

UNIVERSITÀ DI BOLOGNA



School of Engineering
Master Degree in Automation Engineering

Distributed Autonomous Systems

**DISTRIBUTED CLASSIFICATION VIA
NEURAL NETWORKS AND FORMATION
CONTROL**

Professors:

Giuseppe Notarstefano
Ivano Notarnicola

Students:

Simone Cenerini
Giulia Cutini
Riccardo Paolini

Academic year 2022/2023

Abstract

The report shows the development of two projects that aim to show the application of distributed control algorithms in different fields.

The report is organized as follows: in the first chapter we describe the distributed implementation of a binary classification task for images of handwritten digits. The aim is to run in parallel several neural networks by different agents that communicate only locally with neighbors, and thanks to a distributed algorithm we expect them to obtain the same performances on a common test set. We show different test we have tried, such as how the accuracy changes by changing the target digit or how the type of graph changes the epochs required to reach consensus among agents.

In the second chapter we describe the ROS2 implementation of the distance-based formation control law by exploiting also several possible scenarios such as the collision avoidance, the moving formation and the obstacle avoidance. We have done different test in order to validate the implementation using different target shapes both in 2d and in 3d, in by giving to the leaders different inputs, also in order to achieve a target position.

Contents

1	Distributed classification via Neural Network	5
1.1	Initial setup	5
1.2	Neural Network structure	6
1.3	Training	7
1.3.1	Gradient Tracking algorithm	8
1.4	Experiments and Results	8
1.5	First test	9
1.6	Second test	10
1.7	Third test	11
2	Formation Control	14
2.1	Distance-based formation	14
2.2	ROS2 implementation	15
2.2.1	Results	16
2.3	Collision avoidance	18
2.3.1	Results	19
2.4	Moving formation and leaders control	20
2.5	Obstacle avoidance	21
	Conclusions	24

Chapter 1

Distributed classification via Neural Network

In the first task we should classify a set of grayscale images. The `mnist` dataset, downloaded from the Keras, collects images of hand-written digits of 28×28 pixels each one, an example in figure 1.1. In order to perform our task we implemented a distributed classification where, given a predefined number of agents, each of them trains a neural network according to the Gradient Tracking algorithm. This approach ensures that consensus is reached on network weights even though these networks train only on locally available data, that is, they "see" only a fraction of the global optimization problem.

1.1 Initial setup

The first step was to prepare the dataset by reshaping and a normalizing the images (file `lib/data_load.py`). We flattened the 28×28 pixels grayscale images in order to obtain a $[784, 1]$ column vector and we normalized the pixels' intensities dividing them by a factor of 255 in order to avoid a

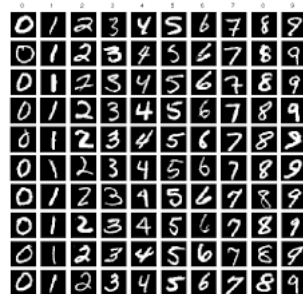


Figure 1.1: Example of images from the `mnist` dataset

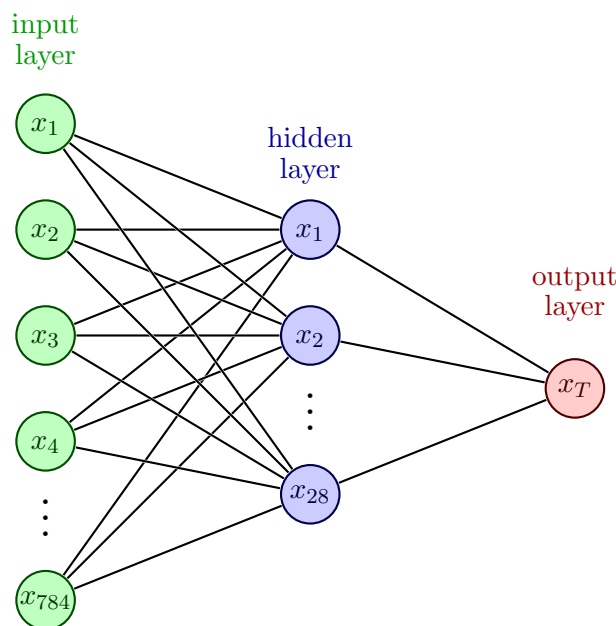


Figure 1.2: Scheme of the neural network for image classification

saturation of the activation function.

The classification problem is binary: we have to choose a digit (named as **target**) and assign the positive label (1) to the corresponding images, while, all the other images are labelled as negatives (0).

$$y_i = \begin{cases} 1, & \text{if label} = \mathbf{target} \\ 0, & \text{otherwise} \end{cases} \quad (1.1)$$

We also perform a **balancing of the dataset** by sampling the same amount of images for each class (50% positives and 50% negatives), this makes the training more efficient and stable.

1.2 Neural Network structure

The structure of our neural network is conical, that is, the deeper layers contain fewer neurons than the earlier layers, it is drawn in figure 1.2. The input layer is composed by 784 neurons, equal to the dimension of the flattened input image. The hidden layer has only 28 neurons, as more neurons were not necessary for our task, but would simply increase the computational effort required. The last layer has just one neuron since we perform a binary classification, and thus only a number between 0 and 1 is required as a prediction.

In this setting we obtain two sets of weights, one for the connections between the input and hidden layers, while the second one is for the connections between the hidden and output layers. In particular, each neuron of the generic layer L^t is connected with all the neurons of the previous layer L^{t-1} plus one connection for the bias, thus each set W^t contains $|L^t| * (|L^{t-1}| + 1)$ weights.

