Optimization of UAV message delivery trajectories in partitioned Mobile Ad-hoc Networks using Deep Queue Learning

Ana Catarina F. Rodrigues, *Instituto Superior Tecnico, Av. Rovisco Pais 1, 1049-001, Lisbon, Portugal*
ana.catarina.f.rodrigues@tecnico.ulisboa.pt

***Abstract*—In the world we live where almost everything depends on wireless and mobile communications, it almost becomes essential this type of connections. This reality is lived in metropolitan cities where network and mobile coverage exists. Sometimes, there may be obstacles and these connections may be interrupted, or even not exist yet, in more rural and unmanaged places, where resources are not enough to build infrastructure, or in a tragic scenario where they were destroyed. To fill this gap, the use of unmanned aerial vehicles (UAVs), were thought to be used in the communication between places forming a network of nodes to be visited offering network coverage. In the previously studied work, the use of learning techniques was proposed. The reinforcement learning (RL) algorithm was chosen to establish the path between the network nodes. In this work, it was proposed the re-creation of the RL algorithm and then the optimization of this algorithm where neural networks are introduced, Deep Reinforcement Learning (DRL). After the development of the algorithms, it is possible to conclude that the most efficient approach was the RL. It would be expected that the introduction of the DRL approach would have better behavior than RL.**

***Index Terms*—Communications, Machine Learning, Reinforcement Learning, Neural Networks, Unmanned Aerial Vehicles**

I. Introduction

We live in the twenty-first century, so wireless communication and connection are ubiquitous, particularly in metropolitan areas with mobile and network coverage, allowing users to stay connected at all times and in all locations. Some places face other realities, where network infrastructures do not exist.

However, there is a need to communicate in these situations, so UAVs (drones) can be a mechanism that allows this to still happen, without the need to build expensive infrastructure.

In this paper, the UAV will be used in the communication relaying theme where UAVs are used to send and receive messages between users.

This study will be done using a DTN, a network in which nodes are static and are not connected continuously. The extra mobile node will be a UAV.

In this report the drone will communicate with all of the nodes allowing them to send messages between all of them, in an All-to-All situation. In order to optimize and choose the drone's best route to deliver the messages between the nodes a learning technique will be used, that learns the optimal and best path through RL and DRL. The second one is a subfield of machine learning that combines Neural Networks and RL, it will be investigated, in order to possible culminate in a solution to the routing problem.

II. State of the Art

The goal corresponds to determining the optimal path through an algorithm for a DTN with *N* static nodes that will interact via a message carrier, which will be a single UAV. The requirements are to deliver the largest number of messages in the shortest period of time possible. To find the optimal path, two algorithms will be developed. The first one uses RL and the other with DRL.

This chapter will provide an overview of the theory the algorithms are based on. To simulate the environment and the used algorithms to find the optimal path a toolkit, Open AI Gym is presented.

A. Markov Decision Processes

The basis of structuring problems solved with RL are Markov Decision Processes (MDP). MDP allows to formalize of sequential decision-making. The main objective is to estimate the value for each element of a function $Q_*(s, a)$, that is the value for each state-action pair or else, for $V_*(s)$, which is the associated value for each state. These values are given by following an optimal policy. The associated value of each state-action pair is the discounted sum of the rewards received in each state and future ones. MDP has a decision maker, the *agent*, that interacts with the *environment* that is placed in. These interactions occur sequentially over time. At each time step, the agent will get some representation of the environment's *state*, and perform a selection of the action to take, the environment will then transition to a new state, and a *reward* will be given to the agent as a consequent of the action the agent chose. The goal is to maximize the actual and cumulative reward. The components of a MDP are a tuple (A, E, S, A, P): Agent (agent being trained to make decisions), Environment (space where the agent, takes the agent's current state and action as input and outputs the reward and next state), State (situation in which the agent finds itself), Action (set of possible moves for the agent to take) and Policy (process behind picking an action)

Another important definition of MDPs is the Markov Property, that given the current state, the future state does not depend on the past ones. The future state only depends on its immediate previous state. At each time step t = 0,1,2,.. the agent receives some representation of the agents state S_t, based on this state the agent selects an action A_t, this gives us (A_t,S_t) pair. Time is incremented to the next time step $t + 1$,

the environment transitions to S_{t+1} and the agent receives a numerical reward, R_{t+1} for the action A_t taken at S_t. At each time t, we have $f(A_t,S_t) = R_{t+1}$.

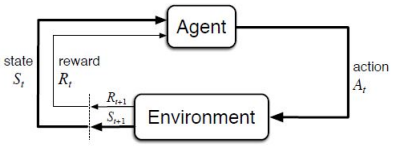


Fig. 1. The agent-environment interaction in RL [1]

In Figure 1, a state transition is represented. At temporal instant t the environment is at state S_t, the agent observes the environment and selects the action A_t. Then, the environment is shifted to state S_{t+1} and grants the agent R_{t+1} as a reward.

The reward R_t and state S have well defined probability distributions. These distributions depend on the preceding state and action that occurred in the previous time $t-1$. For example, $s' \epsilon S$ and $r \epsilon R$, there is some probability $S_t = s'$ and $R_t = r$. This probability is determined by the particular values of the preceding state $s \epsilon S$ and action $a \epsilon A(s)$. $A(s)$ is a set of actions that can be taken from state s. The probability of transitioning to state s' with reward r from taking action a in state s can be defined with for all $s' \epsilon S$, $r \epsilon R$, $a \epsilon A$:

$$p(s', r|s, a) = P_r\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

The goal of an agent in an MDP is to maximize its cumulative rewards. The concept of *expected return* of the rewards at a given time step corresponds to the sum of future rewards, defined G_t. This concept is crucial because the agent's objective is to maximize the expected return, it is what drives the agent to make the decisions it makes. Another important concept is the discounted return, now the agent's goal is to maximize the expected discounted return of rewards, he will choose action A_t at each time t to maximize the expected discounted reward. There is a discount factor, $\gamma\ \varepsilon[0, 1]$, it will be the rate for which the future rewards are discounted and will determine the present value of future rewards. The agent will care more about immediate reward over future rewards since future rewards will be more heavily discounted. After presenting the concept of an MDP, it is necessary to explain policy and value function. The probability of an agent selecting a specific action from a specific state is defined by the policy. In terms of rewards, selecting one action over another in a given state may increase or decrease the agent's reward, the value function will help the agent decide which action to take in order to maximize its reward.

