Air Learning: An AI Research Platform for Algorithm-Hardware Benchmarking of Autonomous Aerial Robots

Srivatsan Krishnan[†], Behzad Boroujerdian[‡], William Fu[†], Aleksandra Faust[∓], and Vijay Janapa Reddi[†‡]

[†] Harvard University, [‡] The University of Texas at Austin, [∓] Robotics at Google

***Abstract*—We introduce Air Learning, an AI research platform for benchmarking algorithm-hardware performance and energy efficiency trade-offs. We focus in particular on deep reinforcement learning (RL) interactions in autonomous unmanned aerial vehicles (UAVs). Equipped with a random environment generator, AirLearning exposes an UAV to a diverse set of challenging scenarios. Users can specify a task, train different deep RL policies and evaluate their performance and energy efficiency on a variety of hardware platforms. To show how Air Learning can be used, we seed it with Deep Q Networks (DQN) and Proximal Policy Optimization (PPO) to solve a point-to-point obstacle avoidance task in three different environments, generated using our configurable environment generator. We train the two algorithms using curriculum learning and non-curriculum-learning. Air Learning assesses the trained policies' performance, under a variety of quality-of-flight (QoF) metrics, such as the energy consumed, endurance and the average trajectory length, on resource-constrained embedded platforms like a Raspberry Pi. We find that the trajectories on an embedded Ras-Pi are vastly different from those predicted on a high-end desktop system, resulting in up to 79.43% longer trajectories in one of the environments. To understand the source of such discrepancies, we use Air Learning to artificially degrade high-end desktop performance to mimic what happens on a low-end embedded system. QoF metrics with hardware-in-the-loop characterize those differences and expose how the choice of onboard compute affects the aerial robot's performance. We also conduct reliability studies to demonstrate how Air Learning can help understand how sensor failures affect the learned policies. All put together, Air Learning enables a broad class of deep RL studies on UAVs. More information and source code for the Air Learning project can be found online here: `http://bit.ly/2JNAVb6`.**

I. Introduction

Unmanned Aerial Vehicles (UAVs) have shown great promise in recent years across a wide variety of robotics applications, such as search and rescue [1], package delivery [2], [3], construction inspection [4], and others. However, a key challenge remaining in the development of UAVs is autonomy.

In recent years, end-to-end learning based on Deep Reinforcement Learning (DRL) has been showing promising results in domains like sensory-motor control for cars [5], indoor robots [6], as well as UAVs [7], [8]. Deep RL's ability to adapt and learn with minimum apriori knowledge makes them attractive for use in complex systems [9]. In DRL, an agent learns a policy that directly maps the robot's input sensor data (such as RGB-D or IMU data) to output actions (such as the direction of movement or linear and angular velocities). The learned policy is approximated with a deep neural network that maximizes the discounted return value.

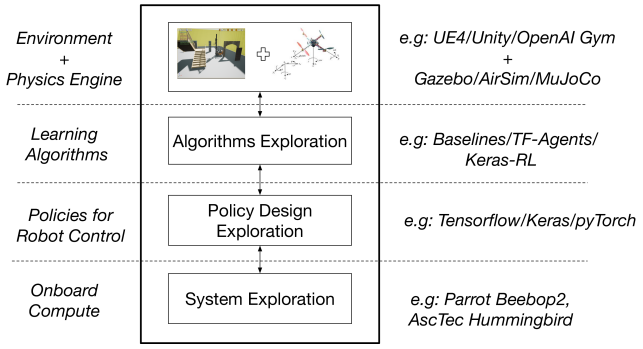


Fig. 1: Aerial robotics is a cross-layer, interdisciplinary field. Designing an autonomous aerial robot to perform a task involves interactions between various boundaries, spanning from environment modeling down to the choice of hardware for the onboard compute.

But despite the promise of Deep RL, there are several challenges in adopting reinforcement learning for UAV trajectory generation. Broadly, the problems can be grouped into three main categories: (1) data collection, (2) policy optimization and (3) hardware evaluation. The first challenge is that Deep RL algorithms need lots of data. Collecting large amounts of data is challenging because most commercial and off-the-shelf UAVs operate for less than 30 mins. To put this into perspective, creating a dataset as large as the latest "ImageNet" by Tencent for ML Images [10] would take close to 8000 flights (assuming a standard 30 FPS camera), thus making it a logistically challenging issue. But perhaps an even more important and difficult aspect of this data collection is the need for negative experiences, such as obstacle collisions, which can severely drive up the cost and logistics of collecting data [8].

The second challenge is that there are many reinforcement learning algorithms. Choosing the right variant of a reinforcement learning algorithm for a given task requires fairly exhaustive exploration. Furthermore, since the performance and efficiency of a particular reinforcement learning algorithm are greatly influenced by the network architecture of the policy and its reward function, to get good performance, there is a need to perform design exploration between the reinforcement learning algorithms, policy, and the reward function.

The third challenge is the limited onboard energy, com-

pute capability and power budget. Since UAVs are mobile machines, they need to accomplish their tasks with a limited amount of onboard energy. Because onboard compute is a scarce resource and RL policies are computationally intensive, we need to carefully co-design the policies with the underlying hardware so that compute can meet the real-time requirements under power constraints. As the UAV size decreases, the problem exacerbates because battery capacity (i.e., size) decreases, which reduces the total onboard energy (even though the level of intelligence required remains the same). For instance, a nano-UAV such as a CrazyFlie [11] must have the same autonomous navigation capabilities as compared to its larger mini counterpart, e.g., DJI-Mavic Pro [12] while the CrazyFlie's onboard energy is $\frac{1}{15}$th that of the Mavic Pro.

To address these challenges, the boundaries between reinforcement learning algorithms, robotics control, and the underlying hardware must soften. Figure 1 illustrates the cross-layer, and interdisciplinary nature of the field, spanning from environment modeling to the underlying system. Each layer, in isolation, has a complex design space that needs to be explored for optimization. In addition, there are interactions across the layers that are also important to consider (e.g., model size on a power-constrained mobile or embedded computing system). Hence, there is a need for a platform that can aid interdisciplinary research. More specifically, we need an AI research platform that can benchmark each of the layers individually (for depth), as well as end-to-end execution for capturing the interactions across the layers (for breadth).

To that end, in this paper, we present Air Learning (Figure 2)—an AI research platform for algorithm-hardware benchmarking for autonomous UAVs.[1] It is a simulation platform that provides a scalable and cost-effective means for generating data for reinforcement learning algorithms. It augments existing frameworks such as AirSim [13] with capabilities that make it suitable for deep RL experimentation.

Air Learning addresses each of the challenges mentioned previously. To address the data availability challenge, we develop a configurable environment generator with a wide range of knobs to generate difficulty levels. The knobs are used to (randomly) tune the number of static and dynamic obstacles, their speed (if relevant), their texture and color, arena size, etc. In the context of our autonomous UAV navigation task, we use the knobs to help the learning algorithms' generalize well and not overfit to a specific instance of an environment.[2]

To address the RL algorithm, policy, and reward optimization challenge, we expose our configurable environment generator as an OpenAI gym [14] interface and integrate it with Baselines [15], which has high-quality implementations of the latest state-of-the-art reinforcement learning algorithms. We provide templates which the researchers can use for building multi-modal input policies based on Keras/Tensorflow.

Air Learning comes equipped with two very different reinforcement learning algorithms, namely Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO). DQN agent is a representative RL algorithm for discrete actions control, and PPO agent is a representative RL algorithm for continuous action control of UAVs. Both algorithms come ready with support for training the agents using curriculum learning [16].[3]

We use these algorithms to describe the training methodology for autonomous navigation. We discuss how we set up the policy architecture, reward function, action space for the DQN and PPO based agents. We also discuss the performance of the DQN and PPO agents and show a DQN agent trained using curriculum learning performs the best compared to the other agents. Also, we evaluate the best policy across a range of environments with no obstacles, static obstacles and dynamic obstacles. We show results for how a policy trained in one environment performs in another environment.

Air Learning uses a "hardware-in-the-loop" (HIL) [17] method to enable robust hardware evaluation. Hardware in the loop, which requires plugging in the processor used in the UAV into the software simulation, is a form of real-time simulation that allows us to understand how the UAV responds to simulated stimuli on a target hardware platform. HIL simulation helps us quantify the real-time performance of reinforcement learning policies on various compute platforms.

We use HIL simulation to understand how a policy performs on an embedded compute platform that might potentially be the onboard computer of the UAVs. To enable systematic HIL evaluation, we use a variety of Quality-of-Flight (QoF) metrics, such as the total energy consumed by the UAV, the average length of its trajectory and endurance, to compare the different reinforcement learning policies.

To demonstrate that Air Learning's HIL simulation is essential and that it can reveal interesting insights, we take the best performing policy from our policy exploration stage and evaluate the performance of the policy on a resource-constrained low-performance platform (Ras-Pi 3) and compare it with a high-performance desktop counterpart (Intel Core-i7).

The difference between the Ras-Pi 3 and the Core-i7 based performance for the policy is startling. The Ras-Pi 3 sometimes takes trajectories that are nearly 80% longer in some environments. We investigate the reason for the difference in the performance of the policy on Ras-Pi 3 versus Intel Core-i7 and show that the choice of onboard compute directly affects the policy processing latency, and hence the trajectory lengths.

To enable further robust hardware evaluation, we show the importance of taking into account the total energy consumed by the reinforcement learning algorithms to accomplish their task. We show that the success rate of the trained policies drops significantly when we include energy as an additional factor to determine the merit of success. Our finding motivates the need to develop energy-efficient policies since UAVs are inherently energy-constrained, battery-based mobile robots.

Finally, given that all of our work so far is in the context of simulation, to bridge the simulation to reality gap, we demonstrate how Air Learning can be used in practice to train a policy that can be put on a real drone. To this end, we train a "tiny" DQN policy for obstacle avoidance that can fit into a severe resource constrained micro-controller based UAV.

[1] Air Learning is an open source project, and it can be downloaded from GitHub: https://github.com/harvard-edge/airlearning

[2] The environment generator can be applied to other challenges in aerial robots, such as detecting thin wires and coping with translucent objects.

[3] Additional algorithms can be easily added into Air Learning as needed.

In summary, we present Air Learning. It is an AI research platform for algorithm-hardware benchmarking of Deep RL based tasks for autonomous aerial vehicles. The specific contributions within this context include:

- We address the data collection problem for Deep RL based methods using our customizable (i.e., scriptable), random environment generator.
- We present a tightly integrated framework to train different RL algorithms, policies, and reward optimizations using regular and curriculum learning.
- We describe the significance of taking energy consumption and the platform's processing capabilities into account when evaluating policies success rates.

The remainder of this paper is organized as follows. Section II reviews prior work. Section III describes Air Learning and its components. Section V describes the training methodology, policy architecture, reward function, action space for the DQN and PPO agents. Section VI evaluates the policies in different environments to study how well the policies generalize. Section VII investigates how hardware resource constraints affect policy performance, and Section VII-C digs deeper to understand the differences we measure. Section VIII discusses the importance of having energy as a success metric for evaluating reinforcement learning algorithms. Section IX shows how Air Learning design philosophy can be used to train a tiny DQN policy for micro-controller based UAVs. Section XI summarizes our contributions and concludes the paper with thoughts and ideas for follow-on work.

II. Related Work

Related work in autonomous navigation in aerial vehicles can be generally classified into six distinct categories. The first category is the algorithms that do not use any learning methods but use perception, planning and control paradigm for point-to-point navigation in an environment with static obstacles and dynamic obstacles. The second category involves various testbed and infrastructure for developing non-learning and learning based control algorithms for UAVs. The third category consists of the use of simulators that are designed explicitly for UAVs. The fourth category includes benchmarking suites explicitly designed for robotics or benchmarking kernels commonly used in a robotics application. The fifth category consists of applying learning based algorithm for complex robot task but not necessarily to UAVs. The final category involves optimizing machine learning kernels for optimizing its performance on mobile form factor devices but not necessarily to robots. We briefly discuss each of these categories and present how our work makes new contributions.

A. UAV Navigation based on Non-Learning based Algorithms

The first category of related work includes navigation of UAVs using a non-learning based algorithm. These algorithms typically follow the perception, planning, and control (PPC) paradigm. The prominent related work using PPC for obstacle avoidance and navigation includes short-range planning method [18], mixed integer programming [19], or geometric controllers [20]. In contrast, Air Learning focuses on using reinforcement learning algorithms for UAV navigation.

B. UAV Testbeds

The second category involves creating infrastructure and testbed for evaluating UAVs in the real world and also in simulators. The most prominent testbed includes Flying Machine Arena [21] from ETH, GRASP testbed from UPenn [22], Raven from MIT [23] and MAHRES testbed from UNM [24]. These testbeds are designed to validate control algorithms developed for solving a particular task. They typically consist of a large area generally of 10 m x 10 m x 10 m, fitted with a motion-capture system that determines the pose of the UAVs. The software infrastructure includes the communication protocol and distribution of computation between the motion-capture system and the onboard computer. Lastly, they also have a simulator for modeling the dynamics of UAVs before deploying it on a real testbed.

While, the testbeds provide an excellent platform for developing and testing control algorithms, using them for developing reinforcement learning algorithms has some limitations. Firstly, for developing learning-based algorithms, it is often impractical to re-design the testbed to include a wide variety of obstacles, different textures, colors and material that the UAV could encounter in real-world. It is shown that learning algorithms show sensitivity towards these features [25]. Secondly, learning algorithms require a large amount of data to train, and it is often impractical to collect them by flying in these testbeds. For instance, to learn not to collide into a door, there has to be data with some form of collision with the door so that those scenarios can be negatively reinforced during training. In such situations, Air Learning fills the gap by providing a photo-realistic environment generator to address the data unavailability problem for learning algorithms.

C. Simulators

The third category of related work on the simulator with a focus on UAVs. An example, AirSim [13] provides a high-fidelity simulation and dynamics for the UAVs in the form of a plugin that can be imported in any UE4 (Unreal Engine 4) [26] project. However, there are three limitations of the AirSim that AirLeaning addreses. First, the generation of the environment that includes domain randomization for UAV task is left to the end user to either develop or source it from UE4 market place. The domain randomizations [25] are very critical for generalization of the learning algorithm and we address this limitation in AirSim using Air Learning environment generator.

Second, AirSim does not model UAV energy consumption. Energy is a scarce resource in UAVs that affects overall mission capability. Hence, learning algorithms need to be evaluated for energy efficiency. Air Learning uses energy model [27] within AirSim to evaluate learned policies. Air Learning also allows studying the impact of the performance of the onboard compute platform on the overall energy of UAVs, allowing us to estimate in simulation how many missions UAV can do, without running in the simulation.