The activation function used to introduce non-linearities into the model is the **sigmoid** function (file `lib/network_dynamics.py`)

$$\sigma(x) = \frac{1}{1 + \exp^{-x}} \quad (1.2)$$

and its derivative is

$$\frac{d}{dx}\sigma(x) = \frac{\sigma(x)}{1 - \sigma(x)} \quad (1.3)$$

1.3 Training

The training happens as a sequence of epochs, every epoch contains multiple mini-batches that are small sets of images. For each mini-batch the computations are the following:

- Forward pass to compute the prediction;
- Loss and gradient evaluation;
- Backward pass to compute the derivative of the loss function with respect to each weight of the network.
- Weights update.

The loss function we used is the **binary cross-entropy**

$$L = -y \cdot \log(\hat{y}) - (1 - y) \cdot \log(1 - \hat{y}) \quad (1.4)$$

and its derivative with respect to the activations in the output layer, i.e. the prediction, is

$$\frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}} \quad (1.5)$$

The initial connection weights for our neural network have been randomly initialized very close to zero.

1.3.1 Gradient Tracking algorithm

The training happens in a distributed fashion: several agents run a neural network with the same structure on different training sets and, thanks to the **Gradient Tracking algorithm**, they reach consensus over the weights. To implement the Gradient Tracking algorithm we compute the new weights x^{k+1} before the forward pass, contrarily the auxiliary variable s^{k+1} is obtained after the backward pass since its computation requires the gradient computed with the new weights x^{k+1} .

$$\begin{aligned} x_i^{k+1} &= \sum_{j \in N_i} a_{ij} x_j^k - \alpha s_i^k \\ s_i^{k+1} &= \sum_{j \in N_i} a_{ij} s_j^k - \nabla f_i(x_i^{k+1}) - \nabla f_i(x_i^k) \end{aligned} \quad (1.6)$$

The adjacency matrix A from which we extract the coefficient a_{ij} is a non-negative matrix such that the entry (i, j) is $a_{ij} > 0$ if (i, j) is an edge of the graph, $a_{ij} = 0$ otherwise. In our implementation (file `lib/graph.py`) we choose the weights according to the *Metropolis Hastings* rule:

$$A_{ij} = \begin{cases} \frac{1}{\max\{d_i, d_j\} + 1}, & \text{if } (i, j) \in E \text{ and } i \neq j \\ 1 - \sum_{h \in N_i} A_{ih}, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases} \quad (1.7)$$

1.4 Experiments and Results

For testing the network we use the test set without splitting it, in this way we can check whether the agents provide equal results, given that they should reach consensus to the same weights.

The performance are then evaluated using accuracy as a metric. A threshold at value 0.5 is applied to the predictions as they are in the range $[0, 1]$.

$$\hat{y}_i = \begin{cases} 1, & \hat{y}_i > 0.5 \\ 0, & \hat{y}_i \leq 0.5 \end{cases} \quad (1.8)$$

We made different experiments to verify the functioning of the network both as the hyper-parameters such as step-size change but also as the target class or graph changes (file `lib/config.py`).

1.5 First test

In the first experiment we want to inspect the convergence of the network. In this configuration we have 5 agents communicating according to a *cycle graph* 1.3, each of them is trained on 32 mini-batches containing 8 samples each for a total of 256 images.

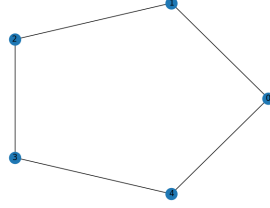


Figure 1.3: Communication graph

From the plots in 1.4 we can appreciate the evolution of the cost function and norm of the gradients, in particular it is possible to observe that the magnitude of the gradients decreases exponentially.

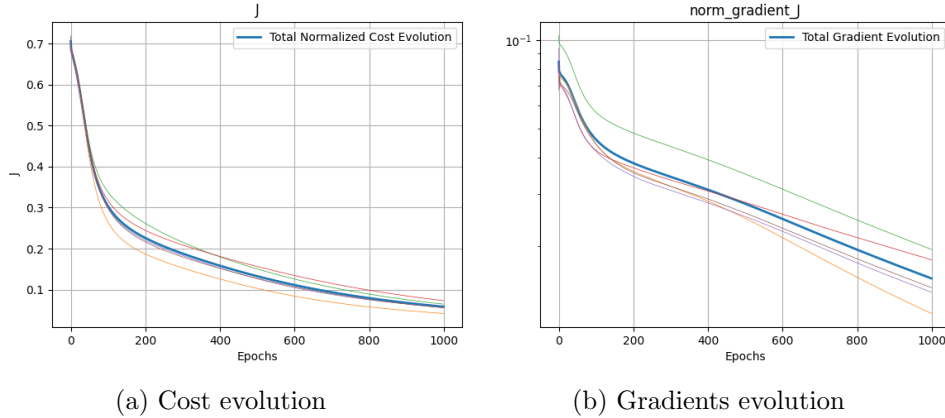


Figure 1.4: Evolution of the cost function and norm of the gradients.