A policy is a function that maps a given state to probabilities of selecting each action from that state and can be denoted as π. An agent follows policy π at time t, so $\pi(a, s)$ is the probability that $A_t = a$ if $S_t = s$. For each state $s \epsilon S$, π is a probability distribution over $a \epsilon A(s)$.

Value functions are functions of state-action pairs, that can estimate how good it is for the agent to perform an action in a given state. The notion of how good a state-action pair is depends on the expected return. Consequently, value functions are defined with respect to specific ways of acting, because the way an agent acts is influenced by the policy it's following. The *state-value function* for policy π is v_π and tells us how good any given state is for an agent following policy π (gives us the value of a state under π). Mathematically, the value of state s under policy π is the expected return from starting from state s at time t and following π:

$$v_\pi(s) = E_\pi[G_t|S_t = s].$$

Another function is the *action-value function*, q_π, which can estimate how good it is for an agent to take any action from a state while following policy π (gives us the value of an action under π). Mathematically, the value of action a in state s under π is the expected return from starting at state s at time t, taking action a and following π.

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a].$$

The action-value function is commonly known as Q-function and the output for any given state-action pair is called Q-value.

An RL algorithm learns optimal policies, the main goal is to find an optimal policy. A policy π is considered to be better or equal to policy π' if the expected return of π is greater than or equal to the expected return of π' for all states.

$$\pi \geq \pi' \Leftrightarrow v_\pi(s) \geq v'_\pi(s) \forall s \epsilon S.$$

$v_\pi(s)$ gives the expected return for starting in state s and following π. An optimal policy is a policy that is better than or at least the same as all other policies. An optimal policy has an associated *optimal state-value* function, that gives the largest expect return achievable by any policy π for each state, defined as:

$$v_*(s) = \max_\pi v_\pi(s). \tag{1}$$

There is an *optimal action-value function* associated too, (optimal Q-function), that gives the largest expected return achievable by any policy π for each possible state-action pair, defined as:

$$q_*(s, a) = \max_\pi q_\pi(s, a). \tag{2}$$

The optimal *Q-function* (q_*) has one fundamental property: it must satisfy the *Bellman optimal equation*. This equation states that for any state-action pair (s, a) at time instant t, the expected return from starting in state s and choosing action a, by following the optimal policy, the expected return will be equal to the expected reward which is R_{t+1}. This reward is obtained by selecting action a in state s plus the maximum of the expected discounted return. The maximum of the expected discounted return is achievable of any possible next state-action pair (s', a') where state s' and a' is the potential next possible state-action pair.

The Bellman equation is defined as:

$$q_*(s, a) = E[R_{t+1} + \gamma \max_{a'} q_*(s', a')]. \tag{3}$$

The use of the Bellman equation is common to find q_*. The optimal policy can be determined because with q_*, for any state s, with a RL algorithm, it is possible to find the action a that maximizes $q_*(s, a)$

B. Exploration vs. Exploitation

There are two important concepts that are presented in learning algorithms. Exploration is the act of exploring an environment to find information about it. Exploitation is the act of exploiting information that is already known about the environment, in order to maximize the expected return.

In order to get the balance between exploiting and exploring, there are some techniques, but we will enhance the epsilon greedy strategy. An exploration rate ε will be initially set to 1, and it is the probability that the agent will explore the environment, rather than exploit. In the beginning, it is one hundred percent sure that the agent will explore. As the agent learns more about the environment, at the beginning of each episode, ε will decay, so the probability of the agent exploring becomes smaller, while the agent is learning and knowing more about the environment, it will become greedy in terms of exploiting. In each time step, to determine if the agent will exploit or explore, a random number is generated, between 0 and 1. If the number is greater than epsilon, will choose via exploitation, chooses the action with the highest Q-value in that state. If the random value is smaller than epsilon, the next action will be chosen via exploration, choosing a random action and seeing what happens in the environment.

C. Reinforcement Learning (RL)

RL is a field of ML, next to Supervised and Unsupervised Learning. It is a way of solving MDP and consists in an agent, placed in an environment that learns to perform actions that are rewarded, leading to the maximum reward, using exploitation and exploration of information.

1) TD-Learning: TD, learns from raw experiences without knowing the dynamics of the environment, it does not have to wait for the outcome of the full simulation to update the value function - bootstrapping. There are two types of TD:

- TD Prediction - there is a constant policy π, experiences to update the $V_\pi(s)$, which represents the expected future discounted reward when the agent starts in state s and follows π. Tries to see how good a policy is by searching the optimal value function $V_\pi(s)$ for that policy;
- TD Control - The policy is not defined. The objective is to learn the optimal policy through experience, which maximizes the expected total reward.

A very well known algorithm of TD-Learning is TD(0). That is is the simplest form of TD-Learning and it is mainly known as one step TD, a prediction algorithm and model-based RL. TD(0) must calculate the next state for each possible action in the current state, and the value function must be updated after each step with the value of the next state, as well as the obtained reward along the way. This reward is the factor that keeps the learning grounded, and the algorithm converges after a sufficient number of episodes. The main objective is to obtain the optimal value function when following a known policy, as the name indicates, "one step TD", TD(0) only needs to wait until the next time step to update the value function $V(S_t)$. TD(0) can be written as

$$V(S_t) = V(S_t) + \alpha * [R_{t+1} + \gamma * V(S_{t+1}) - V(S_t)]; \quad (4)$$

$$V(S_t) = (1 - \alpha) * V(S_t) + \alpha * [R_{t+1} + \gamma * V(S_{t+1})]. \quad (5)$$

where, $V(S_t)$ is the value of the existing state value, and $R_{t+1} + \gamma * V(S_{t+1}) - V(S_t)$ is the new state value or more commonly known TD-Target. It is the received reward when transiting to a new state, plus the value of the next state times the discount rate factor $\gamma\epsilon[0, 1]$, which is used to control the importance given to future rewards. The value of the current state at time t is weighted with $1 - \alpha$ of the previously stored one and α times the TD-target. The learning rate, $\alpha\ \epsilon[0, 1]$ and states the importance of new information over the information that was already learned.