Third, Airsim does not offer interfaces with OpenAI gym or other reinforcement learning framework such as stable baselines[15]. We address this drawback by exposing the Air

Learning random environment generator with OpenAI gym interfaces and integrate it with a high-quality implementation of reinforcement learning algorithm available in the framework such as baselines [15] and Keras-RL [28]. Using Air Learning, we can quickly explore and evaluate different RL algorithms for various UAV tasks.

Another related work that uses a simulator and OpenAI gym interface in the context of UAVs is GYMFC [29]. GYMFC uses Gazebo [30] simulator and OpenAI gym interfaces for training an attitude controller for UAVs using reinforcement learning. The work primarily focuses on replacing the conventional flight controller with a real-time controller based on a neural network. This is a highly specific, low-level task. We are focused on more on high-level tasks, such as point-to-point UAV navigation in an environment with static and dynamic obstacles and we provide the necessary infrastructure to carry research to enable on-edge autonomous navigation in UAVs. Adapting this work to support a high-level task such as navigation will involve overcoming the limitations of Gazebo, specifically in the context of photorealism. One of the motivations of building AirSim is to overcome the limitations of Gazebo by using state-of-the-art rendering techniques for modeling the environment, which is achieved used robust game engines such as Unreal Engine 4 [26] and Unity [31].

D. Robot Benchmarking

The fourth category of related work involves robot benchmarking. The most prominent work is RoboBench [32]. The authors describe an end-to-end flow for benchmarking various robotic application using a Gazebo simulator. The work primarily focuses on perception, planning and control classes of algorithms. The UAV portion in the benchmark suite focuses on navigating to a set waypoint. In contrast, Air Learning is designed for learning based algorithms and uses a state-of-the-art game engine for rendering the environment, which overcomes the limited photorealism in Gazebo. Also, we do not set any waypoint and only give the destination to the agent.

Other related works in robot benchmarking typically focus on kernel-level benchmarking, instead of benchmarking end-to-end application. For instance, Simultaneous Localization and Mapping (SLAM) is an essential kernel in the perception stage for many robots and SLAMBench [33] is a benchmarking suite for characterizing the performance of various SLAM algorithms. Similarly, MoVeMA [34] and OpenGrasp [35] are benchmarking suite specifically targeted for benchmarking motion planning and control task respectively. These kernel-specific benchmarking suites provide great insights about a particular kernel but often misses the interaction of various components in an end-to-end application. Also, these benchmarking suites cater to the perception, planning and control paradigm for robot control. In contrast, Air Learning provides infrastructure for benchmarking learning-based algorithms to the hardware platform and everything in between.

E. Learning-based Approaches

The fifth category of related work involves end-to-end neural network based methods for robot tasks, such as grasping by a robotic arm. This related work is highly relevant to Air Learning because it shows that reinforcement learning algorithms can solve complex robotics tasks. Learning based algorithms overcome the drawback of traditional control theory based algorithm by their ability to approximate certain functions that are hard to model in the first place.

The most prominent work in learning based approaches is robotic grasping [36], [37], where a neural network model or Q-function was trained to grasp objects with different shapes and sizes. These learning-based algorithms achieve a success rate of 96% which genuinely shows the ability to learn a task using reinforcement learning. However, these kinds of robots are fixed in a place; hence they are not limited by energy, or by onboard compute capability. So the inability to process or calculate the policy's outcome in real-time only slows down the grasping rate. It does not cause instability. In UAVs, which have a higher control loop rate, instability due to slow processing latency can cause fatal crashes [38], [39].

Non deep learning has been used primarily for UAV planning and controls. For example, to iterative learn multi-flips [40] and trajectory tracking [41], safe controllers with Gaussian processes [42], [43]. RL methods for control and planning include UAV manipulation [44] and suspended load control [45]. All these methods assume the perfect, given perception. More recently, deep RL was used for trajectory tracking [46], and learned perception with non-learned controls [47]. End-to-end deep RL methods include learning to fly through experimentation on real robot [48] and visual navigation by training the policies from the simulated data [49]. AirLearning aims at being a benchmarking suite for development of end-to-end deep RL policies, and supports on-edge compute considerations, which the pervious methods do not take into account.

In Air Learning infrastructure, we use the HIL methodology to characterize the performance of the learned policy. It helps to carefully co-design the policy for the underlying hardware and quantify the overall performance using QoF metrics.

F. System Optimization

The sixth category of related work, though this category does not explicitly target robotics, involves optimizing machine learning kernels for improving the system performance. Prior work in this category includes characterization and benchmarking of the machine learning kernels as the very first step and then applies optimizations. Applying optimizations early on can help design better learning algorithms and policies for resource and energy constrained platforms.

MobileNets [50], SqueezeNets [51] and Deep Compression [52] are the most prominent examples in this category. The goal of these prior works is to enable state of the art image recognition models, which are typically several hundred MBs, to fit into a mobile phone like form factor where battery life, memory and response time are very critical for user satisfaction. MobileNets targets mobile phones. The authors re-architect the policy by introducing point-wise convolution, which reduces the number of FP-MACs (Floating point Multiply and Accumulate) operation and thus significantly improves the classification time without impacting the accuracy.

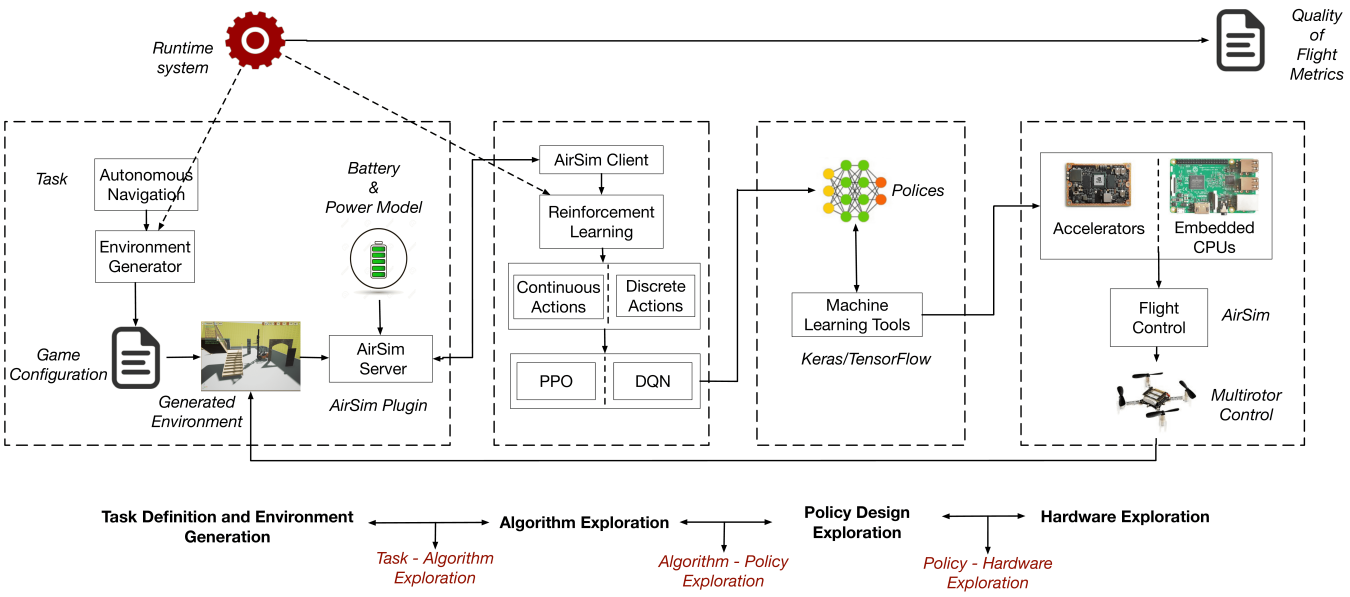


Fig. 2: Air Learning infrastructure and benchmarking suite for end-to-end learning in autonomous aerial machines. Our infrastructure consists of four main components. First, it has a configurable random environment generator built on top of UE4, a photo-realistic game engine that can be used to create a variety of different randomized environments. Second, the random environment generators are integrated with AirSim, OpenAI gym, and baselines for agile development and prototyping different state of the art reinforcement learning algorithms and policies for autonomous aerial vehicles. Third, its backend uses tools like Keras/Tensorflow that allow the design and exploration of different policies. Lastly, Air Learning uses the "hardware in the loop" methodology for characterizing the performance of the learned policies on real embedded hardware platforms. In short, it is an interdisciplinary tool that allows researchers to work from algorithm to hardware with the intent of enabling intra- and inter-layer understanding of execution. It also outputs a set of "Quality-of-Flight" metrics to understand execution.

SqeezeNet, on the other hand, uses model compression, which results in 50X lesser parameters compared to AlexNet, without impacting the accuracy. Deep Compression, involves pruning, quantization, and encoding thereby reducing the model size by 50X without impacting the accuracy.

Another related work uses reinforcement learning algorithms to perform system-level optimization. To determine the optimal system-level parameters requires a large design space exploration. Running the system at these optimal points guarantees the best system performance for the given workload. For instance in HAQ [53], the authors train a Deep Deterministic Policy Gradient (DDPG) to determine the best quantization level to achieve the best system-level performance across multiple and different kinds of hardware.

The learning methods for on-edge autonomous navigation in UAVs can benefit from these system-level optimizations because policies are often neural network based. Hence these policies could also be amenable to these system-level optimizations. Moreover, UAVs are severely constrained by the battery and capability of the onboard computer.

Air Learning provides an infrastructure for researchers to develop learning algorithms for UAVs. It helps design effective policies, and also characterize them on an onboard computer using the HIL methodology and quality-of-flight metrics. With that in mind, it is possible to start optimizing algorithms for UAVs, treating the entire UAV and its operation as a system.

III. Air Learning

In this section, we describe the various Air Learning components. The different stages are shown in Figure 2, which allows researchers to develop and benchmark learning algorithms for autonomous UAVs. Air Learning consists of six keys components: an environment generator, an algorithm exploration framework, closed-loop real-time hardware in the loop setup, an energy and power model for UAVs, quality of flight metrics that are conscious of the UAV's resource constraints, and a runtime system that orchestrates all of these components. By using all these components in unison, Air Learning allows us to fine-tune algorithms for the underlying hardware carefully.

A. Environment Generator

Learning algorithms are data hungry, and the availability of high-quality data is vital for the learning process. Also, an environment that is good to learn from should include different scenarios that are challenging for the robot. By adding these challenging situations, they learn to solve those challenges. For instance, for teaching a robot to navigate obstacles, the data set should have a wide variety of obstacles (materials, textures, speeds, etc.) during the training process.

We designed an environment generator specifically targeted for autonomous UAVs. Air Learning's environment generator creates high fidelity photo-realistic environments for the UAVs to fly in. The environment generator is built on top of UE4 and

Parameter	Format	Description
Arena Size	`[length, width, height]`	Spawns a rectangular arena of "length" x "width" x "height".
Wall Colors	`[R, G, B]`	The colors of the wall of in [Red, Green, Blue] color format.
# Static Obstacles	Scalar Integer	The number of static obstacles in the arena.
# Dynamic Obstacles	Scalar Integer	The number of the dynamic obstacle in the arena.
Seed	Scalar Integer	Seed value used in randomization.
Minimum Distance	Scalar Integer	Minimum distance between two obstacle in the arena.
Goal Position	`[X, Y, Z]`	Sets the goal position in X, Y and Z coordinates.
Velocity	$[V_{max}, V_{min}]$	Velocity of the dynamic obstacle between V_{max} and V_{min} .
Asset	`<folder name>`	Air Learning allows any UE4 asset to be imported into the project.
Materials	`<folder name>`	Any UE4 material can be assigned to the UE4 asset.
Textures	`<folder name>`	Any UE4 Texture can be assigned to the UE4 asset.

TABLE I: List of configurations available in current version of Air Learning environment generator.

uses the AirSim UE4 [13] plugin for the UAV model and flight physics. The environment generator with the AirSim plugin is exposed as OpenAI gym interface.

The environment generator has different configuration knobs for generating challenging environments. The configuration knobs available in the current version can be classified into two categories. The first category includes the parameters that can be controlled via a game configuration file. The second category consists of the parameters that can be controlled outside the game configuration file. The full list of parameters that can be controlled are shown in tabulated in Table I, and they are as described below.

Arena Size: The `Arena Size` is the total volume available in the environment. It is represented by `[length, width, height]` tuple. A large arena size means the UAV has to cover more distance in reaching the goal which directly impacts its energy and mission success (Section VIII). Figure 3 shows different arena sizes. The arena can be customized by adding materials, which we describe in the "materials" section.

Wall Color: The `Wall Color` parameter can be used to set the wall colors of the arena. The parameter takes `[R, G, B]` tuple as input. By setting different values of `[R, G, B]`, any color in the visible spectrum can be applied to the walls. The neural network policies show sensitivity towards different colors [54] and varying these color during training can help the policy to generalize well.

Number of Obstacles: The `# Static Obstacles` is a parameter that describes the total number of static objects that is spawned in the environment. Using this parameter, we can generate environments ranging from very dense to very sparse obstacles. Depending upon the value of this parameter, the navigation complexity can be easy or difficult. A large number of obstacles increases the collision probability and can be used for stressing the efficacy of reinforcement learning algorithms.

Minimum Distance: The `Minimum distance` is a parameter that controls the minimum distance between two static objects in the arena. This parameter in conjunction with `# Static Obstacles` is what determines congestion.

Goal Position: The `Goal Position` is a parameter that specifies the destination coordinate that the UAV must reach. The `Goal Position` coordinates should always be inside the arena, and there is error checking for input errors. Similar to `# Static Obstacles`, it increases task complexity.

Number of Dynamic Obstacles: The `# Dynamic Obstacles` is a parameter that describes the total number of obstacles that can move in the environment.

Velocity: The `Velocity` parameter is a tuple of the form $[V_{min}, V_{max}]$ that works with `# Dynamic Obstacles`. The environment generator randomly chooses a value from this range for the velocity of a dynamic obstacle. This coupled with the `# Dynamic Obstacles` helps control how dynamic and challenging the environment is for the aerial robot.

Seed: The `Seed` parameter is used for randomizing the different parameters in the environment. By setting the same 'Seed' value, we can reproduce (and randomize) the environment (obstacle position, goal position, etc.).