After that, to see if the network converges, we take a single weight from the networks and, for each node, we plot the difference from the mean value of the weight. If the network's weights converge we expect these values to be equal for different nodes, thus they will be equal to the mean and the difference goes to zero. Indeed, from the plot 1.5(a) we can see that these differences approaches the zero value, therefore the networks converge to a consensus. However, it can also be observed that in reaching this consensus the weights of the networks oscillate, but these oscillations have smaller and smaller amplitudes 1.5(b).

In this experiment we reached an accuracy of 94.9% on the test set.

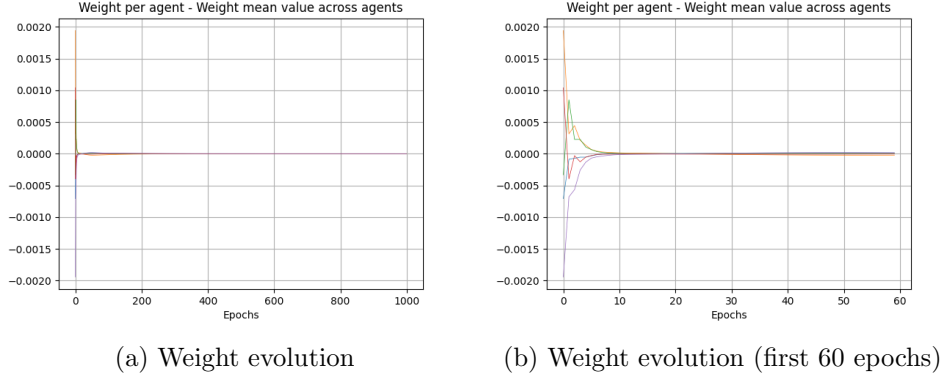


Figure 1.5: Evolution of a single weight.

1.6 Second test

In the second experiment we check whether the target digit impacts the network behaviour, in particular we use the digit 7 as a target instead of the digit 8 used in the first experiment. From the following plots 1.6 and 1.7 we notice that the decrease both in the cost and gradients magnitude is very similar across these experiments, which indicates a good generalization ability of the network, i.e., that the network is capable of distinguishing different input digits.

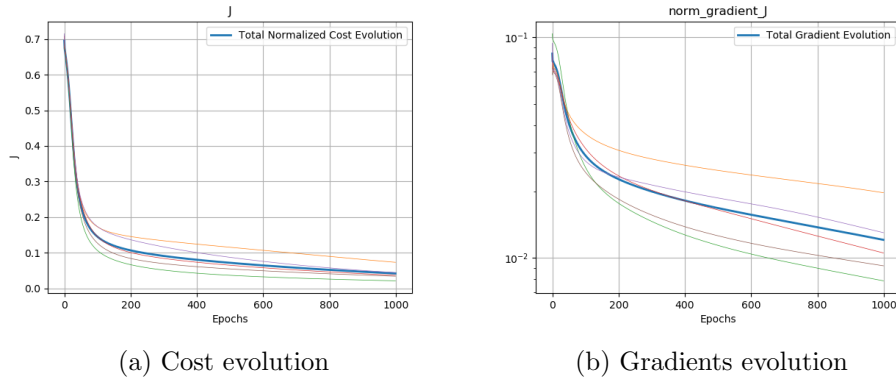


Figure 1.6: Evolution of the cost function and norm of the gradients.

We may also note a slightly slower convergence to consensus in this second experiment, in fact the amplitude of the first oscillations 1.7(b) is slightly bigger than in the first experiment.

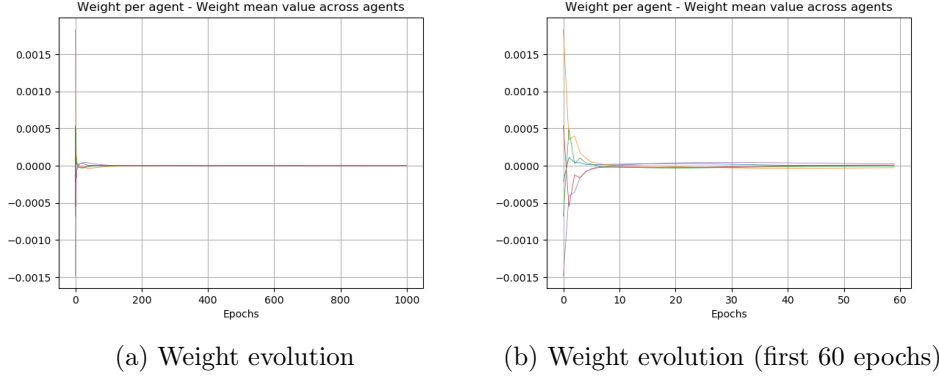


Figure 1.7: Evolution of a single weight.

We repeated the test for all the possible target digits: in the following we show the results obtained in terms of accuracy. The simulations have been obtained with 5 agents connected with a cycle graph, and a step-size of $1e^{-2}$.

