Real-time Adversarial Perturbations against Deep Reinforcement Learning Policies: Attacks and Defenses

Buse G. A. Tekgul
Aalto University
Espoo, Finland
buse.atlitekgul@aalto.fi

Samuel Marchal
WithSecure Corporation & Aalto University
Espoo, Finland
samuel.marchal@aalto.fi

Shelly Wang
University of Waterloo
Waterloo, Canada
shelly.wang@uwaterloo.ca

N. Asokan
University of Waterloo & Aalto University
Waterloo, Canada
asokan@acm.org

ABSTRACT
Recent work has shown that deep reinforcement learning (DRL) policies are vulnerable to adversarial perturbations. Adversaries can mislead policies of DRL agents by perturbing the state of the environment observed by the agents. Existing attacks are feasible in principle but face challenges in practice, either by being too slow to fool DRL policies in real time or by modifying past observations stored in the agent’s memory. We show that using the Universal Adversarial Perturbation (UAP) method to compute perturbations, independent of the individual inputs to which they are applied, can fool DRL policies effectively and in real time. We describe three such attack variants. Via an extensive evaluation using three Atari 2600 games, we show that our attacks are effective, as they fully degrade the performance of three different DRL agents (up to 100%, even when the \( l_\infty \) bound on the perturbation is as small as 0.01). It is faster compared to the response time (0.6ms on average) of different DRL policies, and considerably faster than prior attacks using adversarial perturbations (1.8ms on average). We also show that our attack technique is efficient, incurring an online computational cost of 0.027ms on average. Using two further tasks involving robotic movement, we confirm that our results generalize to more complex DRL tasks. Furthermore, we demonstrate that the effectiveness of known defenses diminishes against universal perturbations. We propose an effective technique that detects all known adversarial perturbations against DRL policies, including all the universal perturbations presented in this paper.

KEYWORDS
deep reinforcement learning, adversarial examples, neural networks

1 INTRODUCTION
Machine learning models are vulnerable to adversarial examples: maliciously crafted inputs generated by adding small perturbations to the original input in order to force a model into generating wrong predictions [9, 31]. Prior work [11, 14, 16, 37] has shown that adversarial examples can also fool deep reinforcement learning (DRL) agents that use deep neural networks (DNNs) to approximate their policy (the decision-making strategy). If this vulnerability is exploited in safety-critical DRL applications such as robotic surgery and autonomous driving, the impact can be disastrous.

A DRL agent can partially or fully observe the state of the environment by capturing complex, high-dimensional observations. For example, a DRL agent playing an Atari 2600 game observes pixels from each image frame of the game to construct states by combining a number of observations. DRL agents use the current state as an input to their policy which outputs a suitable action for that state. Consequently, adversaries can modify the environment to mislead the agent’s sequential decision-making process. Various state-of-the-art attack methods assume white-box knowledge, where adversaries have access to the parameters of the agent’s policy model and the reinforcement learning algorithm. In untargeted attacks, the adversary’s goal is to fool the agent’s policy so that the agent becomes useless. The agent is useless when it 1) cannot complete its task or 2) finishes its task with unacceptably poor performance. Prior work has shown that adversaries with white-box knowledge can successfully destroy an agent’s performance using one-step gradient-based approaches [9], optimization-based methods [6], or adversarial saliency maps [25]. Previous work has also proposed different attack strategies where the adversary generates the perturbation for 1) each state [2, 11], 2) critical states where the agent prefers one action with high confidence [14, 16, 29], or 3) periodically, at every \( N^{th} \) state [14].

Although prior white-box attacks using adversarial perturbations are effective in principle, they are not realistic in practice. First, some attack strategies [11, 16] are based on computing the perturbation using back-propagation or solving an optimization problem. This is computationally expensive even if it is done for every \( N^{th} \) state. DRL agents need to respond to new states very quickly in order to carry out the task effectively and on-the-fly. Therefore, attacks that take longer than the average time between two consecutive observations are too slow to be realized in real-time. Second, in realistic scenarios, the adversary cannot have full control over the environment. However, iterative attacks [16, 37] require querying agents with multiple perturbed versions of the current state and resetting the environment in order to find the optimal perturbation. Therefore, iterative attacks cannot be applied in real-life scenarios such as autonomous agents interacting with a dynamic environment. Finally, the aforementioned state-the-art attacks require seeing all observations to generate and apply perturbations to the state containing multiple observations. However, the agent can store clean observations that are part of the current state in its memory before the adversary can generate perturbations that...
need to be applied to all of those observations. In order to overcome these challenges, Xiao et al. [37] proposed a new attack strategy by producing a single, universal perturbation during the task. This method generates perturbations that are independent of the input state. However, it must freeze the environment to have time to generate the perturbation online for each start (in other words, each start requires a different universal perturbation).

Contributions. We propose an effective, real-time attack strategy to fool DRL policies by computing a state-agnostic, universal perturbations offline. Once this perturbation is generated, it can be added into any state to force the victim agent to choose sub-optimal actions (Figure 1). Similarly to previous work [37], we focus on untargeted attacks in a white-box setting. Our contributions are:

(1) We design two new real-time white-box attacks, UAP-S and UAP-O, using Universal Adversarial Perturbation (UAP) [21] to generate state-agnostic adversarial examples. We also design a third real-time attack, OSFW(U), by extending Xiao et al.’s [37] attack so that it generates a universal perturbation once, applicable to any subsequent episode (Section 3). An empirical evaluation of these three attacks using three different reinforcement learning algorithms playing three different Atari 2600 games (Breakout, Freeway, and Pong) demonstrates that our attacks are comparable to prior work in their effectiveness (100% drop in return), while being significantly faster (0.027 ms on average, compared to 1.8 ms) and less visible than prior adversarial perturbations [11, 37] (Section 4.2). Using two additional tasks involving the MuJoCo robotics simulator [33], which requires continuous control, we show that our results generalize to more complex DRL agents (Section 4.3).

(2) We demonstrate the limitations of prior defenses. We show that agents trained with the state-of-the-art robust policy regularization technique [39] exhibit reduced effectiveness against adversarial perturbations at higher perturbation bounds (≥ 0.05). In some tasks (Pong), universal adversarial perturbations completely destroy the agent performance (Table 2). Visual Foresight [17], which can completely restore an agent’s performance in the presence of prior adversarial perturbations [11], fails to do so when faced with universal perturbations (Section 5.2).

(3) We propose an efficient method, AD3, to detect adversarial perturbations. AD3 can be combined with other defenses to provide stronger resistance for DRL agents against untargeted adversarial perturbations (Section 5.3).

2 BACKGROUND AND RELATED WORK

2.1 Deep Reinforcement Learning

2.1.1 Reinforcement learning. Reinforcement learning involves settings where an agent continuously interacts with a non-stationary environment to decide which action to take in response to a given state. At time step t, the environment is characterized by its state $s \in S$ consisting of N past observations $o$ pre-processed by some function $f_{pre}$, i.e., $s = \{f_{pre}(o_1), \cdots, f_{pre}(o_t)\}$. At each state, the agent takes an action $a \in A$ which moves the environment to the next $s' \in S$ and yields a reward $r$ from the environment. The agent uses this information to optimize its policy $\pi$, a probability distribution that maps states into actions [30]. During training, the agent improves the estimate of a value function $V(s)$ or an action-value function $Q(s, a)$. $V(s)$ measures how valuable it is to be in a state $s$ by calculating the expected discounted return: the discounted cumulative sum of future rewards while following a specific $\pi$. Similarly, $Q(s, a)$ estimates the value of taking action $a$ in the current state $s$ when following $\pi$. During evaluation, the optimized $\pi$ is used for decision making, and the performance of the agent is measured by the return. In this work, we focus on episodic [4] and finite-horizon [32] tasks. In episodic tasks such as Atari games, each episode ends with a terminal state (e.g., winning/losing a game, arriving at goal state). In this case, the return for one episode is computed by the total score in single-player games (e.g., Breakout, Freeway) or the relative score when it is played against a computer (e.g., Pong). Finite-horizon tasks include continuous control, where (e.g., Humanoid, Hopper) the return is measured for a fixed length of the episode.

