Investigating Deep Reinforcement Learning for Autonomous Drone Navigation with Continuous Controls

Sean Kirmani
Department of Computer Science
The University of Texas at Austin
kirmani@utexas.edu

Neil Patil
Department of Computer Science
The University of Texas at Austin
neilpatil@utexas.edu

***Abstract*— As deep learning methods have become more popular, data-driven methods for autonomous drone navigation have shown to be a promising substitute for geometric mesh driven planners. Prior successes have shown successful application of convolutional neural networks in order to map images to discrete high-level behaviors in a supervised manner. We analyze the complexity of attempting to learn a collision-avoidant navigation planner using hybrid model-free and model-based deep reinforcement learning and with continuous controls. Ultimately, we are unable to produce a fully successful planner, so in this work we analyze where this method partially succeeds, and enumerate ideas for potential iteration on this problem.**

I. Introduction

We begin our investigation with the intent to write a planner for a point-to-point obstacle planner for a drone. In free space, this is trivial as the optimal trajectory is a line. However, in the real-world a drone must recognize and avoid obstacles within its trajectory.

The "closed loop" approach to obstacle avoidance is to perform local scene reconstruction and then plan a trajectory around the reconstructed mesh. This approach relies on the scene reconstruction being complete and accurate. However, reconstructing a manifold mesh is both computationally and spatially expensive, and the drone must adapt as it travels further and iteratively constructs the mesh. Other methods simply follow a global plan, using energy functions to repel the drone from obstacle points, usually retrieved as depth information from a stereo or infrared sensor.

Deep learning has arisen as a promising tool to understand high-level representations from lower-level features. Supervised methods have proven rather successful for autonomous aerial vehicles. In the context of obstacle avoidance, there has been success in collecting and learning from data on drone crashes both in simulation and in the real-world [1], [2]. These works focus on detecting the probability of collision and taking actions to minimize this probability.

Most work on obstacle avoidance use discrete actions spaces of some sort, with the Left-Right-Straight (LRS) controller being a popular variant for drones. For ground vehicles, discretizing the action space into a set of linear velocities and angular velocities tends to be a convenient way to get more granularity. In our work, we focus on attempting to learn a continuous control policy.

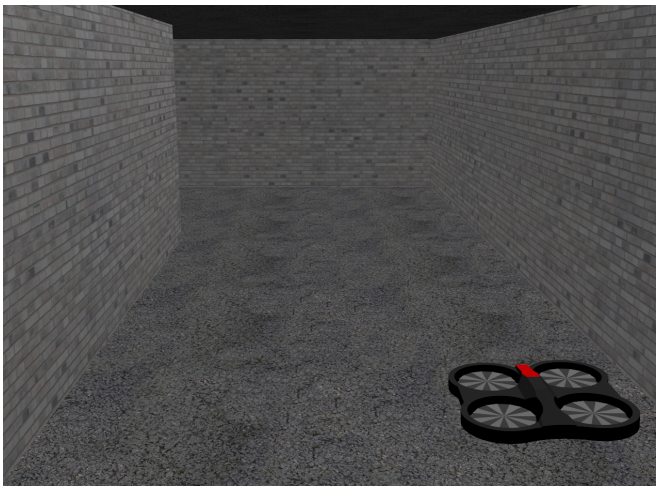


Fig. 1: The drone executing a flight policy a simulated hallway environment.

This is intrinsically difficult to regress towards, especially when considering sequential actions. One of the most classic methods for learning continuous control policies are Deep Deterministic Policy Gradients (DDPG), an actor-critic strategy for learning continuous controls introduced by Lillicrap [3]. We use DDPG to train a continuous control policy for an autonomous drone. This is a model-free approach to the task, which does not explicitly attempt to model the transition dynamics of the world.

In order to improve performance, we follow up on this approach by implementing a model that extends one proposed by Kahn et all. [5], which introduces the idea of a "Generalized Computation Graph", combining model-based and model-free approaches by modeling the probability of collision within a fixed time horizon. Depending on the setting of the time horizon, this creates an implicit model of the world.

While learning end-to-end policies from RGB images to controls is quite popular, we opt to use disparity (depth) maps as our input state space, as they constitute arguably more salient features when planning navigation tasks. We argue that depth-based images are more sample-efficient than their RGB counterparts, and that an RGB to depth network can substitute for directly learning from RGB images.

Taken together, we attempt to train a drone to avoid obstacles using an input of disparity maps and an output of a continuous control policy. We introduce a hybrid model-based and model-free actor-critic algorithm, AC-GCG for learning deterministic policies. While we do not learn a fully successful policy for this task, we show that this implementation does result in improvements to modeling and responding to the environment. Accordingly, we analyze where the task succeeds and fails in order to better understand the task's complexity.

