

Reinforcement Learning and Trustworthy Autonomy



Jieliang Luo, Sam Green, Peter Feghali, George Legrady, and Çetin Kaya Koç

Abstract Cyber-Physical Systems (CPS) possess physical and software interdependence and are typically designed by teams of mechanical, electrical, and software engineers. The interdisciplinary nature of CPS makes them difficult to design with safety guarantees. When autonomy is incorporated, design complexity and, especially, the difficulty of providing safety assurances are increased. Vision-based reinforcement learning is an increasingly popular family of machine learning algorithms that may be used to provide autonomy for CPS. Understanding how visual stimuli trigger various actions is critical for trustworthy autonomy. In this chapter we introduce reinforcement learning in the context of Microsoft's AirSim drone simulator. Specifically, we guide the reader through the necessary steps for creating a drone simulation environment suitable for experimenting with vision-based reinforcement learning. We also explore how existing vision-oriented deep learning analysis methods may be applied toward safety verification in vision-based reinforcement learning applications.

1 Introduction

Cyber-Physical Systems (CPS) are becoming increasingly powerful and complex. For example, standard passenger vehicles have millions of lines of code and tens of processors [1], and they are the product of years of planning and engineering

J. Luo (✉) · S. Green · P. Feghali · G. Legrady
University of California Santa Barbara, Santa Barbara, CA, USA
e-mail: jieliang@ucsb.edu; sam.green@cs.ucsb.edu; peterfeghali@ucsb.edu; glegrady@ucsb.edu

Ç. K. Koç
İstinye University, İstanbul, Turkey

Nanjing University of Aeronautics and Astronautics, Nanjing, China
University of California Santa Barbara, Santa Barbara, CA, USA
e-mail: cetinkoc@ucsb.edu

between diverse teams. In similar ways, both smaller CPS, like smart locks, and larger CPS, like electric power transmission systems, require teams with diverse skills. These modern systems are being augmented with full or partial autonomy. Because of the physical aspect of certain CPS, it is critical that they operate reliably and safely. However, the interdisciplinary nature of CPS makes them difficult to design with safety and security guarantees. When autonomy is incorporated, design complexity and, more importantly, the difficulty of providing safety and security assurances are increased. Yet because of the benefits of autonomy, we will continue to see its use expand.

Reinforcement learning (RL) is a family of machine learning algorithms aimed at providing autonomy, and these algorithms are being used to provide autonomous capabilities to CPS. RL has historic roots in dynamic programming and the psychological concepts of operant conditioning, or learning by trial and error. RL methods are flexible. For example, an RL policy may be developed using simulation and then transferred to a physical agent, or a policy may incorporate prior knowledge about the environment, or a policy may be trained tabula rasa. Because of the flexibility of these methods, RL with CPS is ideally suited for interdisciplinary development, with each field bringing distinct capability-enhancing contributions. On the other hand, an individual can develop basic autonomous methods with RL which may later be improved.

In mammals, vision processing accounts for a high percentage of neural activity. Likewise, in CPS, visuomotor control will be an important area of research, e.g., for self-driving cars or package delivery drones. Understanding the visual stimuli that influences behavior is critical for safe autonomy. Advanced RL methods typically process visual inputs with convolutional neural networks (CNNs). In 2012, CNNs gained popularity when the AlexNet architecture became the first CNN method to win the ImageNet competition [2]. At that time, CNNs were treated like black box functions that worked well, but it was difficult to determine why. Since then, a number of visualization algorithms have been devised to provide introspection into the behavior of a CNN. Such methods may also be applied in the context of RL in order to provide insight into the reliability of a particular CNN.

In this chapter we introduce reinforcement learning in the context of Microsoft's AirSim drone simulator. AirSim is a physics-based simulator which enables experimentation with self-driving and flight applications. We have extended the simulator to support teaching a drone how to autonomously navigate a sequence of waypoints (Fig. 1). The source code for these experiments is available at <https://github.com>.

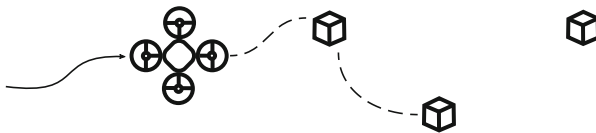


Fig. 1 In this chapter we train a drone to use its camera to perform path planning. Training is performed via reinforcement learning, and the goal is to learn vision-to-action mappings which allow the drone to collect cubes

[com/RodgerLuo/CPS-Book-Chapter](https://www.com/RodgerLuo/CPS-Book-Chapter). After an introduction to RL and AirSim, we explain how to use CNN visualization techniques to increase trust in the safety of RL-based drone control.

2 Reinforcement Learning Preliminaries

Reinforcement learning is a family of methods aimed at training an **agent** to collect rewards from an environment. At each **time step** t , the agent is given information about the **state** of its environment in the form of an observation s_t and then makes an **action** a_t . The agent's **policy** $\pi(s_t)$ is the logic which takes state observations and returns action selections. After each action the environment will return a new state observation s_{t+1} and **reward** r_{t+1} . This cycle is illustrated in Fig. 2.

The agent's policy is parameterized by the tensor θ , giving it a more explicit notation of $\pi_\theta(s_t)$. For example, in a linear model, the policy would have the form:

$$\pi_\theta(s) = \theta_1 s_1 + \theta_2 s_2 + \dots + \theta_m s_m = \theta^\top s, \quad (1)$$

where the time step t has been dropped for notation clarity and m is the number of features in the state space. In a similar manner, a neural network would be parameterized by a parameter tensor. The goal of the agent is to maximize collection of rewards from the environment. In a finite time horizon, the goal is accomplished by finding parameters θ^* which provide this maximization:

$$\theta^* = \arg \max_{\theta} \sum_{t=0}^{T-1} r(s_t, a_t), \quad (2)$$

where $T - 1$ is the number of time steps experienced and $r(s_t, a_t)$ is the environment's reward function. In many real-world cases, the reward function is not given as a closed-form expression, but must be sampled by the agent's interactions with the environment. One of the strengths of RL is its ability to learn using this experiential method.

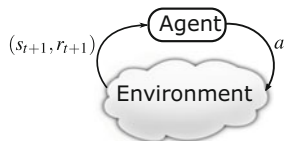


Fig. 2 In reinforcement learning, an agent interacts with an environment. At each time step, the agent receives a state and reward signal from the environment. Based on this information, the agent selects its next action

In the remainder of this section, we describe attributes of the environment in which RL agents are assumed to exist, and we introduce a common method to solve the objective given in Eq. (2). This background will prepare the reader for drone control tasks described in Sect. 3 and for approaching more efficient methods found in literature.

2.1 Markov Decision Processes

Markov Decision Processes (MDPs) are the environments which reinforcement learning was developed to solve. A major aspect of MDPs is that they have states in which an agent exists, and the outcomes of actions depend only on the current state, not on past states and actions; in this sense MDPs are **memoryless**. The memoryless property is captured in the environment’s state-transition and reward function notation:

$$\begin{aligned} p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0) &= p(s_{t+1}|s_t, a_t), \\ r(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0) &= r(s_{t+1}|s_t, a_t). \end{aligned} \quad (3)$$

The state-transition and reward functions in Eq. (3) state that function outcomes depend only on the current state and action and are independent of past states and actions.

A second major aspect of MDPs is that the state-transition and/or reward functions may be **stochastic**, which means their return values are drawn from some underlying probability distributions. In standard RL settings, these distributions must be **stationary** which means the probabilities do not shift over time. Methods exist for using RL in nonstationary environments. Investigating such advanced methods is critical for using RL in safety-critical CPS applications. For example, state-transition and reward distributions may shift from what was observed during training in the event of an anomalous situation, e.g., an emergency. In which case it could be disastrous were the agent to follow its policy decisions blindly. For that reason, consider the methods introduced in this chapter as an introduction to what is possible, but safety mechanisms would be put in place for a real-world RL+CPS application.