2.1.2 DNN. DNNs are parameterized functions $f(x, \theta)$ consisting of neural network layers. For an input $x \in \mathbb{R}^n$ with $n$ features, the parameter vector $\theta$ is optimized by training the DNN over a labeled training set. $f(x, \theta)$ outputs a vector $y \in \mathbb{R}^m$ with $m$ different...
classes. In classification problems, one can find the predicted class as \( \hat{f}(x) = \arg\max_m f(x, \theta) \). For simplicity, we will use \( f(x) \) to denote \( f(x, \theta) \).

2.1.3 DRL. DNNs are useful for approximating \( \pi \) when \( S \) or \( A \) is too large, or largely unexplored. Deep Q Networks (DQN) is one of the well-known value-based DRL algorithms [20] that uses DNNs to approximate \( Q(s, a) \). During training, the DQN aims to find the optimal \( Q(s, a) \) by updating the estimate of \( Q(s, a) \) repeatedly with actions that maximize \( Q(s, a) \) in each step. Therefore, the optimal \( \pi \) is implicitly defined using the optimal \( Q(s, a) \). Despite its effectiveness, DQN cannot be used in continuous control tasks, where \( a \) is a real-valued vector sampled from a range, instead of a finite set. Continuous control tasks require different policy-based DRL algorithms [19, 28]. They use two different DNNs that usually share a number of lower layers, to approximate both \( \pi \) and \( V(s) \) (or \( Q(s, a) \) depending on the algorithm), and update \( \pi \) directly. For example, in actor-critic methods (A2C) [19], the critic estimates \( V(s) \) or \( Q(s, a) \) for the current \( \pi \) and the actor updates the parameter vector \( \theta \) of \( \pi \) by using the advantage. Advantage refers to the critic’s evaluation of the action decision using the estimated \( V(s) \) or \( Q(s, a) \), while the actor is following the current \( \pi \). Proximal Policy Optimization (PPO) [28] is an on-policy variant of A2C. PPO updates the current \( \pi \) by ensuring that the updated \( \pi \) is close to the old one.

2.2 Adversarial Examples

An adversarial example \( x' \) against a classification model \( f(x) \) is a deliberately modified version of an input \( x \) such that \( x \) and \( x' \) are similar, but \( x' \) is misclassified by \( f \), i.e., \( \hat{f}(x) \neq \hat{f}(x') \). An untargeted adversarial example is found by solving

\[
\operatorname{argmax} \ell(f(x'), \hat{f}(x))
\]

\[
s.t.: \|x' - x\|_p = \|r\|_p \leq \varepsilon,
\]

where \( \varepsilon \) is the \( l_p \) norm bound and \( \ell \) is the loss between \( f(x') \) and the predicted label \( \hat{f}(x) \). In this work, we assume \( l_\infty \) norm bound (i.e., any element in the perturbation must not exceed a specified threshold \( \varepsilon \)) as in the state-of-the-art Fast Gradient Sign Method (FGSM) [9]. FGSM calculates \( r \) by

\[
r = \varepsilon \cdot \text{sign}(\nabla_x \ell(f(x), \hat{f}(x))).
\]

Usually, adversarial examples are computed for each \( x \). An alternative is to generate input-agnostic universal perturbations. For instance, Moosavi et al. propose Universal Adversarial Perturbation (UAP) [21] that searches for a small \( r \) that can be added to arbitrary inputs to yield adversarial examples against \( f \). UAP iteratively computes a unique \( r \) that fools the classifier for almost all inputs \( x \) belonging to a training set \( D_{\text{train}} \). UAP utilizes DeepFool [22] to update \( r \) at each iteration. In addition to the \( l_\infty \) constraint, UAP should achieve a desired fooling rate \( \delta \); the proportion of successful adversarial examples against \( f \) with respect to the total number of perturbed samples \( |D_{\text{train}}| \) in \( D_{\text{train}} \).

Following the first work [21] introducing UAP, many different strategies [7, 10, 23, 24] have been proposed to generate universal adversarial perturbations to fool image classifiers. Hayes et al. [10] and Mopuri et al. [24] use generative models to compute universal adversarial perturbations. Mopuri et al. [23] introduce the Fast Feature Fool algorithm, which does not require a training set to fool the classifier, but optimize a random noise to change activations of the DNN in different layers. Co et al. [7] design black-box, untargeted universal adversarial perturbations using procedural noise functions.

2.3 Adversarial Examples against DRL Policies

In discrete action spaces, where agents have finite actions, adversarial examples against a DRL policy is found by modifying Equation 1: \( x \) is changed into \( s \) at time \( t \), and \( f \) is replaced with \( Q(s, a) \). Given \( s \) and \( Q(s, a) \), \( Q(s) \) refers to the action decision. We also denote \( Q(s, a_m) \) as the state-action value of \( m \)th action at \( s \). In this setup, adversarial examples are computed to decrease \( Q(s, a) \) for the optimal action at \( s \), resulting in a sub-optimal decision.

Since Huang et al. [11] showed the vulnerability of DRL policies against adversarial perturbations, several untargeted attack methods [2, 14, 16, 29, 37] have been proposed for manipulating the environment. Recent work [1, 6, 12, 16, 35] has also developed targeted attacks, where the adversary’s goal is to lure the victim policy into a specific state or to force the victim policy into following a specific path. Most of these methods implement well-known adversarial example generation methods such as FGSM [11, 14], JSMA [2] and Carlini&Wagner [16]. Therefore, even though they effectively decrease the return of the agent, these methods cannot be implemented in real-time and have a temporal dependency: they need to compute a different \( r \) for every \( s \). Similarly to our work and Xiao et al. [37], Hussenot et al. [12] designed targeted attacks that use different universal perturbations for each action to force the victim policy into following the adversary’s policy. The universal perturbation from Xiao et al. (“obs-seq-fgsm-wb”, OSFW) is computed by applying FGSM to averaged gradients over \( k \) states to compute a single \( r \), and adding it to the remaining states in the current episode. OSFW has limitations such as the need to compute \( r \) for every new episode. Moreover, it has to freeze the task to calculate \( r \) using the victim agent, and its performance depends on the particular agent-environment interaction for each episode.

In addition to white-box attacks, multiple black-box attack methods based on finite-difference methods [37], and proxy methods approximating the victim policy [13, 41] were proposed, but they cannot be mounted in real-time, since they require querying the agent multiple times.

Recent work [8, 36] shows that adversaries can fool DRL policies in multi-agent, competitive games by training an adversarial policy for the opponent agent that exploits vulnerabilities of the victim agent. These attacks rely on creating natural observations with adversarial effects, instead of manipulating the environment by adding adversarial perturbations. In this paper, we focus on single-player games, where adversaries can only modify the environment in order to fool DRL policies.

3 STATE- AND OBSERVATION-AGNOSTIC PERTURBATIONS

We first define our adversary model and then formally present two new attacks—state-agnostic UAP (UAP-S) and observation-agnostic UAP (UAP-O)—and an improved variant, OSFW(U), of a prior attack.
3.1 Adversary Model

The goal of the adversary $Adv$ is to degrade the performance of the victim DRL agent $v$ by adding perturbations $r$ to state $s$ observed by $v$. $Adv$ is successful if the attack:

1. is effective, i.e., limits $v$ to a low return,
2. is efficient, i.e., can be realized in real-time, and
3. evades known detection mechanisms.

$Adv$ has a white-box access to $v$; therefore, it knows $v$’s action value function $Q_v$, or the policy $\pi_v$ and the value function $V_v$, depending on the DRL algorithm used by $v$. However, $Adv$ is constrained to using $r$ with a small norm to evade possible detection, either by specific anomaly detection mechanisms or via human observation. In addition, we assume that $Adv$ cannot reset the environment or return to an earlier state. In other words, we rule out trivial attacks (e.g., swapping one video frame with another or changing observations with random noise) as ineffective because they can be easily detected.

