UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN INGENIERÍA DE
TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

ANALYSIS AND IMPLEMENTATION OF MULTIAGENT DEEP REINFORCEMENT LEARNING ALGORITHMS FOR NATURAL DISASTER MONITORING WITH SWARMS OF DRONES

DAVID BALDAZO ESCRIÑA

2019

TRABAJO FIN DE MÁSTER

TÍTULO (Español): Análisis e implementación de algoritmos de aprendizaje por refuerzo profundo multiagente para la monitorización de desastres naturales mediante enjambres de drones

TITLE (English): Analysis and Implementation of Multiagent Deep Reinforcement Learning Algorithms for Natural Disaster Monitoring with Swarms of Drones

AUTOR: D. David Baldazo Escriña

TUTOR: D. Juan Parras Moral

DEPARTAMENTO: Departamento de Señales, Sistemas y Radiocomunicaciones

TRIBUNAL:

Presidente:

Vocal:

Secretario:

Suplente:

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN INGENIERÍA DE
TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

ANALYSIS AND IMPLEMENTATION OF MULTIAGENT DEEP REINFORCEMENT LEARNING ALGORITHMS FOR NATURAL DISASTER MONITORING WITH SWARMS OF DRONES

DAVID BALDAZO ESCRIÑA

2019

Summary

Natural disasters are the cause of a great amount of deaths and economic loss every year. The rapid and efficient deployment of mitigation and relief work is of great value but can be hampered by the lack of information in real time about the situation. Technological progress has allowed for a gradual improvement of this process of information retrieval. Automatic detection of wildfires and floods from images is now a reality. However, the real-time update of natural disaster maps still requires the physical presence of sensors.

A natural disaster surveillance strategy that is gaining traction in recent years uses swarms of drones to acquire images of the environment. The cost reduction of this architecture depends on the automation of swarm cooperative navigation.

The goal of this Master's thesis is to build upon the recent advancements in the use of Deep Reinforcement Learning techniques for swarm navigation to improve them and apply them to a wider variety of situations.

In particular, we have developed a platform for the simulation and training of swarms of drones and a series of multiagent algorithms. Lastly, we have trained deep neural networks with and without memory in simulations of both wildfires and floods, following several different strategies with varying levels of training decentralization, and we have studied the potential benefits of each approach.

The final practical outcome is a series of trained neural networks which can be loaded onboard the drones of the swarm to control their navigation in a decentralized manner.

Keywords

Swarms, drones, machine learning, reinforcement learning, optimal control, navigation, surveillance, natural disasters, deep learning, distributed, multiagent, recurrent neural networks.

Resumen

Los desastres naturales son la causa de gran cantidad de muertes y pérdidas económicas cada año. La puesta en marcha rápida y eficiente de los trabajos de mitigación y ayuda es de gran valor pero puede verse obstaculizada por la ausencia de información en directo de la situación. El avance tecnológico ha permitido una gradual mejora de este proceso de adquisición de información. La detección automática de incendios forestales e inundaciones a partir de imágenes es ya una realidad. Sin embargo, la actualización en directo de mapas de desastres naturales sigue requiriendo la presencia física de sensores.

Una estrategia de monitorización de desastres naturales que está afianzándose en los últimos años usa enjambres de drones para adquirir imágenes del entorno. La reducción del coste de esta arquitectura depende de la automatización de la navegación cooperativa de enjambres.

El objetivo de este trabajo fin de Máster es construir sobre la base de los recientes avances en el uso de técnicas de aprendizaje por refuerzo profundo en navegación de enjambres de drones para mejorarlas y aplicarlas a una gama más amplia de situaciones.

En concreto, se desarrollado una plataforma de simulación y entrenamiento de enjambres y una serie de algoritmos multiagente. Finalmente, se han entrenado redes neuronales profundas con y sin memoria tanto en simulaciones de incendios como de inundaciones, siguiendo estrategias con distintos niveles de descentralización del entrenamiento y se han estudiado los beneficios potenciales de cada planteamiento.

El resultado práctico final es una serie de redes neuronales entrenadas que pueden colocarse a bordo de los drones del enjambre para controlar su navegación de forma descentralizada.

Palabras clave

Enjambres, drones, aprendizaje automático, aprendizaje por refuerzo, control óptimo, navegación, monitorización, desastres naturales, aprendizaje profundo, distribuido, multiagente, redes neuronales recurrentes.

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

Adam	Adaptive Moment Estimation	11, 36
AI	Artificial Intelligence	47
BPTT	Backpropagation Through Time	15, 16, 33
CNN	Convolutional Neural Network	13
Dec-POMDP	Decentralized POMDP	8
DL	Deep Learning	11, 17
DNN	Deep Neural Network	11, 13, 14, 32, 38
DQN	Deep Q-Networks	11, 14, 15, 17, 31–33, 35–40
DRL	Deep Reinforcement Learning	1, 2, 11, 13–15, 17, 18, 40, 41, 47
DRQN	Deep Recurrent Q-Networks	15, 18, 31, 33, 38, 39, 41
LIDAR	Light Detection And Ranging	1
LSTM	Long Short-Term Memory	14, 15, 33, 39

MDP	Markov Decision Process	5, 6, 9, 10, 12, 39
ML	Machine Learning	47, 48
POMDP	Partially Observable MDP	7, 15, 16, 39
ReLU	Rectified Linear Unit	13, 32
RL	Reinforcement Learning	9–11, 14, 25, 47
RNN	Recurrent Neural Network	2, 14, 15, 33
SLAM	Simultaneous Localization and Mapping	14, 38, 40, 41
UAV	Unmanned Aerial Vehicle	1, 19, 26, 40, 47, 48

1

Introduction and objectives

1.1 Motivation

From 2007 to 2016 floods and wildfires caused an estimated 367 US$ billion and 26 US$ billion of economic losses respectively worldwide [1]. In 2017, floods led to more deaths than any other type of natural disaster worldwide and wildfires burnt 250 thousand hectares of land in Spain [2].

Relief work in natural disasters is dangerous for the people involved and leads to additional deaths. In the particular case of floods and wildfires, the availability of nearly real-time inundation or fire maps can make a difference when planning relief work because they can help assess the current situation, forecast its evolution and inform the decision-making process so that the resources are optimized and lives are saved.

To provide these maps, different systems have used both satellite and aerial images. For floods, both radar and optical images have been employed [3]. For wildfires, sensors such as Light Detection And Ranging (LIDAR), infrared and visible cameras have been used [4].

In particular, the performance of aerial optical sensing of floods has been proven by works such as [5], which shows accurate detection of floods from images captured by Unmanned Aerial Vehicles (UAVs), and in the case of wildfires [4] presents a monitoring system with teams of UAVs.

The evolution of the techniques for natural disaster monitoring shows a trend towards automation, eliminating the need for humans and reducing operational costs. To this effect, if we were to automate the control of surveillance drones and make it distributed, we could envision systems in which swarms of UAVs automatically fly towards the points of interest, coordinate their actions and provide real-time maps without any human intervention. Julian et al. [6] do precisely this in the case of wildfires: it shows that the techniques of Deep Reinforcement Learning (DRL) can be used to train decentralized controllers for swarms of UAVs. This realization motivates us to test this same approach in other types of natural

disasters and to assess the possible advantages of more advanced DRL techniques.

1.2 Objectives

This Master's thesis has a twofold objective. First, to extend the recent advances in decentralized controllers for swarms of drones for wildfire monitoring to other types of natural disasters, specifically to flood monitoring. Second, to test the suitability of Recurrent Neural Networks (RNNs), neural networks with memory, to solve the same problems and to assess whether this is a better approach to coordinated navigation and monitoring than the one used by [6], which makes use of an explicit representation of the map.

1.3 Contributions

To this end, a simulation environment for wildfires and floods has been integrated with implementations of DRL algorithms, which have been extended to be compatible with multiagent environments.

Specifically, the most notable outcomes of the project have been the following:

- The development and tuning of a multiagent version of the DQN algorithm.
- The successful replication of the results in [6] with our own environment, to be used as a reference.
- The development of a gym compatible flood surveillance environment with realistic random terrain generation and flood simulation.
- The successful application of the multiagent DQN algorithm to solve the flood surveillance problem.
- The development and tuning of a multiagent version of the DRQN algorithm.
- A comparison of the performance of both algorithms and an assessment of the benefits of memory in navigation and natural disaster surveillance.

A part of the outcomes of this Master's thesis has been accepted for publication in the 27th European Signal Processing Conference (EUSIPCO) as an article titled Decentralized Multi-Agent Deep Reinforcement Learning in Swarms of Drones for Flood Monitoring [7].

1.4 Structure of this document

The remainder of this document is structured in seven chapters:

- Chapter 2 introduces the required theoretical background.

- Chapter 3 presents the hardware and software requirements for the project.
- Chapter 4 describes in detail the problem to solve.
- Chapter 5 explains the simulation environments.
- Chapter 6 details the implemented algorithms.
- Chapter 7 presents the simulations and their results.
- Chapter 8 lies down the conclusions that have been drawn from the project and proposes some future lines of work.

2

Theoretical preliminaries

This chapter is dedicated to the definition of the main theoretical concepts on which the project is based. We will begin by explaining the concept of swarm to then define the mathematical model that is required to state the problem. Finally, we will focus on the learning tools that will allow us to solve the problem.

2.1 Swarms

Navarro and Matía propose a concise definition of swarms: "Swarm robotics is a field of multi-robotics in which large number of robots are coordinated in a distributed and decentralised way. It is based on the use of local rules, and simple robots compared to the complexity of the task to achieve, and inspired by social insects." [8]

According to [9], some of the domains of application of swarm systems are tasks that cover a region, tasks that are too dangerous, tasks that scale-up or scale-down in time and tasks that require redundancy. Natural disaster monitoring is in all of these domains at the same time.

The problem of natural disaster monitoring by a swarm of drones can be interpreted as a cooperative game, in which the reward function is the same for all agents, or as a distributed control problem, in which a single reward function has to be optimized but multiple independent agents take actions simultaneously.

In order to solve this control problem, we need a mathematical model that represents the capabilities and limitations of the swarm of drones and its relationship with the natural disaster that is being monitored. We will now present a series of interrelated models from the simplest to the most suitable for the problem.

2.2 Mathematical model for the problem

A suitable mathematical model should tackle several aspects:

- Dynamic environment with mobile agents: the model should take into account the dynamic nature of both the environment outside the agent and the state of the agent itself (position, heading direction), which is considered as a part of the environment.
- Multiagent setting: it should model the presence of an arbitrary number of agents.
- Decentralization: agents should be able to sense the environment and take decisions by themselves.
- Partial observability: the mathematical model should distinguish between the full state of the environment and what is actually observed by each agent.
- Homogeneity of swarm agents and consensus: the model should make all the simplifications that derive from the fact that all agents in a swarm are identical and once trained should behave similarly in identical situations.

We will use two nomenclatures interchangeably for the temporal indexes: s_t or simply s for the current state and s_{t+1} or s' for the next state. The same applies to actions, observations and rewards.

2.2.1. Markov Decision Processes

A Markov Decision Process (MDP) [10] is a simple model of the interaction between a single agent and its environment in a sequential decision making problem. Figure 2.1 shows the components of an Markov Decision Process (MDP) and Figure 2.2 unrolls the decision process for 4 timesteps.

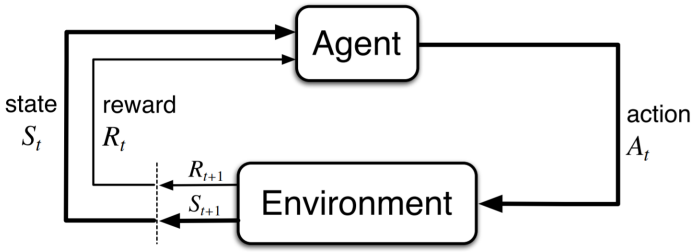


Fig. 2.1: MDP model diagram. [11]

Definition 2.2.1. An MDP is a 5-tuple $\langle S, A, P, R, \gamma \rangle$ where:

- S is the state set, containing all the possible states $s \in S$ of the system.
- A is the action set, containing all the possible action vectors $a \in A$ that the agent can play. In our setting they are discrete.

