ADVERSARIAL POLICY GRADIENT FOR ALTERNATING MARKOV GAMES

Anonymous authors
Paper under double-blind review

ABSTRACT

Policy gradient reinforcement learning has been applied to two-player alternate-turn zero-sum games, e.g., in AlphaGo, self-play REINFORCE was used to improve the neural net model after supervised learning. In this paper, we emphasize that two-player zero-sum games with alternating turns, which have been previously formulated as Alternating Markov Games (AMGs), are different from standard MDP because of its two-agent nature. We exploit the difference in associated Bellman-equations, which leads to different policy iteration algorithms. In principle, as policy gradient method is a kind of generalized policy iteration, we show how these differences in policy iteration are reflected in policy gradient for AMGs. We reformulate an adversarial policy gradient and discuss potential possibilities for developing better policy gradient methods other than self-play REINFORCE. Specifically, the core idea is to estimate the *minimum* rather than the *mean* for the "critic". Experimental results on the game of Hex show that our newly developed Monte-Carlo policy gradient methods are able to learn better pure neural net policy than the REINFORCE variants. We also describe a board-size independent neural net architecture. As a by product, we show that when combined with search, using a single neural net model, the resulting program consistently beats MoHex 2.0, the state-of-the-art multi-boardsize Hex program after 2013, on board sizes from 9×9 to 13×13. Furthermore, we show that integrating *minimum* playout return to Monte-carlo tree search provides additional improvements.

1 INTRODUCTION

Reinforcement Learning (RL) is a trial-and-error paradigm where an agent learns by interacting with an environment. Two problems are involved in reinforcement learning, the *prediction* problem seeks to learn the true value function of a following policy; the *control* problem aims to find an optimal policy that the long-term expected cumulative reward is maximized. The environment is assumed to be a Markov Decision Process (MDP) (Bellman, 1957), which manifests a Markov property. Compared to dynamic programming, RL methods have the advantage of being model-free which learn by treating the environment as a black box. Nevertheless, despite the impracticality in real applications, dynamical programming is theoretically fundamental to reinforcement learning (Bertsekas & Tsitsiklis, 1996). Indeed, almost all RL algorithms are a form of *generalized policy iteration*, which improve the policy while estimating a value function (Sutton & Barto, 2017).

Model-free reinforce learning methods have been successfully applied to many domains, including robotics (Kober et al., 2013), Atari games (Mnih et al., 2015; 2016; Schaul et al., 2016), and two-player board games (Tesauro, 1995; Silver et al., 2016). Most of RL algorithms fall into one of two categories: valued fitting and policy gradient. The value-fitting algorithms try to optimize a value function through iteratively minimizing a sequence of Bellman inconsistencies of observed states or state-action pairs, e.g., on-policy SARSA (Rummery & Niranjan, 1994) and off-policy Q-learning (Watkins & Dayan, 1992). The optimal deterministic policy is implicitly learned when the value function converges to optimal.

The primary disadvantage of value-fitting methods, such as Q-learning (Watkins & Dayan, 1992; Mnih et al., 2015; Wang et al., 2016; Mnih et al., 2016), is that they might be unstable when interacting with function approximation (Bertsekas & Tsitsiklis, 1996; Sutton et al., 2000). To gain stable behavior, often extra heuristic techniques (Lin, 1992; Schaul et al., 2016) and extensive hyper-

parameter tunning are required. By contrast, policy gradient methods explicitly represent a policy as a function of parameters: they learn through iteratively adjusting the parameterized policy by following estimation of gradients, thus converging to at least a local maximum (Peters & Bagnell, 2011). Policy gradients are applicable to continuous control (Silver et al., 2014; Schulman et al., 2015; 2017) or domains with large action space (Silver et al., 2016), whereas action-value learning methods often become infeasible (Lillicrap et al., 2016; Wang et al., 2017).

Both value and policy based reinforcement learning are well-studied in MDP, where a single agent learns by exploiting a stationary environment. In this paper, we focus on policy gradient methods for two-player zero-sum games played in alternating turns, i.e., Alternating Markov Games (AMGs) (Littman, 1996). AMGs are a specialization of Stochastic Games (Shapley, 1953) but a generalization of MDPs by allowing exactly two players pursing diametrical goals. Many popular two-player games are of such kind, such as chess, checkers, backgammon and Go. The restriction of zero-sum in AMGs makes it possible to define a shared reward function such that one agent tries to maximize while other agent tries to minimize it. Indeed, due to such a property, with a notion of *self-play* policy, standard reinforcement learning methods have been successful applied to AMGs (Tesauro, 1995; Silver et al., 2016) by simply negating the reward signals of the opponent's turns.

In this paper, we reexamine the justifications of adapting standard reinforcement learning methods to AMGs. We begin by reviewing the fundamental differences in corresponding Bellman-equations, from which we show how the resulting policy iterations algorithms are disparate. We reformulate an adversarial policy gradient objective specifically for AMGs, based on which we discussion potential opportunities for developing new policy gradient methods for AMGs. Specifically, we apply our approach to the game of Hex, we show by modifying REINFORCE to estimate the *minimum* rather than the *mean* return of a self-play policy, stronger pure neural net players are obtained.

2 FINITE MDP

The finite MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$ where: $\mathcal{S}$ is finite set of states, $\mathcal{A}$ is a set of actions, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function, and $\mathcal{P}$ defines the probabilistic transition among states. An agent lives in an environment characterized by the Markov property, i.e., $\Pr(s_{t+1}|s_1, a_1, \ldots, s_t, a_t) = \Pr(s_{t+1}|s_t, a_t)$. The agent learns by receiving reward signals from interacting with the environment. The goal is to maximize the expected cumulative reward: $\mathbb{E}\sum_{k=t}^{\infty} \gamma^{t-k} R_{t-k+1}$, where γ is a discounting factor $0 < \gamma \leq 1$ that controls the contribution of long-term and short-term rewards; $\gamma = 1$ is only possible in episodic tasks.

The basis for reinforcement learning in MDP is a set of *Bellman Equations*. For a given policy π:

$$v_\pi(s) = \sum_a \pi(s, a) \sum_{s'} p(s'|s, a)(r(s, a, s') + \gamma v_\pi(s')), \tag{1}$$

where $p(s'|s, a)$ is the transition function and $r(s, a, s')$ is the reward of taking a at s arriving s'.