Within an MDP, agents may observe their current state and make actions which attempt to affect the future state. The agent’s objective is to maximize collection of rewards. An example three-state MDP¹ is given in Fig. 3. In this example, the initial state is s_0 , and the agent has two action options: a_0 and a_1 . If the agent chooses action a_0 , it is guaranteed to stay in state s_0 , denoted by $p(s_0|s_0, a_0) = 1$. If the agent chooses action a_1 , there is a 25% probability that it will transition to s_1 , denoted

¹In the notation for this example, the subscripts denote “options,” versus the usual meaning, which is time in this chapter.

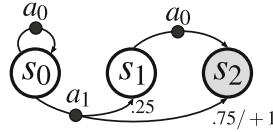


Fig. 3 Example Markov Decision Process. There are three states and two actions. Unless otherwise indicated, the state-transition probability is 1 and reward is 0. Transition from s_0 to s_2 is the most interesting with $r(s_2|s_0, a_1) = 1$ and $p(s_2|s_0, a_1) = 0.75$

by $p(s_1|s_0, a_1) = 0.25$, and a 75% probability it will transition to s_2 , denoted by $p(s_2|s_0, a_1) = 0.75$. The environment returns reward of 0 for all state transitions except for $s_0 \rightarrow s_2$, and in this case it returns $r(s_2|s_0, a_1) = 1$.

The agent's only goal is to maximize collection of rewards. In the context of Fig. 3, the agent should always select action a_1 , as it is the only action that leads to a non-zero reward. While we can see that is the solution, an agent must learn it. There are two general approaches for learning in RL: **value iteration** methods and **policy gradient** methods. Value iteration is the classical approach to RL and includes the Q-learning algorithm and its descendents. Policy gradient methods directly optimize a policy through gradient ascent. Policy gradient methods perform well in many situations, they are relatively straightforward to implement, and we focus on them in this chapter. More advanced methods combine value iteration and policy gradient methods.

2.2 Reinforce Method

In the context of reinforcement learning, our first-order objective was defined in Eq. (2) as the sum of rewards, but here we will refine it. As stated in the previous subsection, MDPs often have stochastic state-transition and reward functions; for that reason the **objective** $J(\theta)$ of the agent is actually to maximize the *expected* sum of rewards under the **trajectory probability distribution** (defined in Eq. (9)). This is achieved by discovering optimal policy parameters θ^* for the objective function $J(\theta)$:

$$\theta^* = \arg \max_{\theta} \mathbb{E}_{\tau \sim p_{\theta}} \sum_{t=0}^{T-1} r(s_t, a_t) = \arg \max_{\theta} J(\theta), \quad (4)$$

where τ is the **trajectory** of state-action pairs $(s_0, a_0, s_1, a_1, \dots, s_T, a_T)$ and p_{θ} is the trajectory probability distribution.

The REINFORCE method uses gradient ascent to adjust the policy parameters in a direction which increases $J(\theta)$ [3]. For notation convenience let

$r(\tau) = \sum_{t=0}^{T-1} r(s_t, a_t)$, and by the definition of expectation, the objective can be written as:

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta} r(\tau) = \int p_\theta(\tau) r(\tau) d\tau, \quad (5)$$

where $p_\theta(\tau)$ is the probability of a specific trajectory. Taking the gradient of $J(\theta)$ with respect to θ then gives:

$$\nabla_\theta J(\theta) = \nabla_\theta \int p_\theta(\tau) r(\tau) d\tau = \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau. \quad (6)$$

For reasons that will become clear, we recall the following identity:

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = p_\theta(\tau) \nabla_\theta \log(p_\theta(\tau)), \quad (7)$$

allowing us to rewrite Eq. (6) as:

$$\begin{aligned} \nabla_\theta J(\theta) &= \int p_\theta(\tau) \nabla_\theta \log(p_\theta(\tau)) r(\tau) d\tau, \\ &= \mathbb{E}_{\tau \sim p_\theta} \nabla_\theta \log(p_\theta(\tau)) r(\tau). \end{aligned} \quad (8)$$

We now explain why the identity in Eq. (7) was used. The probability of a sampled (i.e., experienced) trajectory τ has a probability that can be explicitly calculated only if the underlying state-transition function is known:

$$p_\theta(\tau) = p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t | s_t) p(s_{t+1} | s_t, a_t), \quad (9)$$

where $p(s_0)$ is the probability of starting the trajectory in state s_0 and is independent of θ and $\pi_\theta(a_t | s_t)$ is the probability of the selected action given the state observation s_t . To better understand the notation $\pi_\theta(a_t | s_t)$, note that the policy is **stochastic**. In other words, when the policy is given a state observation s_t , the output of $\pi_\theta(s_t)$ is a vector of probabilities derived from the *softmax* function.² In the discrete action-space environments considered here, there is one output probability per possible action. A random action is then drawn from the given probability distribution, and the probability of the selected action is denoted $\pi_\theta(a_t | s_t)$.

² $softmax(x_i | \mathbf{x}) := \frac{e^{x_i}}{\sum_{j=1}^{|\mathbf{x}|} e^{x_j}}$, where \mathbf{x} is a vector of reals.

In real-world problems, $p(s_{t+1}|s_t, a_t)$ is not known, so $p_\theta(\tau)$ would be impossible to calculate. However:

$$\begin{aligned} \log p_\theta(\tau) &= \log \left(p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \right) \\ &= \log p(s_0) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t), \end{aligned} \quad (10)$$

and replacing $\log p_\theta(\tau)$ in Eq. (8) with its expanded form gives:

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\tau \sim p_\theta} \nabla_\theta \left[\log p(s_0) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t) \right] r(\tau), \\ &= \mathbb{E}_{\tau \sim p_\theta} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) r(\tau). \end{aligned} \quad (11)$$

In this form, we are able to approximate the gradient. Recall that π_θ is a neural network (or some other differentiable function), so the gradient of its log may be calculated explicitly given each a_t and s_t over the trajectory. Also, we know the sum of rewards $r(\tau)$ for each trajectory. Finally, the outer expectation is approximated by performing N **episodes**, i.e., experiencing multiple trajectories, and then averaging the sums giving:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{n,t}|s_{n,t}) r(\tau_n). \quad (12)$$

After having obtained an approximation of the objective's gradient, we may use it to update the neural network parameters with standard stochastic gradient ascent:

$$\theta = \theta + \alpha \nabla_\theta J(\theta), \quad (13)$$

where α is the learning rate and whose appropriate value must be experimentally found.

The REINFORCE method works surprisingly well for a broad range of problems, and there are many improvements that have been made to it to increase its performance. Understanding the method presented here is a good foundation for approaching current literature. The source code we provide for this chapter at <https://github.com/RodgerLuo/CPS-Book-Chapter> uses REINFORCE.

3 Microsoft's AirSim

3.1 Overview

Developed by Microsoft Research AI in 2017, AirSim is an Unreal Engine plug-in to provide physically realistic simulations for autonomous vehicles in Unreal Engine environments [4]. The goal of AirSim is to offer an open-source platform for artificial intelligence research, especially in developing and comparing reinforcement learning algorithms.

The installation of AirSim is straightforward, and its official web page provides explicit instructions: https://github.com/Microsoft/AirSim/blob/master/docs/build_windows.md. AirSim delivers binaries and builds for Windows and for Linux. We recommend installing AirSim using builds because it gives more freedom for customizing environments in Unreal. For this chapter, we built AirSim on Windows 10 Pro version 1709. For the rest of the section, we will introduce the core features of AirSim concerning reinforcement learning and its Python APIs. We will discuss how to customize Unreal environments in the following section.

