Autonomous Networking a.y. 22-23

Homework 2: Report

Ilaria De Sio - desio.2064970@studenti.uniroma1.it
Paolo Pio Bevilacqua - bevilacqua.2002288@studenti.uniroma1.it
Chiara Ballanti - ballanti.1844613@studenti.uniroma1.it

February 1, 2025

1 Introduction

In this homework, we addressed a drone routing problem using the Deep Reinforcement Learning protocol described in [**?**].
We have a set of N drones $\{d_0, ..., d_{N-1}\}$ deployed in Area of Interest (AoI). Each drone is different from the others, it follows a specific path and captures events. Events generate packets that have to be sent to the depot, which is a fixed sink D deployed in the AoI. Every packet expires after 1500 time-steps (or 225 seconds). We can't modify the trajectory of each drone, so we have to implement an algorithm that allows multi-hop drone-to-depot (node-to-sink) communication. Our algorithm aims to maximize the number of packets delivered to the depot on time while minimizing delivery time. Once the packet is generated, the drone can decide to keep the packet in its buffer or send the packet to one of its neighbors.
In particular, we have to implement a state-of-the-art algorithm to intelligently guide the drones in a choice.

2 Learning Model

In [?], the authors consider a routing optimization problem for data transmission from a source to a destination in a mobile robotic network consisting of multiple robotic devices.

State definition The main element is a vector used to characterize the link between a node u_i and its neighbor node b_j:

$$C_{u_i,b_j} = (ct_{u_i,b_j}, PER_{u_i,b_j}, e_{b_j}, d_{b_j,des}, d_{min}) \tag{1}$$

where the components are:

- ct_{u_i,b_j} represents the expected connection time of the link, which is the time duration until the distance between node u_i and its neighbor b_j reaches to communication radius.
- PER_{u_i,b_j} is the packet error rate of the link.
- e_{b_j} represents the remaining energy of neighbor b_j.
- $d_{b_j,des}$ is the distance between neighbor b_j and the destination des (the depot).
- d_{min} is a minimum distance between a two hop neighbor b_k and des.

We assume that at a time step t, node u_i with m neighbors will make a routing decision. To capture the changes of link status, we define these links formed by node u_i and its all neighbors as representing a discrete state s.
The state $s_t \in S$ is defined by a vector $(C_{u_i,b_1}, \dots, C_{u_i,b_j}, \dots, C_{u_i,b_m})$.

It is important to normalize each vector of the state to the range of $[0,\ 1]$:

$$C_{u_i,b_j} = \left(\frac{ct_{u_i,b_j}}{CT_{max}},\ PER_{u_i,b_j},\ \frac{e_{b_j}}{E_{init}},\ min\left\{ \frac{d_{b_j,des}}{d_{u_i,des}},\ 1 \right\},\ min\left\{ \frac{d_{min}}{d_{u_i,des}},\ 1 \right\} \right) \tag{2}$$

where:

- CT_{max} is the maximum expected connection time.
- E_{init} is the initial energy of the neighbor b_j.

Action In the routing process, the u_i node that holds a packet must decide to keep the packet or move it on to select a neighbor as the next hop to forward the packet. At a time step t, we define an action $a_t \in A$, where $A = \{b_1, b_2, \dots, b_m\}$ is the set of neighbors. Action denotes itself (keeping the packet) or the selected neighbor, which means node u_i chooses the neighbor to forward the packet.

Reward Function To achieve rapid and reliable data transmission, a new concept of reliable distance is introduced in the reward function, which is based on not only the distance but also link quality. The reliable distance D_{u_i,b_j} is calculated as:

$$D_{u_i,b_j} = \frac{d_{u_i,des}}{d_{b_j,des}} * (1 - PER_{u_i,b_j}) * \beta \qquad \text{with } \beta = \begin{cases} 1 & ct_{u_i,b_j} \geq ct_{min} \\ 0 & ct_{u_i,b_j} < ct_{min} \end{cases} \tag{3}$$

