Adversarial Policy Training against Deep Reinforcement Learning

Xian Wu[1,*], *Wenbo Guo*[1,*], *Hua Wei*[1,*], *Xinyu Xing*[1]
[1]*The Pennsylvania State University*
{xkw5132, wzg13, hzw77, xxing}@ist.psu.edu

Abstract

Reinforcement learning is a set of goal-oriented learning algorithms, through which an agent could learn to behave in an environment, by performing certain actions and observing the reward which it gets from those actions. Integrated with deep neural networks, it becomes deep reinforcement learning, a new paradigm of learning methods. Recently, deep reinforcement learning demonstrates great potential in many applications such as playing video games, mastering GO competition, and even performing autonomous pilot. However, coming together with these great successes is adversarial attacks, in which an adversary could force a well-trained agent to behave abnormally by tampering the input to the agent's policy network or training an adversarial agent to exploit the weakness of the victim.

In this work, we show existing adversarial attacks against reinforcement learning either work in an impractical setting or perform less effectively when being launched in a two-agent competitive game. Motivated by this, we propose a new method to train adversarial agents. Technically speaking, our approach extends the Proximal Policy Optimization (PPO) algorithm and then utilizes an explainable AI technique to guide an attacker to train an adversarial agent. In comparison with the adversarial agent trained by the state-of-the-art technique, we show that our adversarial agent exhibits a much stronger capability in exploiting the weakness of victim agents. Besides, we demonstrate that our adversarial attack introduces less variation in the training process and exhibits less sensitivity to the selection of initial states.

1 Introduction

With the recent breakthroughs of deep neural networks (DNN) in problems like computer vision, machine translation, and time series prediction, we have witnessed a great advance in the area of reinforcement learning (RL). By integrating deep neural networks into reinforcement learning algorithms, the machine learning community designs various deep reinforcement learning algorithms [29,43,53] and demonstrates their great success in a variety of applications, ranging from defeating world champions of Go [45] to mastering a wide variety of Atari games [30].

Different from conventional deep learning, deep reinforcement learning (DRL) refers to goal-oriented algorithms, through which one could train an agent to learn how to attain a complex objective or, in other words, maximize the reward it can collect over many steps (actions). Like a dog incentivized by petting and intimidation, reinforcement learning algorithms penalize the agent when it takes the wrong action and reward when the agent takes the right ones.

In light of the promising results in many reinforcement learning tasks, researchers recently devoted their energies to investigating the security risk of reinforcement learning algorithms. For example, early research has proposed various methods to manipulate the environment that an agent interacts with (*e.g.,* [4,18,21]). Their rationale behind such a kind of attack is as follows. In a reinforcement learning task, an agent usually takes as input the observation of the environment. By manipulating the environment, an attacker could influence the agent observation as well as its decision (action), and thus mislead the agent to behave abnormally (*e.g.,* subtly changing some pixel values of the sky in the Super Mario game, or injecting noise into the background canvas of the Pong game).

In many recent research works, attacks through environment manipulation have demonstrated great success in failing a well-trained agent to complete a certain task (*e.g.,* [18,19]). However, such attacks are not practical in the real world. For example, in the application of online video games, the input of a pre-trained master agent is the snapshot of the current game scenes. From the attackers' perspective, it is difficult for them to hack into the game server, obtain the permission of manipulating the environment, influence arbitrary pixels in that input image, and thus launch an adversarial attack as they expect. As a result, recent research proposes a new method to attack a well-trained agent [10].

Different from attacks through environment manipulation,

[*]Equal Contribution.

the new attack is designed specifically for the two-agent competitive game – where two participant agents compete with each other – and the goal of this attack is to fail one well-trained agent in the game by manipulating the behaviors of the other. In comparison with the environment manipulation methods, the new attack against RL is more practical because, to trigger the weakness of the victim agent, this attack does not assume full control over the environment nor that over the observation of the victim agent. Rather, it assumes only the free access of the adversarial agent (*i.e.,* the agent that the attacker trains to compete with his opponent's agent).

In [10], researchers have already shown that the method of attacking through an adversarial agent could be used for an alternative, practical approach to attack a well-trained agent in reinforcement learning tasks. However, as we will demonstrate in Section 6, this newly proposed attack usually exhibits a relatively low success rate of failing the opponent (or in other words victim) agent.[1] This is because the attack is a simple application of the state-of-the-art Proximal Policy Optimization (PPO) algorithm [43] and, by design, the PPO algorithm does not train an agent for exploiting the weakness of the opponent agent.

Inspired by this discovery, we propose a new technique to train an adversarial agent and thus exploit the weakness of the opponent (victim) agent. First, we arm the adversarial agent with the ability to observe the attention of the victim agent while it plays with our adversarial agent. By using this attention, the adversarial agent can easily figure out at which time step the opponent agent pays more attention to the adversary. Second, under the guidance of the victim's attention, the adversary subtly varies its actions. With this practice, as we will show and elaborate in Section 4 and 5, the adversarial agent could trick a well-trained opponent agent into taking sub-optimal actions and thus influence the corresponding reward that the opponent is supposed to receive.

Technically speaking, to develop the attack method mentioned above, we first approximate the policy network as well as the state-transition model of the opponent agent. Using the approximated network and model, we can determine the attention of the opponent agent by using an explainable AI technique. Besides, we can predict the action of the opponent agent when our adversarial agent takes a specific action.

With the predicted action in hand, our attack method then extends the PPO algorithm by introducing a weighted term into its objective function. As we will specify in Section 5, the newly introduced term measures the action deviation of the opponent agent with and without the influence of our adversarial agent. The weight is the output of the explainable AI technique, which indicates by how much the opponent agent pays its attention to the adversarial agent. By maximizing the weighted deviation together with the advantage function in the objective function of PPO, we can train an adversarial agent to take the action that could influence the action of the opponent agent the most.

In this paper, we do not claim that our proposed technique is the first method for attacking reinforcement learning. However, we argue that this is the first work that can effectively exploit the weakness of victim agents without the manipulation of the environment. Using MuJoCo [50] and roboschool Pong [33] games, we show that our method has a stronger capability of attacking a victim agent than the state-of-the-art method [10] (an average of 60% vs. 50% winning rate for MuJoCo game and 100% vs. 90% for the Pong game). In addition, we demonstrate that, in comparison with the state-of-the-art method of training an adversarial policy [10], our proposed method could construct an adversarial agent with a 50% winning rate in fewer training cycles (11 million vs. 20 million iterations for MuJoCo game, and 1.0 million vs. 1.3 million iterations for Pong game). Last but not least, we also show that using our proposed method to train an adversarial agent, it usually introduces fewer variations in the training process. We argue this is a very beneficial characteristic because this could make our algorithm less sensitive to the selection of initial states. We released the game environment, victim agents, source code, and our adversarial agents. [2]

In summary, the paper makes the following contributions.

- We design a new practical attack mechanism that trains an adversarial agent to exploit the weakness of the opponent in an effective and efficient fashion.
- We demonstrate that an explainable AI technique can be used to facilitate the search of the adversarial policy network and thus the construction of the corresponding adversarial agents.
- We evaluate our proposed attack by using representative simulated robotics games – MuJoCo and roboschool Pong – and compare our evaluation results with that obtained from the state-of-the-art attack mechanism [10].

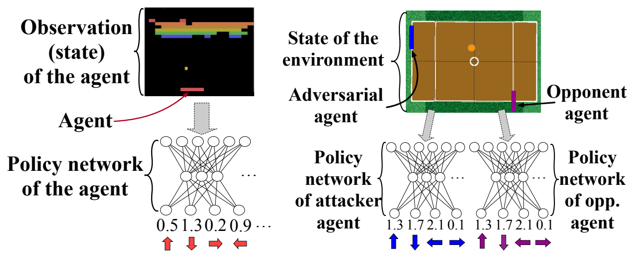
The rest of this paper is organized as follows. Section 2 describes the problem scope and assumption of this research. Section 3 describes the background of deep reinforcement learning. Section 4 and 5 specifies how we design our attack mechanism to train adversarial agents. Section 6 summarizes the evaluation results of our proposed attack mechanism. Section 7 provides the discussion of related work, followed by the discussion of some related issues and future work in Section 8. Finally, we conclude the work in Section 9.

2 Problem Statement and Assumption

Problem statement. Reinforcement learning refers to a set of algorithms that address the sequential decision-making

[1]Note that the paper uses "victim agent" and "opponent agent" interchangeably.

[2]https://github.com/psuwuxian/rl_attack


Observation (state) of the agent
Agent
Policy network of the agent
0.5 1.3 0.2 0.9 ...
State of the environment
Adversarial agent
Opponent agent
Policy network of attacker agent
Policy network of opp. agent
1.3 1.7 2.1 0.1
1.3 1.7 2.1 0.1

(a) Single agent game.   (b) Two-agent game.

Figure 1: The illustration of reinforcement learning tasks.

problem in complex scenarios. As is depicted in Figure 1, a game is formalized as an RL learning task, in which an agent observes and interacts with the game environment through a series of actions. In this process of interaction, the agent collects the reward for each of the actions it takes. Using the reward as a feedback signal, the agent could be aware of how well it performs at each time step.

The goal of RL is to learn an optimal policy, which guides the agent to take actions more effective and thus to maximize the amount of the reward it could gather from the environment. In the setting of deep reinforcement learning, as is shown in Figure 1, the policy learned is typically a deep neural network, which takes as the input the observation of the environment (*i.e.,* the current snapshot of the game) and outputs the actions that the agent would take (*i.e.,* left/right and up/down movements, etc.). In Section 3, we will describe more details about how to model a reinforcement learning problem and thus resolve an optimal policy for the agent involved.

As is demonstrated in Figure 1a, the game is formalized as a reinforcement learning problem in which the environment involves only a single agent. However, in many reinforcement learning tasks, an environment could contain two agents competing with each other while interacting with the environment (see Figure 1b). Recently, such two-agent competitive games driven by reinforcement learning have received great attention and reinforcement learning algorithms have demonstrated a great potential [32, 45]. In this work, we, therefore, focus our problem in the two-agent competitive environment, developing practical methods to train an adversarial policy for one agent to beat the other and win corresponding two-agent games. To be more specific, in this work, we fix one agent and train the other with the goal of having the trained agent build up the ability to exploit the weakness of that fixed other.

Assumption. It should be noted that, in our problem, we do not assume the victim agent adapts its policy based on its opponent immediately. With this assumption, we simulate a real-world scenario, where a game developer deploys an online game with an offline-trained master agent controlling its play with participants (*e.g.,* playing a two-party Texas hold 'em or a GO game), and the goal of an adversary is to figure out a way to defeat that master agent, rule the game, and thus gather maximum rewards for fun or for profits. When playing the game, the game developer could collect the game episodes and retrain the master agent accordingly. However, he cannot pull out the master agent and carry out retraining immediately (or in other words right after each round of its play). On the one hand, this is due to the fact that, training a game agent with an RL algorithm generally requires a long period of episode accumulation to receive a high winning rate (*e.g.,* the task of training the OpenAI's hide-and-seek game agent accumulates hundreds of millions of episodes [35]). On the other hand, this is because, even if the game developer retrains the master agent based on a large amount of game episodes that he collects, he still needs to figure out a way to preserve the generalizability of its master agent. As we will demonstrate in Section 6, after retraining the master agent using the episodes the master agent gathers when interacting with the adversarial agent, the master agent could capture the capability of defeating the adversary. However, it loses its ability to defeat ordinary game agents.

