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
Lorenzo Pichierri

Students:
Luca Santoro
Armando Spennato

Academic year 2022/2023

Abstract

The following project consists in two main task. The first one is related to a binary classification on the image dataset mnist (handwritten digits). The problem is divided in different parts. Is considered first a general consensus optimization problem (distributed optimization), supposing to have a team of N agents and that each agent can access only to a private subset of it, then a centralize training and at the end its distributed version. The distributed training is based on the Gradient Tracking method. At the end, to evaluate the quality of the solution obtained, the confusion matrix, accuracy and other useful metrics are calculated. The second task deals with the control for multi-robot systems in ROS2. A potential-based formation control law is implemented. Barrier functions are also used. The proposed solution involves avoiding collisions between agents and obstacles, as well as achieving the specified desired formation shape and reaching a target point. The effectiveness of the implemented solution is tested for multiple desired 2D/3D formations, with different numbers of agents and choosing different types of target points. Simulations are shown using Rviz. Since with many agents the simulation times are long, the project also includes folders with the simulation data, in order to visualize the desired formation and the trajectories of the agents in less time with mathplotlib.

Contents

Chapter 1

TASK 1 - Distributed Classification via Neural Networks

1.1 Introduction

This task concern the classification of a set of images. In particular, handwritten digits (Mnist). The whole solution of the first task is organized in three points. In the first point we start implementing a the *Gradient Tracking* algorithm to solve a consensus optimization problem (distributed optimization). The performances of this algorithm are compared with those of the *Distributed Gradient* method in order to highlight the advantages of the GT. Several simulations are done to test the implemented algorithms with different weighted graphs. Regarding the following point we implement a centralized multi-sample neural network training. Our goal is to detect a selected digit. After testing the actual functioning of the network we move on to the final point of our first task. In this part the training set is randomly split in N subsets, one for each agent i. We implement a distributed neural network training exploiting the Gradient Tracking algorithm. Then at the and the performances are evaluated exploiting some metrics as the confusion matrix, accuracy etc.

The main concepts to accomplish these tasks are the neural network and the network of the N agents (graph). The basic element of a neural network is the neuron. A generic neuron l is a computational unit that:

- has a set of weights $u_l \in \mathbb{R}^d$
- elaborates a vector $x \in \mathbb{R}^d$
- computes a scalar quantity

$$x_l^+ = \sigma(x^T u_l) \tag{1.1}$$

with $\sigma : \mathbb{R} \to \mathbb{R}$ being the *Activation Function*.

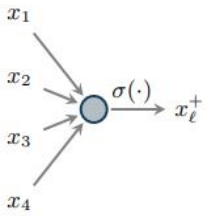


Figure 1.1: Example of a neuron processing a vector in $x \in \mathbb{R}^4$.

Several neurons can be arranged together in order to obtain a "neural network "as shown in figure 1.2.

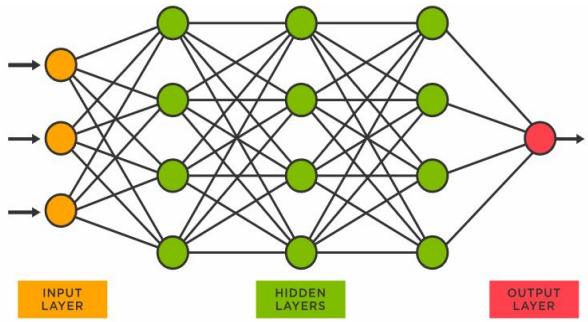


Figure 1.2: Example of a neural network.

The network of agents is represented by a graph. In particular, the graph is mathematical structure used to model pairwise relations between objects. An example is reported in figure 1.3.

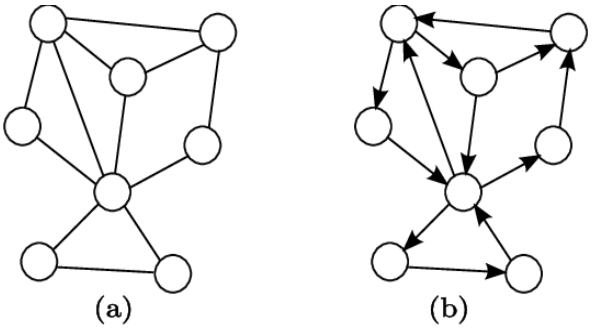


Figure 1.3: Example of graph: (a) Undirected graph, (b) Directed graph.

