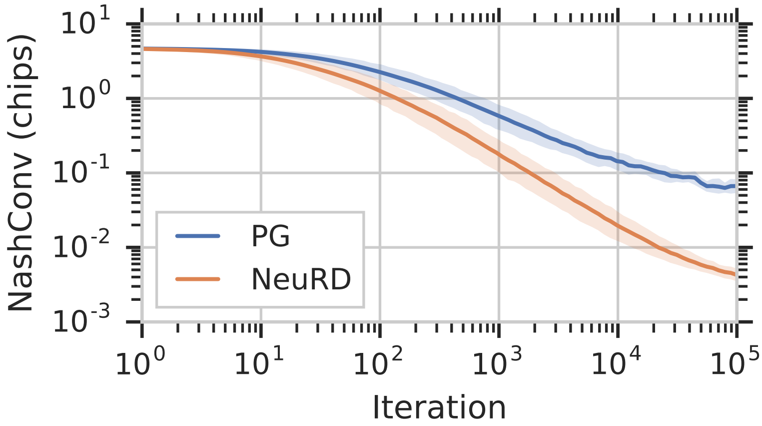


Figure 4 | NashConv of tabular all-actions NeuRD versus tabular all-action policy gradient (policy gradient policy iteration) in Leduc poker. Figure taken from [57].

One practical benefit is that the NeuRD policy updates are not weighted by the policy like policy gradient is. As a result, in non-stationary domains, NeuRD is also more adaptive to changes in the environment. Results for NeuRD are show in Figures 4 and 5.

3.3. Tools and Evaluation

OpenSpiel has a few tools for visualization and evaluation, though some would also be considered algorithms (such as α-Rank). The best response algorithm is also a tool in some sense, but is listed in Section 2 due to its association with partially-observable games.

For now, all the tools and evaluation we mention in this section is contained under the `python/egt` and `python/visualizations` subdirectories of the code base.

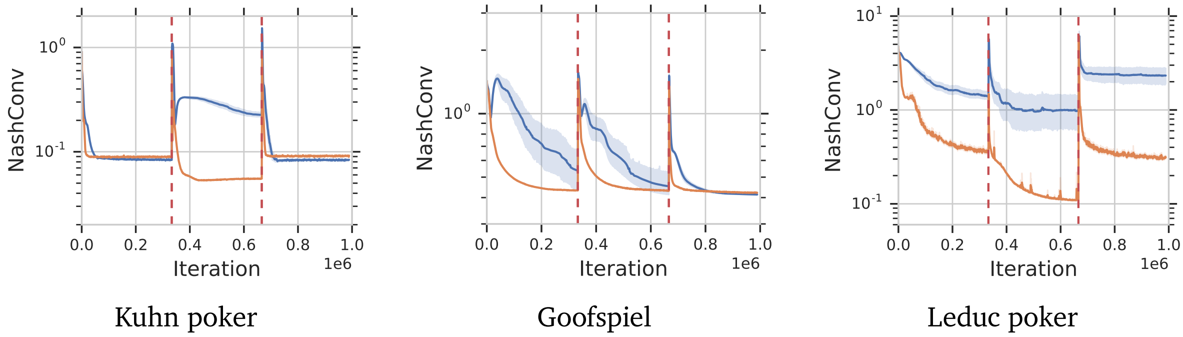


Figure 5 | NashConv of NeuRD using sampling trajectories and function approximation. The games are played in three phases where, between phases, the returns are inverted. NeuRD is the yellow (bottom) line, which policy gradient is the blue (top) line. Figure taken from [57].

3.3.1. *Visualizing a Game Tree*

A game tree can be visualized by using Graphviz. An example is shown in Fig 6.