It should also be noted that this work is very different from many existing works, which assume an attacker has the privilege to manipulate the environment freely or, in other words, change the pixels in the snapshot that the victim agent observes (*e.g.,* [18, 40]). We believe the removal of this assumption is crucial and could make an adversarial attack more practical. To illustrate this argument, we again take for example the aforementioned online games. In these examples, the game environment refers to the game scenes created by the game engine and the agents in the game. The activities of directly manipulating the environment (game scenes) mean that an adversary breaks into the game server or engine, alters the game code related to the game scenes, and thus influences the environment that the agents interacts with. Technically, this inevitably introduces the efforts of the successful identification and exploitation of a software vulnerability on the game server. In practice, having such a capability typically implies tens of thousands of hours of effort from professional hackers, and cannot always guarantee the return of their efforts because of the defense mechanisms enabled in computer systems. With the removal of the assumption commonly made in previous works, we make the adversarial attack more cost-efficient because, instead of putting efforts on breaking into game server without the guarantee of success, an attacker only needs to train an adversarial policy to control his own agent and thus influences its opponent.

As is illustrated in Figure 1b, similar to the single-agent game driven by reinforcement learning, in the setting of a two-agent game, both of the agents take as input the observation of the same environment, and then output the actions through their own policy networks. In this work, when designing methods to train an adversarial agent, we do not assume that an attacker has access to the opponent agent's policy network nor its state transition model. Rather, we assume that the attacker knows the observation of the opponent agent as well as the action that the opponent takes. We believe this assumption is reasonable and practical because, as we mentioned above, both the attacker's agent and the opponent agent take the observation from the same environment, and the action took by agents can be easily observed from the environment as well. For example, the opponent agent's policy network outputs an upward movement, which the adversarial agent could easily observe from the change of the environment.

3 Background of Reinforcement Learning

Recently, many reinforcement learning algorithms have been proposed to train an agent interacting with an environment, ranging from Q-learning based algorithms (*e.g.,* [31,53]) to policy optimization algorithms (*e.g.,* [22,29,41,43]). Among all the learning algorithms, proximal policy optimization (PPO) [43] is the one that has been broadly adopted in the two-agent competitive games. For example, teams from OpenAI utilize this algorithm to play Hide-and-Seek [35] and world-famous game Dota2 [32]. In this work, we design our method of training an adversarial policy by extending the PPO learning algorithm. In this section, we briefly describe how to model a reinforcement learning problem, and then discuss how the PPO algorithm is designed to resolve the reinforcement learning problem.

3.1 Modeling an RL Problem

Given a reinforcement learning problem, it is common to model the problem as a Markov Decision Process (MDP) which contains the following components:

- a finite set of states $\mathcal{S}$, where each state $s^{(t)}$ ($s^{(t)} \in \mathcal{S}$) represents the state of the agent at the time t and $s^{(0)}$ is the initial state;
- a finite action set $\mathcal{A}$, where each action $a^{(t)}$ ($a^{(t)} \in \mathcal{A}$) refers to the action of the agent at the time t;
- a state transition model $\mathcal{P}$: $\mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$, where $P^a_{ss'} = \mathbb{P}[s^{(t+1)} = s' | s^{(t)} = s, a^{(t)} = a]$ denotes the probability that the agent transits from state s to s' by taking action a;
- a reward function $\mathcal{R}$: $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, where $R^a_s = \mathbb{E}[r^{(t+1)} | s^{(t)} = s, a^{(t)} = a]$ represents the expected reward if the agent takes action a at state $s^{(t)}$; here $r^{(t+1)}$ indicates the reward that the agent will receive at the time $t+1$ after taking the action;
- a scalar discount factor $\gamma \in [0,1]$, which is usually multiplied by future rewards as discovered by the agent in order to dampen the effect of rewards upon the agent's choice of an action.

As is mentioned above, the ultimate goal of reinforcement learning is to train the agent to find a policy $\pi(a|s)$: ($\mathcal{S} \rightarrow \mathcal{A}$) that could maximize the expectation of the total rewards over a sequence of actions generated through the policy. Mathematically, this could be accomplished by maximizing *state-value function* $V_\pi(s)$ defined as

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s)(R^a_s + \gamma \sum_{s' \in \mathcal{S}} P^a_{ss'} V_\pi(s')) \,, \tag{1}$$

or the *action-value function* $Q_\pi(s,a)$ defined as

$$Q_\pi(s,a) = R^a_s + \gamma \sum_{s' \in \mathcal{S}} P^a_{ss'} \sum_{a' \in \mathcal{A}} \pi(a'|s') Q_\pi(s',a') \,. \tag{2}$$

In reinforcement learning, the state-value function $V_\pi(s)$ represents how good is a state for an agent to be in. It is equal to the expected total reward for an agent starting from state s. The value of this function depends on the policy π, by which the agent picks actions to perform. Slightly different from $V_\pi(s)$, the action-value function $Q_\pi(s,a)$ is an indication for how good it is for an agent to pick action a while being in state s. By maximizing either of these functions above, one could obtain an optimal policy π_* for the agent to collect the maximum amount of rewards from the environment.

3.2 Resolving an RL problem

Deep Q-learning. To find an optimal policy for an agent to maximize its total reward, one method is to utilize deep Q-learning, which takes a state s and approximates the Q-value for each action based on that state (*i.e.,* $Q_\pi(s,a)$). With this approximation, although the agent cannot extract the policy explicitly, it could still maximize its reward by taking the action with the highest Q-value. As is shown in recent research, such a method demonstrates a great success in many applications, such as playing GO [45] and mastering a wide variety of Atari games [30]. However, since deep Q-learning usually calculates all possible actions in a discrete action space, it has been barely adopted to two-agent games with continuous action space, including simulation games, like MuJoCo and RoboSchool, and real-world strategy games, such as StarCraft and Dota. As a result, the policy gradient approach is typically adopted.

Policy Gradient Algorithm. Policy gradient refers to the techniques that directly parametrize the policy as a function $\pi_\theta(s,a) = \mathbb{P}(a|s,\theta)$. At the time t, this function takes as input the state $s^{(t)}$ and outputs the action $a^{(t)}$. In recent research article [29], researchers modeled the policy π as a deep neural network (*e.g.,* multilayer perceptron [55] or recurrent neural networks [58]), and named the DNN as the policy network.

To learn a policy network for an agent, the policy gradient algorithm defines an objective function $J(\theta) = \mathbb{E}_{s^{(0)},a^{(0)},\ldots\sim\pi_\theta}[\sum_{t=0}^{\infty} \gamma^t r^{(t)}]$ which represents the expectation of the total discounted rewards. By maximizing this objective function, one could obtain the parameters θ and thus the optimal policy. In order to compute parameters θ, the policy gradient algorithm computes the gradient of the objective

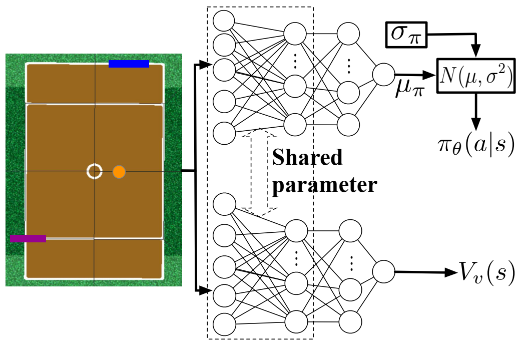


Figure 2: The neural network architecture involved in the PPO algorithm. Note that the two networks share parameters with each other.

function with respect to parameters (*i.e.,* $\nabla_\theta J(\theta)$) and then iteratively apply stochastic gradient-ascend to reach a local maximum in $J(\theta)$. According to the Policy Gradient Theorem [22], for any differentiable policy $\pi_\theta(s,a)$, the policy gradient can be written as

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_{\pi_\theta}(s,a)], \tag{3}$$

where $\pi_\theta(s,a)$ is the policy network and $Q_{\pi_\theta}(s,a)$ denotes the action-value function of the corresponding MDP. As we can easily observe from the equation above, to solve this equation, we need to know function $Q_{\pi_\theta}(s,a)$. In the policy gradient algorithm, the action-value function $Q_{\pi_\theta}(s,a)$ is approximated by a deep neural network $Q_w(s,a)$, which can be learned together with the policy network. However, this design has a limitation. In each iteration of the training process, an agent has to compute the reward at the end of the episode, and then average all actions. Therefore, an agent inevitably concludes all the actions taken were good, if it receives a high reward, even if some were really bad. To address this problem, one straightforward approach is to enlarge the training sample batch. Unfortunately, this could incur slow learning and the agent has to take even longer time to converge.

Actor-Critic Framework. To improve the policy gradient algorithm mentioned above, recent research introduces an actor-critic framework, which defines a critic and an actor. Through an action-value function $Q_{\pi_\theta}(s,a)$, the critic measures how good the action taken is. Through a policy network π_θ, the actor controls how the agent behaves. With both of these, we can rewrite the policy gradient as

$$\begin{aligned} &\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) A_{\pi_\theta}(s)], \\ &A_{\pi_\theta}(s,a) = Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s). \end{aligned} \tag{4}$$

Here, $A_{\pi_\theta}(s,a)$ is an advantage function, which measures the difference between the Q value for action a in state s and the average value of that state [12]. Through this advantage function, we can know the improvement over the average the action taken at that state. In other words, this function calculates the extra reward the agent gets if it takes this action.

To solve equation (4), the actor-critic framework approximates $V_{\pi_\theta}(s)$ through a deep neural network $V_v(s)$ parameterized by v and then utilizes this approximated $V_{\pi_\theta}(s)$ to deduce $Q_{\pi_\theta}(s,a)$. As is specified in [29], this neural network can be learned together with the policy network π_θ through either Monte-Carlo methods or Temporal-Difference methods [42].

Proximal Policy Optimization (PPO) Algorithm. Using the actor-critic framework to train an agent, recent research indicates that the actor usually experiences enormous variability in the training which influences the performance of the trained agent [41]. To stabilize actor training, recent research proposes the PPO algorithm [16, 43], which introduces a new objective function called "Clipped surrogate objective function". With this new objective function, the policy change could be restricted in a small range.

As is discussed in [41], the original mathematical form of clipped surrogate objective function is

$$\begin{aligned} &\text{maximize}_\theta\ \mathbb{E}_{(a^{(t)},s^{(t)})\sim\pi_{\theta_{old}}}\left[\frac{\pi_\theta(a^{(t)}|s^{(t)})}{\pi_{\theta_{old}}(a^{(t)}|s^{(t)})} A_{\pi_{\theta_{old}}}(a^{(t)},s^{(t)})\right], \\ &s.t.\ \mathbb{E}_{s^{(t)}\sim\pi_{\theta_{old}}}[D_{\mathrm{KL}}(\pi_{\theta_{old}}(\cdot|s^{(t)})||\pi_\theta(\cdot|s^{(t)}))] \le \delta, \end{aligned} \tag{5}$$

where $\pi_{\theta_{old}}$ is the old policy. $D_{\mathrm{KL}}(p||q)$ refers to the KL-divergence between distribution p and q [24]. $A_{\pi_{\theta_{old}}}(a^{(t)},s^{(t)})$ refers to the advantage function in Equation (4). By solving Equation (5), the new policy π_θ can be obtained.