II. Related Work

Previous approaches on using deep learning for obstacle avoidance with drones have mostly focused on learning a policy in a supervised manner.

Gandhi et. al [2] attempt to close the gap between simulation and the real world by collecting a large dataset of crashes on a real drone. This negative flying data is combined with a set of positive examples and used to train a supervised classifier mapping RGB input spaces to discrete action outputs. In their experiment, they collected approximately eleven thousand crashes on a real drone. They used this data to train a discrete left-right-straight controller, and successfully showed they were able to fly without crashing. Notably, they do not have to deal with transfer, as their dataset is collected on a real drone.

CAD2RL [1] aims to teach obstacle avoidance by combining a large set of randomly generated environments with reinforcement learning. This is beneficial as collecting data on a real drone is time-consuming and difficult. This method uses single images from an RGB camera to train a deep convolutional neural network that outputs discrete velocity commands. In this work, the image space is discretized into 41 x 41 bins, where each bin corresponds to an action in that direction. By highly randomizing the training set, the authors are able to train a policy that generalizes well to the real world, ultimately evaluating the policy on a real quadrotor flying through a set of hallways. Even though their simulation environment contained fairly unrealistic worlds (such as grass on walls and on tables) transfer to a real drone works well. This interesting finding demonstrates that the simulation environments do not necessarily need to be accurate, but simply diverse, meaning that dynamics randomization can be a useful solution for manipulation and navigation task transfer [6].

Unlike prior works with discrete action spaces, we often desire to learn policies outputting continuous controls, as velocities and torques are continuous values in practice. While quantizing values is convenient to regress on, it is not representative. Zhang et al. [7] propose a model-predictive control based guided policy search method (MPCGPS). In this approach, the training phase alternates between running Model Predictive Control (given the full state of the environment) to attempt the task and collect a dataset of trajectories, and using this data to train a neural network policy based on only the vehicle's onboard sensors. The full state of the environment is removed during test time. The authors are able to successfully train a policy that takes a set of readings from laser rangefinders and outputs a set of 4 rotor velocities. This is an immensely complex task, and proves to be one of the biggest successes in learning continuous controls with unknown dynamics.

Outside of drones, Kahn and Levine [5] have investigated collision avoidance with RC cars. In this paper, they develop a stochastic computation graph to represent both model-free and model-based reinforcement learning, as well as a way to let the agent self-supervise allowing for continuous learning. With only a few hours of self-supervised training, they are able to learn a successful policy outputting a one-dimensional continuous steering angle. They show that their stochastic computation graph outperforms other N-step Q-learning methods for this navigation task.

While many of the previous papers use RGB or grayscale images, we opt to use depth images similar to in Xie et. al [8]. Like their work, we are interested in learning obstacle avoidance using monocular vision. We can leverage the work done by Eigen and Fergus in neural networks that produce depth maps from RGB images [9].

Finally, Lillicrap's deep deterministic policy gradients (DDPG) [3] algorithm has become a popular actor-critic method for learning continuous control tasks. In practice, DDPG performs well with solving navigation tasks, while A3C performs well with manipulation and control tasks.

III. Model-Free Navigation with Deep Deterministic Policy Gradients

For simplicity, we began our implementation using a model-free scenario. We begin by implementing DDPG, a fairly classic actor-critic algorithm. By leveraging the actor-critic method, our algorithm simultaneously learns how to act as well as how to evaluate the actor.

DDPG is an off-policy training algorithm, so we train in an episodic setting. At every episode, we run our drone for some maximum number of steps T, and at every step t add (s_t, a_t, r_t, s_{t+1}) to our experience replay buffer. Each state s_t consisted of some previous number of depth images, each action a_t corresponded to our steering control, and r_t was our reward. Using a replay buffer and sampling it uniformly helps avoid only sampling recent experiences, so that the policy does not converge to some local minima solution prematurely.

Designing a reward function R turned out to be nontrivial. We ended up experimenting with many reward shaping strategies, but ultimately opted for a sparse reward. We describe our reward methodology and results in the following section.

A. Reward Function

Tested variations in the reward function generally fell into two strategies. The first was to encode some metric of distance to a goal position into the reward function itself. The reasoning behind this was that the drone would be incentivized to travel towards a goal location, forcing it to learn to avoid obstacles along the way.

However, balancing this reward shaping of distance to goal with the implicit reward of avoiding obstacles was difficult. As a result, the second class of reward functions did not explicitly encode a goal position, and instead simply terminated the episode when the drone collided into an object. Reward would accrue in such a way that a longer episode would lead to a higher total reward. In this way, the drone would be directly incentivized to avoid obstacles when flying in order to maximize its "time alive".

