2023 IEEE International Conference on Robotics and Automation (ICRA 2023)
May 29 - June 2, 2023. London, UK

Parallel Reinforcement Learning Simulation for Visual Quadrotor Navigation

Jack Saunders[1], Sajad Saeedi[2], and Wenbin Li[1]

***Abstract*— Reinforcement learning (RL) is an agent-based approach for teaching robots to navigate within the physical world. Gathering data for RL is known to be a laborious task, and real-world experiments can be risky. Simulators facilitate the collection of training data in a quicker and more cost-effective manner. However, RL frequently requires a significant number of simulation steps for an agent to become skilful at simple tasks. This is a prevalent issue within the field of RL-based visual quadrotor navigation where state dimensions are typically very large and dynamic models are complex. Furthermore, rendering images and obtaining physical properties of the agent can be computationally expensive. To solve this, we present a simulation framework, built on AirSim, which provides efficient parallel training. Building on this framework, Ape-X is modified to incorporate parallel training of AirSim environments to make use of numerous networked computers. Through experiments we were able to achieve a reduction in training time from 3.9 hours to 11 minutes, for a toy problem, using the aforementioned framework and a total of 74 agents and two networked computers. Further details including a github repo and videos about our project, PRL4AirSim, can be found at `https://sites.google.com/view/prl4airsim/home`**

I. Introduction

Visual Quadrotor navigation is the method of utilising vision sensors to navigate in an environment. Classical navigation pipelines which include sensing, mapping, and planning can lead to latency and compounded error due to the sequential execution of tasks [1]. Furthermore the time complexity of these tasks leads to limited navigation capabilities due to limited size, weight, and power constraints of quadrotors. Very recently, researchers have utilised RL to perform various tasks such as: aerial filming [2], mapless exploration [3], autonomous landing [4], counter drone operations [5], and collision avoidance [6]. However, RL can be very data inefficient, requiring a significant number of simulation steps for an agent to display favorable behaviours. This issue can be compounded by the computationally expensive simulated sensor calculations [7]. For these data-driven approaches, the current issue within the field of visual quadrotor navigation is the time to train agents, see Table 1. Currently training times can take tens of hours and sometimes even days. Typically researchers will create toy problems for debugging purposes. However, these are not always representative of the problem and there is no guaranty of an agent converging to a high episode reward. Furthermore, RL fails silently which further emphasises the need for quicker training times. Therefore, we present a simulation framework, built on AirSim, which provides efficient parallel training. Furthermore, we incorporate a modified Ape-X networking architecture which can be used with AirSim's remote server to make use of distributed training and numerous networked computers. We use the term parallel RL to refer to the distributed training process. This is not to get confused with distributed RL [8], which investigates the value distribution. Through experimentation, We show a dramatic increase in efficiency of using 74 total agents with two computers and a speed up of training time from 3.9 hours to 11 minutes, as shown in Fig. 1.

(a) (b)

Fig. 1: Simulation of 74 agents showing exploration behaviour at the start (a) and greedy behaviour end (b) of training.

II. Literature Review

A. Techniques to improve data inefficiency

Improving the data efficiency of RL is a popular research area. For roboticists, adding demonstration runs from an expert user or alternative controller can aid with better initialization. Pienroj use a PID ground-truth controller at the start of the training run for powerline tracking [9]. Using an exploration training policy, the RL agent was able to refine the policy. This approach presents difficulties; an expert might not be available, and it might be impractical to create an alternative controller for a given problem.

For off-policy based methods, a buffer is used to store transitions, which are sampled from, to train the neural network (NN). Prioritising what samples are chosen can replay important transitions more frequently and hence learn more efficiently [10].

Multiple training environments and/or agents can be used to obtain observations in parallel. Ape-X decouples acting from learning, where actors interact with separate instances of the environment and select actions according to a shared policy [11]. This framework can be used with any off-policy algorithm such as Deep Q-Learning. Additionally, off-policy agents can be parallelised such as advantage and asynchronous advantage Actor-Critic-based algorithms [12]. Both algorithms send gradients, with respect to the parameters of the policy, to a central parameter server. Similar to Ape-X, IMPALA is an alternative algorithm which communicates trajectories of experiences to a centralised learner [13]. Based on the IMPALA architecture, asynchronous Proximal Policy Optimization (PPO) [14] is a policy gradient algorithm which uses a surrogate policy loss function with clipping. More

This work is supported by the UKRI Centre for Doctoral Training in Accountable, Responsible & Transparent AI (ART-AI), under UKRI grant number EP/S023437/1.

[1]Department of Computer Science, University of Bath, UK, {`js3442, w.li`}`@bath.ac.uk`

[2]Department of Mechanical and Industrial Engineering, Toronto Metropolitan University, Toronto, Canada, `s.saeedi@torontomu.ca`

recently, Decentralised Distributed PPO provides further improvements by allowing each worker's GPU to be used for both sampling and training [15].

