8-2018

Deep Reinforcement Learning for Autonomous Search and Rescue

Juan Gonzalo Cárcamo Zuluaga
Grand Valley State University

Deep Reinforcement Learning for Autonomous Search and Rescue

Juan Gonzalo Cárcamo Zuluaga

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements

For the Degree of

Master of Science in Computer Information Systems

School of Computing and Information Systems

August 2018

Abstract

Unmanned Aerial Vehicles (UAVs) are becoming more prevalent every day. In addition, advances in battery life and electronic sensors have enabled the development of diverse UAV applications outside their original military domain. For example, Search and Rescue (SAR) operations can benefit greatly from modern UAVs since even the simplest commercial models are equipped with high-resolution cameras and the ability to stream video to a computer or portable device. As a result, autonomous unmanned systems (ground, aquatic, and aerial) have recently been employed for such typical SAR tasks as terrain mapping, task observation, and early supply delivery. However, these systems were developed before advances such as Google Deepmind's breakthrough with the Deep Q-Network (DQN) technology. Therefore, most of them rely heavily on greedy or potential-based heuristics, without the ability to learn. In this research, we present two possible approximations (Partially Observable Markov Decision Processes) for enhancing the performance of autonomous UAVs in SAR by incorporating newly-developed Reinforcement Learning methods. The project utilizes open-source tools such as Microsoft's state-of-the-art UAV simulator AirSim, and Keras, a machine learning framework that can make use of Google's popular tensor library called TensorFlow. The main approach investigated in this research is the Deep Q-Network.

Contents

List of Figures

1 Introduction

Search and Rescue (SAR) personnel are not strangers to reports of overdue persons. When dealing with these incidents three main tasks must be performed: Investigation, Containment, and hasty Search efforts. Initial actions, those taken during the first 8-12 hours after the start of the operation, usually locate the subject. However when subjects are not located during this early period, search operations can last days, and even weeks[1]. It is easy to see why any effort to improve the efficiency of the initial search effort is crucial for a successful SAR operation.

Fortunately, the increasing development of information and communication technologies in recent decades has brought new tools to the table for Search and Rescue teams. Of special interest is the use of robots: aerial, marine, and land-based, in the aid of SAR teams. In the case of aerial robots, Unmanned Aerial Vehicles (UAV) are currently in use as source of quick birds-eye view, and precise cartography[2].

Artificial Intelligence (AI), understood as intelligent behavior shown by a machine, has been getting a lot of traction in recent years, with algorithms reaching super-human levels of accuracy classifying images, predicting stock prices and even playing games. Within the field of computer science there is an AI branch called Machine Learning. Its main objective is the research and development of methods and algorithms capable of improving their performance of a task based on their previous experience performing it. For example, learning to detect objects in an image by presenting to the algorithm large data sets of images with the corresponding labels of the objects present in them. In order to create autonomous behaviors in robotics, machine learning techniques are used extensively.

Reinforcement Learning (RL) methods create AIs that learn via interaction with their environment. Similar to the behaviorism learning paradigm, RL algorithms try to find the optimal approach to performing a task by executing actions within an environment and receiving positive and negative rewards for the actions taken in that environment. Formally, the RL problem is described as a Markov Decision Process (MDP) which is a tuple of States, Actions, Rewards, and Transition probabilities. The final goal of RL methods is to create a program capable of maximizing the cumulative reward received while moving through the MDP states

until reaching a predefined terminal state. The resulting strategy is referred to as the policy. When RL is used in combination with deep neural networks (Deep Learning) it is called deep reinforcement learning.

Taking advantage of deep reinforcement learning methods, it is believed that a fully autonomous UAV can be trained to perform Search operations using these techniques. By no means would this be a replacement for tried and true search conducted by UAV operators, however, it is believed this could be of help in the coverage of complex areas where the need for manual navigational control distracts the operator from carefully observing the scene.

Unmanned, autonomous search currently happens under the traditional approach of an operator defining a flight path and collecting data as the path is traversed. While this is a very good approach, there are certain terrain conditions that are not easy to navigate or properly plan a pre-determined flight. This places a large burden on the operator of the UAV.

The goal of this research was to determine if Deep Reinforcement Learning methods are robust enough to train an artificially intelligent agent to autonomously perform the task of searching. Results demonstrate that our prototype successfully serves as a proof-of-concept in which an agent trained by these methods has indeed learned a search strategy, and that its "success rate" consistently exceeds that observed of an agent taking random actions.

2 Background

In Computer Science, Machine Learning is the study and development of algorithms capable of improving their performance of a task based on their experience of performing the task. Typically the field has been classified in three major branches: *Supervised learning*, *Unsupervised learning*, and *Reinforcement learning* (RL). While the first one tries to create a model of the task at hand by being fed examples of labeled data, and the second one tries to discover patterns from unlabeled data examples, the third one approaches the learning problem by maximizing the reward obtained from interacting directly with the task at hand.

Recent developments have shed new light on the hope of using Reinforcement Learning as one possible tool to develop an artificial intelligence capable of supporting SAR operations. The remainder of this section will focus on describing these new developments and different techniques.