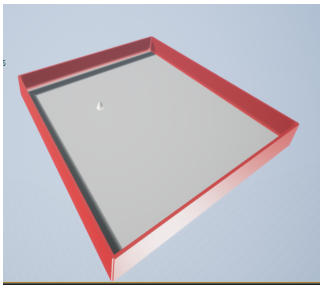
As mentioned previously, there is a second category of parameters that can be configured. These are not included in the configuration file. Instead, they are controlled by putting files into folders. Details about them are as follows.

Asset: An `Asset` in Air Learning is a mesh in UE4 [55]. Any asset that is available in the project can be used as a static obstacle, dynamic obstacle, or both. At simulation startup, Air Learning uses these assets as either a static or dynamic obstacle. The number of assets that will be spawned in the arena will be equal to the `#Static Obstacle` and `#Dynamic Obstacle` parameter. By having the ability to spawn any asset as an obstacle, the UAV agent can generalize to avoid collision with different types of obstacle. Figure 4 shows some of the sample assets used in this work.

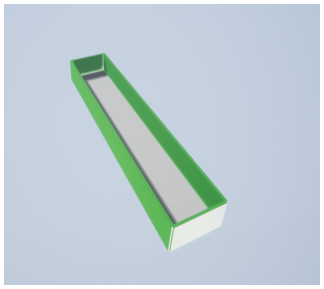
Textures: A `Texture` is an image that is used on an UE4 asset [56]. They are mapped to the surfaces of any given asset. At startup, the environment generator applies textures to matching assets. Textures and materials (below point) help the training algorithm capture different object features, which is important to help the algorithm generalize.

Materials: A `Material` is a UE4 asset [57] that can be applied to meshes to control the visual look of the scene. Material is usually made of multiple textures to create a particular visual effect for the asset. At simulation startup, Air Learning environment generator applies materials to matching assets.

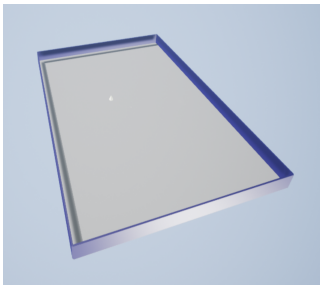
Materials can help training algorithms on two fronts. First, neural network policy has a sensitivity to capture various material features in the objects [54], [25]. For instance, the



(a) Arena 1 with crimson walls.

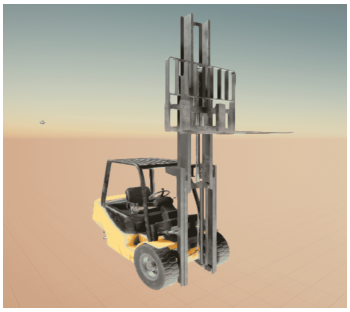


(b) Arena 2 with green walls.

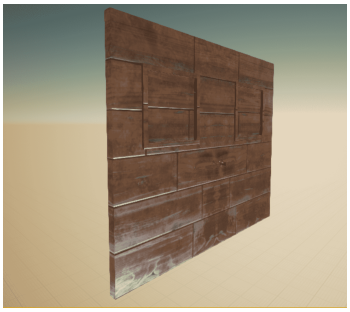


(c) Arena 3 with blue walls.

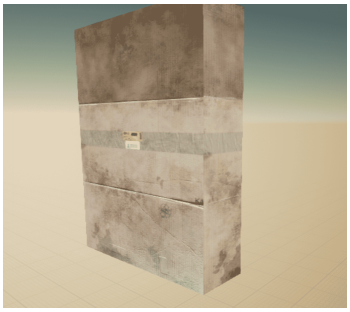
Fig. 3: The environment generator generates different arena sizes with configurable wall texture colors. The arena can be small or several miles long. The wall texture color is specified as an [R, G, B] tuple, which allows the generate to create any color in the visible spectrum. (a) An arena with a volume of `50 m X 50 m X 5 m` with crimson colored walls. (b) An arena with `50 m X 10 m X 5 m` with green colored walls. (c) An arena with `100 m X 75 m X 5 m` with violet colored walls.



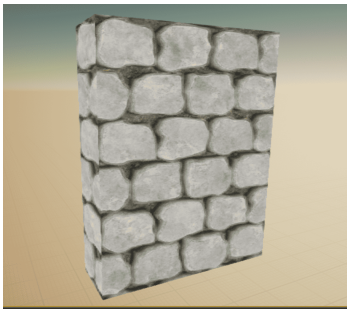
(a) Forklift.



(b) Wall section.



(c) Cardboard box.

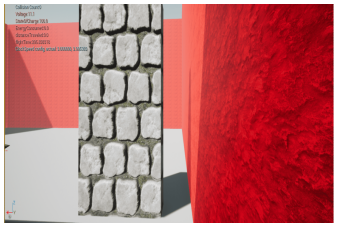


(d) Stone wall section.

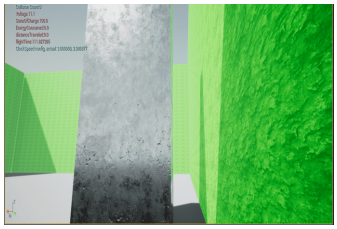
(e) Stair way section.

(f) Trailer section.

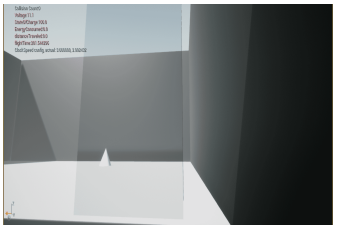
Fig. 4: The environment generator uses a game configuration script that allows creating different scenarios that are challenging for the robot. Any UE4 game engine asset can be imported into the Air Learning environment generator and it will use these assets to spawn them randomly into the game environment. The assets can be either static or made dynamic to move around.



(a) Stone material.



(b) Metallic material.



(c) Glass material.

Fig. 5: The environment generator can apply different materials to a UE4 mesh. (a) A UE4 mesh (cube) applied with stone material. (b) A UE4 mesh with metallic and shiny material. (c) A UE4 mesh with transparent glass material. We can assign different materials to the same UE4 mesh, and the environment generator will randomly choose the material at runtime.

type of material affects how light interacts with the surface, and as a result, an RL based robot that is relying on images as input can learn different things (and act differently) under different materials and the textures that it observes. Second, they can make it challenging for the algorithms using image-based inputs. For instance, shiny and transparent objects are harder to detect [58], [59]. Figure 5 shows how different materials can be applied to the same asset in Air Learning.

In summary, Air Learning's environment generator allows any UE4 asset to be loaded into the project, and provides flexibility in the choice of obstacles, materials, and texture. These features are essential to provide a safe sandbox environment where to train and evaluate various deep reinforcement learning algorithms and policies that can generalize well.

B. Algorithm and Policy Exploration

Deep reinforcement learning is still a nascent field that is rapidly evolving. Hence, there is significant infrastructure overhead to integrate random environment generator and evaluate new deep reinforcement learning algorithms for UAVs.

So, we expose our random environment generator and AirSim UE4 plugin as an OpenAI gym interface and integrate it popular reinforcement learning framework with stable baselines [15], which is based on OpenAI baselines.[4] To expose our random environment generator into an OpenAI gym interface, we extend the work of AirGym [60] to add support for environment randomization, a wide range of sensors (Depth image, Inertial Measurement Unit (IMU) data, RGB image, etc.) from AirSim and support exploring multimodal policies.

We seed the Air Learning algorithm suite with two popular and commonly used reinforcement learning algorithms. The first is Deep Q Network (DQN) [61] and the second is Proximal Policy Optimization (PPO) [62]. DQN falls into the discrete action algorithms where the action space is high-level commands ('move forward,' 'move left' e.t.c.,) and Proximal Policy Optimization falls into the continuous action algorithms (e.g., policy predicts the continuous value of velocity vector).

Another essential aspect of deep reinforcement learning is the policy, which determines the best action to take. Given a particular state the policy needs to maximize the reward. A neural network approximates the policies. To assist the researchers in exploring effective policies, we use Keras/TensorFlow [63], [64] as the machine learning back-end tool. Later on, we demonstrate how one can do algorithm and policy explorations for tasks like autonomous navigation though Air Learning is by no limited to this task alone.

C. Hardware Exploration

Often aerial roboticists port the algorithm onto UAVs to validate the functionality of the algorithms. These UAVs can be custom built [65] or commercially available off-the-shelf (COTS) UAVs [66], [67] but mostly have fixed hardware that can be used as onboard compute. A critical shortcoming of this approach is that the roboticist cannot experiment with hardware changes. More powerful hardware may (or may not) unlock additional capabilities during flight, but there is no way to know until the hardware is available on a real UAV so that the roboticist can physically experiment with the platform.

Reasons for wanting to do such exploration includes understanding the computational requirements of the system, quantifying the energy consumption implications as a result of interactions between the algorithm and the hardware, and so forth. Such evaluation is crucial to determine whether an algorithm is, in fact, feasible when ported to a real UAV with a specific hardware configuration and battery constraints.

For instance, a Parrot Bepop [68] comes with a P7 dual-core CPU Cortex A9 and a Quad core GPU. It is not possible to fly the UAV assuming a different piece of hardware, such as the NVIDIA Xavier [69] processor that is significantly more powerful; at the time of this writing there is no COTS UAV that contains the Xavier platform. So, one would have to wait until a commercially viable platform is available. However, using Air Learning, one can experiment how the UAV would behave with a Xavier since the UAV is flying virtually.

Hardware exploration in Air Learning allows for evaluation of the best reinforcement learning algorithm and its policy on different hardware. It is not limited by the onboard compute available on the real robot. Once the best algorithm and policy are determined, Air Learning allows for characterizing the performance of these algorithms and policies on different types of hardware platforms. It also enables to carefully fine-tune and co-design algorithms and policy while being mindful of the resource constraints and other limitation of the hardware.

A HIL simulation combines the benefits of the real design and the simulation by allowing them to interact with one another as shown in Figure 6. There are three core components in Air Learning's HIL methodology: (1) a high-end desktop that simulates a virtual environment flying the UAV (top); (2) an embedded system that runs the operating system, the deep reinforcement learning algorithms, policies and associated software stack ($left$); and (3) a flight controller that controls the flight of the UAV in the simulated environment ($right$).

The simulated environment models the various sensors (RGB/Depth Cameras), actuators (rotors), and the physical world surrounding the agent (Obstacles). This data is fed into the reinforcement learning algorithms that are running on the embedded companion computer, which processes the input and outputs flight commands to the flight controller. The controller then communicates those commands into the virtual UAV flying inside the simulated game environment.

The interaction between the three components is what allows us to evaluate the algorithms and policy on various embedded computing platforms. The HIL setup we present allows for the swap-ability of the embedded platform under test. The methodology enables us to effectively measure both the performance and energy of the agent holistically and more accurately, since one can evaluate how well an algorithm performs on a variety of different platforms.

In our evaluation, which we discuss later, we use a Raspberry Pi (Ras-Pi 3) as the embedded hardware platform to evaluate the best performing deep reinforcement learning algorithm and its associated policy. The HIL setup includes running the environment generator on a high-end desktop

[4]We also support Keras-RL, another widely used RL framework.

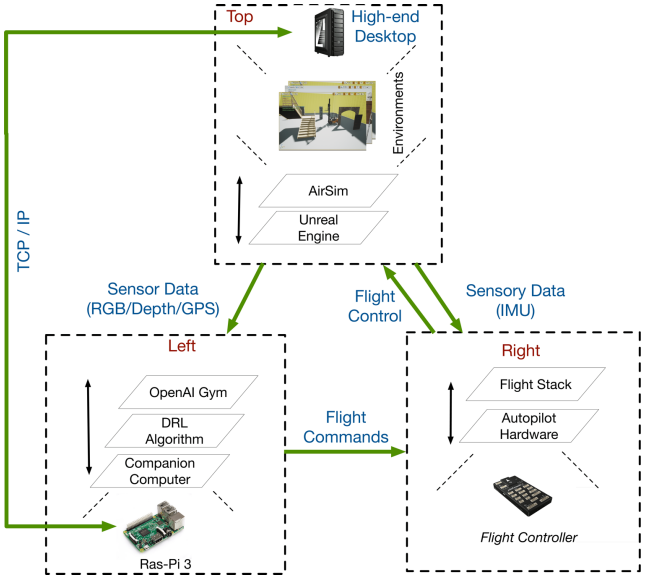


Fig. 6: Hardware-in-the-loop (HIL) simulation in Air Learning.

with a GPU. The reinforcement learning algorithm and its associated policy run on the Ras-Pi 3. The state information (Depth image, RGB image, IMU) are requested by Ras-Pi 3 using AirSim Plugins APIs which involves an RPC (remote procedural calls) over TCP/IP network (both high-end desktop and Ras-Pi 3 are connected by ethernet). The policy evaluates the actions based on the state information it received from the high-end desktop. The actions are relayed back to the high-end desktop through AirSim flight controller API's.

D. Energy Model in AirSim Plugin

In Air Learning, we use the energy simulator we developed in our prior work [27]. The AirSim plugin is extended with a battery and energy model. The energy model is a function of UAVs velocity, acceleration. The values of velocity and acceleration are continuously sampled and using these we estimate the power as proposed in this work [70]. The power is calculated using the following formula:

$$P = \begin{bmatrix}\beta_1\\ \beta_2\\ \beta_3\end{bmatrix}^T \begin{bmatrix}\|\vec{v}_{xy}\|\\ \|\vec{a}_{xy}\|\\ \|\vec{v}_{xy}\|\ \|\vec{a}_{xy}\|\end{bmatrix} + \begin{bmatrix}\beta_4\\ \beta_5\\ \beta_6\end{bmatrix}^T \begin{bmatrix}\|\vec{v}_z\|\\ \|\vec{a}_z\|\\ \|\vec{v}_z\|\ \|\vec{a}_z\|\end{bmatrix} + \begin{bmatrix}\beta_7\\ \beta_8\\ \beta_9\end{bmatrix}^T \begin{bmatrix}m\\ \vec{v}_{xy}\cdot\vec{w}_{xy}\\ 1\end{bmatrix} \quad (1)$$

In Eq. 1, v_{xy} and a_{xy} are the velocity and acceleration in the horizontal direction. v_z and a_z denotes the velocity and acceleration in the z direction. m denotes the mass of the payload. β_1 to β_9 are the coefficients based on the model of the UAV used in the simulation. For the energy calculation model, we use the columb counter technique as described in prior work [71]. The simulator computes the total number of columb that has passed over the battery over every cycle.

Using the energy model Air Learning allows us to monitor the energy continuously during training or during the evaluation of the reinforcement learning algorithm.