By default, AirSim has two built-in vehicle models: `multirotor` and `car`. For the rest of the chapter, we will use “drone” to indicate `multirotor` mode. Users can choose to either manually fly the drone or drive the car with a remote control or programmatically control the vehicles in C++, C#, Python, Java, etc. This section focuses on programmatic control, particularly using APIs in Python. Readers can find the official document for remote control configuration on this web page: https://github.com/Microsoft/AirSim/blob/master/docs/remote_control.md. We chose AirSim for our studies mainly because of two features: (1) the combination of AirSim and Unreal provides a more realistic training environment than other existing RL environments such as OpenAI Gym or Unity ML library, and (2) the support of software-in-loop and hardware-in-loop with popular flight controllers provides a potential smooth transition from the simulator to the real world. This chapter focuses on drone simulation in which AirSim provides four different flight modes, discussed in Table 1.

To choose a simulation mode and configure other settings, users can edit `setting.json` at `Documents\AirSim` on Windows or `~/Documents/AirSim` on Linux. The minimal configuration is as follows:

```
{
  'SettingsVersion':1.0
}
```

Listing 1 The minimal version of AirSim configuration

`SettingsVersion` instructs AirSim to load default settings when the Unreal Engine starts. It is the only item required in `setting.json` and is usually listed as the first item in the file. Any items after the `SettingsVersion` override the related parameters in the default setting.

Table 1 Drone simulation modes in AirSim

Simulation mode	Description
Computer vision	No vehicle physics and dynamics involved in this mode. Users can use keyboards or APIs to navigate and position the virtual drone. This mode is usually used for proof of concept and rapid prototyping
Simple flight	A built-in flight controller provides realistic drone-flying experience in Unreal. It doesn't require additional configurations
Hardware-in-loop ^a	The flight controller runs on physical hardware, which communicates with AirSim using the USB port. The mode is the closest scenario in comparison to flying a real drone, but requires additional setups and is usually hard to debug
Software-in-loop ^a	This mode is similar to hardware-in-loop, except the firmware runs on the computer as opposed to a separate board. Regarding the relationship to flying the drone in the real world, this mode is in-between the simple flight and hardware-in-loop

^aHardware-in-loop and software-in-loop are usually for advanced users. By far, AirSim supports PX4 flight controller and plans to support ROSFlight and Hackflight in the future

Listing 2 shows how to load the virtual drone and use the computer vision mode in the environment. `SimMode` determines whether to load the drone or car by setting the parameter to `Multirotor` or `car`, respectively. Setting `UsageScenario` to `ComputerVision` disables physical simulation, so the drone would hang in the air. Readers can find a comprehensive description about the settings of AirSim from this web page: <https://github.com/Microsoft/AirSim/blob/master/docs/settings.md>.

```
{
  'SettingsVersion': 1.0,
  'SimMode': 'Multirotor',
  'UsageScenario': 'ComputerVision'
}
```

Listing 2 A configuration of AirSim to load the drone model and activate the computer vision mode

3.2 Python APIs

In this section, we introduce two major features of the AirSim Python APIs: drone navigation and image processing.

In the APIs for drone navigation, except for computer vision mode, all simulators need to obey the rules of flying a real drone. Namely, the drone needs to be `armed` before taking off and `disarmed` after landing. All other navigation commands should wait until the drone takes off and hovers at a stable height. Table 2 shows five auto-flight modes in the Python APIs that facilitate the drone navigation process. In

Table 2 Five auto-flight modes in AirSim Python APIs

Auto-flight command	Description
armDisarm	Mandatory before taking off or after landing the drone
takeoff	Take off and ascend to default height
land	Land drone at its current position
goHome	Move drone to its take-off location, followed by land command
hover	Hover the drone at its current position

the computer vision mode, however, the drone is spawned directly in the air with no physics and dynamics and doesn't need to obey any aforementioned rules. We can consider the drone as a non-gravity block with visual inputs in the computer vision mode.

Before we discuss more sophisticated APIs to navigate the drone, it is necessary to introduce four aircraft terms: pitch, roll, yaw, and throttle. Unlike driving a car, controlling a drone is performed by making it rotate in three axes: **normal axis**, **lateral axis**, and **longitudinal axis**. Figure 4 illustrates these axes.

The normal axis, also known as the vertical axis, is perpendicular to the body of the drone and is directed toward the bottom. **Yaw** is the motion for this axis. Positive yaw moves the head of the drone to the right.

The lateral axis is directed to the right of the drone and parallel to an invisible line drawn from the left edge to the right edge. **Pitch** is the motion for this axis. Positive pitch raises the head of the drone and lowers the end.

The longitudinal axis is directed forward, parallel to the body of the drone. **Roll** is the motion for this axis. Positive roll lifts the left side of the drone and lowers the right side.

Besides the three motions, throttle controls the vertical movements of the drone. Positive throttle raises the drone vertically.

Fig. 4 An illustration of the four drone axes: roll, yaw, pitch, and throttle

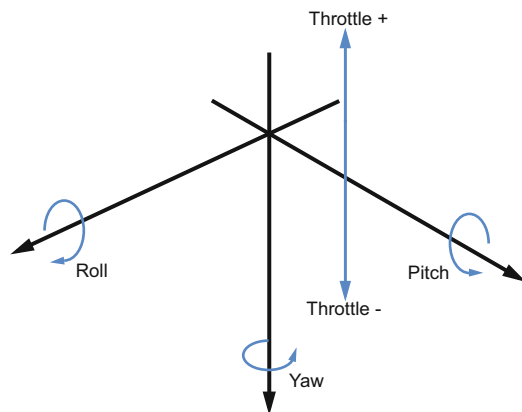


Table 3 Five common navigation methods in AirSim

Methods	Parameters
<code>moveByAngle</code>	<code>pitch, roll, z, yaw, duration</code> ^a
<code>moveByVelocity</code>	<code>vx, vy, vz, duration, drivetrain</code> ^b , <code>yaw_mode</code> ^c
<code>moveByPath</code>	<code>path, velocity, max_wait_seconds</code> ^d , <code>drivetrain, yaw_mode, lookahead</code> ^c , <code>adaptive_lookahead</code> ^c
<code>moveToPosition</code>	<code>x, y, z, velocity, max_wait_seconds, drivetrain, yaw_mode, lookahead, adaptive_lookahead</code>
<code>moveByManual</code>	<code>vx_max, vy_max, z_min, duration, drivetrain, yaw_mode</code>

^aMethods taking the `duration` parameter are usually followed by a `time.sleep` function, because the methods release control immediately. Without the `time.sleep` function, the methods would not have enough time to finish

^b`Drivetrain` has two modes: `ForwardOnly` and `MaxDegreeOfFreedom`. `ForwardOnly` keeps the drone's front always pointing in the direction of travel, while `MaxDegreeOfFreedom` doesn't have such restriction

^c`Yaw_mode` has two fields: `is_rate` and `yaw_or_rate`. Usually, it is set to `yaw_mode` (`false, 0`) to keep the yaw constant

^dIn comparison to `duration`, `max_wait_seconds` blocks the amount of time, in order to make sure the action completes

^eMost of the time, we set `lookahead=-1` and `adaptive_lookahead= 0` to let the drone auto-decide the path

AirSim Python APIs provide five commands to navigate the drone by taking different physical parameters as inputs. Some commands also have simpler versions that take less parameters. Table 3 lists the five commands and the parameters each command takes.

