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

Tutor:
Lorenzo Pichierri

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 systems in different fields.

The report is organized as follows: in the first chapter we describes the distributed implementation of a binary supervised 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, the *Gradient Tracking algorithm* we expect them to obtain the same performances on a common test set. We show different test we have performed, such as how the accuracy changes by changing the target digit or how the type of connection graph among agents influences the number of epochs required to reach consensus.

In the second chapter we describe a ROS2 implementation of the distance-based formation control law by exploiting several real possible scenarios. The first test has been to obtain several different chosen shape as formation for our team of robots, both in 2D and in 3D. Then we have introduced collision avoidance among the agents and tried to steer the formation according to a certain assigned motion or toward a target position. In order to do that, we have defined proper control laws to give to the leaders of the formation the required input to accomplish the task. The simulations have been showed by means of RViz.

Contents

Motivations

Distributed systems play a crucial role in today's connected world where we need efficient and scalable solutions. Our project is motivated by the need to address complex challenges in machine learning and robotics through the power of distributed systems, in which agents aim at cooperatively solving complex tasks by local computation and communication.

The primary motivation behind our work is to leverage the collective computational resources of multiple agents to accelerate the training of neural networks. By dividing the work and doing tasks in parallel, our system helps the networks learn faster and reduces the time needed for training. This is especially useful when dealing with large amounts of data and complicated models that require a lot of computing resources.

In addition to enlarge computational capabilities, distributed systems offer inherent fault tolerance and resilience. Decentralizing computation and data storage reduces the risk of single points of failure, ensuring the system's robustness. This is crucial in critical applications.

Scalability is another key motivation for exploring distributed systems. As datasets grow exponentially, centralized systems often struggle to handle the increasing computational demands. Distributed systems can handle large amounts of data by easily adding more agents to the network, allowing efficient processing.

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, with a value between 0 (black) and 255(white), 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 an own neural network. To make possible to the whole cyber-network to converge to a common value over the connection weights we have implemented a distributed Gradient Tracking algorithm. This approach ensures that consensus is reached 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 normalizating the images (file `lib/data_load.py`). We flattened the 28×28 pixels grayscale

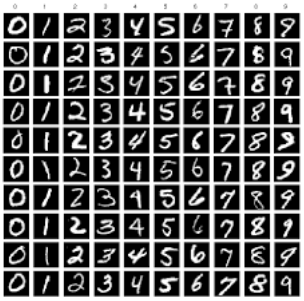


Figure 1.1: Example of images from the `mnist` dataset

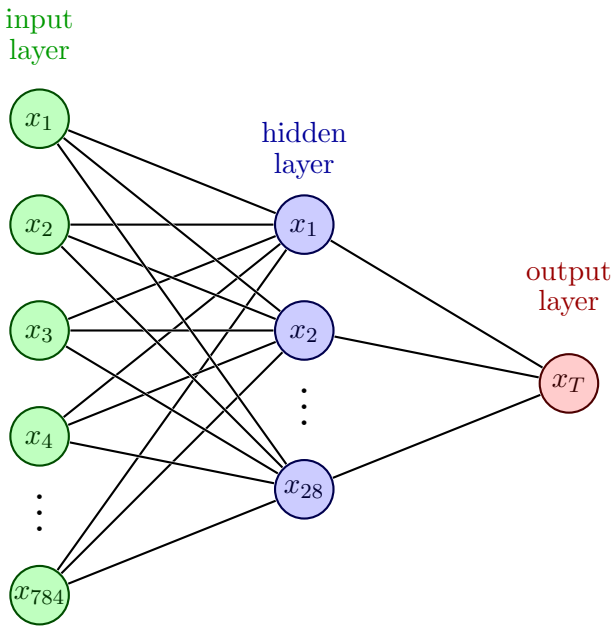


Figure 1.2: Scheme of the neural network for image classification

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 saturation of the activation function.

The classification problem we had to address 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).

$$y_i = \begin{cases} 1, & \text{if label} = \texttt{target} \\ 0, & \text{otherwise} \end{cases} \tag{1.1}$$

We have also performed 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 prediction.

With respect to the structure shown in figure 1.2 we have added the bias (a "fake" neuron) in the code implementation.