3.2 Attack Design

3.2.1 Training data collection. $Adv$ collects a training set $D_{train}$ by monitoring $v$’s interaction with the environment for one episode, and saving each $s$ into $D_{train}$. Simultaneously, $Adv$ clones $Q_v$ or $V_v$ into a proxy agent, $adv$. Specifically, in value-based methods (e.g., DQN), $Adv$ copies weights of $Q_v$ into $Q_{adv}$. In policy based methods, (e.g., A2C, PPO), $Adv$ copies weights of the critic network into $V_{adv}$. In the latter case, $Adv$ can obtain $Q_{adv}(s,a)$ by calculating $V_{adv}(s)$ for each discrete action $a \in \mathcal{A}$.

3.2.2 Data sanitization. $Adv$ sanitizes $D_{train}$ by choosing only the critical states. Following [16], we define critical states as those that can have a significant influence on the course of the game. We identify critical states using the relative action preference function

$$
\text{Var}_{a \in \mathcal{A}} [\text{Softmax}(Q_{adv}(s,a))] \geq \beta
$$

modified from [16], where $\text{Var}$ is the variance of normalized $Q_{adv}(s,a)$ values computed for $a \in \mathcal{A}$. This ensures that both UAP-S and UAP-O are optimized to fool $Q_v$ in critical states, and achieves the first attack criterion.

3.2.3 Computation of perturbation. For both UAP-S and UAP-O, we assume that $s \in D_{train}$ and $D_{train} \subset \mathcal{S}$. $Adv$ searches for an optimal $r$ that satisfies the constraints in Equation 1 while achieving a high fooling rate $\delta$ on $D_{train}$. For implementing UAP-S and UAP-O, we modify the Universal Adversarial Perturbation method in [21] (see Section 2.2) that uses DeepFool [22] to solve Equation 1. The goal of both UAP-S and UAP-O is to find a sufficiently small $r$ such that $\hat{Q}(s+r) \neq \hat{Q}(s)$, leading $v$ to choose sub-optimal actions. Algorithm 1 summarizes the method to generate UAP-S and UAP-O.

In lines 5-6 of Algorithm 1, DeepFool computes the additional $\Delta r$ by iteratively updating the perturbed $s^*_i = s+r+\Delta r_i$ until $Q_{adv}$ outputs a wrong action (see Algorithm 2 in [22]). At each iteration $i$, DeepFool finds the closest hyperplane $\tilde{l}(s^*_i)$ and $\Delta r_i$ that projects $s^*_i$ on the hyperplane. It re-computes $\Delta r_i$ as

$$
\hat{Q}'(s^*_i,a^*_i) - Q_{adv}(s^*_i,a^*_i) = \hat{Q}'(s^*_i,a^*_i) - Q_{adv}(s^*_i,a^*_i), \\
\tilde{w}_i = \nabla_{Q_{adv}(s^*_i,a^*_i)} \hat{Q}'(s^*_i,a^*_i), \\
\Delta r_i = \frac{|\hat{Q}'(s^*_i,a^*_i)|}{\|\tilde{w}_i\|^2}.
$$

where $\nabla$ is the gradient of $Q_{adv}$ w.r.t $s_i$ and $Q_{adv}(s^*_i,a^*_i)$ is the value of the $m$-th action that was chosen for the state $s+r$.

In UAP-S, there is a different perturbation for each observation $s \in \mathcal{S}$, i.e., $r = \{r_1,N-1, \cdots, r_j \}$, $r_j \neq r_k$, $\forall j,k \in \{t-N+1, \cdots, t\}$, $j \neq k$. In contrast, for UAP-O, the same perturbation is applied to all observations in $s$. Therefore, it can be considered as an observation-agnostic, completely universal attack. UAP-O aims to find a modified version $\tilde{r}$ of $r$ by solving

$$
\text{min}(\|r - \tilde{r}\|^2_2)
$$

s.t.: $\tilde{r}_j = r_k, \forall j,k \in \{t-N+1, \cdots, t\}$ and $\|\tilde{r}\|_{\infty} \leq \epsilon$.

To compute UAP-O, we modify lines 5 and 6 of Algorithm 1 that finds $\Delta r_i$. The closest $\hat{\Delta}_r_i$ to $\Delta r_i$ satisfying the conditions in Equation 5 is found by averaging $\tilde{w}_i$ over observations:

$$
\hat{\Delta}r_{ij} \leftarrow \frac{\hat{Q}(s^*_i,a^*_i)}{N\|w_j\|^2} \sum_{k=(t-N+1)}^{t} \tilde{w}_j, \forall j \in \{t-N+1, \cdots, t\}.
$$

In UAP-O, DeepFool returns $\hat{\Delta}r_1 = \hat{\Delta}r_t$ as the optimal, additional perturbation. UAP-O adds the same perturbation $\hat{r}_t$ to every observation $a_j$ in $s$. If the input state consists of only one observation, then UAP-S will simply reduce to UAP-O. The proof for Equation 6 can be found in Appendix A.

3.2.4 Extending OSFW to OSFW($U$). As explained in Section 2.3, OSFW calculates $r$ by averaging gradients of $Q_v$ using first $k$ states in an episode, and then adds $r$ into remaining states. This requires 1) generating a different $r$ for each episode, and 2) suspending the task (e.g., freezing or delaying the environment) and $v$ to perform backward propagation. Moreover, the effectiveness of OSFW varies when $v$ behaves differently in individual episodes. We extend OSFW to a completely universal adversarial perturbation by using the
We compared the effectiveness of our attacks (UAP-S, UAP-O and Arcade Learning Environment [4]). We further extended our experimental setup with the MuJoCo robotics simulator [33] and using three Atari 2600 games (Pong, Breakout, Freeway) in the DQN [20], policy-based PPO [28] and actor-critic method A2C [19].

In continuous control tasks, the optimal action is a real-valued vector that is selected from a range. These tasks have complex environments involving physical systems control such as Humanoid robots with multi-joint dynamics [33]. In continuous control, unlike discrete tasks, there is no available \( Q(s, a) \) for generating perturbations and decreasing the value of the optimal action. Nevertheless, \( \text{Adv} \) can find a perturbed state \( s + r \) having the worst value so that the action chosen by the policy of \( v \) and \( \pi_v \) will be a bad decision [39]. In continuous control, OSFW and OSFW(U) can be simply modified by changing \( Q_{\text{adv}} \) with \( V_{\text{adv}} \). However, in Algorithm 1, lines from 4 and 6 needs to be adjusted to handle these tasks. \( \text{Adv} \) can only use the copied network parameters of \( V_{\text{adv}} \) to find \( r \). The corrected computation for UAP-S and UAP-O is given in Algorithm 2. In each iteration \( i \), \( \Delta r \) is found by solving the optimization problem (lines 5 and 6).

### 3.3 Attacks in Continuous Control Settings

In continuous control tasks, the optimal action is a real-valued array that is selected from a range. These tasks have complex environments involving physical systems control such as Humanoid robots with multi-joint dynamics [33]. In continuous control, unlike discrete tasks, there is no available \( Q(s, a) \) for generating perturbations and decreasing the value of the optimal action. Nevertheless, \( \text{Adv} \) can find a perturbed state \( s + r \) having the worst value so that the action chosen by the policy of \( v \) and \( \pi_v \) will be a bad decision [39]. In continuous control, OSFW and OSFW(U) can be simply modified by changing \( Q_{\text{adv}} \) with \( V_{\text{adv}} \). However, in Algorithm 1, lines from 4 and 6 needs to be adjusted to handle these tasks. \( \text{Adv} \) can only use the copied network parameters of \( V_{\text{adv}} \) to find \( r \). The corrected computation for UAP-S and UAP-O is given in Algorithm 2. In each iteration \( i \), \( \Delta r \) is found by solving the optimization problem (lines 5 and 6).