Algorithm 1 Tabular TD(0) for estimating v_π

```
Initialize policy π to be evaluated;
Initialize Learning rate α ϵ [0, 1];
Initialize V(s) for all s ϵ S+, arbitrarily, with V(terminal) = 0;
For each episode
    Initialize s;
    For each step of episode
        s ← action given by π for s;
        Take action a, observe reward, r and next state s′;
        V(s) ← V(s) + α[r + (s′) − V(s)];
        s ← s′;
        until s is terminal;
    end for
end for
```

This information is presented on the basis of [1].

D. Implementation of Reinforcement Learning in DTNs

In 2005, Henkel et al. [3] had a scenario where in a DTN, there are *N* task nodes distributed on a survey area. These task nodes need to forward data to a monitoring station. Each of them has a buffer that generates data flow, messages to be sent in the network. For that, a mobile helper node is introduced, a ferry, that travels at a constant velocity and has a large buffer. It gathers the messages from the nodes to the hub in order to retrieve the collected messages. The goal of the ferry corresponds to choose routes that minimize the average packet delay through the network. The main idea is to exploit cycles that visit a subset of nodes before returning to the hub for data delivery. The ferry learns by observing a model of the system, and updates its flight plan. To explore this problem, an MDP is formulated and the RL algorithm is applied, TD(0). In order to achieve its goal, the ferry visits the task nodes following an optimal sequence, the policy. The ferry relies on observing the nodes buffer state and learns through the received reward when returning to the hub to traffic deliver. The reward is given by:

$$r(s, a) = -\int_0^{ta} (Ft + N_0)e^{-\beta t}\, dt, \quad (6)$$

where, s is the state, a is the action, t_a is the duration of the action, F is the sum of all flow rates from each node, N_0 is the number of packets in all nodes at the beginning of the action and β is a decay for the reward to compensate nodes that are further away. The agent has an incentive to visit nodes which have accumulated a large buffer of undelivered traffic.

However, it also tries to penalize flying to a far-away node more than visiting a near-by node.

Each state has a value that represents the benefit of being in that state. This value is the expected value of the sum of the discounted future rewards obtained when following a fixed strategy and starting in that state. The value function maps states to state values, once the optimal value function is found, the optimal policy will be easily extracted.

$$\pi(s) = arg\max_{\pi}(r(s,a) + e^{-\beta t_a} V_t(s_{t+1})). \quad (7)$$

where, t_a is the duration of an action and s_{t+1} is the following state of s. Following this, the optimal value function is:

$$V_{t+1}(s) = V_t(s) + \alpha(r(s,a) + e^{-\beta t_a} V_t(s') - V_t(s)). \quad (8)$$

where α is the learning rate. In situations which is applied the action selected by policy π on state s, the resulting state is, s'. In order to explore rather than exploit, the agent will sometimes choose a random action with probability ϵ as explained above in Section II-B.

In [3] it was tested an approach using four algorithms, RR, TSP, Stochastic model and RL. In order to compare the performance of the algorithms, the Relative Gain (RG) was defined, as the percent reduction in Average Delay (AD) compared to the worst performing algorithm. AD is reduced when learning agents were introduced. The computational complexity of the problem rises exponentially with the increase of network nodes. Finding a route for 3 nodes takes 1 hour, 6 hours for 4 nodes, and for 5 nodes takes 20 hours. The state representation comprises all combinations of traffic in each node, and the action space are all the combinations of all possible paths from going from one node to all of the others. The fact that it is a hub to node scenario is very restrictive for real-life situations.

E. Deep Reinforcement Learning (DRL)

Adding Artificial Neural Networks (ANNs) to RL, introduces a new concept, DRL. The main difference between RL vs. DRL, represented in Figure 2, is that instead of having a Q-value table for each state-action pair with the optimal Q-value for a single action, there is a ANN that will train the state-action pair and compute the Q-value for each possible action for that state. So the algorithm doesn't need to run the network for every action, this will increase speed and performance.

ANNs produces a great job in approximating functions. In this case, instead of computing Q-values and finding the optimal Q-function, an approximation function is used to estimate it. A DNN is a neural network with more than one layer, which brings complexity. Q-learning explained earlier is a learning method. Combining this two concepts brings a new learning process DQL. This approximation function is obtained via DNN and is called DQN. Figure 3 illustrates the combination of a Deep Neural Network and Reinforcement Learning algorithms.

1) Deep Q-Learning: There is an arbitrary DNN that, as an input, accepts states from a given environment. For each state, the network computes estimated Q-values for each action that can be taken from that state. This networks goal is to approximate the optimal Q-function, that will satisfy the

Fig. 2. Illustrated Difference between Q-Learning and Deep Q-learning.

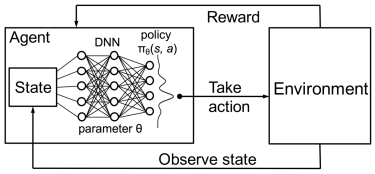


Fig. 3. Combination of Neural Networks with Reinforcement Learning [**?**]