2.1 Related work

In distress situations such as the ones experienced after a disaster, time is of the essence. As with any automation, autonomous vehicles could help relieve the burden on SAR team members, allowing them to focus their energy on more complex cognitive tasks such as providing first-aid to wounded people or coordinating search efforts. The European Community has funded several robotic projects to aid SAR such as ICARUS[3]. The goal was to develop a toolbox of integrated components for unmanned SAR; within the toolbox autonomous UAVs were created for searching and early supply delivery. The SHERPA[4] project aimed to develop a mix of ground and aerial vehicles to support SAR activities in hostile environments. INACHUS[5] sought to improve detection and localization of victims using UAVs for fast modeling of buildings and snake-like grounded robots for searching collapsed structures. Autonomy of these vehicles relies heavily on heuristic based approaches. A heuristic is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow, e.g if the program has as an input a map of the search area partitioned according to the probability of finding a lost person in certain regions,

that map is a heuristic.

To support SAR with UAVs authors in [6] document three categories of methods suitable for real-time search of a location: Greedy heuristics, Potential-based heuristics, and Partially Observable MDPs (POMDP). While in their research it was recognized that POMDPs show a lot of promise for creating autonomous UAVs, they acknowledged that using this method would be computationally demanding and that the available algorithms were not sufficiently advanced.

To illustrate the simplicity of greedy heuristic algorithms for autonomous UAVs Algorithm 1 is presented. A heuristic in the form of a map of the area to be searched must be given as an input to the algorithm. This map is called a *belief map* and it must be partitioned into discrete locations s that can be visited by the UAV. Each of these locations is accompanied by the corresponding probability of finding a target within it.

Algorithm 1 Belief map greedy algorithm

```
1:  Initialize BeliefMap
2:  Define starting position s'
3:  while true do
4:      s = s'
5:      observe(s)
6:      if target detected then
7:          Report that target has been found
8:      end if
9:      updateBeliefMap(s)
10:     s' = moveToNextState(s)
11: end while
```

When a *belief map* is available, using this procedure is straightforward; but these heuristics are often not accurate enough with respect to the real world, especially under situations faced by SAR teams. Another drawback is the need for knowing the search area before deploying a UAV for this task. Ideally autonomous UAVs should not need to know in advance the location where they must operate; hence moving away from heuristic-based approaches seems like a reasonable next step in this research field.

2.2 Artificial Neural Networks

An artificial neural network (ANN) is a Machine Learning architecture inspired by how we believe the human brain works. The human nervous system is comprised of special cells called Neurons, each with multiple connections coming in (dendrites) and going out (axons). When the signal coming into a neuron is strong enough, a process called synapse forwards the signal through the axons, passing it on to other neurons.

Instead of relying on a complex algorithm that describes the learning process, ANNs are built upon simple computational components known as perceptrons or units. Units are organized as layers and each unit is connected to units in the next layer. The first layer is called the *input layer*, the last layer is called the *output layer* and intermediate layers are called *hidden layers*. The signal strength, as applied to each connecting edge, is called its *weight*. Each unit processes its inputs using a non-linear activation function; the signal propagates if the value of the function exceeds a predefined threshold. A visual representation of this concepts is presented in figure 1.

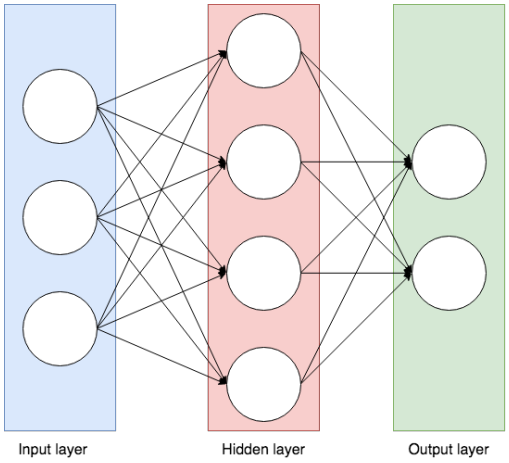


Figure 1: Artificial neural network representation

The learning step of an ANN takes place by updating all weights until the desired function is approximated, e.g the network is able to classify an image into the desired class. Labeled data is used in the process of training an ANN. To update the weights the gradients of the loss function between the final ANN output and the expected output are backpropagated into the network, scaled by a learning factor. This optimization process is generally called Gradient Descent. Different methods to perform Gradient Descent are currently available

including Stochastic Gradient Descent, RMSProp, Adam, and others[7].

Traditionally, each unit in one layer is connected to all the units in the next layer. This network structure is known as a Fully Connected Network (FCN). Other architectures have been proposed, such as the Convolutional Neural Network (CNN) in which each unit is mapped to a specific subset of units in the next layer.

2.3 Convolutional Neural Networks

CNN is an ANN architecture that takes advantage of the hierarchical structure of its input data. Traditionally used in image recognition applications, the idea is to use a convolution operator to aggregate certain regions of the input image into a volume known as a convolutional layer. Each column of the convolutional layer looks at the same region of the input data, and each slice of the volume is a distinctive feature map over the data. By stacking the feature maps in this manner the network can perform automatic feature extraction since each filter contains a different representation of the data. Figure 2 summarize the former concepts. At the end of the convolution layer one or more fully connected layers can be placed to perform the classification task of the input. Training of a CNN happens in the same fashion as FCN, using gradient descent optimization and backpropagation.