### 4 ATTACK EVALUATION

We compared the effectiveness of our attacks (UAP-S, UAP-O and OSFW(U)) with prior attacks (FGSM and OSFW) on discrete tasks using three Atari 2600 games (Pong, Breakout, Freeway) in the Arcade Learning Environment [4]. We further extended our experimental setup with the MuJoCo robotics simulator [33] and compared these attacks in continuous control tasks.

#### 4.1 Experimental Setup

To provide an extensive evaluation, for every Atari game, we trained three agents each using a different DRL algorithm: value-based DQN [20], policy-based PPO [28] and actor-critic method A2C [19]. We used the same DNN architecture proposed in [20] for approximating \( Q_v \) in DQN and \( V_v \) in other algorithms. To facilitate comparison, we used the same setup to implement all DRL policies as well as the different attacks: PyTorch (version 1.2.0), NumPy (version 1.18.1), Gym (a toolkit for developing reinforcement learning algorithms, version 0.15.7) and MuJoCo 1.5 libraries. All experiments were done on a computer with 2x12 core Intel(R) Xeon(R) CPUs (32GB RAM) and NVIDIA Quadro P5000 with 16GB memory.

Our implementations of DQN, PPO and A2C are based on OpenAI baselines, and our implementations achieve similar returns as in OpenAI Baselines. For pre-processing, we converted each RGB frame to grayscale, re-sized those from 210 x 160 to 84 x 84, and normalized the pixel range from \([0, 255]\) to \([0, 1]\). We used the frame-skipping technique, where the victim constructs at every \( N^{th} \) observation, then selects an action and repeats it until the next state \( s' \). We set \( N = 4 \) so the input state size is \( 4 \times 84 \times 84 \). The frame rate of each game is 60 Hz by default [4]; thus, the time interval between two consecutive frames is \( 1/60 = 0.017 \) seconds.

We implemented UAP-S, UAP-O and OSFW(U) by setting the desired \( \delta_{\text{max}} \) to 95%, so that both UAP-S and UAP-O stop searching for another \( r \) when \( \delta \geq 95\% \). As baselines, we used random noise addition, FGSM [11], and OSFW [37]. We chose FGSM as a baseline, since it is the fastest adversarial perturbation generation method from the previous work and it is effective in degrading DRL agents’ performance. We measured attack effectiveness when \( e \) is between 0 and 0.01. We reported the average return over 10 episodes and run each episode with a different seed during training and evaluation.

#### 4.2 Attack Performance

##### 4.2.1 Performance degradation

Figure 2 compares UAP-S, UAP-O, and OSFW(U) with two baseline attacks and random noise addition. As can be seen from the figure, random noise addition cannot cause a significant drop in \( v \)’s performance, and FGSM is the most effective attack reducing the return up to 100% even with a very small \( e \) value. UAP-S is the second most effective attack in almost every setup, reducing the return by more than 50% in all experiments when \( e \geq 0.004 \). Additionally, all adversarial perturbations completely destroy agents’ performance at \( e = 0.01 \), except the PPO agent playing Freeway. The effectiveness of UAP-O, OSFW, and OSFW(U) are comparable across all setups. We also observe that the effectiveness of OSFW fluctuates heavily (A2C agent playing Breakout) or has high variance (PPO agent playing Freeway). This phenomenon is the result of \( v \)’s different behaviours in individual episodes (A2C agent playing Breakout), and OSFW’s inability to collect enough knowledge (PPO agent playing Freeway) in order to generalize \( r \) to the rest of the episode.

##### 4.2.2 Timing comparison

Table 1 presents the computational cost for generating \( r \) (at \( e = 0.01 \)), and the upper bound on the online computational cost to mount the attack in real-time. This upper bound is measured as \( T_{\text{max}} = 1/(\text{frame rate}) \cdot (\text{agent’s response time}) \), where agent response time is the time spent for feeding a forward through \( \pi_v \) or \( Q_v \), and executing the corresponding action. If the online cost of injecting the perturbation during deployment and (also the online cost of generating the perturbation during deployment in

---

[^3]: https://github.com/DLR-RM/rl-baselines3-zoo
the case of FGSM and OSFW) is higher than $T_{max}$, then Adv needs to stop or delay the environment, which is infeasible in practice. Table 1 confirms that the online cost of OSFW is higher than all other attacks and $T_{max}$ due to its online perturbation generation approach. Thus, OSFW has to stop the environment to inject the perturbation to the current state, or has to wait $102 \pm 14$ states on average (online cost/($T_{max} \cdot N$)) to correctly inject the perturbation, which will decrease the attack effectiveness. UAP-S and UAP-O have a higher offline cost than OSFW(U), but the offline generation of $r$ does not interfere with the task, since they do not require interrupting or pausing $v$. The online cost of FGSM, UAP-S, UAP-O and OSFW(U) is lower than $T_{max}$. Although FGSM is computationally fast, it requires rewriting agent’s memory to change all observations of the state in which $v$ is attacked. We claim that modifying agent’s inner workings is too strong of an assumption in realistic adversarial settings because it forces Adv to modify both the environment and the agent. If Adv is able to modify the agent’s memory, then it does not need to compute adversarial perturbations and simply re-writes $v$’s memory to destroy its performance.

4.2.3 Difference in perturbation sizes. We compared the maximum size of perturbation added into observations for different attacks. Figure 3 shows that the perturbation obtained via UAP-S and UAP-O are smaller than other adversarial perturbations for a given $\epsilon$ value. This is because UAP-S and UAP-O try to find a minimal perturbation that sends all $x \in D_{train}$ outside the decision boundary [21]. We conclude that UAP-S and UAP-O are likely to be less detectable based on perturbation sizes (e.g., via visual observation).

As shown in Table 2 summarizing the characteristics of all attacks implemented in the paper, FGSM computes a new perturbation for each $s$. The online cost of OSFW is too high, and it cannot be mounted without interfering with the environment. UAP-S, UAP-O and OSFW(U) are real-time attacks that do not require stopping the agent or the environment while adding the perturbation. UAP-S and OSFW(U) generate a perturbation $r$ that is independent from $s$, but the $r_j$ for each observation $o_j \in s$ is different. On the other hand, UAP-O adds the same $r_j$ to all observations in any $s$, which makes the perturbation generation independent from the size of the state. UAP-O leads to an efficient and effective attack when $\epsilon \geq 0.006$, and it is the only attack that does not have temporal and observational dependency. UAP-S is the best attack considering both effectiveness and efficiency.

In simple environments like Atari 2600 games, sequential states are not i.i.d. Moreover, Atari 2600 games are controlled environments, where future states are predictable and episodes do not deviate much from one another for the same task. OSFW and OSFW(U) leverage this non-i.i.d property. However, their effectiveness might decrease in uncontrolled environments and the physical world due
to the uncertainty of the future states. In contrast, UAP-S and UAP-O are independent of the correlation between sequential states. To confirm our conjecture, we implemented OSFW(U) against VGG-16 image classifiers [40] pre-trained on ImageNet [27], where ImageNet can be viewed as a non i.i.d., uncontrolled environment. We measured that OSFW(U) achieves a fooling rate of up to 30% on the ImageNet validation set with $\epsilon = 10$, while UAP-S has a fooling rate of 78% [21], where the $l_\infty$ norm of an image in the validation set is around 250. Therefore, we conclude that UAP-S can mislead DRL policies more than OSFW(U) in complex, uncontrolled environments.