Reward was computed at each time "step" within the simulation, with a step size of 0.25 seconds. The total reward of the episode would be the sum of the rewards at all time steps.

We experimented with several variations of reward function beyond these two general classes. These were:

1) Distance-based: Given a goal position g, a start position s, and a current drone position x_i, we tested two variations:

$\frac{x_i - s}{g - s}$. This effectively represented the "percentage complete" of the distance to goal. In order to ensure the drone would not move away from the goal and still acquire reward, we would terminate the episode if the drone's distance from the goal increased continuously over several steps.

$(g - x_i)^2$. This was the squared distance to the goal.

We ultimately decided to consider the probability of collision independent of this higher-level navigation task, and ended up removing the concept of a "goal" from our task.

2) Direct Shaping from Depth Map: We experimented briefly with encoding distance to closest object into the reward function by computing this directly from the image depth map. This turned out to be challenging as the depth map contained many regions with no depth information (such as when objects were outside of the operating range of the simulated depth image).

3) Velocity: To encourage the drone to travel as far as possible at each episode, we experimented with a reward for our current velocity at each time-step. This definition is convenient as the sum of velocities at each time-step is correlated with total position travelled.

4) Sparse-Collision Collision: Finally, the drone can simply be given a reward of $+1$ if it has not collided, and of 0 when when it is in collision. As our task is to fly without collision, maximizing the probability of not colliding is directly related to the target task.

Ultimately, we opted to treat the reward as the collision probability, with the resulting goal being to minimize the probability of collision. Importantly, however, this means our previous model-free definition is unable to learn this task beyond just one-step before collision. As a result, we implement an algorithm to predict "into the future" which actions will lead to collision by learning a finite-horizon model of the future.

IV. Overview of Hybrid Model-Based and Model-Free Generalized Computation Graphs

To predict collision probabilities, instead of predicting rewards r_i, we desire to predict "outputs" y_i which do not necessarily correspond with with rewards. Given an input state s_t and actions $A_t^H = (a_t, ..., a_{t_H-1})$ over a horizon H, we produce outputs $\hat{Y}_t^H = (\hat{y}_t, ...\hat{y}_{t+H-1})$ and an additional terminal output $\hat{b}_{t+H}$. This output is mapped as our critic function over some parameterization θ, with the loss to optimize represented as $\epsilon_t(\theta)$.

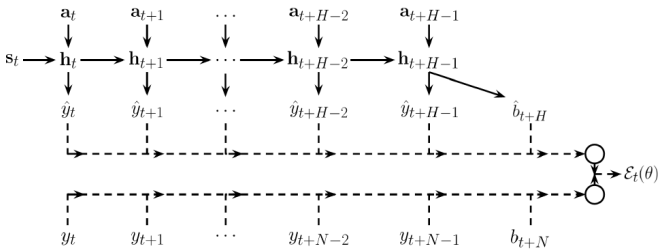


Fig. 2: A stochastic computation graph for model-free, model-based, and hybrid-based reinforcement learning algorithms.

The stochastic computation graph proposed by Kahn [5] subsumes both model-free and model-based reinforcement learning. If $H = T$, where T is the episode length, the model is entirely model-based. If $H = 1$, the model represents the model-free approach. A value of H in between means the model represents some fixed horizon in the future.

In the case of the drone, we pick a horizon of $H = 16$, which corresponds to predicting if the drone will collide with the wall over the next 4 seconds (as our step size corresponds to 0.25 seconds per step). We find that choosing a longer horizon $(H = 16)$ outperforms the short horizon $(H = 1)$ in practice.

As discussed above, we chose a sparse collision-based output, y_i, corresponding to either +1 (if no collision) or 0 (if collision). The loss $\varepsilon_t(\theta)$ can be represented in in two ways: as a mean-squared error regression loss or as a classification loss. We find that treating the collision as a classification problem trains better than as a regression problem when training the model. If our model outputs are collision probabilities, the loss function becomes the cross entropy loss:

$$\varepsilon_t(\theta) = -\big[\sum_{h=0}^{H-1} y_{t+h} \log(\hat{y}_{t+h}) + (1 - y_{t+h}) \log(1 - \hat{y}_{t+h}) + \hat{b}_{t+H} \log(\hat{b}_{t+h}) + (1 - \hat{b}_{t+H} \log(1 - \hat{b}_{t+H}) \big] \tag{1}$$

Finally, we define our terminal output as

$$b_{t+H} = \max_{A^H} \frac{1}{H} \sum_{h=0}^{H} \hat{y}_{t+h} \tag{2}$$

which represents the maximum average probability of flying without collision that the robot can achieve at time $t + H$.