1.2 Distributed Optimization

In this section we want to solve a general a *Consensus Optimization Problem*, in the form:

$$\min_{u \in \mathbb{R}^d} \quad \sum_{i=1}^{N} J_i(u) \tag{1.2}$$

1.2.1 Cost-Coupled Optimization

More in detail the previous set-up is called *Cost-Coupled Optimization*. The cost function is expressed as sum of the local contribution J_i and all of them depend on a a common optimization variable u. The $J_i(u)$ is a quadratic function. We are in a distributed optimization scenario, we have a network of N agents communicating according to a digraph $G = (N, E)$. One agent (node) of the network only knows one piece of the optimization problem (only a portion of the cost function) and cooperates with other agents to compute a solution. Let u^* denote an optimal solution of problem 1.2. The goal is to design a distributed algorithm where each agent updates a local estimate u_i^k that converges (asymptotically or in finite time) to u^* through local computation and neighboring communication only, as shown in figure 1.4. So we want to reach consensus on an optimal solution of the problem.

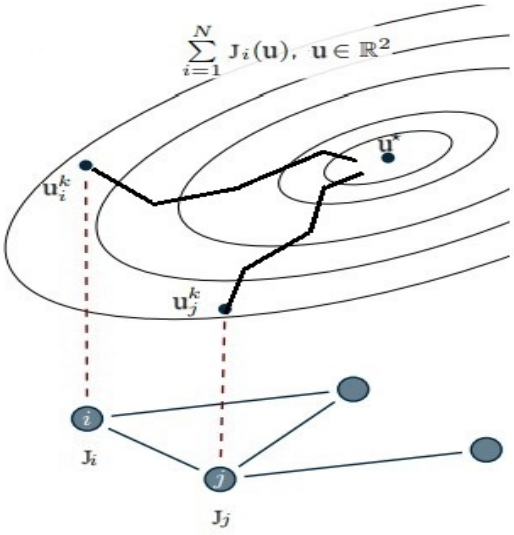


Figure 1.4: Algorithm Goal.

1.2.2 Distributed Gradient Tracking method vs Distributed Gradient method

The previous optimization problem 1.2 can be solved by exploiting two main methods:

- **Distributed Gradient method**

  The Distributed Gradient method is based on the gradient descent algorithm, which is a first-order optimization algorithm that uses the gradient of the cost function to update the parameters in the direction of steepest descent [1]. In explicit way :

$$v_i^{k+1} = \sum_{j \in \mathcal{N}_i} a_{ij} u_j^k \tag{1.3}$$

$$u_i^{k+1} = v_i^{k+1} + \alpha^k d_i^k \tag{1.4}$$

  where:

$$d_i^k = -\nabla J_i(v_i^{k+1}) \tag{1.5}$$

  with:

$$u_i^0 \in \mathbb{R}^2 \tag{1.6}$$

  The main problem with this method is the computation of the descent direction so the convergence. As we can see in equation 1.4 and 1.5 we are using the local direction, in particular we are using only the gradient of the function J_i, $\nabla J_i(v_i^{k+1})$ (so we don't use information about the neighbors). **Note**: To better see the convergence problem we can consider the following example:

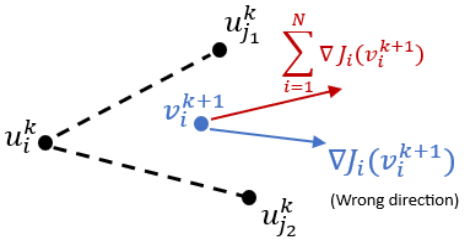


Figure 1.5: Wrong direction.

  Let's suppose that u_i is connected with $u_{j_1}^k$ and $u_{j_2}^k$. When we implement the distributed gradient method we are doing the following:

  – Agent i as first step compute v_i^{k+1} (Aggregating of the estimates of i and other agents j).