As is discussed in [43], solving Equation (5) is computationally expensive because it requires a second-order approximation of the KL divergence and computing Hessian matrices. To address this problem, Schulman *et al.* [43] proposed the PPO objective function, which replaces the KL-constrained objective in Equation (5) by a clipped objective function

$$\begin{aligned} &\text{maximize}_\theta\ \mathbb{E}_{(a^{(t)},s^{(t)})\sim\pi_{\theta_{old}}}[\min(\text{clip}(\rho^{(t)},1-\varepsilon,1+\varepsilon)A^{(t)},\rho^{(t)}A^{(t)})], \\ &\rho^{(t)} = \frac{\pi_\theta(a^{(t)}|s^{(t)})}{\pi_{\theta_{old}}(a^{(t)}|s^{(t)})},\quad A^{(t)} = A_{\pi_{\theta_{old}}}(a^{(t)},s^{(t)}). \end{aligned} \tag{6}$$

Here, $\text{clip}(\rho^{(t)},1-\varepsilon,1+\varepsilon)$ denotes clipping $\rho^{(t)}$ to the range of $[1-\varepsilon,1+\varepsilon]$ and ε is a hyper-parameter. During the training process, in addition to updating the actor by solving the optimization function in Equation (6), the PPO algorithm iteratively updates the action-value function $Q_w(s,a)$ as well as the state-value function $V_v(s)$ (*i.e.,* the critic) by using the Temporal-Difference method.[3] In Figure 2, we show the network structure used in the PPO algorithm. As we can observe from the figure, the network structure contains two deep neural networks, one for approximating the state-value function $V_v(s)$ and the other for modeling the policy network π_θ. It should be noted that the implementation of PPO algorithm does not introduce an additional neural network to approximate action-value function $Q_w(s,a)$ but to deduce it through the state-value function $V_v(s)$.

[3]While the Monte-Carlo method is also available for the training, due to the performance concern, the standard implementation of PPO considers only the Temporal-Difference method.

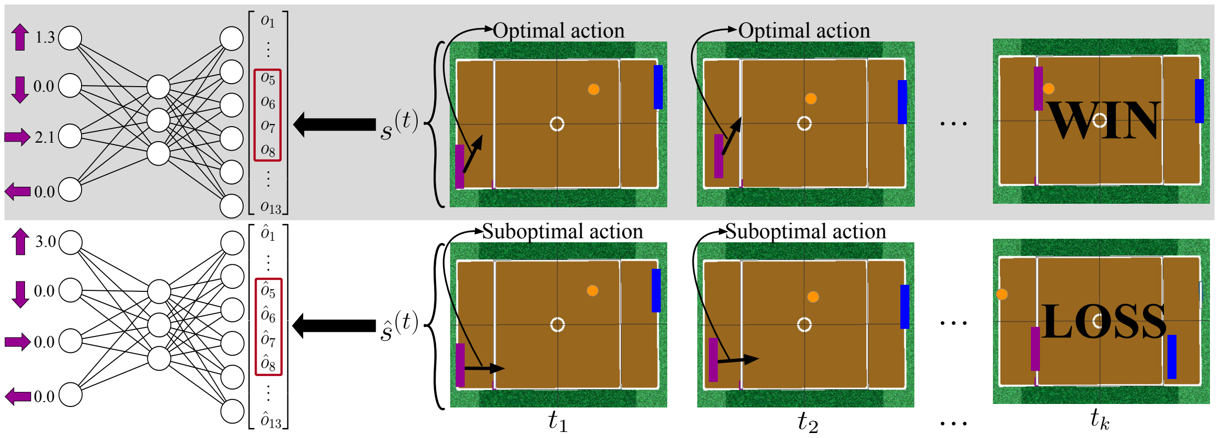


Figure 3: The overview of our proposed attack. The upper part on a grey canvas demonstrates a game episode where the policy network of the opponent agent outputs the optimal actions and the opponent agent (in purple) wins the game. The lower part shows an episode in which the adversarial agent (in blue) subtly manipulates the environment through its actions, forces the opponent agents to choose a sequence of sub-optimal actions, and thus defeats the opponent. The arrow tied to the purple paddle indicates the action the opponent agent takes. At each time step, the adversarial agent only introduces an imperceptible change to the environment and therefore the scenes (or in other words states) on the grey canvas are nearly as same as those on the white canvas (i.e, $\|s^{(t)} - \hat{s}^{(t)}\| \leq \varepsilon$ where ε is a small number restricting the action change of the adversarial). The feature vector passing to the networks indicates the observation of the opponent agent. It is converted from the states of the opponent agent $s^{(t)}$ and $\hat{s}^{(t)}$. The features in the red box ($o_5 \cdots o_8$ and $\hat{o}_5 \cdots \hat{o}_8$) represent those corresponding to the adversarial action.

Different from the previous actor-critic algorithms, which update actor by conducting stochastic gradient-ascend[4] using the approximated policy gradient of Equation (4), the PPO algorithm can guarantee a monotonic improvement of the total rewards when updating the policy network (*i.e.,* $J(\theta) \geq J(\theta_{old})$). With this property, the trained agent could not only reach to the convergence faster but, more importantly, demonstrate more accurate and more stable performance than the previous actor-critic algorithms. To the best of our knowledge, PPO is the state-of-art algorithm for training a policy network for the agent in the two-agent competitive games. As such, we design our attack by extending this PPO training algorithm.

4 Technical Overview

Recall that we attack a well-trained agent by training a powerful adversarial agent. To achieve this, as is mentioned in Section 2, we do not assume that an attacker has access to the policy network of the opponent agent π_v nor its state-transition model $P^v_{ss'}$. Rather, we assume the attacker could obtain the observation and action of the opponent (*i.e.,* the state $s^{(t)}_v$ and action $a^{(t)}_v$ of the opponent agent at each time step t). In this section, we first specify the basic idea of our attack method. Then, we briefly describe how to utilize the aforementioned states and actions to extend the PPO algorithm and thus implement our attack method at a high level.

[4]Note that the performance of stochastic gradient-ascend highly depends on the step size and it cannot guarantee to increase the objective function monotonically.

4.1 Basic idea of the proposed attack

Admittedly, it is possible to design a simple reward function for an adversarial agent to beat its opponent. However, the reward function design is usually game-specific, and it is challenging to design a universal solution. As such, we follow a different strategy to fulfill our objective as follows. In a two-agent competitive game, one could train an agent to take an optimal action at each state via selfplay [3]. Therefore, as is depicted in Figure 3, to influence a well-trained agent, one method is to maximize the deviation of the actions taken by that agent and thus make the agent output a suboptimal action (*i.e.,* given the same/similar environment observation, an agent takes an action which is very different from the one it is supposed to take). With this practice, from the adversary's viewpoint, he can downgrade the opponent agent's performance and thus reduce its winning rate.

To maximize the action deviation, an adversary would inevitably vary the observation of the victim agent. As is mentioned above, a suboptimal action means that, given the same or similar observation, the action of the agent is very different from the one it is supposed to take. Therefore, as we will specify in the following, when maximizing the deviation of an opponent action, we need to ensure the minimal change of the environment observation.

Recall that we do not assume an adversary has the privilege to manipulate the environment, and, in a two-agent compet-

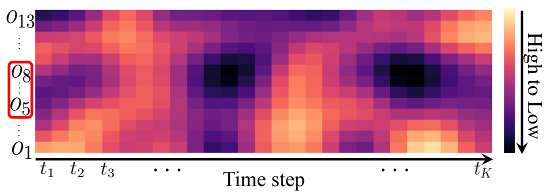


Figure 4: A heatmap indicating the input feature importance of the opponent policy network of Roboschool Pong game. The highlighted features ($o_5 \cdots o_8$) represent those corresponding to the adversarial action. The heapmap is generated by using the output of explainable AI techniques.

itive game, the action of the adversarial agent is converted as part of the environment observation of its opponent agent. Take the example shown in Figure 3. The opponent observation is depicted as a feature vector, within which some of the features represent the adversarial actions. As such, we can subtly manipulate the action of the adversarial agent and thus change the features indicating the adversarial action. With this, we can change the input to the opponent's policy network and indirectly deviate the action of the opponent agent.

However, as is shown in Figure 4, by performing a sensitivity check for the policy network against the input features over time, we note that the opponent's policy network takes the importance of the input features differently over time. Therefore, intuition suggests that the best strategy is to perform the corresponding feature manipulation only at the time when the opponent policy network pays sufficient attention to the features corresponding to the adversarial actions. To achieve this, as we will specify below, we utilize an explanation AI technique to examine the the victim policy network' feature importance at each time step. With this, we can pinpoint the time frame when the victim policy network pays its attention to the adversarial action, and thus employ an adjustable hyperparameter to control the level of action deviation adjustment.

4.2 More details

As is stated above, we design our attack in two steps – ❶ deviating the actions of the opponent agent with a minimal change to its observation, and ❷ adjusting the weight of the action deviation of the opponent agent based on the influence of the adversarial actions upon the opponent. In the following, we specify how we implement this two-step design.

Deviating opponent actions. To deviate the action of the opponent, we extend the PPO loss function L_{PPO} mentioned in Section 3. To be specific, we introduce into the PPO loss function a new loss term

$$L_{ad} = \text{maximize}_{\theta}(-\|\hat{o}_v^{(t+1)} - o_v^{(t+1)}\| + \|\hat{a}_v^{(t+1)} - a_v^{(t+1)}\|), \quad (7)$$

where θ represents the parameters in π_α. $\hat{o}_v^{(t+1)}$ and $\hat{a}_v^{(t+1)}$ indicate the different observation and action taken by the opponent agent if, at the time step t, the adversarial agent takes an action different from the ones indicated by the trajectory rollouts (*i.e.,* different from the actions that the opponent is supposed to take). As we can observe from the equation above, the loss term contains two components. The design of the first component ensures that, when launching attacks, an adversary introduces only minimal variations to the observation of the opponent agent. The design of the second component forces the opponent agent to take a suboptimal action $\hat{a}_v^{(t+1)}$ but not the optimal action $a_v^{(t+1)}$, and thus trigger the drop of its winning rate. It should be noted that we compute both the action difference and observation difference by using a norm, the output of which is a singular. As such, when we can combine the observation and action differences in a linear fashion.

As is mentioned in Section 2, neither the opponent policy network π_v nor its state-transition model $p_v^{ss'}$ is available for our method. Without the state-transition model, we cannot predict the observation of the opponent agent $\hat{o}_v^{(t+1)}$ at the time step $t+1$, when our adversarial agent takes an action at the time step t and subtly varies the observation of the opponent at the time step $t+1$. Without the access to the policy network, even if $\hat{o}_v^{(t+1)}$ is given, we still cannot predict the action of the opponent agent $\hat{a}_v^{(t+1)} = \pi_v(\hat{o}_v^{(t+1)})$ at the time step $t+1$. This imposes the challenge of computing the loss term L_{ad} in Equation (7).

To tackle the challenge, our method approximates the opponent policy network as well as its state-transition model by using two individual deep neural networks. By definition, the state-transition model outputs the predicted observation of the opponent $o_v^{(t+1)}$ at the time step $t+1$. It takes as input the observation of the opponent $o_v^{(t)}$, the action of the adversarial agent $a_\alpha^{(t)}$, and that of the opponent agent $a_v^{(t)}$ at the time step t. As we specify in Section 5, we train both of the neural networks by using trajectory rollouts. It should be noted that, to train the surrogate model, the attack needs to access victim observation and action, which is a legitimate assumption (See Section 2). However, we also admit that the proposed attack would become harder when this information is not available. This is because the attacker needs extra effort to infer such information and then train the surrogate model with the approximated victim observation and action.

Adjusting weights of action deviation. As is mentioned above, the opponent/victim agent weights the action of the adversarial differently over time when deciding its own action through its policy network. As a result, when leveraging the action of the adversarial to influence the environment observation and thus the action of the opponent agent, we adjust the weight of the action deviation based on by how much the victim agent pays attention to the action of the adversarial. To achieve this, when optimizing the extended loss function $L_{ppo} + L_{ad}$, we introduce a hyperparameter λ, indicating the importance of our newly added term L_{ad}. With this, we can rewrite the extended loss function as $L_{ppo} + \lambda \cdot L_{ad}$. To maximize this loss function, we can adjust the weight assigned to the new term (*i.e.,* L_{ad}) based on the weight that the opponent/victim agent pays attentions to the adversarial.