A necessary component for vision-based deep reinforcement learning is acquiring visual inputs from the drone. AirSim provides seven image types: `scene`, `depth planner`, `depth perspective`, `depth vis`, `disparity normalized`, `segmentation`, and `surface normals`. We used the `scene` type to get images for our studies. The following line of code demonstrates how to acquire raw image data from the AirSim Unreal environment:

```
rawData = self.client.simGetImages([ImageRequest(1,
    AirSimImageType.Scene, False, False)])
```

Listing 3 A command using AirSim API to acquire image data. The command takes four variables: `camera_id`, activating a particular camera on the drone; `image_type`, choosing one of the seven image types mentioned above; `pixels_as_float`, determining whether the data type of pixels is float or integer; and `compress`, determining whether the acquired image is compressed or not. The above command chooses the front-center camera, uses `scene` image type, encodes the pixel values as integers, and keeps the image uncompressed

AirSim embedded five cameras on the drone: three of them are in the front, one is in the back, and one is on the bottom of the drone. The three front cameras are located on the right, center, and left, respectively. Camera No.1 refers to the front-center camera, which is the camera we used for our reinforcement learning tasks.

The orientations and positions of the cameras are customizable in Unreal. Besides APIs for drone navigation and image processing, other useful APIs consist of `Collision`, `Reset`, `SimGetObjectPos`, and `ApiControlEnabled`. More details about the AirSim Python APIs can be accessed from: (1) <https://github.com/Microsoft/AirSim/blob/master/docs/apis.md> and (2) <https://github.com/Microsoft/AirSim/blob/master/PythonClient/AirSimClient.py>

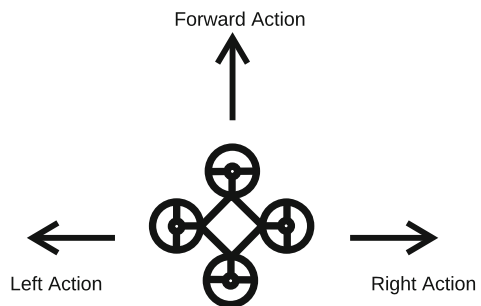
4 Reinforcement Learning in AirSim

In this section we give a detailed introduction of how to implement RL using the AirSim extensions detailed in Sect. 3. Our task is to train the drone to automatically “collect” cubes in the environment, which consists of (1) an Unreal level for dynamically positioning the cubes and monitoring the interactions between the drone and the cubes and (2) a Python library used as an intermediate to receive and send data between the Unreal level and RL algorithms. In terms of the navigation, we simplify the drone’s movements by constraining it to three actions: left action, right action, and forward action. Figure 5 demonstrates the drone’s movements used in the cube collection task.

4.1 Unreal Dynamic Environment

Our Unreal environment was designed specifically for the cube collection task. The environment fulfills the following requirements: (1) for each episode in the RL training, the cubes should be randomly positioned in a restricted area; (2) the cubes should vanish when the drone hits them or bypasses them. We don’t want the drone to accidentally collide with the cubes by moving to the left or right when no cube is visible in the drone’s camera view, because it will result in

Fig. 5 Drone’s movements were simplified to fulfill the requirement for our cube collection task



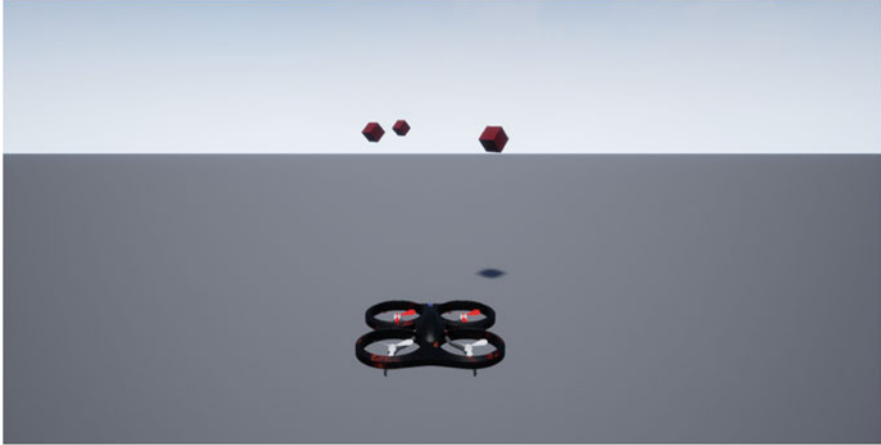


Fig. 6 A screenshot from Unreal, immediately after the drone spawned in the environment

false rewards³; (3) the environment should have invisible boundaries on the front, left, and right to constrain the drone’s movements; (4) the number of the cubes should be dynamic, and the invisible boundaries should be automatically adjusted according to the number of the cubes. We adapted the Block Unreal environment created by AirSim (https://github.com/Microsoft/AirSim/blob/master/docs/unreal_blocks.md) and removed the unnecessary environmental details to create a minimal Unreal environment for our task. Users can also use AirSim in any other Unreal environments. A documentation of how to install AirSim in Unreal environments is here: https://github.com/Microsoft/AirSim/blob/master/docs/unreal_custenv.md.

Generally speaking, Unreal has two modes: (1) *edit mode*, for editing the environment, and (2) *play mode*, for testing the environment. The drone only appears in *play mode* and spawns at the origin in the environment. To change the spawning position of the drone, users need to modify the position of *PlayerStarter*, an object corresponding to the drone’s position. In *play mode*, users can see different camera views by pressing 1, 2, or 3. Pressing F1 can activate a detailed help menu. Figure 6 shows an initial state after spawning the drone in the environment.

The requirements mentioned above were implemented in Unreal’s own visual programming language, called **blueprints**. We created two blueprints to manage the drone-cube interactions in the environment. The first blueprint is applied to all the cubes so that each cube would be removed immediately after the drone hits or passes it. The second blueprint is to randomly spawn all cubes in a constrained space, once the drone is reset to its origin. Given the visual complexity of the two blueprints, we present the high-level abstract structure of the two blueprints in Algorithm 1 and

³This is because our RL policy is memoryless. If an RNN or LSTM were used, instead of a vanilla CNN, the policy gains memory, and it would be possible for the drone to learn to bump into cubes which it can no longer see.

Algorithm 1: Apply to all the cubes in the environment that each cube would be removed immediately after the drone hits or passes it

```

1 while env is running do
2   if collision then
3     remove the cube
4   else if drone.xPosition > cube.xPosition then
5     remove the cube
6 end

```

Algorithm 2: Randomly reposition all the cubes in a constrained space, once the drone is reset to its origin

```

1 while env is running do
2   if drone.position == origin.position then
3     remove_all_cubes()
4     for cube_index in range(num_cubes) do
5       spawn.xPos = cube_index * cube_distance + starting_pos
6       spawn.yPos, spawn.zPos = random(-bound, bound)
7       spawn_cube(spawn.position)
8     end
9   end
10 end

```

Algorithm 2, respectively. Readers can download the blueprint files from our source code repository.

Usually, blueprints and models are located in `Asset` folder in Unreal. However, since the drone is imported from AirSim rather than created in Unreal, it doesn't exist in the `Asset` folder. To modify the drone's blueprint, we need to switch to play mode in which an object called `BP_FlyingPawn` highlighted in gold will appear in the `World Outliner` window. The blueprint exists at the `Event Graph` menu after clicking `Edit` → `BP_FlyingPawn`.

We also expanded the field of view of the original camera on the drone to better keep cubes in the drone's view. To modify the cameras' attributes, we open the editor of `BP_PIPCCamera`, by using the same method of accessing the drone's blueprint. The field of view can be found on the `Details` window, under the `Projection` menu. We changed the field of view from 90.0 to 135.0.

