Transferability of Adversarial Reinforcement Learning

Connor Fuhrman, Johnathan Gill

May 11 2021

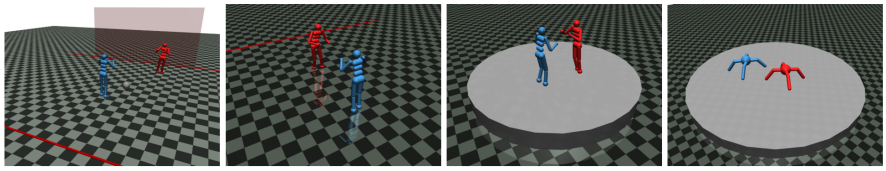
Abstract

It is well known that deep learning and deep reinforcement learning (RL) are vulnerable to adversarial attacks via perturbations applied to a model's input [9]. Recently, it was shown that RL is susceptible to adversarial attacks on the *policy* alone via an attacker trained to develop an adversarial policy [1]. Such attacks are especially interesting because the adversarial attacker does not simply perturb the victim's model's input, i.e., the observation space in the domain of RL, but rather choses physically realistic actions such that the victim's policy breaks down. This method of adversarial attack is critical to understand more deeply because real-world RL agents may only interact with each other and the environment around them in a physically realistic fashion and may not artificially perturb the observation space of another agent. This work investigates the transferability of such adversarial policies between various model architectures via the *Shoot and Defend* zero-sum game. Quadcopter agents are trained to be either the shooter, who's goal is to shoot a ball into a goal, or a defender, who's goal is to keep the ball out of the goal. There are two shooter agents trained with distinctly different model architectures and two keeper agents, one of which is trained adversarially against one shooter agent but **not** the other. We find that ...

1 Introduction

Adversarial attacks against neural network classifiers [9] and (deep) reinforcement learning (RL) [1] has been shown effective via adversarial perturbations of the victim model's input, e.g., altering pixel values for an image classifier or perturbing an observation space for an RL model. However, such attacks are artificial when considering real-world RL agents as an adversarial attacker cannot directly alter, e.g., image pixel or observation space values, but can only interact naturally with the environment and the victim model, e.g., an adversarial attack against an autonomous unmanned aerial vehicle (UAV) cannot access the UAV's perception equipment directly to alter measurement values nor can it alter the environment in a non-physically realistic manner but can only conduct physically-realistic actions, e.g., a flight pattern, which serves to attack and exploit an agent's policy.

Work done in [1] showed that an adversarial policy was effective in multiple zero-sum games, including *Shoot and Defend* where the `shooter` agent attempts to shoot a ball into a goal while the `keeper` agent seeks to keep the ball out of the goal. The adversarial attacker, the keeper agent, demonstrated significantly different activations than the normal models and was overall successful in each game by manipulating only it's own body position to exploit weaknesses in the victim's (the shooter agent) policy. Videos showing the results from [1] are available online and show that adversarial policies are more effective in higher-dimensional spaces such as humanoid models over lower-dimensional spaces such as ants.



(a) Kick and Defend (b) You Shall Not Pass (c) Sumo Humans (d) Sumo Ants

Figure 1: Training environment used in [1] showing all four zero-sum games investigated for adversarial policies

The adversarial agents in [1] were given unlimited black-box access to the victim's policy, π_v, but were not given any white-box access to model weights or activations. The victim model's policy was frozen and actions were sampled from the stochastic policy such that a_v is a sample of the stochastic policy $\pi_v(.|s)$ while the adversary developed a policy π_a such that π_a maximizes the discounted sum of rewards. Agents trained adversarially demonstrate reliable domination of the vicim models, winning upwards of 90% of games in most cases. However, it is interesting to note that when the victim is "blind" to the

attacker, i.e., the victim's observations corresponding to the attacker's position is set statically to a typical initial value, its performance increases significantly from losing over 80% of games to winning 99% of games in *You Shall Not Pass* (this game demonstrates the phenomenon best but the trend exists in other games [1]).

2 Methods

The following sections discuss the gym environment, RL framework, and deployment on the high performance computing (HPC) cluster at the University of Arizona.