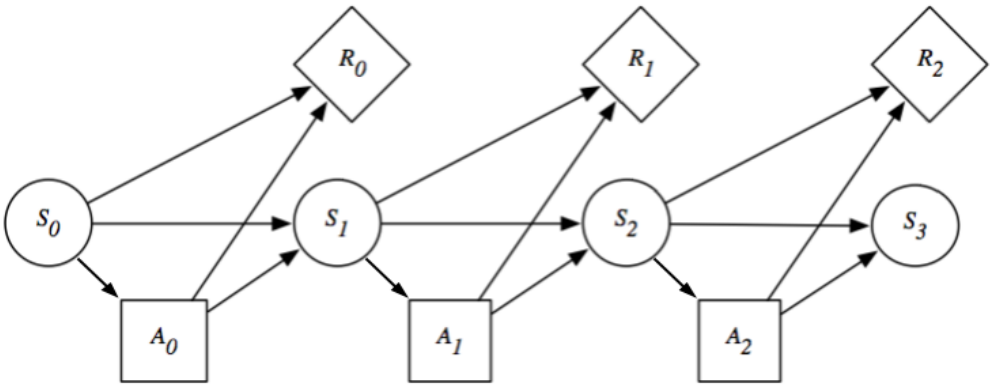


Fig. 2.2: Unrolled MDP model diagram (4 timesteps). [12]

- $P : S \times S \times A \to [0, 1]$ is the transition probability function, where $P(s'|s, a)$ denotes the probability of transitioning to state s' given that the agent is in state s and plays action a.
- $R : S \times A \to \mathbb{R}$ is the reward function, where $r(s, a)$ denotes the reward that the agent receives when it is in state s and plays action a.
- $\gamma \in [0, 1]$ is a discount factor, used to obtain the cumulative reward for the agent.

At each timestep t, the agent observes the current state s, chooses an action a and receives a reward r that depends on the current state and the action. The environment moves to a new state.

An MDP implicitly assumes that the current state provides the same information as the history of states when predicting the future. This is known as the *Markov property*. For this model to have a practical application, the assumption should be made that the state of the environment can be fully sensed at all times by the agent taking the decision.

The solution to an MDP is a policy π, which is formally defined as a mapping from states to probabilities of selecting each possible action, $\pi : S \times A \to [0, 1]$ [11].

2.2.2. Partially Observable MDPs

The MDP model assumes that the agent derives its actions from a perfect knowledge of the state s of the environment. This is not a valid assumption in our problem or in most real-world problems. Each of our drones can only access the image that is captured by its own camera and the limited information that can be shared in real time between drones via radio communication. This means that the action chosen by the drone can only depend on the observation z, which does not necessarily convey all the information required to determine the full state of the environment because it is noisy or incomplete. The Markov property no longer holds when dealing with partial observations, which means that in this case the history of observations might convey more information about the future than the last observation alone.

In a Partially Observable MDP (POMDP) [10] the policy π is not dependent on states but instead on observations, and potentially on their full history. Figure 2.3 shows the components of a POMDP and Definition 2.2.2 defines them.

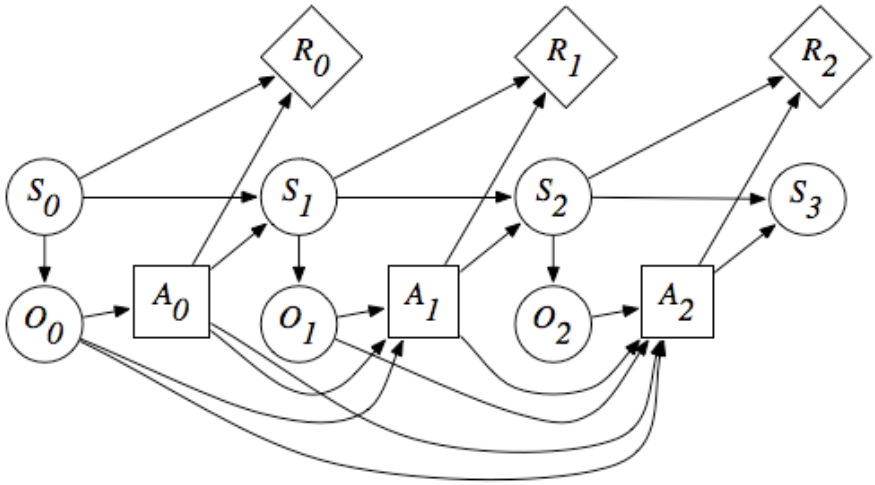


Fig. 2.3: Unrolled POMDP model diagram (4 timesteps). [12]

Definition 2.2.2. A POMDP is an 7-tuple $\langle S, A, Z, P, R, O, \gamma \rangle$ where:

- S is the state set, defined as in the MDP case.
- A is the action set, defined as in the MDP case.
- O is the observation set, containing all possible observations $o \in O$ that the agent observes.
- $P : S \times S \times A \to [0, 1]$ is the transition probability function defined as in the MDP case.
- $R : S \times A \to \mathbb{R}$ is the reward function defined as in the MDP case.
- $Z : S \times O \to [0, 1]$ is the observation model, where $Z(o|s)$ is the probability of observing o given that the environment in state s.
- $\gamma \in [0, 1]$ is a discount factor as in the MDP case.

We define the history of observations as $H_t = O_0 \times O_1 \times ... \times O_t$. A policy π is then defined as a mapping from the history of observations to the probabilities of selecting each possible action, $\pi : H_t \times A \to [0, 1]$.

2.2.3. swarMDPs

Julian et al. [6] use a POMDP as the mathematical model for the swarm wildfire surveillance problem. This model, however, fails to take into account the presence of multiple agents and the fact that these agents are identical to each other. To solve this, we will frame

the problem as a swarMDP [13], which is a particularization of the Decentralized POMDP (Dec-POMDP) model.

Figure 2.4 shows the components of a swarMDP:

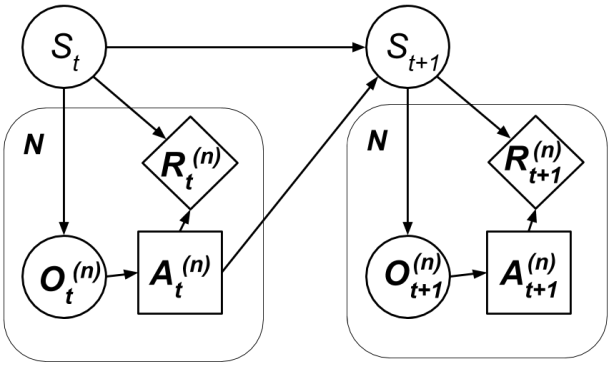


Fig. 2.4: swarMDP model diagram unrolled for two consecutive timesteps.

Definition 2.2.3. A swarMDP is defined in two steps. First, we define a prototype $\mathbb{A} = \langle S, A, O, \pi \rangle$, where $\mathbb{A}$ is an instance of each agent of the swarm and where:

- S is the set of local states.
- A is the set of local actions.
- O is the set of local observations.
- $\pi : O \to A$ is the local policy.

A swarMDP is a 7-tuple $\langle N, \mathbb{A}, P, R, Z, \gamma \rangle$ where:

- N is the index set of the agents, where $i = 1, 2, ..., N$ indexes each of the N agents.
- $\mathbb{A}$ is the agent prototype defined before.
- $P : S \times S \times A \to [0, 1]$ is the transition probability function, where $P(s'|s, a)$ denotes the probability of transitioning to state s' given that the agent is in state s and plays action a. Note that P depends on a, the joint action vector.
- $R : S \times A \to \mathbb{R}$ is the reward function, where $r(s, a)$ denotes the reward that the agent receives when it is in state s and plays action a.
- $Z : S \times O \to [0, 1]$ is the observation model, where $Z(o|s)$ is the probability of observing o given that the agent is in state s.
- $\gamma \in [0, 1]$ is a discount factor, used to obtain the total reward for the agent.

As shown by Definition 2.2.3, by defining an agent prototype we explicitly model the multi-agent setting and the fact that all agents are equal. This model satisfies all of the requirements that we defined at the beginning of Section 2.2.

Team reward

In the swarm context, cooperation between agents has to be promoted. This is done by defining an appropriate reward function R. A strategy that has been found to force cooperation upon agents is the use of a team reward such as the average or the sum of individual rewards [14, 15]. This technique is in harder to implement in practice because it forces each agent to retrieve additional information from all the other agents during the training process, which might pose problems in real time applications.

2.3 Reinforcement Learning

In section 2.2 we have argued that a swarMDP is the best mathematical model for our problem. We will now show how Reinforcement Learning (RL) can solve this problem.

Reinforcement Learning (RL) is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal [11]. RL algorithms offer ways to obtain optimal policies π for problems that can be modeled as we have done in Section 2.2, in which the environment is unknown. They do this by trial and error, making the agent figure out which actions to take in each state of the environment. These actions usually have an effect not only in the current reward but also in future rewards because they affect the environment. Consequently, the objective that the agent has to aim for with each action is to maximize the expected value of G, the return or cumulative reward, which is defined as follows:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^T r_T = \sum_{k=0}^{T} \gamma^k r_{t+k}, \tag{2.1}$$

where T is the final time step and γ is the previously defined discount factor, $0 \leq \gamma \leq 1$, which establishes how far-sighted the agent is.

The state-action value or Q-value, $Q^\pi(s, a)$, of a state-action tuple given the policy π, is the expected value of G, this is, the expected return over all possible trajectories when the agent starts at state s, takes action a and then follows policy π.

$$Q^\pi(s, a) \triangleq \mathbb{E}_{\pi,P}\left[G_t | S_t = s, A_t = a, A_{t+k} \sim \pi\right], \quad k = 1, \ldots, \infty \tag{2.2}$$

In MDPs, this value can be estimated with convergence guarantees by algorithms such as Q-Learning [16]. In this algorithm, this is done by repeatedly updating the estimation with the Bellman equation:

$$Q(s, a) = r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \max_{a' \in A} Q(s', a'). \tag{2.3}$$

For a given Q function, the optimal policy or greedy policy is the following:

$$\pi(s) = \underset{a \in A}{\operatorname{argmax}}\, Q(s, a). \tag{2.4}$$

Q-learning is an off-policy RL algorithm, meaning that it learns the optimal Q-value—the Q-value of the optimal policy—regardless of the policy that is actually executed while training. It can also be classified as a value based RL algorithm because it estimates Q and only obtains the optimal policy indirectly. This is in opposition to other algorithms that work directly in the policy domain, known as policy based algorithms.

During training, the agent faces a trade-off between the exploration of the action-state space through a random policy and the exploitation of the knowledge that has already been acquired through the current estimation of the optimal policy. In practice this is tackled by using an exploratory policy for most of the decisions at the beginning of training and gradually increasing the fraction of exploitation actions. This is known as an ϵ-greedy policy, which can be formally defined as follows:

$$\pi_\epsilon(a|s) \triangleq \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}|, & a = \arg\max_{a' \in \mathcal{A}} Q(s, a') \\ \epsilon/|\mathcal{A}|, & a \neq \arg\max_{a' \in \mathcal{A}} Q(s, a') \end{cases} \tag{2.5}$$

assuming that there is a single greedy action, where $|\mathcal{A}|$ is the cardinality of the action set.

Algorithm 2.1 shows the pseudocode for Q-learning in a MDP.

Algorithm 2.1 Q-learning with ϵ-greedy behaviour policy for episodic MDP.

Input: Parameter ϵ and learning rate sequence α.