Bellman equation presented in Equation 3. The loss of the network is calculated by comparing the outputted Q-values to the targeted Q-values from the right hand side of the Bellman equation. The objective corresponds to minimize this loss. After the loss calculation, the network weights are updated via SGD. This update is performed iteratively until the loss is minimal and an approximate optimal Q-function is found. With the DQN, it is possible the usage of the Bellman equation to estimate the Q-values with the goal to find the optimal Q-function. The layers of a DQN, are just input layers, followed by non-linear activation function, just like a normal neural network. The output layer is a fully connected layer, that produces the Q-value for each action that can be taken from the given state passed as an input. During the training process of a DQN, a technique called experience replay is used, where the agent's experiences at each time step are stored in a data set - replay memory. At temporal instant t, the agent's experience e_t is defined in a tuple: $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. This produces an insight of the agent's experiences at temporal instant t, containing the state of the environment s_t, the action a_t taken from this state, the reward r_{t+1} given to the agent at time $t+1$ as a result of the previous (s, a) pair, and the next state of the environment s_{t+1}. So, the replay memory stores the agent's experiences at each time step over all episodes played by the agent. The stored experiences will be used to sample batches for network training. Experience replay is the act of gaining experience and sampling from replay memory. The key reason to use replay memory instead of providing the network sequential experiences as they occur in the environment is to break correlation between consecutive samples, leading to inefficient learning. In the beginning, the replay memory is initialized with capacity N in order to hold N experiences. The weights in the network are initialized. For each episode it is also initialized the starting state. In order to gain experience, the agent either exploits and chooses a greedy action (highest Q-value) or explores, and selects a random action. The chosen action is executed, reward and next state are observed and this experience of the agent is stored in replay memory. After being stored, random experiences are sampled from relay memory, a state is extracted from those samples and passed to the network as input. The process occurs as explained earlier in this topic, the model outputs a Q-value for each possible action and the loss is calculated with the Bellman equation. From the

Bellman equation, we have this term:

$$\max_{a'} q_*(s', a'). \tag{9}$$

where s' and a' are the state and action of the following time step. So, s' is passed to the policy network that will output Q-values for each (s, a) pair using s' as the state and all the possible actions as a', it is then possible to obtain the max Q-value. Now this term is calculated for the original state input passed to the policy network.

$$E[R_{t+1} + \gamma \max_{a'} q_*(s', a')] \tag{10}$$

The loss is able to be computed between the Q-value given by the policy network for the original experience (s, a) pair tuple and the target optimal Q-value for this same (s, a) pair. In order to update the weights in the network, SGD is performed with the propose of loss minimization, bringing the policy network output Q-values more closer to the target Q-values given by the Bellman equation. This process is done iteratively for each time step, until the end of the episode.

The following pseudo-code represents what was explained above:

Algorithm 2 Deep Q-Learning with Experience Replay

```
Initialize network Q;
Initialize target network Q';
Initialize experience replay memory D;
Initialize the Agent to interact with the Environment;
    For each episode
      /*Sample phase
      ε ← setting new epsilon with ε − decay
      Choose an action a from state s using ε − greedy(Q) policy
     Agent takes action a, observe reward r, and next state s'
     Store transition (s, a, r, s', done) in the experience replay memory D;
       If enough experiences in D
         /* Learn phase
          Sample a random minibatch of N transitions from D
          For every transition (s_i, a_i, r_i, s'_i), done_i in minibatch
                If done_i
                  y_i = r_i
                Else
                  y_i = r_i + γ * max'_{a ϵ A} Q'(s'_i.a')
                end if
          end for
          Calculate the loss L = 1/N Σ_{i=0}^{N-1} (Q(s'_i, a_i) − y_i)^2
          Update Q using SGD algorithm by minimizing the loss
          Every C steps, copy weights from Q to Q'
        end if
    end for
```

F. Open AI Gym

Open AI Gym is a tool/environment used for developing and testing learning algorithms. It has a library for *Python* language, and it allows the agent to interact in the environment. Gym has a set of custom environments that can be used, a class named *Env* implements the simulator where the environment and the agent will be trained. For environment description we need the *observation_space* and *action_state*, the first one defines the structure and the values for the observation of the environment's state, and the other defines the actions that can be applied to that state in the environment. For the agent-environment iteration, there are a set of functions in the class *Env*: reset and step. The first one resets the environment to its initial state. The observation space corresponds to the initial state. The second one takes an action as input and applies it to the environment, that will transit to a new state. This function returns observation (observation of the new state), a reward (the reward the agent will receive for executing such action), done (if the episode is terminated, in order to reset the environment) and info (information about the environment).

In this simulation, the environment is designed *NodesEnv*. This environment is a map with nodes and a UAV, that is the agent, that will send messages to every node in the map. The *action_space* is the number of Nodes that the UAV can visit and the *observation_space* is the number of messages each node and the UAV have to send in its buffer.

III. Proposed Algorithms

The recreation of [2], only for the All-to-All scenario, using TD(0) with linear approximation, was explained in the previous chapter. The implementation of this algorithm is detailed in this chapter. The second approach is the DRL algorithm that will be further explained and the decisions made to optimize are highlighted.

After this brief presentation of the two approaches, it is proceeded the explanation of the algorithms performed to train the agent in the environment.

A. Implementation of RL

As explained before, a solution equal to the one presented in [2], was implemented. This consists in the implementation of TD(0) algorithm.

The algorithm TD(0) is mainly known as one-step TD. TD(0) is the simplest form of Temporal Difference Learning. At the current state, it must calculate the next state for each possible action and update the value function with the value of the next state, as well as the reward. In practical terms, in order to evaluate each possible next state, the drone will visit each state multiple times. The reward will assign better values to states that have a smaller number for the sum of the messages in all nodes, and will also value smaller routes taken by the drone.

The reward will be equal to the one Henkel et.al [3], used, Equation 6. It is calculated using an integral done over ta, that is the duration of the action N_0, is the sum of the messages from all nodes in the beginning of the action. The negative integral will give better state-values to states which have less messages in all nodes.

The policy will be used to choose the action in a certain state, Equation 7. The value function is Equation 8 and updates the state-value each time the state is visited. The algorithm proposed by [2] is presented with the following pseudo-code, Algorithm 3. To extract the final result, it is implemented in the pseudo-code of Algorithm 4 . The state and action space will be initialize for N episodes, that will be the number of times the code will be running in order to update the value function. At this stage, every node has its buffer empty (vector with the messages to send for each node). Each episode will run for i iterations. When the cycle is finished, will proceed for the next episode, the buffers will be emptied, and the algorithm will start in the initial state but the value function is already filled with the calculated state-values. The author of [2] concluded

that this process of repeating states would be useful, instead of going further without having exploited that state enough times. This will also help if the algorithm gets stuck with a set of odd states, giving it the opportunity to start from the beginning again. As presented in Section II-B, to get the balance between exploiting and exploring, the epsilon greedy algorithm will be used. The exploration rate ε is set to 1 and decaying in each episode, this will be the probability that the agent will exploit. A random number is chosen between 0 and 1. If this random number is smaller than the exploration rate, the agent will explore. If the opposite occurs - random number bigger than ε, the agent will exploit.