4.2 Python Environment Library

Our Python environment library serves as an intermediate to send and receive data between the Unreal environment and our reinforcement learning algorithms. It is designed explicitly for the AimSim Python APIs and is compatible with any automatic differentiation framework, such as TensorFlow or PyTorch.

Table 4 A high-level description of the Python environment library

Function	Description
<code>__init__</code>	Setup connections to the AirSim Python APIs. Parse the parameters of the drone and the environment
<code>reset</code>	Position the drone to the origin. Reset parameters
<code>step</code>	Send the updated drone's position to AirSim. Return state, reward, done, and info ^a
<code>get_image</code>	Convert and process raw image data from AirSim to NumPy arrays which are compatible with our convolutional neural network

^aThe details of the four parameters will be introduced in the following paragraph

The library has four major functions: (1) initialize the environment; (2) reset the environment; (3) for each step, navigate the drone based on the received action and return the essential training data, such as state, reward, and done; and (4) process raw image data from Unreal to make them compatible with the deep neural network. Each function can be called independently during the training process. Table 4 shows a high-level description of the library.

The core part in the library is the `step` function of which the logic is presented in Algorithm 3. The function is responsible for returning four essential values for our RL training:

Algorithm 3: *step* function, a core function in our Python Environment Library that takes action as the input and returns state, reward, done, and number of collected cubes

```

Input: action
Output: state, reward, done, number of collected cubes
1 the drone's current position = GetPosition();
2 distance = Move(action);
3 SetPosition(the drone's current position + distance);
4 state = GetImage();
5 collision info = GetCollisionInfo();
6 if collided then
7   |   reward = 1;
8   |   num of collected cubes += 1;
9 end
10 if the drone out of the boundary then
11   |   done = 1
12 else if num of collected cubes == num of cubes placed in the env then
13   |   done = 1
14 else if num of steps == threshold then
15   |   done = 1
16 else
17   |   done = 0
18 end

```

- `state`: represents the drone’s observation of the environment. In our case, the states are the images captured by the camera on the drone.
- `reward`: the amount of rewards acquired by the previous action. We set the reward to 1 for collecting a cube and 0 for any other actions.
- `done`: whether the episode ends. In our case, one episode ends when the drone collects all the cubes in the environment or moves outside of the defined boundaries.
- `info`: diagnostic information useful for debugging. We are particularly interested in the number of cubes collected by the drone.

4.3 REINFORCE Method in AirSim

We now adapt the concepts of the REINFORCE method from Sect. 2.2 to the cube collection task. A simple convolutional neural network is used to represent the policy. We will refine the objective (from Eq. (2)) of finding network parameters which maximize the expected sum of rewards across all time steps in the episode:

$$\theta^* = \arg \max_{\theta} \mathbb{E}_{\tau \sim p_{\theta}} \sum_{t=0}^{T-1} r(s_t, a_t).$$

In the cube collection task, a reward of 1 is provided by the environment each time a cube is collected by the drone, and 0 reward when no cube is collected, so the objective is to collect as many cubes as possible in each episode (i.e., during time step $t = 0 \dots T - 1$, where $T - 1$ is the step when the last cube is collected or the drone has gone out of bounds). In the context of REINFORCE, this objective was discovered by taking its gradient (from Eq. (12)):

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_{n,t} | s_{n,t}) r(\tau_n),$$

and then updating the neural network parameters based on the gradient. Recall that π_{θ} is the neural network, and, in the drone collection tasks, $\pi_{\theta}(a_{n,t} | s_{n,t})$ is the output probability⁴ of going left, right, or forward, given input pixels from the drone’s camera.

The weakness of Eq. (12) for our context is that the rewards are sparse, because there are only three cubes total to collect in each trajectory. If $r(\tau_n)$ is used directly as the reinforcing signal, then *entire* trajectory probabilities are increased or are unchanged. This results in high variance in performance between each episode. An approach to get faster results in the cube collection task is to “smooth” the attribution

⁴Softmax of the network’s logits.

Algorithm 4: REINFORCE algorithm in the context of the AirSim cube collection task

Input: Old policy parameters θ , learning rate α , tuple of (observations s , actions a , rewards r) from last cube collection episode

Output: Updated policy θ

- 1 Apply Eq. (14) to obtain discounted rewards g from a
 - 2 Normalize g
 - 3 Set $sum\ of\ grads = 0$
 - 4 **for** $t = 0 \dots T - 1$ **do**
 - 5 \mid $sum\ of\ grads = sum\ of\ grads + g_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$
 - 6 **end**
 - 7 $\theta = \theta + \alpha sum\ of\ grads$
 - 8 **return** θ
-

of rewards from later stages to earlier stages by applying a **discounted return** to the gradient at each time step. Discounted return is defined as:

$$g_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \dots + \gamma^{t+T} r_{t+T} = \sum_{k=0}^{T-1} \gamma^k r_{t+k+1}, \quad (14)$$

where $\gamma \in [0, 1]$ is the discount rate. The resulting g vector of Eq. (14) is also normalized⁵ in the cube collection task. Using g we update Eq. (12) to give:

$$\nabla_{\theta} J(\theta) \approx \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) g_t, \quad (15)$$

where we are only collecting a single trajectory between applications of gradient ascent. There are better approaches than using the discounted return, and our source code example is parameterized to allow experimentation between other reward function alternatives.

We summarize the use of the REINFORCE method in the context of our AirSim cube collection task formally in Algorithm 4. This algorithm is implemented in the provided source code.

5 Increased Trustworthiness Through Visualization

Machine vision algorithms have changed radically in the era of artificial intelligence. For example, prior to the age of AI, if we asked a machine to look for a face in an image, we knew what the machine would look at. With the development of deep

⁵Normalization is defined as $g \leftarrow (g - \mu(g))/\sigma(g)$, where scalar operations are applied element-wise to the vector.

neural networks, we significantly increased machine vision performance, surpassing that of traditional methods, but we initially had limited understanding regarding which part of the input triggered the machine to come to its conclusion. Such “black-box” performance is tolerated for some applications, but, in order to develop trust in AI’s decision-making process, it is important for engineers to understand the mechanism behind the machine’s decision-making.

This section will introduce three common visualization techniques: T-SNE, action visualization, and attribution visualization. We adapt these visualization methods for applications of understanding deep reinforcement learning. As a case study, we use imagery and trained policies from the cube collecting example which was detailed in previous sections. For each visualization technique, we analyze three policies that have been trained to collect cubes using the REINFORCE algorithm. The policies have the following descriptions:

- **Good policy:** collects most cubes in the environment.
- **Poor policy:** is unable to collect any cubes except randomly
- **Right-and-forward policy:** is only able to collect cubes directly in front or to the right. Ignores all cubes on the left

5.1 *t-SNE*

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a dimensionality reduction algorithm developed by Laurens van der Maaten and Geoffrey Hinton [5]. It is well suited for visualizing high-dimensional datasets in 2D or 3D spaces. Specifically, the visualization maps each high-dimensional data point to a two- or three-dimensional space in a way that similar data points are nearby and dissimilar ones are distant. The technique was adapted later to reveal structure of images at many different scales. The most recent application of visualizing images via t-SNE is to use a convolutional neural network to extract features from images, input the extracted features of each image to t-SNE to get the “position” of each image, and then arrange the images on a 2D or 3D space based on the given positions.