Output: π, the approximate optimal policy $\pi^\star$.

Initialize $Q(s, a)$ arbitrarily for all $(s, a) \in \mathcal{S} \times \mathcal{A}$
Initialize $Q(s, \cdot) = 0$ for all terminal states
repeat(for each episode) ▷*Main loop*
    Initialize s
    repeat(for each step in the episode)
        Choose action $a \sim \pi_\epsilon(\cdot|s)$ using ϵ-greedy policy from current Q
        Take action a and observe r, s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s, a)]$
        $s \leftarrow s'$
    until s is terminal
until we cannot run more episodes
for all $s \in \mathcal{S}$ **do** ▷*Get policy*
    $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} Q(s, a)$
end for
return π

2.4 Deep Reinforcement Learning

In practice, the number of unique state-action pairs in a real problem such as the one we are facing is too high to estimate the Q-value of each one individually. Instead, we resort to an approximation. Deep Reinforcement Learning (DRL) is the intersection of RL and Deep Learning (DL). Deep Neural Networks (DNNs), consisting of compound differentiable functions with trainable parameters, are used to approximate either value functions or policies.

2.4.1. Deep Q-Networks

In the case of Q-learning, its DL extension is Deep Q-Networks (DQN) [17, 18]. DQN estimates Q by training a DNN to approximate the function. Training is performed by gradient descent, with the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')}\left[e^2\right], \tag{2.6}$$

where the Bellman error e is:

$$e = r(s,a) + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s,a,\theta). \tag{2.7}$$

DQN tackles the problem of instability inherent to DRL in two ways. Firstly by randomizing the samples used in training through a mechanism known as experience replay, in order to reduce the effect of the correlation of the sequence. This is implemented through a replay memory $\mathbb{E}$. Secondly, by training two networks instead of one: a first network—known as online network, with parameters θ—is updated at the maximum frequency. At the same time but at a lower update rate than the first network, a second network—known as target network, with parameters θ^-—is used to estimate $Q(s', a')$, so that the online network is moving towards a fixed target. A typical optimizer for the update of θ is Adaptive Moment Estimation (Adam) [19]. The pseudocode for the algorithm is detailed in Algorithm 2.2.

2.4.2. Double DQN

Double-DQN [20] (previously Double Q-learning in its tabular form [21]) is a commonly used improvement to DQN. It tackles the problem of Q-value overestimation in DQN, which arises from the fact that the Bellman equation computes the maximum of a noisy estimator.

The solution offered by Double DQN is to decompose the maximum operation in the target network into two independent operations: action selection and action evaluation. The Bellman error (Equation 2.7) is now as follows:

$$e = r(s,a) + \gamma Q(s', \operatorname*{argmax}_{a'} Q(s', a'; \theta); \theta^{'}) - Q(s,a,\theta). \tag{2.8}$$

Algorithm 2.2 Deep Q-Networks (DQN) in MDPs

Input: Parameters: ϵ, L, N

Output: π, the approximate optimal policy.

Initialize replay memory $\mathbb{E}$ to capacity L
Initialize $\theta = \theta^-$ randomly
for each episode **do**
    Initialize s_0
    for each step t in the episode **do**
        Take action $a_t \sim \pi_\epsilon(\cdot|s)$ and observe r_t, s_{t+1}
        Update memory replay $\mathbb{E}$ with current sample $e_t = (s_t, a_t, s_{t+1}, r_t)$
        Sample randomly a set $\mathbb{N}$ of N indexes from $\mathbb{E}$
        for each experience sampled e_t **do**
            Obtain $B_t = \begin{cases} r_t & \text{for terminal } s_{t+1} \\ r_t + \gamma \max_{a'} Q\left(s_{t+1}, a_{t+1}; \theta^-\right) & \text{for non-terminal } s_{t+1} \end{cases}$
        end for
        Update: $\theta \leftarrow$ Adam(loss function: $\left(B_t - Q\left(s_t, a_t; \theta\right)\right)^2$)
    end for
    Update: $\theta^- \leftarrow \theta$
end for
for all $s \in \mathcal{S}$ **do**
    $\pi(s) \leftarrow \arg\max_{a' \in \mathcal{A}}$ Predict $\left(\left(s, a'\right), \theta^-\right)$
end for
return π, θ

In this way, θ determines the greedy policy and $\theta^{'}$ determines its value. If these two sets of parameters are updated independently, the overestimation is reduced. This requirement is met simply by updating the parameters of the second network $\theta^{'}$ with the parameters of the target network, θ^{-}.

2.5 Network architectures

The use of DRL will allow for the development of an end-to-end controller that maps raw input data—consisting in data and images—to aircraft commands in the form of bank angle corrections. DNNs are able to automatically extract features from the raw observation input.

This capability, however, is improved by using in each case specialized network architectures for the particular type of raw input. Simple parameters such as velocity or distance of aircraft can be processed by standard fully connected layers. Feature extraction from images, on the other hand, can take advantage of an architecture that mimics the visual cortex of the human brain [22, 23]: Convolutional Neural Networks (CNNs).

2.5.1. Convolutional Neural Networks

CNNs [24, 25, 26] are a type of neural network for processing data that has a grid topology. They use the convolution—or, equivalently, the cross-correlation operation—instead of general matrix multiplication [27]. In practice, the typical topology for image processing consist of a series of convolution layers, implemented as 2-D cross-correlations, followed by a rectifier and subsampling. Each layer can be represented as in Figure 2.5.

The convolution layers are characterized by the following parameters:

- The number of independent filters, which determines the number of output images, known as feature maps.
- The kernel size, which determines the size in pixels of the sliding filter.
- The stride, which determines the number of pixels that the filter slides each time.

The typical rectifier or detector layer is the Rectified Linear Unit (ReLU) [28]. The subsampling layer is also known as the pooling layer and performs a summary operation such as the maximum (max pooling) to adjacent locations in the output. It is characterized by the size of the pooling window.

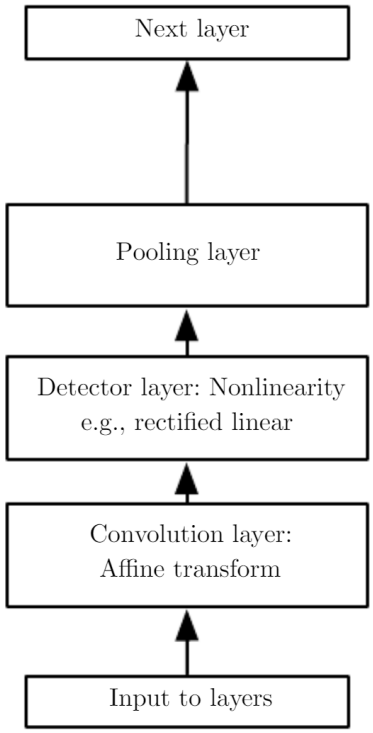


Fig. 2.5: Convolutional layer components. [27]

2.5.2. Recurrent Neural Networks: Partial observability and map representation

Some approaches to navigation such as Simultaneous Localization and Mapping (SLAM) [29] require an explicit representation of the map. This, however, imposes some constraints. As has been explained, an alternative approach to navigation was found in model-free RL methods, which make use of implicit representations but rely on hand-crafted features and are limited to low dimensionality problems. The solution to these limitations came in the form of DRL, which relies on DNNs to automatically extract features from raw input and provide the implicit representation [17].

This however doesn't solve by itself the problem with partial observability: DQN only works if the observations are a good approximation of the state or else the estimation of Q is not guaranteed to be good.

Recurrent Neural Networks (RNNs), which introduce memory to the model, can improve the estimation of the state by taking into account the history of observations. This, in turn, would allow the network to implicitly represent a map in memory.

Many architectures are possible for RNNs but what they all have in common is that they feed some of their outputs from a previous timestep as inputs to the network. This allows in principle for long-term memory, which we require for map representation. In practice however, only some architectures are suited for this. One of them is the Long Short-Term Memory (LSTM) architecture.

Long Short-Term Memory

A Long Short-Term Memory (LSTM) [30, 31] is a type of RNN. Figure 2.6 shows the internal structure of a LSTM, where σ represents a neural network layer with the sigmoid function as activation and *tanh* is the equivalent with a hyperbolic tangent activation.

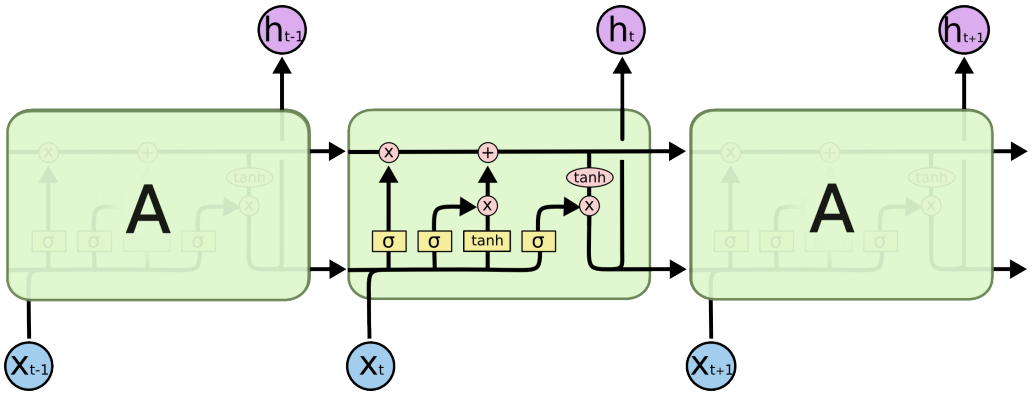


Fig. 2.6: LSTM structure. [32]

The top horizontal line which crosses the cell represents a vector that is known as the cell state and is responsible for long-term memory, whereas the bottom horizontal line is called hidden state.

The vertical line to the left represents the forget gate, which is trained to decide what information is kept in the cell state. The next sigmoid is the input gate layer, which chooses what information coming from the *tanh* layer will be added to the cell state. Finally, the last sigmoid layer decides how to compute the output hidden state from the cell state.

2.5.3. Deep Recurrent Q-Networks

The combination of DQN and LSTM is Deep Recurrent Q-Networks (DRQN) [33], a DRL algorithm that has been shown to be able to solve POMDP problems much more accurately than DQN thanks to its memory. Algorithm 2.3 shows the pseudocode for DRQN, where the input l is the length of the experience traces that are used in training. The update of θ has to perform Backpropagation Through Time (BPTT).

Algorithm 2.3 Deep Recurrent Q-Networks (DRQN) in POMDPs

Input: Parameters: ϵ, L, N, l

Output: π, the approximate optimal policy.

Initialize replay memory $\mathbb{E}$ to capacity L
Initialize $\theta = \theta^-$ randomly
Initialize LSTM state to zero
for each episode **do**
    Initialize s_0
    for each step t in the episode **do**
        Take action $a_t \sim \pi_\epsilon(\cdot|o)$ and observe r_t, o_{t+1}
        Update memory replay $\mathbb{E}$ with current sample $e_t = (o_t, a_t, o_{t+1}, r_t)$
        Sample randomly a set $\mathbb{N}$ of N sets of l consecutive samples from $\mathbb{E}$
        for each trace sampled $[e_t, ..., e_{t+l-1}]$ **do**
            Obtain $B_t = \begin{cases} r_t & \text{for terminal } s_{t+1} \\ r_t + \gamma \max_{a'} Q\left(s_{t+1}, a_{t+1}; \theta^-\right) & \text{for non-terminal } s_{t+1} \end{cases}$
        end for
        Update: $\theta \leftarrow$ Adam(loss function: $\left(B_t - Q\left(s_t, a_t; \theta\right)\right)^2$) with BPTT
    end for
    Update: $\theta^- \leftarrow \theta$
end for
for all $s \in \mathcal{S}$ **do**
    $\pi(o) \leftarrow \arg\max_{a' \in \mathcal{A}}$ Predict $\left(\left(o, a'\right), \theta^-\right)$
end for
return π, θ

3

Development tools and framework

In this chapter we will present an overview of the main development tools and frameworks that have been used for this project.