In each i iteration, an action will be chosen:

- randomly, with probability of ε; (Exploration)
- according to the policy (Equation 7), with probability of $1 - \varepsilon$; (Exploitation)

Algorithm 3 RL-implementation

```
For N episodes
    Initialize State;
    Initialize Action Space;
    If current episode % (N/5) == 0
        /* This means every 20% of the episodes
        α = α ∗ 0.5;
        ε = ε ∗ 0.5;
    end if
    For i iterations
        Choose a random number 0 ≤ rnd ≤ 1;
        If rnd < ε
            Select an action randomly;
        Else
            Select the best action according to the Policy;
        end if
        new_state ← Generate state transition according to
        the action selected and current state ;
        Update state-value function accordingly;
        State ← new_state
        end for
end for
```

The policy will choose the action with the best state value. TD(0) needs to be able to calculate all the possibly next states when choosing that action, a state-value table is used in this algorithm. To calculate the policy, it is necessary to know the next state and its associated value. The parameters, α and ϵ are decayed 20% in every iteration. This is an important aspect, since the author of the previous work, tried several approaches, but concluded this one was the best option, so much for α and ϵ parameters. He opted for the one in Equation 11, because in the first iterations the decay was slower but would later stabilize to a lower value in the last iterations.

$$\alpha = \alpha * rate \tag{11}$$

When the action is chosen it will simulate in the environment, this way it is possible to know every possible next state. The value state is updated and the next iteration begins in the new state returned by the environment.

In algorithm 4, after iterating through the N episodes, the state-value table is already filled with values that will indicate which states are better to be at. It is now easy to extract the best sequence of actions, starting in the initial state and choosing iteratively according to policy, the best 20 actions to choose. The author of [2], chose 20 actions in the learning phase (algorithm 3). A sequence can be extracted by searching for a repetitive pattern in the results of taken actions.

Algorithm 4 RL extracting final result

```
Initialize State;
Initialize Action Space;
    For i iterations
        Choose best action according to the Policy;
        new_state ← transition(old_state, action);
        results.append(action)
        old_state ← new_state
    end for
```

Since, the first work of [2] is done for a different scenario than the one focused in this thesis, first he does a Hub-to-Node scenario, where each node can only send messages to the hub, the state space is smaller than the All-to-All scenario, where every node can receive and send to/from each other node, an adaptation to the value function is done. A linear approximation is used. In this algorithm TD(0) is used a linear approximation that can be performed using a vector of weight. This vector will help to estimate the value of each state. In this case, instead of trying to learn a value for $V(s)$, the value learned is given by $V(s, w)$, where w is a vector of weights.

$$V(s, w) = \sum_{i=0}^{n\_features} w_i * feature_i \tag{12}$$

The value function $V(s)$, instead of iterating until reaching optimal values $V^*(s)$, will iterate with the goal of finding the optimal weight vector. The optimal weight vector is the one that best represents the value of each state according to the features that represent it and are updated after each step:

$$w_i = w_i + \alpha * (r(s, a) + \gamma * V(s', w) - V(s, w)) * feature_i \tag{13}$$

B. Deep Reinforcement Learning Implementation

This implementations aims to find an approach that will possible achieve more efficient results than the previous (RL implementation) and studied related work. The use of ANN instead of a linear approximation was proposed by the author of [2] and this will be the aim, to introduce ANN and RL combined, that are commonly know as DRL.

In order to use a Deep Reinforcement approach, the chosen algorithm is Deep Q-Learning with Experience Replay. The theory for this part was explained in Section II-E1 a pseudo-code for this algorithm is Algorithm 2.

The unique change in this algorithm corresponds to the introduction of ANN and a Replay Memory (where the agent's experiences at each time step are stored). The calculation of the reward (Equation 6), the policy (Equation 7), and value function (Equation 8) used in the RL implementation will be maintained. Another change is instead of using the Linear Approximation function (Equation 12) and the vector of weights, this will not be necessary, because ANN (which is a non linear function) allows for a more complex environment without having the need to do a linear approximation.

In this algorithm it is necessary a training network and a target network(ANNs). The second network is not trained and it is a way to see if the model was well trained, by using the outputs of the training network in a not trained target network. In each episode, to train this environment and learn the optimal

path to deliver the messages, first it needs to fill the experience replay. Using the states as an input, the agent (UAV) will start to choose actions by exploring the environment (network) and getting rewards according to the chosen action. This reward will be calculated by Equation 6 as mentioned above. The tuple $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ that is the experience replay is filled, with this process.

After this, the agent will exploit, in the current state is select an action with probability $1 - \varepsilon$ or an action predicted by the training model. The training model predicts Q-values for each possible action from that state, the calculation of the Q-values is computed with the value function from the RL implementation. The agent will choose the best action according to policy Equation 7. In the RL algorithm, the reward granted better values to the states that had a smaller number for the sum of all the messages in the nodes and also to smaller routes taken by the UAV. This pattern maintains is this DRL approach. In this phase what was stored in the experience replay is not being trained, it is only being used to compute Q-values.

To train the network, a random batch of the experience replay is sampled in order to predict Q-values. Each experience (tuple with state, action, reward and next state) represents the actual state s_t, the reward r_{t+1} the agent got by choosing the action a_t, and next state s_{t+1}. The Q-values predicted are used to calculate the Loss. The chosen states used to predict the Q-values are the input for the target network, that is not trained. With this, the targeted network will predict the best rewarded action to take from each state. This would be the expected action that the trained network should have chosen (when was trained for the first time, with the data in the experience replay).

It is possible to calculate the Loss, with the Bellman Equation:

$$Q(S, A) = R + \gamma \max_i [Q(S', a_i] \tag{14}$$

Where $Q(S', a)$ is the $Q-value$ for the state where the action A' was chosen by the the target network model, and $Q(S, a)$ is the $Q - value$ for the state where action A was chosen in the training network. The expected $Q - value$ is given by the expected reward (R) and the discount factor ($gamma$), it is an hyperparameter that can be regulated in order to give more or less weight to the target $Q - value$. After this, the SGD algorithm is performed to minimize the loss – each iteration and episode trying to approximate the $Q - value$ ($Q(S', A)$) to the expected $Q - value$ ($Q(S, A)$).