If the expected duration time ct_{u_i,b_j} is less than a minimum time ct_{min}, a data packet cannot be successfully transmitted from node u_i to its neighbor b_j. The reason is that during the process of data transmission, the link is disconnected because the distance between the node u_i and its neighbor b_j is greater than the communication range. So, in this case, β is set to 0 to reduce the probability of the link selected to forward packets.
The reward function is defined as:

$$r_t = \begin{cases} R_{max} & \text{when neighbor } b_j \text{ is the destination} \\ -R_{max} & \text{when neighbor } b_j \text{ is the local minimum} \\ \omega\ D_{u_i,b_j} + (1-\omega)\left(\frac{e_{b_j}}{E_{b_j}}\right) & \text{otherwise} \end{cases} \tag{4}$$

Where R_{max} is the maximum reward value and e_{b_j} is the remaining energy and E_{b_j} is the initial energy of neighbor b_j. ω $(0 < \omega < 1)$ is used to adjust the importance of reliable distance D_{u_i,b_j} in reward function. If the selected neighbor b_j is the destination, the maximum reward R_{max} is given, while the negative reward is given to avoid the void area when neighbor b_j is a local minimum (all neighbors of node u_i are further away from the destination than node u_i), in other cases, we consider reliable distance D_{u_i,b_j} and the energy level of neighbor as the reward value. In [1], the authors used a simple ReLU but trying to improve performances, we use the Leaky ReLU, which modifies the function to allow small negative values when the input is less than zero.

Deep Q-Network (DQN) A drone that holds a packet determines its state and selects an action using DQN to decide the next-hop node. The Q-value related to each neighbor b_j is calculated by using the DQN with the normalized state as the input. After the Q-values of all neighbors are obtained, the neighbor with the maximum Q-value is selected as the next-hop node to forward the packet.
The DQN was built as back propagation neural network with three layers, and Leaky ReLU ($negative\_slope = 0.4$) as activation function.

The architecture of the DQN used in the homework is shown below:

$$
\begin{aligned}
&DQN(\\
&\quad (layer_1) : Linear(in\_features = n\_observations * 5,\ out\_features = 128,\ bias = True) \\
&\quad (layer_2) : Linear(in\_features = 128,\ out\_features = 128,\ bias = True) \\
&\quad (layer_3) : Linear(in\_features = 128,\ out\_features = n\_actions,\ bias = True) \\
&\quad)
\end{aligned}
\tag{5}
$$

where *n_observations* and *n_actions* are equal to the number of drones in the simulation.