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 $|L^t| \cdot (|L^{t-1}| + 1)$ weights, where W^t is the set of weights associated to the t layer.

In a neural network, the activation function is the following mathematical function:

$$x_l^+ = \sigma(x^T u_l + u_{l_0}) \tag{1.2}$$

with $\sigma : \mathbb{R} \to \mathbb{R}$.

Notation: l identifies a neuron, a single computational unit, $u_l \in \mathbb{R}^{d+1}$ is the set of weights associated to it, with the bias (u_{l_0}), and x is the "state" vector the neuron updates thanks to the activation function.

The activation function introduces non-linearity to the neuron's output, allowing the network to learn and model complex relationships between inputs and outputs.

The activation function we used in this project is the **sigmoid** function (file `lib/network_dynamics.py`)

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}} \tag{1.3}$$

and its derivative is:

$$\frac{d}{d\xi}\sigma(\xi) = \frac{\sigma(\xi)}{1 - \sigma(\xi)} \tag{1.4}$$

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

1.3 Training

The training happens as a sequence of epochs, every epoch contains multiple minibatches 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**

$$J = -y \cdot \log(\hat{y}) - (1 - y) \cdot \log(1 - \hat{y}) \tag{1.5}$$

where $\hat{y}$ is the prediction, i.e. the output value of the neuron of the output layer.

Its derivative with respect to the prediction $\hat{y}$, is:

$$\frac{\partial J}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}} \tag{1.6}$$

1.3.1 Gradient Tracking algorithm

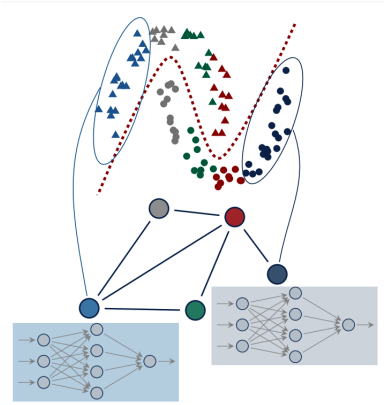


Figure 1.3: Scheme of how distributed learning of neural networks works

The training happens in a distributed fashion: several agents run a neural network with the same structure on different training sets and communicate with neighbors according to a certain communication graph. A scheme can be seen in figure 1.3

In a distributed computation, the communication topology is an important part of the problem. The communication between agents is modeled through a graph G with N nodes, one for each agent. Since in our application we are working with undirected graphs, we can say than an agent i can communicate with another agent j if the graph G contains an edge connecting i to j (or j to i).

Thus, in agent's networks, the goal is to design a proper distributed algorithm, such as the **Gradient Tracking algorithm**, that allows multiple agents in the network to collectively solve an optimization problem by tracking the gradient information of a global objective function. The optimization problem we need to solve is the following:

$$\min_{\mathbf{u}} \sum_{i=1}^{I} J_i(\mathbf{u}) \tag{1.7}$$

where $\mathbf{u}$ is a common decision variable, while the cost function (expressed in formula 1.5) is the sum of N local contributes

If we suppose that our optimization cost coupled problem has an optimal solution $\mathbf{u}^*$, we want our optimization algorithm to make all agents update their own local estimates $\mathbf{u}_i^k$ in such a way to converge to the optimal consensual value.

The formula of the algorithm are the following:

$$\begin{aligned} u_i^{k+1} &= \sum_{j \in N_i} a_{ij} u_j^k - \alpha s_i^k, & u_i^0 &\in \mathbb{R} \\ s_i^{k+1} &= \sum_{j \in N_i} a_{ij} s_j^k - \nabla f_i(x_i^{k+1}) - \nabla f_i(x_i^k), & s_i^0 &= \nabla f_i(x_i^k) \end{aligned} \tag{1.8}$$

where the variable s_i^k is a local estimate used as an estimation of the local descent direction, and u_i^k is the local update of the connection weights.