In this work, we utilize an explanation AI technique to measure the weight that the victim agent pays attention to the adversarial action. As is shown in Figure 3, the actions of the adversarial are part of the observation of the victim agent. They are encoded as part of the features passing to the victim's policy network. In Figure 3, we can easily observe that a policy network is a deep neural network. Over the time, the observation feature vector passing to the network varies. Using an explanation AI technique at each time step against the victim policy network, we can measure by how much the policy network pays attention to the features corresponding to the action of the adversarial.

Intuition suggests that, to obtain an optimal effect upon the deviation of the opponent, the adversarial agent should manipulate its actions at the time when the opponent pays its attention to the adversary. Otherwise, the action manipulation of the adversarial agent will introduce minimal influence upon the action of the opponent agent. Following this intuition, we assign the value for λ at each time step t based on the output of an explainable AI technique. More specifically, we assign a higher value to the weight λ when the opponent pays more attention to the adversarial agent. Otherwise, we assign a relatively low value on the weight to minimize the impact of our newly added term. For more details of our weight assignment, readers could refer to Section 5.

Over the past years, there are many techniques in the field of explanation AI research, ranging from black-box methods (*e.g.,* [9, 39]) to white-box approaches (*e.g.,* [46–48]). Among all these explanation AI techniques, we choose gradient-based interpretation methods, serving as the way to weight the influence of the adversarial actions upon opponent's policy network. The rationales behind our choice is as follow. In comparison with other explanation AI methods, such as some black-box methods [39] which need to perform intensive data sampling before deriving explanation, gradient-based methods are computationally efficient. In the context of deep reinforcement learning, the observation of the opponent/victim agent $o_v^{(t)}$ changes over time rapidly and we need to adjust the hyperparameter λ at each time step. In this work, we rely upon gradient-based methods, which can minimize the computation needed for weight adjustment. Considering that past research [1] indicates different gradient-based explanation methods provide different accuracy in explanation, we thoroughly evaluate by how much the choice a particular gradient based method would influence the performance of our attack. We show our evaluation results in Section 6.

5 Technical Detail

In this section, we provide more details about our proposed method. More specifically, we first formally define the problem that our method targets. Then, we specify the design of our loss term. Finally, we discuss how we extend our loss function through explainable AI and present our learning algorithm as a whole.

5.1 Problem definition

Following the early research [44], we also formulate a two-agent competitive game as a two-agent MDP, represented by $M =< \mathcal{S}, (\mathcal{A}_\alpha, \mathcal{A}_v), \mathcal{P}, (\mathcal{R}_\alpha, \mathcal{R}_v), \gamma >$. Here, $\mathcal{S}$ denotes the state set. $\mathcal{A}_\alpha$ and $\mathcal{A}_v$ are the action sets for adversarial and opponent agents, respectively. $\mathcal{P}$ represents a joint state transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A}_\alpha \times \mathcal{A}_v \to \Delta(\mathcal{S})$. The reward function can be represented as $\mathcal{R}_i : \mathcal{S} \times \mathcal{A}_\alpha \times \mathcal{A}_v \to \mathbb{R}$; $i \in \{\alpha, v\}$.

As is mentioned in Section 3, the state transition is a stochastic process. Therefore, we use $\Delta(\mathcal{S})$ to represent a probability distribution on $\mathcal{S}$, from which the state at each time step can be sampled. Note that using the PPO algorithm for training agents in a two-agent competitive game, we cannot obtain the state $\mathcal{S}$ and the state transitions function $\mathcal{P}$ in an explicit form. From the game environment, each of the agents can get only its own observation O_i; $i \in \{\alpha, v\}$.

In this paper, we assume that the opponent agent follows a fixed stochastic policy π_v. Holding this assumption, our problem can be viewed as a single-agent MDP for the adversarial agent, denoted by $M_\alpha =< \mathcal{S}, \mathcal{A}_\alpha, \mathcal{P}_\alpha, \mathcal{R}_\alpha, \gamma >$. Here, the state-transition model $\mathcal{P}_\alpha$ is unknown, and $\mathcal{S}$ is equivalent to the observation of the adversarial agent O_α. Under this problem definition, the goal of this work is to identify an adversarial policy π_α that can guide the corresponding agent to beat its opponent in the single-agent MDP.

5.2 Expected reward maximization

As is described in Section 4, we extend the PPO loss function when designing our proposed method. As is introduced in the early section, the PPO loss function can be written as

$$\begin{aligned} &\text{maximize}_\theta \ \mathbb{E}_{(a_\alpha^{(t)}, o_\alpha^{(t)}) \sim \pi_\alpha^{old}} [\min(\text{clip}(\rho^{(t)}, 1-\varepsilon, 1+\varepsilon)A^{(t)}, \rho^{(t)}A^{(t)})], \\ &\rho^{(t)} = \frac{\pi_\alpha(a_\alpha^{(t)} | o_\alpha^{(t)})}{\pi_\alpha^{old}(a_\alpha^{(t)} | o_\alpha^{(t)})}, \quad A^{(t)} = A_{\pi_\alpha^{old}}((a_\alpha^{(t)}, o_\alpha^{(t)})) . \end{aligned} \tag{8}$$

Here, π_α^{old} and π_α denotes the old and new policy of the adversarial agent, respectively. $o_\alpha^{(t)}$ is the observation of the adversarial agent at the time step t. It encloses the action of the opponent agent $a_v^{(t)}$. Following the standard PPO algorithm, we use a neural network $V_\alpha(s)$ to approximate the state-value function, and thus obtain the advantage $A^{(t)}$ at the time step t. In this work, the model architectures of the state-value function and the policy network are as same as those in the PPO algorithm (see Figure 2). By solving the objective function above, we could find an adversarial policy π_α, with which the corresponding adversarial agent could maximize the expected total reward: $\sum_0^\infty \gamma^{(t)} \mathcal{R}_\alpha(s^{(t)}, a_\alpha^{(t)})$.

5.3 Action deviation maximization

Recall that we extend the PPO loss function by introducing a new loss term

$$L_{ad} = \text{maximize}_{\theta}(\|\hat{a}_v^{(t+1)} - a_v^{(t+1)}\|_1 - \|\hat{o}_v^{(t+1)} - o_v^{(t+1)}\|_1)\,. \quad (9)$$

As is shown above, we choose l_1 norm distance as the difference measure instead of l_2 norm. This is because l_1 norm encourages a larger difference than l_2 norm, especially when O_v is of a high dimensionality [2]. As we will empirically show in Section 6, an adversarial agent trained with the l_1 norm usually demonstrates a stronger capability of beating opponent agents than that trained with l_2 norm.

State transition approximation. To predict the observation of the opponent agent at the time step $t+1$, we utilize a deep neural network to approximate the state-transition model of the opponent agent. As is mentioned in Section 4, the deep neural network takes as input $(o_v^{(t)}, a_v^{(t)}, a_\alpha^{(t)})$, and predicts $o_v^{(t+1)}$ (*i.e.,* the observation of the opponent agent at the time step $t+1$). In this work, we train this neural network by using the following equation

$$\text{argmin}_{\theta_h} \|H(o_v^{(t)}, a_v^{(t)}, a_\alpha^{(t)}; \theta_h) - o_v^{(t+1)}\|_\infty\,, \quad (10)$$

where θ_h denotes the parameters of the neural network H. It should be noted that $\|\cdot\|_\infty$ is non-differentiable. Therefore, we adopt the approximation technique introduced in [7], and use the alternative objective function

$$L_{st} = \text{minimize}_{\theta_h} \|(|H(o_v^{(t)}, a_v^{(t)}, a_\alpha^{(t)}) - o_v^{(t+1)}| - \varepsilon_s)^+\|_2^2\,, \quad (11)$$

to train the approximated state-transition model H. In the equation above, $(\cdot)^+$ is equivalent to $\max(\cdot, 0)$. ε_s is a hyperparameter, which controls the maximum l_∞ between $H(o_v^{(t)}, a_v^{(t)}, a_\alpha^{(t)})$ and $o_v^{(t+1)}$. To solve this objective function, we collect the ground truth training data $(o_v^{(t)}, a_v^{(t)}, a_\alpha^{(t)}, o_v^{(t+1)})$ by using trajectory rollouts. Then, we utilize the `ADAM` optimization method [20] to minimize this objection function. More specifically, as is shown in Algorithm 1 (step 7), the state-transition model is trained jointly with the policy network of the adversarial agent. At each iteration, we first collect a set of trajectories by using current adversarial policy to play against the opponent agent. The information contained in the collect trajectories includes the opponent agent's actions and observations. Using these actions and observations as the ground truth, we can update the surrogate networks by minimizing the loss functions above. It should be noted that, while the state transition model H should be obtained based on the old adversarial policy, we predict the state transition under the new adversarial policy. We argue this does not introduce negative effect to our training process because the PPO objective function guarantees a minor change in the adversarial policy at each iteration.

Opponent policy network approximation. As is shown in Equation (9), computing action deviation requires $a_v^{(t+1)}$ and $\hat{a}_v^{(t+1)}$. In addition, as is mentioned earlier, our attack relies upon the capability of knowing how a victim agent weights the importance of the adversarial actions. To do that, as we will elaborate in Section 5.4, we leverage gradient-based explanation AI techniques, which need to take as input the policy network of the victim agent. As such, in addition to the state transition approximation, we use a deep neural network F to approximate the policy network of the opponent agent.

In this work, to learn the victim's policy network, we follow existing imitation learning methods [52] and design the following objective function

$$L_{op} = \text{minimize}_{\theta_f} \|(|F(o_v^{(t)}; \theta_f) - a_v^{(t)}| - \varepsilon_a)^+\|_2^2\,. \quad (12)$$

Here, θ_f represents the parameters of the deep neural network F. As we can observe from the equation above, we also use the approximated l_∞ loss to train F. Similar to the method above, we also collect the training samples $(o_v^{(t)}, a_v^{(t)})$ through trajectory rollouts and then apply the `ADAM` algorithm to minimize the loss. As we will empirically illustrate in Section 6, the network trained with l_∞ norm usually exhibits better performance than those trained with l_2 and l_1.

Note that, in MDP, both the state transition and the policy network should be in the form of stochastic. This means that the most typical way of approximating $\mathcal{P}$ and π should be density estimation [14]. In this work, we, however, conduct point estimations to reduce the computational cost. As we will show in Section 6, while point estimate ignores the variance of the original distribution and may introduce a bias, our attack is still able to achieve decent performance in terms of beating the opponent in the two-agent competitive game.

After obtaining the approximated models H and F, we can predict the observation of the opponent agent $\hat{o}_v^{(t+1)}$ through $H(o_v^{(t)}, a_v^{(t)}, \hat{a}_\alpha^{(t)})$, and its action $\hat{a}_v^{(t+1)}$ through $F(H(o_v^{(t)}, a_v^{(t)}, \hat{a}_\alpha^{(t)}))$. With these predictions, we can rewrite Equation (9) as

$$\begin{aligned} L_{ad} = \text{maximize}_{\theta}(&\|F(H(o_v^{(t)}, a_v^{(t)}, \hat{a}_\alpha^{(t)})) - a_v^{(t+1)}\|_1 \\ &- \|H(o_v^{(t)}, a_v^{(t)}, \hat{a}_\alpha^{(t)}) - o_v^{(t+1)}\|_1)\,. \end{aligned} \quad (13)$$

Here, it should be noted that $\hat{a}_\alpha^{(t)}$ is the new action derived from the adversarial policy π_α.