E. Quality of Flight Metrics

Reinforcement learning algorithms are often evaluated based on success rate where the success rate is based on whether the algorithm completed the mission. This metric only captures the functionality of the algorithm and grossly ignores how well the algorithm performs in the real world. In the real world, there are additional constraints for a UAV, such as the limited onboard compute capability and battery capacity.

Hence, we need additional metrics that can quantify the performance of learning algorithms more holistically. To this end, Air Learning introduces Quality-of-Flight (QoF) metrics that not only captures the functionality of the algorithm but also how well they perform when ported to onboard compute in real UAVs. For instance, the algorithm and policies are only useful if they accomplish the goals within finite energy available in the UAVs. Hence, algorithms and policies need to be evaluated on the metrics that describe the quality of flight such as mission time, distance flown, etc. In the first version of Air Learning, we consider the following metrics.

Success Rate: The percentage of time the UAV reaches the goal state without collisions and running out of battery. Ideally, this number will be close to 100% as it reflects the algorithms' functionality, taking into account resource constraints.

Time to Completion: The total time UAV spends finishing a mission within the simulated world.

Energy Consumed: The total energy spent while carrying out the mission. Limited battery available onboard constrains the mission time. Hence, monitoring energy usage is of utmost importance for autonomous aerial vehicles, and therefore should be a measure of policy's efficiency.

Distance Traveled: Total distance flown while carrying out the mission. This metric is the average length of the trajectory that can be used to measure how well the policy did.

F. Runtime System

The final part is the runtime system that orchestrates the overall execution. The runtime system starts the game engine with the correct configuration of the environment before the agent starts. It also monitors the episodic progress of the reinforcement learning algorithm and ensures that before starting a new episode that it randomizes the different parameters, so the agent statistically gets a new environment. It also has resiliency built into it to resume the training in case any one of the components (for example UE4 engine) crashes.

In summary, using Air Learning environment generator, researchers can develop various challenging scenarios to design better learning algorithms. Using Air Learning interfaces to OpenAI gym, stable-baselines and TensorFlow backend, they can rapidly evaluate different reinforcement learning algorithms and their associated policies. Using Air Learning HIL methodology and QoF metrics, they can benchmark the performance of learning algorithms and policies on resource-constrained onboard compute platforms.

IV. Experimental Evaluation Prelude

The next few sections focus heavily on how Air Learning can be used to demonstrate its value. As a prelude, this section presents the highlights to focus on the big picture.

Algorithm Exploration (Section V): We focus on how Air Learning can be used to study different algorithms (such as PPO, DQN, etc.) for accomplishing a specific task. In this work, we focus on the autonomous navigation task. We explore two different algorithms, one for non-continuous and continuous, namely DQN and PPO, respectively, for the autonomous navigation task and compare the performance of the agents trained with and without curriculum learning.

Policy Evaluation (Section VI): We show how Air Learning can be used to explore different reinforcement learning based policies. We use the best algorithm determined during the algorithm exploration step and use that algorithm to explore the best policy. In this work, we use Air Learning environment generator to generate three environments namely *No Obstacles*, *Static Obstacles*, and *Dynamic Obstacles*. These three environments create a varying level of difficulty by changing the number of static and dynamic obstacles in the environments for the autonomous navigation task.

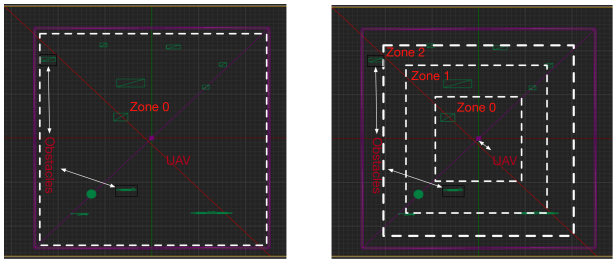
We also show how Air Learning allows end users to perform benchmarking of the policies by showing two examples. In the first example, we show how well the policies trained in one environment generalize to the other environments. In the second example, we show to which of the sensor inputs the policy is most sensitive towards. This insight can be used while designing the network architecture of the policy. For instance, we show that image input has the highest sensitivity amongst other inputs. Hence a future iteration of the policy can have more feature extractors (increasing the depth of filters) dedicated to the image input.

System Evaluation (Section VII): We show the importance of benchmarking algorithm performance on resource-constrained hardware such as what is typical of a UAV compute platform. In this work, we use a Raspberry Pi 3 (Ras-Pi 3) as an example of resource-constrained hardware. We use the best policies determined in the policy exploration step (Section VI) and use that to compare the performance between Intel Core-i7 and Ras-Pi 3 using HIL and the QoF metrics available in Air Learning. We also show how to artificially degrade the performance of the Intel Core-i7 to show how compute performance can potentially affect the behavior of a policy when it is ported over to a real aerial robot.

In summary, using these focused studies, we demonstrate how Air Learning can be used by researchers to design and benchmark algorithm-hardware interactions in autonomous aerial vehicles, as shown previously in Figure 2.

V. Algorithm Exploration

We explore two RL algorithm types for end-to-end navigation task in autonomous UAVs. The choice of the seed algorithm we used in this work can be classified into discrete action algorithms and continuous action algorithm. For discrete action reinforcement learning algorithm, we use Deep Q Networks (DQN), and for the continuous action algorithm, we use Proximal Policy Optimization (PPO). For both these algorithms, we keep the observation space, policy architecture and reward structure same and compare agent performance.



(a) Non-curriculum learning based on static obstacles.

(b) Curriculum learning based on static obstacles.

Fig. 7: Zoning used in the training methodology for curriculum learning and non-curriculum learning. Here we show the top view of our environment in wireframe mode [72] available in UE4. (a) In non-curriculum learning, the end goal is randomly placed anywhere in the arena. Unlike curriculum learning, the entire arena is one zone. (b) In curriculum learning, we split the arena into virtual partitions, and the end goal is placed within a specific zone and gradually moved higher zone once it succeeds in more than 50% over 1000 latest episode.

A. Training Methodology

The training methodology, policy architecture, reward function, and action space for PPO and DQN agent with and without curriculum learning is described below.

Non-Curriculum Learning: We train the DQN agent and PPO agent on the environment with static obstacles. To determine the baseline performance for both the algorithms, we train each agent to 1 Million steps using non-curriculum learning. For non-curriculum learning, we randomize the position of the goal and obstacles every episode to be anywhere in the arena. Simply put, the entire arena acts like one zone as shown in Figure 7a. The checkpoints are saved every 50000 steps and use the last saved checkpoint after 1 Million steps.

Curriculum Learning: To improve the baseline performance for DQNs and PPO, we employ the curriculum learning [16] approach where the goal position is progressively moved farther away from the starting point of the agent. To implement this, we divide the entire arena into multiple zones namely Zone 0, Zone 1 and Zone 2 as shown in Figure 7b. Zone 0 corresponds to the region that is within 16 m from the UAV starting position and Zone 1 and Zone 2 are within 32 m and 48 m respectively. Initially, the position of goal for the UAV is determined randomly such that the goal position lies within Zone 0. Once the UAV agent achieves 50% success over a rolling window of past 1000 episodes, the position of the goal expands to Zone 1 and so forth. To make sure that the agent does not forget learning in the previous zone, the goal position in the next zone is inclusive of previous zones. We train the agent to progress until Zone 2. Both the agents (PPO and DQN) are trained for 1 Million steps. We checkpoint the policy at every zone so that it can be evaluated on how well it has learned to navigate across all three zones.

Policy Architecture: The policy architecture for both PPO and DQN agent used is multi-modal in nature. It receives depth image, velocity vector (V_t) and position vector (X_t) as inputs as shown in Figure 8. The V_t is a 1-dimensional vector of

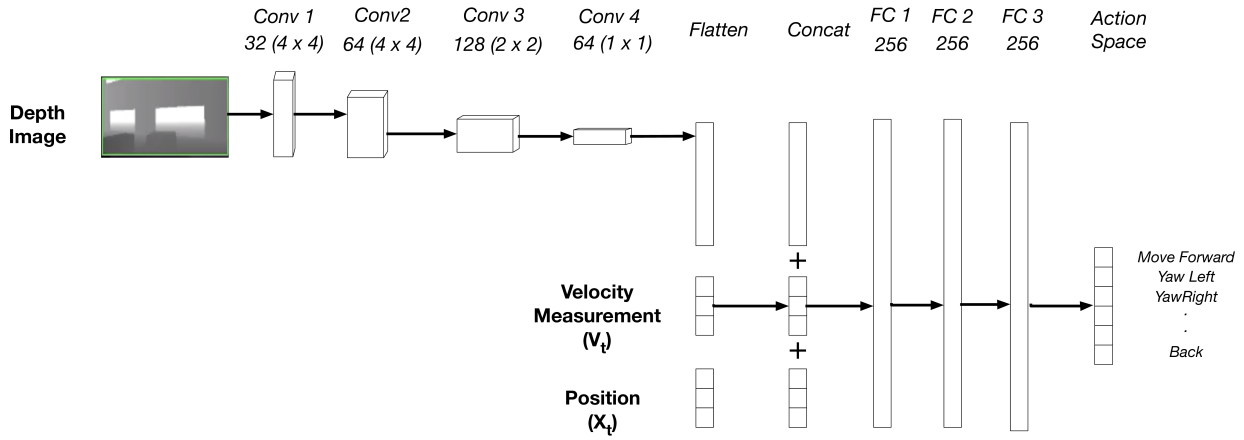


Fig. 8: The network architecture for the policy in the PPO and DQN agents. Both the agents take a depth image, velocity vector, and position vector as inputs. The depth image has four layers of convolutions after which the results are concatenated with the velocity and position vectors. In a `32 (4 X 4)` convolution filter, `32` is the depth of the filter and `(4 X 4)` is the filter size. The combined vector space is applied to the three layers of a fully connected network, each with 256 hidden units. The action space determines the number of hidden units in the last fully connected layer. For DQN agent, we have twenty-five actions, and for PPO agent we have two actions which control the velocity of the UAV agent in X and Y direction.

the form $[v_x, v_y, v_z]$ where v_x, v_y, v_z are the components of velocity vector in x, y and z directions at time *'t'*. The X_t is a 1-dimensional vector of the form $[X_{goal}, Y_{goal}, D_{goal}]$, where X_{goal} is the difference in the x-coordinate of the goal and x-coordinate of the agent's current position, Y_{goal} is the difference in the y-coordinate of the goal and y-coordinate of the agent's current position and D_{goal} is the euclidean distance to the goal from the agent's current position.

The depth image is processed by four convolutions layers whose filter depth and size are `32 (4 X 4)`, `64 (4 X 4)`, `128 (2 X 2)`, and `64 (1 X 1)` respectively. As an example, in a `32 (4 X 4)` filter, `32` is the depth of the filter and `(4 X 4)` is the size of the filter. The fourth layer's output is flattened and concatenated with the velocity vector (V_t) and position vector (X_t). The combined inputs are then fed to three layers of fully connected layers with `256` hidden units each. The action space for the agent determines the number of hidden units in the final fully connected layer. For the DQN agent, we have twenty-five discrete actions whereas, for PPO agent, we have two actions. Hence, the final layer for the DQN agent has twenty-five hidden units, and PPO agent has two hidden units. For DQN agent, the activation used for all convolution and the fully connected layer is *ReLU*, and for PPO agent, we use *ReLU* except for the last layer where we use *Tanh* for producing continuous values.

Action Space: The action space for DQN consists of twenty-five discrete actions. Out of these twenty-five action spaces, ten actions are for moving forward with different fixed velocities ranging from 1 m/s to 5 m/s, five actions are for moving backward, five actions for yawing right with fixed yaw rates of 108 °, 54 °, 27 °, 13.5 °and 6.75 °and another five actions for yawing left with fixed yaw rates of -216 °, -108 °, -54 °, -27 °and -13.5 °. At each time step, the policy takes observation space as inputs and outputs one of the twenty-five actions based on the observation. The high-level actions are mapped to low-level flight commands using the flight controller show in Figure 6 and as it is implemented.[5]

The action space for PPO on the other hand consist of velocity components v_x (velocity in x-direction) and v_y (velocity in y-direction). At each time step, the policy takes observation as the input and generates continuous values for v_x and v_y. The values of v_x and v_y are scaled such that values of the magnitude of velocity lie anywhere between 1 m/s to 5 m/s. We use the `MaxDegreeOfFreedom` option in the AirSim API that calculates the yaw rates automatically to make sure the drone is pointed in the direction it moves.

Reward: The reward function for both PPO agent and DQN agent are kept the same and is defined as follows.

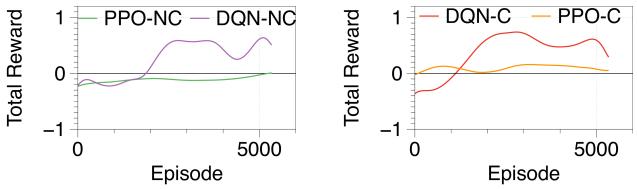
$$r = 1000 * \alpha - 1000 * \beta - D_g - D_c * \gamma \tag{2}$$

α is a binary variable where '1' denotes if the goal is reached else it is '0'. β is also a binary variable where '1' denotes if there is a collision with walls, obstacles or ground else it is '0'. D_g is the distance to the goal at any time steps from the agents' current position. If the agent is going away from the goal, the distance to the goal increases thus penalizing the agent. γ is also a binary variable which is set to '1' if the agent is closer to the goal. D_c is the distance correction which is applied to penalize the agent if it chooses actions which speed up the agent away from the goal. The distance correction term is defined as follows:

$$D_c = (V_{max} - V_{now}) * t_{max} \tag{3}$$

V_{max} is the maximum velocity possible for the agent which for DQN is fixed at 5 m/s and for PPO the outputs are scaled to lie between 1 m/s to 5 m/s. V_{now} is the current velocity of the agent and t_{max} is the duration of the actuation.

[5]https://microsoft.github.io/AirSim/docs/simple_flight/



(a) Non-curriculum learning. (b) Curriculum learning.

Fig. 9: (a) Normalized reward during training for algorithm exploration between PPO-NC and DQN-NC. (b) Normalized reward during training for algorithm exploration between PPO-C and DQN-C. We find that the DQN agent performs better than the PPO agent irrespective of whether the agent was trained using curriculum learning or non-curriculum learning.

B. PPO vs. DQN

We compare the performance of the agent trained using the DQN algorithm versus the PPO algorithm. We compare the performance of DQN and PPO agent at two levels. In the first level of exploration, we compare the performance of the PPO and DQN agents trained using non-curriculum learning. In the second level of exploration, we compare the performance of PPO and DQN agents trained using curriculum learning; curriculum learning techniques have shown to improve the performance of reinforcement learning agents [16], [6].