The assumptions we need to satisfy in order the algorithm to converge are the following:

- *Ass.1*: Let a_{ij}, $i, j \in \{1, \dots, N\}$ be non-negative entries of a weighted adjacency matrix A associated to the undirected and connected graph G, with $a_{ii} > 0$ and A doubly stochastic.
- *Ass.2*: For all $i \in \{1, \dots, N\}$ each cost function $f_i : \mathbb{R}^d \to \mathbb{R}$ satisfies the following conditions: strongly convex with coefficient $\mu > 0$, and Lipschitz continuous gradient with constant $L > 0$.

Thanks to these assumptions it can be shown that there exists a $\alpha^* > 0$ (constant stepsize) such that for all $\alpha \in (0, \alpha^*)$, the sequence of local solution estimates $\{x_i^k\}_{k \geq 0}$ asymptotically converge to a (consensual) solution $\mathbf{u}^*$ of the problem.

The adjacency matrix A from which we extract the coefficient a_{ij} is, as previously said, a non-negative matrix such that the entry (i, j) is $a_{ij} > 0$ if (i, j) is and 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:

$$A_{ij} = \begin{cases} \frac{1}{\max\{d_i, d_j\}+1}, & \text{if } (i,j) \in E \text{ and } i \neq j \\ 1 - \sum_{h \in N_i} A_{ih}, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases} \tag{1.9}$$

In our implementation of the Gradient Tracking algorithm, applied to the federated learning of a neural network, 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}.

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

$$\hat{y}_i = \begin{cases} 1, & \hat{y}_i > 0.5 \\ 0, & \hat{y}_i \leq 0.5 \end{cases} \tag{1.10}$$

where i indicates a single image, and $\hat{y}_i$ its prediction value.

A prediction is said to be correct if the label is equal to $\hat{y}$, incorrect otherwise. The accuracy has been computed as the ratio between the number of correct predictions of an agent over the the total number of tests.

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`). The results are reported in the following.

1.5 First test: efficiency of the neural network

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* 1.4, each of them is trained on 32 mini-batches containing 8 samples each for a total of 256 images. We have trained our network over 5000 epochs with a constant stepsize equal to $1e^{-4}$.

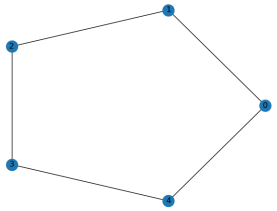
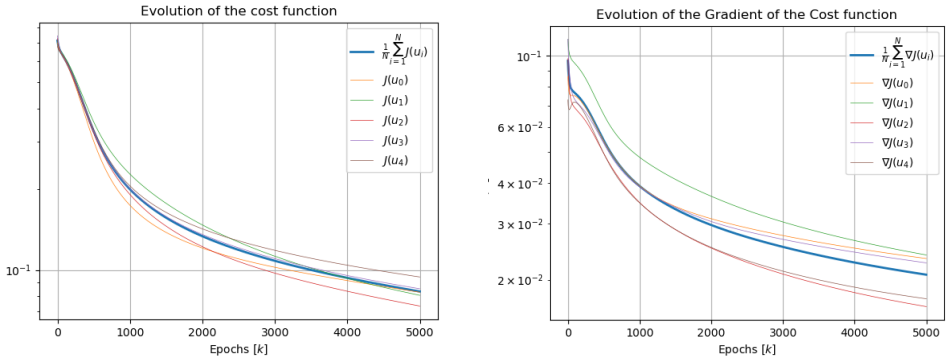


Figure 1.4: Communication graph

From the plots in 1.5 we can appreciate the evolution of the cost function and the norm of the gradients, both on a logaritmic scale. We have decided to plot both the evolution of the mean values over the agents, and the single agent's values (dotted lines). In particular it is possible to observe that the magnitude of the gradients decreases exponentially over the epochs.



(a) Cost evolution in log-scale

(b) Gradient evolution in log-scale

Figure 1.5: Evolution of the cost function and of the gradient.