2.1 Gym Environment

We utilize an OpenAI-based gym environment introduced in [6] which provides a multi-agent RL environment based on the `Bullet` physics engine. The environment, `gym-pybullet-drones`, (original repository available online at `github.com/utiasDSL/gym-pybullet-drones`) allows users to rapidly create new environments and instantiate models within defined by their Unified Robot Description Format (URDF).

The drone model utilized in this work is the Bitcraze Cracyflie 2.1 as this model ships with the gym environment. The quadcopter itself is a small open-source development platform which weight under 30g and is intended for education, research, and in particular swarm applications. Note, however, that this drone model in particular is not integral to this work but was chosen simply because it was readily available and already integrated/tested within the gym environment. The use of quadcopters was chosen because [1] showed that agents were more susceptible to adversarial attacked in higher dimensional spaces and quadcopters, as they are aerial vehicles, introduce a vertical component.

2.2 Shoot and Defend Environment

The *Shoot and Defend* environment is configured as in Figure 2. The shooter and defender agents are each constrained to remain in their respective areas and receive a punishment if this boundary is violated. The shooter agent receives a reward if the ball enters the goal area and a penalty if the ball goes out of bounds or becomes stationary. Both agents also receive penalties for crashing and other undesirable behavior. The exact rewards and conditions are shown in Appendix A.

The observation space for each agent is the physical properties of all physical objects in the scene, i.e., the position, attitude, velocity, and angular velocity. However, in order to reduce the complexity and without loss of generality the observations for the ball object are simply position and velocity as the ball's

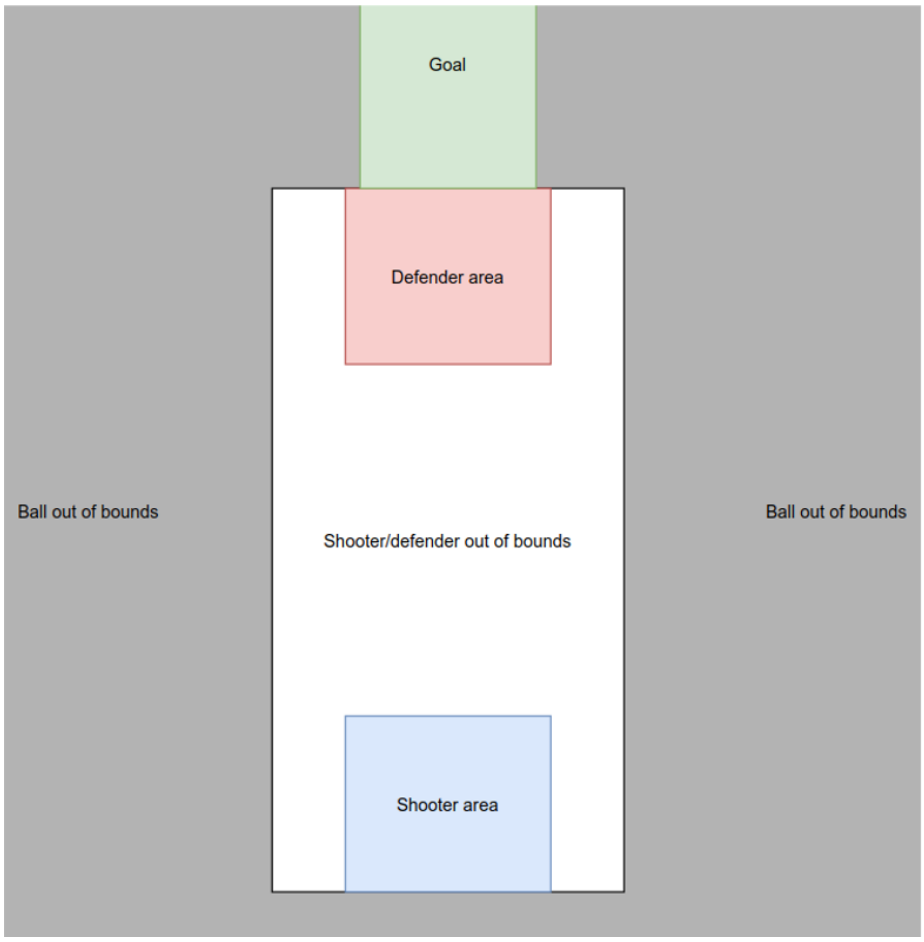


Figure 2: Diagram depicting the *Shoot and Defend* environment. Note this is a top-down view as the shooter and defender may travel in the vertical direction.