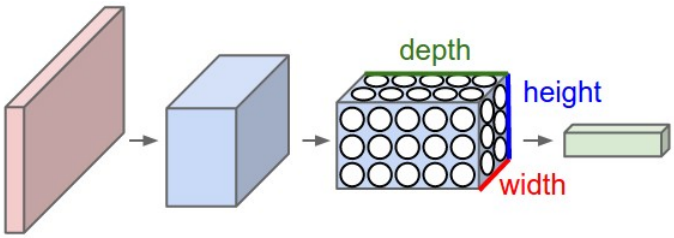


Figure 2: Convolutional neural network visual representation[8]

Four parameters are used to create the convolution layer: filter size, stride, depth, and zero-padding. Filter size determines how large is the portion of the input that should be convolved. The stride refers to how big are the jumps between each convolution, e.g in an image a stride of one applies the convolution to every pixel, while a stride of three applies the convolution every three pixels. The depth refers to the number of filters to be created on the input data; more feature maps mean more features to be automatically extracted. Zero padding is

used to preserve the input and output size by making the feature map the same size as the input.

The CNN network class is considered the heart of Deep Learning given the size of the convolutional layer (typically consisting of many layers comprising millions of units), and its ability to automatically extract features. As will be described later, the Deep Q-Network relies on this architecture.

2.4 Reinforcement Learning

When talking about RL there are two major components: The decision-maker, referred to as the Agent; and the Environment that receives the actions of the agent and provides feedback on the consequences of taking an action, e.g where it will end up. These components are modeled in the form of a Markov Decision Process (MDP).

The constant interaction between agent and environment can be thought of as a closed loop, in which the agent depends on the environment to receive observations and rewards, while the environment needs the agent to observe it. Figure 3 illustrates this behavior. Learning in RL methods takes place by allowing the agent to refine its strategy by interacting within this loop.

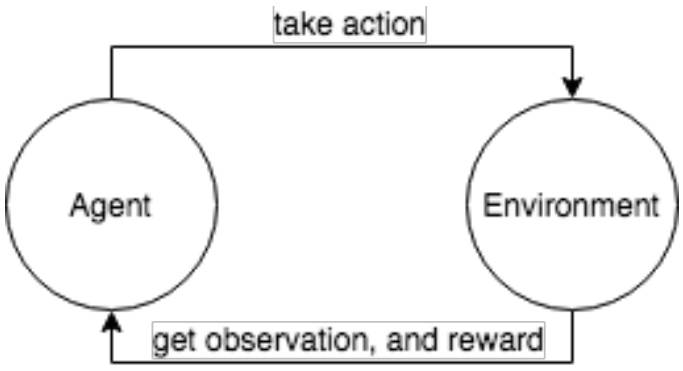


Figure 3: Reinforcement Learning loop

2.4.1 Markov Decision Process

An MDP is a decision-making framework consisting of a tuple $(S, A, R(s), T(s, a, s'))$ where S is the State space, everywhere an agent might end up; $A(s)$ is the Action space, what the

decision maker is allowed to perform in a given state s, executing those actions will cause the environment to return a new current state and a reward; $R(s)$ is a Reward function, which returns a scalar value representing the goodness of taking an action, this will be the driver of the learning process of a RL method; and $T(s,a,s') \sim P(s'|s,a)$ is the set of Transition probabilities, the probability of ending in state s' given that the agent takes action a while being in state s; this is usually referred to as the model.

It is called a *Markov Process* because it has the *Markovian Property*. That is, the probability of ending in the state s' depends only on the present state s and taking the action a, regardless of the walker's history. In other words, only the present state matters.

An initial state $s_t \in S$ is the state from which a decision-maker will start to walk an MDP. A goal or terminal state $g \in S$ is defined as one where all actions transition to itself with a probability of 1 and the reward for taking any action in that state is 0. An MDP can have one or many initial states and 0 or many terminal states. The sequence of actions from an initial state, potentially reaching a goal state is known as an episode, and each action within an episode is known as a step.

In order for an agent being able to navigate an MDP a policy must be set in place. A policy $\pi(s)$ is a map of states to actions. At any given state s the policy $\pi(s)$ will return which action to take. The core problem of MDPs is to find an optimal policy π*; to find the $\pi(s)$ that maximizes the cumulative reward while walking the MDP.

2.4.2 Bellman equation

The value of a policy π is the function that gives the sum of all the rewards received by following a policy on an environment. When there is a limit on how many actions the agent will perform in the environment, starting from state s_t the value function is: $V^{\pi}(s_t) = \sum_{i=1}^{T} E[r_{t+i}]$ where T is the limit on the number of actions and $E[.]$ is the expected reward.

When there is no fixed limit of steps to be taken by episode a discount factor $\gamma : 0 \leq \gamma < 1$ is used in order to calculate the reward in the form of the geometric sequence $V^{\pi}(s_t) = \sum_{i=1}^{\infty} \gamma^{i-1} E[r_{t+i}]$. When $\gamma = 0$ only immediate rewards count, and as γ approaches 1 later rewards count more. The higher the discount factor the more farsighted the agent will be.

The optimal policy is then the one that yields the highest value from all possible policies, commonly denoted as $V^*(s_t)$. To find the optimal Value function Bellman's equation is used:

$$V^*(s_t) = \max_{a_t} (E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) V^*(s_{t+1}))$$

The value function tells how good it is to be at a given state, and since the agent is following a policy, an action-value function can be defined in terms of the goodness of taking an action a_t when the agent is in state s_t. This function is known as action-value and is expressed as $Q(s_t, a_t)$. Similarly to the value function the optimal action-value $Q^*(s_t, a_t)$ is the one that maximizes the expected reward. Expressing Bellman's equation in terms of action-value gives the following form:

$$Q^*(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$$

It is possible then to find the optimal policy by solving the previous equation due to the fact that the optimal value has to be following the optimal policy in order to yield the maximum expected return.