Data: computation graph for critic $Q(s, A^H|\theta^Q)$, actor $\mu(s|\theta^m u)$ with weights θ^Q and θ^μ, error function for computation graph $\varepsilon_t(\theta)$, and policy evaluation function $J(s_t, A_t^H)$

begin
- initialize dataset $D \leftarrow \emptyset$
- **for** $e \in$ *number of epochs* E **do**
  - initialize trajectory buffer $B \leftarrow \emptyset$
  - **for** $t = 1$ **to** T **do**
    - get current state s_t
    - select action $a_t \leftarrow \mu(s|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise.
    - execute first action a_t
    - receive labels y_i
    - add (s_t, a_t, y_t, s_{t+1}) to trajectory buffer B
    - **if** s_t *is terminal* **then**
      - break
    - **end**
  - **end**
  - **for** $(s_t, a_t, y_t, s_{t+1} \in B$ **do**
    - initialize $Y_t^H \leftarrow \emptyset$
    - initialize $A_t^H \leftarrow \emptyset$
    - **for** $h = 1$ **to** H **do**
      - add y_{t+h} to Y_t^H
      - add a_{t+h} to A_t^H
    - **end**
    - add $(s_t, A_t^H, Y_t^H, s_{t+1}$ to dataset D
  - **end**
  - **for** *some number of optimization steps* **do**
    - sample a random minibatch of N transitions $(s_t, A_t^H, Y_t^H, s_{t+1})$ from D
    - update the the critic by minimizing the loss: $\theta^Q \leftarrow \theta^Q + \arg\min_{\theta^Q} \varepsilon_{\theta^Q}(\theta)$ using the minibatch
    - update the actor policy using the sampled policy gradient:

$$\nabla_{\mu^\theta} J \approx \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_{\mu^\theta} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\right]$$

    - Update the target networks: $\theta^{Q\prime} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q\prime}$ and $\theta^{\mu\prime} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu\prime}$
  - **end**
- **end**

end

Algorithm 1: Reinforcement Learning with Actor-Critic Generalized Computation Graph (AC-GCG)

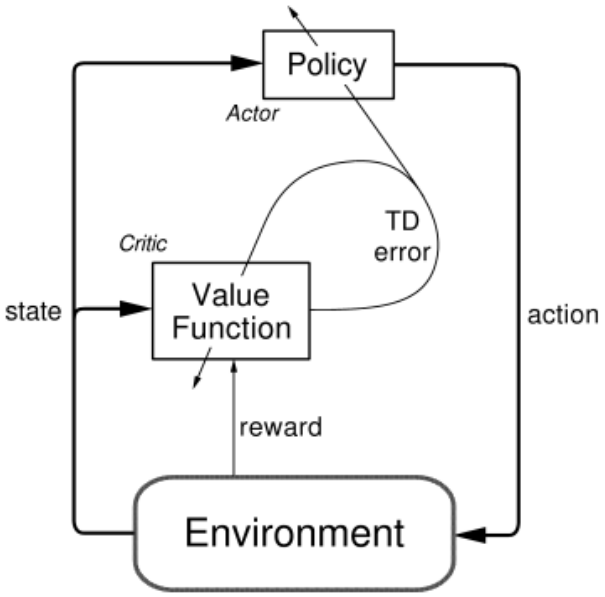


Fig. 3: The general actor-critic framework.

V. Actor-Critic Generalized Computation Graphs

We replace our critic network in our DDPG implementation with the model described above. This is unlike the work done in [5], which did not use an actor network and simply chooses the best action out of a set of K random actions. While their work is most analogous to N-Step Q-Learning, our work is still an actor-critic architecture, simultaneously learning how to act as well as how to evaluate the drone's performance.

Our algorithm chooses actions $a_t \in \mathbb{R}^N$, selecting N actions to perform over horizon H. It fundamentally tries to optimize actions that maximize b_{t+H}, which is the maximum probability of not colliding. Our parameterized actor function is represented as $\mu(s|\theta^\mu)$.

We can interpret Y_t^H and its corresponding loss as how we learn our model, and b_{t+H} as the actual value we want our actor to optimize. Thus, we can treat b_{t+H} as the target which our actor network is attempting to learn. As $t \to \infty$, b_{t+H} will become stationary, as it will converge to a learned model of the world. Initially, the algorithm is taking the best actions it can under its belief of the world. We can apply the chain rule and update the action with respect to error in the belief. Importantly, we update our model before we update the belief of the best actions we can do. The actor is updated by applying the chain rule to the expected return from the start belief distribution J with respect to the actor parameters:

$$\begin{aligned}\nabla_{\mu^\theta} J &\approx \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_{\mu^\theta} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\right] \\ &= \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\mu^\theta} \mu(s|\theta^\mu)|_{s=s_t}\right]\end{aligned} \tag{3}$$

where $Q(a, s) = b_{t+H}$ in our optimization formulation.