Figure 9a shows the normalized episodic reward of the DQN agent (DQN-NC) and PPO agent (PPO-NC) trained using non-curriculum learning. One of the critical observations is that the PPO agent trained using non-curriculum learning consistently accrues negative reward throughout the training duration. In contrast, the DQN agent trained using non-curriculum learning starts at the same as the PPO agent but the DQN agent accrues more positive reward beginning in the 2000^{th} episode.

Figure 9b shows the normalized episodic reward for the DQN (DQN-C) and PPO (PPO-C) agents trained using curriculum learning. We observe a similar trend as we saw with the agents trained using non-curriculum learning where the DQN agent outperforms the PPO agent. However, in this case, the PPO agent has a positive total reward. But the DQN agent starts to accrue more reward starting from the 1000^{th} episode.

Reflecting on the results, we have gathered in Figure 9a and Figure 9b, continuous action algorithms have generally been known to show promising results for low-level flight controller tasks that are used for stabilizing UAVs [73]. However, as our results indicate, applying these algorithms for a complex task, such as end-to-end navigation in a photo-realistic simulator, can be challenging for a couple of reasons.

First, we believe that the action space for the PPO agent limits the exploration compared to the DQN agent. For the PPO agent, the action space is the components of velocity vector v_x and v_y whose value can vary from [-5 m/s, 5 m/s]. Having such an action space can be a constraining factor for PPO. For instance, if the agent observes an obstacle at the front, it needs to take action such that it moves right or left. Now for PPO agent, since the action space is continuous values of [v_x, v_y], for it to move forward in the x-direction, the v_x can be any positive number while the v_y component has to be '0'. It can be quite challenging for the PPO agent (or continuous action algorithm) to learn this behavior, and it might require a much more sophisticated reward function that identifies these scenarios and rewards or penalizes these behaviors accordingly. In contrast, for the DQN agent, the action space is much simpler since it has to only yaw (i.e., move left or right) and then move forward or vice versa.

Second, in our evaluation, we keep the reward function, input observation and the policy architecture same for DQN and PPO agent. We choose to fix these because we want to focus on showcasing the capability of the Air Learning infrastructure. Since RL algorithms are sensitive to hyperparameters and the choice of the reward function, it could be possible that our reward function, policy architecture could have inadvertently favored the DQN agent compared to the PPO agent. The sensitivity of the RL algorithms to the policy and reward is still an open research problem [74], [75].

The takeaway is that we can do exploratory studies with Air Learning, though an in-depth algorithmic exploration is outside the scope of this general work, and we defer such more detailed studies for future work. So, in summary, for the autonomous navigation task, the DQN agent outperforms the PPO agent when trained both with and without curriculum learning. From here on, we use DQN as the algorithm for autonomous navigation and explore the best policy for different environments with and without static or dynamic obstacles.

VI. Policy Evaluation

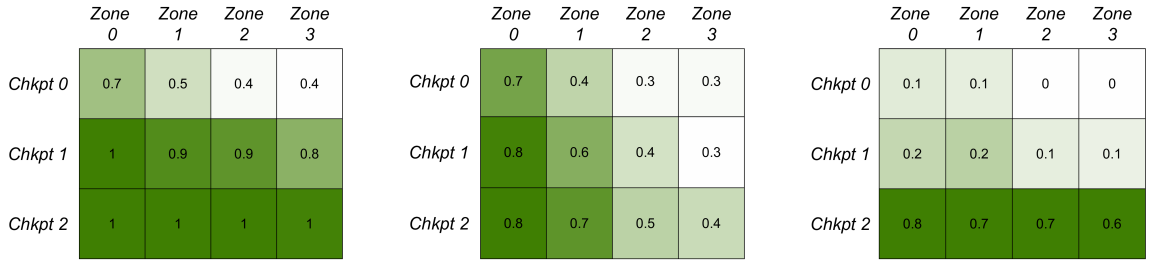
In this section, we show how Air Learning can be used for policy exploration. We use a DQN agent with curriculum learning, since it performed better than a PPO agent, to determine the best policy for navigation in different environments.

A. Training and Testing Methodology

The training and testing methodology for the DQN agent running in the different environments is described below.

Environments: For the point-to-point autonomous navigation task for UAVs, we create an environment with varying levels of static obstacles and dynamic obstacles. We produce three randomly generated environments namely *No Obstacles*, *Static Obstacles* and *Dynamic Obstacles*. The environment size for all three levels is 50 m x 50 m. For the *No Obstacles* environment, there are no obstacles in the arena, but the goal position is changed every episode. For *Static Obstacles*, the number of obstacles varies from five to ten, and it is changed every four episodes, and the end goal and position of the obstacles are changed every episode. For *Dynamic Obstacles* along with five static obstacles, we introduce up to five dynamic obstacles of whose velocities range from 5 m/s to 10 m/s. The position of the obstacles and the goal are placed in random locations in every episode to ensure that the policy does not over-fit to the environment.

Training Methodology: We train the DQN agent using curriculum learning in the environments described above. We use the same methodology described in Section V where we checkpoint policy in each zone for the three environments.



(a) No obstacles. (b) Static obstacles. (c) Dynamic obstacles.

Fig. 10: (a), (b), and (c) show the confusion matrix for the *No Obstacles*, *Static Obstacles*, and *Dynamic Obstacles* environments. Each confusion matrix shows the success rate for the policies saved in *Zone 0* (*Chkpt 0*), *Zone 1* (*Chkpt 1*) and *Zone 2* (*Chkpt 2*) when the end goal is within *Zone 0*, *Zone 1*, and *Zone 2* respectively. *Zone 3* is the region that is not used during training. We use the success ratio to determine the best policy for DQN agent in the environment with no obstacles, static obstacles and dynamic obstacles. We find that *Chkpt 2* is the best performing policy that shows the highest rate of success.

The hardware used in training is an Intel Core-i7 CPU with an Nvidia GTX1080-TI GPU.

Testing Methodology: For testing the policies we train, we evaluate the checkpoints saved at each zone. The hardware we use for testing the policies is the same as the hardware used for training them (Intel Core-i7 with Nvidia GTX1080-TI).

The rationale behind evaluating the policies saved in each zone is two-fold. First, we want to determine the best policy for a particular environment. Second, we want to make sure that the policy does not forget how to navigate in lower zones as we gradually place the end goals in the higher zones.

B. Success Rate Within the Different Environments

Our results across the different environments are shown in Figure 10. Figure 10a, Figure 10b and Figure 10c show the confusion matrix for evaluating the DQN policy trained using curriculum learning on *No Obstacles*, *Static Obstacles*, and *Dynamic Obstacles* environments, respectively. In the figures, *Chkpt 0*, *Chkpt 1*, *Chkpt 2* correspond to policy checkpoints that were saved within *Zone 0*, *Zone 1*, and *Zone 2* respectively. *Zone 3* is special because it is the region where the agent was not trained before, so it has no corresponding checkpoint. Each cell in the confusion matrix represents the success rate of a particular checkpoint in the zone.

For the *No Obstacles* case (Figure 10a), the policy saved at *Zone 2* (*Chkpt 2*) performs the best in reaching to the goal anywhere from *Zone 0* (nearest to the starting position) to *Zone 3* (farthest from starting position). Since there are no obstacles present in the arena, the only time an agent will fail to reach the goal is when it collides the wall or exhaust the maximum number of permissible steps in an episode. Hence, we see a 100% success rate for this environment.

For the *Static Obstacles* case (Figure 10b), the policy saved at *Zone 2* (*Chkpt 2*) is still the best performing policy in navigating around the static obstacles and reaching the goal. Since there are static obstacles in this environment, every episode, the position of the goal and the static obstacles' position keeps changing. *Chkpt 2* achieves 80% success rate in *Zone 0*, 70% success rate in *Zone 1*, 50% success rate in *Zone 2* and 40% success rate in *Zone 3*. The drop in the success rate is not too surprising since there is a higher chance for the agent to collide with obstacles, walls of the arena or exhaust the maximum number of steps permissible in an episode.

For the *Dynamic Obstacles* case (Figure 10c), we see a similar trend as in the *No Obstacles* and *Static Obstacles* environment where the policy saved at *Zone 2* performs the best but with lower success rate at each successive zone. The success rate at *Zone 0*, *Zone 1*, *Zone 2*, and *Zone 3* are 80%, 70%, 70%, and 60%, respectively. The success rate loss can be attributed to an increase in the possibility of collisions because the agent has to navigate around static and dynamic obstacles.

C. Success Rate Across the Different Environments

To study how a policy trained in one environment performs in other environments, we take the best policy trained in the *No Obstacles* environment and evaluate it on the *Static Obstacles* and *Dynamic Obstacles* environments. We do the same for the best policy trained on *Dynamic Obstacles* and assess it on the *No Obstacles* and *Static Obstacles* environments.

The results for the generalization study are tabulated in Table II. We see that the policy trained in the *No Obstacles* environment has a steep drop in success rate from 100% to 39% in *Static Obstacles* and 21% in *Dynamic Obstacles* environment respectively. In contrast, we observe that the policy trained in the *Dynamic Obstacles* environment has increasing success rate from 56% to 84% in the *No Obstacles* and 63% in the *Static Obstacles* environment respectively.

The drop in the success rate for the policy trained in the *No Obstacles* environment is expected because, during its training, the agent might not have encountered a variety of obstacles (static and dynamic obstacles) to learn from as it might have encountered in the other two environments. The same reasoning can also apply to the improvement in the success rate observed for the policy trained in the *Dynamic Obstacles* environment when it is evaluated on the *No Obstacles* and *Static Obstacles* environments.

In general, the agent performs best in the environment where it is trained, which is expected. But we also observe that training an agent in a more challenging environment can yield good results when evaluating in a much less challenging

Policy	No Obstacles	Static Obstacles	Dynamic Obstacles
No Obstacles Chkpt3	1	0.39	0.21
Dynamic Obstacles Chkpt3	0.84	0.63	0.56

TABLE II: Evaluation of the best-performing policies trained in one environment, tested in another environment. We evaluate the policy trained on *Dynamic Obstacles* in *No Obstacles* and *Static Obstacles* environment. Likewise, we also evaluate the policy trained in the *No Obstacles* environment in the *Static Obstacles* and *Dynamic Obstacles* environments.

environment. Hence, having a random environment generator, such as what we have enabled in Air Learning, can help the policy generalize well by creating a wide variety of different experiences for the agent to experience during training.

D. Success Rate Sensitivity to Sensor Input Ablation

In doing policy exploration, one is also interested in studying the sensitivity of the policy towards a particular sensor input. So we ablate the sensor inputs to the policy to understand the effects. We ablate the inputs to the policy one by one and see the impact of various ablation and its success rate.

The policy architecture we used for the DQN agent in this work is multi-modal in nature which receives depth image, velocity measurement V_t and position vector X_t as inputs. The V_t is a 1-dimensional vector of the form $[v_x, v_y, v_z]$ where v_x, v_y, v_z are the components of velocity vector in x, y and z directions at time *'t'*. The X_t is a 1-dimensional vector of the form $[X_{goal}, Y_{goal}, D_{goal}]$, where X_{goal} is the x-coordinate of the goal, Y_{goal} is the y-coordinate of the goal and D_{goal} is the distance to the goal from the agent's current position.

The baseline success rate we use in this study is when all the three inputs are fed to the policy. The velocity ablation study refers to removing the velocity input measurements from policy inputs. Likewise, the position ablation study and depth image ablation study refers to removing the position vector and depth image from the input stream to the policy. The results of various input ablation study are plotted in Figure 11.

For the *No Obstacles* environment, the policy success rate drops from 100% to 50% when velocity measurements are ablated. When the depth image is ablated, we find that the success rate drops to 4% and when the position vector is ablated, the success rate drops to 37%. Similarly, for *Static Obstacles*, we find that if the depth image input is ablated, it fails to reach the destination. Likewise, when the velocity and position inputs are ablated, we observe the success rate drops from 40% to 25%. Similarly, we see a similar observation in a *Dynamic Obstacles* environment where the success rate drops to 0% when the depth image is ablated.

The depth image is the highest contributor to the success of the policy whereas the velocity input is significant but least among the other two inputs. Using Air Learning, researchers can gain better insights into how reliable a particular set of inputs in the case of sensor failures. The reliability studies and its impact on learning algorithms are essential given the kind of application the autonomous aerial vehicles are targeted.

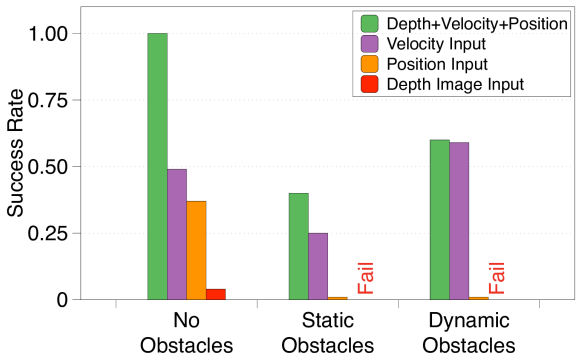


Fig. 11: The effect of ablating the sensor inputs on success rate. We observe that the depth image contributes the most to the success of the policy whereas velocity input affects the least in the success. All the policy evaluation are in `Zone3` on Intel Core i7 platform.

Also understanding the sensitivity of a particular input towards the success can lead to the design of better policies where more feature extraction can be assigned to those set of inputs.

VII. System Evaluation

In this section, we demonstrate how Air Learning can be used to benchmark the performance of algorithm and policy on a resource-constrained onboard compute platform, post training. We use HIL methodology (Section III-C) and QoF metrics (Section III-E) for benchmarking the DQN agent and its policy. We evaluate them on the three different randomly generated environments described in Section VI.

A. Experimental Setup

The experimental setup has two components namely, the server and System Under Test (SUT), as shown in Figure 12. The server component is responsible for rendering the environment (for example, *No Obstacles*). The server consists of an 18 core Intel Core-i9 processor with an Nvidia RTX-2080. The SUT component is the system on which we want to evaluate the policy. The SUT is the proxy for the onboard compute system used in UAVs. In this work, we compare the performance of the policies on two systems namely Intel Core-i7 and Ras-Pi 3. The key differences between the Intel Core-i7 and Ras-Pi 3 platform are tabulated in Table III. The systems are vastly different in their performance capabilities and represent ends of the performance spectrum.