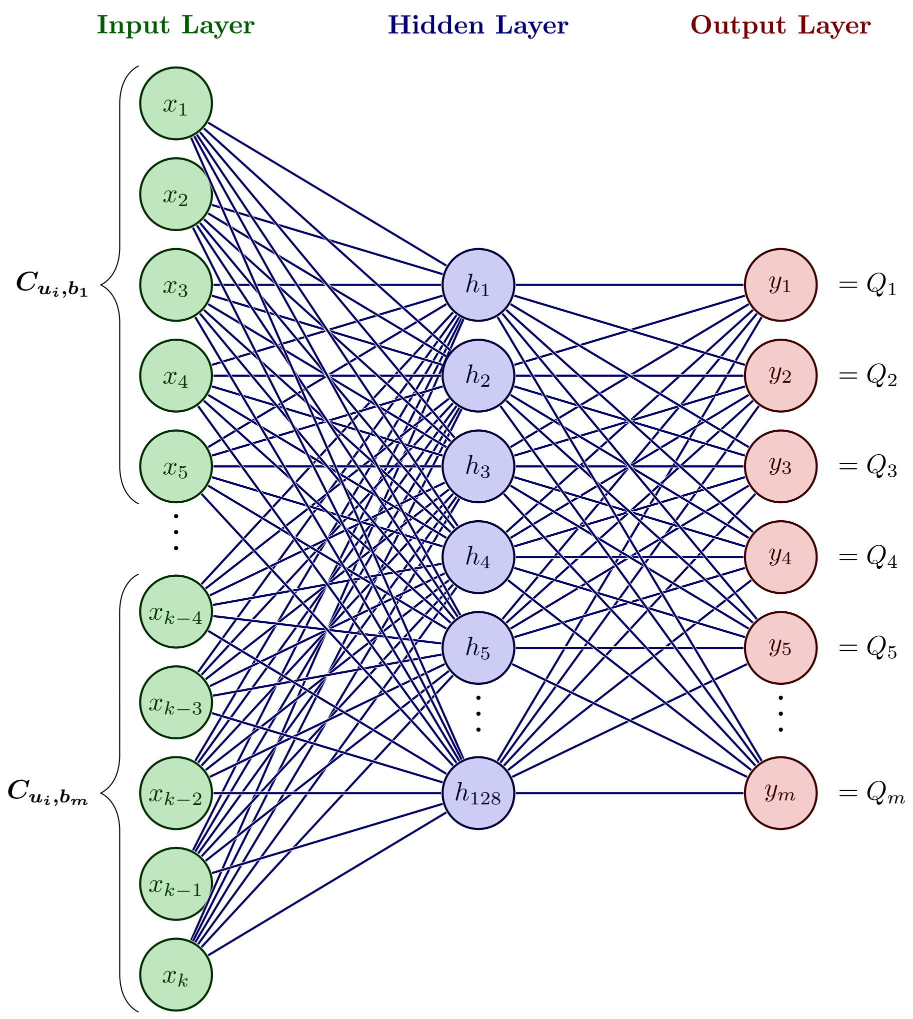


Figure 1: Architecture of the DQN (where $k = n\_observations * 5$, $m = n\_actions$ and Q_j is the Q-value related to the neighbor b_j).

2.1 ARdeep implementation in the simulator

In this section, we describe how we implemented the state-of-the-art algorithm according to the already existing simulator.

Relay Selection The *relay_selection*() function is called by the simulator every time an agent (a drone) has to choose an action to select a relay for packets.
We chose the epsilon-greedy strategy in order to decide if the agent (the drone) should explore or exploit:

$$A_t \leftarrow \begin{cases} \arg\max_a \ Policy\_net(s_t, a) & p = 1 - \epsilon \quad \text{(exploitation)} \\ Geo(\{a_1, ..., a_k\}) & p = \epsilon \quad \text{(exploration)} \end{cases} \tag{6}$$

We have applied a slight modification in the case of exploration. We decided to select a relay for packets according to geographic routing using C2S criteria. In fact, by selecting a random action (Uniform($\{a_1, ..., a_k\}$)), the probability of moving the packet to a local minimum increases. Consequently, the probability of receiving -1 as a reward also increases.

Note that the dimension and the shape of the state are fixed but a drone u_i rarely will have all the other drones as neighbors. Hence, we decided to use a mask to filter the real neighbors of u_i and we apply it to the input of the DQN (the state) and the output of the DQN.

In order to record the performed actions we use a data structure called *taken_actions*, described as follows:

$$taken\_actions = \{id\_event : (current\_state,\ action,\ next\_state)\} \tag{7}$$

Note that *next_state* in *relay_selection* is set to $None$. It will be calculated later, after the change of position of the drones.

Feedback The $feedback()$ function is called by the simulator to give a reward to the drones when a packet expires ($outcome = -1$) or arrives to the depot ($outcome = 1$).
We applied a slight modification to the reward function (4), adapting it to the value of the *outcome*. Our reward function is defined as:

$$r_t = \begin{cases} R_{max} & \text{when } outcome = 1 \text{ and the neighbor } b_j \text{ is the destination} \\ x & \text{when } outcome = 1 \text{ and the neighbor } b_j \text{ is not the destination} \\ -R_{max} & \text{when } outcome = -1 \text{ and the neighbor } b_j \text{ is a local minimum} \\ -(1-x) & \text{when } outcome = -1 \text{ and the neighbor } b_j \text{ is not a local minimum} \end{cases} \tag{8}$$

$$\text{where} \ \ x = \omega \ D_{u_i, b_j} + (1-\omega)\left(\frac{e_{b_j}}{E_{b_j}}\right) \tag{9}$$