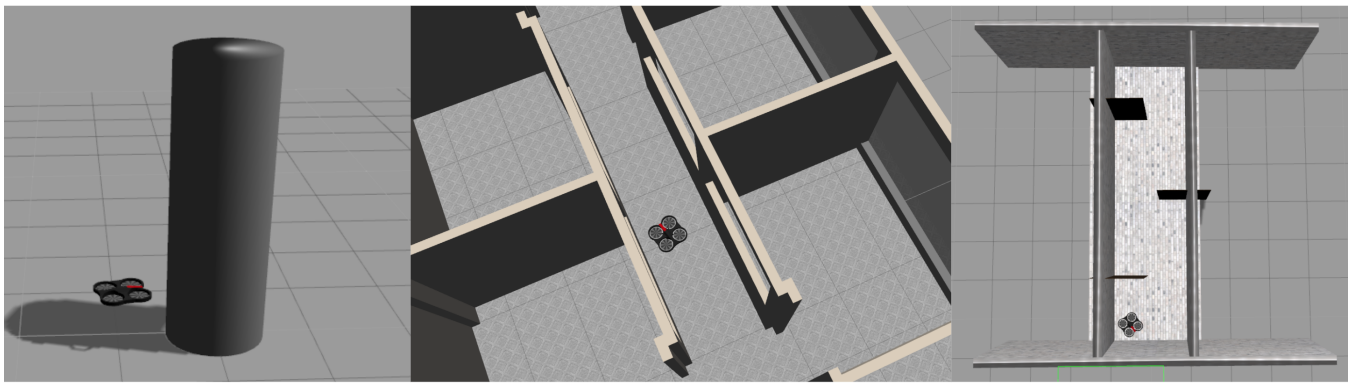


Fig. 4: A subset of the environments used in training. From left to right: Simple Cylinder, House, Constrained Hallway.

Taken together, the main contribution of this paper is the introduction of a general model-based and model-free actor-critic algorithm for learning deterministic policies. This algorithm combines ideas from both Deep Deterministic Policy Gradients and Generalized Computation Graphs. Since we formulate the computation graph in an actor-critic framework, we refer to our algorithm as an actor-critic generalized computation graph (AC-GCG).

VI. Implementation Structure

A. Environment

A simulated environment based within ROS Indigo and Gazebo 2 was used to train and evaluate the drone. The drone being simulated was heavily modeled after the Parrot AR Drone 2.0, a common drone used in research. We made extensive use of the open-source tum_ardrone and tum_simulator ROS packages in order to simulate the drone. All algorithms were implemented with a combination of rospy and Tensorflow.

We manually designed a simple set of 3D Gazebo worlds in which the drone would fly. These were:

- "Single Cylinder": a simple, open-ended world only containing a single cylinder.
- "House": a large "house" structure with many hallways and separate rooms.
- "Structured Hallway": a constrained hallway structure with barriers along its length, forcing the drone to travel in a zig-zag pattern to keep moving.
- "Enclosed Hallway": a more open-ended hallway forming a ring shape.

Figure 4 shows a sample of the training environments used in testing. A video of the drone training within this environment can be viewed at `https://youtu.be/SpHzPUGTd3c`.

B. Policy State Inputs

As our goal was to train a policy mapping directly from sensor data to actions, our state input consisted of direct visual and pose sensor data. We decided to use depth data instead of raw RGB values, as we reasoned depth would contain more salient information relating to predicting collisions.

Sensor data was collected from ROS and resized to a 64x32 image. This was concatenated with the sensor data from the previous three steps and fed to the drone. At first, this sensor data was also combined with a length three vector describing the drone's position in free space. However, after switching to a reward function not dependent on location, pose was removed from the input.

C. Collision Behavior

As mentioned above, our final reward function is paired with resetting the episode each time the drone collides with an object. In order to do this, we make use of a bump collision sensor provided by Gazebo.

At first, our episode restarts were accompanied by resetting the drone to its initial position and orientation at the beginning of the episode. As a result, the drone would start from the same place each time.

However, we found that allowing the drone to continue from its current location allowed for faster training and a greater variety of collision experiences, and prevented spatial biasing of experiences. To facilitate this, we would have the drone back up for several seconds after colliding with an object. The episode would then be restarted from its current position. Often, this would result in the drone repeatedly colliding and backing up from the wall several times, adding collision-related information without requiring the drone to fly a full path to the barrier again.

D. Action Space

As we intended to output continuous controls, the action output consisted of a vector of scalar values. Initially, this consisted of 4 values x, y, z, r corresponding to a velocity command along the x-axis, y-axis, z-axis, and rotation along the z-axis respectively.