Three latencies affect the overall processing time. The first is t_1, which is the latency to extract the state information (Depth Image, RGB Image, etc.) from the server. The state information is fetched from the server to the SUT. The communication protocol used between the server and the SUT is TCP/IP. Initially, we found that ethernet adapter on Intel Core-i7 faster compared to the ethernet adapter on Ras-Pi 3. We make the t_1 latencies between Intel Core-i7 and Ras-Pi 3 same by adding artificial sleep for Intel Core-i7 platform.[6] The

[6]The sleep latency value that was added to Intel Core-i7 was determined by doing a ping test with the packet size equal to the size of the data (Depth Image) we fetch from the server and averaged it over 50 iterations.

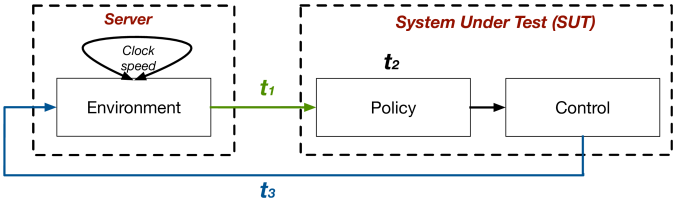


Fig. 12: Experimental setup for policy evaluation on two different platforms. The platform under test is called the System Under Test (SUT). The environments are rendered on a server with Intel Core-i9 with Nvidia RTX 2080. Clock speed is a function in AirSim plugin which allows to speed up the environment time relative to the real world clock. In our evaluation, we set the clock speed to 2X. Time t_1 is the time it takes to get the state information from the environment to the SUT. We use an Intel Core-i7 and a Ras-Pi 3 as the two SUTs. Time t_2 is the time it takes to evaluate the forward pass of the neural network policy. This latency depends on the SUT. It is different for the Intel Core-i7 and the Ras-Pi 3. Time t_3 is the actuation time for which the control is applied.

second latency is t_2, which is the policy evaluation time for the SUT (i.e., the Intel Core-i7 or the Ras-Pi 3). The policies are evaluated on the SUT, which predicts the output actions based on the input state information received from the server. The policy architecture used in this work has 4.4 Million parameters. The t_2 latency on Ras-Pi 3 is 68 ms, while on the desktop, equipped with GTX 1080 Ti GPU and Intel Core i7 CPU, it is 3 ms. The desktop is 20$\times$ times faster.

The third latency is t_3. Once the policies are evaluated, it predicts actions. These actions are converted to the low-level actuation using the AirSim flight controller APIs.[7] These APIs have a duration parameter which controls the duration a particular action must be applied. This duration parameter is denoted by t_3, and it is kept the same for both SUTs.

To evaluate the impact of the SUT on the overall learning behavior, we hold the t_1 and t_3 latencies constant for the Intel Core-i7 and Ras-Pi 3 systems. We focus only on the difference in the policy evaluation time (i.e., t_2) and study how it affects the overall performance time. Using this setup, we evaluate the best policy determined in Section VI for environments with no obstacles, static obstacles, and dynamic obstacles.

B. Desktop vs. Embedded SUT Performance

In Table IV, we compare the performance of the policy on a Intel Core-i7 (high-end desktop) and the Ras-Pi 3. We evaluate the policy on the *No Obstacles*, *Static Obstacles* and *Dynamic Obstacles* environments described previously in Section VI.

In the *No Obstacles* case, the policy running on the high-end desktop is 5% more successful compared to the policy running on the Ras-Pi 3. The flight time to reach the goal on the desktop is on an average is 40.59 s, whereas on the Ras-Pi 3 it is 59.22 s, which yields a performance gap of around 45.88%. The distance flown for the same policy on the desktop

[7]https://github.com/Microsoft/AirSim/blob/master/docs/apis.md

Platform	Intel Core-i7	Ras-Pi 3
CPU Cores	4 x-86	4 Arm-A53
CPU Frequency	4.2 GHz	1.2 GHz
GPU	Nvidia GTX 1080 TI	None
Power	350 W	<1.7 W
Cost	$1500	$35

TABLE III: The most pertinent System Under Test (SUT) specifications for the Intel Core-i7 and Ras-Pi 3 systems.

is 32.57 m, whereas on the Ras-Pi 3 it is 58.59 m, which contributes to a difference of 79.87%. Finally, the desktop consumes on an average 29 kJ of energy, while the Ras-Pi 3 consumes an average of 38 kJ, which is 33.57% more energy.

In the *Static Obstacles* case, the policy running on the desktop is 9% more successful compared to the policy running on Ras-Pi 3. The flight time to reach the goal on the high-end desktop is on average 37.75 s, whereas on the Ras-Pi 3 it is 49.39 s. That yields a performance gap of around 30.85%. For the distance flown, the policy running on the desktop has a trajectory length of 34.38 m, whereas the same policy on the Ras-Pi 3 has a trajectory length of 43.81 m. This contributes to a difference of 27.40%. For energy, the policy running on the desktop on an average consumes 30 kJ of energy, while policy running on Ras-Pi 3 on an average consumes 39 KJ of energy, which is about 32% more energy.

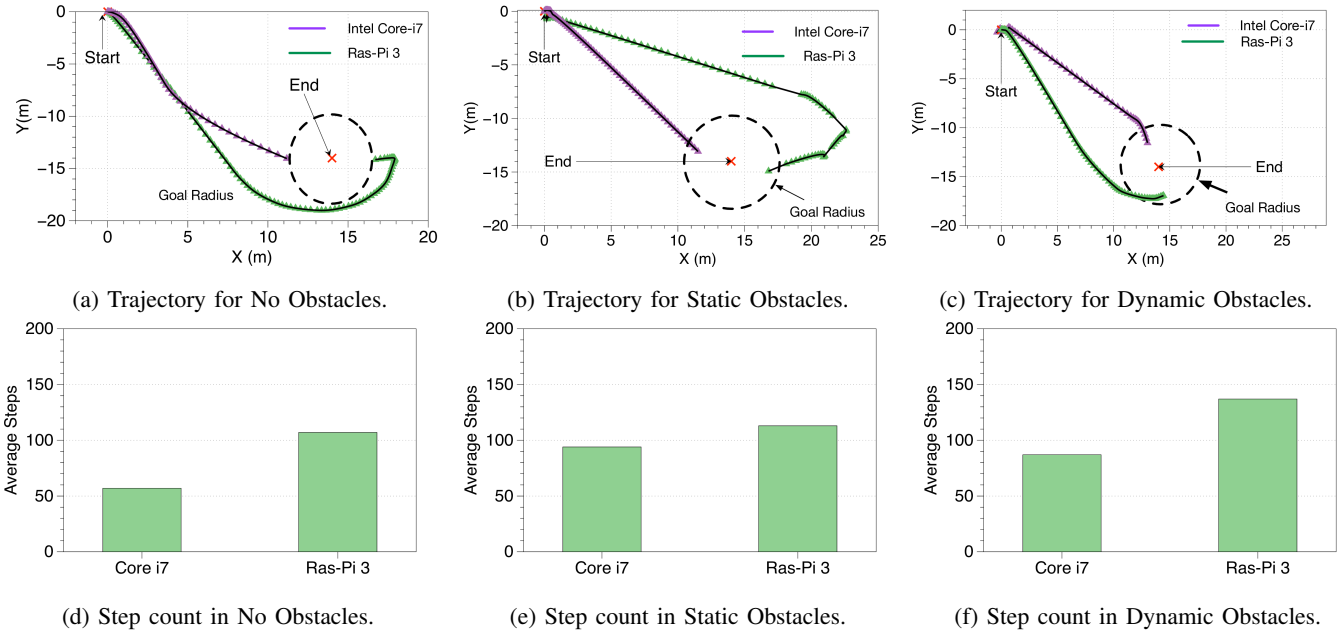
In the *Dynamic Obstacles* case, the success rate between the desktop and the Ras-Pi 3 is 8%. The flight time to reach the goal on the desktop is on average 25.12 s whereas on the Ras-Pi 3 it is 32.41 s, yielding a performance gap of around 29.02%. For the distance flown, the policy running on the desktop has a trajectory length of 34.28 m, whereas the same policy running on Ras-Pi 3 has a trajectory length of 38.76 m. This contributes to a difference of 13.07%. For energy, the policy running on the desktop on average consumes 22.24 kJ of energy while policy running on Ras-Pi 3 consumes 27.09 KJ of energy, which is about 21% more energy.

Overall, across the three different environments, the policy evaluated on the Ras-Pi 3 achieves a success rate that is within 10% compared to the policy assessed on the desktop. While some degradation in performance is expected, the magnitude of the deterioration is more severe for the other QoF metrics, such as flight time, energy and distance flown. This difference is significant to note because when the policies are ported to resource-constrained compute like the Ras-Pi 3 (a proxy for onboard compute in real UAVs), they could perform worse such as being unable to finish the mission due to low battery.

In summary, the takeaway is that evaluations on a high-end machine do not accurately reflect the real-time performance on an embedded compute system such as those available on UAVs. Hence, relying on success rate as the sole metric is insufficient, though this is by and large the state of the art means to report success. By using Air Learning, and its HIL methodology and QoF metrics, we can understand if the choice of onboard compute affects the performance of the algorithm.

	No Obstacles			Static Obstacles			Dynamic Obstacles		
Metric	**Intel Core i7**	**Ras-Pi 3**	**Perf. Gap (%)**	**Intel Core i7**	**Ras-Pi 3**	**Perf. Gap (%)**	**Intel Core i7**	**Ras-Pi 3**	**Perf. Gap (%)**
Inference Latency (ms)	3.00	68.00	2166.66	3.00	68.00	2166.66	3.00	68.00	2166.66
Success Rate (%)	100.00	96.00	4.00	42.00	34.00	9.00	56.00	64.00	8.00
QOF metrics									
Flight Time (s)	40.59	59.22	45.88	37.75	49.39	30.85	25.12	32.41	29.02
Distance Flown (m)	32.57	58.59	79.87	34.38	43.81	27.40	34.28	38.76	13.07
Energy (kJ)	29.01	38.83	33.59	29.66	39.18	32.11	22.24	27.09	21.42

TABLE IV: Inference time, success rate, and Quality of Flight (QoF) metrics between Intel Core i7 desktop and Ras-Pi 3 in Zone3 level for *No Obstacles*, *Static Obstacles*, and *Dynamic Obstacles*. The policy under evaluation is the best policy obtained from policy evaluation (Section VI).



(a) Trajectory for No Obstacles.

(b) Trajectory for Static Obstacles.

(c) Trajectory for Dynamic Obstacles.

(d) Step count in No Obstacles.

(e) Step count in Static Obstacles.

(f) Step count in Dynamic Obstacles.

Fig. 13: Figures (a), (b), (c) compare the trajectories of Ras-Pi 3 and Intel Core-i7. The goal radius is the minimum distance around the goal in which the agent has to reach for it to be successful. Figures (d), (e) and (f) show the average number of steps to reach the goal. The average number of steps is always higher for the Ras-Pi 3 compared to Intel Core-i7.

C. Root-cause Analysis of SUT Performance Differences

It is important to understand why the policy performs differently on the Intel Core i7 versus the Ras-Pi 3. So, we perform two experiments. First, we plot the trajectories of the policy on the Ras-Pi 3 and compare it to the Intel Core-i7 to understand if there is a flight path difference. Visualizing the trajectories helps us build intuition about the variations between the two platforms. Second, we take an Intel Core-i7 platform and degrade its performance by adding artificial sleep such that the policy evaluation times are similar to that of Ras-Pi 3. This helps us validate if it is indeed the processing time that is giving rise to the QoF metric discrepancy.

To plot the trajectories, we fix the position of the end goal, obstacles and evaluate 100 trajectories with the same configuration in the *No Obstacles*, *Static Obstacles*, and *Dynamic Obstacles* environments. The trajectories are shown in Figure 13a, Figure 13b, and Figure 13c. They are representative of repeated trajectories between the start and end goal. The trajectories between the desktop and Ras-Pi 3 are very different—the desktop trajectory orients towards the goal and the proceeds directly. The Ras-Pi 3 trajectory starts toward the goal, but then makes a zig-zag pattern resulting in a longer trajectory. This is likely a result of the actions taken because of stale sensory information, due to the longer inference time; recall there is a 20× difference in the inference time between the desktop and Ras-Pi 3 (Section VII-A and Table IV).

Also, the distance between each step is smaller in the Ras-Pi 3, suggesting that the agent is yawing more (stuck in the same position). Figure 13d, Figure 13e, and Figure 13f show that the total steps taken to reach the goal is higher in Ras-Pi 3 compared to the desktop across all of the environments. These plots suggest that the trajectories are longer to compute.

To further root-cause and test whether the (slower) processing time (t_2) is giving rise to the long trajectories, we take the best performing policy trained on the high-end desktop in the *Static Obstacles* environment and gradually degrade the policy's evaluation time by introducing artificial sleep times

Metric	Core i7	Core i7 (A)	Core i7 (B)	Core i7 (C)
Inference latency (ms)	3.00	43.00	53.00	63.00
QoF metrics				
Flight time (s)	37.75	34.57	53.68	60.38
Distance Flown (m)	34.38	41.89	48.42	55.56
Energy (kJ)	29.6	43.00	46.5	47.7

TABLE V: Degradation in policy evaluation using artificially injected program sleep (proxy for performance degradation). The policy is the best performing policy trained on 'Static Obstacles.' The baseline is Intel Core i7 without any artificial sleep. Intel Core i7 (A), Intel Core i7 (B) and Intel Core i7 (C) represent scenarios where 40 ms, 50 ms, and 60 ms of artificially injected delay added to the policy evaluation.

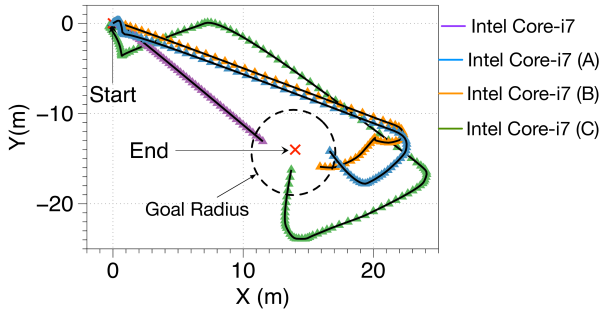


Fig. 14: Trajectory visualaization of the best-performing policy on Intel Core i7 and artificially degraded versions of Intel Core i7 (A), Intel Core i7 (B), and Intel Core i7 (C).

into the program.[8] Sleep time injection allows us to model the big differences in the behavior of the same policy and its sensitivity to the performance of the onboard compute.

