Abstract

Unmanned aerial vehicle control for object avoidance is no easy task. This paper uses the recent popularity and effectiveness of end-to-end deep reinforcement learning to train an autonomous unmanned aerial vehicle to navigate static objects in a simulated environment. Specifically, Asynchronous Advantage Actor-Critic, a state-of-the-art deep reinforcement learning algorithm, is used to train the agent. Monocular RGB images are used as input to indirectly facilitate the closing of the 'reality gap', which is the problem of transferring a policy learned in simulation to reality, as it can be expensive or difficult to add extra cameras to real unmanned aerial vehicles. Unreal Engine and AirSim are used to create the photorealistic and physics-realistic training environments.

I Introduction

Researchers from around the world are designing methods to optimize processes to train robotics to perform certain tasks autonomously. Many researchers have been using machine learning, reinforcement learning (RL), and deep reinforcement learning (DRL) to automate robots and unmanned aerial vehicles (UAVs) alike [Tai et al., 2016]. These new technologies are attempting to solve many of today's problems by having robots perform the dull, dirty, and dangerous jobs that humans are currently doing.

A Robot Control

Various forms of learning have been used in today's robotics. Some methods used are RL, DRL, imitation learning, transfer learning, and Simultaneous Localization and Mapping (SLaM) [Tai et al., 2016]. These methods have been used in robotics to perform tasks such as image classification, semantic segmentation, navigation, and obstacle avoidance. Various DRL algorithms have been developed to increase the functionality and reliability of robotics. Autonomous intelligent robotics require two building blocks: perception and control [Tai et al., 2016]. The robot perceives its surroundings, takes in information, and then processes that information to make a decision.
B  **UAV Control**

UAVs have become prominent in today’s society. Creating new methods to achieve autonomy and reliability has been the focal point for many researchers. Autonomous drones promote better safety standards with obstacle avoidance, increase productivity, and establish a new paradigm of human and technology interaction. The main goals for autonomy in drones at the moment are localization, navigation, obstacle recognition and obstacle avoidance.

C  **Reality Gap**

The reality gap refers to the difficulty of effectively transferring a control policy learned in simulation to reality [Mouret et al., 2013]. This can be difficult because many simulators cannot effectively mimic the physics, photorealism or control constraints of reality. Unreal Engine allows for realistic physics and photorealistic environments to be crafted. AirSim is a plugin for Unreal Engine that allows for the realistic flight of a UAV [Shah et al., 2018]. Using a realistic simulator can help with the transfer problem, however, it has been found in many papers that diversity in images and environments for training actually helps bridge the gap more than realistic simulation [Tobin et al., 2017]. This paper does not test this theory but still uses Unreal and AirSim as components of a realistic simulation to help indirectly bridge the gap, possibly for future work.

D  **Monocular Cameras and Other Hardware**

Solely using a single monocular camera to feed in information of nearby objects for navigation and object avoidance for UAVs is a non-trivial control problem [Ross et al., 2012]. However, it is of great research importance to learn control policies that can deal with only using monocular cameras and RGB images as adding in additional cameras, such as depth detecting cameras [Henry et al., 2012] or constructing depth images in the algorithm itself [Xie et al., 2017], can be difficult or error-prone. Another useful image type is object/semantic segmentation. Depth and RGB images have been used to construct object segmentation images [Cao et al., 2017]. However, this is not easy to do and can require too much training to effectively construct segmentation images while training a control policy.

E  **Object Detection with Deep Learning**

Deep learning has been effectively used for multiple applications in both research and industry such as object detection [Krizhevsky et al., 2012], medical imaging [Litjens et al., 2017] and air analysis [Qi et al., 2018]. Deep Convolutional Neural Networks (CNNs) became prominent in 2012 with AlexNet for the classic ImageNet object classification problem [Krizhevsky et al., 2012]. Since then research into using CNNs to detect objects and classify them has flourished. They have also been used in many DRL agent architectures for navigation and object avoidance [Gandhi et al., 2017, Sadeghi and Levine, 2016, Hong et al., .]. In this paper object avoidance is the main goal and therefore a CNN is used as the base of the architecture.

