Decentralized Multi-Agent Deep Reinforcement Learning in Swarms of Drones for Flood Monitoring Using DDQN

Kartik Upadhyay
B.Tech Mathematics and Computing
Pre-Final Year
IIT Ropar
2019mcb1223@iitrpr.ac.in

Gattadi Vivek
B.Tech Mathematics and Computing
Pre-Final Year
IIT Ropar
2019mcb1217@iitrpr.ac.in

I. Motivation and Problem Definition (Introduction)

Floods are the most common natural disasters, causing thousands of casualties every year in the world. In particular, flash flood events are particularly deadly because of the short timescales on which they occur. Most casualties could be avoided with advance warning, for which real time monitoring is critical. While satellite-based high resolution weather forecasts can help predict floods to a certain extent, they are not reliable enough, as flood models depend on a large number of parameters that cannot be estimated beforehand.Drone-led surveillance helps with quick anomaly or change detection. These drones can alter their flight-path mid-air to investigate a suspicious occurrence. This significantly reduces the time taken to 'spot' a potential or apparent disaster. Drones reduce the overall disaster response time by (up to) 44.46%. This time helps reduce the extent of a potential disaster. Above all, it helps save lives.

II. Literature Review (Related published works)

There is a paper published named "Decentralized Multi-Agent Deep Reinforcement Learning in Swarms of Drones for Flood Monitoring" which is trying to solve this problem. The solution to these limitations came in the form of Deep Reinforcement Learning (DRL), which relies on deep neural networks to automatically extract features from raw input and provide the implicit representation.

The use of DRL allows for the development of an end-to-end controller that maps raw input data to aircraft commands in the form of bank angle corrections.

The development of a controller requires a flood simulation model. Landlab provides an implementation of a stable and simple formulation of the shallow water equations for 2-D flood modeling by Almeida et al. 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. Regarding solution methods, we use Deep Q-Networks (DQN) and, on top of that, we explore the effect of reward sharing, which is an alternative reward scheme in swarm applications.

III. Preliminaries and Experimentation (Setup, Criteria, etc.)

A. Environment

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

1. *Terrain generation*: 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 (Braun-Willett Fastscape) modules in Landlab while the terrain is uplifted. The resulting terrain typically has a maximum relief of 10-30 m.

2. *Flood simulation*: 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 demand 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. During the episode, the terrain is fixed. The Almeida et al. model (OverlandFlow Landlab component) is run with Manning's roughness coefficient n = 0.01 $sm^{-1/3}$, a relatively low friction configuration, in order to simulate fast evolving floods.

B. Agents

The agents are fixed-wing aircraft. 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 position is updated with the kinematic model described in the following expressions:

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

where g is standard gravity, 9.8 ms^{-2}. The maximum bank angle is set to 50 degrees and actions exceeding this limit have no effect. This establishes a realistic limit to angular velocity.

C. SWARMDP Implementation

We frame the problem as a SwarMDP, which is a particularization of the Decentralized POMDP (Dec-POMDP) model. In this way, we explicitly model the multi-agent setting and the fact that all agents are equal.

Definition 1: A swarMDP is defined in two steps. First, we define a prototype A = ⟨S, A, Z, π⟩, where 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.
- Z is the set of local observations.
- π : Z→A is the local policy.

A swarMDP is a 7-tuple ⟨N, A, P, R, O, γ⟩ where:

- N is the index set of the agents, where i = 1,2,...,N indexes each of the N agents.
- A is the agent prototype defined before.
- P : S × S × A → [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×A→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. Note that R depends on the joint action vector a.
- O : S ×Z ×A → [0,1] is the observation model, where O(z'|s,a) is the probability of observing z' given that the agent is in state s and plays action a. Note that O depends on the joint action and observation vectors a and z respectively.
- $\gamma \in [0,1]$ is a discount factor, used to obtain the total reward for the agent.

*1) **State**:* The local state of each agent has several dimensions:

- The surface water depth map D_t(x,y), with 100 x 100 dimensions, updated every 10 seconds by the flood simulator.
- The position of the aircraft (x_i , y_i).
- The heading angle of the aircraft ψ_i.
- The bank angle of the aircraft ϕ_i.

*2) **Actions**:* The set of local actions is ⟨ increase bank angle by 5 degrees, decrease bank angle by 5 degrees ⟩. Each agent has to choose one of them every 0.1 s.

