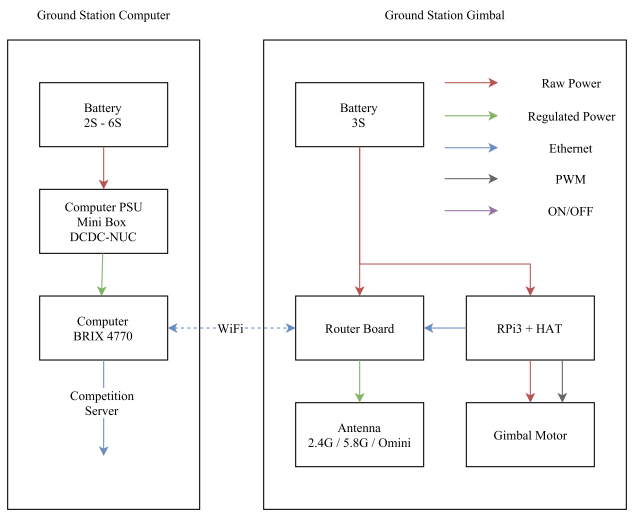

Fig. 5. Martlet-1's ground station system architecture.

teams have attempted in the past, a reinforcement learning approach has been investigated to address the uncertainty in the UAV's and moving obstacles' dynamics models.

1) Deep Q-Networks: Google DeepMind [2] managed to bring deep reinforcement learning to human-level control when playing a variety of arcade games. Despite training their agent on raw images and using deep convolutional neural networks (CNN), their methods can still be applied to problems such as path planning and control for a UAV.

The basic idea of Google DeepMind [2] is to use a Deep Q-Network (DQN) to approximate the utility of different actions. For this to work, they simply assume that there exists a function $Q^*(s,a)$ that accurately approximates the Q-value of any state-action pair (s,a). As with any reinforcement learning technique, such a function would need to address the credit assignment problem [3]. The credit assignment problem is the problem of determining which of the preceding actions was responsible for getting a reward and to what extent. This is especially important in the case of an autonomous path planner since reaching an unavoidable crash state could have usually only been prevented several actions prior. This problem can be mitigated simply by discounting future rewards with a constant factor γ. This discount allows us to represent the approximate function $Q^*(s,a)$ as in Equation 4,

$$Q^*(s,a) \leftarrow \max_{\pi} \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} | s_t = s, a_t = a, \pi\right] \quad (4)$$

which is the maximum sum of rewards r discounted by γ at every time step t given a policy π.

To approximate Equation 4, [2] uses a deep CNN. Intuitively, the inputs to the neural network would be a state-action pair (s,a) and the output would be the corresponding Q-value $Q^*(s,a)$. However, since we're dealing with a finite and discrete action space A, we can optimize the network such that it simply takes the state s as input and outputs the Q-value $Q^*(s,a)$ for every possible action $a \in A$ at once. The agent can then simply select to play the action with the maximal Q-value.

Such a network needs a slightly different approach to back-propagation, however, since we cannot infer the actual reward for actions the agent did not take. Instead, we only correct the network's estimate of the action the agent does take. For all other actions, we simply feed the network the same value as it had output such that their error is 0. As for the action taken, we update the network similarly to in Q-learning, but noting that the learning rate α and the update is now addressed directly by the back-propagation algorithm. This yields Equation 5.

$$Q'^*(s,a) \leftarrow \begin{cases} r_t + \gamma \max_a Q^*(s',a) & a \text{ is selected} \\ Q^*(s,a) & \text{otherwise} \end{cases} \quad (5)$$

We can finally back-propagate the error using the mean square error L as in Equation 6 with a learning rate α. This can be shown to yield an equivalent update rule as in Q-learning.

$$L = \frac{1}{2}[Q'^*(s,a) - Q^*(s,a)]^2 \quad (6)$$

After sufficient iterations, the result is a DQN agent that can accurately approximate $Q^*(s,a)$ and that was trained using a model-free and generalizable method.

2) Assumptions: For preliminary results, to simplify the problem, the UAV was constricted to a two-dimensional plane by fixing its altitude, roll and pitch. It was also assumed that the UAV will be maintaining a fixed forward velocity. Both of these assumptions allow for a greatly reduced action space. This leaves the action as simply a control of the UAV's yaw rate. However, DQNs require a discretized action space, and this needs to be done with care.