2.4.3 RL methods

Making use of Bellman's equation, two classes of algorithms are known to find the optimal Value: Value iteration and Policy iteration. When the transition probabilities, commonly referred to as the model, are known beforehand the RL method is said to be *model-based*. This means that no exploration during the walking of the MDP has to be made in order to find the optimal Value and consequently the optimal policy. When the model is not known in adevance the exploration vs. exploitation dilemma arises: the agent needs to walk different states following a policy in order to find the value of said policy, but at the same time it needs to explore new states that might have better values for the policy. MDPs where the model is unknown are called *model-free*.

The idea behind value iteration[9] is to start by initializing a value function to arbitrary values, $\hat{V}_0$. Then calculate the next iteration $\hat{V}_{t+1}$ based on the estimate of $\hat{V}_t$, and repeat this

process until the Value converges. The model-based version of this process is given in Algorithm 2.

Algorithm 2 Model-based Value iteration

1: **Initialize** $\hat{V}$ **to arbitrary values**
2: **while** $\hat{V}(s)$ has not converged **do**
3: **for all** $s \in S$ **do**
4: **for all** $a \in A$ **do**
5: $Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} (T(s,a,s') \hat{V}(s'))$
6: **end for**
7: $\hat{V}(s) \leftarrow \max_a Q(s,a)$
8: **end for**
9: **end while**

Instead of iterating through the Value function, the policy itself could be updated repeatedly until convergence occurs. That is the idea behind policy iteration[9]. Pseudocode is provided in Algorithm 3. In the former it is of special interest line 3.5. Note that there is no *max* function involved, which effectively turns the equations into linear equations. Although this approach takes longer to compute at each step than the value iteration counterpart, fewer iterations of the algorithm are needed in ordxer to find the optimal policy.

Algorithm 3 Model-based policy iteration

1: **Initialize** π' **to arbitrary values**
2: **repeat**
3: $\pi \leftarrow \pi'$
4: Compute values of π by solving the linear equations:
5: $V^{\pi}(s) = R(s,\pi(s)) + \gamma \sum_{s'} T(s,\pi(s),s') V^{\pi}(s')$
6: Improve π' at each state by:
7: $\pi'(s) \leftarrow \text{argmax}_a (R(s,a)) + \gamma \sum_{s'} T(s,a,s') V^{\pi}(s'))$
8: $\pi' \leftarrow \pi$
9: **until** $s == g$

For the case where the model is not available a technique called Temporal Difference (TD) is used. This technique relies on exploration of the states to find new Values needed for the finding of the optimal policy. To solve the exploration vs exploitation dilemma an epsilon greedy (ε-*greedy*) approach is suggested, which is simply to choose a random action $a \in A$ with probability ε and follow the policy $1 - \varepsilon$ times. The most well-known TD algorithm is called Q-learning[9] and is presented in Algorithm 4.

Algorithm 4 Q-learning

Initialize $Q(s,a)$ **to arbitrary values**
for all episodes **do**
    Initialize s
    repeat
        choose a from Q with ε-greedy strategy
        take action a, receive r, s' from environment
        $Q(s,a) \leftarrow Q(s,a) + \eta(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$ ▷ η is a learning factor
        $s \leftarrow s'$
    until $s == g$
end for

In a small finite State space storing the Q-values in a table for the Q-learning algorithm is feasible, however high dimensional spaces such as a continuous State or Action spaces pose a problem both in memory management and in the difficulty of updating the values efficiently to realistically train the agent. Basically, a large State space or Action space could lead to situations in which the RL-driven search for an optimal policy would take infinite time.

To solve the problem the use of a universal function approximator such as an Artificial Neural Network, designed to replace stored Q-values, has been proposed. However, in the setting of RL this approach was historically unsuccessful due to the nature of Q-learning – recent experience tends to dominate the training examples in the Q-value network, causing it to "unlearn" the correct outputs of the Q-value. Recent developments in other areas of machine learning (e.g. deep neural nets) provided the means of creating RL algorithms capable of moving beyond the heuristic approach to one based on MDPs.

When the state received by the agent does not contain the entire state of the environment but rather only a fraction of the states, the process is called Partially Observable (POMDP). The partial state received from the environment is called an observation. Instead of finding the policy the agent is trying to find the model of the MDP, therefore a belief function is needed to describe to the agent the probability of observing the state s while being in that state.

2.5 Deep Q-Network

The first algorithm that successfully incorporated deep learning in RL is known as the Deep-Q Network (DQN)[10]. As described in the previous subsection, Q-learning is an RL algorithm

that uses value function iteration based on the optimal Bellman equation. The result of the value function for each action is stored and updated in what is known as the Q-table. The main idea behind DQN is to replace the traditional Q-table with a deep neural network that is trained from samples of stored experiences.

At its heart DQN proposes three innovations in the RL space. The first is to use a convolutional neural network-based architecture to process images obtained from the environment. These deep, specialized neural networks extract different visual features in each layer by applying filters, known as convolutions, to the image. Using a Deep network instead of a traditional ANN allows the algorithm to work with only one state representation, in contrast to former proposed solutions where the history of the agent and the actions were fed into the ANN. To differentiate this network from the algorithm the name Q-network will be used to refer to this type of architecture, while DQN will refer to the algorithm.