According to the original reward function (4), we should have given a positive reward in the case of an expired packet, but we considered it an inaccuracy.
Note that $x \in [0,\ 1]$ in our reward function (8), so we have a continuous distribution of the reward:

$$reward \in [-1,\ 1] \tag{10}$$

Once the reward is calculated, we have all the elements to create a new observation (sample) and store it in the $Experience\_Replay\_Memory$.
A sample is described as follows:

$$sample = (s_t,\ a_t,\ s_{t+1},\ r_t) \tag{11}$$

Training The $optimize\_model()$ function is called by the simulator to train the DQN $Policy\_net$. At each time step, a sufficiently large training batch of random samples from the experience replay memory is selected as input for both networks.
We used the $current\_state$ s_t and the $next\_state$ s_{t+1} from the sample as input to the $Policy\_net$ and the $Target\_net$ respectively for all actions.
From the output Q values of the $Policy\_net$, we select the one for the sample action to obtain the $state\_action\_values$ (the predicted Q values). From the outputs of the $Target\_net$, we select the one with the maximum value to obtain the $next\_state\_values$. Now we can calculate the Target Q values (expected Q values) as follows:

$$expected\_state\_action\_values = (next\_state\_values * \gamma) + reward \tag{12}$$

Once the predicted and the expected Q values are calculated, they are used to compute the loss, back-propagate it and update the weights of the $Policy\_net$ using gradient descent.

The $Policy\_net$ weights have been updated, while the $Target\_net$ is not trained so no loss is computed, and back-propagation is not done. This allows the $Policy\_net$ to learn to predict more accurate Q values, while the Target Q values remain fixed for a while, so we are not chasing a moving target. After $update\_weights\_delta$ time steps, copy the $Policy\_net$ weights to the $Target\_net$. This lets the $Target\_net$ get the improved weights so that it can also predict more accurate Q values. In this way, the $Policy\_net$ weights and the $Target\_net$ are again equal.

2.2 Simulator changes

This section describes the changes we introduced in the simulator to implement the state-of-the-art protocol.

State implementation In order to represent the vector (1) we applied the following modifications:

- **Expected connection time of the link**: since we don't know the drone's path and in the simulator there is no network layer we don't know how long two drones will continue to communicate. Not being able to calculate this information at runtime, we adopted a different solution by running the simulator twice.
  The first run is used to collect connection time data. Meanwhile, the second one is the simulation campaign aimed at gathering all the data necessary to evaluate the performance of the algorithm. To record the connection time of all the links we use a data structure called *conn_time_history*. At each step for each drone, we open a connection based on the neighbors present at that time, and in the next step, we check if they are still present, otherwise, we close the connection. For each drone pair, we save a matrix in *conn_time_history* containing the intervals during which the two drones remained in communication.
  conn_time_history is described as follows:

$$
\begin{aligned}
&conn\_time\_history = \{\\
&Drone_0 : \{Drone_0 : [\ ], Drone_1 : [[ts_{start_1}, ts_{end_1}],\ \dots\ , [ts_{start_x}, ts_{end_x}]],\ \dots\ , Drone_n : [[...],\ \dots\ , [...]]\},\\
&Drone_1 : \{Drone_0 : [[ts_{start_1}, ts_{end_1}],\ \dots\ , [ts_{start_x}, ts_{end_x}]], Drone_1 : [\ ],\ \dots\ , Drone_n : [[...],\ \dots\ , [...]]\},\\
&\quad \vdots\\
&Drone_n : \{Drone_0 : [[...],\ \dots\ , [...]], Drone_1 : [[...],\ \dots\ , [...]],\ \dots\ , Drone_n : [\ ]\}\\
&\}
\end{aligned}
$$

- **Remaining energy of the neighbor**: we introduced an attribute for each drone that represents its residual energy. At the beginning of the run, we decided the minimum number of times m in which a drone must discharge. The initial value of the energy is set to $simulation\_duration/m$. During the run, the residual energy of a drone is decreased every time it moves or it sends any kind of message. When the drone runs out of energy or when the drone arrives at the depot we reset it.

3 Experiments

In this section, we discuss the simulation settings and the obtained results.