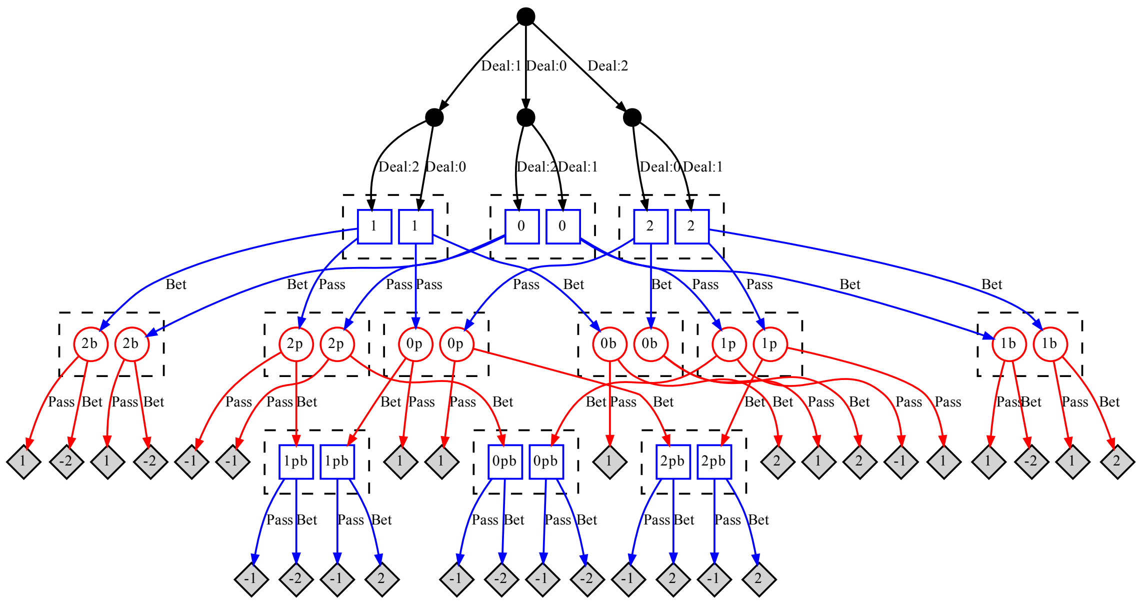


Figure 6 | A visualization of Kuhn poker generated by `python/examples/visualization_example.py`. Black, blue, and red edges correspond to chance, first player, and second player outcomes/actions, respectively. Nodes correspond to histories h and are labeled by their information state strings, and dotted boxes group these histories $h \in s$ by their information state s. Diamonds correspond to terminal states which are labeled by the utility to the first player.

3.3.2. *Visualization of Evolutionary and Policy Learning Dynamics*

One common visualization tool in the multiagent learning literature (especially in games) is a **phase portrait** that shows a vector field and/or trajectories of particle that depict local changes to the policy under specific update dynamics [72, 80, 13, 79, 11, 82, 2, 84, 83, 9, 78].

For example, consider the well-known single-population replicator dynamic for symmetric games, where each player follows a learning dynamic described by:

$$\frac{\partial \pi_t(a)}{\partial t} = \pi_t(a)\left(u(a, \boldsymbol{\pi}_t) - \bar{u}(\boldsymbol{\pi}_t)\right) \qquad \forall a \in \mathcal{A},$$

where $u(a, \boldsymbol{\pi}_t)$ represents the expected utility of playing action a against the full policy $\boldsymbol{\pi}_t$, and $\bar{u}(\boldsymbol{\pi}_t)$ is the expected value over all actions $\sum_{a \in \mathcal{A}} \pi_t(a) u(a, \boldsymbol{\pi}_t)$.

Figure 7 shows plots generated from OpenSpiel for replicator dynamics in the game of Rock–Paper–Scissors. Figure 8 shows plots generated from OpenSpiel for four common bimatrix games.