It is also popular to use the notion of action-value function:

$$q_\pi(s, a) = \sum_{s'} p(s'|s, a)(r(s, a, s') + \gamma v_\pi(s')) \tag{2}$$

The above Bellman equations are the basis for *policy evaluation*, which is to approximate the true value function for a given π. This corresponds to the *prediction* problem in RL. The *Optimal Bellman Equation* is a recursive relation for the optimal policy:

$$v_*(s) = \max_a \sum_{s'} p(s'|s, a)(r(s, a, s') + \gamma v_*(s'), \tag{3}$$

$$q_*(s, a) = \sum_{s'} p(s'|s, a)(r(s, a, s') + \gamma \max_{a'} q_*(s', a')). \tag{4}$$

A value iteration procedure derived by the above optimal Bellman-equation converges to optimal (Bellman, 1957). Alternatively, it is also possible to derive a *policy iteration* (Howard, 1960; Bertsekas & Tsitsiklis, 1996) by combining Equation (1) and (3). In reinforcement learning, the precise model is usually assumed unknown, the learning typically interleaves *policy evaluation* and *policy improvement*, which is summarized as generalized policy iteration (Sutton & Barto, 2017).

3 ALTERNATING MARKOV GAMES

Alternating Markov Games are a specialization of Stochastic Games with only two players: it is a tuple $(\mathcal{S}_1, \mathcal{S}_2, \mathcal{A}_1, \mathcal{A}_2, \mathcal{R}, \mathcal{P}, \gamma)$ where $\mathcal{S}_1$ and $\mathcal{S}_2$ are respectively *this* and *other* agent's states, $\mathcal{A}_1$ and $\mathcal{A}_2$ signifies respectively the actions at each player's states. It is clear that MDP can be viewed as a special case of AMG by restricting $|\mathcal{S}_2| = 0$.

In general, since there are two players in AMG, the *policy evaluation* thus involves two policies, i.e., π_1 and π_2. Bellman equations for policy evaluation are:

$$\begin{cases} v_{\pi_1}(s) = \sum_a \pi_1(a|s) \sum_{s'} p(s'|s,a)(r(s,a,s') + \gamma v_{\pi_2}(s')), s \in \mathcal{S}_1 \text{ and } s' \in \mathcal{S}_2 \\ v_{\pi_2}(s) = \sum_a \pi_2(a|s) \sum_{s'} p(s'|s,a)(r(s,a,s') + \gamma v_{\pi_1}(s')), s \in \mathcal{S}_2 \text{ and } s' \in \mathcal{S}_1 \end{cases} \tag{5}$$

Action-value functions can also be defined:

$$\begin{cases} q_{\pi_1}(s,a) = \sum_{s'} p(s'|s,a)(r(s,a,s') + \gamma \sum_{a'} \pi_2(a'|s') q_{\pi_2}(s',a')), s \in \mathcal{S}_1 \text{ and } s' \in \mathcal{S}_2 \\ q_{\pi_2}(s,a) = \sum_{s'} p(s'|s,a)(r(s,a,s') + \gamma \sum_{a'} \pi_1(a'|s') q_{\pi_1}(s',a')), s \in \mathcal{S}_2 \text{ and } s' \in \mathcal{S}_1 \end{cases} \tag{6}$$

Further, suppose π_1 is the max player while π_2 is the min player, assuming the π_2 is an optimal "counter policy" with respect to π_1, then we may rewrite the above equation as:

$$\begin{aligned} q_{\pi_1}(s,a) = \sum_{s'} p(s'|s,a) \Bigg\{ & r(s,a,s') + \gamma \min_{a'} \sum_{s''} p(s''|s',a') \\ & \left[r(s',a',s'') + \sum_{a''} \pi_1(a''|s'') q_{\pi_1}(s'',a'') \right] \Bigg\}, \\ & \text{where } s \in \mathcal{S}_1, s' \in \mathcal{S}_2 \text{ and } s'' \in \mathcal{S}_1. \end{aligned} \tag{7}$$

The replacement of π_2 with a $\min$ operator is caused by the observation that, since π_1 is fixed, the problem reduces to a single-agent MDP where an agent tries to minimize the recevied rewards.

The optimal Bellman equation (also implies Nash equilibrium) can be expressed as:

$$\begin{cases} v^*(s) = \max_a \sum_{s'} p(s'|s,a)(r(s,a,s') + \gamma v^*(s')), s \in \mathcal{S}_1 \text{ and } s' \in \mathcal{S}_2, \\ v^*(s) = \min_a \sum_{s'} p(s'|s,a)(r(s,a,s') + \gamma v^*(s')), s \in \mathcal{S}_2 \text{ and } s' \in \mathcal{S}_1, \end{cases} \tag{8}$$

assuming that states in $\mathcal{S}_1$ belongs to the max player, the other one is the min player. A *value iteration* algorithm according to this minimax recursion converges to optimal (Condon, 1992). However, because the *policy evaluation* in AMGs consists of two policies π_1, π_2, the policy iteration, which alternates between policy evaluation and policy improvement, could have the following four formats:

Algo.1 Fix π_1^t, compute the optimal counter policy π_2^t, then fix π_2^t compute the optimal counter policy π_1^{t+1}, alternating this procedure repeatedly,

Algo.2 Policy evaluation with π_1^t, π_2^t, swith both π_1^t and π_2^t to greedy policies with respect to current state-value function, continue this procedure repeatedly,

Algo.3 Policy evaluation with π_1^t, π_2^t, swith π_1^t to greedy policy with respect to the current state-value function and then compute the optimal counter policy for π_2^t, continue this procedure repeatedly,

Algo.4 Policy evaluation with π_1^t, π_2^t, swith π_2^t to greedy policy with respect to the current state-value function and then compute the optimal counter policy for π_1^t, continue this procedure repeatedly.

Intuitively speaking, all of those procedures are somewhat sensible, and perhaps would yield practical success. However, Condon (1990) showed that oscillation could be a problem that prevents **Algo.1** and **Algo.2** from converging, only **Algo.3** and **Algo.4** are correct, guaranteed to converge in general (Hoffman & Karp, 1966). **Algo.3** and **Algo.4** are duals. Variants of **Algo.3** or **Algo.4**, such as only switching one node to greedy every iteration then optimize the other player's strategy, are also correct, although they may have slower convergence rate (Condon, 1990).

4 ADVERSARIAL POLICY GRADIENT

Before presenting adversarial policy gradient for AMGs, we first review policy gradient in single agent MDP. In MDP, the strength of a policy π can be measured by (note that for brevity, we implicitly state that the policy is parameterized by θ):

$$J(\pi) = \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a} \pi(a|s) q_{\pi}(s,a), \tag{9}$$

where $d^{\pi}(s)$ is a state distribution under π. The gradient of $J(\pi)$ is:

$$\nabla J(\pi) = \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a} \pi(a|s) \nabla \log \pi(a|s) q_{\pi}(s,a) \tag{10}$$

The above formula is the Policy Gradient Theorem in MDP (Sutton et al., 2000), which implies that the gradient of the strength of a policy can be estimated by sampling according to π. The requirements are 1) π is differentiable, and 2) $q_{\pi}(s,a)$ can be estimated. This theorem can also be interpreted as a kind of generalized policy iteration, where the gradient ascent corresponds to policy improvement (Kakade, 2002), $q_{\pi}(s,a)$ is obtained by some policy evaluation. Depending on how $q_{\pi}(s,a)$ is estimated, policy gradient algorithms can be categorized into two families: Monte-Carlo policy gradients that use Monte-Carlo to estimate q_{π}, e.g., REINFORCE (Williams, 1992) and actor-critic (Sutton et al., 2000; Wang et al., 2017) methods that use another learning parameter to approximate the action-value under π. Recent approaches (Gu et al., 2017) interpolate both.