3.1 TensorFlow

TensorFlow [34] is an open source machine learning library for research and production which enables DL applications. Its development is led by Google and it is written in Python, taking Theano, an older framework, as a reference.

In recent years, TensorFlow has become the most popular DL framework, especially for deployment. This is in part due to the approach to the definition of computational graphs, which are static in TensorFlow. Other frameworks such as PyTorch are popular in academia, among other things, because graphs are dynamic and thus more flexible, allowing for an easier implementation of novel architectures. This however limits the efficiency, giving an advantage to TensorFlow in production. In this project we use TensorFlow 1.9 and Tensorflow-GPU 1.9.

TensorBoard is the official visualization toolkit for TensorFlow. It allows the visualization of the computation model graph and of metrics.

3.2 Stable Baselines

Several sets of implementations of DRL algorithms are available for TensorFlow. One of the most prominent is Baselines by OpenAI [35].

In this project we use Stable Baselines version 2.5.1 [36], which is an improved version of OpenAI Baselines. It serves us as the framework for the development of a multiagent version of DQN, of which a single agent implementation is available, and for the development of

a multiagent DRQN implementation from scratch. We use its common interface for DRL algorithms and calls to its auxiliary functions whenever it is possible.

3.3 Landlab

The training of agents in flood monitoring requires a flood simulation model. Landlab [37] provides an implementation of a stable and simple formulation of the shallow water equations for 2-D flood modeling by Almeida et al. [38]. This tool also provides us with a quick mechanism of generating the topography of the terrain by means of erosion of a random initial terrain.

We use Landlab version 1.5.4.

3.4 Deep RL for swarms

The environment structure is partly based in and inspired by Deep RL for Swarm Systems [39], a repository containing the codebase for [40]. We extend the Agent class to develop our Aircraft class and modify the world definition to fit the dynamics of the agents, rewards and observations and to integrate the natural disaster simulators.

3.5 Hardware and setup

The simulations were performed with two hardware setups:

- Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz CPU, 16402MB of RAM, NVidia Titan V GPU, Ubuntu 18.04.2 LTS x64.
- AMD Ryzen Threadripper 2950X 16-Core CPU, 65882MB of RAM, NVidia GeForce RTX 2080 Ti GPU, Xubuntu 18.04.2 LTS x64.

We used Python version 3.6, which is compatible with all of the previous tools, and Pycharm Community 2019 as the development framework. For Tensorflow-GPU to access the GPU, both computers use CUDA 9.0 and CuDNN 7.2.1.

4

Problem description

4.1 Natural disaster surveillance

We will now formally describe the problem of natural disaster surveillance as a swarMDP, which we have argued in Section 2.2 to be the best mathematical model for the problem. Our implementation of the problem, which we will detail in Chapter 5, follows this description.

4.1.1. Agents

The agents are fixed-wing UAV flying at different altitudes, so that collisions are not possible. Aircraft velocity v is constant and equal to 20 m/s. Each agent decides whether to increase or decrease its bank angle ϕ by 5 degrees at a frequency of 10 Hz and the position is updated with the kinematic model described by the following expressions:

$$\dot{x} = v\cos\psi,\ \dot{y} = v\sin\psi,\ \dot{\psi} = \frac{g\ \tan\phi}{v}, \tag{4.1}$$

where (x, y) is the position of the aircraft, ψ is the heading angle, ϕ is the bank angle and g is standard gravity, 9.8 m s^{-2}.

4.1.2. State

The full state of the system has two sets of components:

- The local state of each agent, which itself has several components:
  - The position of the aircraft in cartesian coordinates (x_i, y_i).
  - The heading angle ψ_i of the aircraft or direction towards which it is moving, measured with respect to the north and expressed as an azimuth.
  - The bank angle of the aircraft ϕ_i, which determines the rate of change of the heading angle.

- The maps required to fully describe the state of the natural disaster, which will be detailed for each case in this chapter. The environment is discretized into cells. Thus, in practice the maps will be images with a resolution of 10 meters per pixel that describe the state of each cell.

4.1.3. Actions

The set of local actions is ⟨ increase bank angle ϕ_i by 5 degrees, decrease bank angle ϕ_i by 5 degrees ⟩. Each agent has to choose one of them every 0.1 s. The maximum bank angle ϕ_{max} is set to 50 degrees and actions exceeding this limit have no effect. This establishes a realistic limit to angular velocity.

4.1.4. Observations

Each agent collects two kinds of observations: a vector of features representing some information of the local state of the own aircraft and the other aircraft, and an image representing a partial observation of the natural disaster from the perspective of the aircraft.

The vector of features of each agent i is itself the concatenation of four vectors of continuous variables: all the bank angles $\{\phi_j \mid j \in 1, 2, ..., N\} = \phi$, the range or distance to other aircraft $\{\rho_{ij} \mid j \in 1, 2, ..., N \wedge \ j \neq i\} = \rho_i$, the relative heading angle to other aircraft $\{\theta_{ij} \mid j \in 1, 2, ..., N \wedge \ j \neq i\} = \theta_i$ and the relative heading angles of other aircraft $\{\psi_{ij} \mid j \in 1, 2, ..., N \wedge \ j \neq i\} = \psi_i$, where N is the number of agents.

Given the update frequency, in practice this requires that the agents have the communications capabilities necessary to receive $40 \cdot N$ parameters/s.

The images are assumed to be the processed output of a hemispherical camera with a view angle of 160 degrees always pointing downwards, resulting in a maximum range of 500 m. The final image is a 30 x 40 pixel representation of the points of interest of the natural disaster around the aircraft. This image can represent either fire or flood detections and is aligned with the heading direction of the aircraft.

The image is generated by sampling the points of interest at 40 azimut angles uniformly and each of these angles at 30 elevation angles uniformly from nadir to 10 degrees below the horizon. This results in a non-uniform sampling in range with more resolution right below the aircraft than at 500 m.

The degree of partial observability of the problem is determined mainly by the characteristics of this image.

4.1.5. Rewards

We define a reward function that should be maximized during the training process. The reward corresponding to each agent i consists of the sum of four distinct components:

$$r_1 = -\lambda_1 \min_{\{s \in S | F_t(s)\}} d_s/d_{max}, \tag{4.2}$$

$$r_2 = -\lambda_2 \sum_{\{s \in S | d_s < r_0\}} 1 - F_t(s)/n_c, \tag{4.3}$$

$$r_3 = -\lambda_3 (\phi_0/\phi_{max})^2, \tag{4.4}$$

$$r_4 = -\lambda_4 \sum_{\{j \in 1,2,\ldots,N \wedge\ j \neq i\}} \exp\left(-\frac{\rho_{ij}}{c}\right), \tag{4.5}$$

where d_s is the distance from the aircraft to cell s, d_{max} is a normalization factor for the distance, n_c is the number of cells in the circle of radius r_0 and $F_t(s)$ represents whether cell s is active (on fire or flooded). These are penalties for:

- Distance from points of interest (r_1): proportional to the distance to the closest cell of interest.
- Dry cells nearby (r_2): proportional to the number of cells in which there is nothing of interest in a radius r_0.
- High bank angles (r_3): proportional to the square of the bank angle.
- Closeness to other aircraft (r_4): sum of the contributions of each one of the other aircraft, which saturate to λ_4.

The components of the reward are normalized to be in the same range for equal tuning parameters. These tuning parameters have been set to the following values:

$$\lambda_1 = 1, \tag{4.6}$$

$$\lambda_2 = 2 \cdot 10^{-1}, \tag{4.7}$$

$$\lambda_3 = 5 \cdot 10^{-2}, \tag{4.8}$$

$$\lambda_4 = 1 \cdot 10^{-2}, \tag{4.9}$$

$$r_0 = 5, \tag{4.10}$$

$$c = 20. \tag{4.11}$$

The reasoning behind the selection is that flying far away from the points of interest should be heavily penalized and, once over the points of interest, the penalties from bank angle and closeness to other aircraft should become important.

As mentioned in 2.2.3, some problems can benefit from the use of a team reward shared between all the agents. For our problem, we will define this alternative reward as the sum of the individual rewards of all the agents.

4.2 Wildfires

We will define our wildfire surveillance environment as in [6], whose results we take as a reference. The problem is set on a 1km^2 area that is represented as a 100 x 100 grid of burnable cells.

4.2.1. State

As mentioned in Subsection 4.1.2, the state of the natural disaster is described by maps. In the case of wildfires, two dynamic maps are required:

- $B(s)$: Fuel map which represents the amount of flammable material that is present at each cell.
- $F(s)$: Fire map consisting of a boolean representation of whether each cell is burning.

4.2.2. Wildfire propagation model

The cells are updated every 2.5 s by means of the following two equations:

$$B_{t+1}(s) = \begin{cases} \max\left(0, B_t(s) - \beta\right) & \text{if } F_t(s) \\ B_t(s) & \text{otherwise} \end{cases} \tag{4.12}$$

where β is the burning rate, which we will set to 1, and

$$p(s) = \begin{cases} 1 - \prod_{(s' | d(s,s') < d_{mp})} \left(1 - P\left(s, s'\right) F_t\left(s'\right)\right) & \text{if } B_t(s) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

where $p(s)$ is the probability that cell s will ignite, $d(s, s')$ is the distance between cells s and s', d_{mp} is the maximum distance at which propagation can happen and $P(s, s')$ is the probability that cell s' ignites cell s, which we define as follows:

$$P(s, s') = w \cdot d(s, s')^{-2} \tag{4.14}$$

where w is a constant proportional to the wind speed.

According to equation 4.12, the burnable material is reduced by one unit in each interaction if the cell is on fire. The cell only extinguishes once the fuel runs out.

Equation 4.13 establishes that only cells with fuel left can ignite and it determines the influence of neighboring cells in the ignition.

4.2.3. Initialization

The fuel map $B(s)$ is randomly initialized as $B_0(s) \sim U(15, 20)$, with a value between 15 and 20 units of fuel. The simulated wildfires start at the center as a 3x3 cells burning square and are left to propagate for 50 s before the aircraft are placed in random positions on the map.

4.2.4. Observations

In our wildfire model the observation is directly obtained from the state: it is the output of the process described in Subsection 4.1.4 when the input is $F(s)$, the fire map.

4.3 Floods

The environment consists of a 2D map representing a 1 km^2 area. It is discretized into 100 x 100 cells, each with an elevation and a water level.

4.3.1. State

Given that the terrain is fixed during each episode, the state of the flood is fully described by the surface water depth map $D_t(x, y)$, a 100 x 100 matrix, updated every 10 seconds by the flood simulator.

4.3.2. Terrain generation and flood initialization

The terrain is randomly generated at the beginning of each episode of training. The elevation map is created in a two step process: first, a 100 x 100 Brownian surface is generated. This initial map is then eroded for 50 years using the FlowAccumulator and FastscapeEroder modules in Landlab while the terrain is uplifted.

FastscapeEroder computes the amount of erosion caused by rainfall at each node of the grid using the Braun-Willett Fastscape approach [41], while FlowAccumulator accumulates flow and calculates the drainage area.

The resulting terrain typically has a maximum relief of 10-30 m.

We focus on those floods which are the product of dam collapse. This allows for a controlled environment in which the optimal solution is particularly intuitive: the flooded area will move downhill from the dam and the aircraft should fly through this valley. Episodes are initialized by placing a 20 m x 20 m, 5 meter high body of water in the center of the map before letting the flood simulation run.

4.3.3. Flood simulation

During the episode, the terrain is fixed. The flood is modeled as a shallow water flow over topography using the numerical approximation of de Almeida et al. [38]. This algorithm is based on a 2-D regular grid in which a system of hyperbolic partial differential equations is solved numerically.