Training distributed architectures in simulators, like the aforementioned, can benefit from physics acceleration. Physics and frame rendering can take place asynchronously [16]. As a result, it is possible to increase the speed of the simulation without increasing the frame rate of the renderer. Simulator physics engines typically run on a fixed time-step and reducing the time-step can lead to higher a fidelity simulation while increasing wall-time. Alternatively, some simulators allow researchers to speed up the relative wall-clock of the simulator leading to faster simulations. However, increasing the relative wall-clock of the simulator can lead to undetected collisions which is problematic for collision avoidance tasks [17]. Alternatively, GPU-accelerated physics engines have been shown to overcome CPU bottlenecks. Liang used GPU-accelerated RL simulations to train continuous-control locomotion tasks [18].

B. Robotic Simulators

Advanced robotic simulators have been built on top of modern 3D games engines. Self-driving cars benefit from the Unreal Engine's high fidelity renderer and physics engines. For aerial vehicle simulators, AirSim [17] also benefits from Unreal Engine but incorporate quadrotor dynamics for aerial vehicle navigation problems. Flightmare [19], a more recent aerial simulator, is built on top of Unity's platform. They make use of parallel threading and decouple the quadrotor's dynamic modelling from the rendering engine for fast dynamic simulation. Both quadrotor simulators make use of an asynchronous messaging library, Remote Procedural Call for AirSim and ZeroMQ for Flightmare. Both libraries have used this architecture as it is currently not possible to train NNs using these game engines (Except for inference purposes using Onnx). Flightmare provides excellent functionality for vectorised environments by extending their `vec_env` in flightlib. Here, we have tailored our method towards specifically reducing training time for visual quadrotor navigation specifically and the overheads caused from gathering image observations. Furthermore, further work will investigate performing our framework on Flightmare to make use of the parallel dynamic model calculations to further reduce training time. Other aerial simulators include RotorS [20] using Gazebo which can utilise OGRE and OptiX rendering engines, and Gym-pybullet-drones [21] which has a highly accurate physics engines but lacks high fidelity rendering for visual-navigation tasks. Furthermore, PyRep is a toolkit which interfaces with CoppeliaSim which can be used to generate quadrotor simulations [22], [23]. For more information regarding simulators, the reader should direct their attention to this survey [24].

C. RL for Robotic and quadrotor Navigation

Early work of parallel deep RL had been used for visual-motor skill acquisition for physical robotic manipulators [**?**], [25]. More recently, using Isaac Sim, Ridin *el al.* trained a quadroped to walk with thousands of quadropeds in parallel which was validated on a real robot [26]. The authors used PPO with an observation space consisting of: velocities, joint position and velocities, previous actions, and distance measurements from the robot body to the terrain. Finally, Song *et al.* [27] use Deep RL to compute near-time-optimal trajectories for drone racing which is tested on a physical quadrotor. The state consists of relative gate observations and velocity, acceleration, orientation, and body rates. They can simulate 100 environments in parallel but don't suffer from the rendering overhead of visual-navigation problems.

D. RL based Visual Quadrotor Navigation

RL-based Visual quadrotor navigation techniques have been applied to a wide range of problems. One such example is mapless exploration, Jang *et al.* utilise hindsight intermediate targets within the environment and validates the approach on a ground robot [3]. Autonomous landing is a well established field which has benefited from RL [4]. Furthermore, security measures surrounding quadrotors has recently gained popularity. Çetin *et al.* use RL to capture target quadrotors without crashing with obstacles [5]. Finally, collision avoidance has been thoroughly researched. Shin found continuous based Actor-Critic networks to perform best in cluttered environments [28]. Furthermore, Bézier curves have been investigated as motion primitives for action selection [29]. Generative Adversarial Networks have been explored to generate depth maps from a single image and then used as an input to an RL-based high-level planner to overcome computational power constraints on quadrotors [30]. As can be seen in Table I, training times can take tens of hours and sometimes even days to complete. More recently, researchers have utilised parallel architectures to improve training time. Jang *et al.* train 5 soft Actor-Critic agents in parallel for mapless navigation [3]. It is unclear if 5 actors perform within the simulator or 5 individual instances of the unreal engine. We take the phrasing of '5 worker threads' to convey the latter. Running 5 individual instances of the Unreal Engine leads to huge overheads, instead we propose vectorising the environment such that these 5 agents can act within the same environment instance and benefit from the memory and processing efficiency. This includes less assets being loaded in memory, allowing for larger replay buffers on centralised architectures and additional total agents for distributed architectures. Similar to us, Fang *et al.* [31] and Devo *et al.* [32] have used multiple agents within a single simulator instance. Although the vectorised environment provides less computational burden compared to Jang's study, the authors could benefit from a distributed architecture to utilise multiple environment instances.