The second innovation, similar to what is proposed in [11], uses a "memory" of experiences for training the Q-network. In this context an experience is a tuple of (s, a, r, s') where s is the current state at that given time, a is the action, r is the received reward, and s' is the new state after taking action a. Training the network on a subset of states, rather than on all states seen by an agent, helps the network avoid learning from only what it is immediately experiencing.

As a final innovation, the inclusion of a target Q-Network is designed to control the loss function used during training; that is, instead of calculating the loss against the gradients carried by the acting network, gradients are backed up in a separate network which is used to calculate the target values of the function. The target network is backed up at a set interval preventing the gradients from falling into a local minima, similar to what it is accomplished by training over the experience replay.

2.6 Hindsight Experience Replay

Hindsight Experience Replay[12] is an improvement on Experience Replay that adds the concept of an achieved goal to the Q-network architecture. In sparse environments, one where no positive reward is received until the desired goal is achieved, Q-network approaches need ex-

tensive training periods. The idea is then to add the concept of goal to the Q-network, rewarding not only reaching the desired goal state, but also rewarding intermediate achievements per visited states e.g not crashing, so even if the agent does not succeed in a given episode it is learning something.

To implement this strategy the episode experience needs to be stored. For every experience in the episode, the episode must be stored in the experience replay, then a set of next states are sampled from the episode experience as goals for the current state. For every sampled goal, a new experience is created with the sampled goal as the goal of the experience. If the next state of the current state of the experience is equal to the sampled goal a reward equal to that received for achieving the target goal is set to that state as the reward of that experience. Finally the new experiences are stored into the experience replay.

2.7 Actor-Critic methods

As described in previous sections, an alternative to value iteration learning is policy iteration. Similar to ANNs, in order to find the optimal policy its gradients are followed to update the policy.

Policy gradient methods incorporate two major, interacting steps: policy evaluation and policy improvement. Policy evaluation estimates the value function based on the latest policy improvement, while policy improvement updates the policy with the action that maximizes the value function at each evaluated state. Policy gradient methods are particularly useful when deployed in continuous action spaces by informing the agent as to how good selecting a specific action turned out to be.

Asynchronous Advantage Actor-Critic network[13], combines the Q-network approach with policy gradient. In this context, the Actor network behaves as the policy evaluation component. The Critic implements the policy improvement step, using Q-learning based on the observed reward. The Advantage Actor-Critic method extends this approach further by using an advantage estimate function that helps the agent determine not only how good the selected action was, but how much better it was than other actions.

3 Implementation

3.1 Environment

The experimental environment for this research project was constructed using Microsoft's AirSim vehicle simulator[14], which provides ground truth observations based on its accurate physics simulation of quadcopter UAVs. The AirSim API also includes high level vehicle maneuver instructions that enable agent behavior consisting of four simple actions: move forward, turn right or left by thirty degrees, and hover. An overview of the different components of the environment can be seen in Figure 4.

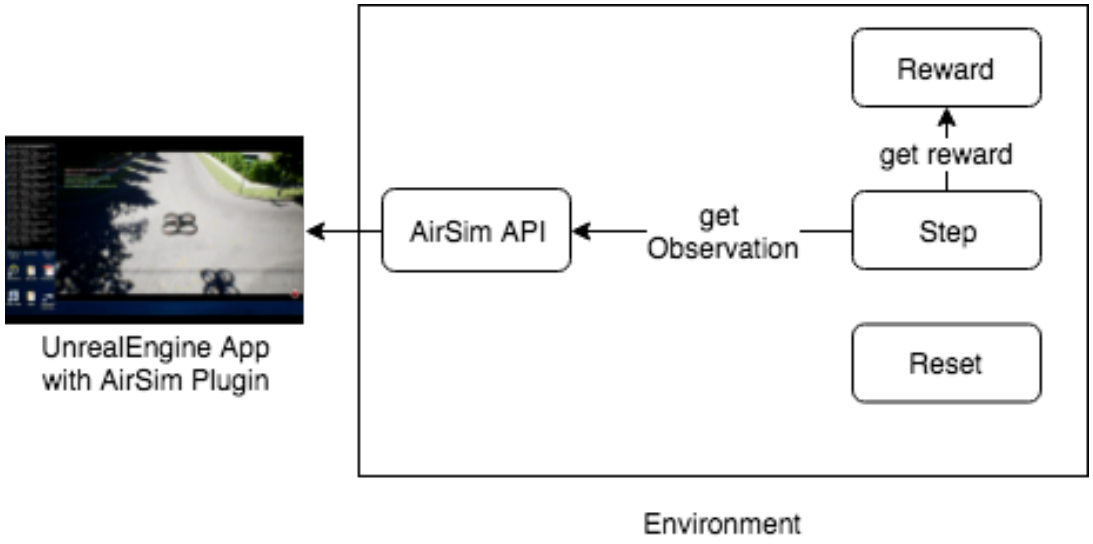


Figure 4: Overall Environment Architecture

The simulation performed in AirSim executes as a plug-in for Unreal Engine, a video game engine with photo-realistic rendering capabilities. This makes AirSim a very convenient platform on which to train vehicles in various types of visually realistic environments, provided that the needed 3D modeling skills are available within the team. Figure 5 shows an example of three different simulated scenes with different levels of complexity, as created in Unreal Engine[15].