*3) **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 flood 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 to other aircraft $\{\rho_{ij} \mid j \in 1,2,...,N \wedge j \neq i\}=\rho_i$, the relative heading angle to other aircraft $\{\text{theta}_{ij} \mid j \in 1,2,...,N \wedge j \neq i\}=\text{theta}_i$ and the relative heading angles of other aircraft $\{\psi_{ij} \mid j \in 1,2,...,N \wedge j \neq i\}=\psi_i$.

The images are assumed to be the processed output of an optical 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 flood.

To generate the image, the surface water depth map D_t(x, y) is compared with a fixed water level threshold, here set to 5cm, 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 to compute the observation of the aircraft by sampling 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 500m.

*4) **Rewards**:* The reward corresponding to each agent i consists of the sum of four distinct components:

$$
\begin{aligned}
r_1 &= -\lambda_1 \min_{\{s \in S | F_t(s)\}} d_s, \\
r_2 &= -\lambda_2 \sum_{\{s \in S | d_s < r_0\}} 1 - F_t(s), \\
r_3 &= -\lambda_3 \phi_0^2, \\
r_4 &= -\sum_{\{j \in 1,2,\ldots,N \wedge\ j \neq i\}} \lambda_4 \exp\left(-\frac{\rho_{ij}}{c}\right),
\end{aligned}
$$

- Distance from flood (r_1): proportional to the distance to the closest flooded cell.
- Dry cells nearby (r_2): proportional to the number of dry cells 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 tuning parameters have been set to the following values:

$$\lambda_1 = 10, \lambda_2 = 1, \lambda_3 = 100, \lambda_4 = 1000, r_0 = 10, c = 100.$$

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

*5) **Deep Q Networks**:* The objective of the solution method is to find the policy which maps observations to actions optimally. The optimality criterion is determined by the reward function R that we have defined. From it, a value Q(s,a) can be deduced for each state-action pair. This value

function represents the expected accumulated reward by an agent that takes action a in state s. The Q function for the optimal policy takes the values given by the Bellman equation:

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

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

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

DQN estimates Q by training a deep neural network to approximate the function. Training is performed with the following loss function:

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

where the Bellman error e is:

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

During training, the agent faces a trade-off between exploration of the action-state space through a random policy and 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.

DQN also tackles the problem of instability inherent to DRL, 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 at the same time but at a lower update rate than the first network (with parameters θ) a different network, known as target network (with parameters θ^-), to estimate Q(s',a'), so that the first network is moving towards a fixed target. The complete algorithm is detailed in Algorithm 1:

Algorithm 1 Deep Q-Networks (DQN)

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
        Update: $\theta \leftarrow \text{Adam}((B_t - Q(s_t, a_t; \theta))^2)$
    Update: $\theta^- \leftarrow \theta$
for all $s \in \mathcal{S}$ **do**
    $\pi(s) \leftarrow \arg\max_{a' \in \mathcal{A}} \text{Predict}((s,a'), \theta^-)$
return π, θ

*6) **Network Architecture**:* The neural network treats 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 below:

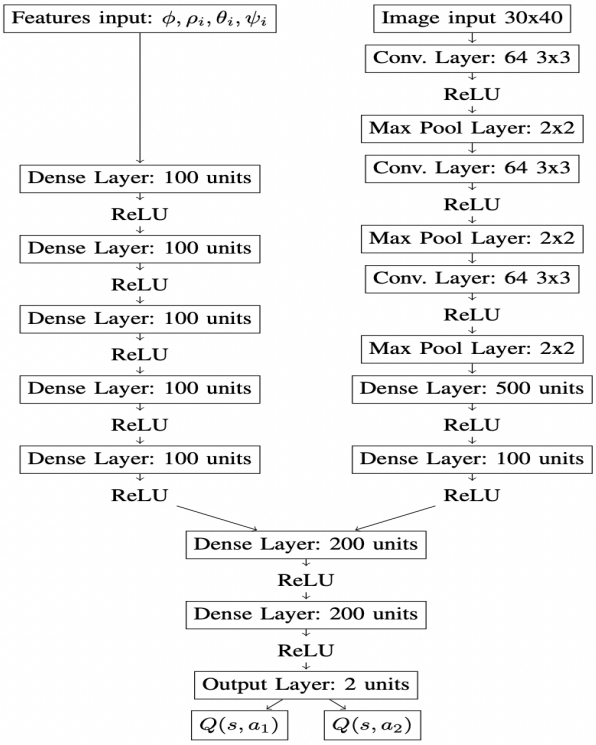


Fig.: Neural Network Architecture

*7) **Reward Configuration**:* We compare two reward schemes, which differ in the level of decentralization during training. Once trained, both approaches allow for decentralized control.

a. **Independent rewards**: Each agent receives its own reward as previously defined.

b. **Shared reward**: All agents share the same reward, consisting of the mean of the independent rewards.