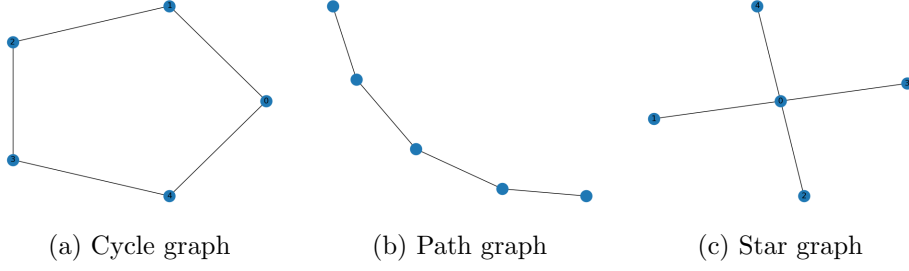
Target digit	Accuracy
0	97.66%
1	96.88%
2	94.92%
3	92.96%
4	96.87%
5	91.01%
6	95.70%
7	96.48%
8	96.48%
9	94.53%

1.7 Third test

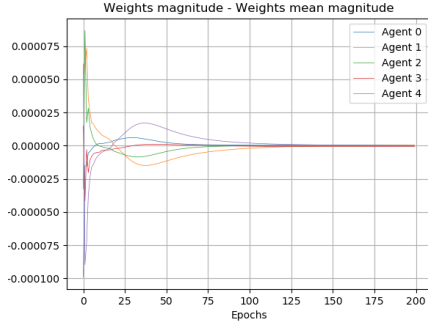
In the last test, we checked the impact of the shape of the communication graph on the epochs required to reach consensus.

We have tested three different graphs:

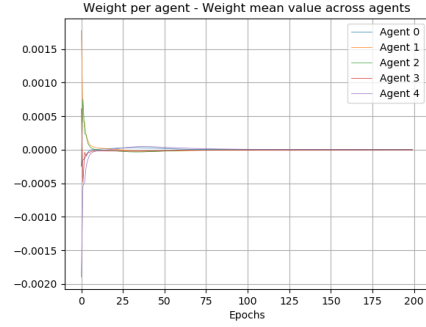
- cycle graph
- star graph
- path graph



From these experiment we would expect that fewer epochs are needed in the case of a more connected graph such as the cycle than in the case of a less connected one such as the star graph.



(a) Weights evolution

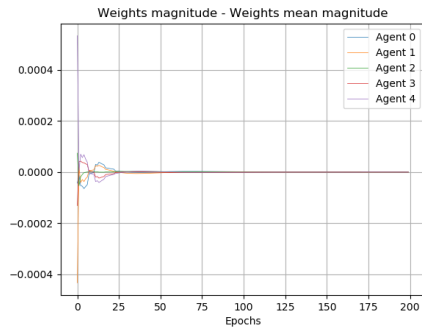


(b) Single weight (last neuron) evolution

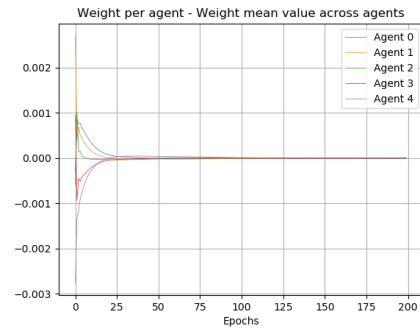
Figure 1.9: Evolution of a weights to consensus with **cycle** graph

In fact, the cycle graph is the fastest to reach consensus as the nodes have all the same "connectivity" and the information travels smoothly. This is immediately reflected in figure 1.10 where we see how the weights approach the average at approximately the same rate.

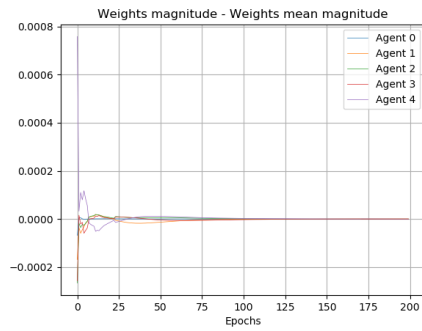
Differently, in the figures 1.10 and 1.11, representing respectively the path and star graph, we observe that the more connected nodes approach the average at a faster pace with respect to the less connected ones. In particular, the inner node of the star graph, i.e. agent 2, goes to the consensus value really fast and then drives the other agents to it. The same applies to the path graph, where inner nodes converge in few epochs while external nodes need much more time to communicate with each other, as they are 'far away' in the communication graph.



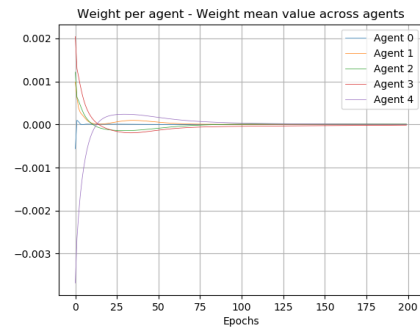
(a) Weights evolution



(b) Single weight (last neuron) evolution

Figure 1.10: Evolution of a weights to consensus with **path** graph

(a) Weights evolution



(b) Single weight (last neuron) evolution

Figure 1.11: Evolution of a weights to consensus with **star** graph

Chapter 2

Formation Control

In the second task we were asked to implement in ROS2 a discrete-time version of a **formation control law** for a team of N robots.

2.1 Distance-based formation

The aim of the task was to obtain a desired geometric formation among a group of N autonomous agents, by acting on the relative positions of agents.

The desired formation can be encoded in terms of an undirected graph, called *formation graph*, whose set of vertices is indexed by the team agents $\mathcal{N} = \{1, \dots, N\}$, and whose set of edges $E = \{(i, j) \in \mathcal{N} \times \mathcal{N} | j \in \mathcal{N}_i\}$ contains pairs of agents. Each edge $(i, j) \in E$ is assigned a scalar parameter $d_{ij} = d_{ji} > 0$, representing the distance at which agents i, j should converge to.