Analogously, in AMGs, for a pair of parameterized policies π_1 and π_2, the joint-strength of π_1 and π_2 may be defined as:

$$J(\pi_1, \pi_2) = \sum_{s} d^{\pi_1,\pi_2}(s) \sum_{a} \pi_1(a|s) q_{\pi_1}(s,a), \tag{11}$$

where $d^{\pi_1,\pi_2}(s)$ is the state-distribution given π_1 and π_2. A nature question is what is the gradient of $J(\pi_1, \pi_2)$ with respect to π_1 and π_2 respectively? One tempting derivation is to calculate the the gradient for both π_1 and π_2 simultaneously by treating the other policy as the "environment". Similar to the mutual greedy improvement in **Algo.2**, such a method tries to adapt based on the value function under current π_1, π_2, ignoring the fact that the opponent is a dynamic adversary who will also adapt its strategy for better counter-payoffs. Another possible algorithm is to fix π_1 and do a fix number of iterations to optimize π_2 by normal policy gradient as in MDP, and then fix π_2 for optimizing π_1, repeat such alternatively. However, this algorithm is an analog to **Algo.1**, which has shown to be non-convergent in general.

Following **Algo.3** and **Algo.4**, given the action-value function under π_1, π_2, a more reasonable approach for policy improvement is thus to switch π_2 to "greedy" (not necessary every node of π_2) and optimize the π_1 by policy gradient. Therefore, assuming π_1 is the max player, we advocate the following objectives

$$\begin{cases} J^{\pi_1}(\pi_1,\pi_2) = \sum_s d^{\pi_1,\pi_2}(s) \sum_a \pi_1(a|s) \sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma \min_{a'} q_{\pi_2}(s',a') \right] \\ J^{\pi_2}(\pi_1,\pi_2) = \sum_s d^{\pi_1,\pi_2}(s) \sum_a \pi_2(a|s) \sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma \max_{a'} q_{\pi_1}(s',a') \right]. \end{cases} \tag{12}$$

Consequently, the gradients can be estimated by

$$\begin{cases} \nabla J^{\pi_1}(\pi_1,\pi_2) = \mathbb{E}_{\pi_1,\pi_2} \left[\nabla \log \pi_1(a|s) \sum_{s'} p(s'|s,a) (r(s,a,s') + \gamma \min_{a'} q_{\pi_2}(s',a')) \right] \\ \nabla J^{\pi_2}(\pi_1,\pi_2) = \mathbb{E}_{\pi_1,\pi_2} \left[\nabla \log \pi_2(a|s) \sum_{s'} p(s'|s,a) (r(s,a,s') + \gamma \max_{a'} q_{\pi_1}(s',a')) \right]. \end{cases} \tag{13}$$

The above formulation implies that, when computing the gradient for one policy, the other policy is simultaneously switched to "greedy". This joint-change forces the current player to adjust the action preferences according to the worst-case response of the opponent, which is desirable due to the adversarial nature of the game.

4.1 ADVERSARIAL MONTE-CARLO POLICY GRADIENT

A straightforward implementation of Equation (12) is to do separate Monte-Carlo from each next-action, and then apply the $\min$ or $\max$ operator. However, this may not be practical feasible when the action space is large. We introduce a parameter k, by which only a subset of actions are considered in our adversarial Monte-Carlo policy gradient design.

We conduct minimal modification on self-play REINFORCE. The algorithms work as follows: after a batch of n games is generated by a self-play policy with game results, for each game, a single state-action pair (s, a) is uniformly sampled as a training example. Instead of directly using the observed return z in batch as the estimated action-value for (s, a), we perform extra Monte-Carlo simulations to estimate the *minimum* return for (s, a). In below are two methods we propose:

> **AMCPG-A:** Run k self-play games from the (s, a) using self-play policy π, take the minimum the k additional returns and z.
>
> **AMCGP-B:** Sample a s' from (s, a) according to state-action transition $p(\cdot|s, a)$, select top k predicated actions using self-policy π, doing separate self-play afterwards, take the minimum of the k additional returns and z.

Note that only "minimum" is used, because we assume the game result is with respect to the player to play at state s. It is easy to see that in **AMCPG-A**, if *mean* operator is used rather than *minimum*, a self-play REINFORCE variant is recovered. When $k = |\mathcal{A}(s')|$, **AMCPG-B** becomes a full number Monte-Carlo estimate, but in this situation, even though each action-value's estimation is unbiased, bias could still be incurred when applying the $\min$ or $\max$. This is due to the well-known "winner's curse" problem (Capen et al., 1971; Smith & Winkler, 2006). By introducing a parameter k, the above two methods can be seen as some kind of "soft min", which is usually better than applying a "hard min" on noisy estimates.

4.2 ADVERSARIAL ACTOR-CRITIC

As in MDPs, another possible direction is to implement Equation (12) in an adversarial actor-critic framework. Compared to the actor-critic in MDP, the difference is that a $\min$ operator will be used when estimating the "critic". In practice, for example, when neural net is used as a function approximator, the $\min$ operator might be easily achieved by employing an architecture with both policy and action-value outputs. More desirably, as in Q-learning (Fox et al., 2016), the biased caused by $\min$ operator can be addressed by certain soft updates.

5 RELATED WORK

Previous studies of games are divided among different disciplines, including game theory, reinforcement learning as well as computational complexity. Shapley (1953) introduced the notion of Stochastic Games, which is a multi-player framework that generalizes both Markov Decision Process (only one player) and repeated games (only one state). Condon (1990; 1992) initiated the study of *Simple Stochastic Games*, which are two-player games played in a directed graph with min, max and restricted probabilistic transition nodes. While her concern was largely in a computational complexity perspective, Condon (1990) showed that several variants of Hoffman-Karp's algorithms are incorrect. Littman (1996) formulated the notion of Alternating Markov Games, which is a more general *Simple Stochastic Games* without the restriction in action sets and probabilistic transitions. Littman (1994) also proposed a minimax-Q learning algorithm that is applicable to Alternating Markov Games as well as two-player zero-sum games played with matrix payoffs.

One of the most cited work that uses reinforcement learning to play AMGs is by Tesauro (1995), who trained multi-layer neural network to play backgammon. Tesauro's program takes advantage of the symmetric (due to alternating and zero-sum) property of AMGs and learn the optimal value function by "greedy" self-play, relying the stochastic environment for exploration. It was conjectured that the smoothness of the value function is one major factor to the particular success of TD-Gammon (Tesauro, 1995).