5.4 Hyperparameter adjustment

As is mentioned in Section 4, we introduce a hyperparameter to balance the weight of the newly added loss term. In this work, we automatically adjust λ by using an explainable AI technique. More specifically, by using the gradient saliency methods (*e.g.,* [46]) at the time step t, we first compute $g^{(t)} = \nabla_{o_v^{(t)}} F(o_v^{(t)})$ which indicates the importance of each element in the opponent agent's observation.[5] In this equation, $F(o_v^{(t)})$

[5]Note that we do not have the access to the opponent policy network and, therefore, we compute the gradient on the basis of its approximation F.

denotes the action of the opponent agent $a_v^{(t)}$ predicted by F.

Supposing $o_v^{(t)} \in \mathbb{R}^{p\times 1}$ and $F(o_v^{(t)}) \in \mathbb{R}^{q\times 1}$, the gradient $g^{(t)} \in \mathbb{R}^{p\times q}$ is a matrix, in which each element $g_{ij}^{(t)} = \nabla_{(o_v^{(t)})_i} F(o_v^{(t)})_j$ indicates the importance of the i-th element in $o_v^{(t)}$ to the j-th element in $F(o_v^{(t)})$. To assess the overall importance of each element in $o_v^{(t)}$ to $F(o_v^{(t)})$, we sum the elements in each row of $g^{(t)}$ and transform it into a normalized vector $\tilde{g}^{(t)} = \sum_{j=1:q} g_{ij}^{(t)}$. Here, $\tilde{g}^{(t)} \in \mathbb{R}^{p\times 1}$ indicates the importance of the i-th element in $o_v^{(t)}$ to $F(o_v^{(t)})$.

After obtaining $\tilde{g}^{(t)}$, we then calculate the importance of the adversarial agent's action to the opponent agent's action at the time t. Recall that the observation of the opponent agent $o_v^{(t)}$ contains three components – environment, the action of the opponent agent, and that of the adversarial agent – and we focus only on the action of the adversarial agent. Therefore, we eliminate the feature importance tied to the environment and the action of the opponent agent. To do this, we first perform an element-wise multiplication between $\tilde{g}^{(t)}$ and a mask $M \in R^{p\times 1}$. Then, we borrow the idea of an early research work [9], through which we compute λ as follows

$$I^{(t)} = \|F(o_v^{(t)}) - F(o_v^{(t)} \odot (\tilde{g}^{(t)} \odot M))\|_\infty, \ \lambda^{(t)} = \frac{1}{1+I^{(t)}}. \quad (14)$$

Here, the vector $o_v^{(t)}$ is a vector, indicating the observation at time t. M is a vector with the same dimensionality as the vector $o_v^{(t)}$. In $o_v^{(t)}$, if the corresponding observation dimensions indicate the actions of adversarial agent, we assign 1 to the corresponding element in M. Otherwise, we assign 0 accordingly. For example, assuming the $k^{th} \sim (k+N)^{th}$ dimensions of $o_v^{(t)}$ indicate the actions of the adversarial agent. Then, we assign 1 to the k^{th} to $(k+N)^{th}$ dimensions of M, and the rest is assigned to 0. In this work, we normalize $\lambda^{(t)}$ to $[0,1]$.[6] From this equation, we can easily discover that, the higher value of $I^{(t)}$ indicates a lower importance score, resulting in a lower value of $\lambda^{(t)}$. In Algorithm 1, we illustrate how to combine λ with our extended loss function, and thus train an adversarial agent with the ability to attack its opponent.

6 Evaluation

In this section, we evaluate our proposed attack technique from various aspects, compare it with the state-of-the-art method, and demonstrate its effectiveness and efficiency by using representative two-agent competitive games. Below, we first present our experiment setup. Then, we discuss the design of our experiment, followed by our experiment results.

[6]Normalization could capture temporal changes and prevent the influence of its extreme values upon the PPO learning process.

Algorithm 1: Adversarial policy training algorithm.

```
1  Input: the adversarial agent's policy $\pi_\alpha$ parameterized by
          $\theta_\alpha$, the adversarial agent's value function network
          $V_\alpha$ with parameter $v_\alpha$, the state transition model $H$ with
          parameter $\theta_h$, the opponent's policy approximation
          model $F$ with parameter $\theta_f$, and the pretrained
          opponent agent's policy $\pi_v$.
2  Initialization: Initialize $\theta_\alpha^{(0)}$, $\theta_h^{(0)}$, $\theta_f^{(0)}$, and $v_\alpha^{(0)}$.
3  for $k = 0,1,2,...,K$ do
4      Collect a set of trajectories $\mathcal{D}^k = \{\tau_i\}$ by using adversarial
         policy $\pi_\alpha^k$ to play against the opponent agent $\pi_v$, where
         $i = 1,2,...,|\mathcal{D}^k|$ and each trajectory contains $T$ time step.
5      Obtain the reward of the time $t$ in each trajectory $\tau_i$: $r_\alpha^{i(t)}$.
6      Compute the estimated advantage of each time in each
         trajectory: $A^{i(t)}$ based on the current value function $V_{\alpha^k}$:
         $A^{i(t)} = r_\alpha^{i(t)} + \gamma V_{\alpha^k}(o_\alpha^{i(t+1)}) - V_{\alpha^k}(o_\alpha^{i(t)})$.
7      Update the state transition approximation function $H$ and
         the opponent policy approximation function $F$ using the
         current trajectories according to the following objective
         function
           $\theta_h^{k+1} = \text{argmin}_{\theta_h} \frac{1}{|\mathcal{D}^k|T} \sum_{\tau\in\mathcal{D}^k} \sum_{t=0}^{T} L_{st}$,
           $\theta_f^{k+1} = \text{argmin}_{\theta_f} \frac{1}{|\mathcal{D}^k|T} \sum_{\tau\in\mathcal{D}^k} \sum_{t=0}^{T} L_{op}$.        (15)
8      Based on the updated $o_v^{i(t)}$, $a_v^{i(t)}$ in $\mathcal{D}^k$, and $F_{\theta_f^{k+1}}$, compute
         the penalty term for each time $t$ in each trajectory $i$: $\lambda^{i(t)}$
         according to Equation (14).
9      Update the policy by maximizing the following objective
         function
           $\theta_\alpha^{k+1} = \text{argmax}_{\theta_\alpha} \frac{1}{|\mathcal{D}^k|T} \sum_{i=1}^{|\mathcal{D}^k|} \sum_{t=0}^{T} L_{ppo} + \lambda^{i(t)} L_{ad}$.        (16)
10     Update the value function by minimizing the following
         objective function
           $v_\alpha^{k+1} = \text{argmin}_{v_\alpha} \frac{1}{|\mathcal{D}^k|T} \sum_{i=0}^{\mathcal{D}^k} \sum_{t=0}^{T} (V_{\alpha^k}(o_\alpha^{i(t)}) - (r_\alpha^{i(t)} + \gamma V_{\alpha^k}(o_\alpha^{i(t+1)})))^2$.
                                                                       (17)
11 end
12 Output: the well trained adversarial policy network $\pi_\alpha$.
```

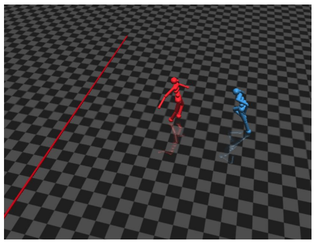
6.1 Experiment setup

In our experiment, we choose the game "You Should Not Pass" in the MuJoCo game zoo [50], which has recently been adopted to demonstrate the effectiveness of a state-of-the-art adversarial attack [10]. As we will specify in the consecutive session, by using this game, we not only evaluate the key components of our proposed design but, more importantly, compare the effectiveness of our proposed technique with that of the state-of-the-art method [10]. In addition to the MuJoCo game, we demonstrate our method on the roboschool Pong game [33]. Together with the MuJoCo game, we quantify by how much our proposed method outperforms the state-of-the-art technique [10]. We believe the games of our choice are representative for the following three reasons. First, both games provide us with the interface to train agents using reinforcement learning algorithms, giving us the freedom to develop our attack method. Second, as is discussed in Section 2, our attack targets competitive games in which reinforcement learning algorithms are commonly used to train agents. Both games of our choice are commonly used in academia for evaluating reinforcement learning algorithms in two-agent settings (*e.g.,* [3]) and attack methods in adversarial learning (*e.g.,* [10]). Third, we design our attack based on the PPO algorithm. When we choose games, we need to ensure, the PPO algorithm should be the one most commonly used for the games of our choice. Both MuJoCo and Roboschool hold this selection criterion. In the following, we briefly introduce both of these games, the opponent agents in both games, and the evaluation metric.

MuJoCo. In this game, two agents (*i.e.,* players) are first initialized to face each other. As is illustrated in Figure 5a, the blue humanoid robot then starts to run towards the finish line (indicated by the red line in Figure 5a). In this process, the red humanoid robot in the figure attempts to block the blue robot from reaching the line right behind it. By design, the blue robot could win the game only if it reaches the finish line. Otherwise, the other robot wins. When playing this game, both robots observe the game environment, the current status of themselves (*e.g.,* the position and velocity of their body), and that of their opponent. Based on the observation, they both utilize a policy network to decide their actions (*i.e.,* the direction and velocity of the next movement). The game ends when the winning condition is triggered. At that time, the winner receives a reward, whereas the loser gets penalized.

Roboschool Pong. As is depicted in Figure 5b, the Pong game features two paddles and a ball. The reinforcement learning agents control the movement of the paddles through policy networks. At the beginning of the game, one agent serves the ball, and the other returns the serve. In each round of the game, an agent can claim a win only if its opponent fails to return the ball or violates the rule of the game (*e.g.,* successively hit the ball twice). If a single round of the game runs out of time, a timeout will be triggered and the game will conclude a tie. In this game, the observation of an agent contains the agent itself, the opponent agent, and the position and velocity of the ball. Based on the observation, through its policy network, the agent can take an action indicated by the direction and velocity of its next movement. When playing this game, agents will receive a reward or be penalized based on the performance of the agent.

Opponent agents. Following the work proposed in [10], regarding the MuJoCo game, we treat the blue humanoid robot as the opponent agent and the red one as our adversarial agent. For the Pong game, we take the purple paddle (on the right of Figure 5b) as the opponent agent whereas the other as


(a) MuJoCo.

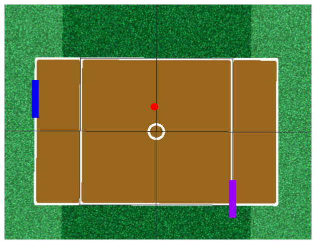

(b) Roboschool Pong.

Figure 5: The illustration of the selected games.

the adversarial one.[7] In this work, the policy networks of opponent agents are all modeled as multilayer perceptrons, which are trained through a self-play mechanism [3] because this neural architecture has been broadly used by previous research [3, 10, 33] and already demonstrated the best performance in both MujoCo and Pong game. To be more specific, for the MuJoCo game, we used the pre-trained policy network released in the "agent zoo" [3] as the opponent policy network. For the roboschool Pong game, we first trained a policy network through the self-play mechanism by using the PPO algorithm. Then we treated it as the opponent policy network. We specify the architectures of these two opponent policy networks in the Appendix.

Evaluation metric. Different from supervised learning algorithms, many reinforcement learning algorithms typically do not involve a data set collected offline for training an agent. Instead, they usually expose a learning agent to interact with the environment for many iterations. In each iteration, the learning agent collects trajectories by using its policy network learned from the last iteration, update its current policy network with the new trajectories, and proceed to the next iteration. In our experiment, we follow the metric commonly used for evaluating reinforcement learning, measuring the winning rate of the adversarial agent at each iteration. Given the property of the competitive game, by subtracting the winning rate of the adversarial agent, we can easily obtain that of the opponent. The higher the winning rate for an adversarial agent is, the more powerful the adversarial agent is in terms of exploiting the weakness of its opponent.