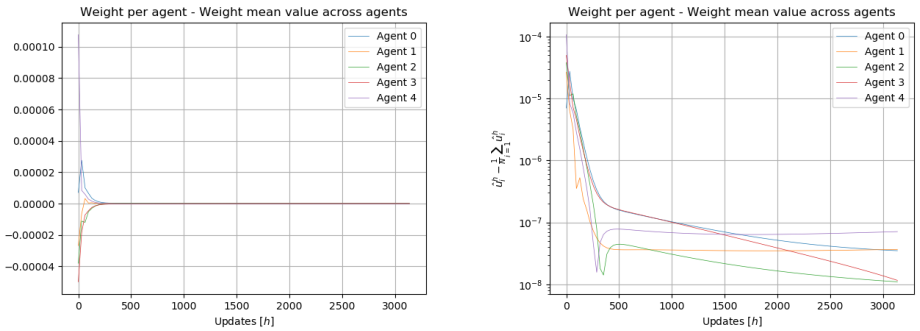
After that, to check if the network reaches the consensus, we choose the weight associated to a single connection (in particular the one that connects the first neuron of the hidden layer with the neuron of the output layer, called u_c). We have decided to plot, in order to have a more readable graph, the difference between the value of this connection and its mean value across

all the agents.

$$\text{Error} = u_{ci} - \frac{1}{N}\sum_{i=1}^{N} u_{ci} \tag{1.11}$$

Theoretically, at consensus, this value should be equal to to zero. Indeed, from the plot 1.6 we can see that this difference approaches the zero value, therefore the networks converge to a consensus. This plots have been done over 3000 updates, in fact, since we are working with the minibatches, each epoch corresponds to a number of updates equal to the number of a minibatches.

In this experiment we reached an accuracy of 97.6% on the test set.



(a) Consensus error evolution (b) Consensus error evolution in log-scale

Figure 1.6: Consensus error evolution

1.6 Second test: different digits

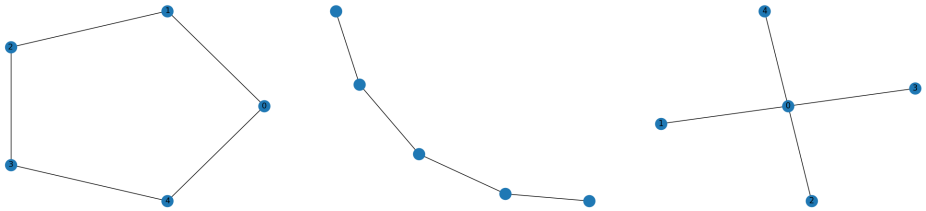
In the second experiment we check whether the target digit impacts the network behaviour, in particular we repeat the previous test for all the possible target digits. The following results are obtained over a reduced number of epochs in order to train the network in a quite short time and, as a consequence a bigger stepsize equal to $1e^{-2}$.

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%

The table highlights own the simplest digit to be recognized is 0 while the most difficult is 5. Nevertheless the accuracy obtained are very good in all the several simulations.

1.7 Third test: impact of graph shape on consensus

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



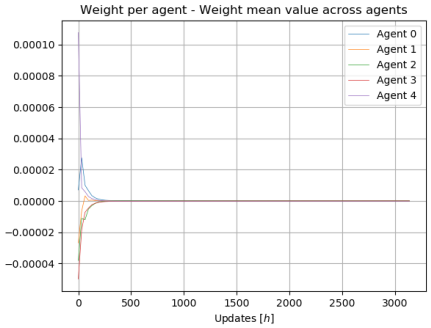
(a) Cycle graph (b) Path graph (c) Star graph

Figure 1.7: Different graphs checked

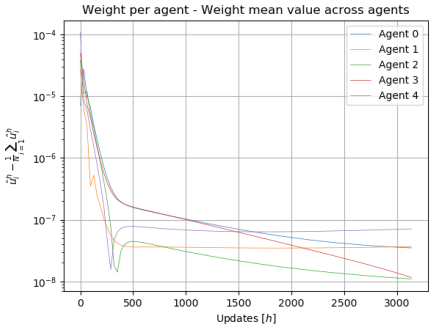
We have tested three different graphs represented in figure 1.7:

- cycle graph
- star graph
- path graph

From these experiments 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.