attitude and angular velocity [1] do not play as significant a role in the game as the position and velocity. Therefore, the dimensionality for the observation space of each agent is a 30-length vector for both the shooter and defender agents. The action space for the defender agent is an 8-vector representing the mean and standard deviation of a Gaussian distribution by which motor revolution per minute (RPM) values are sampled as this encourages early exploration as the standard deviation decreases over training iterations. The shooter agent adds an additional mean and standard deviation pair which indicates that the shooter agent should shoot the ball. If the value of this sampled distribution is

[1] The ball is shot with a force of 50N and there is no external force applied by moving air. The ball is then essentially only affected by gravity and would not experience a curved trajectory as in a windy environment

above a threshold, the shooter launches the ball in the drone platform's current orientation [2].

2.3 Training Methodology

Both the shooter and defender agents are trained using Proximal Policy Optimization (PPO) [8] via the Ray RLlib library [4] which provides RL-specific libraries and functionality atop the Ray API for distributed operation. A policy gradient method, in particular PPO, was chosen over Q networks due to the method's ability to handle continuous action spaces, i.e., the quadcopter's motor RPM commands. Policy gradient methods have been demonstrated to achieve state-of-the-art performance using deep neural networks in multiple application areas including, but not limited to, Go, Atari and other video games, and 3D locomotion [2]. PPO addresses issues with various other policy optimization methods such as sensitivity to step size and poor sample efficiency while avoiding overcomplicated implementations, e.g., Actor-Critic with Experience Replay (ACER) [11] which introduces more implementation complexity via off-policy correction and a replay buffer while showing only marginal improvement over PPO in OpenAI's Atari benchmark [2], or algorithms, e.g, Trust Region Policy Optimization (TRPO) [7], which are unable to share parameters between the policy and value function or auxiliary losses such as dropout (e.g., Atari games or other domains heavily-reliant on visual input [2]). PPO utilizes the objective function

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t] \tag{1}$$

where

- θ is the policy parameter
- $\hat{E}_t$ is the empirical expectation over timesteps
- r_t is the ratio of probabilities under the new and old policies respectively
- $\hat{A}_t$ is the estimated advantage at time t
- ϵ is a hyperparameter [3]

We utilize the PPO implementation via Ray RLlib [10] as the Ray library provides abstractions for parallelization and distributed workflows and the RLlib library contains a suite of highly parallelizable GPU-enabled algorithms which are (mostly) compatible with both TensorFlow and PyTorch. Figure 3 shows the relationship between Ray, the gym environment, and RLlib algorithms.
The complete Ray configuration is shown in Appendix B.

TODO: Discuss model configuration such as LSTM and what the different models were.

[2] Note that the ball is instantiated as a model at the time of shooting and the shooter agent does not carry the ball beforehand.

[3] The authors of [8] and [2] recommend usually 0.2 or 0.3

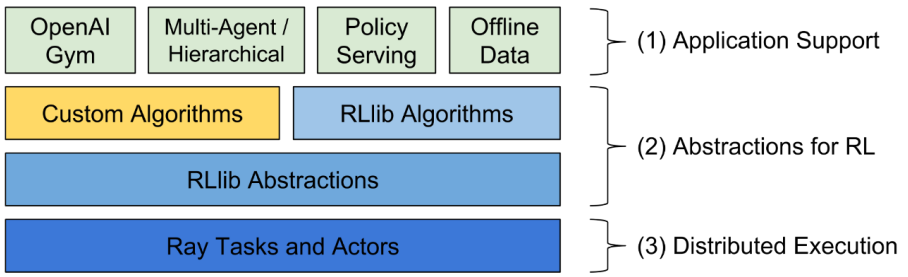


Figure 3: Ray library conceptual block diagram [10]. The *Shoot and Defend* environment created herein is an OpenAI-compatible Gym and we utilize RLlib algorithms which utilize Ray's abstraction layer to effectuate distributed execution.