However, during experimentation, we progressively simplified the action space:

1) After observing there were few actions requiring y that could not be done with some combination of x and r, this parameter was removed.
2) After observing that height of the drone contributed little to its ability to avoid obstacles in the particular

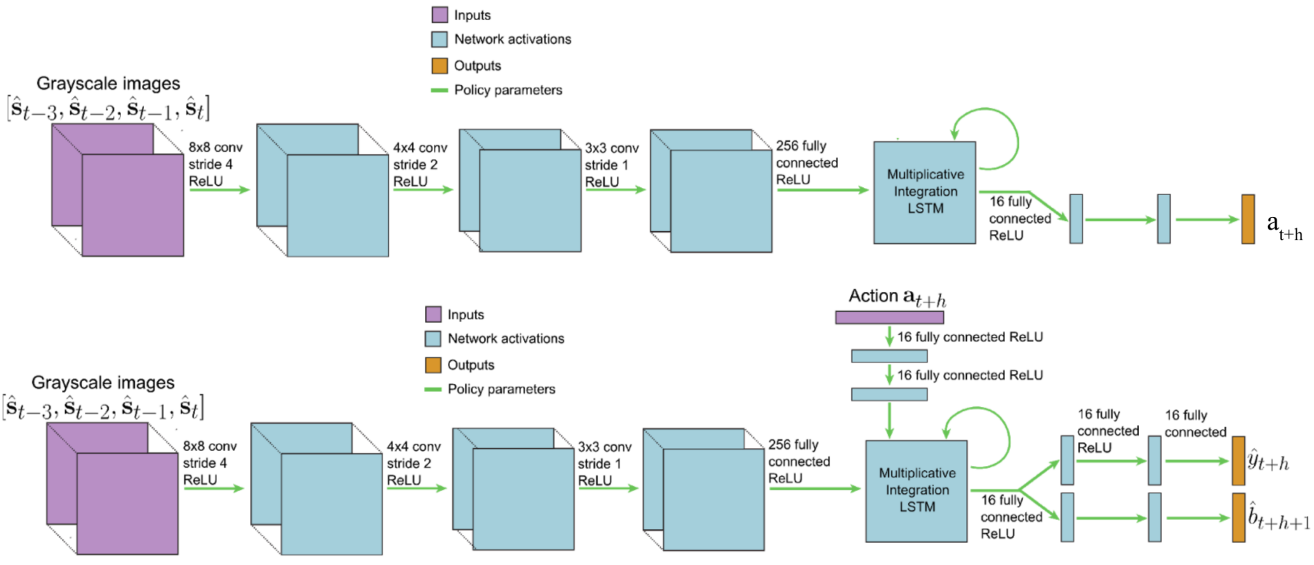


Fig. 5: The structure of the actor and critic networks.

worlds we tested, the drone was configured to start at a fixed height and z was set to zero.

3) We initially tried to encourage positive values of x (asking the drone to move forward) with some reward shaping. However, this behavior was eventually forced by fixing the value of x.

As a result, our final action space was a single scalar value r, corresponding to a rotational velocity output.

E. Network Architecture

Within the DDPG implementation, the final architecture consisted of a number of convolutional layers combined with ReLU activations. These were then combined with two fully connected layers to output the final action scalar value.

When switching to the hybrid-model approach, a multiplicative integration LSTM was added. Additionally, the action choices were fed into a series of convolutional layers that were then fed into the LSTM, and were followed by two fully connected layers to output the values of y and b.

Figure 5 contains a detailed diagram of the network architecture.

VII. Results

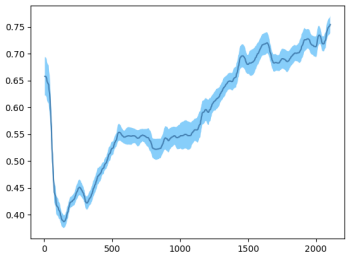
Qualitatively, the drone failed to learn a fully functional obstacle avoidance policy with both algorithms. However, the drone showed promising behavior that indicated a partial success of the models. Additionally, the Generalized Computation Graph method resulted in better performance than the naive DDPG implementation, as expected.

Under the DDPG implementation, the drone would occasionally learn a simple policy that allowed it to avoid a wall or barrier in early iterations. However, further training would often lead to catastrophic failure, resulting in a policy that would lead the drone to crash directly into the wall.