– if all of them could compute the two gradients, they should move along the red direction (Correct direction). But, since agent i only using its own gradient, it is moving along a wrong direction. As Shown in figure 1.5. For this reason only under some condition the algorithm converges.

For convergence we need to have some assumption:

– **Assumption 1:** The weighted adjacency matrix A, associated to the undirected and connected graph G must be doubly stochastic (row stochastic not enough) and must have all $a_{ii} > 0$.
– **Assumption 2:** The step-size sequence $\{a^k\}_{t\geq 0}$, must satisfy the conditions: $a^k \geq 0$; $\sum_{k=0}^{\infty} a^k = \infty$; $\sum_{k=0}^{\infty} (a^k)^2 < \infty$. So we must have diminishing step-size. This assumption is required because at each iteration, of the distributed gradient algorithm, the descent direction is wrong. If a constant step-size is used some errors can occurs.
– **Assumption 3:** Each cost function $J_i : \mathbb{R}^d \to \mathbb{R}$ must be convex and must have a bounded gradient. Furthermore, the optimization problem must have at least one optimal solution.

The Distributed Gradient Algorithm can be written as:

Algorithm 1 Distributed Gradient Method algorithm

Initialize u_i^0 for each agent i
while $k \leq MAXITERS$ **do**
  for each agent i **do**
    $v_i^{k+1} \leftarrow \sum_{j\in\mathcal{N}_i} a_{ij} u_j^k$
    $u_i^{k+1} \leftarrow v_i^{k+1} - \alpha^k d_i^k$
  end for
end while

In our project we implement the distributed gradient method only to compare with a more powerful but also more complex algorithm, the *Distributed Gradient Tracking* algorithm.

- **Distributed Gradient Tracking method**

  The basic idea is to use the informations of neighbors in order to compute a better direction that asymptotically track the global gradient with respect distributed gradient method case, where the direction defined by 1.5. The Distributed Gradient Tracking method is an improvement that allows to track the correct gradient by replacing the

exact descent direction with a local descent direction which is updated by using the dynamic average consensus. So the aim is to improve the exact descent direction in order to gain a linear convergence rate. So the new direction that reconstruct the global gradient is given by the sum of all cost functions of the network as we can see in 1.7:

$$d_i^k \xrightarrow{k\to\infty} -\frac{1}{N}\sum_{h=1}^{N}\nabla J_h(u_h^k) \tag{1.7}$$

We can write the Distributed Gradient Tracking in explicit form in the following way:

$$u_i^{k+1} = \sum_{j\in\mathcal{N}_i} a_{ij}u_j^k - \alpha y_i^k \tag{1.8}$$

$$y_i^{k+1} = \sum_{j\in\mathcal{N}_i} a_{ij}y_j^k + (\nabla J_i(u_i^{k+1}) - \nabla J_i(u_i^k)) \tag{1.9}$$

with:

$$u_i^0 \in \mathbb{R} \tag{1.10}$$

$$y_i^0 = \nabla J_i(u_i^0) \tag{1.11}$$

Where $\alpha \in \mathbb{R}$ is the step-size, $u_i^k \in \mathbb{R}^d$ are the agent estimates, $y_i^k \in \mathbb{R}^d$ is the local agent estimate of the cost function gradient according to the consensus algorithm and $a_{ij} \in \mathbb{R}$ are the graph weights.

The distributed gradient tracking algorithm can be written as:

Algorithm 2 Distributed Gradient Tracking algorithm

Initialize u_i^0 and $y_i^0 = \nabla f_i(u_i^0)$ for each agent i
while $k \leq MAXITERS$ **do**
  for each agent i **do**
    $u_i^{k+1} \leftarrow \sum_{j\in\mathcal{N}_i} a_{ij}u_j^k - \alpha y_i^k$
    $y_i^{k+1} \leftarrow \sum_{j\in\mathcal{N}_i} a_{ij}y_j^k + \nabla J_i(u_i^{k+1}) - \nabla J_i(u_i^k)$
  end for
end while

In order to use this algorithm, we need to fulfill some assumptions:

- **Assumption 1:** Let a_{ij}, $i, j \in \{1, ..., N\}$ be a non negative entry of the weighted adjacency matrix A associated to the undirected and connected graph G, with $a_{ii} > 0$ and A doubly stochastic.
- **Assumption 2:** For all $i \in 1, .., N$, each cost function $J_i : \mathbb{R}^d \to \mathbb{R}$ satisfies the following conditions:
  - it is strongly convex with coefficient $\mu > 0$ (Implies uniqueness of the optimal solution u^*)
  - it has Lipschitz continuous gradient with constant $L > 0$.