An example could be a predefined formation with hexagon shape with the following *distances matrix*:

$$d_{ij} = \begin{bmatrix} 0 & l & 0 & d & h & l \\ l & 0 & l & 0 & d & 0 \\ 0 & l & 0 & l & 0 & d \\ d & 0 & l & 0 & l & 0 \\ h & d & 0 & l & 0 & l \\ l & 0 & d & 0 & l & 0 \end{bmatrix} \quad (2.1)$$

The desired distance value serve as reference value for the control law. The control law computes control signals for each agent based on its current position and the position of the neighboring agents, aiming to drive the agents towards the desired formation. The neighbors of a certain agent i can be extracted from the i -th row of the distances matrix, by taking the indexes of the elements with $d_j > 0$.

By denoting the position of agent $i \in \{1, \dots, N\}$ at time $t > 0$ with $x_i(t) \in \mathbb{R}^3$, one way to approach distance-based formation control involves the usage of **potential functions**, similar to the following one:

$$V_{ij}(x) = \frac{1}{4} \left(\|x_i - x_j\|^2 - d_{ij}^2 \right)^2 \quad (2.2)$$

This kind of potential function represents the energy associated with the relative positions of the agents. By minimizing this potential function, the agents can achieve and maintain the desired formation. As a consequence, the control law we have implemented for each agent i is the following:

$$\dot{x}_i(t) = f_i(x(t)) = - \sum_{j \in \mathcal{N}_i} \left(\|x_i - x_j\|^2 - d_{ij}^2 \right) (x_i - x_j) \quad (2.3)$$

Since we were required to work in discrete time, we denote with $p_i^k \in \mathbb{R}^3$ the discretized version of $x_i(t)$ at iteration $k \in \mathbb{N}$ and, once we have computed the dynamics update for agent i we discretize it with Euler formula in the following way:

$$p_i^{k+1} = p_i^k + \Delta f_i(p^k) \quad (2.4)$$

where $\Delta > 0$ is the sampling period.

2.2 ROS2 implementation

The solution proposed in previous section has been implemented in ROS2 environment with a code written in Python language.

By means of a launch file (named `formation_control.launch.py`) we can generate the desired number of agents and give them the required informations in order to accomplish the target formation. In particular for each agent the launch file generates a ROS2 node that executes the file named `the_agent.py`. Each node gets the following main parameters from the launch file:

- id: a number from 0 to N associated to the agent's identity
- initial position, randomly generated
- distances: each agent gets only "its" row of the distances matrix. In this way it will be able to know who are its neighbors and the distances he has to keep from them in the desired configuration
- maximum number of dynamics update

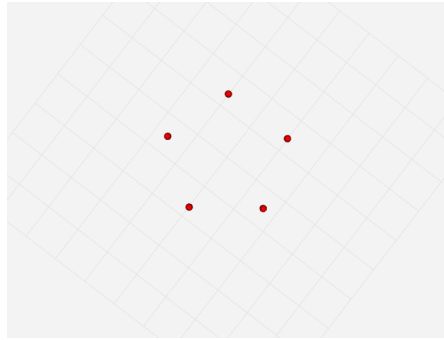
This file manages the dynamic evolution of the agent and the communication with the other ones (the neighbors' ones). For what regards the communication, we have used a publisher-subscriber protocol implemented in ROS2: each node communicates its position only to its neighbors. The communication between nodes is made using a topic for each edge defined in the formation graph. In order to write to a topic or to read from it, the node must know the name of the topic he want to access.

We have developed also a plotter agent (`the_plotter.py`) that can read the actual positions of all the other agents in order to visualize the evolution of the trajectories of the whole formation.

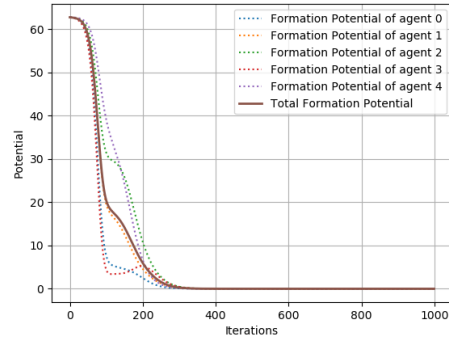
For what regards the simulation of the team behaviour we have used RViz.

2.2.1 Results

In the following we can see some results obtained during the simulation of the `formation_control` package. We have tested several formation-shapes that involve different number of agents.



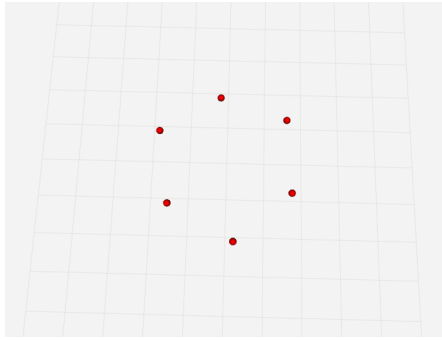
(a) RViz visualization



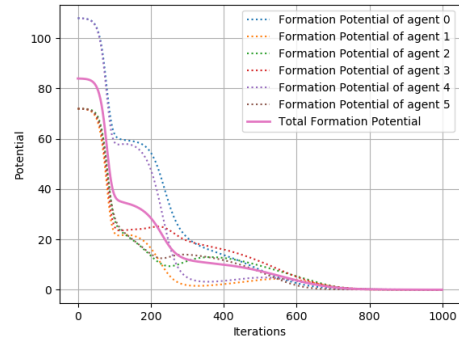
(b) Formation potential plot

Figure 2.1: Pentagon formation with 5 agents

First of all we have tried 2D-formations like a pentagon and an hexagon, the results are represented respectively in figure 2.1 and 2.2. We have initialized the agent's position randomly and close to zero for the x and the y dimensions, while z position has been initialized equal to zero. Since our target formation is planar, the z remains always equal to zero during the whole evolution. We show both the final RViz visualization and the potential evolution during the iterations: for the single agents and for the whole formation. The potential diminishes along the iterations while the agents reach the desired configuration. For these formations we have also