6.2 Experiment design

We design our experiment from two different perspectives. One is to evaluate some components of our proposed technique, and the other is to quantify the overall performance of our proposed method. In the following, we describe the detail of each of our experiment designs.

Experiment I. Recall that we utilize gradient-based explainable AI techniques to guide the selection of the hyperparameter λ. To understand the contribution of the explanation

[7]Note that the two agents are symmetric; therefore, the choice of the opponent agent does not influence the effectiveness of the learning algorithm.

component in our loss function, we first design an experiment, in which we set up the hyperparameter λ with different constant values, run our learning algorithm under this setting on the MuJoCo game, and compare the performance of the trained agent under each constant value with the one obtained through our explanation-based method. With respect to the explanation-based method, we choose different gradient-based explainable AI techniques to serve as the explanation component. In this experiment, we compare the corresponding performance of the adversarial agent under each of our choices. More specifically, the gradient-based explainable AI methods in our choice set include vanilla gradient [46], integrated gradient [48], and smooth gradient [47]. In addition to these well-recognized gradient-based methods, our choice set encloses a random explanation approach as a baseline method, which derives feature importance score randomly.

Experiment II. We also design an experiment to validate the choice of our distance measure. As is mentioned in Section 5, we carefully design the measure of distance indicated by Equation (11), (12), and (13). To ensure our choice of the distance measure could truly benefit the agent trained by our proposed method, we replace the corresponding distance measures with the l_1 and l_2 norm respectively. In this work, we compare the performance of the trained agent under each of these setups.

Experiment III. We further design an experiment to examine whether the approximated opponent policy network involved in our technique imposes any risk of downgrading our agent's performance. As is mentioned in Section 4, to derive an explanation and thus guide the adjustment of the hyperparameter λ, we approximate the policy network of the opponent. Since this approximation is based on point estimation, this inevitably incurs errors and thus potentially influences the performance of the adversarial agent trained by our method. To test its impact upon the adversarial agent's performance, we replace the approximated policy network with the actual policy network of the opponent agent, run the proposed learning algorithm, and compare the performance of the corresponding agent with the one obtained through our method.

Experiment IV. Using the state-of-the-art attack method [10] as our baseline, we also design an experiment to evaluate our proposed method. To be specific, we use both methods to train adversarial agents and then apply them in the MuJoCo game and the Pong game. In each of the games, we then compare the winning rate of the adversarial agents across the number of iterations involved in the training process. This is similar to the setup proposed in an early research [10].

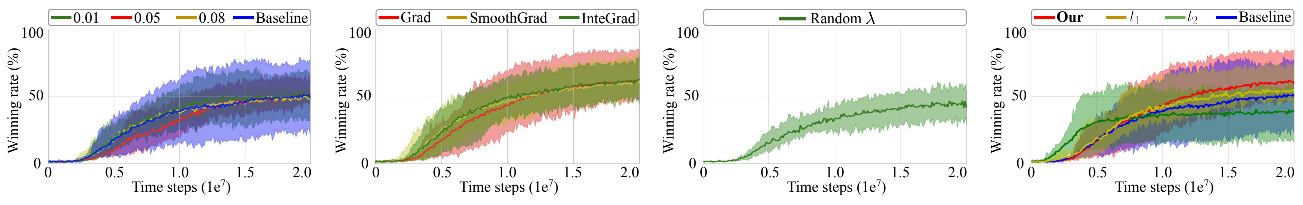
Experiment V. Finally, we investigate a simple adversarial training approach to safeguard victim models against the proposed attack. More specifically, We play the victim agent with the adversarial agent trained by our attack in the corresponding game environment and collected the game episodes. With these episodes, we then utilized our proposed learning algorithm (Algorithm 1) to retrain the victim agent. Similar to the experiment above, we compare the winning rate of the retrained victim agent against the adversarial agent across the number of iterations involved in the retraining process. In addition, we employ the retrained victim agent to play with an agent trained with self-play methods. With this setup, we emulate a scenario where a robustified agent plays with a regular (non-robustified) game agent. Through this, we study if retrained victim agent could pick up the generalizability in competitive game. In other words, we study whether a victim agent still performs well when playing with a regular agent even after we retrain it with adversarial training.

Additional experiment notes. It should be noted that, when running any learning algorithms to train adversarial agents and perform the aforementioned experiments, we go beyond the suggestion mentioned in [10], increasing the number of different initial states from 5 to 8 for each agent training. With this setup, we can not only obtain the average performance of each learning algorithm but also further reduce the influence of randomness. It should also be noted that, when training an agent, we cut off our training process after the training reaches 20 million iterations for the MuJoCo game and 4 million iterations for the Pong game. This is because our empirical evidence indicates that, after these numbers of iterations, the performance of the adversarial agent (the winning rate) converges. Our method involves multiple hyper-parameters. In our experiment, we conduct the sensitivity test for the main hyper-parameters: the explanation method, λ, the distance measure, η (Appendix). We find that our attack is robust to all these hyper-parameters except the distance measure. We present our choice of distance measure in Section 5 and validate our choice in Experiment II. Regarding the hyper-parameters inherited from our baseline [10], we apply the default choices in [10] for a fair comparison. In the Appendix, we specify the choices of the other hyper-parameters that are not varied in the sensitivity test and how we decide them. For the video demonstration of our adversarial agents, readers could find them at `https://tinyurl.com/vsnp5jr`.

6.3 Experiment result

Here, we present the experiment results and analyze the reasons behind our findings.

Comparison of hyperparameter selection strategies. Figure 6a shows the performance of the adversarial agent trained with different hyperparameter selection strategies. As we can observe from the figure, when the hyperparameter λ is set up with a constant (*i.e.,* red, green, and yellow lines in Figure 6a), the winning rate of the adversarial agent converges at about 50% on average, which is comparable to the performance of the adversarial agent trained by the state-of-the-art method [10] (indicated by baseline in Figure 6a). However, when using an explainable AI technique to adjust this hyperparameter over time, we can easily observe about 10% improvement in the winning rate (about 60% vs. 50%). This



(a) Constant λ. (b) Explainable AI techniques. (c) Random λ. (d) Distance measures.

Figure 6: The winning rates of our adversarial agent trained with different hyperparameter selection strategies and distance measures. The darker solid lines in the figures are the average winning rate of the corresponding agent. The lighter shadow represent the variation between the maximal and minimal winning rates.

aligns with the rationale behind our design. That is, the distribution of the trajectory used for agent training is very unstable, and it is generally difficult to find a constant value suitable for all possible distributions.

As is shown in Figure 6b, we also discover that, although the explanation methods in our choice set provide different fidelity [1], integrated as a component of our attack, they do not deviate the effectiveness of the attack. The adversarial agent with each of the three explanation methods demonstrates about 60% of a winning rate. This indicates the choice of explanation methods has nearly no influence upon the performance of our attack. In addition, we observe that, using a randomly generated explanation to adjust hyperparameter λ, the adversarial agent has only about 40% winning rate (see Figure 6c). From a different angle, this implies the importance of the explanation AI techniques upon our attack.

Comparison of distance measures. Figure 6d shows the performance comparison of the adversarial agents trained under different distance measures. As we can observe from the figure, the adversarial agent trained under the l_2 norm demonstrates the worst winning rate, which is even lower than that observed from the baseline method. The reason behind this observation is as follows. The observations and actions in the MuJoCo game are of high dimensionality. When minimization or maximization problems involve high dimensional input, the l_2 norm is typically not able to impose a strong penalty, and thus the model trained on such a distance measure usually exhibits poor performance.

From Figure 6d, we can also observe that the proposed method under the setup of the l_1 norm demonstrates better performance than the baseline approach as well as that under the l_2 norm. However, it is still slightly below the performance observed from our carefully selected distance measure. While this observation could be used as an argument to support the selection of our distance measure, we do not claim the l_∞ norm cannot be replaced with the l_1 norm, but argue that they can be interchangeable. This is because, the performance difference is subtle and, presumably in a different game, the adversarial agent trained under the l_1 norm might demonstrate a slightly higher winning rate.

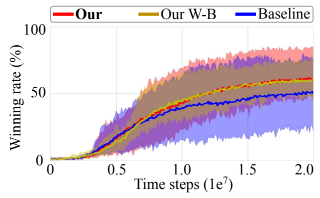
Comparison of our attack with the baseline method. Figure 7 shows the comparison of our method with the baseline approach across two different games. First, we can discover that using the baseline approach for the MuJoCo game, the winning rate of the adversarial agent converges just slightly above 50%. [8] This implies that the adversarial agent trained by the baseline method can impose only a minimal risk to its opponent. We believe the reason behind this is as follows.

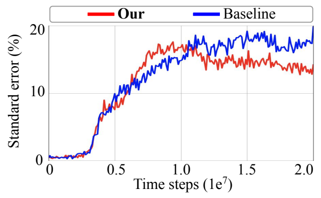
As is mentioned in the section above, the baseline is a simple application of the PPO algorithm, which is not designed specifically for training an agent to exploit the weakness of the opponent. As a result, when used to train an adversarial agent in a game, the algorithm may not be able to find a policy that could significantly pull down the winning rate of the opponent. From the perspective of the adversary, this is indicated by the increase of his agent in the winning rate.

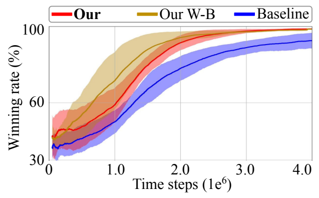
From Figure 7, we can also observe that the adversarial agent trained by our method demonstrates significant improvement. For the MuJoCo and the Pong game, our adversarial agent could converge at 60% and 100% of the winning rates, respectively. This indicates that the action deviation term could better guide our algorithm to search adversarial policy subspace, identifying the one that could exploit the weakness of the opponent most effectively.

In addition to the improvement of the average winning rates, our proposed method, to some extent, escalates the efficiency of the training process. As we can imply from Figure 7, to train an adversarial agent with a certain winning rate, our method usually takes fewer iterations than the baseline approach. Take the MuJoCo game for example. To train an adversarial agent with 50% of the winning rate, the baseline takes about 20 million iterations where our method takes only about 11 million iterations. We argue this is beneficial because reinforcement learning is known to be computationally heavy and, with the capability of reducing the training iterations, one could obtain an adversarial agent more efficiently.

[8] Note that the adversarial agent trained by this baseline approach does not demonstrate the same winning rate as is stated in [10]. We believe this is caused by the choice of initial states. Existing research [17] has shown that DRL algorithms are sensitive to the choice of initial states. As such, the standard method of evaluating a DRL algorithm is to run the algorithm multiple times with different initial states and report the statistics of the results. In this work, we follows this standard process and run each method with eight randomly selected initial states.


(a) MuJoCo winning rate.


(b) MuJoCo standard error.


(c) Roboshool winning rate.

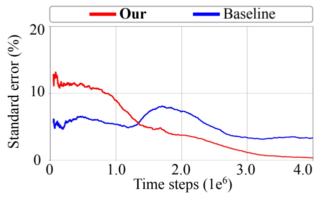
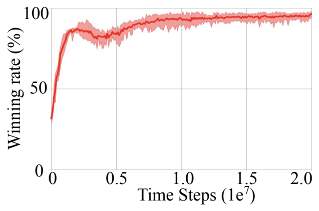

(d) Roboshool standard error.

Figure 7: Our attack vs. the baseline approach [10] in two different games. Note that "Our W-B" represents our attack in the white-box setting, where the approximated policy network of the victim agent was replaced with its actual policy network.