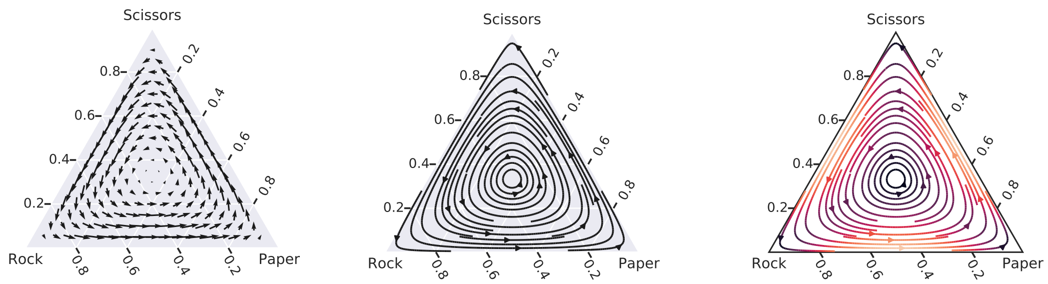


Figure 7 | Phase portraits of single-population replicator dynamics in *Rock–Paper–Scissors*. The colored plot shows the relative magnitude of the dynamics.

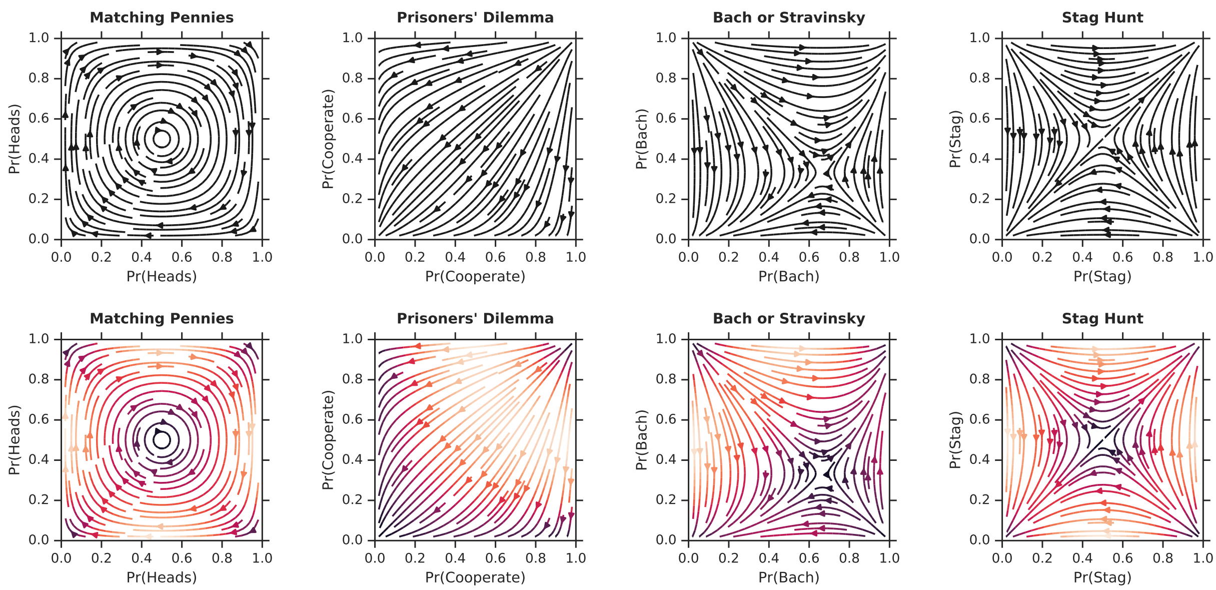


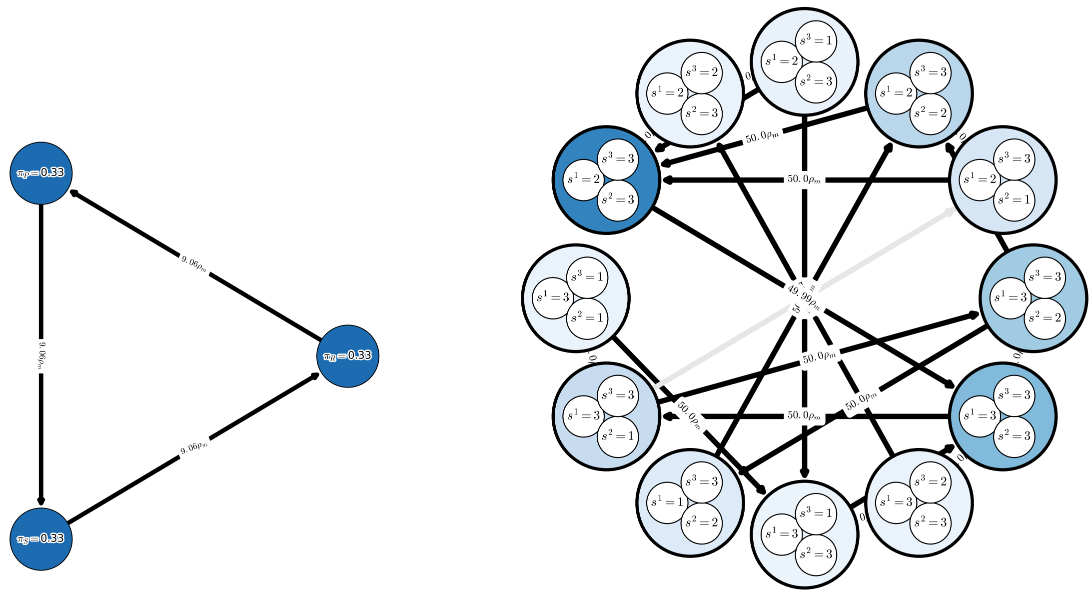
Figure 8 | Phase portraits of the two-population replicator dynamics for four common bimatrix games. The colored plots shows the relative magnitude of the vectors.

3.3.3. α-Rank

α-Rank [58] is an algorithm that leverages evolutionary game theory to rank AI agents interacting in multiplayer games. Specifically, α-Rank defines a Markov transition matrix with states corresponding to the profile of agents being used by the players (i.e., tuples of AI agents), and transitions informed by a specific evolutionary model that ensures correspondence of the rankings to a game-theoretic solution concept known as a Markov-Conley Chain. A key benefit of α-Rank is that it can rank agents in scenarios involving intransitive agent relations (e.g., the agents Rock, Paper, and Scissors in the eponymous game), unlike the Elo rating system [5]; an additional practical benefit is that it is also tractable to compute in general games, unlike ranking systems relying on Nash equilibria [22].