1.2.2.1 Differences

A remarkable difference between the two methods is that the Gradient Tracking Method has a faster convergence rate because it allows for the use of a constant step-size. So thanks to this there is an exact convergence; this means that the gradient tracking scheme presents an equilibrium point which is reached when there is a condition in which the gradient of the total function is zero and all agents reach consensus. However, one of the main advantages is that the Gradient Tracking can reduce the amount of communication between the nodes of the network; in other words the nodes not communicate the entire gradient but only a small amount of information about this one, such as its direction and the magnitude. Another aspect is that this algorithm is more complicated, with respect the Distributed Gradient Method, because it needs more states in order to have a distributed implementation of the linear state. Nevertheless, the Distributed Gradient Tracking is able improve the robustness of the optimization process to noisy communication. By tracking the global gradient, each node can better adapt to changes in the network or the cost function. Overall, as mentioned before, the Distributed Gradient Tracking is a powerful technique for improving the convergence rate and accuracy of distributed optimization.

1.2.3 Code features

In this section we want to highlight all the main characteristic of the implemented code.

1.2.3.1 Cost Function

As mentioned in section 1.2.1 the cost is designed as a quadratic function, in order to have a convex optimization problem with only one global minimum, and results:

$$J_i(u) = \frac{1}{2} u^T Q u + R u \tag{1.12}$$

with Gradient:

$$\nabla J_i(u) = Qu + R \tag{1.13}$$

where $u \in \mathbb{R}^d$, $R \in \mathbb{R}^N$ and $Q \in \mathbb{R}^{Nd^2}$. The matrix Q is defined as a positive definite matrix, to see this we can look at the eigenvalues.

In order to implement the quadratic cost function $J_i(u)$, we define the python's function *quadratic_fn* schematized in table 1.1. The function return two values, J_{value} because we want to track the evolution of the cost and J_{grd} because we want to compute the gradient of cost function.

quadratic_fn			
Input:	u	$Q\_$	r
0utput:	J_{val}	J_{grd}	

Table 1.1: Quadratic function *quadratic_fn*.

1.2.3.2 Graphs and Weights used

In the implemented code there is the possibility to choose between different types of graphs, as it is required in the task 1 assignment. In particular we can select the following graphs;

1. Binomial Graph (Erdos-Renyi graph).
2. Path Graph.
3. Star Graph.
4. Cycle Graph.
5. Complete Graph

To each Graph is connected a different adjiacency matrix $A \in \mathbb{R}^{N \times N}$ where N is the number of node/agents, which is used in the implementation of the algorithms to solve the consensus optimization problem, as we can see in the figure 1.6. Changing the graph we change the adjiacency matrix, for this reason we can improve or decrease the performance of the implemented algorithm.

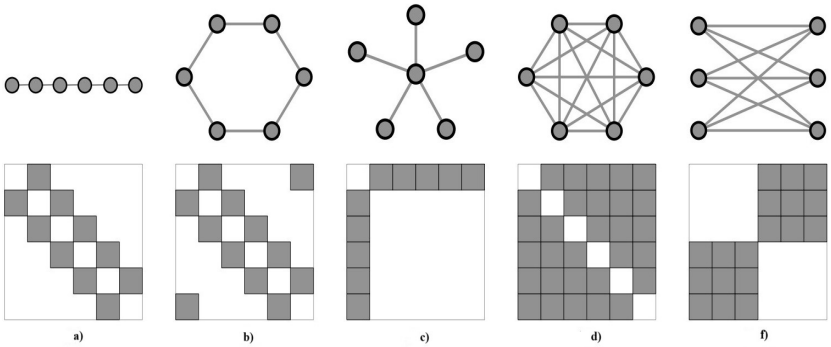


Figure 1.6: Path ,Cycle ,Star ,Complete and Complete Bipartite graph and their binary adjacency matrices in pixel picture representation.