3.1 Setup

We explain our implementation choices on the following values:

- $\varepsilon = \varepsilon_{end} + (\varepsilon_{start} - \varepsilon_{end}) * e^{-curr\_step/\varepsilon_{decay}}$
  where $\varepsilon_{start} = 0.9,\ \varepsilon_{end} = 0.05,\ \varepsilon_{decay} = 1000$.
  The agent (the drone) explores with a time-varying probability. At the beginning of the simulation, ε will be a very high value, equal to ε_{start}. While the simulation goes on, ε will decrease its value and the agent will exploit more frequently. At the end of the simulation, ε will reach ε_{end}.

- $\omega = 0.3$
  According to the reward function (8) an high value of ω will give more importance to the reliable distance D_{u_i,b_j} (3) but it will end up in a very small range of rewards, around $(0, 0.2] \cup (-1, -0.8] \cup \{1\} \cup \{-1\}$.
  For this reason, we decided to give more importance to the residual energy of the neighbor e_{b_j} and have a larger range of rewards (10).

- $calculate\_next\_state\_delta = 10$
  The time between the calculation of the state s_t and the next state s_{t+1}.

- PER_{u_i,b_j}: it is set to a random value in the range $[0, 0.2]$.

Training We used *Weights&Biases* [**?**] in the training phase, an experiment tracking tool for ML. It is a powerful platform for building better models and offers several functionalities. Among them, we chose experiment tracking and hyperparameter tuning. At first, we logged the loss, the cumulative sum and the mean of the rewards on a few simulator runs to keep track of the impact of our setup values on performances. To automate hyperparameter search and explore the space of possible models we created a *W&B Sweep*. We used it directly in the simulator run cycle since the taken actions could change according to the update of the *Policy_net*.
The sweep configuration that we used is:

- *metric* = minimize the loss

- *batch_size* = 64, 128, 256, 512

- *learning_rate* = [*lr_min*, *lr_max*]
  where $lr\_max = 1e{-5}$ and the $lr\_min = lr\_max/(1 + sim\_duration/lr\_dec\_speed)$.
  lr_dec_speed allows us to choose from which time step the learning rate decreases.

- *method* (search strategy) = *bayes*
  The bayesian hyperparameter search method uses a Gaussian Process to model the relationship

between the parameters and the model metric and chooses parameters to optimize the probability of improvement.

To collect metrics during the simulations we set the following training values:

- $\gamma = 0.9$
- $update\_weights\_delta = 500$
  The frequency with which the weights of the $Policy\_net$ are copied into the $Target\_net$.

The following values are computed from a $W\&B$ $Sweep$ run with the lowest loss (0.24) :

- $learning\_rate = 0.00004690593429019178$
- $batch\_size = 128$

3.2 Results

In the plots, we show the results obtained by our algorithm compared to geographic routing, random routing and our hw1 q-learning solution.

We evaluated our model using the following metrics:

- packet mean delivery ratio;
- packet mean delivery time;
- mean number of relays

The model aims to maximize the number of packets delivered to the depot on time while minimizing delivery time.

We show the results obtained by our approach on 50k time steps simulation (training on the first 25k time step) to understand the behavior of the model. We tested on 10, 15, 20, 25 and 30 drones. We could not conduct experiments on more than 30 drones due to the high computation and time costs not available to us.
For the same reason, we could not compute the standard deviation for the three metrics to see performance variability across seeds. The standard deviation of the number of possible relays and of the packet delivery time is calculated on the collected data of a single run.
The plots of the packet mean delivery ratio, the packet mean delivery time and the mean number of relays are shown respectively in figure 2, 3 and 4.