OpenSpiel currently supports using α-Rank for both single-population (symmetric) and multi-population games. Specifically, users may specify games via payoff tables (or tensors for the >2 players case) as well as Heuristic Payoff Tables (HPTs). Note that here we only include an overview of the technique and visualizations; for a tour through the usage and code please see the α-Rank doc on the web site.

Figure 9(a) shows a visualization of the Markov transition matrix of α-Rank run on the Rock, Paper, Scissors game. The next example demonstrates computing α-Rank on an asymmetric 3-player meta-game, constructed by computing utilities for Kuhn poker agents from the best response policies generated in the first few rounds of via extensive-form fictitious play (XFP) [29]. The result is shown in Figure 9(b).



(a) (b)

Figure 9 | (a) Markov transitions matrix of solution found by α-Rank on Rock, Paper, Scissors. (b) Markov transitions matrix of meta-game computed by the first few rounds of XFP in 3-player Kuhn poker.

One may choose to conduct a sweep over the ranking-intensity parameter, α (as opposed to choosing a fixed α). This is useful for general games where bounds on utilities may be unknown, and where the ranking computed by α-Rank should use a sufficiently high value of α (to ensure correspondence to the underlying Markov-Conley Chain solution concept). In such cases, the following interface can be used to both visualize the sweep and obtain the final rankings computed. The result is shown in Figure 10.