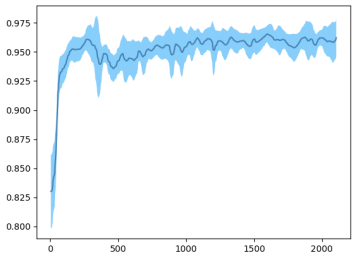
Switching from DDPG to the Generalized Computation Graph resulted in an improvement in learning. While the drone still failed to learn a policy that reliably avoided obstacles, in some cases the drone learned promising simple policies. For example, after some training the drone learned to travel in a tight circle. By doing so, the drone was able to avoid walls completely and maximize its reward function. A video of this trained policy can be viewed at `https://youtu.be/LHbQWyGzTqs`. Figure 6 shows graphs of the horizon success probability, model accuracy, and reward during training in this scenario. Notably, the model accuracy reaches convergence, and the probability of avoiding a collision does increase during training, as expected.

The initial drop in the horizon success probability shown in Figure 6 also makes sense. This is explained by the fact that at the beginning of training, the drone has not developed a good model of the world. As a result, it overestimates its probability of avoiding walls - hence the high horizon success probability. As the model improves, however, this probability quickly falls. Eventually, as the actor improves further, the probability slowly climbs again. A simplified analogy can be made to the "Dunning Kruger Effect" - the robot initially does not know what it does not know, so it overestimates its performance, but as it develops a better understanding, this is quickly corrected.

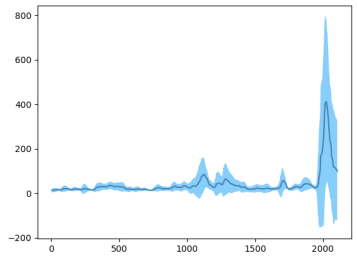
Occasionally, simulation error would lead to the drone "escaping" its hallway during training. Figure 7 shows the horizon success probability, model accuracy, and reward in this scenario. Interestingly, this unexpected case demonstrates some desirable aspects of the algorithm, serving as a sanity check. As the drone leaves the constrained environment, it becomes much easier for it to avoid crashing, as there are fewer obstacles to collide with (as the drone is now in an environment close to free space). This is reflected by the fact



Horizon Success

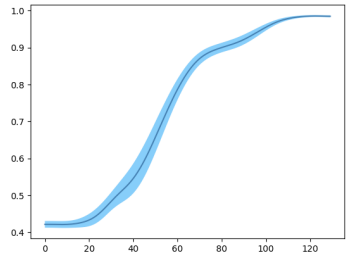


Model Accuracy

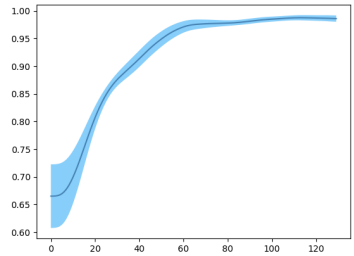


Reward

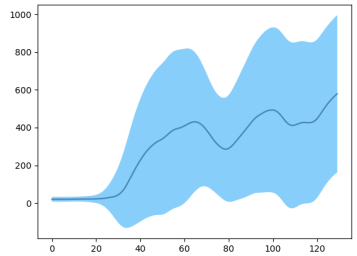
Fig. 6: The learning curves and uncertainty of the final policy trained over 14 hours when $H = 16$ inside the "Enclosed Hallway" world. Performance over the horizon gets better as more data is collected to improve the model. From left to right, all plotted as a function of number of iterations: the probability of not colliding over the horizon (actor loss), the accuracy of the model predictions (critic loss), and the reward (performance on actual task).



Horizon Success



Model Accuracy



Reward

Fig. 7: The learning curves and uncertainty of the final policy trained over 14 hours when $H = 16$ in the "Enclosed Hallway" world. In this case, the drone inadvertently escapes the hallway early on in the training process. outside the building. This functions as a quick sanity check - we prove that in a scenario where with very few obstacles to collide with (i.e. close to free-space), our model learned to avoid collisions and avoid the building all together. Notably, the probability of success over the horizon (left) quickly approaches 1. From left to right: the probability of not colliding over the horizon (actor loss), the accuracy of the model predictions (critic loss), and the reward (performance on actual task).

that the horizon success probability, accuracy, and reward functions rapidly increase.

These results lead us to believe that with principled tweaks to the algorithm, the final policy can be significantly improved.

VIII. Discussion

In this section, we analyze the various causes that could explain the performance of our algorithm. What follows are factors we believe contributed to the failure of the algorithm to learn a general policy:

A. Sparsity of Information for Assigning Correspondence

The primary goal of the training was to learn a policy to avoid obstacles. However, information corresponding to whether this goal was successful only appeared at the very end of a trajectory - where the drone collided into an obstacle. This may have been too sparse of a reward, especially with regards to the DDPG implementation. For example, if the drone collided with a wall, the correct action may have been to start turning five seconds prior, but this action may have been too difficult to associate with the end result. Alternatively, if the drone did succeed in avoiding the barrier, it may be unclear if this was due to slight change in action at the beginning of the trajectory, or an active change before collision. More generally, this problem is known as the correspondence problem.