F  **Deep Reinforcement Learning**

DRL has gained momentum in a diverse set of applications such as playing video games [Mnih et al., 2013], controlling robots [Gandhi et al., 2017], chatbot training [Serban et al., 2017], and many others [Arulkumaran et al., 2017]. The current effectiveness of DRL grew out of the popular algorithm Deep Q-Learning from DeepMind [Mnih et al., 2013]. Many other algorithms have grown from this field including Asynchronous Advantage Actor-Critic (A3C) [Mnih et al., 2016] which is the algorithm used in this paper.
G Simulation Training

Our goal is to use these techniques in a new way to optimize the training process for autonomous drones. Specifically, we train a UAV agent end-to-end with A3C to navigate in realistically textured and lit rooms in simulation using RGB images from a monocular camera.

The organization of this paper is as follows: I Introduction, II Related Works, III Background, IV Experiments, V Results, VI Future Work, VII Conclusion and finally, VIII Contributions

II Related Works

This paper belongs in the domain of autonomous UAV control and deep reinforcement learning. In this section we discuss related works to our own in these fields.

A UAV and other Robot Control

A variety of control methods have been used in works for UAV and other robot control. There are various use cases for autonomous robotics and drones as well. Different forms of localization and navigation techniques are actively being employed. To aid in perception many robotics and drones use devices such as: a monocular camera, stereo vision, LiDar, RGB-D camera, laser range sensor, IMU, and ultrasonic sensors [Chakravarty et al., 2017]. Many robotics have also used SLaM to navigate autonomously [Dijkshoorn, 2012]. The addition of these devices can improve perception of environments but can also add cost, weight, and computational strain. N. Dijkshoorn used SLam in his paper involving the AR.Parrot drone [Dijkshoorn, 2012]. Some researchers are using techniques such as imitation learning and human in the loop [Reddy et al., 2018] to train their autonomous agents. Imitation learning consists of mimicking human behavior in a given task to jumpstart performance [Tai et al., 2016]. Human in the loop combines autonomous behavior with human intervention. The agent and operator work together to complete a task. One group used human in the loop to play a game in which a pilot and copilot had to maneuver a lunar lander without crashing it. They used an AI they created called LAGGYPILOT to assist the human. They found “that a copilot which leverages input from LAGGYPILOT outperforms the solo LAGGYPILOT and solo copilot: the combined pilot-copilot team crashes and goes out of bounds less often, uses less fuel, follows stabler trajectories, and finds the landing site more often than the other two solo teams” [Reddy et al., 2018].

B Deep Reinforcement Learning for Navigation and Object Avoidance

Deep reinforcement learning gained ground and research popularity when DeepMind trained an agent with Deep Q-Learning to be able to perform at a superhuman level on most of the games on the platform Atari [Mnih et al., 2013]. A3C is another state-of-the-art DRL algorithm also developed by DeepMind [Mnih et al., 2016]. Our training method uses A3C as the principal learning and control algorithm. Recently, UAV control and object avoidance have been combined, such as in our paper, for cutting-edge drone control. Researchers have trained a real drone to learn how to avoid objects and navigate through crashing data [Gandhi et al., 2017]. Another paper, attempts to bridge the reality gap, and transfers a policy fully learned in simulation to the real-world [Sadeghi and Levine, 2016]. Through diversifying the environments seen through the textures and objects they were able to achieve a good zero-shot transfer to the real-world. Another paper attempts to transfer a trained policy through fine tuning in the real-world [Hong et al., ]. They use predicted segmentation maps for objects to help assist the robot through cluttered and uncluttered hallways and use A3C [Hong et al., ]. We have not found many sources that attempt to use AirSim, a realistic simulator, to train a UAV with DRL. We did find one however that trained a UAV agent in AirSim to navigate a space of obstacles [Kersandt, 2018] that worked well. However, the agent in this paper is trained using A3C while theirs used a Deep Q-Learning Network (DQN).
III Background

This section explains deep reinforcement learning and Asynchronous Advantage Actor-Critic in more detail.

A Deep Reinforcement Learning

The reinforcement learning setting is where an agent interacts with an environment $\varepsilon$ and through this interaction learns to better perform a task or multitude of tasks. Specifically, at each timestep $t$ the agent receives a state $s_t$ from the environment and outputs an action $a_t$ which affects the environment in some way. Given $a_t$ and the state the environment is in $s_t$, a scalar reward signal $r_t$ and updated state $s_{t+1}$ are sent back to the agent. The actions can be chosen from a discrete set of possible actions or from a specific range in a continuous action space setting. The policy $\pi$ decides which action is taken given the state. $\pi$ can be any function, but usually is a parameterized function approximator, where the optimal set of parameters are learned over time, such as a deep neural network if the environment dynamics are complex and/or stochastic. This process repeats until the episode ends by the agent reaching a terminal state which triggers a terminal flag of true being sent to the agent. However, this process can reset for however many episodes are needed or in the continuous learning domain there are no episodes only a single run. This paper uses a discrete action set and episodic learning. The total return from timestep $t$ to the end of the episode or infinite horizon is $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$. A discount factor $\gamma \in (0, 1]$ is used to lessen the impact of future rewards from $t$ as the agent must optimize action selection to maximize expected return. The maximization of future expected return is a non-trivial problem when the agent is given a complex environment without known dynamics and it must learn purely from the environment’s reward function through this simple process.