The RL environment communicates to the simulated scene through AirSim's API. In the case of the UAV, the API provides convenient commands, similar to those available with commercial autopilots such as take off, move by velocity, move to coordinate, get position, get orientation, get scene image, get depth image and so on.

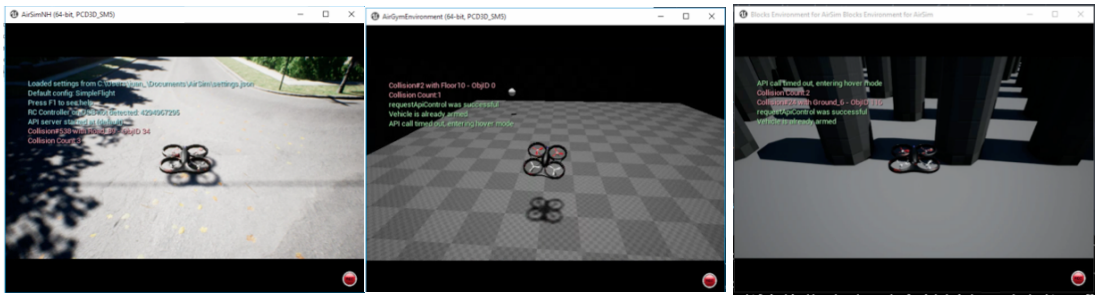


Figure 5: Three different simulator scenes created in Unreal Engine

When training is taking place two functions of the environment are called: Step and Reward. The step function is in charge of executing the action selected by the agent, retrieving an image, and position and orientation information using the AirSim API, and calling the reward function on the state and action passed by the agent. The environment reward system includes two modes: shaped reward and sparse reward. When shaped reward is in place the agent is rewarded according to the area of the observed image occupied by the target object. Sparse reward means that every action except reaching the goal state rewards the agent with the same negative reward. This system was included to facilitate adding Hindsight Experience Replay to the agent.

3.2 Agent

The agent was developed using Python, which allowed for smooth integration with libraries such as TensorFlow[16] and Keras[17], used to implement the desired deep learning network architectures.

The agent receives observations from the environment, consisting of a depth image and collision detection information. An overview of the implemented DQN architecture can be seen in Figure 6

The DQN network structure is comprised of three convolutional layers: the first with 32 filters and the other two with 64 filters. The activation function for all layers is the Rectified Linear Unit (ReLU) and the activation function for the output is Softmax. To include the concept of goal for Hindsight Experience Replay a similar network is placed parallel to the DQN network to receive the goal input. At the end of both convolutional layers a concatenation of the output of both networks is placed before connecting it to a fully-connected layer com-

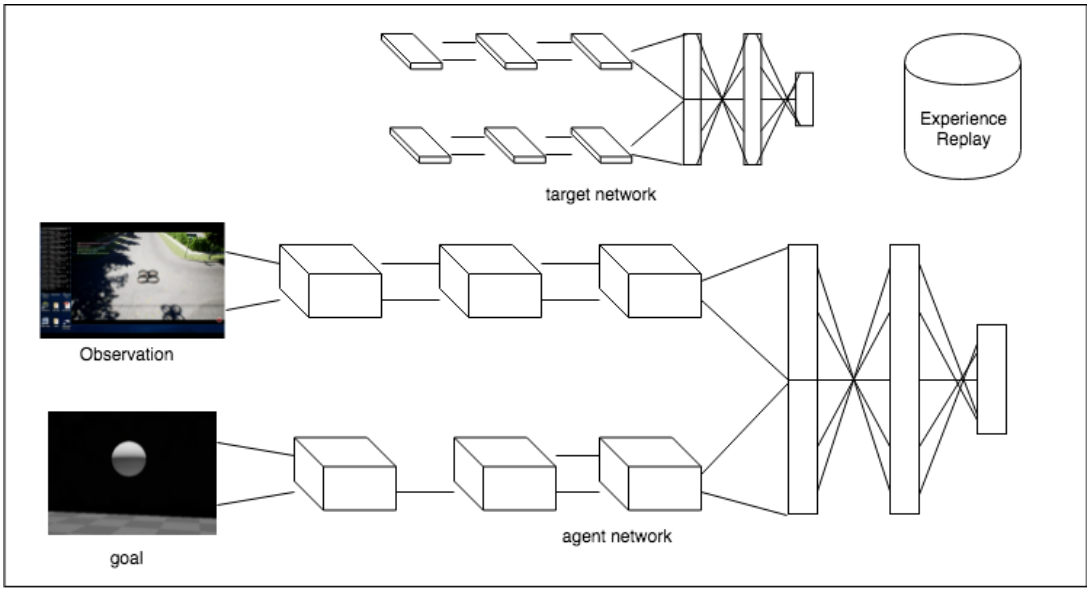


Figure 6: Overall Agent Architecture

posed of 512 neurons that precedes the output of the four possible actions: Move forward, move left or right by 30 degrees or no operation.

The procedure used in implementing the DQN part of the agent was based on ideas described in [18], but was significantly modified to include the concept of Hindsight Experience Replay.

4 Results

Training the agent took 96 hours and 25 minutes to complete 10000 episodes on a Windows Machine with 16GB of RAM, an Intel Core i-7 4750K and an Nvidia 780 Ti GPU. After training the agent, 100 episodes of evaluation mode were conducted to measure its performance. Trained model performance was then compared against a random agent, one where all actions taken in the MDP are arbitrary selected.