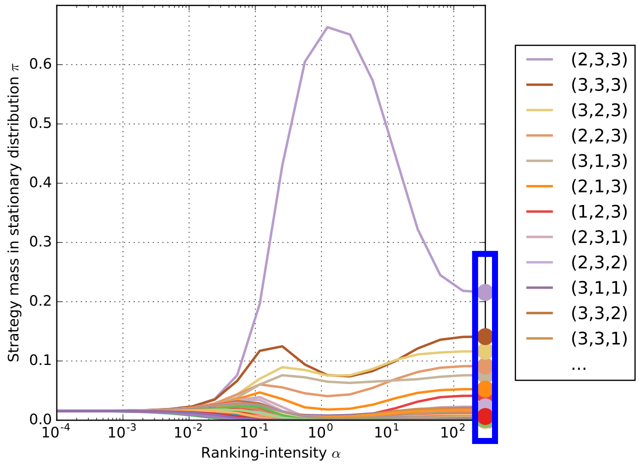


Figure 10 | Effect of ranking-intensity parameter α on policy mass in stationary distribution in meta-game generated by XFP in 3-player Kuhn poker.

4. Guide to Contributing

If you are looking for ideas on potential contributions or want to see a rough road map for the future of OpenSpiel, please visit the Roadmap and Call for Contributions on github.

Before making a contribution to OpenSpiel, please read the design philosophy in Section 3. We also kindly request that you contact us before writing any large piece of code, in case (a) we are already working on it and/or (b) it's something we have already considered and may have some design advice on its implementation. Please also note that some games may have copyrights which could require legal approval(s). Otherwise, happy hacking!

4.1. Contacting Us

If you would like to contact us regarding anything related to OpenSpiel, please create an issue on the github site so that the team is notified, and so that the responses are visible to everyone.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Sherief Abdallah and Victor Lesser. A multiagent reinforcement learning algorithm with non-linear dynamics. *JAIR*, 33(1):521–549, 2008.

[3] David Auger. Multiple tree for partially observable Monte-Carlo tree search. In *Applications of Evolutionary Computation (EvoApplications 2011), Part I*, volume 6624 of *LNCS*, pages 53–62, 2011.

[4] David Balduzzi, Marta Garnelo, Yoram Bachrach, Wojciech M. Czarnecki, Julien Pérolat, Max Jaderberg, and Thore Graepel. Open-ended learning in symmetric zero-sum games. *CoRR*, abs/1901.08106, 2019. http://arxiv.org/abs/1901.08106.

[5] David Balduzzi, Karl Tuyls, Julien Perolat, and Thore Graepel. Re-evaluating evaluation. In *Advances in Neural Information Processing Systems*, pages 3268–3279, 2018. Also available at http://arxiv.org/abs/1806.02643.

[6] B. W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350, 1983.

[7] Nolan Bard, Jakob N. Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H. Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, Iain Dunning, Shibl Mourad, Hugo Larochelle, Marc G. Bellemare, and Michael Bowling. The hanabi challenge: A new frontier for AI research. *CoRR*, abs/1902.00506, 2019. http://arxiv.org/abs/1902.00506.

[8] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013.

[9] Daan Bloembergen, Karl Tuyls, Daniel Hennes, and Michael Kaisers. Evolutionary dynamics of multi-agent learning: A survey. *J. Artif. Intell. Res. (JAIR)*, 53:659–697, 2015.

[10] Branislav Bošanský, Viliam Lisý, Marc Lanctot, Jiří Čermák, and Mark H.M. Winands. Algorithms for computing strategies in two-player simultaneous move games. *Artificial Intelligence*, 237:1–40, 2016.

[11] Michael Bowling. Convergence and no-regret in multiagent learning. In *Advances in Neural Information Processing Systems 17 (NIPS)*, pages 209–216, 2005.

[12] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up Limit Hold'em Poker is solved. *Science*, 347(6218):145–149, January 2015.

[13] Michael Bowling and Manuela Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136:215–250, 2002.

[14] G. W. Brown. Iterative solutions of games by fictitious play. In T.C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 374–376. John Wiley & Sons, Inc., 1951.

[15] Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization. In *Proceedings of the Thirty-Sixth International Conference on Machine Learning (ICML)*, pages 793–802, 2019. Also available at https://arxiv.org/abs/1811.00164.