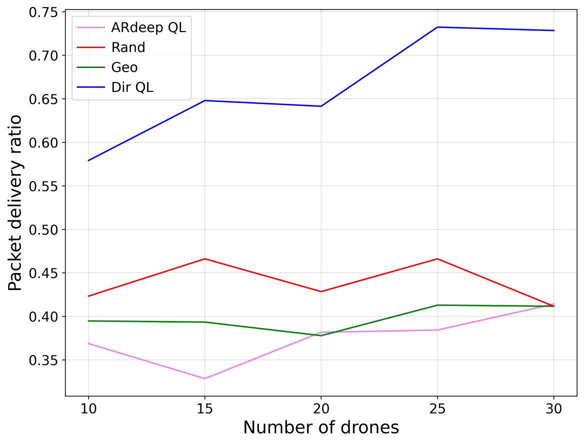
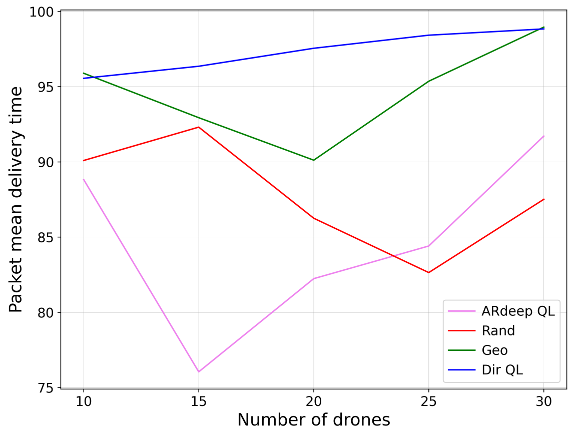
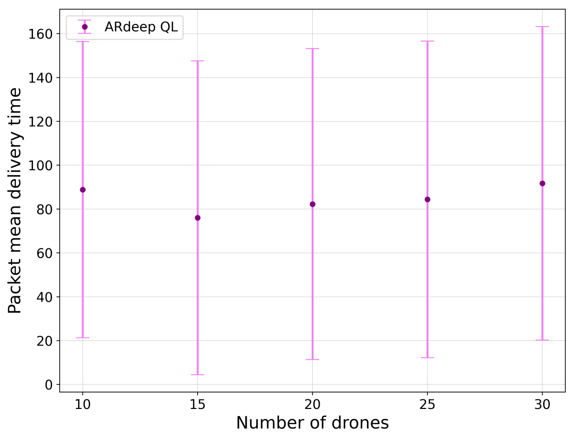


Figure 2: Plot on 50k simulation duration for the packet delivery ratio.

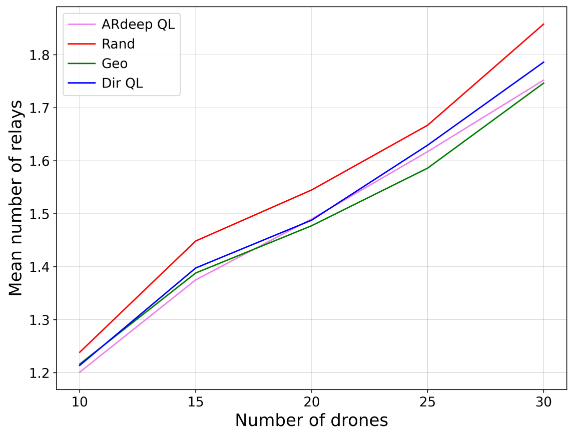


(a) Packet mean delivery time.

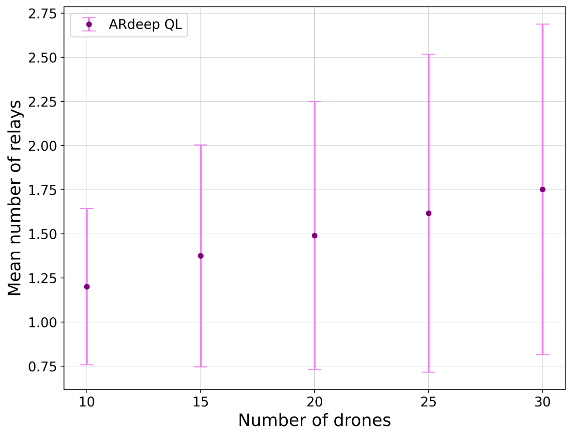


(b) Packet mean delivery time with standard deviation.

Figure 3: Plots on 50k simulation duration for the packet mean delivery time.



(a) Mean number of relays.



(b) Mean number of relays with standard deviation.

Figure 4: Plots on 50k simulation duration for the mean number of relays.

Training The following plot shows the trend of the loss for each time step of the training.