The training scene for the test case was a small room of 20 by 20 meters, with the target goal being a floating ball on the right side of the room at position (15,15). States where the UAV crashed were considered terminal states.

Since the agent did not have its global position it remembered its starting position as (0,0,0). Turning of the UAV was recorded as movement values in the positive or negative direction of the Y-axis. Values less than zero in the X-axis were restricted since the starting position of the UAV is against a wall. The Z-axis was fixed to 1.5 meters.

Selected metrics for this comparison were the average amount of decisions made by an agent during an episode (Average steps), the average length per episode in seconds during test (Average time), and the percentage of times the agent found the desired target (Success rate). For the trained agent, Average steps taken were 258.54, and Average episode time was 35 seconds. Table 1 summarizes the results of this comparison.

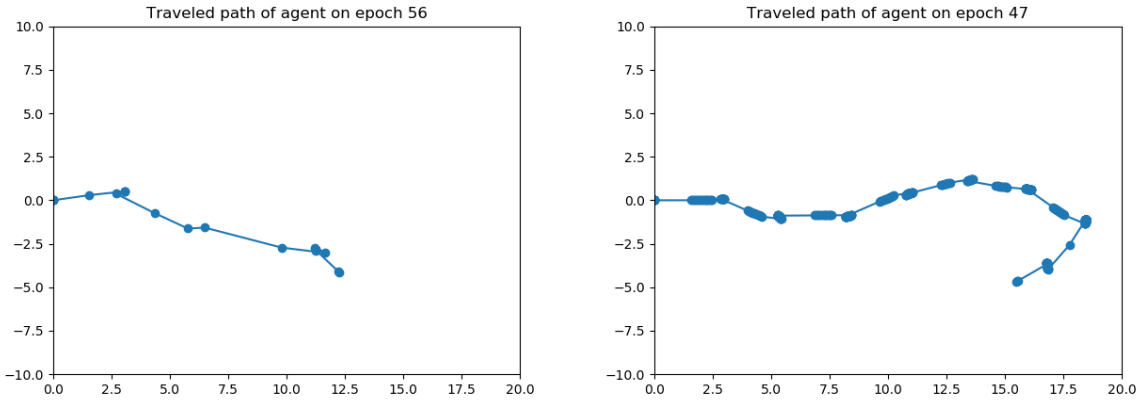
	Average Steps	Average Time	Success rate
Random Agent	22.35	28s	5%
Trained Agent	258.54	35s	8%

Table 1: Random agent and Trained agent performance

In general, taking more steps is an indicator of an agent which is analyzing its surroundinga before taking an action (i.e. closed loop iterations). The fact that the trained agent was able to correct (reverse) its course when the back wall was encountered (as seen in Figure 7) is an improvement over the random agent, which would have to randomly select the action of turning to one side six consecutive times in order to complete a 180 degree turn. In other words, this is evidence that the trained agent demonstrates a beneficial strategy. It could be said that when

a random agent finds the target it is truly by chance, in contrast to the trained agent that was following a learned policy.

The traveled path of both the agents was recorded for each evaluation episode. Figure 7 provides a top-down perspective of a sample successful run for each type of agent. Of special interest is the trained agent; it has "learned", when reaching the end of the room, to reverse its path to look in a different direction.



(a) Random agent path

(b) Trained agent path

Figure 7: Random agent and Trained agent paths in episodes where target was found

5 Conclusion and Future Work

The prototype successfully demonstrated the feasibility of using an artificial intelligence to direct unmanned aerial vehicles to search.

However, given the real-time, real-physics nature of a single simulated run, training time simply takes too long, inhibiting the success rate of the intelligent system. Two alternatives could be implemented to address this problem: distributing the training process onto different machines which collaborate to compute a global target network, or spawning multiple agents into the same AirSim application and assigning a processor core to each agent.

The reward function, which serves as the truth holder of the learning process, greatly influences agent learning. For this research, a sparse reward was used since the goal was simply to find an object. Shaped reward functions would allow the incorporation of expert knowledge, potentially enhancing agent learning.

Finally, one reason the AirSim package was chosen was smooth integration with the photorealistic capabilities of Unreal Engine. To develop a general-purpose solution an agent would need to be trained on many different environments. This would require both the artistic creation of those environments and substantial additional training time.

References

[1] Ken Phillips et al. “Wilderness Search Strategy and Tactics”. In: *Wilderness & Environmental Medicine* 25.2 (June 2014), pp. 166–176. ISSN: 10806032. DOI: 10.1016/j.wem.2014.02.006. URL: http://www.ncbi.nlm.nih.gov/pubmed/24792134%20http://linkinghub.elsevier.com/retrieve/pii/S1080603214000829.

[2] Geert De Cubber et al. *Search and Rescue Robotics - From Theory to Practice*. 2017. ISBN: 978-953-51-3375-9. DOI: 10.5772/intechopen.68449. URL: http://www.intechopen.com/books/search-and-rescue-robotics-from-theory-to-practice.

[3] Geert De Cubber et al. “ICARUS: Providing Unmanned Search and Rescue Tools”. In: *Remotely Piloted Aircraft Systems - The Global Perspective - Yearbook 2013/2014* (2013). URL: http://www.fp7-icarus.eu/sites/fp7-icarus.eu/files/publications/Providing%20Unmanned%20Search%20and%20Rescue%20Tools.pdf%20http://www.uvs-info.com/index.php?option=com%7B%5C_%7Dflippingbook%7B%5C&%7Dview=book%7B%5C&%7Did=16%7B%5C&%7Dpage=1%7B%5C&%7DItemid=731.