The recent advances of reinforcement learning are arguably primarily due to the use of deep neural networks (LeCun et al., 2015) as a much more expressive function approximator. Deep neural nets

have been applied to two-agent board game Go (Maddison et al., 2015; Tian & Zhu, 2016; Clark & Storkey, 2015), leading to super-human Go-playing system AlphaGo (Silver et al., 2016). AlphaGo consists of several independent components including supervised learning for move prediction, reinforcement learning and Monte-Carlo Tree Search (MCTS) (Coulom, 2006). AlphaGo successfully combined those strengths and achieved remarkable success. Newly updated AlphaGo Zero uses a "heavy" self-play based on MCTS to improve its parameterized neural net (Silver et al., 2017). A similar approach is Expert Iteration (Anthony et al., 2017). However, one drawback of such methods is that huge computation could be required when using MCTS self-play for data generation, i.e., when applied to Shogi and Chess, 5000 TPUs were used (Silver et al., 2017).

Policy gradient reinforcement learning, on the other hand, is a conventional fast RL method that can be used to refined a neural net policy by pure neural net self-play. Indeed, in AlphaGo (Silver et al., 2016), policy gradient RL is used to improve the neural network model obtained by supervised learning. The improved model is usually less useful in search, primary because its strong bias towards some local optima. However, due to its better playing strength and fast speed, a possible usage is using it to generate training data for a value net (Silver et al., 2016), which can be integrated into Monte-Carlo tree search. The policy gradient employed by AlphaGo is a variant of self-play REINFORCE that resembles to **Algo.2**. A practical innovation is to select an opponent parameter from a previous iteration, so as to increase the stability of the training. Policy gradient has also been applied to train a balanced policy (Silver & Tesauro, 2009); however, its requirement of an optimal value function makes it less useful in practice (Huang et al., 2010).

Adversarial methods have also been adopted in MDP (Pinto et al., 2017), with the observation that errors may occur in simulated models, thus maximizing a worst-case return will generally produce more robust results. The alternating procedure proposed by Pinto et al. (2017) resembles to **Algo.1**. The idea of adversarial learning is also used in generative models (Goodfellow et al., 2014), which leverages adversarial examples to train a more robust classifier.

6 EXPERIMENTS

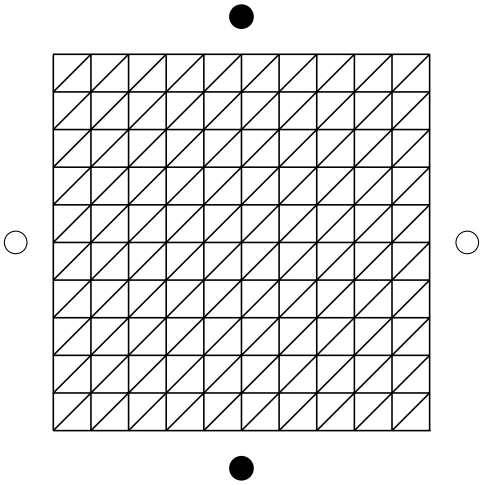
In this section, we present experimental results of the proposed adversarial Monte-Carlo policy gradient algorithms in Section 4. We first introduce the testbed game, and present experimental results subsequently.

6.1 GAME OF HEX

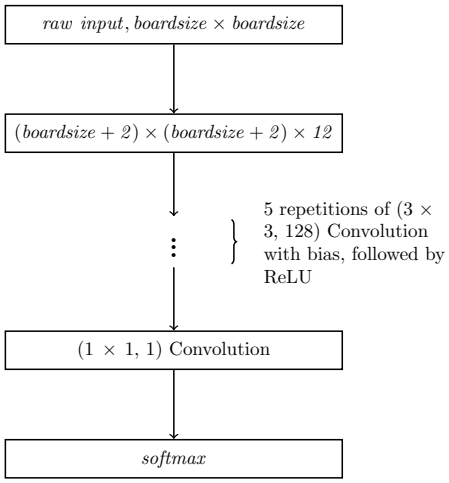
The game of Hex is played on a $N \times N$ rhombus board, where Black and White alternatively place a stone on an unoccupied hexagonal cell. The goal is to connect two sides of the board. Since its invention, the game of Hex has been an active research problem for both mathematicians (Nash, 1952) and computer scientists (Shannon, 1953). Nash (1952) proved by strategy stealing argument that from the empty board, the game theoretic value is a first-player win. However, explicit winning strategy is unclear, i.e., after an arbitrary opening move on the board, the theoretical result become unknown. Hex board can also be mapped into Go-style, in which stones are played at intersections, as shown in Figure 1 (left). Hex is a game collected in OpenAI gym; usually 11×11 is considered as a regular board size, 13×13 Hex is also played at computer Olympiad Hex tournaments.

Similar to Go of same board size, Hex is challenging to play primarily because of its large and near-uniform branching factor and the difficulty of constructing reliable evaluation function. Hex is simpler than Go in the sense that perfect play can often be achieved whenever virtual connections are found by H-Search (Anshelevich, 2002).

We consider using policy gradient reinforcement learning to play the game of Hex. Similar to the previous work in Go, we use deep convolutional neural networks as a function approximator. Figure 1(right) shows our architecture design. Instead of fixing the board size to be particular number, we design an architecture that is able to receive different board size inputs, which can be achieved by having multiple parallel inputs with different shape when building the computation graph using Tensorflow (Abadi et al., 2016). Specifically, the architecture allows board size $8 \leq N \leq 15$. After boarding padding, each raw board state is processed into 12 binary feature planes, which are respectively: Black stones, White Stones, empty points, to-play plane, black bridge endpoints, white bridge



(a) A regular Hex board in Go-style, stones are played at intersections. Black or White win by connecting a group of its stones to its two sides.



(b) Neural network architecture, it accepts different board size inputs, and padded with an extra border using Black or White stones.

Figure 1: Hex board (left) and neural network design (right)

endpoints, to-play save bridge points, to-play make-connection points, to-play form bridge points, opponent save bridge points, opponent form bridge points, opponent make-connection points.