[16] Noam Brown and Tuomas Sandholm. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science*, 360(6385), December 2017.

[17] Noam Brown and Tuomas Sandholm. Superhuman AI for multiplayer poker. *Science*, 11, 2019.

[18] C.B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[19] M. Buro. Solving the oshi-zumo game. In Van Den Herik H.J., Iida H., and Heinz E.A., editors, *Advances in Computer Games*, volume 135 of *IFIP - The International Federation for Information Processing*. Springer, 2004.

[20] Kris Cao, Angeliki Lazaridou, Marc Lanctot, Joel Z. Leibo, Karl Tuyls, and Stephen Clark. Emergent communication through negotiation. In *Proceedings of the Sixth International Conference on Learning Representations (ICLR)*, 2018. Also available at https://arxiv.org/abs/1804.03980.

[21] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Proceedings of the 5th international conference on Computers and games*, volume 4630 of *CG'06*, pages 72–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[22] Constantinos Daskalakis, Paul W Goldberg, and Christos H Papadimitriou. The complexity of computing a Nash equilibrium. *SIAM Journal on Computing*, 39(1):195–259, 2009.

[23] Jakob N. Foerster, H. Francis Song, Edward Hughes, Neil Burch, Iain Dunning, Shimon Whiteson, Matthew Botvinick, and Michael Bowling. Bayesian action decoder for deep multi-agent reinforcement learning. *CoRR*, abs/1811.01458, 2018. Also available at http://arxiv.org/abs/1811.01458.

[24] Y. Freund and R. E. Shapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory: Second European Conference (EuroCOLT'95)*, pages 23–37. Springer-Verlag, 1995.

[25] M. Genesereth, N. Love, and B. Pell. General game-playing: Overview of the AAAI competition. *AI Magazine*, 26:62–72, 2005.

[26] Steven Hansen, Pablo Sprechmann, Alexander Pritzel, André Barreto, and Charles Blundell. Fast deep reinforcement learning using online adjustments from the past. *CoRR*, abs/1810.08163, 2018. http://arxiv.org/abs/1810.08163.

[27] S. Hart and A. Mas-Colell. A simple adaptive procedure leading to correlated equilibrium. *Econometrica*, 68(5):1127–1150, 2000.

[28] He He, Jordan L. Boyd-Graber, Kevin Kwok, and Hal Daumé III. Opponent modeling in deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning (ICML 2016)*, 2016. Preprint available at https://arxiv.org/abs/1609.05559.

[29] Johannes Heinrich, Marc Lanctot, and David Silver. Fictitious self-play in extensive-form games. In *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*, 2015.

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

[31] Josef Hofbauer and Karl Sigmund. *Evolutionary Games and Population Dynamics*. Cambridge University Press, 1998.

[32] M. Johanson, N. Bard, N. Burch, and M. Bowling. Finding optimal abstract strategies in extensive form games. In *Proceedings of the Twenty-Sixth Conference on Artificial Intelligence (AAAI)*, pages 1371–1379, 2012.

[33] J. S. Jordan. Three problems in learning mixed-strategy Nash equilibria. *Games and Economic Behavior*, 5:368–386, 1993.

[34] Donald E. Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[35] L. Kocsis and C. Szepesvári. Bandit-based Monte Carlo planning. In *15th European Conference on Machine Learning*, volume 4212 of *LNCS*, pages 282–293, 2006.

[36] D. Koller, N. Megiddo, and B. von Stengel. Fast algorithms for finding randomized strategies in game trees. In *Proceedings of the 26th ACM Symposium on Theory of Computing (STOC '94)*, pages 750–759, 1994.

[37] Vojtech Kovarík, Martin Schmid, Neil Burch, Michael Bowling, and Viliam Lisý. Rethinking formal models of partially observable multiagent decision making. *CoRR*, abs/1906.11110, 2019. http://arxiv.org/abs/1906.11110.