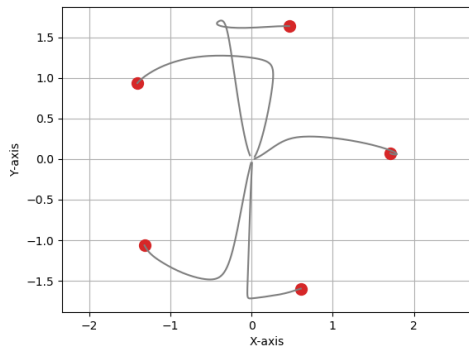


(a) RViz visualization

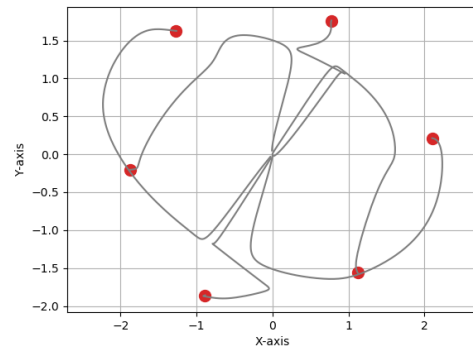


(b) Formation potential plot

Figure 2.2: Hexagon formation with 6 agents



(a) Paths for pentagon formation



(b) Paths for hexagon formation

Figure 2.3: Paths followed by agents to reach formation

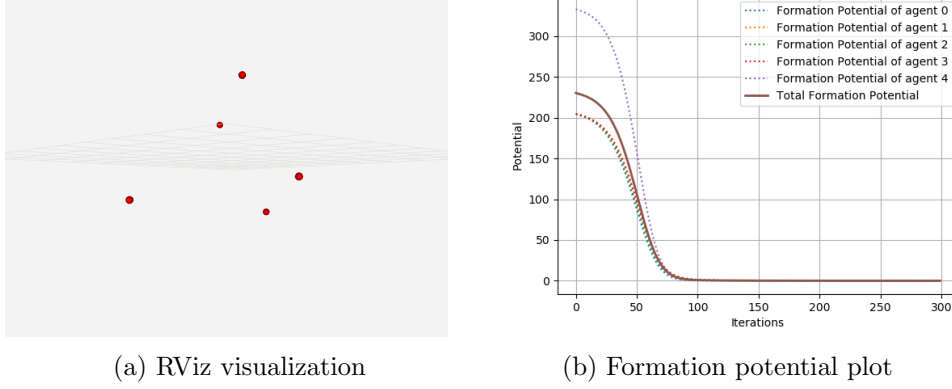


Figure 2.4: Square-based pyramid with 5 agents

represented in figure 2.3 the paths the single agents follow during the time evolution in order to accomplish the formation.

After that, we have tried a 3D-formation, a square-based pyramid. In order to do that we have initialized all the x , y and z positions close to zero. In figure 2.4 we can see the results obtained with simulations. The plane the "base-agents" chose, changes from one simulation to the others since we don't have any constraint in this sense.

2.3 Collision avoidance

In formation control, the control laws typically consists of two components: attraction and repulsion. The attraction component pulls the agents towards their desired positions relative to their neighbors, while the repulsion component prevents collision by creating a repulsive force when the agents get too close to each other.

The control law that models the agent's dynamics up to now, had just one component, the one linked to the *formation potential*, formula 2.2, this was not enough in order to avoid collision among agents (as we can see in figure 2.5).

For this reason the **task 2.2** asked to introduce a second potential to avoid agent's collision. In order to do that the following proper barrier function has been used:

$$V_{ij}(x) = -\log(\|x_i - x_j\|^2) \quad (2.5)$$

As a result, the control law has a new term due to this potential that is

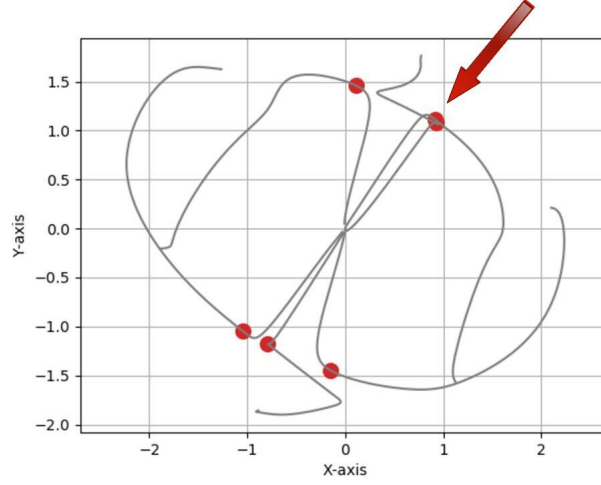


Figure 2.5: Collision among two agents during hexagon formation

the following one:

$$\dot{x}_i(t) = -2 \sum_{j \in \mathcal{N}_i} \frac{(x_i - x_j)}{\left(\|x_i - x_j\|^2 - d_{ij}^2 \right)} \quad (2.6)$$