The main difference between the different matrices that can be obtained by changing the type of graph concerns the number of elements of the matrix. The most powerful is the complete graph because we have a complete communication at each node (all node has all data available). Different graphs can have an impact on the performance of the algorithms described in section 1.2.2, whose performance is affected by the connectivity of the network. In particular, the convergence rate and optimality of the algorithm can be affected by the structure of the graph. For example, the performance of Distributed Gradient Tracking algorithms is affected by the connectivity of the network, the heterogeneity of the data, and the communication protocol used for exchanging information among nodes.

In general, in distributed algorithms and control laws we need to weight the information that comes from neighbors (averaging algorithms). For this reason is useful to have weighted graphs and weighted adjacency matrices. There are different way to generate the weights. Another characteristic is that the code allows to switch between two types of adjiacency matrix's weights:

- **Euristic Weights**: In the context of distributed optimization, heuristic weights can be used to represent the weights of the edges in a graph, which can affect the convergence rate and performance of the optimization algorithm. The heuristic weight is typically calculated based on some heuristic function that estimates the distance between two nodes. It is important to notice that the choice of this heuristic function and the weight assigned to it can have a significant impact on the efficiency and accuracy of the algorithm.

- **Metropolis-Hastings Weights**: In this case the weights matrix of the agents graph is computed using the *Metropolis Hasting method*, which is reported below:

$$a_{ij} = \begin{cases} \frac{1}{1+max(d_i,d_j)}, & \text{if } (i,j) \in E \text{ and } i \neq j \\ 1 - \sum_{h \in N_i \setminus \{i\}} a_{ih}, & \text{if } i = j \\ 0 & \text{Otherwise} \end{cases} \tag{1.14}$$

Speaking about a weighted graph we can extend the notation of IN-DEGREE AND OUT-DEGREE:

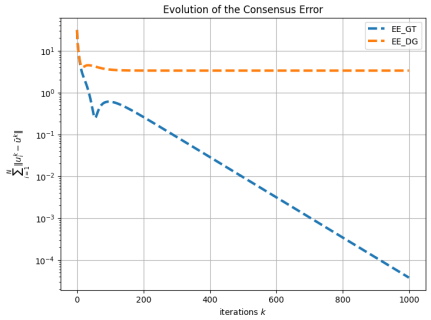
- weighted IN-DEGREE: $deg_i^{IN} = \sum_{j=1}^{N} a_{ji}$
- weighted OUT-DEGREE: $deg_i^{OUT} = \sum_{j=1}^{N} a_{ij}$

Whit this definitions we are able to understand if the selected graph is *Weight-Balanced*: A graph is weighted-balanced if $deg_i^{IN} = deg_i^{OUT}$. Having a weight-balanced graph is important in distributed optimization because it can improve the performance and convergence rate of the algorithms.

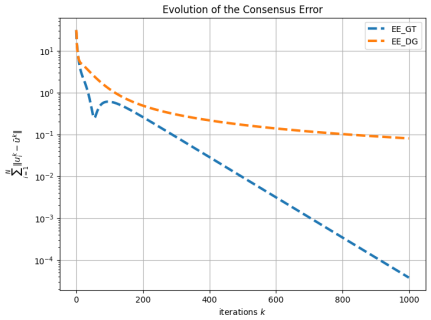
1.2.4 Result and Consideration

In this section we report all the main results obtained using the Distributed Gradient and the Distributed Gradient Tracking algorithm. In particular we test the algorithms with different graph types and different weights types. For all the simulation the following parameter are fixed : $NN = 10$, $MAXITERS = 1000$, $dd = 2$ and $ss = 0.01$. For each case we compare the performances of the two algorithms. As mentioned above in section 1.2.2.1 the more efficient is the Distibuted Gradient Tracking.

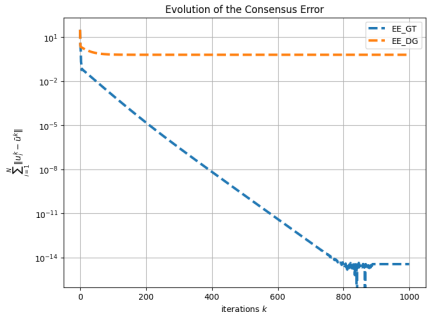
As first step we show the **Consensus error** which is a measure of the discrepancy between the local estimates of the agents in the network and the true value of the quantity being estimated.