[38] H. W. Kuhn. Simplified two-person Poker. *Contributions to the Theory of Games*, 1:97–103, 1950.

[39] M. Lanctot, K. Waugh, M. Bowling, and M. Zinkevich. Sampling for regret minimization in extensive games. In *Advances in Neural Information Processing Systems (NIPS 2009)*, pages 1078–1086, 2009.

[40] Marc Lanctot. *Monte Carlo Sampling and Regret Minimization for Equilibrium Computation and Decision-Making in Large Extensive Form Games*. PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, June 2013.

[41] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Perolat, David Silver, and Thore Graepel. A unified game-theoretic approach to multiagent reinforcement learning. In *Advances in Neural Information Processing Systems*, 2017. Also available at https://arxiv.org/abs/1711.00832.

[42] Joel Z. Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. Multi-agent reinforcement learning in sequential social dilemmas. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2017. Preprint available at https://arxiv.org/abs/1702.03037.

[43] Mike Lewis, Denis Yarats, Yann Dauphin, Devi Parikh, and Dhruv Batra. Deal or no deal? End-to-end learning of negotiation dialogues. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2433–2443, 2017. Also available at https://arxiv.org/abs/1706.05125.

[44] V. Lisy. Alternative selection functions for information set Monte Carlo tree search. *Acta Polytechnica: Journal of Advanced Engineering*, 54(5):333–340, 2014.

[45] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *In Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163. Morgan Kaufmann, 1994.

[46] Edward Lockhart, Marc Lanctot, Julien Pérolat, Jean-Baptiste Lespiau, Dustin Morrill, Finbarr Timbers, and Karl Tuyls. Computing approximate equilibria in sequential adversarial games by exploitability descent. In *In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2019. See also full version at https://arxiv.org/abs/1903.05614.

[47] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[48] H. McMahan, G. Gordon, and A. Blum. Planning in the presence of cost functions controlled by an adversary. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*, 2003.

[49] D. Michie. Game-playing and game-learning automata. *Advances in Programming and Non-Numerical Computation*, pages 183–200, 1966.

[50] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, pages 1928–1937, 2016.

[51] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent models of visual attention. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2204–2212. Curran Associates, Inc., 2014.

[52] 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*, 518:529–533, 2015.

[53] 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*, 358(6362), October 2017.

[54] Dustin Morrill. Using regret estimation to solve games compactly. Master's thesis, Computing Science Department, University of Alberta, April 2016.

[55] Todd W. Neller and Marc Lanctot. An introduction to counterfactual regret minimization. In *Proceedings of Model AI Assignments, The Fourth Symposium on Educational Advances in Artificial Intelligence (EAAI-2013)*, 2013. http://modelai.gettysburg.edu/2013/cfr/index.html.

[56] Todd W. Neller and Clifton G.M. Presser. Optimal play of the dice game pig. *The UMAP Journal*, 25(1):25–47, 2004.

[57] Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Rémi Munos, Julien Pérolat, Marc Lanctot, Audrunas Gruslys, Jean-Baptiste Lespiau, and Karl Tuyls. Neural replicator dynamics. *CoRR*, abs/1906.00190, 2019. http://arxiv.org/abs/1906.00190.

[58] Shayegan Omidshafiei, Christos Papadimitriou, Georgios Piliouras, Karl Tuyls, Mark Rowland, Jean-Baptiste Lespiau, Wojciech M. Czarnecki, Marc Lanctot, Julien Perolat, and Remi Munos. α-rank: Multi-agent evaluation by evolution. *Scientific Reports*, 9(1):9937, 2019.

[59] Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepezvari, Satinder Singh, Benjamin Van Roy, Richard Sutton, David Silver, and Hado Van Hasselt. Behaviour suite for reinforcement learning. *CoRR*, abs/1908.03568, 2016. https://arxiv.org/abs/1908.03568.

[60] M.J. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.

[61] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[62] Julien Pérolat, Bilal Piot, Matthieu Geist, Bruno Scherrer, and Olivier Pietquin. Softened approximate policy iteration for markov games. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 1860–1868. JMLR.org, 2016.