And the total control law becomes:

$$\dot{x}_i(t) = - \sum_{j \in \mathcal{N}_i} \left(\|x_i - x_j\|^2 - d_{ij}^2 \right) (x_i - x_j) - 2 \frac{(x_i - x_j)}{\left(\|x_i - x_j\|^2 - d_{ij}^2 \right)} \quad (2.7)$$

2.3.1 Results

In figure 2.6 we can see the different shape of the paths followed by the agents in order to reach the same hexagon formation. The plot on the left shows the paths due to the dynamic evolution expressed in formula 2.3 (the one without the collision avoidance term), while the plot on the right shows the behaviour due to the introduction of collision avoidance term (formula 2.7).

In figure 2.7 we can see the evolution of the two contributions of the overall potential function. The formation potential, the attractive one that pulls the agents toward the desired formation, diminishes but we can see a "flat area" when the agents goes too close to some neighbors. Accordingly, the barrier potential that repels the agents from neighbors, has a peak.

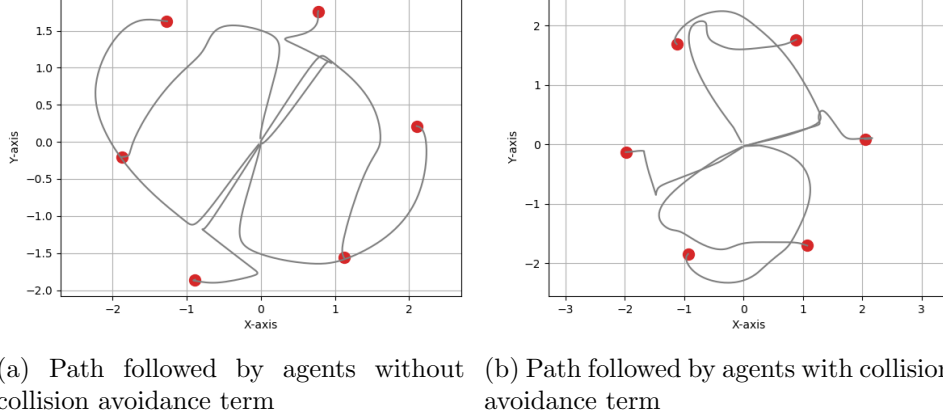


Figure 2.6: Comparison among paths with and without collision avoidance term

2.4 Moving formation and leaders control

At this point robots are partitioned in two groups, namely leaders and followers. The objective of **task 2.3** is to control the translation and/or the rotation of the formation (by giving proper inputs to the leaders) while maintaining the chosen pattern. Distance constraints are invariant to both translation and rotation of the formation, so the distance-based approach can be applied to realize translational and rotational formation maneuvers.

We have modified the `formation_control.launch.py` in order to pass to the `the_agent.py` file the information about their identity (leader or follower) and about the motion the leaders have to accomplish.

We use the `the_plotter` agent as a "supervisor" that tells all the agents when the target formation has been reached. Once all the agents are in formation the leaders start moving. As a trigger condition the supervisor checks when the potentials' derivatives of all the agents are smaller than a certain threshold. At this moment, he notifies all the agents the formation has been reached and so, the leaders can start moving. This threshold value depends on the formation shape we chose. In figure 2.8 the red line represents this notification criterion.

From this moment on, the dynamics evolution changes: the followers continue with the same motion, trying to maintain the formation, while the leader's dynamic has two contributions, the formation one and the motion we have decided them to follow. Both have also the collision avoidance term.

Figure 2.9 represents the motion of the formation along the x -axis due to a constant input assigned to leaders' dynamic. The leaders are represented