To create the Artificial Neural Network that will receive the states as an input, were created layers – creates a kernel with the input layer (states) to produce an output. This will be followed by one fully-connected hidden layer and one fully-connected linear output that computes the Q-values for each action from the input state.

In this algorithm a set of hyperparameteres are configured in order to maximize the training of the model and adapt the model to the data being trained. Such as in the RL implementation, the learning rate, the ε for the greedy algorithm, the batch size and the discount factor need to be defined. The learning rate, speeds the process of learning in the beginning and will help the model to converge. The variable ε starts in a value close to 1 and will decay its value in order for the environment to explore in the beginning of the simulation and to exploit during the rest of the iterations. The batch size of the stored experiences can't be too small or too big. For last, the discount factor ($gamma$) used to calculate the policy that will choose the best action according to the $Q-Value$ of such action is used to define the weight given to the calculated reward in comparison to the computed Q-values.

IV. Evaluation and Comparison

For this work, the environment consists in N nodes that represent the nodes that can generate and receive messages and one UAV. In practice the UAV, is considered as a node that will start in a random position of the network.

This agent has N actions to choose in the first iteration in which the UAV is in a random position. It will after travel to the first node the algorithm chooses. After this, there are only $N - 1$ available actions, because in the next step it can only travel to the rest of the nodes except the initial position of the UAV. For this algorithm, the key factor correspond to the number of messages each node has to send to each other node, and the different generation rates of messages, we used buffers to store this information: N buffers to store the messages that each node has to send to each other node, one buffer of the UAV, that has the sum of messages each node has to receive in that step and N buffers with the generation rate of the messages of each node to the other nodes in the network. In this scenario, where every node can send messages to every other node in the network, there is a need to have into account the number of messages in the buffers of each node, and in the UAV. A linear adaptation is done to follow the needs of a larger state space. Instead of finding the optimal value of the value function, the algorithm will iterate in order to find the optimal weight vector. In this work, the algorithm uses the same approach as [2]. There is a vector called $feature$, which contains the number of messages of each node to another, followed by the number of messages in the UAV buffer and one-hot encoding of the node the UAV is at.

To test this algorithms, the number of episodes, the max steps per episode and the hyperparameters, such has: learning rate (α); exploration rate (ε); decay rate for the reward (β); discount factor (γ); velocity for the UAV, v, need to be defined.

TABLE I
Parameters used in the RL and DRL simulation

Parameters	Initial Value RL	Initial Value DRL
Number of Episodes	10000	10000
Maximum Steps	200	50000
Batch Size	-	200
ε - Exploration Rate	0.3	0.3
α - Learning Rate	0.3	0.01
β - Decay Rate	0.01	0.01
γ - Discount Factor	$e^{-\beta t}$	$e^{-\beta t}$
v - Velocity	20	20

In this work, four different networks were simulated. For two of them, with 4 nodes and the other two with 6 nodes, each of these networks has an UAV that initiates the simulation

in a random position in the network. Figure 4 represents the simulated networks. The first two networks (Fig. 4 a) and b)) are small networks have smaller dimensions then the second two (Fig. 4 c) and d)).

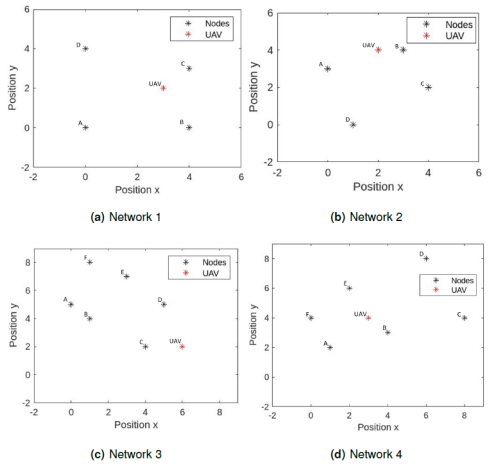


Fig. 4. Simulated networks representation for RL and DRL implementations

In a network there are nodes that have higher generation rates from each node to another.In network 1, nodes C and D generate more messages, in network 2 node A and B generate more messages, in network 3 node A generates less messages for the other nodes and in network 4 node B generates less messages. In order to be a fair and reasonable comparison, the same four networks (Figure 4) for RL and DRL approaches were used, such as the same generations rates for both algorithms.

A. Performance Metrics

To evaluate the performance of the RL algorithm, AD will be used as a metric. The AD consists in the time it took for the UAV to deliver the messages to the destination node since the moment they were generated by the sending node. The message needs to be successfully delivered. There were always encountered cases in the simulations where the UAV buffer (vector used in the simulations to store the number of messages for each node) still had data, which means that not every generated message was delivered in the end of the simulation. These messages are not considered for the AD.

In order to compare with the algorithm that motivated this work (RL implementation done in [2]), the RG is used. This performance metric is computed using the difference of the AD results in [2] and the AD results of this work.

$$RG = \frac{baseline - target}{baseline} * 100 \tag{15}$$

where $baseline$ is the AD obtained in [2] and $target$ is the AD computed in this work.

To conclude which algorithm had the best performance it is measured in terms of optimal path computation for the UAV to take in order to visit every node in the network and delivering the messages. A comparison to evaluate both of the implementations of this work is computed in terms of AD, execution time and memory usage.

B. Results of the RL implementation

For the RL Algorithm, implemented with TD(0) and linear approximation to the optimal weight value computation, it was verified that the increase of the network size increases the execution time of the algorithm and the time to send a message from a node to another increases as well.