[4] L. Marconi et al. “The SHERPA project: Smart collaboration between humans and ground-aerial robots for improving rescuing activities in alpine environments”. In: *2012 IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2012* 00.c (2012). DOI: 10.1109/SSRR.2012.6523905.

[5] Styliani Verykokou et al. “UAV-based 3D modelling of disaster scenes for Urban Search and Rescue”. In: *2016 IEEE International Conference on Imaging Systems and Techniques (IST)*. IEEE, Oct. 2016, pp. 106–111. ISBN: 978-1-5090-1817-8. DOI: 10.1109/IST.2016.7738206. URL: http://ieeexplore.ieee.org/document/7738206/.

[6] Sonia Waharte and Niki Trigoni. “Supporting search and rescue operations with UAVs”. In: *Proceedings - EST 2010 - 2010 International Conference on Emerging Security Technologies, ROBOSEC 2010 - Robots and Security, LAB-RS 2010 - Learning and Adaptive Behavior in Robotic Systems* (2010), pp. 142–147. DOI: 10.1109/EST.2010.31.

[7] Andrej Karpathy and Justin Johnson. *Neural Networks Part 3: Learning and Evaluation*. URL: http://cs231n.github.io/neural-networks-3/ (visited on 08/05/2018).

[8] Andrej Karpathy and Justin Johnson. *Convolutional Neural Networks (CNNs / ConvNets)*. URL: http://cs231n.github.io/convolutional-networks/ (visited on 08/07/2018).

[9] Richard S Sutton and Andrew G Barto. “Reinforcement learning: an introduction.” In: *UCL,Computer Science Department, Reinforcement Learning Lectures* (2017), p. 1054. ISSN: 1045-9227. DOI: 10.1109/TNN.1998.712192. arXiv: arXiv:1011.1669v3. URL: http://incompleteideas.net/sutton/book/bookdraft2017june.pdf.

[10] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 0028-0836. DOI: 10.1038/nature14236. URL: http://www.nature.com/articles/nature14236.

[11] Chris Gaskett, David Wettergreen, and Alexander Zelinsky. “Q-learning in continuous state and action spaces”. In: *Advanced Topics in Artificial Intelligence. AI 1999. Lecture Notes in Computer Science*. Vol. 1747. Springer, Berlin, Heidelberg, 1999, pp. 417–428. ISBN: 3540668225. DOI: 10.1007/3-540-46695-9_35. URL: http://link.springer.com/10.1007/3-540-46695-9%7B%5C_%7D35.

[12] Marcin Andrychowicz et al. “Hindsight Experience Replay”. In: (July 2017). arXiv: 1707.01495. URL: http://arxiv.org/abs/1707.01495.

[13] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: 48 (2016). ISSN: 1938-7228. DOI: 10.1007/s10107-003-0404-8. arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[14] Shital Shah et al. “AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles”. In: (May 2017). arXiv: 1705.05065. URL: http://arxiv.org/abs/1705.05065.

[15] Epic Games. *What is Unreal Engine 4*. 2017. URL: https://www.unrealengine.com/en-US/what-is-unreal-engine-4%20https://www.unrealengine.com/en-US/what-is-unreal-engine-4%7B%5C%7D0Ahttps://www.unrealengine.com/what-is-unreal-engine-4 (visited on 08/07/2018).

[16] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. Tech. rep. 2015. URL: www.tensorflow.org..

[17] François Chollet et al. *Keras*. 2015. URL: https://keras.io.

[18] Fangwei Zhong et al. *Gym-UnrealCV: Realistic virtual worlds for visual reinforcement learning*. 2017. URL: https://github.com/unrealcv/gym-unrealcv (visited on 08/05/2018).

Thesis Submission Agreement

I agree to grant the Grand Valley State University Libraries a non-exclusive license to provide online **open access** to my submitted thesis ("the Work") via ScholarWorks@GVSU. *You retain all copyright in your work, but you give us permission to make it available online for anyone to read and to preserve it in ScholarWorks@GVSU.*

ScholarWorks@GVSU is an open-access repository maintained by the Grand Valley State University Libraries to showcase, preserve, and provide access to the scholarly and creative work produced by the university community. For more information, email scholarworks@gvsu.edu, or visit our webpage at http://scholarworks.gvsu.edu/about.html.

I warrant as follows:

1 I hold the copyright to this work, and agree to permit this work to be posted in the ScholarWorks@GVSU institutional repository.

2 I understand that accepted works may be posted immediately as submitted, unless I request otherwise.

3 I have read, understand, and agree to abide by the policies of ScholarWorks@GVSU. (Our policies can be found at: http://scholarworks.gvsu.edu/about.html)

By typing my name into the Author field I am agreeing to the terms above and attaching my electronic signature. I understand that if I do not agree to these terms, I should not type my name.

Title of thesis: Deep Reinforcement Learning for Autonomous Search and Rescue

Author: Juan Gonzalo Cárcamo Zuluaga Date: 07-17-2018

Embargo Option: *(You may choose to restrict electronic access to your work for up to three years from the date the work is submitted to ScholarWorks. When the embargo expires, your work will automatically become available.)*

Length of Embargo: _____

Keywords: *(six relevant words to describe the content of your thesis)*

RL, MDP, SAR, Simulation, UAV, DQN