Our contributions are the following: (1) Our study improves upon the state-of-the-art by providing a more effective way of vectorising the environment, increasing the number of agents within a single simulation instance , thus improving space complexity. To do so, we incorporate a non-interactive environment setup by preventing rendering and collisions of all agents. Also, we improve the quantity of quadrotors in

a given simulator by modifying AirSim's functionality for parallel training using batched rendering and asynchronous episode learning which was not possible before. (2) Finally we formulate a modified version of Ape-X, for AirSim simulations, to provide a distributed learning framework to get up to 74 quadrotors in two individual environments on two networked computers.

III. Method

A. Deep Q-Learning

Our work makes use of Reinforcement Learning (RL), an agent-based modelling technique that studies the interactions between the environment and the agent. RL is formulated as a Markov Decision Process, defined by a tuple $(\mathcal{S},\mathcal{A},P,R)$. For each time step t, an agent in state $s \in \mathcal{S}$ takes an action $a \in \mathcal{A}$. By executing the action a, the agent receives a reward $R(s,a)$ and reaches a new state s' determined by the probability distribution $P(s'|s,a)$. The objective of the RL agent is to maximise the expected future cumulative discounted reward $V(s) = \mathbb{E}[G_t|S_t = s]$, where $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ and γ is the discount factor $\gamma \in [0,1]$. In this study we use Deep Q-Learning [33], where the aim is to estimate the state-action value function $Q(s,a)$. The value function governs the agent's policy by calculating the expected returns for the state-action pairs, where $a_t = argmax_a Q(s,a;\theta)$. In this case the value function is parameterised using a neural network, also known as a Deep Q-network DQN which updates the network weights θ_i based on the loss function $L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(R + \gamma \max_{a'} Q(s',a';\theta_i^-)\right) - Q(s,a;\theta_i)\right]$. where $\mathcal{D}$ is the replay buffer and θ_i^- is the target network used to stabilise training.

B. Problem Formulation

We showcase the parallel simulation framework, based on AirSim, to simulate multiple agents performing the task of collision avoidance. To do this, we utilise both an Inertial Measurement Unit (IMU) and depth camera which has been simulated within AirSim. The IMU provides linear acceleration measurements, while the depth camera provides a mask of distances for each pixel. The DQN is used as a high-level controller [33] to learn the representation between the state and action to avoid obstacles.

State: The state space consists of stacked depth images and the linear velocity from the IMU, simulated within AirSim. The Depth image and linear velocity have a resolution of (32×32) and (1×3) respectively. Image stacking was first introduced by Mnih *et al.* [33], and now utilised for quadrotor visual navigation which incorporates memory into the state space [29].

Action: We use a small action space consisting of left and right high level commands with a magnitude of 0.25m/s, $\mathcal{A} = \{0.25, -0.25\}$. The relative simulation clock is set to 4x the wall-clock and the action time step is 1s.

Network Architecture: The network architecture can be seen in Fig. 3. The depth image is processed by convolutional layers with ReLu activation and maxpooling, and the resulting latent space is flattened and concatenated with the quadrotor's linear velocity. The concatenated output then passes through two additional layers. The output vector size is the number of actions $|\mathcal{A}|$ which represents the quality values $Q(s,a)$. These fully connected layers also use the ReLu activation function. For the reinforcement learning agent, we use an experience replay size $|\mathcal{D}|$ of 15000 and mini-batches of 32. The target network is updated every 150 steps and a discount factor γ of 0.99 is used. Before any training occurs, the experience replay is filled with 15000 transitions, using a random action selection policy, $\varepsilon = 1$. Once filled, the Adam optimiser is used to update the parameter weights of the NN. We also then linearly decrement ε using the number of experiences in the buffer $\varepsilon = \max(0, 1 - \frac{a_T - |\mathcal{D}|}{|\mathcal{D}|})$. Where a_T is the total number of actions performed by all agents.

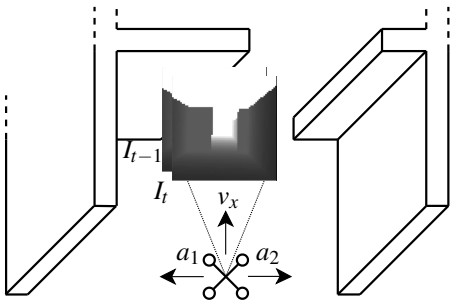

Fig. 2: Our study involves a 2D visual navigation toy problem, where each time step involves collecting an observation from the environment, including the current and previous depth images, as well as the current linear velocity. Actions are selected through ε-greedy selection, consisting of high-level commands of either go left or go right, which adjust the desired horizontal velocity. A constant forward velocity of $1m/s$ is sent to the flight controller.

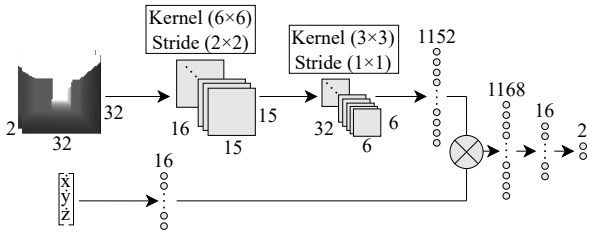

Fig. 3: The Deep Q-learning network architecture consists of two convolutional layers fed with the depth image at concurrent time-steps. The resulting latent space is concatenated with the linear velocity of the quadrotor. The output are the state-action value of both actions. Note, for our case $\dot{x} = 1$ and $\dot{z} = 0$.