(a) MuJoCo.

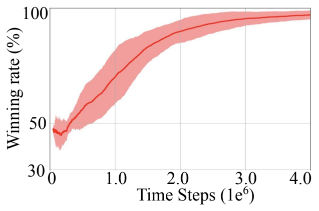

(b) Roboschool Pong.

Figure 8: The winning rate of adversary-retrained victim agent against our adversarial agent in two different games.

Game	Min	Max	Mean	Std
MuJoCo	6.0%	25.0%	16.3%	6.2%
Roboschool Pong	40.0%	44.0%	41.4%	1.4%

Table 1: The winning rate of the adversary-retrained victim agent against the corresponding regular agent in two different games. Note that after retraining the victim agents, we test them for one hundred episodes.

Finally, from Figure 7, we can observe that our method exhibits fewer variations of the winning rates (*i.e.,* less shadow area) than the baseline approach when the initial state varies. This implies our proposed method is less sensitive to the initial state of the training process. Similar to the property of training efficiency above, this is also a critical characteristic because reinforcement learning is also known to be sensitive to the initial random states, with our method, one does not need to set up a good initial state to obtain an adversarial agent with decent performance.

Comparison of black-box approximation with white-box prior. Figure 7 illustrates the performance of the adversarial agents trained with the approximated opponent policy as well as the one actually used by the opponent. As is indicated by the lines marked as "W-B" in the figure, the performance observed from the white-box setting and our approximated approach is approximately the same. This indicates that, while our point estimate inevitably introduces errors in approximation, for both games used for our evaluation, they have not yet been amplified to a level that could jeopardize the performance of our adversarial agent.

Comparison of adversary-retained agents with regular agents. Figure 8 depicts the winning rate of the victim agent against our adversarial agent, after we retrained it by using the method proposed to train the adversarial agent. As we can observe in the figure, this adversarial training approach significantly improves the robustness of the victim agent. The retained victim agent demonstrates more than 95% winning rates for both MuJoCo and Roboschool Pong games. This indicates a simple adversarial training approach could be used as an adversary-resistant method to robustify a game agent. However, in Table 1, we also note that, when using the retrained victim agent to play with a regular agent (*i.e.,* the agent trained through self-play), the robustified agent does not demonstrate a sufficient capability in beating the regular agent. This implies that, though a simple adversarial training improves agent robustness, it cannot help a victim agent obtain sufficient generalizability. We suspect this is caused by the composition of the retraining episodes. Specifically, if retraining the victim agent with the episodes of it playing with both an adversarial agent and a regular agent, the retrained agent will not only pick up adversarial robustness agent but also preserve its generalizability.

7 Related Work

There is a large body of research on adversarial attacks against deep neural networks (*e.g.,* [7,8,11,13,27,36,49]). Recently, the interest has been extended to deep reinforcement learning (*e.g.,* [18,19,40]). From the technical perspective, these previous works can be categorized into ❶ attacking reinforcement learning through trojan backdoors, ❷ attacking reinforcement learning through an adversarial environment, and ❸ attacking reinforcement learning through an adversarial agent. In the following, we summarize the existing works and highlight the key difference between these works and ours.

Trojan backdoors. A trojan backdoor attack refers to a hidden pattern implanted in a deep neural network [8, 13, 27]. When activated, it could force that infected deep neural network misclassifying the contaminated inputs into a wrong class. Recently, such an attack has been introduced to the context of deep reinforcement learning. For example, in recent works [21,56], researchers demonstrate that an attacker could follow the approach below to insert trojan backdoors into the policy networks of a trained agent.

First, the attacker injects a trigger into an environment. Then, he runs the victim agent in that manipulated environment, collecting the contaminated training trajectories. By assigning high rewards to these contaminated trajectories, the attacker could train a trojan-implanted policy network for the victim agent. As is shown in [21, 56], when a trigger is presented to the trojan-inserted agent, the agent generally exhibits undesired behaviors.

When launching the trojan backdoor attack, an adversary not only has to involve the training process of the victim agent but also obtain the control over the environment that the agent interacts with. In our work, we neither assume the involvement of the training process nor the freedom to change the environment when attacking an agent. As is mentioned in Section 2, we assume an adversary could only control the attacking agent and observe the actions of the victim agent. As such, our proposed attack is orthogonal to trojan attacks and more realistic in the physical world.

Adversarial environment. Over the past years, many research works have discovered that deep neural networks are vulnerable to adversarial attacks [7, 11, 36, 49], in which an attackers could subtly perturb a data input to a deep neural network (*e.g.,* an image) and thus force that network to misclassify the perturbed data into the wrong class. Recently, such a kind of adversarial attacks has been extended and launched against the deep reinforcement learning, or more precisely, the policy network of a trained agent.

In a pioneering research work [18], Huang *et al.* leverage the idea of adversarial learning to manipulate the environment at each time step and thus the observation passing to the policy network. They demonstrate that using this approach, the perturbed environment could easily fail a game agent – making it exhibit poor performance – regardless whether it is trained by deep Q-learning or actor-critic algorithms.

Following the step of Huang and his colleagues, recent research [23, 25, 40] designed and developed new approaches to improve the efficiency of this attack. For example, Kos *et al.* [23] suggest to perform environment manipulation only at the times steps when the output of the value function exceeds a certain threshold. Russo *et al.* [40] model the selection of attacking time steps as a Markov decision process. By solving this Markov decision process, an attacker could identify the optimal time steps to launch attacks and thus minimize his effort on environment manipulation.

Going beyond the efficiency improvement, recent research also proposes methods to launch the aforementioned attack in black-box settings. Rather than assuming an attacker has the free access to the internal weights, the training algorithms, and the training trajectories of the corresponding policy network, the black-box setting restricts an attacker's access only to the input and output of the policy network. Under this setup, Huang *et al.* [18] improves their adversarial attack. More specifically, they trained a surrogate policy network by using different training trajectory rollouts and algorithms. Then, they utilized that network to construct an adversarial environment (observations). Through a series of experiments, they showed that the adversarial environment derived from the surrogate network can still be useful for attacking the original policy network. In addition to this black-box approach, recent research proposes many other methods to generate an adversarial environment in black-box settings (*e.g.,* , [4, 54, 59]). Similar to the work proposed in [18], they also demonstrated that a trained agent could be attacked by an adversarial environment, even if an attacker does not have prior knowledge about its policy network.

Different from the works mentioned above, our attack does not craft an adversarial environment but manipulate the action of the adversarial agent through its policy network with the goal of failing the opponent agent. This is a more practical setup because, in the real world, an attacker could only control its own agent but not have the freedom to change the environment that the victim agent interacts with (*e.g.,* changing the color of the sky in the super Mario game).

Adversarial agent. In terms of the problem setup, the work most relevant to ours is the attacks through adversarial agents. Different from the attacks mentioned above, this kind of attack can be launched without the requirement of changing an environment and/or accessing the training process of victim agents. In early research, Gleave *et al.* [10] propose a training method that learns the policy network of the adversarial agent by directly playing with the opponent agent.

Technically speaking, they first treat the opponent agent as part of the observation of the adversarial agent and then simply train the adversarial agent by using the PPO algorithm. In [10], Gleave and his colleagues show that, by using their learning method to train an adversarial agent for MuJoCo game [3], an attacker could make that adversarial agent defeat the opponent agent in the two-agent competitive setting.

However, as is discussed in the section above, the method proposed in [10] demonstrates only a low success rate in attacking opponent agents because the proposed method is a simple application of the PPO algorithm, which has less guidance for the adversarial agent to identify the weakness of the opponent agent. In this work, we propose a new method to guide the construction of an adversarial policy network. Technically, it not only extends the objective function of the PPO algorithm but, more importantly, utilizes the explainable AI techniques to find the weakness of the opponent agent. As we demonstrated in Section 6, the adversarial agent trained through our method significantly outperforms that trained through the method in [10].

8 Discussion and Future Work

In this section, we discuss some related issues of our proposed method and our future plan.

Multiple agents. In this work, we develop our attack against two-agent competitive games, whose real-world applications include real-time strategy games (*e.g.,* StarCraft II and Dota 2), online board games (*e.g.,* Go, Poker), etc. In the future, we plan to extend our work to multi-agent environments, where multiple participants collaborate and/or compete with each other. To achieve this, we will explore the solutions to tackling the following challenges. First, different from our game setting or typical single-agent reinforcement learning settings, which can be modeled as a Markov Decision Process [41,53], multi-agent reinforcement learning games require re-defining the game model as either Markov game or extension form game with totally different value function and action-value function [38,57]. Under these new game settings, the PPO algorithm is no longer a standard learning method to train an agent. In this work, we design our method based on the PPO algorithm. As such, migrating our attack method to multi-agent settings might require non-trivial modification and even a completely new design. Second, even if recent research proposes adaptive methods [34] to extend the PPO algorithm into a multi-agent setting, it is still challenging to integrate our proposed attack into a multi-agent game. On the one hand, this is because recent research [57] demonstrates that a multi-agent environment introduces non-stationary status and more intense variance into the game environment, which inevitably makes the training of our adversarial agent more difficult. On the other hand, this is due to the fact that the integration of our method inevitably introduces intensive computation and increases the difficulty in tuning hyperparameters. For example, in a multi-agent environment, agents compete with each other. This indicates that we have to modify the aforementioned loss term by deviating actions between each other. Under this setup, we have to increase the number of loss terms by n^2, where n is the total number of agents in the environment. Assume n is a number larger than 5. Then, we can expect a final loss function with more than 25 loss terms, which makes the optimization of that loss function hard to be resolved and the hyperparameter tuning relatively difficult.

Defense and detection. Researchers have proposed several defense and detection mechanisms for reinforcement learning. With respect to the efforts of defense, many research works extend the idea of adversarial training [11,51]. For example, the works proposed in [5,28,37] utilize the technique proposed in [18] to generate adversarial samples and then leverage these samples to retrain deep Q-networks or the policy networks for the goal of improving their robustness. The work proposed in [6] introduces random noise to the weights of a deep Q-network during the training process. It demonstrates that the trained network can be robust against the adversarial sample attack proposed in [4].

Regarding the efforts of the detection, there have been two existing works [15,26]. They build independent neural networks to identify adversarial samples to the policy network, and demonstrate great success in pinpointing adversarial attacks against reinforcement learning. However, existing defense and detection are designed for the attack through environment manipulation. Thus, they cannot be easily adopted or extended to defeat our attack. As we show in Section 6, the victim agent robustified by adversarial training loses its generalizability, and we suspect this is caused by the trajectory split. As a part of future work, we plan to verify this hypothesis by retraining the victim agent on two sets of game episodes. One is from the victim agent's interactions with the corresponding regular agent. The other is from the victim agent's interactions with the adversarial agent learned through our proposed approach. We will also vary the percentage of the adversarial/regular episodes and observe the changes in the retrained victim agent's robustness and generalizability.

Transferability. Following the efforts of exploiting reinforcement learning through an adversarial environment, recent research has extended their interest to study the transferability of adversarial environments (*e.g.,* [18]). More specifically, for the same reinforcement learning task, researchers have shown that the adversarial environment crafted for one particular policy network can be easily transferred to a different policy network, misleading the corresponding agent to behave in an undesired manner. As part of our future work, we plan to explore the transferability of our adversarial policy. We will examine whether an adversarial policy network trained against one particular opponent agent could also be used to defeat the other agents trained differently but serving for the same reinforcement learning task.