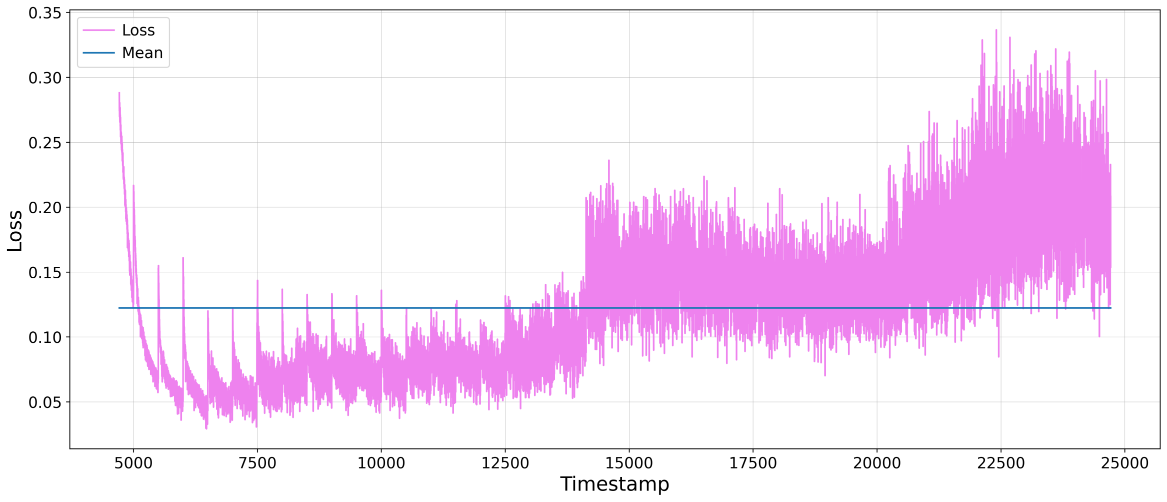


Figure 5: Plot on 25k time step training for the loss.

Reward ARdeep q-learning The following plots show the trend of the sum of the rewards and the distribution of them among actions in a run with 10 drones and 50k time steps.

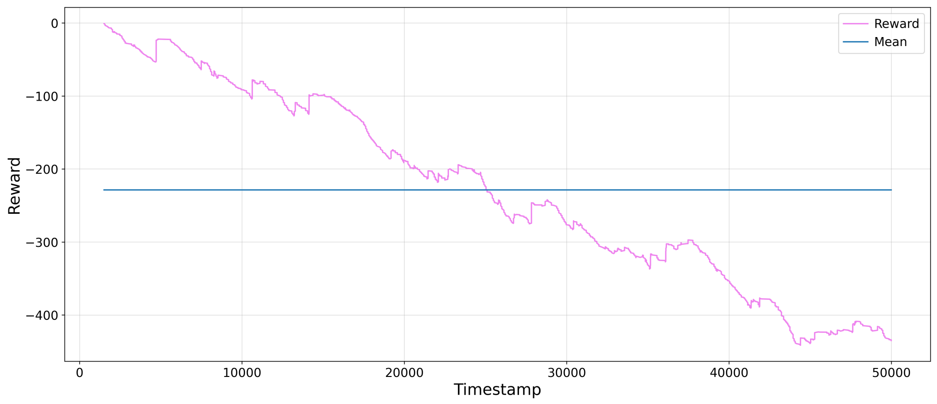


Figure 6: Sum of the rewards at each time step.