Reward Function: We use a simple reward function to train the agent. A negative reward of -100 is given to the agent when it collides with the walls of the arena. Then, for each step it survives without colliding, the agent receives a positive reward of $+3$.

C. Simulation Setup

We modify AirSim [17] to provide improved capabilities for parallel reinforcement learning. We make use of the quadrotor physics model, sensor models and flight controller. AirSim is a plugin for the Unreal Engine which provides a high fidelity rendering engine. Using the Blueprint system

we create an event graph which ignores overlapping events caused by other quadrotors, thus disabling agent collisions. Furthermore we make all quadrotors hidden in the scene capture to prevent the depth map rendering other agents. These modifications allow us to vectorise the simulation and control multiple agents within the same Unreal Engine instance in a non-interactive way. By sharing the same virtual space, agents don't need their own individual environments loaded, resulting in less complex setups with fewer meshes required for navigation.

The training and test environments created within the Unreal Engine is depicted in Fig. 4. Both environments have a variety of obstacles which have been inspired from previous works of Camci *et al.* [29] and Shin *et al.* [6]. We understand recently, research has been done to apply RL collision avoidance controllers for very complex tasks. Our aim here is illustrate a comparably competent controller at the same time as reducing training time. While contrasting simulation time for one versus multiple agents.

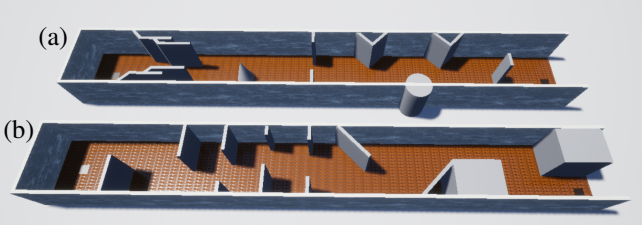

Fig. 4: Environments to Train (b) and Test (a) the visual-based collision avoidance RL agent. The aim is to navigate through the environment, starting on the left and traversing to the right without colliding into a wall.

D. Batched Image Requests Through Single Synchronisation

The original command to request images from AirSim took the form `simGetImages([ImageRequests], vehicle_name, external)`. The underlying issue with this command in the context of parallel reinforcement learning is the repetitive requests of individual images for each quadrotor. An image request is an object which contains metadata on the image that is requested to be rendered. This includes, for example, the type of image such as depth or RGB, resolution, and also FOV. For this purpose, we require the same image metadata but for all vehicles within the simulator. On-top of convenience, requesting multiple images this way can cause latency issues within the simulator. AirSim requires a connection between the Unreal Engine and a python or c++ client using a Remote Procedure Call (RPC) server. This is due to incompatibility issues of using NN libraries within Unreal Engine. The `simGetImages` command is sent asynchronously on the client side, however is computed synchronously on the AirSim server side where both the game and render threads are synchronised. The synchronicity of both threads is caused by obtaining both physical properties of the camera and rendering the image at a specific time-step. Hence, every call from `simGetImages` causes a delay in order to process the request, which can scale with more agents in the simulator.

Therefore, we add the functionality to request images from all vehicles within the simulator with one command in the same thread tick. We call this new command `simGetBatchImages([ImageRequests], [vehicle_names])`. We modify the AirSim plugin to incorporate the game logic to render all images within the same game thread. To differentiate between both the original AirSim command `simGetImages` and our `simGetBatchImages` command, we refer to these as non-batched and batched respectively. The current downside of this approach is all images for all vehicles are required to be of the same type. Further work will investigate how to abstract away from our current rigid messaging protocol to allow for multiple sensor types to be called at once for data fusion techniques. Furthermore, another limitation is the assumption that all agents act simultaneously at the same time-step.

E. Vectorised Asynchronous Training

Currently AirSim only contains the functionality to reset the entire simulator, `reset()`, by placing all vehicles at the initial spawn location. This is inefficient as the episode ends until all agents have reached a terminal state. This has an impact on the training time as experiences can only be gathered during flight.

Instead, we add the functionality to reset individual quadrotors and to spawn at a specified location with `resetVehicle (vehicle_name, pose)`. This function provides the capability to train agents asynchronously and increase the rate at which experiences are generated. Furthermore, we introduce `moveByVelocityZBatchAsync (..., vehicle_names)` to reduce API calls and better conform to the vectorised framework.

F. Ape-X, Parallel Reinforcement Learning

We modify Ape-X to incorporate the additional RPC server node created within AirSim. Fig. 5 illustrates the block diagram for the proposed distributed framework.