9 Conclusion

When launching an attack against an opponent agent in a reinforcement learning problem, an adversary usually has full control over his agent (adversarial agent) as well as the freedom to passively observe the action/observation of his opponent. However, it is very common that the adversary has no access to the policy network of the opponent agent nor has the capability of manipulating the input to that network arbitrarily (i.e., observation). In this practical scenario, using existing techniques, it is usually difficult to train an adversarial agent effectively and efficiently because the algorithms applied to this problem either make strong assumptions or lack the ability to exploit the weakness of the target agent. In this work, we carefully extend a state-of-the-art reinforcement learning algorithm to guide the training of the adversarial agent in the two-agent competitive game setting. The empirical evidence demonstrates that an adversarial agent can be trained effectively and efficiently, exhibiting a stronger capability in exploiting the weakness of the opponent agent than those trained with existing techniques. With all these discoveries and analyses, we safely conclude that attacking reinforcement learning could be achieved in a practical scenario and demonstrated in an effective and efficient fashion.

Acknowledgments

We would like to thank our shepherd Lujo Bauer and the anonymous reviewers for their helpful feedback. This project was supported in part by NSF grant CNS-1718459, by ONR grant N00014-20-1-2008, by the Amazon Research Award.

References

[1] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. Sanity checks for saliency maps. In *Proc. of NeurIPS*, 2018.

[2] Martin Arjovsky, Soumith Chintala, et al. Wasserstein generative adversarial networks. In *Proc. of ICML*, 2017.

[3] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. In *Proc. of ICLR*, 2018.

[4] Vahid Behzadan and Arslan Munir. Vulnerability of deep reinforcement learning to policy induction attacks. In *Proc. of MLDM*, 2017.

[5] Vahid Behzadan and Arslan Munir. Whatever does not kill deep reinforcement learning, makes it stronger. *arXiv preprint arXiv:1712.09344*, 2017.

[6] Vahid Behzadan and Arslan Munir. Mitigation of policy manipulation attacks on deep q-networks with parameter-space noise. In *Proc. of SAFECOMP*, 2018.

[7] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Proc. of S&P*, 2017.

[8] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[9] Piotr Dabkowski and Yarin Gal. Real time image saliency for black box classifiers. In *Proc. of NeurIPS*, 2017.

[10] Adam Gleave, Michael Dennis, Neel Kant, Cody Wild, et al. Adversarial policies: Attacking deep reinforcement learning. In *Proc. of ICLR*, 2020.

[11] Ian J Goodfellow, Jonathon Shlens, et al. Explaining and harnessing adversarial examples. In *Proc. of ICLR*, 2015.

[12] Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 2004.

[13] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. In *Proc. of NeurIPS Workshop*, 2017.

[14] Arthur Guez, David Silver, and Peter Dayan. Efficient bayes-adaptive reinforcement learning using sample-based search. In *Proc. of NeurIPS*, 2012.

[15] Aaron Havens, Zhanhong Jiang, and Soumik Sarkar. Online robust policy learning in the presence of unknown adversaries. In *Proc. of NeurIPS*, 2018.

[16] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, SM Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. In *Proc. of NeurIPS*, 2017.

[17] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proc. of AAAI*, 2018.

[18] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. In *Proc. of ICLR workshop*, 2017.

[19] Yonghong Huang and Shih-han Wang. Adversarial manipulation of reinforcement learning policies in autonomous agents. In *Proc. of IJCNN*, 2018.

[20] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint:1412.6980*, 2014.

[21] Panagiota Kiourti, Kacper Wardega, Susmit Jha, and Wenchao Li. Trojdrl: Trojan attacks on deep reinforcement learning agents. *arXiv preprint arXiv:1903.06638*, 2019.

[22] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Proc. of NeurIPS*, 2000.

[23] Jernej Kos and Dawn Song. Delving into adversarial attacks on deep policies. In *Proc. of ICLR Workshop*, 2017.

[24] Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1997.

[25] Yen-Chen Lin, Zhang-Wei Hong, Yuan-Hong Liao, Meng-Li Shih, et al. Tactics of adversarial attack on deep reinforcement learning agents. In *Proc. of IJCAI*, 2017.

[26] Yen-Chen Lin, Ming-Yu Liu, Min Sun, and Jia-Bin Huang. Detecting adversarial attacks on neural network policies with visual foresight. *arXiv preprint arXiv:1710.00814*, 2017.

[27] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *Proc. of NDSS*, 2018.

[28] Ajay Mandlekar, Yuke Zhu, Animesh Garg, et al. Adversarially robust policy learning: Active construction of physically-plausible perturbations. In *Proc. of IROS*, 2017.

[29] 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 *Proc. of ICML*, 2016.

[30] Volodymyr Mnih, Koray Kavukcuoglu, et al. Playing atari with deep reinforcement learning. In *Proc. of NeurIPS Deep Learning Workshop*, 2013.

[31] Volodymyr Mnih, Koray Kavukcuoglu, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[32] OpenAI. Openai at the international 2017. `https://openai.com/the-international/`, 2017.

[33] OpenAI. Roboschool: open-source software for robot simulation. `https://openai.com/blog/roboschool/`, 2017.

[34] OpenAI. Openai five. `https://openai.com/blog/openai-five/`, 2018.

[35] OpenAI. Emergent tool use from multi-agent interaction. `https://openai.com/blog/emergent-tool-use/`, 2019.

[36] Nicolas Papernot, Patrick McDaniel, Somesh Jha, et al. The limitations of deep learning in adversarial settings. In *Proc. of Euro S&P*, 2016.

[37] Anay Pattanaik, Zhenyi Tang, Shuijing Liu, Gautham Bommannan, and Girish Chowdhary. Robust deep reinforcement learning with adversarial attacks. In *Proc. of AAMAS*, 2018.

[38] Tabish Rashid, Mikayel Samvelyan, et al. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *Proc. of ICML*, 2018.

[39] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proc. of KDD*, 2016.

[40] Alessio Russo and Alexandre Proutiere. Optimal attacks on reinforcement learning policies. *arXiv preprint arXiv:1907.13548*, 2019.

[41] John Schulman, Sergey Levine, et al. Trust region policy optimization. In *Proc. of ICML*, 2015.

[42] John Schulman, Philipp Moritz, et al. High-dimensional continuous control using generalized advantage estimation. In *Proc. of ICLR*, 2016.

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

[44] Lloyd S Shapley. Stochastic games. *Proc. of the national academy of sciences*, 1953.

[45] David Silver, Aja Huang, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 2016.

[46] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Proc. of ICLR*, 2013.

[47] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. Smoothgrad: removing noise by adding noise. *arXiv preprint arXiv:1706.03825*, 2017.

[48] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proc. of ICML*, 2017.

[49] Christian Szegedy, Wojciech Zaremba, et al. Intriguing properties of neural networks. In *Proc. of ICLR*, 2015.

[50] Emanuel Todorov, Tom Erez, et al. Mujoco: A physics engine for model-based control. In *Proc. of ICIRS*, 2012.

[51] Florian Tramèr, Alexey Kurakin, et al. Ensemble adversarial training: Attacks and defenses. In *Proc. of ICLR*, 2018.

[52] Florian Tramèr, Fan Zhang, et al. Stealing machine learning models via prediction apis. In *Proc. of USENIX Security Symposium*, 2016.

[53] Hado Van Hasselt, Arthur Guez, and other. Deep reinforcement learning with double q-learning. In *Proc. of AAAI*, 2016.

[54] Chaowei Xiao, Xinlei Pan, et al. Characterizing attacks on deep reinforcement learning. *arXiv:1907.09470*, 2019.

[55] Tianbing Xu, Qiang Liu, Liang Zhao, and Jian Peng. Learning to explore via meta-policy gradient. In *Proc. of ICML*, 2018.

[56] Zhaoyuan Yang, Naresh Iyer, Johan Reimann, and Nurali Virani. Design of intentional backdoors in sequential models. *arXiv preprint arXiv:1902.09972*, 2019.

[57] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *arXiv preprint arXiv:1911.10635*, 2019.

[58] Marvin Zhang, Zoe McCarthy, Chelsea Finn, Sergey Levine, and Pieter Abbeel. Learning deep neural network policies with continuous memory states. In *Proc. of ICRA*, 2016.

[59] Yiren Zhao, Ilia Shumailov, et al. Blackbox attacks on reinforcement learning agents using approximated temporal information. *arXiv preprint arXiv:1909.02918*, 2019.

Appendix

Victim policies. The network architecture of the victim policy in the MuJuCo game and the roboschool Pong game are: MLP-380-128-128-17 [3] and MLP-13-64-64-2, respectively.

Hyper-parameters of the baseline. The baseline has two sets of hyper-parameters: the adversarial policy/value network architecture, and the hyperparameters of the PPO algorithm. For the MuJoCo game, we directly used the default choices in [10]. For the roboschool Pong game, we set the adversarial policy network and its value function as MLP-13-64-64-2 and MLP-13-64-64-1, and use the same set of PPO hyper-parameters with the MoJuCo game.

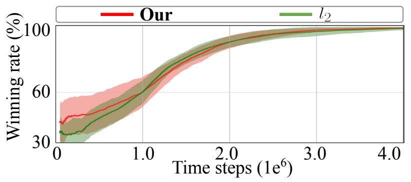
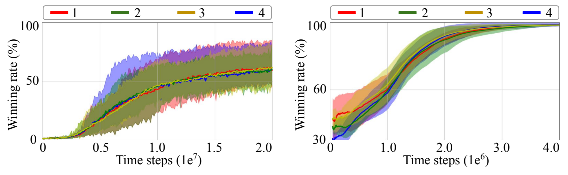


Figure 9: Comparison of our attack and the attack with l_2.

Hyper-parameters of our method. Here, we specify the hyper-parameters that are not varied in the sensitivity test. First, we applied the choices of [10] for those inherent from [10] (*i.e.,* policy/value network architectures and the PPO hyper-parameters). In addition, our attack has four hyper-parameters: H, F, ϵ_s, and ϵ_a. We set ϵ_s/ϵ_a as widely used empirical values [7] and H/F similar to the policy network architectures. Specifically, for the MuJoCo game, we set $\epsilon_s = 1$, $\epsilon_a = 0.05$, H: MLP-414-40-64-380, and F: MLP-380-64-64-17. For the roboschool Pong game, we set $\epsilon_s = 0.01$, $\epsilon_a = 0.05$, H: MLP-17-40-16-13, and F: MLP-13-64-64-2.

Effectiveness of l_2 norm on the Pong game. In Figure 6d, we show the solution developed on l_2 norm is worse than those developed on l_1, l_∞, and our baseline. We argue that this is because l_2 norm is not suitable for high-dimensional input. In order to validate this, we run the similar experiment on Robotschool Pong game. Different from the MuJoCo game, here, the agent takes a low-dimensional input (13 features). In Figure 9, we depict that, for the Pong game, the solution developed on the l_2 norm is just as good as our final solution which utilizes l_1. This well confirms our argument. That is, the l_2 norm is not suitable for a situation where the input high dimensionality inputs, and l_1 or l_∞ is a better fit.



(a) MuJoCo. (b) Roboschool Pong.

Figure 10: The performance of our attack with different η.

Additional parameter sensitivity test. In our experiments, we set equal weight to the action difference term and the observation difference term in Eqn. (9). Here, we vary the relative weight between two terms and observe its influence upon our attack performance. Specifically, we introduce a weight η to the observation different term (*i.e.,* $-\eta\|\hat{o}_v^{(t+1)} - o_v^{(t+1)}\|_1 + \|\hat{a}_v^{(t+1)} - a_v^{(t+1)}\|_1$) and train adversarial agent with $\eta = [1,2,3,4]$. Figure 10 shows the winning rate of the adversarial agent on two selected games. The results show that subtly varying η imposes only a negligible influence upon the performance of the adversarial agents trained by our attack.