The introduction of the the hybrid model-based model-free approach potentially helped mitigate this problem by computing probabilities over a fixed time horizon. Still, however, the reward's sparsity may have made modeling this sample-inefficient. Moreover, choosing an appropriate time horizon to use within this model is difficult - a small horizon prevents the drone from being able to correct itself in time, while a large one leads to an exponential increase in probability space.

B. Errors in Simulation

Occasionally, simulation error would result in catastrophic failure during training. For example, when resetting and backing up the drone, the drone would often be able to "tunnel" through the walls of its enclosed space, exposing it to an open environment without walls. Since we did not reset the drone's position after each episode, this could influence

the policy. Additionally, errors in modeling of the drone's behavior after colliding into the wall could result in flipping which could also catastrophically affect the results.

C. Poor Actor Performance

As discussed above, the estimates of collision horizon probability did generally improve with training. This leads us to believe further improvements may lie in the improvement of the actor network. Currently, we update our model of the world and our expectation of the best action we can do given that previous model simultaneously, and then we actually update our actor network. It may be better in practice to actually update our model of the world, then update our expectation of the best we can do given our ınew model, and finally update our actor with respect to our updated expectation. It's unclear at the time of writing whether evaluating our best action (i.e. our Q-Value), should be based on the last model which we acted against or the newest model we're acting on.

IX. Conclusion

In this work, we attempt to learn a policy outputting continuous controls in a simulation environment. We test both DDPG and introduce an actor-critic hybrid model-based model-free approach that extends the Generalized Computation Graph presented in previous work. We find that this hybrid approach does result in an improvement over a basic DDPG algorithm. While we ultimately do not learn a successful policy, we demonstrate that the model does exhibit promising behavior which leads us to believe that this method is tractable provided some additional improvements and tweaking.

For example, one of the policies we observed was that the drone would choose to turn in a loop. This can be easily corrected by designing a more shaped environment structure or action space to prevent this from happening.

Additionally, we began our experiments by encoding a goal position into the reward, but this was ultimately removed. We would like to reintroduce this by integrating our model with a high level navigation task. In this system, our trained policy would determine the optimal actions to avoid obstacles, which could be weighted with the action that would move the drone closer towards the goal.

We could also factor uncertainty into our planning. Khan et al. has also considered learning on a real drone and RC car where the velocity of the vehicle is proportional to how certain the agent's future actions will be successful [10]. Thus we don't set a fixed velocity, and actually leverage the intrinsic uncertainty in our actions. This is also interesting from a human-robot interaction perspective because it actual communicates our uncertainty through our actions.

The final goal is to transfer this policy from simulation to a real-world drone, as with prior works [6], [1], [11], [12]. This may be tractable provided the drone is trained on a large and varied set of simulation environments.

References

[1] F. Sadeghi, S. Levine, "CAD2RL: Real Single-Image Flight without a Single Real Image", 2017.

[2] D. Gandhi, L. Pinto, A. Gupta "Learning to Fly by Crashing", 2017.

[3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, D. Wierstra, "Continuous control with deep reinforcement learning", 2015.

[4] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, W. Zaremba, "Hindsight Experience Replay", 2017.

[5] G. Kahn, A. Villaflor, B. Ding, P. Abbeel, S. Levine, "Self-supervised Deep Reinforcement Learning with Generalized Computation Graphs for Robot Navigation", 2017.

[6] X. B. Peng, M. Andrychowicz, W. Zaremba, P. Abbeel, "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization", 2017.

[7] T. Zhang, G. Kahn, S. Levine, P. Abbeel, "Learning Deep Control Policies for Autonomous Aerial Vehicles with MPC-Guided Policy Search", 2016.

[8] L. Xie, S. Wang, A. Markham, N. Trigoni, "Towards Monocular Vision based Obstacle Avoidance through Deep Reinforcement Learning", 2017.

[9] David Eigen, Rob Fergus, "Predicting Depth, Surface Normals and Semantic Labels with a Common Multi-Scale Convolutional Architecture", 2014.

[10] G. Kahn, A. Villaflor, V. Pong, P. Abbeel, S. Levine, "Uncertainty-Aware Reinforcement Learning for Collision Avoidance", 2017.

[11] K. Bousmalis, A. Irpan, P. Wohlhart, Y. Bai, M. Kelcey, M. Kalakrishnan, L. Downs, J. Ibarz, P. Pastor, K. Konolige, S. Levine, V. Vanhoucke, "Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping", 2017.

[12] A. A. Rusu, M. Vecerik, T. Rothrl, N. Heess, R. Pascanu, R. Hadsell, "Sim-to-Real Robot Learning from Pixels with Progressive Nets", 2016.