The de Almeida et al. model is run with Manning's roughness coefficient n = 0.01 s $m^{-1/3}$, a relatively low friction configuration, in order to simulate fast evolving floods.

4.3.4. Observations

To generate the image, the surface water depth map $D_t(x, y)$ is compared with a fixed water level threshold, here set to 5 cm, to simulate the inability of the image processing algorithm to detect very shallow waters. The resulting boolean map of detectable flooded areas $F_t(x, y)$ is then used as an input to compute the observation of the aircraft through the process that was described in 4.1.4.

5

Environments

In Chapter 4 we described the problems to solve as swarMDPs. In this chapter we will explain how we implemented these swarMDPs in Python.

5.1 Natural disaster surveillance

The environment has been implemented in a modular fashion, through class inheritance, so that the common dynamics of all natural disaster surveillance problems are shared and compatibility with new environment definitions and algorithms is made easier.

5.1.1. SurveillanceEnv class with Gym interface

Gym [42] is a collection of libraries by OpenAI which provides, among other things, a common interface for single-agent RL environments to interact with the algorithms. We implement our environment as a class called SurveillanceEnv that inherits from gym.Env and extends it to allow for multiagent surveillance environments. In this way, the environment implementation is completely independent from the algorithms that control the agents.

A SurveillanceEnv instance is initialized with several atributes and properties, of which the most relevant are the following:

- The state, observation and action space and reward definitions.
- Multiple instances of the Aircraft class, defined as described in Subsection 5.1.3, one for each each agent.
- The timestep limit of each environment, which is 5000 timesteps (500 s or 8:20 min) for wildfires and 10000 timesteps (1000 s or 16:40 min) for floods.

- An instance of the corresponding World child class, defined as described in Subsection 5.1.2.

The SurveillanceEnv class allows the user to choose between an individual or a shared reward scheme, also known as team reward, as explained in Subsection 2.2.3.

This class also defines the reset method to reset the whole environment for a new episode, the step method to run a timestep of both the natural disaster simulation and the aircraft dynamics, the reward method that implements the reward function and a render method that provides a graphical representation of the environment using the Matplotlib library.

5.1.2. World

Our world implementation is based on *Deep RL for swarms* [39]. The *Deep RL for swarms* World class is modified to provide the dynamics of the fixed wing UAV described in Subsection 4.1.1 in its step function.

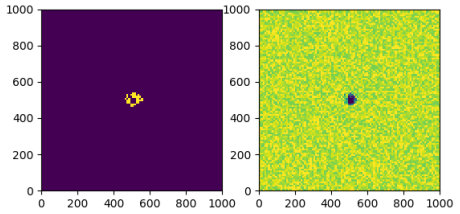
The ActiveCellWorld class, which inherits from the modified World class, contains the grid of cells that can represent either wildfires or floods. Each environment will have to provide its own subclass of ActiveCellWorld by overriding the reset and step methods.

5.1.3. Aircraft

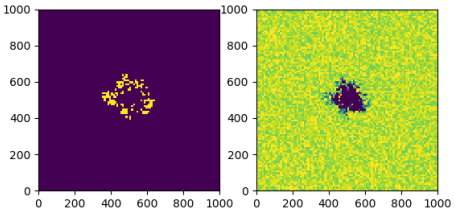
The Aircraft class inherits from the Agent class in *Deep RL for swarms*. It contains as attributes the configuration parameters for the dynamic equations of the UAV and the current position, heading angle and bank angle of the aircraft. It also implements the get_observation function, which performs the operations described in Subsection 4.1.4 to generate from $F(s)$ the observation image corresponding to the position and heading of the aircraft and concatenates the data from the own aircraft and from other aircraft to generate the observation vector.

5.2 Wildfires

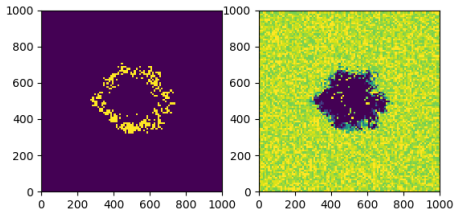
The WildfireWorld class, which inherits from the ActiveCellWorld class, implements the fuel and fire maps defined in Subsection 4.2.1. It also implements the reset method as described in Subsection 4.2.3 and the update_cells method using the ignition and fuel depletion equations in 4.2.2. For efficiency, we implement $P(s, s')$ as a convolution by defining a burning cell map filter that performs the product operation in Equation 4.13 as a sum of logarithms. Figure 5.1 shows the evolution of a wildfire throughout an episode. Note that actually t = 0 s is does not correspond with the initial conditions because the simulation runs for 50 s before the episode starts.



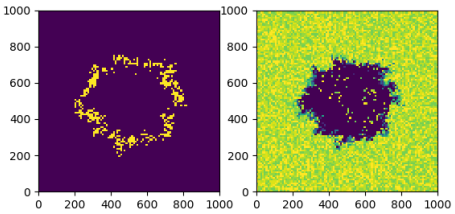
(a) t = 0 s.



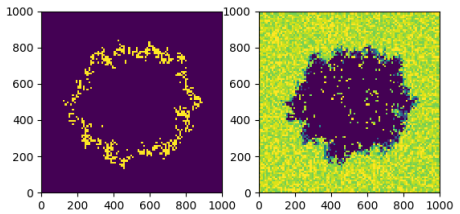
(b) t = 100 s.



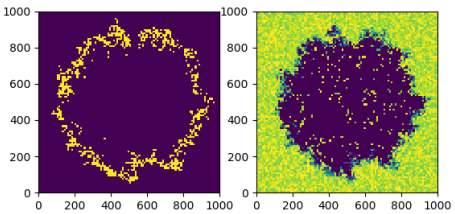
(c) t = 200 s.



(d) t = 300 s.



(e) t = 400 s.



(f) t = 500 s.

Fig. 5.1: An example wildfire evolution over time. Left: burning cells. Right: fuel left.

5.3 Floods

Our implementation of the terrain generation and flood simulation makes use of LandLab components. [37].

The FloodWorld class, which inherits from the ActiveCellWorld class, also implements the maps defined in Subsection 4.3.1 and the methods reset and update_cells.

Figure 5.2a shows an example of the terrain relief that is generated through the method that we explained in Subsection 4.3.2.

5.3.1. Terrain generation

The process goes as follows:

1. We create the Brownian surface using a Fourier-based power law noise with frequency bounds and apply a vertical scale to it.
2. A Landlab RasterModelGrid is created and open boundary conditions are set.
3. We add a surface water depth map field to the RasterModelGrid and initialize it to zero.
4. The Brownian surface is added as an elevation field to the RasterModelGrid.
5. FlowAccumulator and FastscapeEroder simulation parameters are configured.
6. We use the FlowAccumulator and the FastscapeEroder Landlab classes to simulate 50 years of terrain erosion.

As we mentioned in Subsection 4.3.2, FastscapeEroder computes the amount of erosion at each node of the grid using the Braun-Willett Fastscape approach. Meanwhile, FlowAccumulator computes the accumulation of flow from erosion and then calculates the drainage area. Landlab provides default configurations that set the simulation parameters in step 5, such as the stream power equation constants, to adequate values for our purposes.

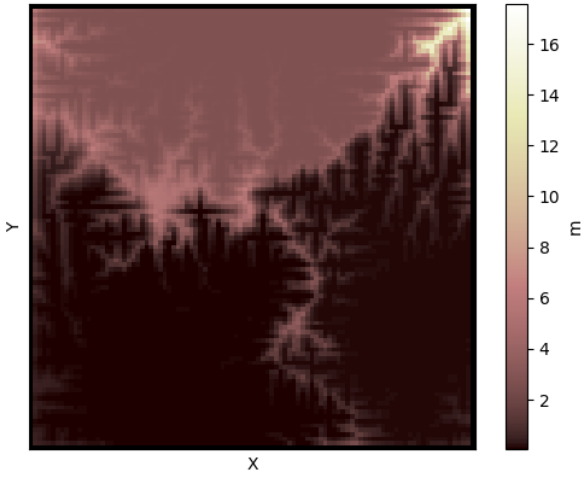
5.3.2. Flood simulation

Figures 5.2b to 5.2d show the evolution of the surface water depth map $D_t(x, y)$ of a simulated flood.

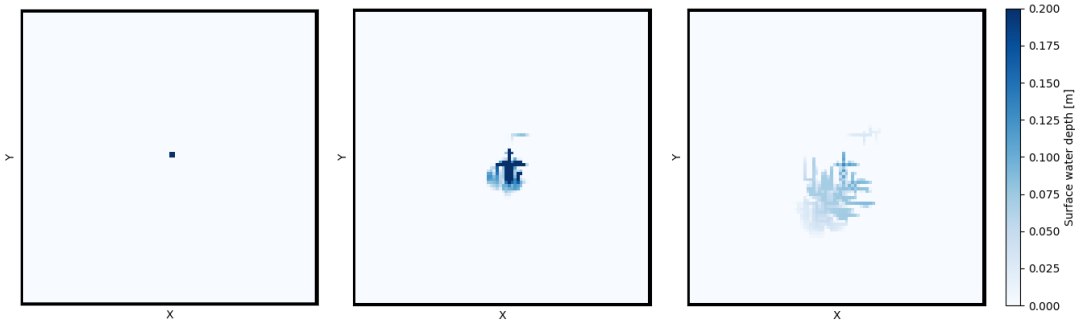
The simulation of the flood itself uses OverlandFlow, a different Landlab component that implements the model of shallow water equations for 2-D floods by Almeida et al. [38]. We set Manning's roughness coefficient to 0.01 s $m^{-1/3}$, a relatively low friction configuration, in order to simulate fast evolving floods. The initial layer of water in the four central cells is set to 5 meters to simulate the initial conditions of the dam collapse. The cell update

period is set to 10 seconds, a good compromise between simulation accuracy and computing time.

Figure 5.3c exemplifies the process of obtaining the observation for an agent from the true map of the natural disaster. In the case of floods, we apply the water level threshold mentioned in 4.3.4 and then execute get_observation as explained in Section 5.1.3.

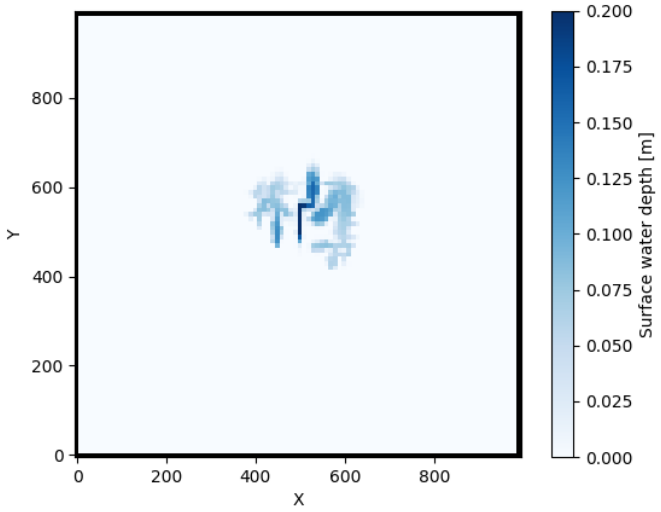


(a) Terrain.

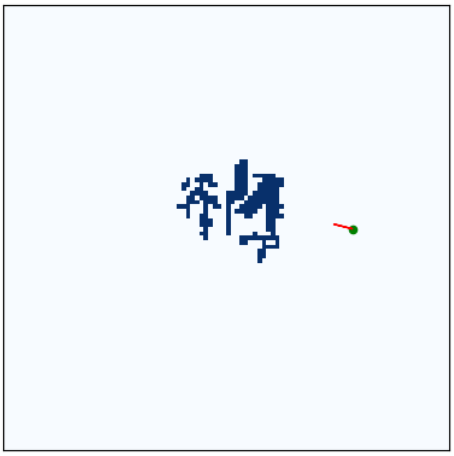


(b) Flood at t = 0 s. *(c)* Flood at t = 100 s. *(d)* Flood at t = 500 s.