[63] Jan Peters. Policy gradient methods for control applications. Technical Report TR-CLMC-2007-1, University of Southern California, 2002.

[64] Roberta Raileanu, Emily Denton, Arthur Szlam, and Rob Fergus. Modeling others using oneself in multi-agent reinforcement learning. *CoRR*, abs/1802.09640, 2018. http://arxiv.org/abs/1802.09640.

[65] J Robinson. An iterative method of solving a game. *Annals of Mathematics*, 54:296–301, 1951.

[66] J. Rubin and I. Watson. Computer poker: A review. *Artificial Intelligence*, 175(5–6):958–987, 2011.

[67] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[68] T. Sandholm. The state of solving large incomplete-information games, and application to poker. *AI Magazine*, 31(4):13–32, 2010.

[69] William H. Sandholm. *Population Games and Evolutionary Dynamics*. MIT Press, 2010.

[70] Sven Seuken and Shlomo Zilberstein. Improved memory-bounded dynamic programming for decentralized pomdps. *CoRR*, abs/1206.5295, 2012. http://arxiv.org/abs/1206.5295.

[71] Y. Shoham and K. Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

[72] Satinder P. Singh, Michael J. Kearns, and Yishay Mansour. Nash convergence of gradient dynamics in general-sum games. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, UAI '00, pages 541–548, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[73] Finnegan Southey, Michael Bowling, Bryce Larson, Carmelo Piccione, Neil Burch, Darse Billings, and Chris Rayner. Bayes' bluff: Opponent modelling in poker. In *Proceedings of the 21st Annual Conference on Uncertainty in Artificial Intelligence (UAI*, pages 550–558, 2005.

[74] Sriram Srinivasan, Marc Lanctot, Vinicius Zambaldi, Julien Perolat, Karl Tuyls, Remi Munos, and Michael Bowling. Actor-critic policy optimization in partially observable multiagent environments. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3422–3435. Curran Associates, Inc., 2018. Full version available at https://arxiv.org/abs/1810.09026.

[75] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[76] Richard S. Sutton, Satinder Singh, and David McAllester. Comparing policy-gradient algorithms, 2001. Unpublished.

[77] Oskari Tammelin, Neil Burch, Michael Johanson, and Michael Bowling. Solving heads-up limit Texas Hold'em. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, 2015.

[78] Karl Tuyls, Julien Perolat, Marc Lanctot, Joel Z Leibo, and Thore Graepel. A Generalised Method for Empirical Game Theoretic Analysis . In *AAMAS*, 2018.

[79] W. E. Walsh, D. C. Parkes, and R. Das. Choosing samples to compute heuristic-strategy Nash equilibrium. In *Proceedings of the Fifth Workshop on Agent-Mediated Electronic Commerce*, 2003.

[80] William E Walsh, Rajarshi Das, Gerald Tesauro, and Jeffrey O Kephart. Analyzing Complex Strategic Interactions in Multi-Agent Systems. In *AAAI*, 2002.

[81] Kevin Waugh, Dustin Morrill, J. Andrew Bagnell, and Michael Bowling. Solving games with functional regret estimation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2015. https://arxiv.org/abs/1411.7974.

[82] Michael P. Wellman. Methods for empirical game-theoretic analysis. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, pages 1552–1556, 2006.

[83] Michael Wunder, Michael Littman, and Monica Babes. Classes of multiagent q-learning dynamics with ϵ-greedy exploration. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 1167–1174, 2010.

[84] Chongjie Zhang and Victor Lesser. Multi-agent learning with policy prediction. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 927–934, 2010.

[85] M. Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of Twentieth International Conference on Machine Learning (ICML-2003)*, 2003.

[86] M. Zinkevich, M. Johanson, M. Bowling, and C. Piccione. Regret minimization in games with incomplete information. In *Advances in Neural Information Processing Systems 20 (NIPS 2007)*, 2008.