*8) **Simulation Setup**:* We test both independent and shared reward configurations with two agents. Training is performed every 100 steps, once 100 new samples have been collected. Each of the training steps uses a batch size of 2000 samples, randomly taken from a buffer of the last 100000 samples. The target network is updated every 1000 steps, that is, once for every 10 action- values update. Learning only starts after the first 2000 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. The simulated episodes have a duration of 10000 steps. We use Adam as the method

for stochastic optimization and we set the learning rate to a value of $5x10^{-4}$.

IV. Solution/ Proposed Model (Algorithm, flow, methodology, architecture, equations, etc.)

In our proposed model we are using Double DQN (DDQN) [GitHub Repo] to solve the current flood problem. The reason behind using DDQN over DQN is:

- DQN approximate a set of values that are very interrelated (DDQN solves it)
- DQN tend to be overoptimistic. It will over-appreciate being in this state although this only happened due to the statistical error (Double DQN solves it)

The complete Algorithm is given below:

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q_1(s,a)$ and $Q_2(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(terminal,\cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:

$$Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \mathrm{argmax}_a\, Q_1(S',a)\big) - Q_1(S,A)\Big)$$

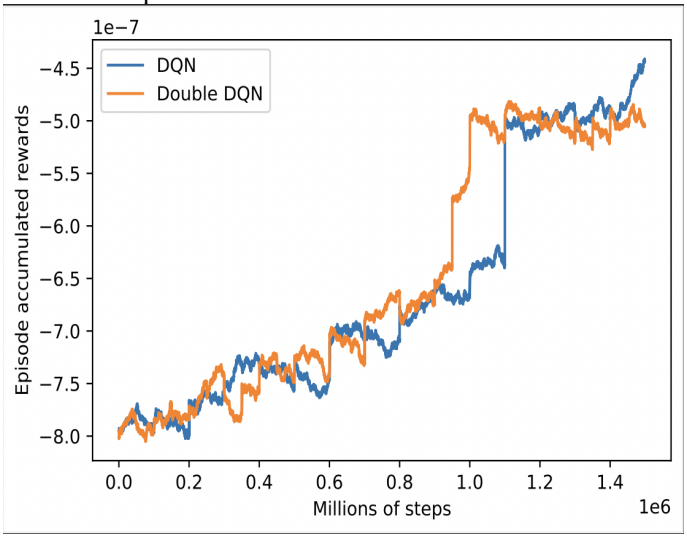
    else:

$$Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \mathrm{argmax}_a\, Q_2(S',a)\big) - Q_2(S,A)\Big)$$

    $S \leftarrow S'$
  until S is terminal

V. Results and Discussions

We have run both the algorithms. After certain no.of iterations the DDQN model has achieved convergence whereas the graph of DQN model is still going up. Below is the attached plot of the result:



VI. Conclusions and Future work

In this report we have presented an improvised version/algorithm of the existing algorithm. We offer a method for training a deep neural network capable of piloting numerous fixed-wing aircraft to monitor floods in a decentralised manner using DRL. Agents 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.

In the future, we aim to extend the environment to 3D which we have already implemented in Unity [3D Environment]. So we aim to use this technique in the 3D environment to better visualize the drone moments.

Acknowledgment

We would like to thank **Dr. Shashi Shekar Jha** for giving this opportunity to such a project and explore the domain of Reinforcement Learning. We would also like to thank **Armaan Garg** for constantly guiding and giving insights to the project.

References

[1] D. Baldazo, J. Parras and S. Zazo, "Decentralized Multi-Agent Deep Reinforcement Learning in Swarms of Drones for Flood Monitoring," 2019 27th European Signal Processing Conference (EUSIPCO), 2019, pp. 1-5, doi: 10.23919/EUSIPCO.2019.8903067.

[2] Terrain Generation Unity