Two important functions are the state value function and state-action pair value function. The state value function under the policy is defined as $V^\pi(s) = \mathbb{E}[R_t | s_t = s]$ and is the expected return given $s_t$. The state-action pair value function (also known as the Q-value) under the policy is defined as $Q^\pi(s, a) = \mathbb{E}[R_t | s_t = s, a]$ and is the expected return given $s_t$ and $a_t$. Deep reinforcement learning refers to using a deep neural network to represent the parameterized policy $\pi_\theta$ or the approximate value functions: $V(s; \theta), Q(s, a; \theta)$ where $\theta$ represents the parameters of the approximators.

The two main kinds of reinforcement learning algorithms are value-based and policy-based algorithms. Value-based algorithms attempt to learn the state value function and/or state-action pair value function and then overlay a way to pick the action given this information. One important value-based algorithm is Q-learning where the agent learns to learn the optimal action-value function defined as $Q^\ast(s, a) = \max_{a_t}Q^\pi(s, a)$. In the case of DRL the approximate action value function is defined as $Q(s, a; \theta)$. An $\epsilon$-greedy policy overlay can then be used where at each timestep there is a $1 - \epsilon$ chance to pick the action with the highest Q-value and a $\epsilon$ chance to pick a random action. This method is usually better than a full greedy policy as it allows for exploration of the environment. This paradigm is known as the exploration-exploitation problem, where a balance must be found to help the agent to find optimal states while also using this found information in optimal ways. Policy-based algorithms directly parametrize the policy $\pi(a|s; \theta)$ to pick an action given a state, instead of overlaying a policy, such as $\epsilon$-greedy, on top of the value prediction. REINFORCE is an important policy-based algorithm that updates the policy’s parameters in the direction $\nabla_\theta \log \pi(a_t|s_t; \theta) R_t$ using gradient descent. Using $R_t$ increases the variance of the updates so a baseline is usually subtracted off of this estimate in increase stability by decreasing differences between updates. The baseline is usually a learned state value approximation which turns the gradient update into: $\nabla_\theta \log \pi(a_t|s_t; \theta) R_t - b_v(s_t)$.

B Asynchronous Advantage Actor-Critic

The agent learns by using a state-of-the-art form of Actor-Critic called Asynchronous Advantage Actor-Critic or A3C [Mnih et al., 2016]. Advantage actor-critic is a mix of value-based and policy-based algorithms where a parameterized policy $\pi(a_t|s_t; \theta)$ and parameterized state value function approximator $V(s_t; \theta_v)$ are used and learned by the agent. An advantage function estimate is used to update the parameters: $\nabla_\theta \log \pi(a_t|s_t; \theta) R_t - b_v(s_t)$. 

4
∇_θ logπ(α_t|s_t; θ) A(s_t, a_t; θ, θ_v) which decreases the bias. An n-step forward look at the next n rewards is factored into the advantage estimate which also uses the value estimate as a baseline. So now an estimate of the advantage function is given by: \[ \sum_{k=1}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v) \] where \( k \) varies depending on if there is a terminal state in the rollout in the n-steps ahead. The n-step lookahead and the use of the state value estimate allows for a decrease in variance while giving the agent enough information for the advantage estimate. The agent picks actions and explores for up to \( t_{\text{max}} \) steps or when a terminal state is reached and then this trajectory information in the form of \((s_t, a_t, r_t, s_{t+1})\) tuples for every step is used to update the parameters.

The asynchronous part of this algorithm means that there are multiple agents each with their own instance of the environment. Each agent has a local copy of the global network and uses this trajectory information to get the gradient update using their respective local copies (which all share copies of the same parameters) and then send the gradient update to the global network to update it. Finally, the new parameters are sent back to all of the local copies. In addition to the value function loss and policy loss in practice it is also a good idea to add an entropy loss into the mix for increased exploration. The multiple running agents and aggregated updates makes this a state-of-the-art algorithm by speeding up training, stabilizing learning and decorrelating experience tuples seen.