6.2 Results

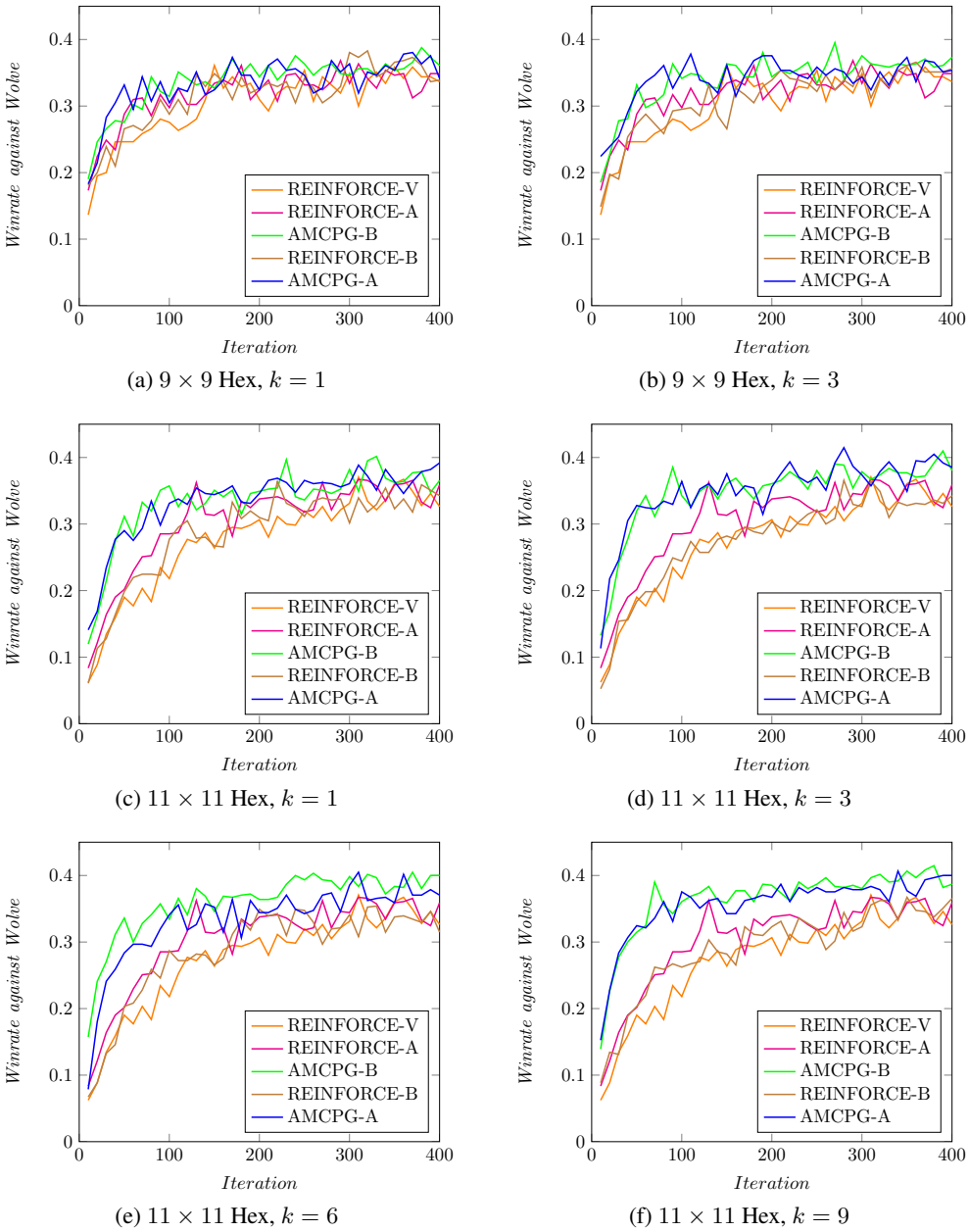
We apply policy gradient reinforcement learning to improve the resulting policy net trained from supervised learning. In addition to our proposed adversarial Monte-Carlo policy gradient algorithms, for comparison purpose, we implement three REINFORCE variants, as shown in below.

- REINFORCE-V: Vanilla REINFORCE using a parameterized self-play policy. After a batch of n self-played games, each game is then replayed to determine batch policy gradient update $\frac{\alpha}{n}\sum_i^n \sum_t^{T_i} \nabla \log \pi(s_t^i, a_t^i; \theta) z_t^i$, where z_t^i is either $+1$ or -1.
- REINFORCE-A: An "AlphaGo-like" REINFORCE. It differs from REINFORCE-V by randomly selecting a set of previously saved parameter weights from former iterations as the opponent for self-play.
- REINFORCE-B: A new implementation that for each self-played game, only one state-action pair is uniformly selected for policy gradient update. It differs from AMCPG-A by using the mean of all $k+1$ observed returns.

All methods are implemented using Tensorflow, sharing the same code base. In fact, they are only different in a few lines of code. A self-play policy is employed for all algorithms, which is equivalent as forcing π_1 and π_2 to share the same set of parameters. The game batch size n is set to 128, with opening move for each game played uniform randomly; learning rate is set to 0.001 after a grid search, vanilla stochastic gradient ascent is used as the optimizer. The reward signal is either $+1$ or -1, and only appears after a complete self-play game; no discounting is used. For all algorithms, the same neural net architecture is adopted, with initial parameter weights obtained from supervised learning on a dataset generated by MoHex (Henderson, 2010) self-play on board size 9×9. This supervised learning is only on 9×9 Hex, trained parameter weights are reused in other board sizes as the starting policy.

We run policy gradient reinforcement learning for 400 iterations for each method, on different board sizes (smaller board 9×9 and regular board 11×11) with varied k. We use open source player Wolve [1] (Henderson, 2010) as a benchmark to measure the relative performance of learned models.

[1] http://benzene.sourceforge.net/



(a) 9×9 Hex, $k = 1$ (b) 9×9 Hex, $k = 3$

(c) 11×11 Hex, $k = 1$ (d) 11×11 Hex, $k = 3$

(e) 11×11 Hex, $k = 6$ (f) 11×11 Hex, $k = 9$

Figure 2: Comparison of playing strength against Wolve on 9×9 and 11×11 Hex with different k, the curve represents the average winrate among 10 trails with Wolve as Black or White.

After every 10 iteration, model weights are saved and then evaluated by playing against 1-ply Wolve. The primary strength of Wolve comes from its inferior cell analysis augmented H-Search, making it is able to discover winning strategies for perfect play much earlier than unenhanced H-Search, along with more accurate position evaluation with electric resistance evaluation (Anshelevich, 2002; Henderson, 2010) if no winning virtual connection can be found. H-Search (Anshelevich, 2002) computes virtual connections by iteratively applying an AND-OR hierarchical algebra, this also makes it orders of magnitude slower than a pure neural net evaluation. The tournaments with Wolve are played by iterating all opening moves, each repeated 5 times with Wolve as Black or White.

Figure 2 compares the strength of these five different algorithms. It is interesting that REINFORCE-B is able to achieve similar performance with REINFORCE-V, even though the number of training samples used in the former algorithm is significantly less, this is perhaps because the reward signal in the same game is too much correlated. Consistent with the finding in Go (Silver et al., 2016), the method REINFORCE-A obtained small performance improvements on both 9×9 and 11×11 Hex. However, the newly developed algorithms AMCPG-A and AMCPG-B are able to learn better polices. They tend to learn faster, also generally achieve better results even when $k = 1$. The better performances are more eminent in regular board size 11×11. Those results confirm the benefit of estimating the *minimum*, rather than *mean* return, when applying policy gradient methods to Alternating Markov Games.