Table V shows the effect of degrading the compute performance on policy evaluation. The baseline is the performance on the high-end Intel Core i7 desktop. Intel Core i7 (A), Intel Core i7 (B) and Intel Core i7 (C) are the scenarios where the performance of Intel Core i7 is degraded by 40 ms, 50 ms, and 60 ms, respectively. As performance deteriorates from 3 ms to 60 ms, the flight time degrades by 60%, the trajectory distance degrades by 61%, and energy degrades by 61%.

We visualize degradation impact by plotting the trajectories for the same policy on the baseline Intel Core i7 system and the degraded versions of Intel Core i7 systems (A, B and C). The trajectory results are shown in Figure 14. As we artificially degrade, the trajectories get wider which increases the average number of steps to reach the goal position thus degrading the QoF metrics. We also see that the trajectory of the degraded Intel Core i7 closely resembles the trajectory of the Ras-Pi 3.

In summary, the choice of the onboard compute along with algorithm profoundly affects the resulting UAV behavior and shape of the trajectory. Additional quality of flight metrics (energy, distance, etc.) capture the differences better than just success rate. Moreover, evaluations done purely on a high-end desktop might show lower energy consumption in a mission, but when the solution is ported to real robots, the solution might consume more energy due to sub-par performance of the onboard compute. Using the hardware-in-the-loop (HIL) methodology allows us to identify these differences and other performance bottlenecks arising due to the onboard compute without having to port things to the real robots necessarily. Hence, a tool like Air Learning with its HIL methodology is useful for identifying such differences at the early stage.

To mitigate these variations in the behavior of the policy from a training system to embedded onboard compute systems, one can model these variations in workload performance as noise similar to training a policy to be robust to noisy input [**?**]. However, modeling workload performance for given compute architecture and different architectures requires in-depth characterization [76], [77], [78] and often requires benchmarking suite [79], [80], [81] and simulation tools [82], [83], [84], [85]. Hence having an end-to-end tool like Air Learning can be the starting point for computer architects to characterize the end-to-end learning algorithms and model these characteristics to create robust and performance-aware policies.

VIII. Significance of Energy

Energy is a crucial resource. In this section, we show the significance of using energy in the evaluation of reinforcement learning algorithm and its policy by doing energy infraction studies. Many researchers often overlook energy from their evaluation and focus only on raw mission success or completion rates. We demonstrate that in some cases the agent may successfully complete the navigation task, but in reality the UAV would have ran out of battery if one considered the UAV's battery capacity. So, the UAV would have failed its mission and this should be reflected in the success rate.

A. Experimental Setup for Energy Infractions

An "energy infraction" occurs when the UAV has exhausted the total energy available in its battery but manages to reach the destination (in simulation). To study the energy infractions, we evaluate the policy trained in a dynamic obstacle environment on three arenas namely *Arena 1*, *Arena 2*, and *Arena 3*. *Arena 1* dimension is 50 m X 50 m in area, *Arena 2* dimension is 200 m X 200 m and *Arena 3* dimension is 350 m X 350 m. We also scale the number of static and dynamic obstacles in *Arena 2* and *Arena 3* such that the obstacle density is more or less similar compared to *Arena 1*. We evaluate the policies on Intel Core-i7 and Ras-Pi 3 platform for 100 trajectories.

B. Energy Impact on Mission Success Rate

The results of energy infraction studies are shown in Figure 15. For *Arena 1*, the agent evaluated in Intel Core-i7 and Ras-Pi 3 show no infraction in energy. The reason is that *Arena 1* is small, and hence UAV does not run out of energy.

For *Arena 2*, the agent evaluated on Intel Core-i7 and Ras-Pi 3 without accounting for energy infraction has a success

[8]Adding artificial sleep into the high-end desktop is a simple first-order approximation of the Ras-Pi 3 system. In reality, we cannot fully equate the high-end desktop to the Ras-Pi 3 since there are other differences (e.g., system architecture, memory sub-system, and power).

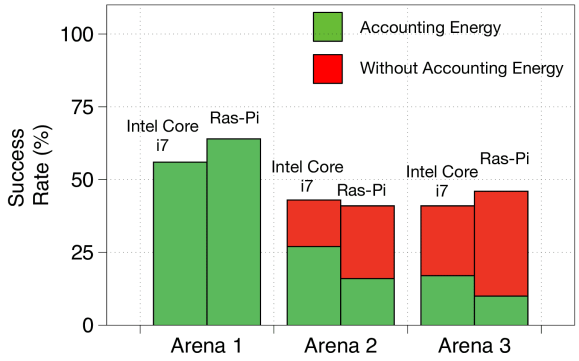


Fig. 15: Energy infractions and its impact on success rate. *Arena 1* area is 50 m X 50 m, *Arena 2* area is 200 m x 200 m and *Arena 3* area is 350 m X 350 m. For large arenas, success rate drops when energy is taken into account because the UAV runs out of battery before it can complete its task.

rate of 43% and 41% respectively. However, with energy infractions, the actual success rate for policy evaluated on Intel Core-i7 and Ras-Pi 3 is 27% and 16% respectively.

Similarly, for *Arena 3*, we observe that the agent evaluated on Intel Core-i7 and Ras-Pi 3 without accounting for energy infraction has a success rate of 41% and 46%. However, with energy infractions, the actual success rate for policy evaluated on Intel Core-i7 and Ras-Pi 3 is 17% and 10% respectively.

Based on this study, we highlight two points. First, success rate alone is not enough to evaluate an algorithm for mobile robots like UAVs that are severely constrained by energy. Hence, any algorithm explicitly designed for UAVs should also include energy in conjunction with success rate. Another way to interpret the significance of this study is that success rate defines the functionality of the algorithm, but energy as a metric used in conjunction with success rate defines the performance which is also equally important. Second, is the importance of the onboard compute platform. We observe that just by evaluating the performance of the policy based on success rate, both Intel Core-i7 and Ras-Pi 3 show similar performance. However, accounting for energy, we observe that Intel Core-i7 typically has a higher success rate compared to Ras-Pi 3. Hence having a better onboard computer can save energy and thus increases the overall range of UAVs.[9]

IX. Air Learning for Microcontroller UAVs

Air Learning is neither limited to aerial navigation tasks nor policies with RGB/IMU input. Different tasks and policy inputs can be evaluated by modifying the environment definition and algorithm exploration steps. Moreover, instead of testing specific hardware platforms only in simulation, it is possible to fully bridge the 'Sim2Real' gap and perform real tests.

We show this by using the same workflow as in Figure 2 for a different task, object avoidance on a Crazyflie UAV, which has severe resource constraints. It is based on a sub-1 Watt micro-controller, and it has severe memory resource constraints. Hence, to design an object avoidance algorithm to run fully onboard on Crazyflie is challenging. So, we conduct "Policy-Hardware Exploration" to train a functional DQN policy in the simulation that can fit into a few kilobytes of memory. We then port the trained model to run fully onboard on Crazyflie and test it in flying conditions (Figure 16).

Environment and Task: We create an Air Learning environment with two dynamic obstacles traveling toward the aerial vehicle. The system is surrounded by four walls, similar to the *Dynamic Obstacles* environment, except the only task the vehicle needs to perform is not to collide with obstacles.

Algorithm Exploration: Instead of the camera RGB input and IMU data, the Crazyflie platform has 5 Time-of-Flight (ToF) sensors using the multiranger add-on deck, which provides distances in the left/right/front/back/up directions. Due to the memory limitations of a microcontroller, we explored DQN policies with two fully connected layers that consume these distance inputs and returns a decision to stay still or move in four directions with a set velocity.

Policy Exploration: Several DQN models with two fully connected layers were trained and evaluated in the simulator that fit into the size constraint of the Crazyflie microcontroller. The explored DQNs had around 20 to 80 hidden units per layer, which fit the 32 kB model size constraint. The network checkpoint was quantized to 8 bits and deployed using Tensorflow Micro, a version of Tensorflow targeting small deployment sizes for microcontrollers.[10]

Hardware Exploration: The Crazyflie platform features easily replaceable parts, so we elected to bridge the 'Sim2Real' gap and also test directly on the real platform. The platform weighs 27 grams, and has dimensions 92 mm $\times$ 92 mm $\times$ 29 mm. The processing is done on an ARM Cortex-M4 operating at 160 MHz on an STM32F405 microcontroller, with 192 KB of SRAM and 1024 KB of flash memory.

Testing: We evaluated the policy on the CrazyFlie by having it hover in place in the presence of moving obstacles. The obstacles were made out of moving Roomba like robots that had styrofoam pillars stacked on them to serve as moving posts. The CrazyFlie was able to avoid them successfully. We also evaluated a more challenging scenario where the CrazyFlie had to avoid a human moving in close proximity (less than one foot). The CrazyFlie successfully avoids the "obstacle" 100% of the time. The video, code, and examples for the CrazyFlie test are open source and available online.[11]

X. Future Work

The Air Learning infrastructure that we built can be used for solving several open problems related to UAVs which spans multiple disciplines. The goal of this work was to demonstrate the breadth of Air Learning as an interdisciplinary tool. To that end, we demonstrate the interdisciplinary nature of our

[9]Assuming increase in compute capability comes with the same form factor of the chip. The trend of achieving better compute within the same area footprint is commonly observed in the semiconductor industry and is famously known as Moore's law.

[10]TF Micro: https://github.com/tensorflow/tensorflow, under the subfolder `lite/experimental/micro`.

[11]CrazyFlie demo: https://www.youtube.com/watch?v=cvO5YOzI0mg. The source code and scripts for putting the TF-micro model on the CrazyFlie are available at: https://github.com/harvard-edge/crazyflie-firmware.

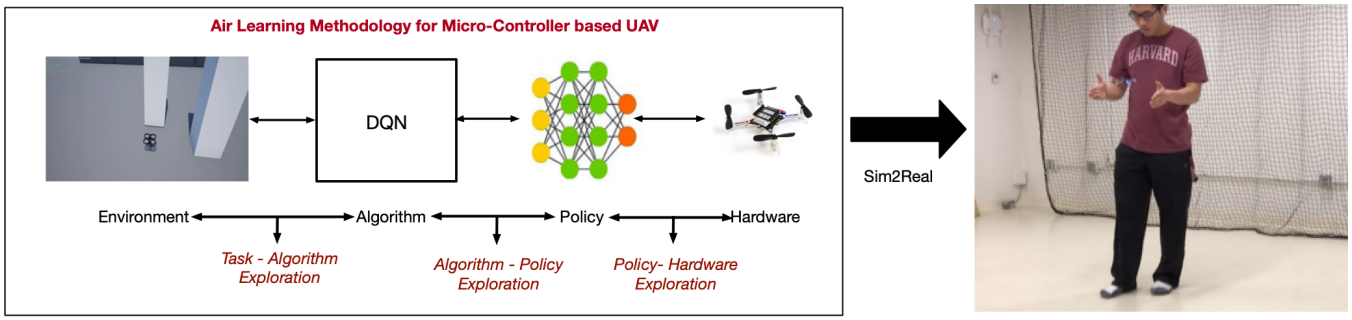


Fig. 16: Air Learning methodology for training a DQN algorithm for dynamic obstacle avoidance. We explore a tiny DQN policy to fit into the memory available in the crazyyflie. Once we determine the best policy for the crazyflie hardware, we verify the functionality of the policy in the Air Learning environment. After verification of the functionality in the simulator, we deploy the learned policy on a real crazyflie drone to see if it avoids dynamic obstacles.

tool by following the methodology described in Figure 2. For a given task (autonomous navigation), we generated various challenging environments (environment generator), explored the best algorithm and its associated policies (algorithm-policy exploration). We evaluated the best policy on different hardware (hardware exploration) and showed the significance of the performance of onboard compute and how it might affect the behavior of policies when ported to real UAVs. In the future, Air Learning can be used to address numerous other questions, including but not limited to the following.

Environments: Different environmental factors can significantly influence the UAVs QoF metrics. For instance, a foggy environment can reduce visibility while wind/turbulence can cause loss of control [86]. In this work, we focus primarily on UAV navigation for indoor applications [87]. Future work can extend Air Learning's environment generator to explore new robust reinforcement learning policies for UAV control under harsh environmental conditions. For instance, since AirSim supports different environmental weather APIs, such as rain, snow, dust and fog, researchers can use the Air Learning environment generator and weather APIs to explore new reinforcement learning algorithms for UAV control under outdoor environments with different weather conditions.[12]

Algorithm Design: Reinforcement algorithms are susceptible to many variables and optimizations, such as hyperparameter tuning, policy architecture, and reward function. Future work could involve using techniques such as AutoML [88] and AutoRL [89] to determine the best hyperparameters, and explore new policy architectures for different UAV tasks.

Another category of future work could expand our work by applying reinforcement learning for multi-agent UAV control [90]. Air Learning supports rapid prototyping of different reinforcement algorithms for UAVs. Also, AirSim allows support for adding multiple UAVs in the same environment. Researchers can combine these and train new reinforcement learning algorithms and policies for multi-agent UAV control.

Policy Exploration: We designed a simple multi-modal policy and kept the policy architecture same across DQN and PPO agent. In future work, one could explore other types of policy architectures, such as LSTM [91] and recurrent reinforcement learning [92]. Also, in this paper, we emphasized the significance of energy as a QoF metric in the evaluation of the policies. Future work could expand our work by exploring energy efficient policies by using the capability available in Air Learning to monitor energy consumption continuously. Energy-aware policies can be associated with open problems in mobile robots, such as charging station problem [93].

System Optimization Studies: Opportunities for future work on the system optimization front can be classified into two categories. First, one can perform a thorough workload characterization for improving the training performance of reinforcement learning from a system standpoint. An accurate characterization followed by optimization will speed up the training process, thus allowing us to build more complex policies and strategies [94] for solving open problems in UAVs. Second, researchers can explore the path to building custom hardware accelerators to improve the onboard compute performance. Having specialized hardware onboard would allow better real-time performance for UAVs (Section VIII).

XI. Conclusion

We develop AirLearning, a cross-disciplinary tool which enables an end-to-end holistic analysis of reinforcement learning algorithms for autonomous aerial vehicles. We use Air Learning to compare the performance of two reinforcement learning algorithms namely DQN and PPO on a configurable environment with varying static and dynamic obstacles. We show that for an end to end autonomous navigation task, DQN performs better than PPO for a fixed observation inputs, policy architecture and reward function. We show that the curriculum learning based DQN agent has a better success rate compared to non-curriculum learning based DQN agent with the same number of experience (steps). We then use the best policy trained using curriculum learning and expose the difference in the behavior of aerial robot by quantifying the performance of the policy using HIL methodology on a resource-constrained Ras-Pi 3. We evaluate the performance of the best policy using quality of flight metrics such as flight time, energy

[12]AirSim plugin weather APIs can be found here: https://github.com/microsoft/AirSim/blob/master/PythonClient/computer_vision/weather.py

consumed and total distance traveled. We show that there is a non-trivial behavior change and up to 79.43% difference in the performance of policy evaluated in high-end desktop and resource-constrained Ras-Pi 3. We also artificially degrade the performance of the high-end desktop where we trained the policy. We observe a similar variation in the trajectory as well as other QoF metrics as observed in Ras-Pi 3 thereby showing how the onboard compute performance can affect the behavior of policies when ported to real UAVs. We also show the impact of energy QoF on the success rate of the mission. Finally, we use the Air Learning policy-hardware exploration to fit a fully functional DQN model on a severely resource-constrained Crazyflie for dynamic obstacle avoidance task.