For the path identification, the actions that the agent took in the environment were analyzed, and when a set of the nodes being visited was repeated several times in the running of the simulation, is considered the path taken by the agent. In the following tables is present the AD, the execution time, memory used and the path taken, for each network: Throughout

TABLE II
RESULTS IN TERMS OF AVERAGE MEMORY USAGE[MB] AND EXECUTION TIME[S]

Network	Average Memory Usage[MB]	Execution time[s]
1	277	25200
2	289	23400
3	328	40520
4	376	493900

TABLE III
RESULTS IN TERMS OF AVERAGE DELAY[T.U.] AND PATH TAKEN

Network	Average Delay[t.u.]	Path taken
1	86,8	C,B,A,B,C,D
2	75,6	B,C,B,A,D,C
3	109,7	C,D,E,F,E,D,C,B,A,F
4	203,4	B,A,F,E,B,C,B,D,E,F,A,B,C,D

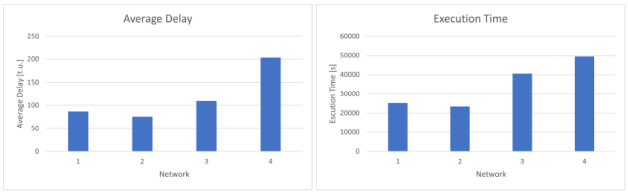
the simulation on all networks it is noticed that the chosen actions are being chosen, because in the reward calculation it is given more importance to smaller routes over the number of messages to be sent The main difference in the networks corresponds to the distance between the nodes, when it is smaller makes the AD compared to bigger networks smaller as well. The 3rd and 4th networks, are different than the two previous. Instead of having 4 nodes, they have 6 nodes and the dimensions of the network are larger, in network 4 the arrangement of the nodes is more dispersed. So, the execution time will be higher as it is confirmed, is almost the double of the execution time of the previous networks.

Following this, it was considered for comparison the results obtained in [2]. In table IV the results are shown:

TABLE IV
RESULTS FOR RL IMPLEMENTATION IN [2]

Network	Average Delay[t.u.]	Execution time[s]
Network 1	79,9	822
Network 2	67,1	1451
Network 3	84,0	1445
Network 4	127,6	2012

Analyzing the results presented in Table IV with the ones in Tables II and III, it is clearly possible to understand that the algorithm performed in this work had a superior execution time, it used more memory and the AD is bigger, which means that this algorithm was slower and more memory consumer. Besides this, it managed to find paths to deliver the messages



(a) Average delay for each network (b) Execution time for each network

Fig. 5. Comparison of the average delay for each network as well as the comparison of the execution time of the simulation for each network

without leaving any node unvisited. The AD showed that this algorithm takes a longer time to deliver a message to its destination node. This can be due to the fact that the network nodes are organized in a more dispersed way than the ones used in [2] or because the UAV starts in a random position and is not considered a regular node. The first action in the environment is to choose where the path should start.

To do the comparison between the developed algorithm with the one studied in [2], the RG is calculated with Equation 15 but instead of subtracting the AD of the results obtained in [2] by the AD of the results of this work the opposite is done. Saraiva et. al, [2] obtained better results for the AD, it is clear it will have a higher RG compared with the implementation performed in this work. For network 1 RG is 7,95%, network 2 RG is 11,24%, network 3 RG is 23,42% and network 4 RG is 37,27%.

It is seen in Figure 5 the comparison of the networks in terms of AD and execution time of the algorithms. As mentioned above, it was already concluded that with larger and more dispersed networks the AD tends to grow as well, because the UAV needs to travel longer distances between nodes, the messages will wait more time to be delivered on the opposite of a network where the nodes are more closer to each other. The execution time increases with the number of nodes, and the dimensions of the network.

C. Results of the DRL implementation

For the DRL implementation with experience replay the same networks and generation rates is maintained (Figure 4). The hyperparameters for this algorithm are defined in Table I.

Both of the networks, training network and target network, are Neural Networks that are trained to find the optimal $Q - Value$ for some possible action from a state that is the input on the networks. The ANN allows for more complex scenarios but also brings more complexity to solve this problem of the optimal path. In this simulation of DRL using experience replay more obstacles were presented with a not so good expected result.

In Figure 6 it is possible to observe in the axis-y the value of the rewards and in the axis-x the training episodes. It is observed that several times the value of the reward is 0. This happens because the reward is calculated over the integral of 0 to the time of completing an action, Equation 6. Sometimes, this algorithm wrongly considers that the UAV is in a position of the network where is considering that the time of completing an action is infinite ($t = \infty$), $e^{-\beta t}$ will be equal to zero. The policy that is used to choose the action corresponding to the one that minimizes the Loss between the Q-values computed in the training network and the Q-values computed in the target network is calculated with Equation 7. The calculated policy for values of $t = \infty$ will also be equal to zero or equal to a value that the model will approximate to zero. This value will lead to the algorithm always choosing the same action. Per example, a network with $N = 5$ has $N - 1$ possible actions when the UAV is at a certain node, Action = 0 : chooses node A; Action = 1: chooses node B; Action = 2: chooses node C; Action = 3: chooses node D. The value of the policy will choose the action, so for the case where the policy is zero, it will always chose Node A, without taking into account if it is the closest one or the buffer of messages of node A has less or more messages then the buffer of the other nodes. This will increase the AD of the network, leading to a higher value than the ones computed in the RL implementation. Also, the path does not go to every node in the network.

In this evaluation it is not considered the execution time [s] of the algorithm since it grows to much considerable values caused by the problem of the $t = \infty$ and will not be very well representative of the performed simulations.

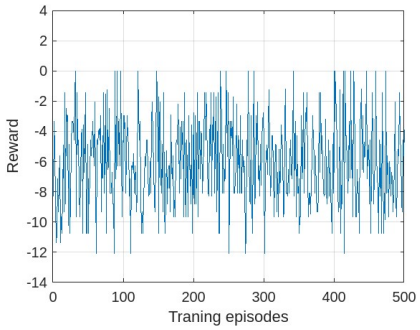


Fig. 6. Evolution of the reward in a training simulation of the DRL algorithm.