### 4.3 Attack Performance in Continuous Control

We now consider continuous control tasks to see whether our observations regarding the effectiveness of universal adversarial perturbations generalize beyond discrete tasks. As we explained in Section 2, only policy-based DRL methods are suitable in continuous control tasks. We focus on PPO, since it is more recent in Section 2, only policy-based DRL methods are suitable in continuous control tasks. We focus on PPO, since it is more recent

| Experiment | Attack method | Offline cost std (seconds) | Online cost std (seconds) |
|------------|---------------|---------------------------|---------------------------|
| Pong, DQN, $T_{max} = 0.0163 \pm 10^{-6}$ seconds | FGSM - | $13 \times 10^{-5} \pm 10^{-5}$ |
| | UAP-S | $36.4 \pm 21.1$ | $2.7 \times 10^{-5} \pm 10^{-5}$ |
| | UAP-O | $138.3 \pm 25.1$ | $2.7 \times 10^{-5} \pm 10^{-5}$ |
| | OSFW(U) | $5.3 \pm 0.1$ | $2.7 \times 10^{-3} (\pm 10^{-5})$ |
| Pong, PPO, $T_{max} = 0.0157 \pm 10^{-5}$ seconds | FGSM - | $21 \times 10^{-5} \pm 10^{-5}$ |
| | UAP-S | $41.9 \pm 16.7$ | $2.7 \times 10^{-5} \pm 10^{-5}$ |
| | UAP-O | $138.3 \pm 25.1$ | $2.7 \times 10^{-5} \pm 10^{-5}$ |
| | OSFW(U) | $7.02 \pm 0.6$ | $2.7 \times 10^{-5} \pm 10^{-5}$ |
| Pong, A2C, $T_{max} = 0.0157 \pm 10^{-5}$ seconds | FGSM - | $21 \times 10^{-5} \pm 10^{-5}$ |
| | UAP-S | $11.4 \pm 4.3$ | $2.7 \times 10^{-5} \pm 10^{-5}$ |
| | UAP-O | $55.5 \pm 29.3$ | $2.7 \times 10^{-5} \pm 10^{-5}$ |
| | OSFW(U) | $7.2 \pm 1.1$ | $2.7 \times 10^{-5} \pm 10^{-5}$ |

Table 3: Offline and online comparison of attacks in terms of the maximum upper bound for the perturbation generation and mounting the attack during deployment. Victim agents are PPO trained for Walker2d and Humanoid at $\epsilon = 0.02$. Attacks that cannot be implemented in real-time are highlighted in red.
than A2C, and is considered to be the state-of-the-art reinforcement learning algorithm [28]. For υ, we used previously trained PPO agents [39] for two different MuJoCo environments: Walker2d and Humanoid\(^3\). We used the experimental setup given by authors in their GitHub repository to test PPO agents and compared our attacks with baseline attacks as well as random noise addition. In our experiments, PPO agents show similar performance as the ones reported in the original paper [39]. We implemented UAP-S by copying the parameters of \(V_v\) into \(V_{adv}\), and set \(\delta_{max} = 95\%\). We should note that FGSM, OSFW and OSFW(U) also use \(V_v\) to minimize the value in a perturbed state, instead of \(Q_v\), which is not available in continuous control. Additionally, in both tasks, \(s\) contains only one observation, thus reducing UAP-S to UAP-O.

Figure 4 shows the attack effectiveness when \(\epsilon\) is between 0.0 and 0.2. FGSM is the most effective attack in Humanoid, while UAP-S decreases the return more than FGSM in Walker2d when \(\epsilon \geq 0.12\). Overall, all attacks behave similarly in both discrete and continuous action spaces. Therefore, our conclusions regarding the effectiveness of universal adversarial perturbations generalize to continuous control tasks, where \(Q_v\) is not available. However, in these tasks, the adversary \(Adv\) only decreases the critic’s evaluation of state \(s\) when taking an action \(a\). Even if \(Adv\) did decrease the value of \(V_{adv}(s)\), it does not necessarily lead to \(v\) choosing a sub-optimal action over the optimal one. Therefore, attacks that only rely on the value function requires \(\epsilon \geq 0.2\) to fool DRL policies effectively. For a more efficient attack, adversaries can clone \(v\)’s policy \(\pi_v\) into a proxy agent as \(\pi_{adv}\) and try to maximize the total variation distance [39] in \(\pi_{adv}\) for states perturbed by universal perturbations. However, UAP-S, UAP-O and OSFW(U) need to be modified to compute the total variation distance, and we leave this as a future work.

Table 3 presents the online and offline computational cost for perturbation generation in continuous control tasks. The results are comparable with Table 1, and confirm that FGSM, UAP-S, UAP-O and OSFW(U) can be mounted in real-time, although FGSM needs write-access to the agent’s memory. OSFW cannot inject the generated noise immediately to subsequent states, since it has a higher online cost than the maximum upper bound.

5 DETECTION AND MITIGATION OF ADVERSARIAL PERTURBATIONS

A defender’s goal is the opposite of the adversary \(Adv\)’s goal. Section 3.1 outlines three criteria for a successful attack. A good defense mechanism should thwart one or more of these criteria. We begin by surveying previously proposed defenses (Section 5.1), followed by an evaluation of the effectiveness of two of the most relevant defenses (Section 5.2) that aim to address the first (lowering the victim \(v\)’s return) and third attack criterion (evasion of detection). We then present a new approach to detect the presence of adversarial perturbations in state observations (Section 5.3.1), aimed at thwarting third attack criterion on evading defenses.

---

\(^3\)Agents are downloaded from https://github.com/huanzhang12/SA_PPO and the frame rates are set to default values as in https://github.com/openai/gym

5.1 Defenses against Adversarial Examples

5.1.1 Defenses in image classification. Many prior studies proposed different methods to differentiate between normal images and adversarial examples. Meng and Chen [18] or Rouhani et al. [26] model the underlying data manifolds of the normal images to differentiate clean images from adversarial examples. Xu et al. [38] propose feature squeezing to detect adversarial examples. However, these detection methods are found to be inefficient [5] and not robust against adaptive adversaries that can specifically tailor adversarial perturbation generation methods for a given defense [34].

5.1.2 Defenses in DRL. Previous work [3, 14] has demonstrated that adversarial training improves agents’ resilience to adversarial perturbations. However, Zhang et al. [39] show that adversarial training leads to performance degradation, and is not robust against strong attacks. In order to overcome the challenges of adversarial training, Zhang et al. [39] propose state-adversarial Markov decision process (SA-MDP). SA-MDP aims to find an optimal \(\pi\) under the strongest \(Adv\) using policy regularization. This regularization technique helps DRL agents maintain their performance even against adversarially perturbed inputs.

Visual Foresight [17] is a defense that is intended to recover the performance of a DRL agent in the presence of \(Adv\). Visual Foresight deploys an action-conditioned frame prediction module to predict the current observation \(\hat{o}_t\) at time \(t\) using \(k\) previous observations \(o_{t-k} : o_{t-1}\) and corresponding actions \(a_{t-k} : a_{t-1}\). It also predicts the possible action \(\hat{a}_t\) for the partially predicted \(\hat{o}\) using \(Q(\hat{s}, a)\), where \(\hat{s} = \{fpred(o_{t-N+1}), \ldots, fpred(\hat{o}_t)\}\). The difference between the action distribution (i.e., Softmax(\(Q(s, a \in A)\))) obtained by the predicted \(\hat{s}\) and the current \(s\) is used to detect whether \(s\) is perturbed or not. In the case of detection, \(\hat{a}_t\) is selected to recover the performance of the agent. Both Visual Foresight and SA-MDP can recover the performance of agents when states are perturbed using prior attacks strategies in [11, 12, 35].

5.2 Effectiveness of Existing Defenses

While adversarial training [3, 14] presents promising results at defending against adversarial examples, it also leads to unstable training, performance degradation and is ineffective against strong attacks [39]. Moreover, Moosavi et al. [21] prove that despite a slight decrease in the fooling rate \(\delta\) on the test set, \(Adv\) can easily compute another universal perturbation against retrained agents. Thus, directly applying adversarial retraining is not an effective defense against attacks using universal perturbations.

To investigate the limitations of previously proposed defenses for DRL, we implemented two defense methods that aim to retain the average return of \(v\) when it is under attack: Visual Foresight [17] and SA-MDP [39] (Section 5.1) both of which seek to prevent the first attack criterion that limits \(v\) to a low return. Visual Foresight also prevents the third attack criterion, and detects adversarial perturbations. Since we want to evaluate the effectiveness of Visual Foresight and SA-MDP, we focus on DQN agents for Pong and Freeway as these are the ones that are common between our experiments (Section 4) and these defenses ([17, 39]).