We note that policy gradient training on both 9×9 and 10×10 Hex is very fast. On an Intel i7-6700 CPU computer with single GTX 1080 GPU, most took only a few hours, the longest training by AMCPG-A, AMCPG-B and REINFORCE-B ($k = 9$, 11×11 Hex) took about 12 hours. For $k = 3$ case, AMCPG-A, AMCPG-B and REINFORCE-B are about twice slower than REINFORCE-V and REINFORCE-A.

6.2.1 MINIMUM RETURN IN MONTE-CARLO TREE SEARCH

In the previous section we have shown policy gradient can benefit from minimum Monte-Carlo return. State-of-the-art methods for Hex use Monte-Carlo tree search, which can be viewed as using the Monte-Carlo return as a "critic" to guide a tree represented policy. To further see the effect of worst "critic", we proceed to apply minimum rollout return to MoHex 2.0 (Huang et al., 2013), the world champion Monte-Carlo tree search Hex player since 2013. Mohex 2.0 is an improved version of MoHex 2011 (Arneson et al., 2010), with stronger virtual connections (Pawlewicz et al., 2015) and pattern based playout.

In MoHex 2.0, after an *in-tree* phase, a leaf node is selected, then one or more rollouts are conducted thereafter by a fast playout policy trained on 13×13 expert Hex games. MoHex 2.0 back-propagates the average rollout result to the tree. Following the same idea in the previous section, we modify MoHex 2.0 to use the minimum of the rollout results instead. To see the strength of minimum return, we experimentally compare three programs: MoHex 2.0, MoHex-A and MoHex-Mixed. MoHex-A is directly modified on MoHex 2.0's code, for all programs, at the leaf node, 2 playouts are conducted. The only difference between MoHex-A and MoHex 2.0 is that MoHex-A backs up the minimum playout results (w.r.t the player to play at the leaf) while MoHex 2.0 uses average playout results. Another program MoHex-Mixed uses the linear combination of the minimum and average return, with factors 0.5 and 0.5.

Table 1 summarizes the comparative results. It is easy to see that MoHex-A and MoHex-Mixed can still achieve slightly better results, due to their deliberate consideration of minimum "critics".

Table 1: Winrates against Mohex 2.0 of MoHex-A and MoHex-Mixed with same number of simulations. All programs use the default parameters the same as MoHex 2.0 except that *number_playouts_per_visit* is set to 2. On each board size, every opening was tried 10 times with each player as black or white. 1690 games were played for each match.

Opponent	#Simulations	MoHex2.0 was white	MoHex2.0 was black	Overall winrate	Average search time per game of MoHex 2.0 (s)	Average search time per game of Opponent (s)
MoHex-A	10^4	62.1%	43%	52.5%	42.0	49.0
MoHex-Mixed	10^4	62.8%	44.1%	53.5%	42.7	50.0

6.2.2 MoHex-CNN for multiple board size Hex

Unlike previous work in Go (Silver et al., 2017) or Hex (Anthony et al., 2017), the neural net we proposed has no board size dependent fully connected layers, this makes it possible to easily reuse the trained neural network on other board sizes other than 9×9. To see the generalization ability across different board sizes in search, we incorporate the trained neural net to MoHex 2.0, the resulting program is MoHex-CNN. The difference between MoHex-CNN and MoHex 2.0 is that MoHex-CNN replaces the prior probability of MoHex 2.0 with the policy neural net trained on 9×9 Hex games, i.e., the neural net model before policy gradient learning.

Table 2: Winrates against Mohex 2.0 of MoHex-CNN with same number of simulations. All programs use the default parameters the same as MoHex 2.0. Every opening was tried twice with each player as black or white. Column 6 and 7 are respectively the average consumed time per game for each corresponding program.

Opponent	#Simulations	MoHex2.0 was white	MoHex2.0 was black	Overall winrate	Average time of MoHex-2.0	Average time of MoHex-CNN	Board size
MoHex-CNN	10^4	77.8%	59.3%	68.5%	4.5	6.8	9×9
MoHex-CNN	10^4	83%	58.0%	70.5%	12.3	17.1	10×10
MoHex-CNN	10^4	75.2%	48.8%	62.0%	13.4	18.2	11×11
MoHex-CNN	10^4	78.5%	55.6%	67%	29.4	36.0	12×12
MoHex-CNN	10^4	69.8%	45.6%	57.7%	41.7	49.9	13×13

Table 2 contains the results of MoHex-CNN on board sizes from 9×9 to 13×13. We note that board sizes smaller than 9 are not investigated because they can easily be solved by virtual connection computation or tree based proof number search (Henderson, 2010). MoHex 2.0 was not designed to play board size > 13, so we stop at board size 13. From Table 2, it is easy to notice that MoHex-CNN is able to defeat MoHex 2.0 on every board size, which is remarkable since the neural net model was only trained on a set of 9×9 Hex games for only a few hours on a single GPU. The computation overhead is also very small, because in our implementation, neural net evaluation is computed in parallel with *prior pruning* [2] when expanding a node.

For 13×13 Hex, we have seen that adversarial Monte-Carlo return slightly improved MoHex 2.0, we further show the results in Table 3 by combining the minimum Monte-Carlo return with MoHex-CNN. Same as the notation in previous section, MoHex-CNN-A uses the minimum of two playout returns while MoHex-CNN-Mixed uses the linear combination of *minimum* and *average* by factors 0.5 and 0.5.

Table 3: Winrates against Mohex 2.0 of MoHex-CNN-A and MoHex-CNN-Mixed with same number of simulations. All programs use the default parameters the same as MoHex 2.0 except that at the leaf node *number_playouts_per_visit* is set to 2. On 13×13 Hex, every opening was tried twice times with each player as black or white.

Opponent	#Simulations	MoHex2.0 was white	MoHex2.0 was black	Overall winrate	Average search time per game of MoHex 2.0 (s)	Average search time per game of Opponent (s)
MoHex-CNN-A	10^4	68.6%	47.3%	58.0%	41.2	55.8
MoHex-CNN-Mixed	10^4	69.2%	56.2%	62.7%	40.5	57.3

Results in Table 3 indicate that using the mixed return achieve the best performance. This is perhaps because, different from policy gradient, the *in-tree* phase in MCTS is already an implicit form *minimax*. However, results in Table 3 suggest that a moderate skewing the critic to the worst-case still helped the search.

6.2.3 Comparison with ExIt

ExIt (Anthony et al., 2017) was competed against MoHex 2011 (Arneson et al., 2010) on 9×9 Hex. To have a grasp of the relative strengths of ExIt, MoHex 2.0 and MoHex-CNN, we also present

[2] Used by MoHex to prune proven inferior nodes whenever expanding a node

results of MoHex 2.0 and MoHex-CNN played against MoHex 2011 on 9×9 Hex [3]. Following the experiment in ExIt, we show the results with 10^4 and 10^5 simulations, in Table 4.