The entropy \( H \) of the policy is added to the objective function to encourage exploration. The gradient change of the full objective function is given by:

\[
\nabla_\theta \logπ(α_t|s_t; \theta) A(s_t, a_t; \theta, \theta_v) + β\nabla_\theta H(π(s_t; \theta))
\]

where \( β \) is a hyperparameter that controls the weighting of the entropy.

IV Experiments

A3C was used as the training algorithm for the agent to learn to fly without colliding with obstacles. The architecture consists of AlexNet convolutional layers pretrained on ImageNet [Krizhevsky et al., 2012]. This convolutional body is followed by a fully-connected layer consisting of 4096 units with a relu activation function and then another fully-connected layer consisting of 1024 units with a relu activation function. Then finally both the actor and critic streams. Therefore, most of the parameters are shared amongst the actor and critic in the network. The architecture takes as input a single RGB image and outputs the critic value (state prediction value) and actor values (action probabilities) for the allowed actions of going forward, turning left or turning right. Movement was therefore constrained to only the X and Y axes and not the Z axis. The total loss includes an actor loss, critic loss and entropy loss, which is used to drive exploration.

Adam is used to optimize the parameters of the network given the losses [Kingma and Ba, 2014]. Unreal Engine was used to craft the environments. This engine allows for the use of realistic textures and lighting which combined with the realistic physics of AirSim furthers the complexity of object avoidance. If the goal was to jump start training in a real environment then the realism exhibited by our experiments would help do so. The agent had access only to a single RGB image for the state at each timestep, so having realistic and bright enough rooms is important for the quality of the RGB images. If the rooms were too dark then the agents would have a much harder time discerning features of the room. The camera is on the front of the UAV and has a FOV of 90 degrees.
Two experiments with different goals, reward functions and initial agent placements were run and analyzed. 8 agents total were used in both experiments for A3C. Therefore, 8 rooms were created (figure 1.1), each housing one UAV. The first experiment consisted of training the global network (parameters randomly initialized) through having the agents fly around and try to not hit any of the walls. The rooms were small enough as to force the agents to deal with the walls. Each agent started at different positions around the room to simulate random initial positions for the purpose of the agent not being able to memorize the actions that it should take but actually learn to avoid objects or walls in this case given an image. In addition each agent had a random chance of facing to the left, right or staying straight at the start of each episode i.e. a random initial orientation. The reward function consisted of -1 for crashing, +0.1 for going forward and +0.01 for turning (left or right). Crashing and reaching the max number of actions/steps allowed per episode both resulted in terminal states and the episode was reset. The reward function was crafted to encourage going forward or covering ground and discourage crashing.

The second experiment conducted consisted of training the global network (parameters randomly initialized) by having the agents go towards a white wall (figure 1.2) on the opposite side of the room from which it started. Each agent started on a different part of the starting wall across from the white goal wall and the same random initial orientation method from experiment one was used. The wall was colored white so it could be visually discerned more easily for faster learning. The reward function consisted of -1 for crashing, +1 for reaching the goal wall, and for going forward the change in distance from the agent to the goal wall from the previous step to the current step (which was normalized to be between 0 and 1 and then divided by 10 as to not affect the learning as much as avoiding crashing and reaching the goal). Colliding with walls and reaching the goal triggered terminal states and flags with a reset in episode. This reward function was designed to help lead the agent to the goal and discourage colliding with objects (or walls in this case).

Two baselines were tested along with the trained agents. One was a random agent that selected actions completely at random. Then a human that attempted the environments.

V Results

![Figure 2.1](image1.png) ![Figure 2.2](image2.png)

Experiment one consisted of training for 1560 global (total amongst all agents) episodes or about 127500 global steps and testing on the same environment for 50 episodes. The graphs for the testing phase are included in figure 2. Figure 2.1 indicates the number of actions taken per episode and figure 2.2 indicates the reward received per episode for experiment one. All of the graphs show average data because both the human and the trained A3C agent stray very little from their respective averages indicated for both experiments. The random action agent’s data was also averaged. The max episode lengths for testing was 200 steps. Both the agent and the human never crashed into a wall as indicated. However, the reward was consistently about
2 for the trained A3C agent while the human tester averaged about 14. So while the A3C agent was able to avoid all walls it did not move forward as often (which gives higher reward than turning) as the human tester. The random agent averaged about 22 actions per episode and -0.7 total reward per episode showing that the A3C agent did learn quite a lot past its randomly initialized parameters. Qualitatively, by the end of training the agent learns to go towards walls but then turn to avoid collisions. This can be seen by the fact that the agent during testing never collides with a wall. However, the agent does not learn to effectively go forward and turn only when necessary like the human tester does from the start. This is likely because the reward function gives too much reward for turning so the agent instead of learning to move efficiently to collect reward instead learns to turn most of the time to avoid collision. The distribution of actions is indeed heavily skewed to turning (specifically turning left as this seems to be the action it latched on to selecting).