Formally, t-SNE measures the pair-wise distance between all points in a high-dimensional dataset. It then projects the high-dimensional dataset to a 2D or 3D space and adjusts the points in the projection to have pair-wise distances which are similar to the high-dimensional dataset. The pair-wise distance between points x_i and x_j in dataset \mathbf{x} is defined as the probability that x_j would be chosen from a Gaussian centered at x_i :

$$p_{j|i} = \frac{e^{(-\|x_i - x_j\|^2 / 2\sigma_i^2)}}{\sum_{k \neq i} e^{(-\|x_i - x_k\|^2 / 2\sigma_i^2)}}, \quad (16)$$

where σ_i is the variance of a Gaussian centered on data point x_i . Using a similar metric, the distances between low-dimensional points y_i and y_j are measured as:

$$q_{j|i} = \frac{e^{(-\|y_i - y_j\|^2)}}{\sum_{k \neq i} e^{(-\|y_i - y_k\|^2)}}. \quad (17)$$

For each low-dimension point y_i , the similarity of the high-dimensional and low-dimensional dataset is measured by taking the sum of the Kullback-Leibler divergences between $p_{*|i}$ and $q_{*|i}$. A cost function is defined by this sum, and then the projection of point y_i is adjusted via gradient descent. By iteratively adjusting each low-dimension point in this way, the low-dimension dataset takes on distance characteristics of the high-dimensional dataset.

In this subsection, we use t-SNE to examine the representations learned from the cube collection task to have a visual overview of the three trained policies. To distinguish the three different actions (forward, right, and left) triggered by the visual inputs, we tinted each visual input based on its predicted action: red indicates forward, green indicates left, and blue indicates right.

Figure 7 shows the two-dimensional t-SNE embedding of the visual inputs when using the poor policy (left) and the right-and-forward policy (right). The t-SNE visualization provides insight into the policy's quality. In the poor policy, e.g., the drone only moved to the left from the beginning to the end of each episode, regardless of the visual inputs. In the right-and-forward policy, the majority of the

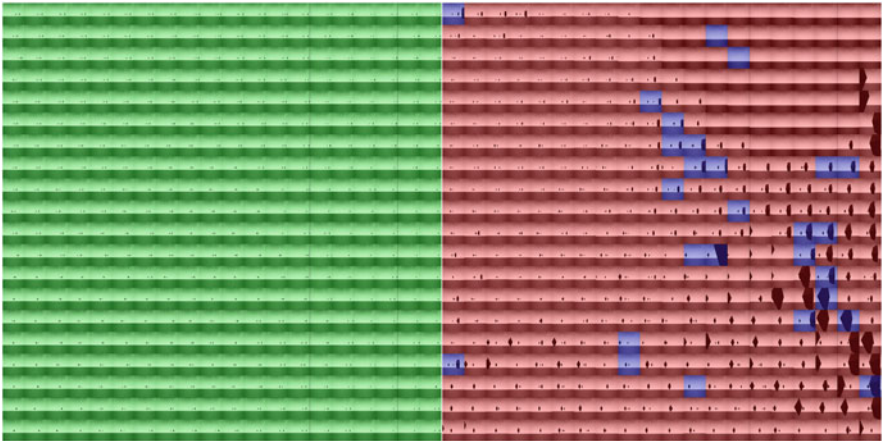


Fig. 7 Two-dimensional t-SNE embedding of visual inputs obtained by applying the poor policy (left) and right-and-forward policy (right) to the cube collection task. The left shows 400 inputs from 15 episodes from the poor policy; the right presents 400 inputs from 8 episodes from the right-and-forward policy. Tinted colors indicate the predicted actions: red, forward; green, left; and blue, right. It can be seen that the poor policy always chooses the left action (green), no matter the input. The right-and-forward policy usually chooses forward (red) and will occasionally go right (blue)

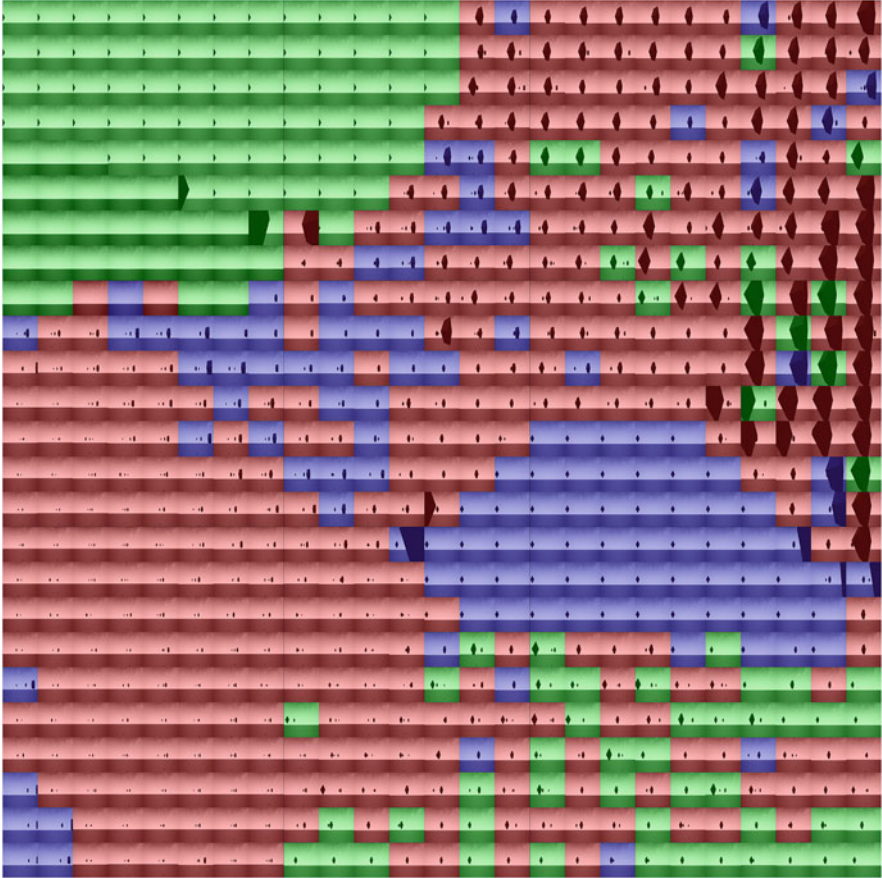


Fig. 8 Two-dimensional t-SNE embedding of 625 visual inputs from 7 episodes of applying the good policy to the cube collection task. Tinted colors indicate the predicted actions: red, forward; green, left; and blue, right. The large red cluster makes sense, because the policy should choose forward in those images. Furthermore, most of the scattered blue and green make sense. But the larger blue cluster highlights a weakness in the so-called good policy, as the policy should go left in such settings, but instead chooses right

moves are forward, and occasionally the policy chose to move to the right. As t-SNE arranges images based on their visual similarity, the scattered blue images indicate that something is wrong with the policy. Images that trigger the same behavior should logically be placed together by t-SNE.

By observing the t-SNE visualization of the so-called good policy in Fig. 8, we see that cubes in the center of the image are likely to trigger the forward action and cubes in the left or right of the images are likely to trigger the left or the right action, respectively. By examining Fig. 8, we also see that t-SNE may be used as an efficient tool to detect flaws in RL policies, even if the policies have good performance, like

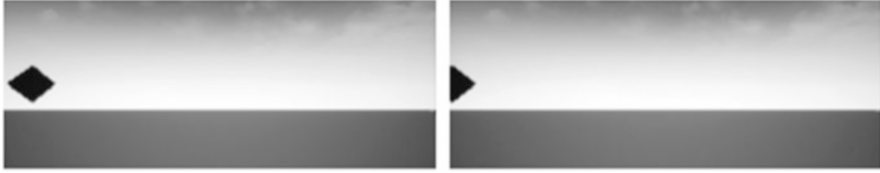


Fig. 9 These two visually similar images triggered opposite actions (left and right) in the so-called good policy. This indicates a flaw in the trained policy. t-SNE enables the RL policy developer to quickly spot such anomalies

this one. For example, in Fig. 8 the green area on the upper left and the blue area on the lower right both contain cubes located on the left edge of the image; however it can be seen that those images triggered two opposite actions, which is against the intuition of a good policy for the cube collection task. After examining the raw data, we found the drone stuck in such situations by moving back and forth between left and right. Figure 9 highlights the two individual visual inputs causing the opposite actions.