We implemented Visual Foresight from scratch for our DQN models following the original experimental setup in [17] by setting \(k = 3\) to predict every 4th observation. We also set the pre-defined
| $\epsilon$ | Defense               | Average return ± std in the presence of adversarial perturbation attacks |
|---------|-----------------------|-------------------------------------------------------------------------|
|         | No attack                  | FGSW            | OSFW          | UAP-S         | UAP-O         | OSFW(U)        |
| 0.01    | No defense                | 21.0 ± 0.0      | −21.0 ± 0.0   | −20.0 ± 3.0   | −21.0 ± 0.0   | −19.8 ± 0.4    | −21.0 ± 0.0    |
|         | Visual Foresight [17]     | 21.0 ± 0.0      | 21.0 ± 0.0    | −19.7 ± 0.5   | 0.7 ± 1.7     | 0.4 ± 2.7      | 21.0 ± 0.0     |
|         | SA-MDP [39]               | 21.0 ± 0.0      | 21.0 ± 0.0    | 21.0 ± 0.0    | 21.0 ± 0.0    | 21.0 ± 0.0     | 21.0 ± 0.0     |
| 0.02    | No defense                | 21.0 ± 0.0      | 21.0 ± 0.0    | −20.8 ± 0.6   | −20.0 ± 0.0   | −21.0 ± 0.0    |
|         | Visual Foresight [17]     | 21.0 ± 0.0      | 21.0 ± 0.0    | −19.7 ± 0.6   | 9.4 ± 0.8     | 5.3 ± 3.9      | 20.5 ± 0.5     |
|         | SA-MDP [39]               | 21.0 ± 0.0      | 21.0 ± 0.0    | −20.6 ± 0.5   | −20.6 ± 0.5   | −21.0 ± 0.0    |
| 0.05    | No defense                | 21.0 ± 0.0      | 21.0 ± 0.0    | −20.6 ± 0.8   | −20.0 ± 0.0   | −21.0 ± 0.0    |
|         | Visual Foresight [17]     | 21.0 ± 0.0      | 21.0 ± 0.0    | 7.6 ± 4.7     | 14.1 ± 1.1    | 21.0 ± 0.0     |
|         | SA-MDP [39]               | 21.0 ± 0.0      | 21.0 ± 0.0    | −20.6 ± 0.5   | −20.6 ± 0.5   | 21.0 ± 0.0     |

(a) DQN agent playing Pong

| $\epsilon$ | Defense               | Average return ± std in the presence of adversarial perturbation attacks |
|---------|-----------------------|-------------------------------------------------------------------------|
|         | No attack                  | FGSW            | OSFW          | UAP-S         | UAP-O         | OSFW(U)        |
| 0.01    | No defense                | 34.0 ± 0.0      | 0.2 ± 0.0     | 2.0 ± 1.1     | 2.1 ± 0.8     | 4.0 ± 0.6      | 0.5 ± 0.5      |
|         | Visual Foresight [17]     | 32.0 ± 1.5      | 32.6 ± 1.7    | 24.1 ± 1.0    | 22.9 ± 0.9    | 25.8 ± 1.1     | 20.9 ± 1.2     |
|         | SA-MDP [39]               | 30.0 ± 0.0      | 30.0 ± 0.0    | 30.0 ± 0.0    | 30.0 ± 0.0    | 30.0 ± 0.0     |
| 0.02    | No defense                | 34.0 ± 0.0      | 0.2 ± 0.0     | 1.0 ± 0.0     | 0.1 ± 0.3     | 0.8 ± 0.6      | 0.0 ± 0.0      |
|         | Visual Foresight [17]     | 32.0 ± 1.5      | 32.6 ± 1.7    | 1.1 ± 0.3     | 24.0 ± 2.0    | 25.6 ± 1.0     | 4.4 ± 1.1      |
|         | SA-MDP [39]               | 30.0 ± 0.0      | 29.8 ± 0.6    | 29.9 ± 0.3    | 29.4 ± 1.2    | 29.4 ± 1.2     |
| 0.05    | No defense                | 34.0 ± 0.0      | 0.2 ± 0.0     | 1.2 ± 0.0     | 2.2 ± 1.7     | 2.2 ± 1.4      | 0.0 ± 0.0      |
|         | Visual Foresight [17]     | 32.0 ± 1.4      | 32.6 ± 1.6    | 1.0 ± 0.0     | 29.0 ± 1.1    | 23.9 ± 0.3     |
|         | SA-MDP [39]               | 30.0 ± 0.0      | 21.1 ± 1.3    | 20.9 ± 0.8    | 21.1 ± 1.7    | 21.1 ± 1.7     |

(b) DQN agent playing Freeway

Table 4: Average returns (10 episodes) in the presence of different adversarial perturbations, and agents are equipped with different types of defenses. In each row, the best attack (lowest return) is in bold font. In each column, for a given $\epsilon$ value, the most robust (highest return) defense for that particular adversarial perturbation attack is shaded (green).

threshold value to 0.01, which is used for detecting adversarial perturbations, in order to achieve the highest detection rate and performance recovery. We downloaded state adversarial DQN agents, which are trained using SA-MDP, from their reference implementation.4 SA-MDP agents only use one observation per $s$; therefore, UAP-O would reduce to UAP-O for SA-MDP in this setup. Table 4 shows the average return for each agent under a different attack, and while equipped with different defenses. In Section 4, we established that when the perturbation bound $\epsilon$ is 0.01, all attacks are devastatingly effective. In the interest of evaluating the robustness of the defense, we also consider two higher $\epsilon$ values, 0.02 and 0.05.

Visual Foresight is an effective defense against FGSW and UAP-S as it can recover $v$’s average return. However, it is not very effective against OSFW, OSFW(U), and UAP-O. SA-MDP is better than Visual Foresight against OSFW and OSFW(U) when $\epsilon = 0.01$, but it fails to defend any attack in the case of Pong when $\epsilon \geq 0.02$. Notably, Visual Foresight’s detection performance depends on the accuracy of the action-conditioned frame prediction module [17], and the pre-defined threshold value used for detecting adversarial perturbations. SA-MDP is able to partially retain the performance of $v$ when $\epsilon = 0.01$ and 0.02 in the case of Freeway. Notably, $v$’s average return when it is under attack is still impacted, and SA-MDP cannot retain the same average return as when there is no attack. Furthermore, SA-MDP’s effectiveness decreases drastically when the perturbation bound $\epsilon$ is increased to 0.05.

5.3 Detecting Adversarial Perturbations

The third criterion for a successful attack is evading detection (Section 3.1). For tasks with discrete action spaces, we now consider a defense that works in opposition to this criterion.

5.3.1 Action Distribution Divergence Detector (AD$^3$). In a typical DRL episode, the sequential actions exhibit some degree of temporal coherence: the likelihood of the agent selecting a specific current action given a specific last action is consistent across different episodes. We also observed the temporal coherence is disrupted when the episode is subjected to an attack. We leverage this knowledge to propose a detection method, Action Distribution Divergence Detector (AD$^3$), that calculates the statistical distance between the conditional action probability distribution (CAPD) of the current episode to the learned CAPD in order to detect whether the agent is under attack. Unlike prior work on detecting adversarial examples in the image domain [18, 26, 38], AD$^3$ does not analyze the input image or tries to detect adversarial examples. Instead, it observes the distribution of the actions triggered by the inputs and detects unusual action sequences.