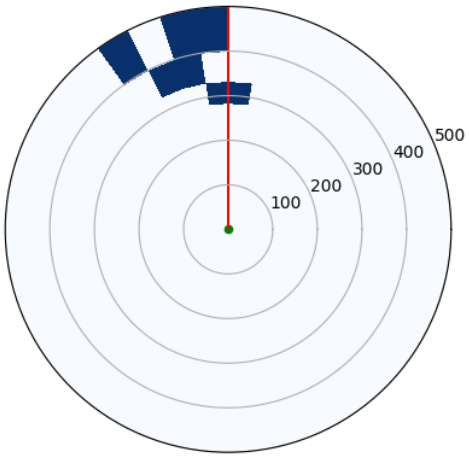
Fig. 5.2: An example of random terrain generation and flood evolution over time.



(a) True flood.



(b) Visible flood.



(c) Agent observation.

Fig. 5.3: An example of the generation of observation images in the flood environment.

6

Algorithms

In sections 2.4 and 2.5 we have presented the theory underpinning the algorithms that were implemented in this project. In this chapter we discuss the implementation details of these algorithms.

Both our DQN and DRQN implementations were made to take Gym environments as input and to be compatible with our modified multiagent Gym environments.

6.1 DQN

In the case of DQN, our objective was to adapt Stable Baselines' implementation [36] to deal with multiagent environments as ours.

The main changes that had to be made to the algorithm implementation to make it compatible with multiagent environments are the following:

- Independent handling of the actions selected by each agent during training, by calling the act function once per agent.
- Modified experience storage to save the transitions from all agents in the same timestep.
- Modified reward logging.
- Modified predict function for execution with multiple agents.

Double DQN is available in the Stable Baselines implementation of DQN without any additional change other than the previously mentioned.

6.1.1. Network architecture

The other main aspect of the implementation of the algorithm is the network architecture. We implemented a very similar architecture to the one that is used in [6].

As we have explained in Section 2.5, the DNN has to deal with the two types of observations separately. The features go through a series of dense layers with ReLU as the activation function. The image is instead processed by a series of convolutional layers and max pooling layers which reduce the dimensionality of the image in an efficient way before entering its own dense layers. The outputs of both networks are then concatenated and go through two additional dense layers. The complete architecture is shown in Fig. 6.1.

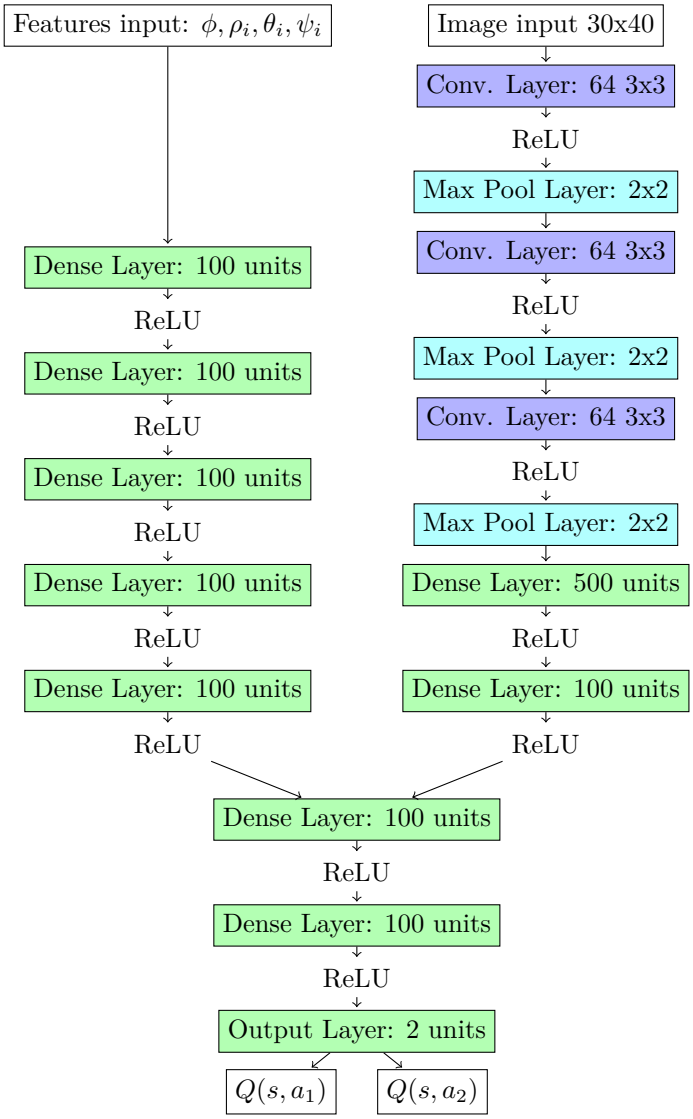


Fig. 6.1: DQN network architecture.

On a practical note, it is worth mentioning that the DQN implementation in Stable Baselines is not compatible with gym.spaces.Dict (dictionary) or gym.spaces.Tuple observation spaces, which would allow us to define a gym.spaces.Discrete vector for the features and a gym.spaces.MultiBinary 2-D matrix for the images. What this means is that for compatibility we had to define the observation space as a single vector with the Gym Box class and make the network itself reshape the observation again to match the inputs.

6.2 DRQN

In the case of DRQN, the implementation was not as straightforward as with DQN because there is no DRQN implementation available in Stable Baselines nor is there compatibility of the DQN implementation with recurrent policies. Instead, a much deeper refactoring was required for recurrency to work.

Firstly, regarding the experience replay, a new buffer is required. As we saw in Algorithm 2.3, it has to be able to save the experience from all agents and then sample traces, each of which should correspond to consecutive samples from a single agent and a single episode.

We base our implementation in [43]: we create a ReplayBuffer class which can be used both as a buffer of episodes and as a buffer of samples of a single episode and from a single agent. Thus, we have to modify the logic of our multiagent DQN implementation to create a new buffer of samples per agent at the beginning of each episode. Later, at the end of each episode, we add the contents of these buffers to the episodes buffer, as lists inside a list. If the buffer of episodes is full, we drop the oldest episode before adding a new one. In this way, we can now sample by first randomly choosing an episode of experience and then choosing a random set of consecutive samples from that episode.

The second main change with respect to our multiagent DQN implementation is with regards to the network architecture and usage. We have to introduce one or more LSTM layers into the model in order to introduce memory. We will do this as we show in Figure 6.2, by substituting the last hidden dense layer with a LSTM layer, shown in red. This has been shown to be a successful approach by multiple works [33, 44].

Also, we no longer train with batches of experience samples, but with batches of experience traces. Accordingly, the LSTM layer has to receive a reshaped input and then reshape its output so that the rest of the network processes all samples the same way as in DQN.

Tensorflow's LSTM cell implementation makes BPTT transparent to us, since the cell is automatically unrolled when performing backpropagation and no other changes are required. However, it has been shown in [44] that it is better to train only with samples for which the state has not been reinitialized recently. We will apply this training technique by masking the losses for the first half of the samples in each trace.

In practice, RNNs have a problem with gradient descent based method because of what is known as exploding gradient [45]. The cause is that BPTT does the equivalent of unrolling the recurrent network into as many copies as the trace length to compute the required partial derivatives. This means that the gradients that propagate to the older components are the product of many components and can easily diverge in some situations. To avoid this, we apply what is known as gradient clipping, establishing a threshold to the range of each gradient component.

The presence of memory also implies that we have to maintain a state in execution. For this reason we modify the predict function accordingly.

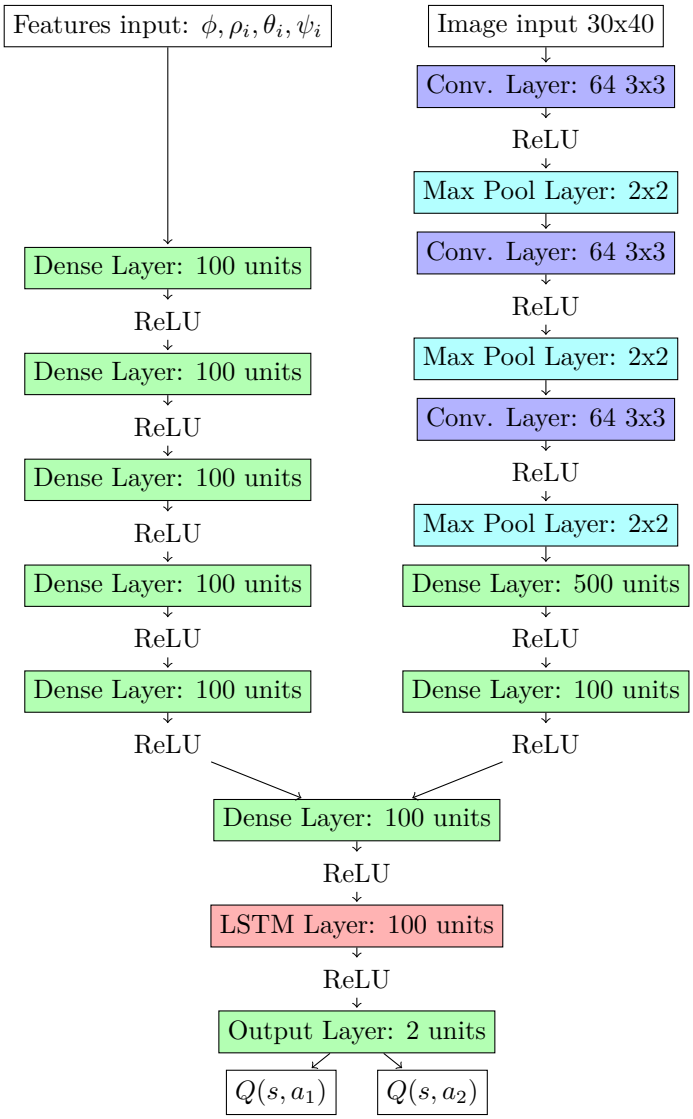


Fig. 6.2: DRQN network architecture.

7

Simulations and results

In this chapter we will present a selection of the most relevant simulations that were performed in the environments showcased in Chapter 5 with the algorithms detailed in Chapter 6. The main performance metric will be the episode accumulated reward, which is the sum over an episode of the instantaneous rewards of all agents.

7.1 Wildfires: DQN with independent and team reward schemes

The first result of the simulations was on the one hand, the successful replication of the results in [6] with our own wildfire environment and multiagent DQN implementation, and, on the other hand, the comparison of independent and team reward configurations as defined in 4.1.5. These results will serve us as a baseline for the performance that can be expected.

We test both independent and team reward schemes with two agents. The following parameters are common to both training schemes:

- Batch size $N = 2000$ timesteps.
- Target network update frequency = 1000 timesteps.
- Learning starts at 10000 timesteps.
- Training frequency = 100 timesteps.
- Buffer size $L = 100000$ timesteps.
- Fraction of training with additional exploration = 70%.
- Exploration in the final episodes = 10%.
- Maximum training steps = $3 \cdot 10^6$ timesteps.
- Learning rate = $5 \cdot 10^{-4}$.

This means that training is performed every 100 steps, once 100 new samples have been collected. Each of the training steps uses a batch size of 2000 timesteps, randomly sampled from a buffer containing the last 100000 timesteps of experience. The target network is updated every 1000 steps, that is, once for every 10 action-values update. Learning only starts after the first 10000 steps, so that the network does not overfit to the first samples. The fraction of exploratory actions starts at 1, decreases linearly during the first 70% of training time and stays at 0.1 until the end of training. We use a learning rate of $5 \cdot 10^{-4}$ for the stochastic optimization method Adam [19].