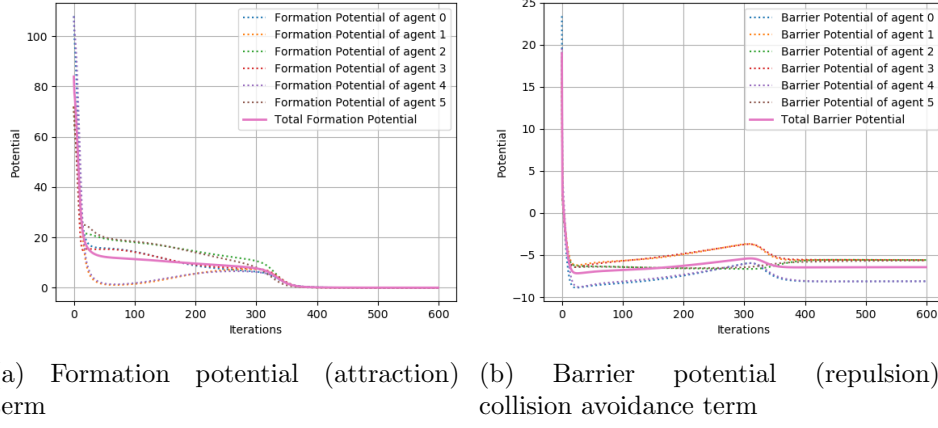


Figure 2.7: Potential contribution in hexagon formation with collision avoidance

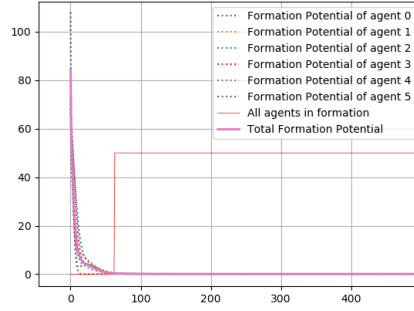


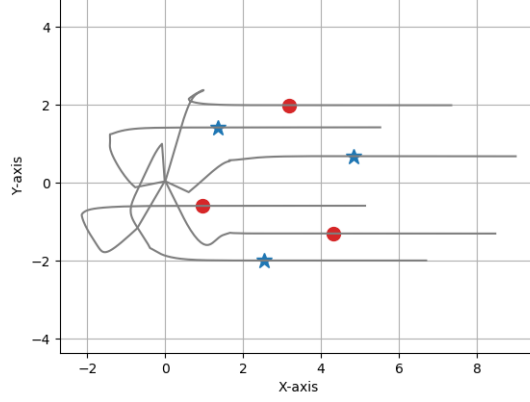
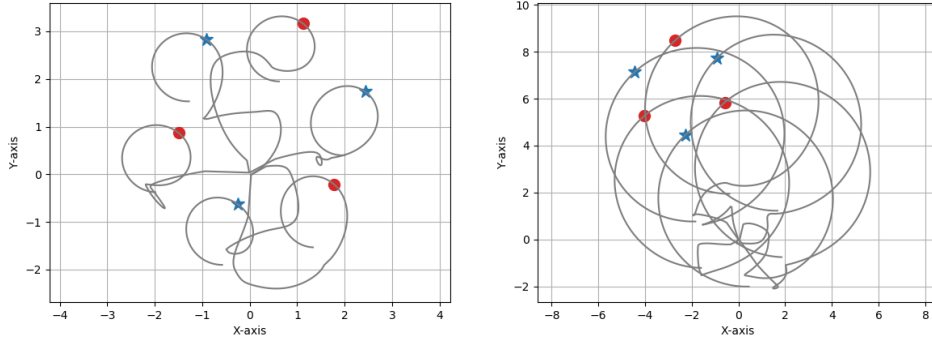
Figure 2.8: Check on the formation accomplishment

as the blue stars.

We have also tried to give the leaders a proper input in order to accomplish a circular motion during the number of iterations given. In figure 2.10 we can see the motion of the hexagon formation: in the left plot we have given as input a radius equal to 2 while in the right one the radius is equal to 5.

2.5 Obstacle avoidance

Task 2.4 asked to us to introduce obstacle avoidance in our `formation_control` package. Obstacle avoidance in formation control refers to the ability of a group of N coordinated agents to navigate through an environment while maintaining a desired formation and avoiding collision with fixed or moving obstacles.

Figure 2.9: Motion of the formation along the x -axis

(a) Moving along a circle of radius 2 (b) Moving along a circle of radius 5

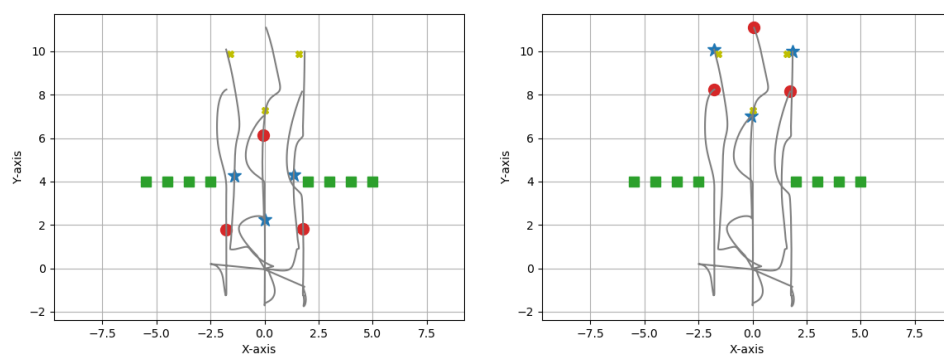
Figure 2.10: Motion of the formation along a circular path

In order to accomplish this task we have created a certain number of fixed obstacles as ROS2 nodes. The obstacles get their position from the launch file and communicate it to the follower agents. In `the_agent.py` we have still two dynamics: the leaders move according to the assigned input while the followers move according to the formation and trying to avoid obstacles. For this reason we have introduced in the dynamics a potential in order to avoid obstacles:

$$V_{ij}(x) = -\log(\|x_i - x_h\|^2) \quad (2.8)$$

where x_h denotes the position of the obstacles.

In the following simulation we have given to the leaders of the formation a target position "over" the obstacles: to reach it the formation has to pass inside the window left by the obstacles.



(a) Motion of formation to pass in (b) Formation that reaches target position between obstacles

Figure 2.11: Obstacle avoidance

Conclusions

In the **First task** we explored a distributed optimization framework where each individual agent possesses limited knowledge of the global optimization problem. To tackle the classification problem, agents rely solely on local computations and communication with their neighboring agents, without the involvement of a central unit. In order to solve this problem we have implemented a Gradient Tracking algorithm. We have obtained good results both in terms of consensus and accuracy of the classification.

In the **Second task** we have seen the power of a ROS2 implementation of formation control for a network of N robotic agent, by applying a distance-based control law, and by allowing only local communication with neighbors. We have also seen how the agents are able to follow a predefined trajectory maintaining the formation very well. We have also explored the possibility of introduce obstacle and collision avoidance between agents and it seems to work very well.