4https://github.com/chenhongge/SA_DQN
Downloaded SA-MDP models were trained with $\epsilon = 1/255$ as in the original paper [39]. Using higher $\epsilon$ values in training led to poor performance.
To train AD$^3$, v first runs $k_1$ episodes in a safe, controlled environment before the deployment. AD$^3$ saves all actions taken during that time and approximates the conditional probability of the next action given the current one using the bi-gram model. We call the conditional probability of actions approximated by $k_1$ episodes as the \textit{learned} CAPD. Second, to differentiate between the CAPD of a normal game versus a game that is under attack with high confidence, v runs another $k_2$ episodes in a safe environment. AD$^3$ decides a threshold value $t_h$ where the statistical distance between the CAPD of the normal game and the learned CAPD falls mostly under this threshold. We use Kullback–Leibler (KL) divergence [15] as the statistical distance measure. The KL divergence between the learned CAPD and the CAPD of the current episode is calculated at each time-step starting after the first $t_1$ steps. We skip the first $t_1$ time-steps because the CAPD of the current episode is initially unstable and KL divergence is naturally high at the beginning of every episode. We set the threshold $t_h$ as the $p^{th}$ percentile of all KL-divergence values calculated for $k_2$ episodes. During deployment, AD$^3$ continuously updates the CAPD of the current episode, and after $t_1$ time-steps, it calculates the KL-divergence between the CAPD of the current episode and the learned CAPD. If the KL-divergence exceeds the threshold $t_h$ by $\%$ or more during a time window $t_2$, then AD$^3$ raises an alarm that the agent is under attack.

We evaluate the precision and recall of AD$^3$ in three tasks with discrete action spaces (Pong, Freeway, and Breakout) against all proposed attacks. AD$^3$ can detect all five attacks in Pong with perfect precision and recall scores in all configurations. In Freeway, AD$^3$ has a perfect precision and recall scores for 12 out of 15 different setups (FGSM against DQN agent, OSFW against PPO agent, and OSFW against PPO agent). AD$^3$ is less effective against Breakout as this task often terminates too quickly when it is under an attack. Detailed discussion on the performance of AD$^3$ and the full result of the evaluation can be found in Appendix C.

Visual Foresight aims to detect adversarial perturbations for each s in an episode. This means that Visual Foresight can raise many false alarms in episodes with no attack because it cannot differentiate clean states from adversarially perturbed states when using a low pre-defined threshold value. On the other hand, AD$^3$ does not detect the presence of Adv at every s but instead AD$^3$ takes information from multiple actions within an episode and makes a judgement regarding the presence of Adv. Therefore, it is difficult to compare AD$^3$ and Visual Foresight directly, as these methods are optimized for different purposes.

AD$^3$ is designed for attacks where adversaries apply their adversarial perturbations consistently throughout the episode. As such, Adv with the knowledge of the detection strategy could apply their adversarial perturbation at a lower frequency to avoid detection. However, lowering the attack frequency also decreases the attack effectiveness. Another way for Adv to evade AD$^3$ is to perform targeted attacks to lure v into a specific state, where the adversarial perturbation is applied to a limited number of states in an episode. This attack type is out of the scope of our paper, and defense strategies against it will be explored in future work.

### 5.3.2 Adversary vs. defender strategy with negative returns

In any DRL task where there is a clear negative result for an episode (e.g., losing a game) or the possibility of negative return, a reasonable choice for v is to suspend an episode when Adv’s presence is detected. For example, in Pong, a negative result would be when the computer (as the opponent) reaches the score of 21 before v does. Suspending an episode prevents v from losing the game.

Defense mechanisms such as Visual Foresight and SA-MDP are useful in retaining or recovering v’s return; however, they may not always prevent v from falling into a negative result, e.g., losing the game in Pong. Combining a recovery/retention mechanism with suspension on attack detection can reduce the number of losses for v.

To illustrate the effectiveness of combining a detection mechanism like AD$^3$ with a retention/recognition mechanism, we designed an experiment using a QN agent playing Pong to compare the \textit{losing rate of v} when it is under attack. We only use Pong for this experiment as Pong has a clear negative result. In this setting, an episode ends with the loss when (a) the computer reaches 21 points before v, or (b) AD$^3$ did not raise an alarm. The result of this experiment can be found in Table 5. As shown in this table, Visual Foresight is not effective in reducing the losing rate of v for OSFW and OSFW(U) for all $\epsilon$. SA-MDP is effective in avoiding losses when $\epsilon = 0.01$, however it fails against all universal perturbations when $\epsilon = 0.02$. In contrast, AD$^3$ can detect the presence of adversarial perturbations in all games.

Although retention/recognition and detection are two orthogonal aspects of defense, our results above suggest that they can be combined in tasks with negative returns or results in order to more effectively thwart Adv from achieving its first goal.

### 6 CONCLUSION

We showed that white-box universal perturbation attacks are effective in fooling DRL policies in real-time. Our evaluation of the three different attacks (UAP-S, UAP-O and OSFW(U)) demonstrates that universal perturbations are effective in tasks with discrete action spaces. Universal perturbation attacks are also able to generalize to continuous control tasks with the same efficiency. We confirmed that the effectiveness of prior defenses depends on the perturbation bound, and fail to completely recover the agent performance.
when they are confronted with universal perturbations of larger bounds. We proposed a detection mechanism, AD3, that detects all five attacks evaluated in the paper. AD3 can be combined with other defense techniques to protect agents in tasks with negative returns or results to stop the adversary from achieving its goal. In the future, we plan to extend our attacks to the black-box case by first mounting a model extraction attack and then applying our current techniques to find transferable universal perturbations.

REFERENCES

[1] Shumeet Baluja and Ian Fischer. 2018. Learning to Attack: Adversarial Transformation Networks. In Proceedings of AAAI-2018. http://www.esprockets.com/papers/aaa2018.pdf

[2] Sahid Behzadan and Arslan Munir. 2017. Vulnerability of deep reinforcement learning to policy induction attacks. In International Conference on Machine Learning and Data Mining in Pattern Recognition. Springer, 262–275.

[3] Vahid Behzadan and Arslan Munir. 2017. Whatever does not kill reinforce- ment learning, makes it stronger. arXiv preprint arXiv:1712.09344 (2017).

[4] Marc G Bellemare, Yavar Nadar, Joel Veness, and Michael Bowling. 2013. The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research 47 (2013), 253–279.

[5] Nicholas Carlini and David Wagner. 2017. Adversarial examples are not easily detected: Bypassing ten detection methods. In Proceedings of the 10th ACM workshop on artificial intelligence and security. 3–14.

[6] Nicholas Carlini and David Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In 2017 IEEE Symposium on Security and Privacy (SP) 39–57. https://doi.org/10.1109/SP.2017.49

[7] Kenneth T Co, Luis Muñoz-González, Sixte de Maupeou, and Emil C Lupu. 2019. Procedural noise adversarial examples for black-box attacks on deep convolu- tional networks. In Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. 275–289.

[8] Adam Gleave, Michael Dennis, Neil Kant, Cody Wild, Sergey Levine, and Stuart Russell. 2019. Adversarial policies: Attacking deep reinforcement learning. arXiv preprint arXiv:1905.06115 (2019).

[9] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In International Conference on Learning Representations. http://arxiv.org/abs/1412.6572

[10] Jamie Hayes and George Danezis. 2018. Learning universal adversarial perturba- tions with generative models. In 2018 IEEE Security and Privacy Workshops (SPW). IEEE. 43–49.

[11] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. 2017. Adversarial Attacks on Neural Network Policies. arXiv (2017). https://arxiv.org/abs/1702.08284

[12] Leonard Hussenot, Matthieu Geist, and Olivier Pietquin. 2020. CopyCAT: Taking Current Techniques to Find Transferable Universal Perturbations. In Proceedings of the IEEE conference on computer vision and pattern recognition. 1765–1773.

[13] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. Deepfool: a simple and accurate method to fool deep neural networks. In Proceedings of the IEEE conference on computer vision and pattern recognition. 2574–2582.

[14] Konda Reddy Mopuri, Utsav Garg, and R Venkatesh Babu. 2017. Fast feature fool: A data independent approach to universal adversarial perturbations. arXiv preprint arXiv:1707.05572 (2017).

[15] Konda Reddy Mopuri, Utkarsh Ohja, Utsav Garg, and R Venkatesh Babu. 2018. NAG: Network for adversary generation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 742–751.