Table 4: Results of ExIt, MoHex 2.0 and MoHex-CNN against MoHex 2011 on 9×9 Hex. All MoHex programs use the same default parameters, with 10^4 simulations (left) or 10^5 simulations (right). Every opening was tried twice times with each player as black or white. ExIt's result is from (Anthony et al., 2017).

Opponent	MoHex 2011 was white	MoHex 2011 was black	Overall winrate	MoHex 2011 was white	MoHex 2011 was black	Overall winrate
MoHex 2.0	82.7%	66.7%	74.7%	79.0%	65.4%	72.2%
MoHex-CNN	85.2%	65.4%	75.3%	84.0%	71.6%	77.8%
ExIt	-	-	75.3%	-	-	59.3%

It is easy to see that when 10^4 simulations is used, both MoHex 2.0 and MoHex-CNN has similar winrates as ExIt. However, the 10^5 simulations MoHex 2.0 and MoHex-CNN obtain much better winrates than ExIt (Anthony et al., 2017). We further note that, the state-of-art for 9×9 Hex is that all openings have been solved (Pawlewicz & Hayward, 2013) [4]. This parallel solver has been embedded into MoHex 2.0, but in default setting, it is turned off.

7 Conclusions

We reviewed MDP and AMG, and proposed a new adversarial policy gradient paradigm that uses the worst-case return as the "critic". We introduced two practical adversarial Monte-Carlo policy gradient methods. We evaluated our algorithms on game of Hex, the experimental comparisons show that the new algorithms are notably better than routinely adopted self-play REINFORCE variants. We also applied *minimum* return to Monte-carlo tree search, and observed better performance on 13×13 Hex. Another contribution we made is a multi-boardsize neural net architecture, we demonstrated that a single neural net model trained on smaller board size can effectively generalize to larger board sizes. As a result, using it as a prior knowledge, we are able to improve MoHex 2.0 consistently from 9×9 to 13×13 Hex. Since in practice, expert data is often scarce, difficult to generate with limited computation resources, in this paper we provided a more feasible approach for high-performance playing for multi-boardsize Hex.

On the other hand, based on the adversarial policy gradient formulation in AMGs, the Monte-carlo policy gradients, either self-play REINFORCE-V or others, are not unbiased anymore, yet still have the drawback of being sample-inefficient, and may exhibit high variance. In this sense, an adversarial actor-critic framework is perhaps more appealing. In Monte-carlo tree search framework, value neural network can also be used to replace Monte-carlo rollouts. We leave this line of research as future work.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Vadim V Anshelevich. A hierarchical approach to computer hex. *Artificial Intelligence*, 134(1-2): 101–120, 2002.

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *arXiv preprint arXiv:1705.08439*, 2017.

Broderick Arneson, Ryan B Hayward, and Philip Henderson. Monte carlo tree search in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, 2010.

[3] In the literature, MoHex 2.0 has been demonstrated to be stronger than MoHex 2011, but playing results on 9×9 board size were never presented.

[4] The solver shows that among all 81 openings on 9×9 board, 53 are first player win.

Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, pp. 679–684, 1957.

Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1st edition, 1996. ISBN 1886529108.

Edward C Capen, Robert V Clapp, William M Campbell, et al. Competitive bidding in high-risk situations. *Journal of petroleum technology*, 23(06):641–653, 1971.

Christopher Clark and Amos Storkey. Training deep convolutional neural networks to play go. In *International Conference on Machine Learning*, pp. 1766–1774, 2015.

Anne Condon. On algorithms for simple stochastic games. In *Advances in computational complexity theory*, pp. 51–72, 1990.

Anne Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pp. 72–83. Springer, 2006.

Roy Fox, Ari Pakman, and Naftali Tishby. Taming the noise in reinforcement learning via soft updates. In *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence*, UAI'16, pp. 202–211, Arlington, Virginia, United States, 2016. AUAI Press.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pp. 2672–2680, 2014.

Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E Turner, and Sergey Levine. Q-prop: Sample-efficient policy gradient with an off-policy critic. In *ICLR*, 2017.

Philip Henderson. *Playing and solving the game of Hex*. PhD thesis, University of Alberta, 2010.

Alan J Hoffman and Richard M Karp. On nonterminating stochastic games. *Management Science*, 12(5):359–370, 1966.

Ronald A Howard. Dynamic programming and markov processes. 1960.

Shih-Chieh Huang, Rémi Coulom, Shun-Shii Lin, et al. Monte-carlo simulation balancing in practice. *Computers and Games*, 6515:81–92, 2010.

Shih-Chieh Huang, Broderick Arneson, Ryan B Hayward, Martin Müller, and Jakub Pawlewicz. Mohex 2.0: a pattern-based mcts hex player. In *International Conference on Computers and Games*, pp. 60–71. Springer, 2013.

Sham M Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pp. 1531–1538, 2002.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *ICLR*, 2016.

Long-H Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3/4):69–97, 1992.

Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the ehoffman1966nonterminatingleventh international conference on machine learning*, volume 157, pp. 157–163, 1994.

Michael Lederman Littman. *Algorithms for sequential decision making*. PhD thesis, Brown University Providence, RI, 1996.

Chris J Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. In *ICRL*, 2015.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pp. 1928–1937, 2016.

John F Nash. Some games and machines for playing them. 1952.

Jakub Pawlewicz and Ryan B Hayward. Scalable parallel dfpn search. In *International Conference on Computers and Games*, pp. 138–150. Springer, 2013.

Jakub Pawlewicz, Ryan Hayward, Philip Henderson, and Broderick Arneson. Stronger virtual connections in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(2):156–166, 2015.

Jan Peters and J Andrew Bagnell. Policy gradient methods. In *Encyclopedia of Machine Learning*, pp. 774–776. Springer, 2011.

Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. In *ICML*, 2017.

Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering, 1994.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. 2016.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 1889–1897, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Claude E Shannon. Computers and automata. *Proceedings of the IRE*, 41(10):1234–1241, 1953.

Lloyd S Shapley. Stochastic games. *Proceedings of the national academy of sciences*, 39(10): 1095–1100, 1953.

David Silver and Gerald Tesauro. Monte-carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 945–952. ACM, 2009.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 387–395, 2014.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

James E Smith and Robert L Winkler. The optimizers curse: Skepticism and postdecision surprise in decision analysis. *Management Science*, 52(3):311–322, 2006.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Preliminary Draft, MIT press Cambridge, second edition, 2017.

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pp. 1057–1063, 2000.

Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3): 58–68, 1995.

Yuandong Tian and Yan Zhu. Better computer go player with neural network and long-term prediction. In *ICLR*, 2016.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. In *ICML*, 2016.

Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. In *ICLR*, 2017.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

APPENDIX

A PSEUDOCODE

We present the pseudocode of **AMCPG-A** and **AMCPG-B**.