2.4 Containerization via Singularity and Deployment on High Performance Computing (HPC) Cluster

In order to effectively train multiple RL agents with varying model architectures, a GPU-enabled high-performance computing (HPC) cluster is all-but necessary. The University of Arizona (UArizona) provides three distinct HPC clusters with varying CPU/GPU/RAM capabilities. We utilize the El Gato supercomputer cluster which boasts 131 compute nodes each with 2 Xeon E5-2650v2 8-core (Ivy Bridge) CPUs for a total of 16 cores/node running at 2.66GHz. 90 of El Gato's nodes are equipped with an NVIDIA K20X (47 of the 90 nodes are equipped with 2x GPUs) and have 256GB memory. The remaining 41 CPU-only nodes are equipped with 64GB memory.

In order to reliably deploy our learning framework across platforms and operating systems, the entire gym was containerized using Singularity [3]. Containers are independent, cross-platform software packages which incorporate any dependancies into the container image and effectuate reliably reproducible, i.e., exactly the same, operation between host platforms, i.e., hardware or software. The `gym-pybullet-drones` gym is designed and tested on Ubuntu 18.04 and therefore a container was created based on the Ubuntu 18.04 image which, despite El Gato running CentOS 6 [4], allows for training to take place on an Ubuntu operating system [5]. Singularity, however, is a Linux-only program [6] and many scientists and researchers run MacOS or Windows so Singularity was designed to "bootstrap" from a preexisting Docker [5] container as Docker supports Linux

[4] El Gato, despite being the oldest and least capable of the UArizona's three HPC clusters, was chosen because the Ocelote and Puma HPC clusters both run CentOS 7 which has a kernel too new for compatibility with Ubuntu 18.04

[5] Note that instillation of the required Python packages for the gym environment was unsuccessful within a virtual environment on the native CentOS 6 or 7 host and therefore containerization is absolutely required for our application

[6] At the time of this writing there are beta releases for MacOS

container execution on MacOS, Windows, and Linux but requires a daemon for execution (which does not easily integrate into a shared resource system such as an HPC) and does not easily utilize GPU resources. The *Shoot and Defend* environment within the `gym-pybullet-drones` gym was therefore containerized first into a Docker container [7] [8] then bootstrapped into a Singularity container which was used for all training.

3 Results

TODO: this whole section ...

4 Conclusion

TODO: this whole section ...

4.1 Future Work

TODO: maybe keep this if we get any interesting results ...

[7] This was chosen simply to test execution on a MacOS host system before deployment on the HPC

[8] The Docker container is available via connorfuhrman/gym-pybullet-drones

A Agent Rewards

A.1 Defender Agent Rewards

Value	Condition
0	The shooter moves outside it's designated box
0	The shooter crashes
-500	The shooter scores a goal
-10	The defender crashes
-10	The defender moves outside it's designated box
0	The ball goes out of bounds
5	The ball is stationary [9]
-8	The alloted time per episode is exceeded
10^{-2}	Reward to encourage level flight

A.2 Shooter Agent Rewards

Value	Condition
0	The defender moves outside it's designated box
0	The defender crashes
500	The shooter scores a goal
-10	The shooter crashes
-10	The shooter moves outside it's designated box
0	The ball goes out of bounds
-5	The ball is stationary
-8	The alloted time per episode is exceeded
10^{-2}	Reward to encourage level flight
10^{-2}	Reward to encourage the ball to move towards the goal [10]

B Ray PPO Configuration

TODO

[9] I.e., the ball was shot, did not enter the goal, and the ball's velocity is below a small threshold, e.g., 10^{-6}

[10] This reward encourages the shooter to shoot rather than not

References

[1] Adam Gleave, Michael Dennis, Neel Kant, Cody Wild, Sergey Levine, and Stuart Russell. Adversarial policies: Attacking deep reinforcement learning. *arXiv*, pages 1–16, 2019.

[2] Filip Wolski Prafulla Dhariwal Alec Radford John Schulman, Oleg Klimov. Proximal policy optimization, 2017.

[3] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12:1–20, 05 2017.

[4] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. *arXiv*, 2017.

[5] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[6] Jacopo Panerati, Hehui Zheng, SiQi Zhou, James Xu, Amanda Prorok, and Angela P. Schoellig. Learning to Fly – a Gym Environment with PyBullet Physics for Reinforcement Learning of Multi-agent Quadcopter Control. 2021.

[7] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[8] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. pages 1–12, 2017.

[9] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings*, pages 1–10, 2014.

[10] The Ray Team, 2021.

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