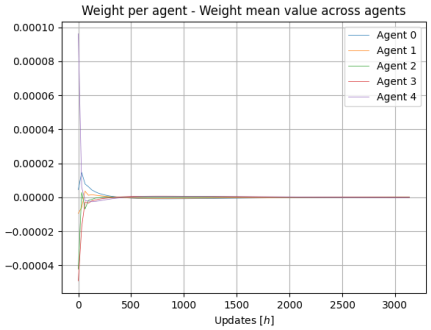
(a) Consensus error evolution



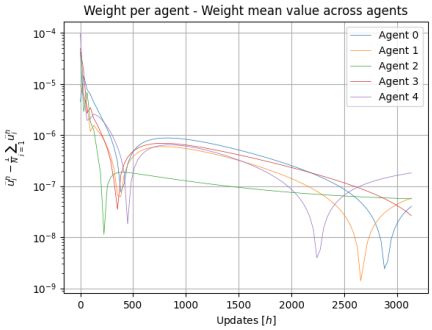
(b) Consensus error evolution in log-scale

Figure 1.8: Evolution of a weights to consensus with **cycle** graph over 100 epochs

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 1.8 where we see how the weights approach the average at approximately the same rate. Considering as a threshold value to say that the consensus is reached 10^{-7}, in this training this value is reached in about 1000 updates.



(a) Consensus error evolution

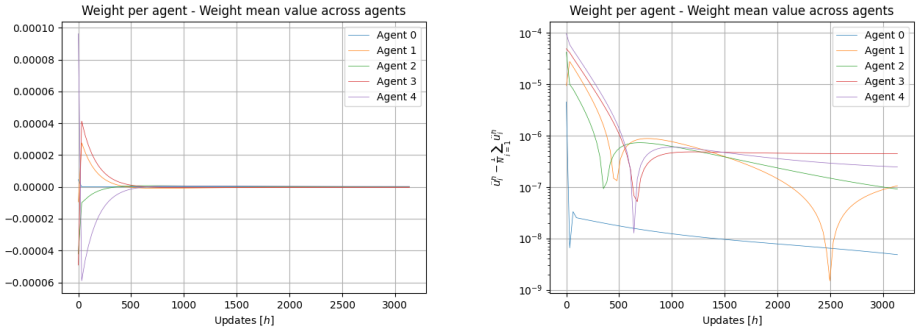


(b) Consensus error evolution in log-scale

Figure 1.9: Evolution of a weights to consensus with **path** graph over 100 epochs

In case of the path graph, less connected than the cycle, the value of 10^{-7} requires more updates to be reached.

Differently, in the figure 1.10, representing the star graph, we observe that the more connected node (the centre of the star, agent 0) approach the average at a faster pace with respect to the less connected ones. In particular, the inner node of the star graph.



(a) Weights evolution (b) Single weight (last neuron) evolution

Figure 1.10: Evolution of a weights to consensus with **star** graph over 100 epochs

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*:

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

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 $i \in \{1, \cdots, N\}$ 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:

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

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:

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

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:

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

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

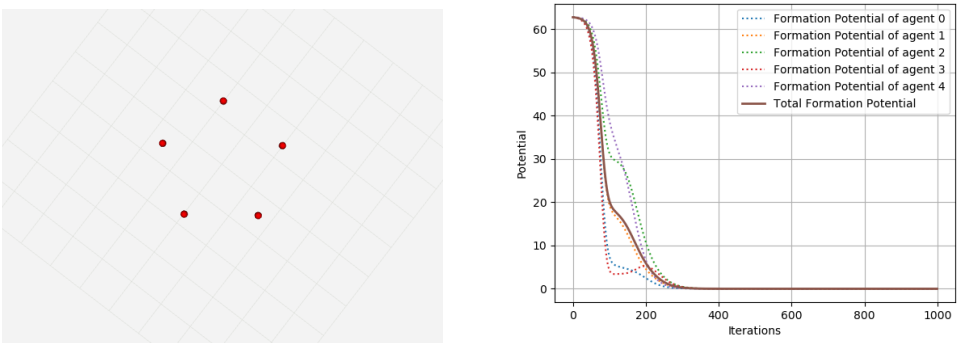
This file manages the dynamic evolution of the agent and the communication with the other ones (the neighbors' ones). For what regards the communication, we have used a publisher-subscriber protocol implemented in ROS2: each node communicates its position only to its neighbors. The communication between 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 it 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.