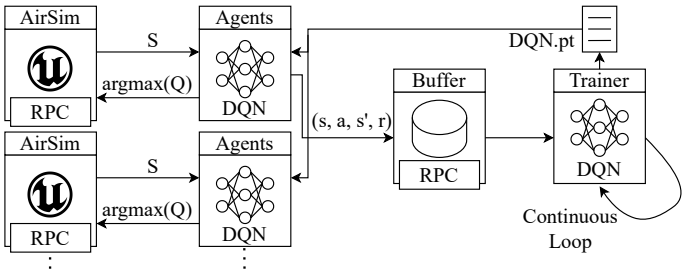

Fig. 5: Parallel networking architecture based off Ape-X consisting of local policy networks performing actions on AirSim instances. Separate servers are used for the replay buffer and trainer which continuously samples from the buffer to train the global NN weights.

Local NN clients (Agents) are individually paired with AirSim instances. For each local client, states are obtained using our batch render technique. Actions are calculated using local instances of the global DQN network hosted

on the Trainer client. For every episode termination, the local NN parameters are updated with the global parameters from the Trainer client. After every step of an episode, the experience consisting of the state, action, next state, and reward for each agent within the local AirSim instance, (s, a, s', r) are sent to a buffer. This buffer contains a set of the most recent experiences with first-in-first-out replacement. Once the database is filled, the Trainer client continuously requests 32 mini-batches of experiences to perform gradient descent and update the global NN parameters. Decoupling the trainer and clients provides the benefits of Ape-X. The shared replay buffer enables faster data collection and an increased diversity of experiences.

The distributed framework can be run on networked processes, as a result of using RPC servers for AirSim instances and the experience replay buffer. Performing gradient updates using the Trainer Client on a separate thread leads to an increase in rate of parameter updates as the updates are asynchronous to data generation and action selection, such as in [34]. For all experiments, we keep the training frequency constant at 50Hz.

IV. Results

We illustrate the delay from repeatedly synchronising the game and render thread by comparing the batched (ours) and non-batched render requests. Then, we show the impact of this effect on the client side where we analyse the average time to obtain the quadrotor rendered image and linear velocity for a range of agents within an environment. Finally, we demonstrate the reduction in training time, for the given collision avoidance task, accomplished by comparing the training time against the number of agents within the simulator. The main goal is to showcase the reduction in training time due to the structure of the training framework.

A. Batched and Non-Batched Comparison Results

First, we analyse the delay caused by the game and render thread synchronisation, when requesting quadrotor states. As mentioned before, this delay is required to ensure the pose of the vehicle matches the location where the image is rendered. When requesting 10 quadrotor states individually, we calculate the synchronisation delay to be an average of 14.2ms. Comparably, requesting all quadrotor states in a batch, using one API call and one thread synchronisation, leads to an average delay of 1.6ms as shown in Fig. 6(b).

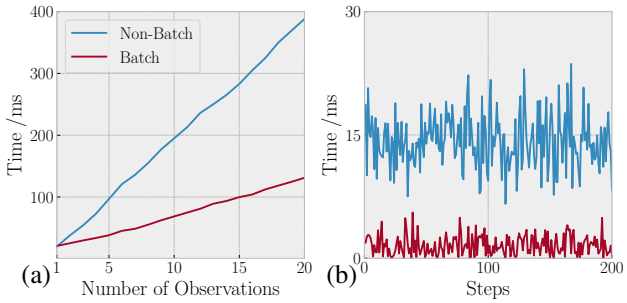

Fig. 6: (a) presents the computation time to render quadrotor states for both batched and non-batched techniques. (b) presents the magnitude of the delay caused by synchronising the game and render thread either once or multiple times for 10 quadrotors, using the batched and non-batched method.

Then, we analyse the effect of thread synchronisation from the perspective of the RPC client. Similarly, when requesting 10 quadrotor states individually, or non-batched, using 10 separate API calls leads to an average time of 197ms to generate. Whereas, when requesting all 10 quadrotor states together, or batched, in a single API call leads to 61ms. We show the comparison between the two methods in Fig. 6(a).

Further work to profile the RPC server and client is required to understand the additional discrepancies caused by the additional RPC calls from the non-batched method. We anticipate additional overheads are caused by the python global interpreter lock when running RPC on a python node. Furthermore if the data requested by a client is not immediately available, i.e gathering simulated data, the server has to wait until this data is processed [37]. The delay when waiting for data can be alleviated using a multi-threaded server.

B. Parallel Collision Avoidance Agents

we show the efficiency of using Ape-X with a fully vectorised environment using the new collective render technique. The vectorised environment involves multiple non-interactive agents and multiple environments all running in parallel to train a single policy. We present a comparison,

TABLE I: Comparison of state-of-the-art work, illustrating the training time and state space for various problems. Furthermore, computer power is listed and the number of agents used to train the policy. Our work increases the number of agents and reduces training time using similar hardware. Further work will include a more complete analysis by implementing the following works using our framework.