ACKNOWLEDGEMENTS

The effort at Harvard University and The University of Texas at Austin was sponsored by support from Intel.

REFERENCES

[1] S. Waharte and N. Trigoni, "Supporting search and rescue operations with uavs," in *2010 International Conference on Emerging Security Technologies*, pp. 142–147, IEEE, 2010.

[2] A. Goodchild and J. Toy, "Delivery by drone: An evaluation of unmanned aerial vehicle technology in reducing co2 emissions in the delivery service industry," *Transportation Research Part D: Transport and Environment*, vol. 61, pp. 58–67, 2018.

[3] A. Faust, I. Palunko, P. Cruz, R. Fierro, and L. Tapia, "Automated aerial suspended cargo delivery through reinforcement learning," *Artificial Intelligence*, vol. 247, pp. 381 – 398, 2017. Special Issue on AI and Robotics.

[4] K. Peng, L. Feng, Y. Hsieh, T. Yang, S. Hsiung, Y. Tsai, and C. Kuo, "Unmanned aerial vehicle for infrastructure inspection with image processing for quantification of measurement and formation of facade map," in *2017 International Conference on Applied System Innovation (ICASI)*, pp. 1969–1972, IEEE, 2017.

[5] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016.

[6] H. L. Chiang, A. Faust, M. Fiser, and A. Francis, "Learning navigation behaviors end-to-end with autorl," *IEEE Robotics and Automation Letters*, vol. 4, pp. 2007–2014, April 2019.

[7] F. Sadeghi and S. Levine, "(cad)^{2}rl: Real single-image flight without a single real image," *CoRR*, vol. abs/1611.04201, 2016.

[8] D. Gandhi, L. Pinto, and A. Gupta, "Learning to fly by crashing," *CoRR*, vol. abs/1704.05588, 2017.

[9] R. M. Kretchmar, *A synthesis of reinforcement learning and robust control theory*. Colorado State University Fort Collins, CO, 2000.

[10] B. Wu, W. Chen, Y. Fan, Y. Zhang, J. Hou, J. Liu, J. Huang, W. Liu, and T. Zhang, "Tencent ml-images: A large-scale multi-label image database for visual representation learning," *CoRR*, vol. abs/1901.01703, 2019.

[11] Crazyflie, "Crazyflie 2.0." https://www.bitcraze.io/crazyflie-2/, 2018.

[12] DJI, "Dji-mavic pro." https://www.dji.com/mavic, 2018.

[13] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-fidelity visual and physical simulation for autonomous vehicles," *CoRR*, vol. abs/1705.05065, 2017.

[14] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016.

[15] A. Hill, A. Raffin, M. Ernestus, A. Gleave, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines." https://github.com/hill-a/stable-baselines, 2018.

[16] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48, ACM, 2009.

[17] W. Adiprawita, A. S. Ahmad, and J. Semibiring, "Hardware in the loop simulator in UAV rapid development life cycle," *CoRR*, vol. abs/0804.3874, 2008.

[18] S. Liu, M. Watterson, S. Tang, and V. Kumar, "High speed navigation for quadrotors with limited onboard sensing," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1484–1491, IEEE, 2016.

[19] S. Tang and V. Kumar, "Mixed integer quadratic program trajectory generation for a quadrotor with a cable-suspended payload," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2216–2222, May 2015.

[20] S. Tang, V. Wüest, and V. Kumar, "Aggressive flight with suspended payloads using vision-based control," *IEEE Robotics and Automation Letters (RA-L)*, vol. 3, pp. 1152–1159, April 2018. The first two authors contributed equally to this work.

[21] S. Lupashin, M. Hehn, M. W. Mueller, A. P. Schoellig, and M. Sherback, "A platform for aerial robotics research and demonstration: The flying machine arena," *Mechatronics*, vol. 24, no. 1, pp. 41 – 54, 2014.

[22] N. Michael, D. Mellinger, Q. Lindsey, and V. Kumar, "The grasp multiple micro-uav testbed," *IEEE Robotics & Automation Magazine*, vol. 17, no. 3, pp. 56–65, 2010.

[23] J. P. How, J. Teo, and B. Michini, "Adaptive flight control experiments using raven," *Simulation*, vol. 1, p. 1.

[24] I. Palunko, P. Cruz, and R. Fierro, "Agile load transportation: Safe and efficient load manipulation with aerial robots," *IEEE robotics & automation magazine*, vol. 19, no. 3, pp. 69–79, 2012.

[25] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," *CoRR*, vol. abs/1703.06907, 2017.

[26] N. Valcasara, *Unreal Engine Game Development Blueprints*. Packt Publishing Ltd, 2015.

[27] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, A. Faust, and V. Reddi, "Mavbench: Micro aerial vehicle benchmarking," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 894–907, IEEE, 2018.

[28] M. Plappert, "keras-rl." https://github.com/keras-rl/keras-rl, 2016.

[29] W. Koch, R. Mancuso, R. West, and A. Bestavros, "Reinforcement learning for uav attitude control," *ACM Trans. Cyber-Phys. Syst.*, vol. 3, pp. 22:1–22:21, Feb. 2019.

[30] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2149–2154, 2004.

[31] M. Menard and B. Wagstaff, *Game development with Unity*. Nelson Education, 2015.

[32] J. Weisz, Y. Huang, F. Lier, S. Sethumadhavan, and P. Allen, "Robobench: Towards sustainable robotics system benchmarking," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3383–3389, IEEE, 2016.

[33] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. J. Kelly, A. J. Davison, M. Luján, M. F. P. O'Boyle, G. D. Riley, N. P. Topham, and S. B. Furber, "Introducing slambench, a performance and accuracy benchmarking methodology for SLAM," *CoRR*, vol. abs/1410.2167, 2014.

[34] D. Calisi, L. Iocchi, and D. Nardi, "A unified benchmark framework for autonomous mobile robots and vehicles motion algorithms (movema benchmarks),"

[35] B. León, S. Ulbrich, R. Diankov, G. Puche, M. Przybylski, A. Morales, T. Asfour, S. Moisio, J. Bohg, J. Kuffner, *et al.*, "Opengrasp: a toolkit for robot grasping simulation," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pp. 109–120, Springer, 2010.

[36] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine, "Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation," *CoRR*, vol. abs/1806.10293, 2018.

[37] L. Pinto and A. Gupta, "Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours," *CoRR*, vol. abs/1509.06825, 2015.

[38] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter, "Control of a quadrotor with reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2096–2103, 2017.

[39] A. Giusti, J. Guzzi, D. C. Ciresan, F.-L. He, J. P. Rodríguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. Di Caro, *et al.*, "A machine learning approach to visual perception of forest trails for mobile robots.," *IEEE Robotics and Automation Letters*, vol. 1, no. 2, 2016.

[40] S. Lupashin, A. P. Schoellig, M. Sherback, and R. D'Andrea, "A simple learning strategy for high-speed quadrocopter multi-flips," in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1642–1648, 2010.

[41] A. P. Schoellig, F. L. Mueller, and R. D'Andrea, "Optimization-based iterative learning for precise quadrocopter trajectory tracking," *Autonomous Robots*, vol. 33, no. 1-2, pp. 103–127, 2012.

[42] F. Berkenkamp and A. P. Schoellig, "Safe and robust learning control with Gaussian processes," in *Proc. of the European Control Conference (ECC)*, pp. 2501–2506, 2015.

[43] F. Berkenkamp, A. P. Schoellig, and A. Krause, "Safe controller optimization for quadrotors with Gaussian processes," in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 491–496, May 2016.

[44] I. Palunko, A. Faust, P. Cruz, L. Tapia, and R. Fierro, "A reinforcement learning approach towards autonomous suspended load manipulation using aerial robots," in *2013 IEEE International Conference on Robotics and Automation*, pp. 4896–4901, May 2013.

[45] A. Faust, I. Palunko, P. Cruz, R. Fierro, and L. Tapia, "Learning swing-free trajectories for uavs with a suspended load," in *2013 IEEE International Conference on Robotics and Automation*, pp. 4902–4909, May 2013.

[46] Q. Li, J. Qian, Z. Zhu, X. Bao, M. K. Helwa, and A. P. Schoellig, "Deep neural networks for improved, impromptu trajectory tracking of quadrotors," in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5183–5189, 2017.

[47] E. Kaufmann, A. Loquercio, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza, "Deep drone racing: Learning agile flight in dynamic environments," in *2nd Annual Conference on Robot Learning, CoRL 2018, Zürich, Switzerland, 29-31 October 2018, Proceedings*, pp. 133–145, 2018.

[48] D. Gandhi, L. Pinto, and A. Gupta, "Learning to fly by crashing," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3948–3955, Sep. 2017.

[49] F. Sadeghi and S. Levine, "Cad2rl: Real single-image flight without a single real image," in *RSS*, 2017.

[50] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[51] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016.

[52] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[53] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: hardware-aware automated quantization," *CoRR*, vol. abs/1811.08886, 2018.

[54] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *CoRR*, vol. abs/1311.2901, 2013.

[55] G. Epic, "Ue4 materials." https://docs.unrealengine.com/en-US/Engine/Basics/AssetsAndPackages, 2018.

[56] E. Games, "Ue4 textures." https://docs.unrealengine.com/en-us/Engine/Content/Types/Textures, 2018.

[57] G. Epic, "Wire frame." https://docs.unrealengine.com/en-us/Engine/Rendering/Materialss, 2018.

[58] P.-J. Lai and C.-S. Fuh, "Transparent object detection using regions with convolutional neural network," in *IPPR Conference on Computer Vision, Graphics, and Image Processing*, pp. 1–8, 2015.

[59] K. Berger, R. Voorhies, and L. H. Matthies, "Depth from stereo polarization in specular scenes for urban robotics," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1966–1973, IEEE, 2017.

[60] K. Kjell, "Project title." https:https://github.com/Kjell-K/AirGym, 2018.

[61] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[62] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[63] F. Chollet, "keras." https://github.com/fchollet/keras, 2015.

[64] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[65] Nvidia-Ai-Iot, "Nvidia-ai-iot/redtail."

[66] A. Hummingbird, "Asctec hummingbird." https:http://www.asctec.de/en/uav-uas-drones-rpas-roav/asctec-hummingbird/, 2018.

[67] Intel, "Intel aero ready to fly drone." https:https://www.intel.com/content/www/us/en/products/drones/aero-ready-to-fly.html, 2018.

[68] Parrot, "Parrot bebop-2." https://www.parrot.com/us/drones/parrot-bebop-2-fpv?ref=#parrot-bebop-2-fpv-details, 2019.

[69] Nvidia, "Nvidia xavier." https://developer.nvidia.com/embedded/buy/jetson-agx-xavier-devkit, 2019.

[70] C. Tseng, C. Chau, K. M. Elbassioni, and M. Khonji, "Flight tour planning with recharging optimization for battery-operated autonomous drones," *CoRR*, vol. abs/1703.10049, 2017.

[71] K. R. Kumar, V. Sastry, O. C. Sekhar, D. Mohanta, D. Rajesh, and M. P. C. Varma, "Design and fabrication of coulomb counter for estimation of soc of battery," in *2016 IEEE International Conference on Power Electronics, Drives and Energy Systems (PEDES)*, pp. 1–6, IEEE, 2016.

[72] E. Games, "Wire frame." https://docs.unrealengine.com/en-us/Engine/UI/LevelEditor/Viewports/ViewModes, 2018.

[73] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter, "Control of a quadrotor with reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2096–2103, 2017.

[74] P.-H. Su, D. Vandyke, M. Gasic, N. Mrksic, T.-H. Wen, and S. Young, "Reward shaping with recurrent neural networks for speeding up on-line policy learning in spoken dialogue systems," *arXiv preprint arXiv:1508.03391*, 2015.

[75] K. Judah, A. P. Fern, P. Tadepalli, and R. Goetschalckx, "Imitation learning with demonstrations and shaping rewards," in *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[76] M. Calzarossa and G. Serazzi, "Workload characterization: A survey," *Proceedings of the IEEE*, vol. 81, no. 8, pp. 1136–1150, 1993.

[77] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 412–423, 2007.

[78] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory system characterization of commercial workloads," in *ACM SIGARCH Computer Architecture News*, vol. 26, pp. 3–14, IEEE Computer Society, 1998.

[79] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pp. 41–51, IEEE, 2010.

[80] K. M. Dixit, "The spec benchmarks," *Parallel computing*, vol. 17, no. 10-11, pp. 1195–1209, 1991.

[81] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, "Dawnbench: An end-to-end deep learning benchmark and competition,"

[82] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[83] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: a binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, p. 22, ACM, 2004.

[84] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[85] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[86] E. A. Ranquist, M. Steiner, and B. Argrow, "J3. 1 exploring the range of weather impacts on uas operations,"

[87] Y. Khosiawan and I. Nielsen, "A system of uav application in indoor environment," *Production & Manufacturing Research*, vol. 4, no. 1, pp. 2–22, 2016.

[88] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *CoRR*, vol. abs/1707.07012, 2017.

[89] H.-T. L. Chiang, A. Faust, M. Fiser, and A. Francis, "Learning navigation behaviors end-to-end with autorl," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 2007–2014, 2019.

[90] M. Hüttenrauch, A. Sosic, and G. Neumann, "Deep reinforcement learning for swarm systems," *CoRR*, vol. abs/1807.06613, 2018.

[91] B. Bakker, "Reinforcement learning with long short-term memory," in *Advances in neural information processing systems*, pp. 1475–1482, 2002.
[92] X. Li, L. Li, J. Gao, X. He, J. Chen, L. Deng, and J. He, "Recurrent reinforcement learning: A hybrid approach," *CoRR*, vol. abs/1509.03044, 2015.
[93] T. Kundu and I. Saha, "Charging station placement for indoor robotic applications," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3029–3036, IEEE, 2018.
[94] OpenAI, "Openai five." https://blog.openai.com/openai-five/.