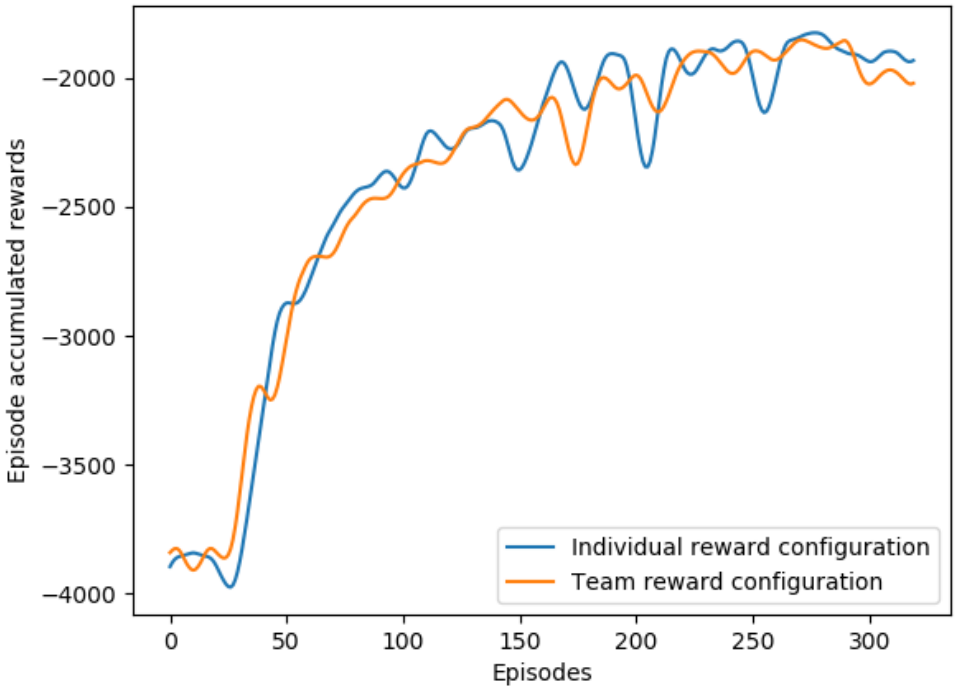


Fig. 7.1: Training curves of DQN in wildfires. 21 episodes running average.

Reward scheme	Independent	Team
Average accumulated reward in validation	-1837	-1868

Tab. 7.1: Comparison of independent and team reward configurations in wildfires with DQN.

Figure 7.1 shows a comparison of the evolution of the episode accumulated reward during training. It is shown that both independent and team reward configurations allow the agents to learn at similar rates and reach the same level of performance during training. Validation is performed in execution with the trained model and full exploitation, i.e., with no random actions. It is configured to validate the same 100 episodes with equal environment conditions for all models. Table 7.1 is a comparison of the validation results, which show that the performance of both reward configurations is virtually identical.

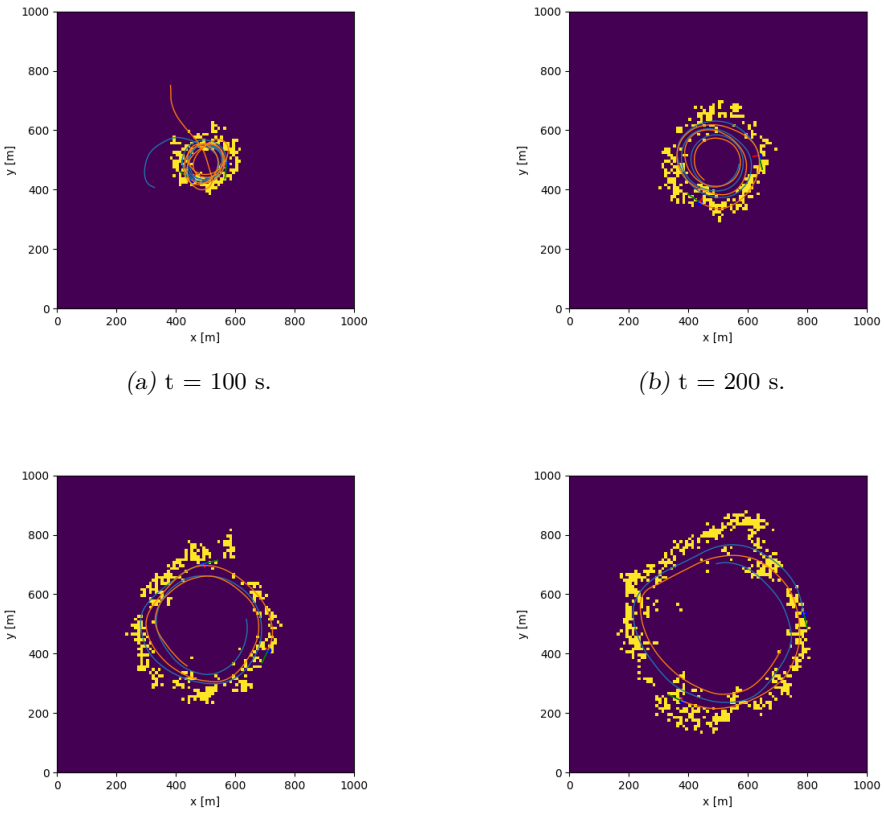
A formal proof of this is provided by Welch's t-test, which is a two-sample location test that is used to test the hypothesis that two populations have equal means. The T-statistic

represents the ratio of the difference between two sets and the difference within the sets. The P-value is the probability that the result is not significant, so large values indicate weak evidence against the null hypothesis. Welch's t-test does not assume equal variances.

Given that for this test the T-statistic is equal to 1.07 and the P-value is equal to 0.28, the two distributions of accumulated rewards cannot be considered significantly different in mean.

We could expect a similar behaviour in both configurations for this particular multi-agent problem because three of the four penalty components (r_1, r_2 and r_3) are independent of the behaviour of the other aircraft, meaning that the margin of improvement is only related to the optimization of the relative positions of aircraft with respect to each other.

Figure 7.2 shows a simulated episode in wildfires as an example of the behaviour that is learnt by the agents. Aircraft learn to head towards the areas on fire and start making wider circles as the wildfire propagates. The penalization to the distance between agents is effective in making both aircraft circle the fire in the same direction and in opposite ends of the fire, far from eachother.



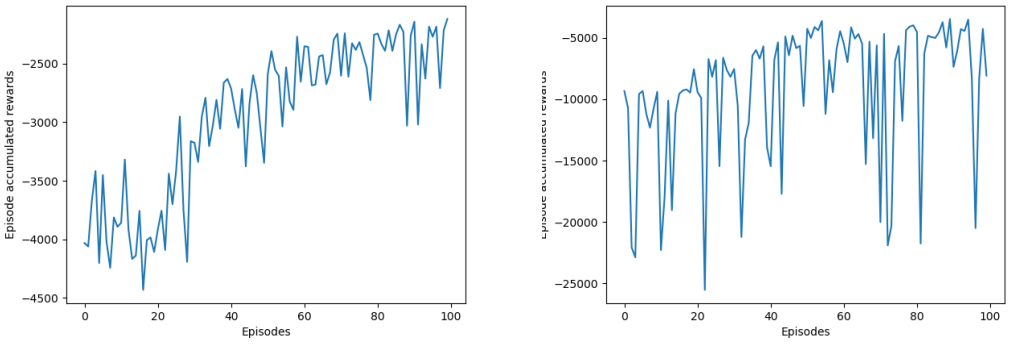
(a) t = 100 s. *(b)* t = 200 s.

(c) t = 300 s. *(d)* t = 400 s.

Fig. 7.2: An example of an episode in wildfires with two aircraft trained with DQN, team reward scheme.

7.2 Floods

We apply the same approach to solve the problem of flood surveillance. The experiments show that this approach successfully solves the surveillance problem and the same conclusions that were reached in the wildfire problem apply. We notice however a notable difference in the stability of the reward during training. The absolute value of the rewards is not comparable because the flood episodes are twice in length and have different dynamics. Note in Figure 7.3 the difference in the relative magnitude and frequency of the outliers, which are significantly worse in the flood training process. This is due mainly to the fact that the flood problem is much more diverse than the wildfire problem: some floods are local, some reach far from the origin, some disappear quickly and some divide into several affected zones.



(a) Wildfire. *(b)* Flood.

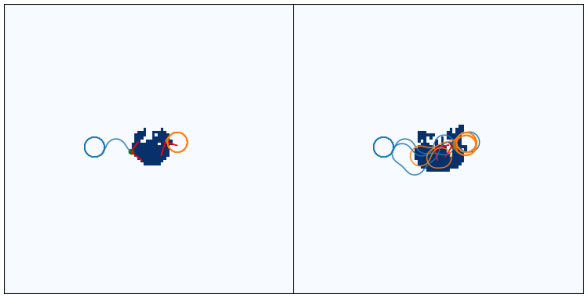
Fig. 7.3: Comparison of unfiltered accumulated reward during training in wildfires and floods.

Figure 7.4 shows a simulated episode in floods as an example of the behaviour that is learnt by the agents. Aircraft learn to head towards the flooded areas once they are detected and stay close to them, but are forced to avoid staying in the same area making small circles because high bank angles are penalized. Instead, the agents go through the flood while keeping a distance from other aircraft. When the area of interest is too small, the penalty to the proximity between aircraft outweighs the rest and one of the aircraft stays where it is making small circles.

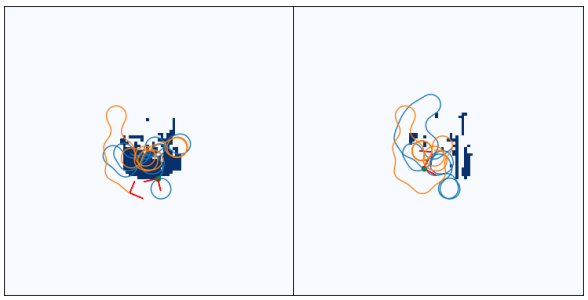
7.3 DRQN

Julian et al. [6] showed that an approach to the problem of wildfire surveillance that combines DQN and SLAM does not show significant improvement with respect to a solution with no memory at all. This approach consists in making the agents read and update a shared 2-D map of the natural disaster.

The DRQN experiments that we have performed are meant to test whether a different approach to add memory, i.e., adding a recurrent layer to the DNN, can improve the solution.



(a) t = 50 s. *(b)* t = 100 s.



(c) t = 150 s. *(d)* t = 200 s.

Fig. 7.4: An example of an episode in floods with two aircraft trained with DQN

We tested several LSTM layer sizes and concluded that a size of 100 units is a good compromise between the complexity added to training and the capabilities of the network. We tested several trace lengths l, from 8 to 500 timesteps. The rest of the parameters were kept constant except for batch size, which was tuned for GPU efficiency in each case.

The results obtained for DRQN in wildfires show no improvement with respect to DQN. A Welch's t-test equivalent to that of Table 7.1 results this time in a T-statistic value of 1.17 with a P-value of 0.24, which indicate no significant difference in the accumulated reward distributions in the evaluation experiments.

We hypothesize that this is a sign of observability being too complete for memory to make a significant difference in the learnt behaviour. What this would mean is that, for our intents and purposes, the problem can be solved as the swarm equivalent of an MDP instead of a POMDP.

8

Conclusions and future lines

8.1 Conclusions

This thesis has brought me the opportunity to dive deep into the theoretical underpinnings of DRL, but also to come into contact with the practicalities of the field. It has allowed me to gather experience, among others, in Python and TensorFlow software development, TensorBoard logging, debugging and testing and DRL hyperparameter tuning.

The project has provided a series of results which stem from the original technique in [6] for training through DRL a deep neural network capable of guiding multiple fixed-wing aircraft to wildfires in a decentralized fashion.

Firstly, we aimed to find out whether a usage of memory different from the SLAM technique chosen by [6] could actually improve the results with respect to DQN, which doesn't use memory at all. We also wanted to test whether alternative reward schemes such as the team reward in [14] are beneficial in this problem. Finally, we aimed to formalize and test the application of similar techniques to the problem of decentralized flood monitoring.