Author	**Agents**			**Image State Dimensions**	**Training**		**Compute Power**		
	Envs	Agents	Total		Steps	Time	CPU	GPU	RAM
Kersandt 2018 [34]	1	1	1	(30×100)	150,000	41 hours	-	-	-
Shin 2019 [28]	1	1	1	$(4 \times 64 \times 64)$	100,000	-	Intel i7 3.4 GHz	Nvidia Titan X	-
Çetin 2020 [35]	1	1	1	(30×100)	50,000	14 hours	Intel i7	Nvidia GTX 1060	16GB
Polvara 2020 [4]	1	1	1	$(4 \times 84 \times 84)$	-	7.6 days	Intel i7	Nvidia Tesla K-40	32GB
He 2020 [36]	1	1	1	(80×100)	50,000	-	-	-	-
Çetin 2021 [5]	1	1	1	(30×100)	75,000	48 hours	Intel i7	Nvidia GTX 1060	16GB
Singla 2021 [30]	1	1	1	(84×84)	300,000	-	Intel i7	Nvidia GTX 1050	8GB
Fang 2021 [31]	1	9	9	(64×64)	-	29 mins	Intel i7-9700	Nvidia RTX 2080Ti	64GB
Jang 2022 [3]	5	1	5	$(3 \times 84 \times 84)$	4,000,000	33h	Intel i9-9900K	Nvidia RTX 2080Ti	-
Devo 2022 [32]	1	8	8	(84×84)	2,700,000	-	Intel i9-9900K	2×Nvidia RTX 2080Ti	64GB
Ours	2	37	74	$(2 \times 32 \times 32)$	65,000	11 mins	Intel i7-11700k	Nvidia RTX 2070	64GB

shown in Fig 7, of both the batched and non-batched methods for both a single quadrotors and 50 quadrotors running in parallel. The rate at which states were rendered for both methods of a single quadrotor was similar, as expected. This is because the thread is only being paused once for both methods and as a result observation sample rate was the same, shown in Fig. 7(b). Whereas for 50 quadrotors, obtaining observations in batches resulted in quicker training time as a result of the increased efficiency, shown in Fig. 7(c).

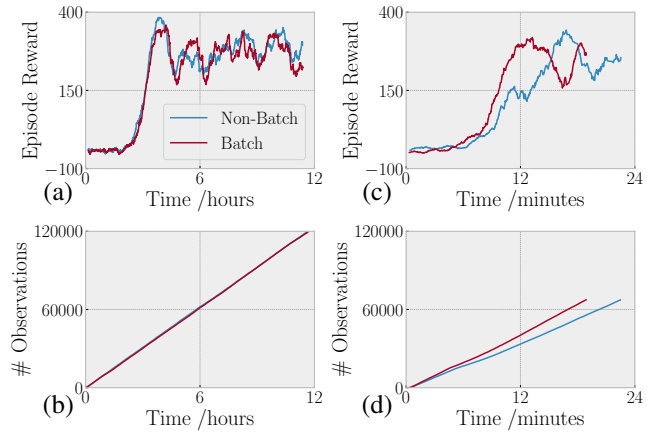

Fig. 7: Comparison of episode reward and number of observations rendered for both batched and non-batched methods using 1 (a),(b) and 50 (c),(d) quadrotors in parallel. Reduction in training time is observed for 50 quadrotors, but not for 1 quadrotor.

All experiments are shown in Table II. Tests were performed using an Intel i7-11700K CPU, Nvidia RTX 2070 Super GPU, and 64GB of RAM. To illustrate the computational power used when training, metrics on CPU and GPU utilisation with GPU power consumption were collected. Training time consists of time for the model to converge to an episode reward of 300. We validated our model on the test arena to show generalisation of the policy, shown in Fig. 4. The average success rate of all experiments was 78%, with most number of collisions occurring with the cone. We believe this is due to the unique geometry of the shape, which the network had no prior experience with, and field of view from the camera.

As expected, increasing the number of agents collecting experiences resulted in shorter training times. Furthermore, we noticed a configuration of 25 quadrotors acting in parallel using batched rendering achieved roughly the same time to converge as the non-batched method using 50 quadrotors. The fastest training time was achieved with 74 agents acting in parallel over 2 environments with 11 minutes to converge. Whereas, the slowest total training time took 3.9 hours to converge.

The total RAM utilised by the distributed framework for a single environment amounts to 7.3GB. This scaled by approximately 4GB for each additional simulation instance. Although, we only utilised approximately 2GB of RAM for the replay buffer, larger state spaces for more complicated tasks such as the ones identified in Table I illustrate the need for additional RAM. Distributing the computational resources over many computer allows for larger state spaces and replay buffer sizes.

TABLE II: Comparison between the distributed and non-distributed framework. Batched rendering performed slightly better, however CPU utilisation sharply rose at 50 agents. This was solved by using two instances of AirSim to train a total of 74 agents.