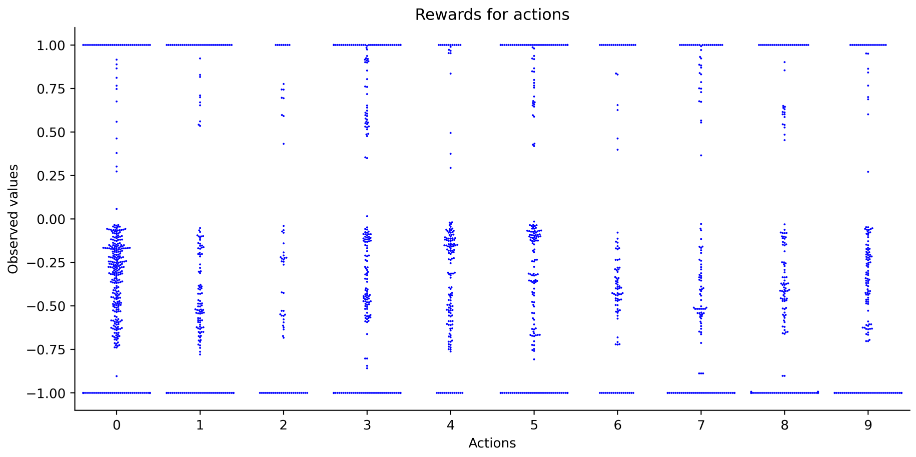
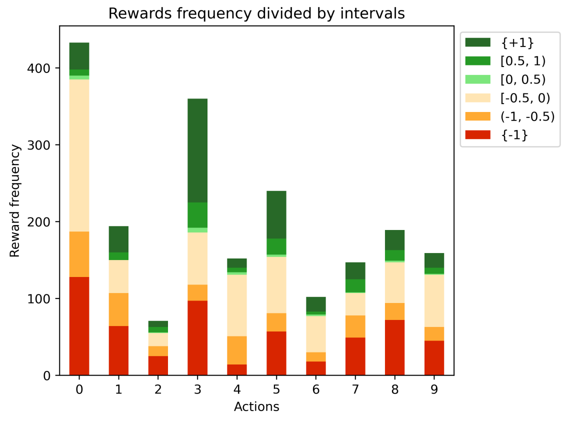
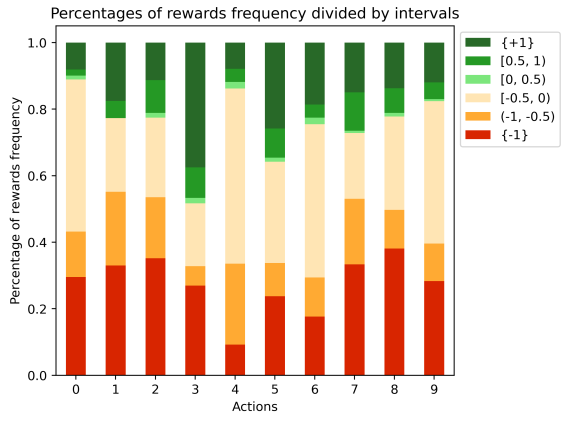


Figure 7: Swarm plot of the rewards given to actions.



(a) Frequency reward given to actions grouped by intervals.



(b) Frequency reward given to actions grouped by intervals (normalized groups to 100%)

Figure 8: Plots on 50k simulation duration for the rewards.

4 Conclusions

As shown in section 3, the ARdeep algorithm has very low performance. It underperforms all the other algorithms tested.
We have noticed that with more drones, performance increases, but not enough. This happens because there is a higher probability that one of the drones, following its mission, will enter the communication range of the depot.
We are aware that our model does not learn and we assume that the problem is related to the training. Looking at the loss (5) it does not converge to zero. We can see that it decreases correctly for the first part of the training, but it starts to rise soon afterward.
As we can see in figures 7 and 8, low performances are reflected in a prevalent distribution of negative rewards compared to positive ones. For this reason, the trend of the sum of the rewards (figure 6) converges to a negative value.

In order to implement the ARdeep algorithm we had to adapt it to our simulation. Changing the reward function was fundamental because the authors do not handle the case where packets expire. Their environment and the features of the agents differ from ours, consequently it causes different results. In [?], the drone speed is variable between 10m/s and 50m/s, the simulation space is set to 1km x 1km and the drone's communication range is set to 300m, while in our simulation are set respectively at 8m/s, 1.5 km x 1.5 km and 150m. Their setup increases the probability to reach the destination early. Our adjustments may have impacted our performances, along with the model that does not learn.

Contributions

The algorithm's code has been developed collaboratively with scheduled meetings on the Google Meet platform and using a GitHub repository and PyCharm's "Code with me" plugin. We discussed all the problems and possible solutions together. The same approach has been used to write the report.

Ilaria De Sio

Chiara Ballanti

Paolo Pio Bevilacqua

References