Algorithm 1: Adversarial Monte-Carlo Policy Gradient (AMCPG-A and AMCPG-B)

Initialize θ ;
$ite \leftarrow 0$;
while $ite < maxIterations$ **do**
  Self-play a batch of n games E using π ;
  for $e_i \in E$ **do**
    Select a state-action pair (s_j, a_j) uniform randomly;
    Let $z(s_j, a_j)$ be the outcome with respect to action a_j at s_j in e_i;
    From (s_j, a_j),

$$z'(s_j, a_j) = \begin{cases} A: & \text{Self-play } k \text{ games using } \pi\text{, record the minimum outcome with respect to } s_j; \\ B: & \text{Select top } k \text{ moves from } s'_j\text{, from each move self-play a game using } \pi, \\ & \text{record the minimum outcome w.r.t } s_j; \end{cases}$$

    $R^i \leftarrow \min\{z(s_j, a_j), z'(s_j, a_j)\}$;
    Write (s_j, a_j, R^i) to the batch ;
  $\theta \leftarrow \theta + \frac{\alpha}{n} \sum_{i=1}^{n} \nabla \log \pi(s^i_j, a^i_j; \theta) R^i$;
  $ite \leftarrow ite + 1$;

We also present the pseudo-code of **Algo.1**, **Algo.2** and **Algo.3** in Section 3.

B EXPERIMENTS

B.1 SUPERVISED LEARNING FOR INITIALIZATION

Since high quality data is not instantly available, to initialize the neural net weight, we first generated a dataset containing 10^6 state-action pairs by computer player MoHex played against MoHex, on 9×9 board. We train the policy neural net to maximize the log-likelihood on this dataset. The training only took $100,000$ steps with mini-batch 64, optimized using Adam (Kingma & Ba, 2015) with learning rate 0.001.

The resulting neural net could be used for multiple board sizes, its win-percentages against 1-play Wolve on 9×9 and regular board size 11×11 are respectively 13.2% and 4.6% (iterated all opening moves, 10 trails each opening with Wolve as Black or White).

Input feature has 12 binary planes, utilizing a "bridge pattern" in Hex.

B.2 MOHEX 2.0'S IN-TREE PHASE

Starting from the root node, MoHex 2.0 selects a child node according to the score function in below:

$$score(s, a) = (1 - w)\left(q(s, a) + c_b \sqrt{\frac{\log N(s)}{N(s, a)}}\right) + wR(s, a) + c_{pb} \frac{\rho(s, a)}{\sqrt{N(s, a) + 1}},$$

where $N(s), N(s, a)$ are respectively the visit counts of s and (s, a), $q(s, a)$ is the Q-value of (s, a), $R(s, a)$ is the RAVE value, $\rho(s, a)$ is the prior probability from move pattern weights, w, c_b, c_{pb} are weighting parameters. MoHex-CNN uses the same formula except that the prior probability is from neural net.

Algorithm 2: Policy Iteration algorithms for AMGs

```
Procedure Algo.1():
    π_1, π_2 are policies for player 1 and player 2;
    Initialize π_1^0, π_2^0 arbitrarily;
    t ← 1;
    π_1^t ← π_1^0;
    stable ← false;
    while not stable do
        Fix π_1^t;
        Compute the optimal counter strategy for player 2 given player 1's stragy is fixed;
        Let π_2^t be the computed optimal counter strategy;
        Fix π_2^t;
        Compute the optimal counter strategy for player 1 given player 2's strateg is fixed;
        Let π_1^{t+1} be the computed optimal counter strategy;
        t ← t + 1;
        if π_1^{t-1} = π_1^t and π_2^{t-2} = π_2^{t-1} then
            stable ← true;
Procedure Algo.2():
    π_1, π_2 are policies for player 1 and player 2;
    Suppose player 1 is the max player, palyer 2 is the min player;
    Initialize π_1^0, π_2^0 arbitrarily;
    t ← 0;
    stable ← false;
    while not stable do
        Fix π_1^t, π_2^t;
        Compute the value function V_t under π_1^t, π_2^t by policy evaluation;
        Let π_1^{t+1} be the greedily (max) modified policy w.r.t V_t;
        Let π_2^{t+1} be the greedily (min) modified policy w.r.t V_t;
        t ← t + 1;
        if π_1^t = π_1^{t-1} and π_2^t = π_2^{t-1} then
            stable ← true;
Procedure Algo.3():
    π_1, π_2 are policies for player 1 and player 2;
    Suppose player 1 is the max player, palyer 2 is the min player;
    Initialize π_1^0, π_2^0 arbitrarily;
    t ← 0;
    stable ← false;
    while not stable do
        Fix π_1^t, π_2^t;
        Compute the value function V_t under π_1^t, π_2^t by policy evaluation;
        Let π_1^{t+1} be the greedily (max) modified policy w.r.t V_t;
        Fix π_1^{t+1};
        Compute the optimal counter policy for player 2 given player 1's policy is fixed;
        Let π_2^{t+1} be this optimal counter policy;
        t ← t + 1;
        if π_1^t = π_1^{t-1} and π_2^t = π_2^{t-1} then
            stable ← true;
```

Plane index	Description	Plane index	Description
0	Black played stones	6	To play save bridge points
1	White played stones	7	To play make-connection points
2	Unoccupied points	8	To play form bridge points
3	Black or White to play	9	Opponent's save bridge points
4	Black bridge endpoints	10	Opponent's form bridge points
5	White bridge endpoints	11	Opponent's make-connection points

B.3 WITH DFPN SOLVER TURNED ON

In computer Hex Olympiad tournament, MoHex is usually played with the parallel solver turned on. To demonstrate the full strength of MoHex-CNN compared to MoHex 2.0, we perform another experiments by limiting both programs with the same thinking time 5s per move, all with the DFPN solver turned on.

Table 5: Winrates against Mohex 2.0 of MoHex-CNN with the same time per move. All programs use the default parameters the same as MoHex 2.0 with parallel solver on. Every opening was tried twice with each player as black or white.

Opponent	Time per move (s)	MoHex2.0 was white	MoHex2.0 was black	Overall winrate	Average length of played games	Board size
MoHex-CNN	5	66.7%	51.9%	59.3%	39.5	9×9
MoHex-CNN	5	64.0%	56.0%	60.0%	48.2	10×10
MoHex-CNN	5	68.6%	56.2%	62.4%	58.1	11×11
MoHex-CNN	5	64.0%	52.8%	59.4%	69.7	12×12
MoHex-CNN	5	57.4%	55.6%	56.5%	80.6	13×13

As expected, the results show there on smaller board size like 9×9, due to the DFPN solver in both players, there is a significant decrease of MoHex-CNN's winrate. However, on larger board sizes, the influence of DFPN solver is small. We note that, in (Anthony et al., 2017), on 9×9 board size, ExIt's winrate against 4s-MoHex2011 with DFPN solver is 55.6%