Threading Method	**Agents**			**Train Time**	**Utilisation%**		**GPU Power**
	Envs	Agents	Total		CPU	GPU	
Non-Batched Observations	1	1	1	3.8h	24%	53%	101W
	1	50	50	18m	62%	59%	95W
Batched Observations	1	1	1	3.9h	24%	54%	92W
	1	25	25	17m	27%	53%	93W
	1	50	50	14m	73%	55%	95W
	2*	37	74	11m	31%	56%	91W

* Environment instances run on networked computers to overcome the CPU bottleneck.

We notice CPU usage dramatically increases after 37 quadrotors. This illustrates the requirement for fast quadrotor dynamics as shown in Fightmare [19]. Therefore, it can be assumed for any given computer specification, a sweet spot exists where a maximum number of quadrotors can be used for the fastest throughput. For this, we would need to run further experiments. We attempted to run simulations with 100 and 200 quadrotors in a single environment with no increase in performance. For our setup, the sweet-spot was approximately 37 quad rotors per environment where training times converged. We also noticed the frequency of gradient calculations of the NN reduced when multiple AirSim instances were run on a single computer. For two instances, the training frequency resulted in 14Hz, further illustrating the need for the parallel framework we present here.

V. Conclusion

In this paper, we improve upon the state-of-the-art by providing a more efficient way of distributed agent training for quadrotor visual navigation. We introduce a non-interactive vectorized environment in order to mitigate memory costs and enable asynchronous training for all agents. Batched rendering is explored to alleviate redundant game and render synchronization, resulting in increased data throughput and reduced training time. These modifications to AirSim are used within a modified Ape-x architecture which allows for multiple AirSim instances to run concurrently. Our method enables scalable parallel operation where we experimented with 74 quadrotors utilizing two individual environments across two networked computers. The greater number of agents resulted in a significant reduction of training time from 3.9 hours to just 11 minutes. Future work will investigate how GPU-accelerated physics engine can improve the performance of training. Also, a more systematic comparison will be done by implementing the following approaches in Table I using our framework and while analysing the effects on different quadrotor simulators mentioned in Section 2. Furthermore, we will test an agent trained using our framework on a physical quadrotor, investigating the the sim-to-reality gap.

REFERENCES

[1] A. Loquercio, E. Kaufmann, R. Ranftl, M. Müller, V. Koltun, and D. Scaramuzza, "Learning high-speed flight in the wild," *Science Robotics*, vol. 6, Oct. 2021.

[2] K. Goh, R. Ng, Y.-K. Wong, N. Ho, and M. Chua, "Aerial filming with synchronized drones using reinforcement learning," *Multimedia Tools and Applications*, vol. 80, pp. 18125–18150, May 2021.

[3] Y. Jang, J. Baek, and S. Han, "Hindsight Intermediate Targets for Mapless Navigation With Deep Reinforcement Learning," *IEEE Transactions on Industrial Electronics*, vol. 69, pp. 11816–11825, Nov. 2022.

[4] R. Polvara, M. Patacchiola, M. Hanheide, and G. Neumann, "Sim-to-Real Quadrotor Landing via Sequential Deep Q-Networks and Domain Randomization," *Robotics*, vol. 9, p. 8, Feb. 2020.

[5] E. Çetin, C. Barrado, and E. Pastor, "Counter a Drone and the Performance Analysis of Deep Reinforcement Learning Method and Human Pilot," *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, 2021.

[6] S.-Y. Shin, Y.-W. Kang, and Y.-G. Kim, "Reward-driven U-Net training for obstacle avoidance drone," *Expert Systems with Applications*, vol. 143, p. 113064, Apr. 2020.

[7] D. Hanover, A. Loquercio, L. Bauersfeld, A. Romero, R. Penicka, Y. Song, G. Cioffi, E. Kaufmann, and D. Scaramuzza, "Autonomous Drone Racing: A Survey," Jan. 2023. arXiv:2301.01755 [cs].

[8] M. G. Bellemare, W. Dabney, and R. Munos, "A Distributional Perspective on Reinforcement Learning," July 2017. arXiv:1707.06887 [cs, stat].

[9] P. Pienroj, S. Schonborn, and R. Birke, "Exploring Deep Reinforcement Learning for Autonomous Powerline Tracking," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, (Paris, France), pp. 496–501, IEEE, Apr. 2019.

[10] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," Feb. 2016. arXiv:1511.05952 [cs].

[11] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver, "Distributed Prioritized Experience Replay," Mar. 2018. arXiv:1803.00933 [cs].

[12] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," June 2016. arXiv:1602.01783 [cs].

[13] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu, "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures," in *International conference on machine learning (PMLR)*, pp. 1407–1416, June 2018.

[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," Aug. 2017. arXiv:1707.06347 [cs].

[15] E. Wijmans, A. Kadian, A. Morcos, S. Lee, I. Essa, D. Parikh, M. Savva, and D. Batra, "DD-PPO: Learning Near-Perfect PointGoal Navigators from 2.5 Billion Frames," Jan. 2020. arXiv:1911.00357 [cs].