5.2 Action Visualization

Action visualization methods generate visual inputs which would activate a particular action in a *trained* policy network. This approach allows for a high level of human comprehension about the behavior of a network, rather than treating the network like a black box model. For our specific action visualization approach, we use the **Class Model Visualization** (CMV) technique introduced in [6]. CMV generates inputs which will trigger any specific output class in a trained convolutional neural network.

In this subsection we are interested in understanding *what* visual input causes specific actions to be selected by the RL policy of interest. In our study, we generate inputs to optimize for the three action cases: left, right, and forward. After choosing an action to optimize for, a uniformly random image is generated.⁶ This initially random image is then evaluated by the trained policy network. The output probability for the desired action is then increased through back-propagation, where the gradient is calculated with respect to the image pixels. We repeat this process of forward and back-propagation, until the action probability is maximized.

Formally, we let a represent the action for which we want to generate an input image to trigger; s is the input image which will be optimized such that the action probability is maximized. We let $\pi_{\theta}(a|s)$ represent the probability of taking action

⁶For the cube collection task used in this chapter, we used a simple CNN with a grayscale image as input, so we generate grayscale images for action visualization.



Fig. 10 Action visualizations for the good policy. The drone learns that when the camera is occluded (dark spots), it should move in the direction of the occlusion. (a) Left action. (b) Forward action. (c) Right action

$a \in \{\text{left, right, forward}\}$, given the image s . The goal then is to solve the following optimization problem:

$$s^* = \arg \max_s \pi(a|s). \quad (18)$$

In practice, the optimal image s^* is found using gradient ascent by an automatic differentiation tool like TensorFlow or PyTorch.

Note that CMV differs from the normal application of back-propagation, which typically considers the input (s in this case) to be fixed and instead finds neural network parameters θ which optimize the objective function. In the case of CMV, parameters are locked after policy training, and it is instead s which is optimized.

Generating action visualizations is currently an art, and, unfortunately, if the basic objective derived from Eq. (18) is used, an unsatisfactory optimal image s^* may be generated. However, there are a number of refinements that generate more meaningful images using CMV. If images are unsatisfactory, one refinement is to preprocess the current input image s by subtracting the mean and standard deviation of the training set images from s between each iteration.⁷ Another refinement is to blur s between iterations. Another refinement is to use large learning rates during optimization, e.g., 20. See [7] and [6] for more optimizations.

Using CMV, we generated action visualizations shown in Figs. 10, 11, and 12. These visualizations illustrate the inputs which would maximize probabilities for selecting the three possible drone actions by the good policy, poor policy, and the right-and-forward policy. If we choose a single random image s and iterate on it for an extended period of time using the methods described above, we will obtain an input image which will trigger the desired action in the policy of interest.

The good policy's action visualization in Fig. 10 clearly explains what the network is looking for. The bias toward the left, forward, or right is apparent in each image, depending on the position of the cube. Specifically, if a cube blocks the view of the camera on the left, then the policy will most likely choose the left action. Similarly for the forward and right actions, note that horizon is somewhat visible (near the center) in the left and forward action visualizations. It is also interesting to

⁷In this case, the training set consists of the set of images captured by the drone during its episodes.



Fig. 11 Action visualizations for the poor policy. The forward action visualization shows that only the forward action is most probable. All actions have roughly equivalent action visualizations, indicated by the “murky” figures. (a) Left action. (b) Forward action. (c) Right action

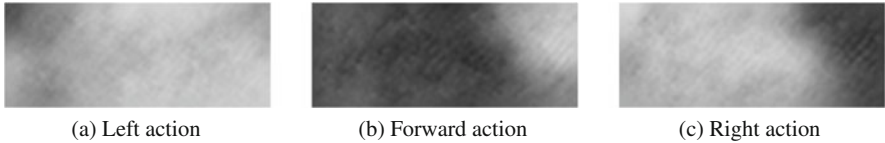


Fig. 12 Action visualizations for the right-and-forward policy. In (a) we observe that only an empty field of view (which should never happen in our simulation) would cause the drone to move left. In (b) the forward action visualization is also incorrect (see Fig. 10 for a correct version) and objects on the left will trigger a forward action. In (c) the right action has the expected action visualization

note that the images are not quite symmetric (the left action visualization does not look like the right action visualization).

Action visualizations of the poor policy help explain why the drone performs so poorly. Figure 11a–c illustrates that the actions of the poor policy are equally triggered by noise. The policy used to generate the images in Fig. 11 was unable to collect cubes in the simulation.

Action visualizations for the right-and-forward policy are given in Fig. 12. At a high level, we can see that only the right action visualization makes sense. That is, when the camera is occluded on the right, the policy will choose to move to the right. The left action visualization shows that there is a small response to camera occlusion on the left, but the forward action visualization dominates the left action. These visualizations explain why the policy collects 100% of the cubes that occur on the right side of the drone, but no cubes on the left side.

The degenerate right-and-forward policy is especially insightful and highlights one of the challenges in reinforcement learning. If the learning rate (α in Eq. (13)) is too high, the policy might finalize its decision-making process based on early experience. In this case, the drone experienced an early success by moving right and forward almost exclusively during early experiences. Combined with a learning rate that was too high, this early success led to a catastrophic elimination of left action probabilities. The remedy for this was to lower the learning rate of the policy updates and retrain the policy.

The ability to visualize and understand the desired inputs for any individual policy action is a useful tool for verification and debugging of policies. The downside to the Class Model Visualization technique presented here is the number of hyperparameters which must be manually tuned.

We now move to a technique which is able to highlight areas of an experienced input image responsible for triggering specific actions.

5.3 Attribution Visualization

Attribution visualization techniques highlight regions in an input which were most responsible for a particular action in a CNN-based policy. In this subsection we use an attribution visualization technique called **Gradient-Weighted Class Activation Mapping** (Grad-CAM) [8].

CNNs are particularly well suited for attribution visualization, because they maintain spatial structure of the input as it flows through the network; this is why we can extract meaning from the last layer. A **feature map** is the output of a convolutional layer after it has passed through a nonlinearity function (e.g., ReLU⁸). Feature maps typically have many channels, and the goal of Grad-CAM is to find which channels contribute the most to an action taken. Grad-CAM achieves that goal by calculating the average derivative of the policy network, given a specific action a and input image s , with respect to the feature map of interest⁹:

$$\alpha_k = \frac{1}{Z} \sum_i \sum_j \frac{\partial \pi_\theta(a|s)}{\partial A_{ij}^k}, \quad (19)$$

where A is the feature map of our target convolutional layer, A^k is the channel k of A , A_{ij}^k is the neuron at position i, j , and $Z = i \times j$. α_k is known as the **importance weight** for feature map channel k .

To help clarify Eq. (19), consider that each partial derivative $\partial \pi_\theta(a|s) / \partial A_{ij}^k$ gives the change in the probability of the desired action, with respect to activation i, j . Summing over all i, j in a channel gives the total change in the probability with respect to all of the channel activations. Finally, α_k is the average derivative of the feature map channel. Feature maps typically have many channels, denoted by K , and a unique α_k is calculated for each one.

Once importance weights $\alpha_1, \alpha_2, \dots, \alpha_K$ are known, they may be used to linearly combine feature map channels 1 through K , giving a “class activation map”:

$$\text{Grad} - \text{CAM} = \text{ReLU}\left(\sum_k \alpha_k A^k\right), \quad (20)$$

where ReLU is being used to filter for derivatives with a positive effect.