3) Dubins Path: In the field of robotics, a common approach to solving this type of path planning problem is trying to solve for Dubins path [4]. Dubins path refers to the shortest curve that a vehicle can take to travel between two points in the two-dimensional Euclidean plane when the vehicle is constrained as follows:

1) The vehicle has a maximum turning rate.
2) The initial and terminal headings to the path are defined.
3) The vehicle can only travel forward.

A key point of this method is that Dubins proved that such a path will only consist of a sequence segments of either maximum curvature or straight lines [4]. In other words, a path can be represented by a simple series of either a left turn (L), a straight line (S) or a right turn (R). Figure 6 shows example optimal paths from points A to B.

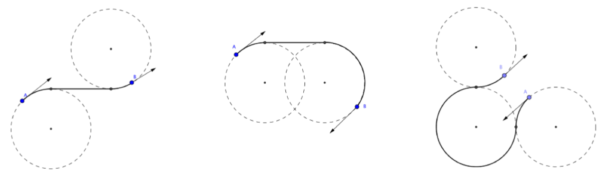

Fig. 6. Sample Dubins paths from left to right: RSL, RSR, and LRL.

In two-dimensions, the kinematics of such a vehicle can be defined as in Equation 7, where v is the velocity of the vehicle, which, for simplicity, is usually kept constant, and u is the rate of the heading θ. For a Dubins path, u need only be limited to one of $\{-\phi, 0, \phi\}$ radians per second, where ϕ is the maximum turning rate of the vehicle. An optimal controller that yields a Dubins path would then simply be controlling u as a function of the vehicle's current state (x, y, θ) and its target. This approach is often also dubbed Dubins Car.

$$\begin{aligned} \dot{x} &= v\cos\theta \\ \dot{y} &= v\sin\theta \\ \dot{\theta} &= u \end{aligned} \tag{7}$$

4) Simulation: A simulation environment was built using the Director visualization package developed at MIT [5]. A UAV with simple dynamics was modeled along with stationary and moving obstacles. A virtual range sensor was also developed to receive feedback from the environment. The sensor works similarly to light detection and ranging (LIDAR): rays protrude from the plane as seen in Figure 7 and measure the distance to the nearest object on their path. The UAV then uses the data to inform its decisions. The model currently uses 16 rays evenly distributed across a forward-facing 90 degrees FOV and the rays have a 40 meter range.

Obstacles and target set-points were also generated at random during the training process.

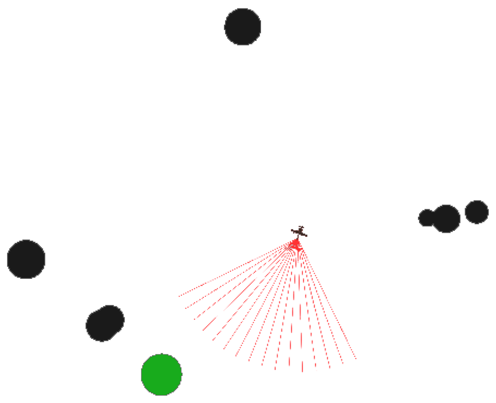

Fig. 7. Visualization of plane's sensor in red, obstacles in black, and target set-point in green.

5) State & Action Space: In order to simplify the problem and reduce training time, the state space has been reduced to a 19-element vector. This vector is comprised of the 16 rays' distances normalized to the range $[0, 1]$ followed by the relative x, y and heading θ to the target set-point.

The action space is inspired by Dubins curves which proved that the optimal path would only consist of a sequence segments of either maximum curvature or straight lines [4]. Hence, the action space A was limited as such with $A = \{-\phi, 0, \phi\}$ radians per second, where ϕ is the maximum turning rate. In this model, we used a ϕ of $\frac{\pi}{2}$ radians per second.

6) Implementation: The DQN was implemented using Google's TensorFlow framework, [6].

Following the DQN neural network model, we set up a neural network with 19 input nodes for the state and 3 output nodes for the Q-values of each action in the action space. Hidden layer nodes were rectified linear unit (ReLU) activated whereas the output layer uses simple linear activation. A list of the hyperparameters used and their values can be found in Table IV. No convolutional layers were necessary.

TABLE IV. DQN HYPERPARAMETERS

Hyperparameter	**Value**
Learning rate (α)	0.01
Discount factor (γ)	0.9
Exploration factor	0.5
Hidden layer 1 node count	11
Hidden layer 2 node count	7

The final result would be a small light-weight neural network that can compute a locally optimal policy for avoiding obstacles and reaching a target set-point. The neural network's small footprint has the added benefit of being able to run in real-time.