With regards to the last objective, the simulation results are very clear: with the current detection techniques and the capabilities of DRL, UAVs are able to take decisions from raw input data, consisting of a processed optical image and some information of the local state of the swarm, and successfully coordinate to monitor efficiently a flood.

These controllers could be installed in real UAVs in order to reduce operational costs. We trained on simulated floods from dam collapses but the training approach should perform well on all kinds of floods as long as the training data is general enough.

Regarding the team reward scheme, we have shown that the natural disaster surveillance problem does not benefit from this technique, showing that the independent rewards are actually well aligned with the objective.

Finally, the results relative to the use of memory make us draw some interesting conclu-

sions and raise new questions to tackle in the future. Julian et al. [6] showed that their use of SLAM techniques did not improve significantly the results. At a first glance, we could think that their use of a manually defined memory might be preventing a more efficient use of memory. The use of DRQN has shown, on the contrary, that even a recurrent network which trains itself to use the memory without any additional contraint is unable to improve the results.

This points to a feature of the problem itself: even though the distributed natural disaster surveillance problem is formally a swarMDP, the partial observability corresponding to the camera model that has been assumed is close enough to meeting the Markovian assumption of full observability of the state that adding memory cannot significantly increase performance.

8.2 Future lines

8.2.1. Meaningful partial observability

With this hypothesis in mind, one of the future lines of research is the testing of environments where the partial observability is expected to play a bigger role:

- New kinds of natural disasters which require closer visual inspection.
- Other floods and wildfires monitoring techniques that offer additional information, so that the observation radius of the agents is reduced.

8.2.2. Alternative DRL approaches

The field of multiagent DRL has made great advances in recent years. Some new approaches to decentralized coordination of swarms appear to be of interest for future research. In particular new algorithms such as QMIX [46] have found ways to use centralized learning to extract decentralized policies.

In the opposite end, in some settings, improved communication capabilities might make centralized learning and execution techniques a field worth exploring.

Bibliography

[1] CRED, "Natural disasters 2017. Brussels: CRED." 2018 EM-DAT file dated 02/07/2018.

[2] MAPAMA – Ministerio de Agricultura y Pesca, Alimentación y Medio Ambiente, "Los incendios forestales en España: Avance informativo. 1 de enero al 31 de diciembre del 2017." `https://www.mapa.gob.es/es/desarrollo-rural/estadisticas/iiff_2017_def_tcm30-446071.pdf`.

[3] P. Matgen, R. Hostache, G. Schumann, L. Pfister, L. Hoffmann, and H. Savenije, "Towards an automated sar-based flood monitoring system: Lessons learned from two case studies," *Physics and Chemistry of the Earth, Parts A/B/C*, vol. 36, no. 7, pp. 241 – 252, 2011. Recent Advances in Mapping and Modelling Flood Processes in Lowland Areas.

[4] L. Merino, F. Caballero, J. R. Martínez-De-Dios, I. Maza, and A. Ollero, "An unmanned aircraft system for automatic forest fire monitoring and measurement," *Journal of Intelligent & Robotic Systems*, vol. 65, no. 1-4, pp. 533–548, 2012.

[5] Q. Feng, J. Liu, and J. Gong, "Urban flood mapping based on unmanned aerial vehicle remote sensing and random forest classifier—A case of Yuyao, China," *Water*, vol. 7, no. 4, pp. 1437–1455, 2015.

[6] K. D. Julian and M. J. Kochenderfer, "Distributed wildfire surveillance with autonomous aircraft using deep reinforcement learning," *Journal of Guidance, Control, and Dynamics*, pp. 1–11, 2019.

[7] D. Baldazo, J. Parras, and S. Zazo, "Decentralized Multi-Agent deep reinforcement learning in swarms of drones for flood monitoring," in *2019 27th European Signal Processing Conference (EUSIPCO) (EUSIPCO 2019)*, (A Coruña, Spain), Sept. 2019.

[8] I. Navarro and F. Matía, "An introduction to swarm robotics," *Isrn robotics*, vol. 2013, 2012.

[9] E. Şahin, "Swarm robotics: From sources of inspiration to domains of application," in *International workshop on swarm robotics*, pp. 10–20, Springer, 2004.

[10] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT press, 2005.

[11] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. Adaptive Computation and Machine Learning, MIT press, 2 ed., 2018.

[12] D. L. Poole and A. K. Mackworth, *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[13] A. Šošić, W. R. KhudaBukhsh, A. M. Zoubir, and H. Koeppl, "Inverse reinforcement learning in swarm systems," in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pp. 1413–1421, International Foundation for Autonomous Agents and Multiagent Systems, 2017.

[14] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls, *et al.*, "Value-decomposition networks for cooperative multi-agent learning," *arXiv preprint arXiv:1706.05296*, 2017.

[15] M. Hüttenrauch, A. Šošić, and G. Neumann, "Deep reinforcement learning for swarm systems," *arXiv preprint arXiv:1807.06613*, 2018.

[16] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[20] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[21] H. V. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems*, pp. 2613–2621, 2010.

[22] R. M. Cichy, A. Khosla, D. Pantazis, A. Torralba, and A. Oliva, "Comparison of deep neural networks to spatio-temporal cortical dynamics of human visual object recognition reveals hierarchical correspondence," *Scientific reports*, vol. 6, p. 27755, 2016.

[23] I. Kuzovkin, R. Vicente, M. Petton, J.-P. Lachaux, M. Baciu, P. Kahane, S. Rheims, J. R. Vidal, and J. Aru, "Activations of deep convolutional neural networks are aligned with gamma band activity of human visual cortex," *Communications biology*, vol. 1, no. 1, p. 107, 2018.

[24] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[25] Y. LeCun, F. J. Huang, L. Bottou, *et al.*, "Learning methods for generic object recognition with invariance to pose and lighting," in *CVPR (2)*, pp. 97–104, Citeseer, 2004.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[28] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[29] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on robotics*, vol. 32, no. 6, pp. 1309–1332, 2016.

[30] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[31] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Fifteenth annual conference of the international speech communication association*, 2014.

[32] C. Olah, "Understanding LSTM networks." `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015.

[33] M. Hausknecht and P. Stone, "Deep recurrent Q-learning for partially observable MDPs," in *2015 AAAI Fall Symposium Series*, 2015.

[34] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[35] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines." `https://github.com/openai/baselines`, 2017.

[36] A. Hill, A. Raffin, M. Ernestus, A. Gleave, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines." `https://github.com/hill-a/stable-baselines`, 2018.

[37] D. E. J. Hobley, J. M. Adams, S. S. Nudurupati, E. W. H. Hutton, N. M. Gasparini, E. Istanbulluoglu, and G. E. Tucker, "Creative computing with Landlab: an open-source toolkit for building, coupling, and exploring two-dimensional numerical models of earth-surface dynamics," *Earth Surface Dynamics*, vol. 5, no. 5, pp. 21–46, 2017.

[38] G. A. Almeida, P. Bates, J. E. Freer, and M. Souvignet, "Improving the stability of a simple formulation of the shallow water equations for 2-D flood modeling," *Water Resources Research*, vol. 48, no. 5, 2012.

[39] M. Huettenrauch, "Deep RL for swarm systems." `https://github.com/LCAS/deep_rl_for_swarms`, 2018.

[40] M. Hüttenrauch, A. Šošić, and G. Neumann, "Deep reinforcement learning for swarm systems," *Journal of Machine Learning Research*, vol. 20, no. 54, pp. 1–31, 2019.

[41] J. Braun and S. D. Willett, "A very efficient o (n), implicit and parallel method to solve the stream power equation governing fluvial incision and landscape evolution," *Geomorphology*, vol. 180, pp. 170–179, 2013.

[42] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[43] A. Juliani, "Deep recurrent Q-network notebook." `https://gist.github.com/awjuliani/35d2ab3409fc818011b6519f0f1629df`, 2016.

[44] G. Lample and D. S. Chaplot, "Playing fps games with deep reinforcement learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[45] R. Pascanu, T. Mikolov, and Y. Bengio, "Understanding the exploding gradient problem," *CoRR, abs/1211.5063*, vol. 2, 2012.

[46] T. Rashid, M. Samvelyan, C. S. de Witt, G. Farquhar, J. Foerster, and S. Whiteson, "QMIX: monotonic value function factorisation for deep multi-agent reinforcement learning," *arXiv preprint arXiv:1803.11485*, 2018.

[47] D. Dewey, "Learning what to value," in *International Conference on Artificial General Intelligence*, pp. 309–314, Springer, 2011.

[48] C. Kozyrkov, "The simplest explanation of machine learning you'll ever read." `https://hackernoon.com/the-simplest-explanation-of-machine-learning-youll-ever-read-bebc0700047c`, 2018.

Annexes

ANNEX A

Ethical, economic, social and environmental aspects

A.1 Introduction

This thesis has focused in basic research destined to apply new technologies to a problem that, as has been argued in Section 1.1, has a great impact in society, in the economy and in the natural environment. The availability of a real-time map of a natural disaster can prevent deaths and save resources.

A.2 Description and analysis of relevant impacts

In a more general sense, the implications for society of the main technologies of interest in this project, UAVs and Artificial Intelligence (AI), are far reaching and their future impacts are hard to predict. We can analyze both fields independently.

Regarding AI and DRL, there is concern about the alignment between human goals and expected reward maximization goal [47]. As Google's Chief Decision Intelligence Engineer Cassie Kozyrkov puts it, AI allows you to automate the ineffable [48]. In this automation process, the engineer should always keep in mind that the final goal of an AI trained with RL is to maximize the expected rewards.

The alignment with human goals is only the product of careful consideration by humans to make sure that the AI only receives rewards when human goals are met. It could be argued that as AI's intelligence and capabilities increase, it might more easily find ways to be rewarded for behaviours that were not considered by the designer.

Current Machine Learning (ML) systems require several orders of magnitude more examples of that which has to be learnt than humans. Given the current rate of growth of

ML implantation and applications, the energy consumption of ML might become an issue. Researchers have to find ways to make learning more sample efficient while keeping privacy concerns in mind when using Big Data for training.

Regarding UAVs, the proliferation of drone usage has brought about safety, privacy and noise concerns. The failure and crash of a surveillance drone in a populated area can cause deaths while trying to save lives. The presence of drones by itself can have a psychological impact on people: it can cause a permanent sense of unsafety if crashes are frequent, and the same technology that allows for natural disaster surveillance allows also for the surveillance of people. The noise concern is not as serious in the application that we have considered as it is in cases where drones are flying close to residential areas on an hourly bases.

A.3 Conclusions

In conclusion, we have identified the social, economic and environmental benefits that can be obtained from the implementation of this basic research in commercial products. On the other hand, we have also identified the misuses that could be made of this technology and the concerns that should guide future research for a more beneficial ethical and social impact.

COST OF LABOUR (direct cost)	Hours	Cost/hour	Total
	900	18.00 €	16,200.00 €

COST OF MATERIAL RESOURCES (direct cost)	Buy price	Months of use	Amortization (years)	Total
Personal computer 1	4,000.00 €	6	5	400.00 €
Personal computer 2	3,000.00 €	6	5	300.00 €

TOTAL COST OF MATERIAL RESOURCES	700.00 €

GENERAL EXPENSES (indirect cost)	15%	(over DC)	2,535.00 €
INDUSTRIAL BENEFIT	6%	(over DC + IC)	1,166.10 €

FUNGIBLE MATERIAL	
Printing	40.00 €

BUDGET SUBTOTAL		20,601.10 €
APPLICABLE VAT	21%	4,326.23 €

BUDGET TOTAL	24,927.33 €

The cost of labour has been estimated assuming 1,800 effective hours of work per year, a net salary of 24,000€ and an extra cost for the employer of 35%, including social security and taxes.