⁸ReLU stands for rectified linear unit and is defined as $\text{ReLU}(x) = \max(0, x)$.

⁹When using Grad-CAM, a and s are sampled from the policy and environment, while the policy controlled the drone, whereas with CMV (presented in the previous subsection) s was generated by the method and a was specified.

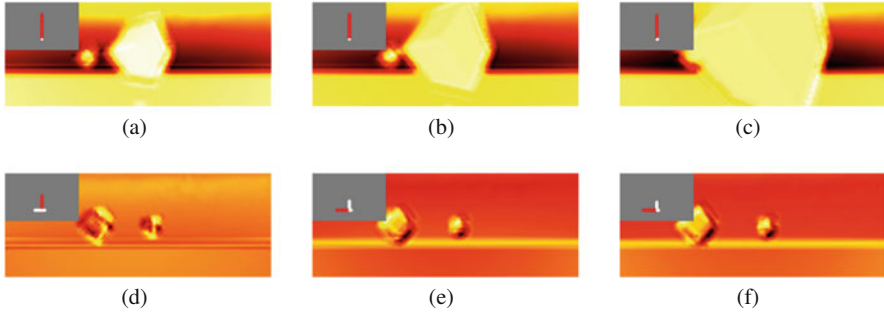


Fig. 13 Attribution visualizations for a sequence of observation-action pairs from the good policy. The \perp shape in the gray region indicates action probabilities (by the lengths) and the action that was chosen (colored red). Brighter areas indicate regions most responsible for actions

In the remainder of this section, we use Grad-CAM to create attribution visualizations for the left, forward, and right actions. In addition, the action visualization created the good policy, poor policy, and right-and-forward policy. In each example, we have six images which were captured while the drone performed in the simulator. Each visualization shown below also has an indicator which shows three bars: one for left, forward, and right, where the length of each bar gives the action probability output from the policy. The action indicated by the red bar is the action for which attribution visualization is generated in each image.

In Fig. 13, we first consider attribution visualizations for the good policy. First consider Fig. 13a. In this image, we see that the left action has the highest probability. But because the forward action was chosen, it is colored red. Grad-CAM was then used to visualize exactly what in the input image caused the forward action to have the probability that it had. The bright spots in the image give that information. We can see that the cube is very bright, which indicates that it had a high influence on choosing the forward movement. Figure 13f is also worth considering. In this case the forward action was the only action with a high probability, and the entire cube is bright, indicating its responsibility for the action. Also note in Fig. 13f how the ground is bright, which indicates that our policy has learned not just to look for cubes, but also pays attention to other aspects of the environment.

In Fig. 14 we see visualizations for the poor policy. In that figure, observe that all action probabilities are roughly the same, as indicated by the \perp shape, regardless of the position of the drone relative to the cubes. For example, in Fig. 14f, the action probability for left is slightly larger than that for right, even though the cube is on the right. For a rational probability for that observation, consider Fig. 13c, where the left action probability dominates other possibilities.

Figure 15 provides attribution visualizations for the right-and-forward policy. In the top row, we see a sequence (a–c) where the policy should be predicting left actions, but it does not. The policy is blind to objects on the left. In the bottom row, sequence (d–f), the policy does respond in an expected manner to objects to the front

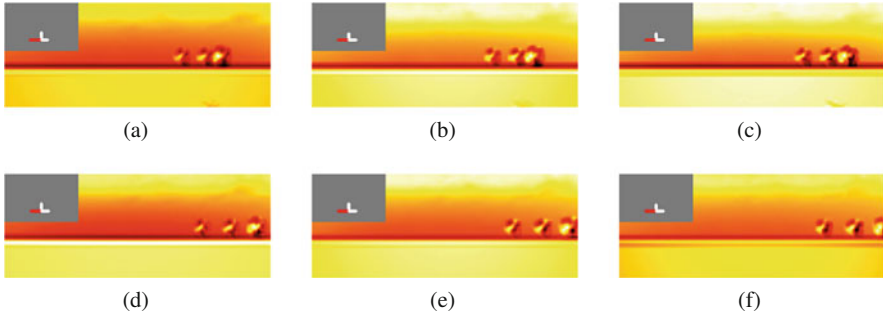


Fig. 14 Attribution visualizations for a sequence of observation-action pairs from the poor policy. Highlighting varies randomly from frame to frame, indicating that the policy has not learned to pay attention to the correct features

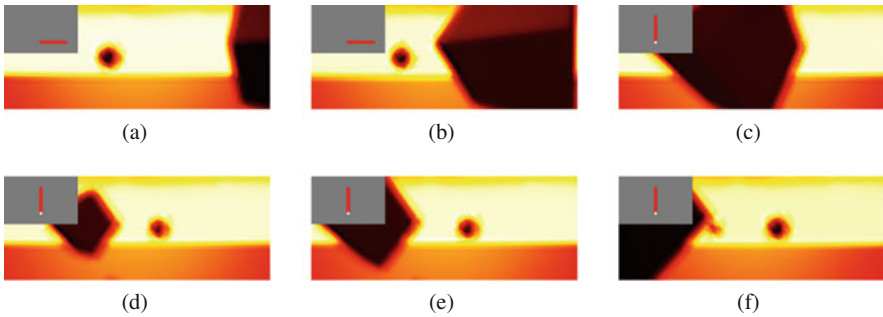


Fig. 15 Attribution visualizations for a sequence of observation-action pairs from the degenerate right-and-forward policy. The policy is not paying any attention to the cube, as it is black. All decisions are made based on the view of the horizon

and right, but it is not the object that triggers the action, but only the shape of the horizon.

6 Conclusion

In this chapter we have presented how to use Microsoft’s drone simulation environment and reinforcement learning to train a drone to navigate to and “collect” cubes which are scattered in front of it. In addition, we showed how to use existing deep neural network visualization techniques to understand the reinforcement learning-derived control policies. Because they can be improved and extended, the methods introduced here are a good starting point for multidisciplinary teams aiming to apply reinforcement learning to Cyber-Physical Systems.

Specifically, this chapter may be used by academic or industrial engineering teams as an entry point into the field of vision-based deep reinforcement learning.

In this chapter, a basic reinforcement learning algorithm was introduced, and it may be extended in many ways. For example, if a team has the ability to build optimal control policies, then more advanced reinforcement learning methods may be used. Similarly, if a team has the ability to build physical drones, then the policies learned in simulation may be transferred to the physical drone. The avenues for enhancing what was presented here are limited only by the diversity of the team.

The source code for the cube collection environment and solution is available at <https://github.com/RodgerLuo/CPS-Book-Chapter>.

References

1. R.N Charette, This car runs on code. *IEEE Spectr.* **46**(3), 3 (2009)
2. A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in *Advances in Neural Information Processing Systems* (2012), pp. 1097–1105
3. R.J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, in *Reinforcement Learning* (Springer, Berlin, 1992), pp. 5–32
4. S. Shah, D. Dey, C. Lovett, A. Kapoor, Airsim: high-fidelity visual and physical simulation for autonomous vehicles, in *Field and Service Robotics* (2017)
5. L. van der Maaten, G. Hinton, Visualizing data using t-SNE. *J. Mach. Learn. Res.* **9**, 2579–2605 (2008)
6. K. Simonyan, A. Vedaldi, A. Zisserman, Deep inside convolutional networks: visualising image classification models and saliency maps. Preprint, arXiv:1312.6034 (2013)
7. C. Olah, A. Mordvintsev, L. Schubert, Feature visualization, in *Distill* (2017)
8. R.R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, D. Batra, Grad-CAM: visual explanations from deep networks via gradient-based localization, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017), pp. 618–626