[16] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A General Platform for Intelligent Agents," May 2020. arXiv:1809.02627 [cs, stat].

[17] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," July 2017. arXiv:1705.05065 [cs].

[18] J. Liang, V. Makoviychuk, A. Handa, N. Chentanez, M. Macklin, and D. Fox, "GPU-Accelerated Robotic Simulation for Distributed Reinforcement Learning," in *Conference on Robot Learning (PMLR)*, pp. 270–282, Oct. 2018.

[19] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A Flexible Quadrotor Simulator," in *Conference on Robot Learning (PMLR)*, pp. 1147–1157, May 2021.

[20] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, "RotorS—A Modular Gazebo MAV Simulator Framework," in *Robot Operating System (ROS): The Complete Reference (Volume 1)* (A. Koubaa, ed.), pp. 595–625, Cham: Springer International Publishing, 2016.

[21] J. Panerati, H. Zheng, S. Zhou, J. Xu, A. Prorok, and A. P. Schoellig, "Learning to Fly – a Gym Environment with PyBullet Physics for Reinforcement Learning of Multi-agent Quadcopter Control," in *International Conference on Intelligent Robots and Systems (IROS)*, (Prague, Czech Republic), pp. 7512–7519, IEEE/RSJ, Mar. 2021.

[22] S. James, M. Freese, and A. J. Davison, "PyRep: Bringing V-REP to Deep Robot Learning," June 2019. arXiv:1906.11176 [cs].

[23] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison, "RLBench: The Robot Learning Benchmark & Learning Environment," in *IEEE Robotics and Automation Letters*, pp. 3019–3026, Feb. 2020.

[24] J. Saunders, S. Saeedi, and W. Li, "Autonomous Aerial Delivery Vehicles, a Survey of Techniques on how Aerial Package Delivery is Achieved," *arXiv:2110.02429 [cs, eess]*, Oct. 2021.

[25] A. Yahya, A. Li, M. Kalakrishnan, Y. Chebotar, and S. Levine, "Collective Robot Reinforcement Learning with Distributed Asynchronous Guided Policy Search," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 79–86, Sept. 2017.

[26] N. Rudin, D. Hoeller, P. Reist, and M. Hutter, "Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning," in *Conference on Robot Learning (PMLR)*, pp. 91–100, Jan. 2022.

[27] Y. Song, M. Steinweg, E. Kaufmann, and D. Scaramuzza, "Autonomous Drone Racing with Deep Reinforcement Learning," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (Prague, Czech Republic), pp. 1205–1212, IEEE, Sept. 2021.

[28] S.-Y. Shin, Y.-W. Kang, and Y.-G. Kim, "Obstacle Avoidance Drone by Deep Reinforcement Learning and Its Racing with Human Pilot," *Applied Sciences*, vol. 9, p. 5571, Dec. 2019.

[29] E. Camci, D. Campolo, and E. Kayacan, "Deep Reinforcement Learning for Motion Planning of Quadrotors Using Raw Depth Images," in *2020 International Joint Conference on Neural Networks (IJCNN)*, (Glasgow, United Kingdom), pp. 1–7, IEEE, July 2020.

[30] A. Singla, S. Padakandla, and S. Bhatnagar, "Memory-Based Deep Reinforcement Learning for Obstacle Avoidance in UAV With Limited Environment Knowledge," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, pp. 107–118, Jan. 2021.

[31] J. Fang, Q. Sun, Y. Chen, and Y. Tang, "Quadrotor navigation in dynamic environments with deep reinforcement learning," *Assembly Automation*, vol. 41, pp. 254–262, July 2021.

[32] A. Devo, J. Mao, G. Costante, and G. Loianno, "Autonomous Single-Image Drone Exploration With Deep Reinforcement Learning and Mixed Reality," *IEEE Robotics and Automation Letters*, vol. 7, pp. 5031–5038, Apr. 2022.

[33] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[34] K. Kersandt, G. Munoz, and C. Barrado, "Self-training by Reinforcement Learning for Full-autonomous Drones of the Future*," in *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, (London), pp. 1–10, IEEE, Sept. 2018.

[35] E. Çetin, C. Barrado, and E. Pastor, "Counter a Drone in a Complex Neighborhood Area by Deep Reinforcement Learning," *Sensors*, vol. 20, p. 2320, Apr. 2020.

[36] L. He, N. Aouf, J. F. Whidborne, and B. Song, "Deep Reinforcement Learning based Local Planner for UAV Obstacle Avoidance using Demonstration Data," *arXiv:2008.02521 [cs]*, Aug. 2020. arXiv: 2008.02521.

[37] K. Qureshi and H. Rashid, "A Performance Evaluation of RPC, JAVA RMI, MPI and PVM," *Malaysian Journal of Computer Science*, vol. 18, no. 2, pp. 38–44, 2005.