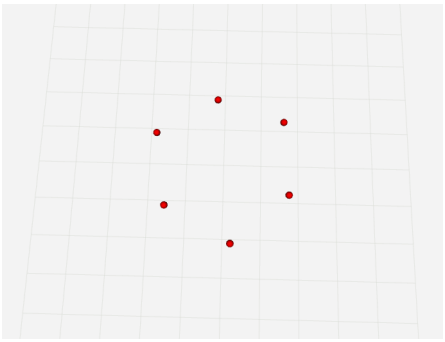


(a) RViz visualization

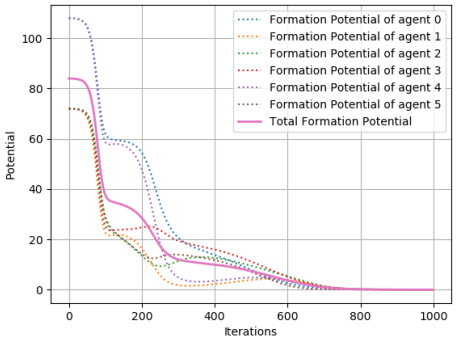
(b) Formation potential plot

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 positions 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

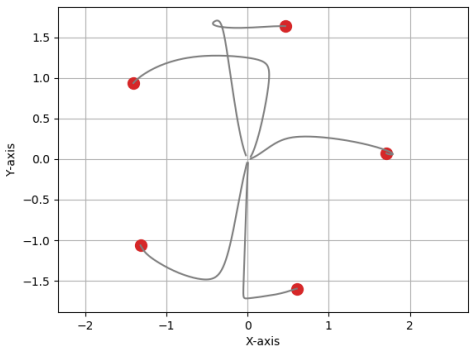


(a) RViz visualization

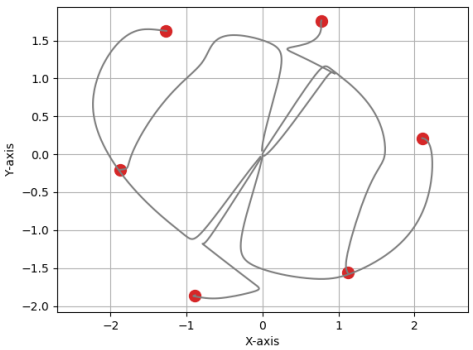


(b) Formation potential plot

Figure 2.2: Hexagon formation with 6 agents

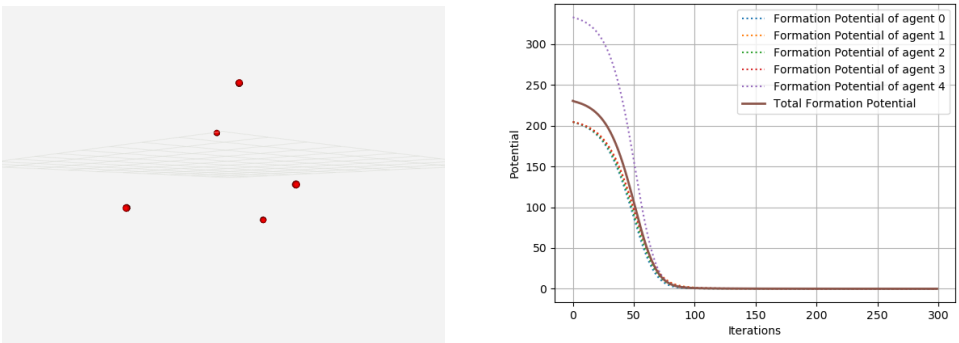


(a) Paths for pentagon formation



(b) Paths for hexagon formation

Figure 2.3: Paths followed by agents to reach formation



(a) RViz visualization (b) Formation potential plot

Figure 2.4: Square-based piramid 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 piramid. 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:

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

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

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

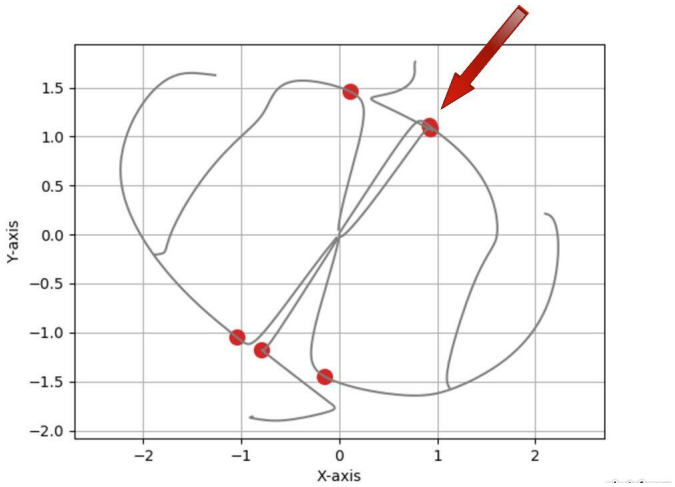
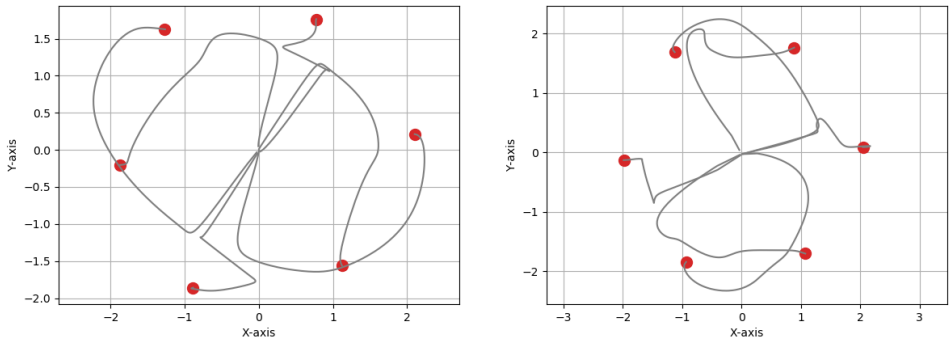


Figure 2.5: Collision among two agents during hexagon formation

And the total control law becomes:

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

2.3.1 Results



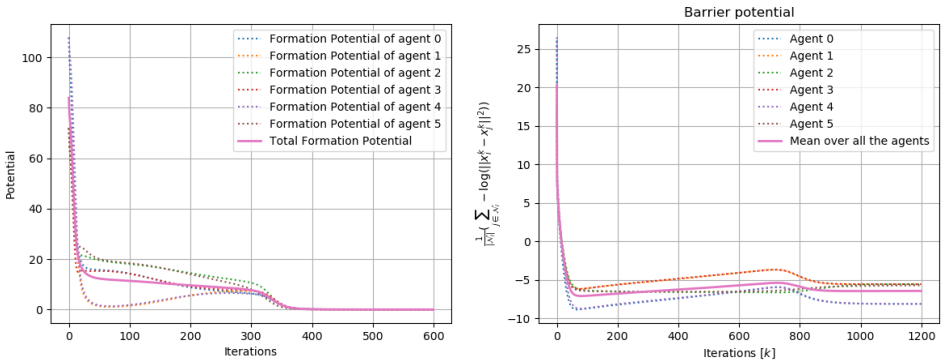
(a) Path followed by agents without collision avoidance term

(b) Path followed by agents with collision avoidance term

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

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).



(a) Formation potential (attraction) term

(b) Barrier potential (repulsion), collision avoidance term

Figure 2.7: Potential contribution in hexagon formation with collision avoidance

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.

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 constrains 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 then a certain threshold. At this moment, he notifies all the agents the formation

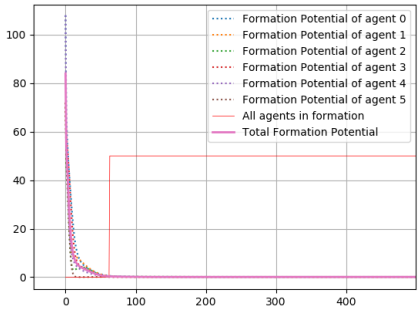


Figure 2.8: Check on the formation accomplishment

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.