[16] Yen-Chen Lin, Zhang-Wei Hong, Yuan-Hong Liao, Meng-Li Shih, Ming-Yu Liu, and Dongyu Meng. 2017. Magnet: a two-pronged defense against adversarial perturbations. In Proceedings of the IEEE conference on computer vision and pattern recognition. 1423–1432.

[17] Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. Journal of the American Statistical Association 46 (1951), 79–86.

[18] Yen-Chen Lin, Ming-Yu Liu, Min Sun, and Jia-Bin Huang. 2017. Detecting other defense techniques to protect agents in tasks with negative returns or results to stop the adversary from achieving its goal. In the future, we plan to extend our attacks to the black-box case by first mounting a model extraction attack and then applying our current techniques to find transferable universal perturbations.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dhariash Kumar, Janusz Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. Nature 518, 7540 (2015), 529–533. https://doi.org/10.1038/nature14236

[20] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asyn- chronous Methods for Deep Reinforcement Learning. In Proceedings of the 33rd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, Vol. 48. 1928–1937.
A PROOF OF EQUATION 6

In this section, we provide a formal proof of Equation 6 that was used to find the perturbation in UAP-O.

In UAP-O, we defined an additional constraint for the perturbation as $F_{ij} = F_{ik}$, $\forall j, k \in \{t - N + 1, \ldots, t\}$, so that the same $F_{ij}$ is applied to all observations $o_j$. As pointed out in Section 3.2, at $i$-th iteration, the DeepFool algorithm computes the $i$-th perturbation $F_i$ for $i$-th state $s_i$ as

$$\Delta F_i \leftarrow \frac{Q(s_i', a_i')}{\|w_i'\|^2} - w_i.$$  \hfill (8)

We can keep the term $|Q(s_i', a_i')/\|w_i'\|^2$ as constant, since it gives the magnitude of the perturbation update $\Delta F_i$, and we can maintain the same magnitude of $\Delta r_i$ in $\Delta F_i$. Then, in UAP-O, DeepFool finds the closest $\Delta F_i$ to $\Delta r_i$ by minimizing the direction of the perturbation as $\|w_i' - \Delta F_i\|^2$.

We can find the optimal $\Delta F_i$ by setting $\frac{\partial \|w_i' - \Delta F_i\|^2}{\partial \Delta F_i}$ to zero for all $i \in \{t - N + 1, \ldots, t\}$, where $N$ denotes the total number of observations inside state $s$. We also set all $\Delta F_i$ into the same variable $\hat{F}_i$. Therefore, we simply obtain $\hat{F}_i$ by finding the point where the partial derivative is equal to zero as in Equation 9.

$$\frac{\partial \|w_i' - \Delta F_i\|^2}{\partial \Delta F_i} = \frac{\partial ((w_i' - \hat{F}_i)^2 + \cdots + (w_{i-N+1}' - \hat{F}_i)^2)^{1/2}}{\partial \Delta F_i} = 0$$

$$= \frac{\partial f((\hat{F}_i)^{1/2})}{\partial \hat{F}_i} = 0.5 f((\hat{F}_i)^{-3/2})$$

The squared difference function $f((\hat{F}_i)^{-3/2}) \geq 0$, and only equal to zero when $\hat{F}_i = 0$, which does not give any useful perturbation. Therefore, Equation 9 is solved when $w_i' = \hat{F}_i$, $\forall k \in \{t - N + 1, \ldots, t\}$. Therefore, by putting back the constant, DeepFool computes the perturbation at $i$-th iteration as

$$\hat{F}_i \leftarrow \frac{Q(s_i', a_i')}{N\|w_i'\|^2} \sum_{k=t-N+1}^{t} w_i'. \hfill (10)$$

B ADVERSARIAL PERTURBATIONS

On top of comparing the perturbation size in Pong (Figure 3, Section 4.2), we also include results comparing the perturbation size for other games. As shown in Figure 5, FGSM generates a perturbation that contains grey patches where the sign of the gradient of the loss is zero against the DQN agent. It has also completely black and white pixels due to the $\ell_0$ constraint, which makes the perturbation more visible. In addition, Figure 6 compares the clean and perturbed frames generated by different attacks. Note that we inject the perturbation into pre-processed frames (i.e., resized and gray-scale frames), so we show both RGB and gray-scale clean frames in Figure 5 and Figure 6. In Pong, the perturbation size in UAP-S is the smallest when compared to other attacks.

C OPTIMAL PARAMETERS FOR AD$^3$

We performed a grid search to find optimal parameters for AD$^3$ in Breakout, Pong, and Freeway using three different agents (DQN, A2C, and PPO) and choose the parameters with the best F1-score. The optimal parameter values can be found in Table 2. The precision and recall scores of AD$^3$ against five different attacks at $\epsilon = 0.01$ can also be found in this table. AD$^3$ is able to detect the presence of the adversary $Adv$ with perfect precision and recall for most environments and agents. However, there are some combinations of environment and agent, where AD$^3$ has a low recall and/or precision.

Attacks such as OSFW(U) in Freeway against PPO agents have a low action change rate (as low as 20% – 30% in some episodes), and

| Game     | Agent | Parameters | Attack | Precision/Recall |
|----------|-------|------------|--------|------------------|
| Pong     | dqn   | $k_1 = 12$, $k_2 = 24$, $p = 100$, $\epsilon = 0.5$ | FGSM | 1.0/1.0 |
|          |       | $r = 0.5$, $t_1 = 400$, $t_2 = 200$ | OSFW(U) | 1.0/1.0 |
|          | a2c   | $k_1 = 12$, $k_2 = 24$, $p = 100$, $\epsilon = 0.5$ | FGSM | 1.0/1.0 |
|          |       | $r = 0.5$, $t_1 = 400$, $t_2 = 200$ | OSFW(U) | 1.0/1.0 |
|          | ppo   | $k_1 = 12$, $k_2 = 24$, $p = 100$, $\epsilon = 0.5$ | FGSM | 1.0/1.0 |
|          |       | $r = 0.5$, $t_1 = 400$, $t_2 = 200$ | OSFW(U) | 1.0/1.0 |

Table 6: Optimal values of AD$^3$ to detect five different attacks: FGSM, OSFW, UAP-S, UAP-O, and OSFW(U) at $\epsilon = 0.01$ for all games and agents over ten episodes.
Figure 5: Comparison of perturbation size added into the same clean (in both RGB and gray-scale) observation in different attacks when $\epsilon = 0.01$. UAP-S and UAP-O generate smaller perturbations. Top row: DQN agent playing Pong, Middle row: PPO agent playing Breakout, Bottom row: A2C agent playing Freeway. In perturbation images, black pixels: $-0.01$, white pixels: $+0.01$, gray pixels: $0.0$.

Figure 6: Comparison of the clean (in both RGB and gray-scale) and perturbed observations in different attacks when $\epsilon = 0.01$. UAP-S and UAP-O generate smaller perturbations that are less visible. Top row: DQN agent playing Pong, Middle row: PPO agent playing Breakout, Bottom row: A2C agent playing Freeway.

the resulting average return is still high. This means that $Adv$ does not change many actions in an episode. This low action change rate is consistent with all of the combinations of the game, agent, and attack that have a low recall score. As $AD^3$ relies on modeling the distribution of the victim $v$’s action distribution during an episode, it cannot detect the presence of $Adv$ if there are not enough actions influenced by the attack in an episode. This means that a low action change rate from $Adv$ would make it difficult for $AD^3$ to detect. The precision of $AD^3$ in Breakout is lower than other environments, since episodes in Breakout terminate very quickly (as fast as in 112 time steps) when $v$ is under attack. This means that to detect the presence of $Adv$, $AD^3$ has to raise an alarm early in Breakout. However, $AD^3$ relies on modeling CAPD of the current episode, and it takes time to converge. Before CAPD converges, the anomaly score is going to be high for any episode, including episodes with no attack presence. As an episode can terminate quickly in Breakout, $AD^3$ cannot store enough actions for CAPD to converge, and this is reflected in the reported low precision values.