TABLE V
RESULTS FOR DRL IMPLEMENTATION FOR THE 4 NETWORKS ANALYZED

Network	Average Delay[t.u.]	Average Memory Usage[MB]
Network 1	306,4	< 456
Network 2	300,2	< 395
Network 3	408,7	< 541
Network 4	432,6	< 575

In table V it is possible to observe that the AD is higher than the ones observed in the RL Implementation (Table III) and the ones in [2]. This happens because of the problem explained above, the nodes in the network will keep generating messages for the other nodes. These generated messages will be waiting a long time to be delivered. The average memory used for this simulations is also higher compared to the other solutions because this algorithm is more complex due to the introduction of the neural network.

Even though the behavior of this algorithm was not the expected one the AD followed the same pattern as in the RL

implementation, it increases with the size/dimensions of the network and with the number of nodes.

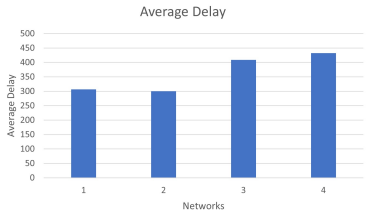


Fig. 7. Average delay for each network in the Deep Q-Learning with experience replay algorithm

In order to compare the algoritms, the RG was calculated. According to Equation 15 the baseline is the AD of the RL implementation and target is the AD of DRL implementation. For network 1 RG is 71,7%, network 2 RG is 74,8%, network 3 RG is 73,1% and network 4 RG is 52,9%. With these values of RG it is possible to conclude that the first algorithm had much better results than the later one. This was mainly because of the problems enunciated above.

V. Conclusion

The work proposal was to find the optimal path that the UAV should take in a set of nodes that generate messages to each other. Two approaches were developed and performed: using a RL algorithm and the other one using a DRL algorithm.

The first one consists of TD(0) with a linear approximation and it was a simulation based on [2]. The algorithm results were compared with the ones in [2]. In every simulated network of this implementation, the AD was more elevated, this means that the previous algorithm, on which this algorithm was based, had a better RG in relation to the one performed in this paper. This implementation was less well performed.

DRL is introduced: the combination of Neural Networks and RL. Instead of having a Q-table with the Q-values for each state-action pair, a Neural Network is used, where the model is trained. For the Deep Q-Learning algorithm with experience replay, the results were not as successful as expected. Besides using a non-linear approximation for the calculation of the Q-values that is a favorable point in relation to the RL implementation, the training of the model in this work, was not executed perfectly. This resulted in several times where the time it took for an action to be completed being a very large value of [t.u.]. This way the policy was not well calculated and the model chooses the same action several times, not visiting every network node.

As a result of this problem, the AD of this approach was even worse than the ones obtained in the RL algorithm, when the opposite was predicted to happen.

Despite the advantage of using a non-linear approach in this type of formulated problems, where the state space easily grows to a more complex and larger state space, the improvement in relation of the use of a linear approximation function was not obtained. This happened because of a poorly trained model in the DRL approach.

Nevertheless, in this work it was achieved to simulate the RL implementation and able to observe the results computed previously, and draw the same conclusions. Even though the DRL implementation did not manage to perform as well as it was expected it was a good starting point in introducing Neural Networks and a way to understand the differences and similarities between these two different types of learning.

A. Future Work

In order to find a better approach for this problem, the use of Policy Gradient, Actor Critic Algorithms, or the combination of both (Policy Gradient - Actor Critic algorithm) can be explored. The Policy Gradient - Actor Critic algorithm is a policy based algorithm, that learns the policy function relying on experience. This algorithm uses two neural networks, one to learn the policy and the other one to learn the value function. In this case, the policy and the value function don't need to be the same as those used in previous work. These functions will be learned with the neural networks - environment interaction.

This type of algorithms can handle bigger actions spaces, that grow over the course of the simulations, and converges faster than Deep-Q learning with experience replay.

If the main goal is to correct the Deep Q-learning with experience replay, the used hyperparameters to train the algorithm can be varied and more studied. The used neural networks, training and target networks can have another hidden layer, resulting in more complexity and larger state space.

References

[1] Sutton, Richard S and Barto, Andrew G, "Reinforcement learning: An introduction", MIT Press, 2018, ISBN 9780262039246.

[2] Saraiva, M.C., "Optimization of message ferrying UAV paths using monte carlo tree search and reinforcement learning", Master's thesis, Instituto Superior Técnico, May 2019.

[3] Henkel, Daniel and Brown, Timothy X, "Towards autonomous data ferry route design through reinforcement learning" in 2008 International Symposium on a World of Wireless, Mobile and Multimedia Networks, IEEE, 2008 pp: 1–6.

[4] Zeng, Yong and Zhang, Rui and Lim, Teng Joon, "Wireless communications with unmanned aerial vehicles: Opportunities and challenges" in IEEE Communications Magazine, IEEE, 2016, pp. 36–42.

[5] Zhao, Wenrui and Ammar, Mostafa and Zegura, Ellen, "Controlling the mobility of multiple data transport ferries in a delay-tolerant network" in Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 2, IEEE, 2005, pp. 1407–1418.

[6] Zhang, Zhen and Fei, Zongming, "Route design for multiple ferries in delay tolerant networks" in 2007 IEEE Wireless Communications and Networking Conference, IEEE, 2007, pp. 3460–3465.

[7] Guo, Hui and Li, Jiang and Qian, Yi, "Modeling and evaluation of homing-pigeon based delay tolerant networks with periodic scheduling" in 2009 IEEE International Conference on Communications, IEEE, 2009, pp. 1–5.

[8] Barroca, Cintia and Grilo, António and Pereira, Paulo Rogério, "Improving message delivery in UAV-based delay tolerant networks" in 2018 16th International Conference on Intelligent Transportation Systems Telecommunications (ITST), IEEE, 2018, pp. 1–7.

[9] Li, Kai and Ni, Wei and Tovar, Eduardo and Jamalipour, Abbas, "On-board deep Q-network for UAV-assisted online power transfer and data collection" in IEEE Transactions on Vehicular Technology, vol. 68, number 12, IEEE, 2019, pp. 12215–12226.

[10] Smith, Phillip and Hunjet, Robert and Aleti, Aldeida and Barca, Jan Carlo, "Adaptive data transfer methods via policy evolution for UAV swarms" in 2017 27th international telecommunication networks and applications conference (ITNAC), IEEE, 2017, pp. 1–8.