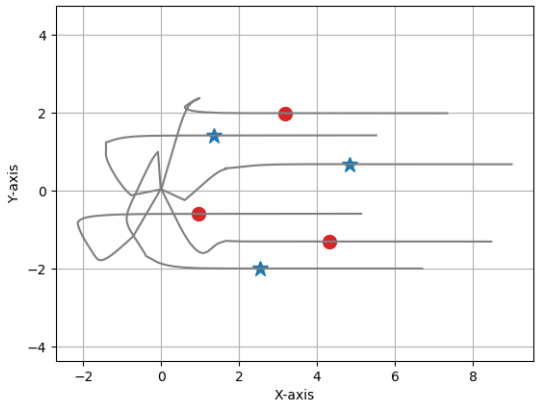
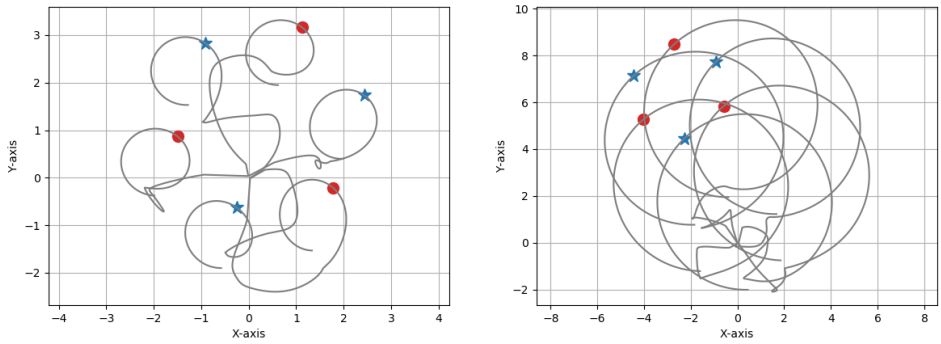


Figure 2.9: Motion of the formation along the x-axis

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



(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

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 en environment while maintaining a desired formation and avoiding collision with fixed or moving obstacles.

In order to accomplish this task we have created a certain number of fixed obstacles as ROS2 nodes. The obstacles gets 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:

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

where x_h denotes the position of the obstacles.

In the following simulation we give to the leaders of the formation a target position (indicated in the plot with the yellow x symbol) "over" the obstacles: to reach it the formation has to pass inside the window left by the obstacles. The target position is assigned to the leaders in this way: when the formation in reached, the leader compute their target position as in the following:

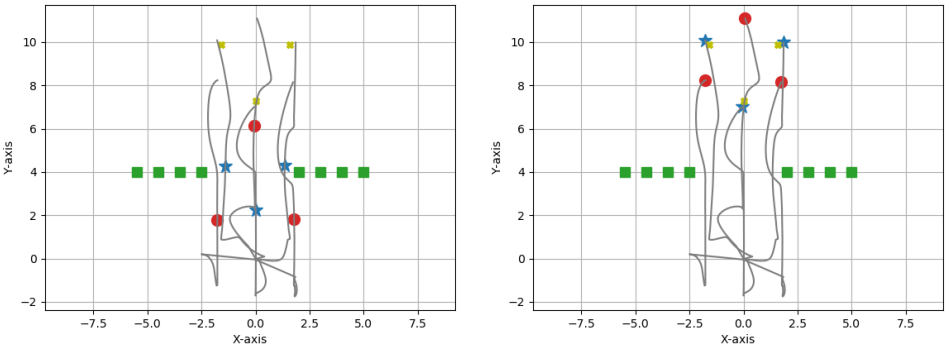
$$x_{i,tar} = x_i(\bar{t}) + h_y, \ i \text{ are leaders} \tag{2.9}$$

where $\bar{t}$ is the time instant in which the formation is reached, and h_y is a proper distance (in this case along y direction). In our simulation we chose $h_y = 7$ in order to make the formation move toward the obstacles.

In order to steer the leaders, and a consequence the whole formation, toward the target position, we have use the following proportional controller:

$$\dot{x}_i(t) = k \cdot (x_{i,tar} - x_i(t)), \ i \text{ are leaders} \tag{2.10}$$

where k is a proper proportional gain.



(a) Motion of formation to pass in between obstacles

(b) Formation that reaches target position

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. Future additional developments could be to try the formation control according to a bearing-based tecnique in such a way to have also scale invariance in addition to translation and rotation one.