Experiment two consisted of training for 1390 global episodes or about 36100 global steps and testing for 50 episodes with the same environment. Figure 2.3 indicates the number of actions taken per episode and figure 2.4 indicates the reward received per episode for experiment 2. As indicated by the graphs the A3C agent received very close to the amount of average reward and average number of actions per episode as the human tester. Also it is important to note that in experiment two, unlike experiment one, a lower number of steps per episode is better as the goal was not to fly around but to get to a goal. Both the human tester and A3C agent did substantially better in reaching the goal than the random action agent. Qualitatively, the A3C agent learns to go forward only or turn when necessary because of the random starting orientation. This is learned very quickly as the goal of crossing the room is very easy. Therefore, the trained A3C achieves very close to optimal performance across reward and number of actions per episode during testing.

VI Future Work

The current research available along with our contribution on the area of end-to-end autonomous UAV navigation and object avoidance only scratches the surface of what is possible given the success of DRL in other areas. Given our lack of time and computational resources there are many areas of improvement and possible future work.

A Object Avoidance and Dynamic Environments

The environments trained on consisted of no internal objects, only walls. The initial formulation of this paper and resulting experiments included a curriculum training of increasingly difficult environments (through increasing quantity of objects and diversity of objects/textures) along with testing transfer to novel layouts and objects never seen before in training. This would be the first area to apply future iterations as training on different objects in an attempt to transfer the knowledge to unseen objects and layouts is in the spirit
of closing the ‘reality gap’. In addition, most real rooms and environments contain dynamic or moving objects and people which the agent would need to be able to avoid. However, to infer velocity enhancements and changes to the architecture are necessary. Frame stacking, stacking multiple images from the current timestep and past timesteps, as input instead of using only the current timestep image would allow the agent to infer velocity of moving objects [Mnih et al., 2013]. In addition, recurrent layers can learn temporal dependencies of the data or in this case can help the agent remember where an object is going and it’s velocity.

B  Finer Control and Altitude Control

The agent trained in this paper had three possible actions: going forward for a set duration and speed and turning both left or right given an angle. These actions are very high level and most real world applications would require finer actions. These could be more discrete actions such as turning at smaller angles or continuous control. A great amount of progress has been made in the realm of continuous action spaces in areas such as simulated robot control in MuJoCo [Lillicrap et al., 2015]. Allowing movement along the Z axis with altitude changing actions is another important control addition to UAVs specifically. These finer and additional actions would allow the UAV to excel in more cluttered or dynamic environments.

C  Feeding in Depth Images

With the availability of 3D and depth detecting cameras such as RealSense by Intel or the Kinect 3D camera by Microsoft depth images can feasibly be fed into the network as input. Many paper’s have researched using depth images as input to a control module such as the paper mentioned above that used AirSim [Kersandt, 2018], which had good results from just feeding in a depth image at each timestep. Researchers in the field of DRL have also looked into using ground truth depth images for auxiliary rewards instead of input. One paper found the result of using the depth as part of an auxiliary reward in an environment of sparse extrinsic reward to be more fruitful than as additional input to an RGB image [Mirowski et al., 2016].

VII  Conclusion

In this paper, A3C was used to train a UAV agent in AirSim to both fly and go towards a goal while avoiding obstacles in the form of walls in a realistically lit and textured room. The agent had access only to an RGB image from a front camera on its body at each timestep to process partially observable information about the room. Avoidance in an end-to-end manner with a partially observable RGB view is a non-trivial problem. This paper contributes the findings that A3C can provide a good base to doing this.

VIII  Contributions

Here are the contributions made by each of the three team members to the paper and experiments. Max Brenner researched and coded the algorithm along with the rest of the code base. Keaton Grimmett was chief designer of the environments made in Unreal Engine. Both Max Brenner and Keaton Grimmett contributed to writing the paper and analyzing the results of training/testing. Basanta Adhikari helped with understanding Unreal Engine.
References Cited