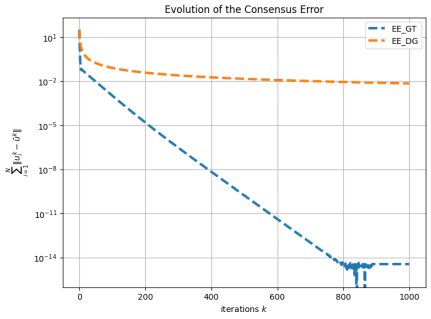
(a) Constant step size and a *Path* graph



(b) Diminishing step size and a *path* graph



(c) Constant step size and a *Complete* graph

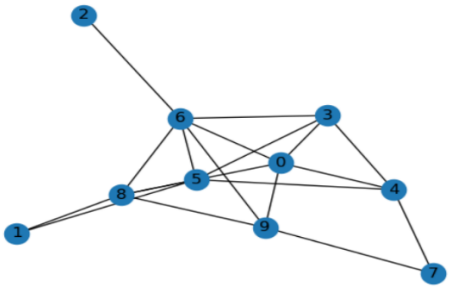


(d) Diminishing step size and a *Complete* graph

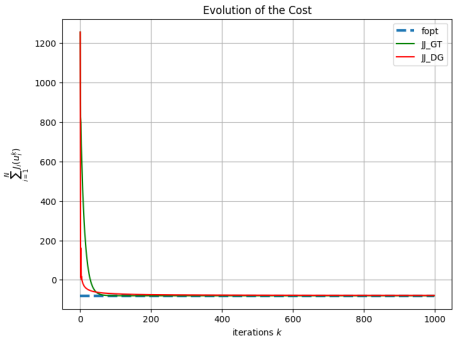
Figure 1.7: Consensus Error.

We can observe that the type of a graph can have an impact on the evolution of the consensus error in a distributed algorithm. The structure and topology of the graph can affect the convergence rate, accuracy, robustness, and resiliency of the algorithms. In the case of the Distributed Gradient algorithm we have convergence only with a diminishing step size. If we use a small enough step size we still converge to a value close to the optimal one.

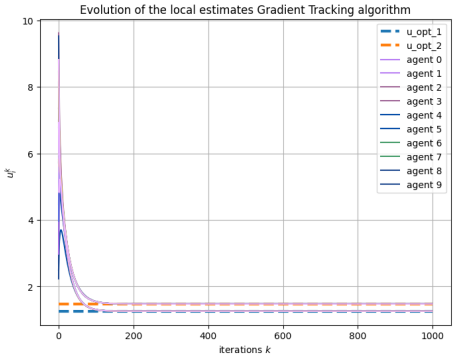
1.2.4.1 Binomial graph results



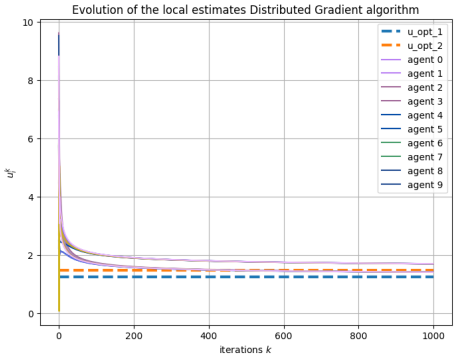
(a) Binomial Graph



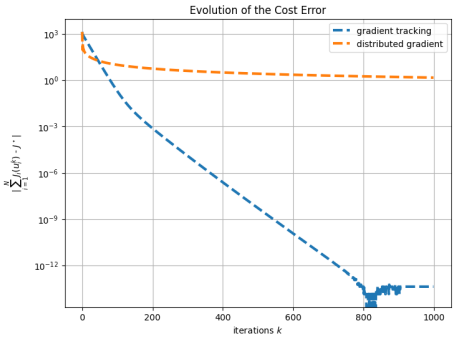
(b) Cost Evolution



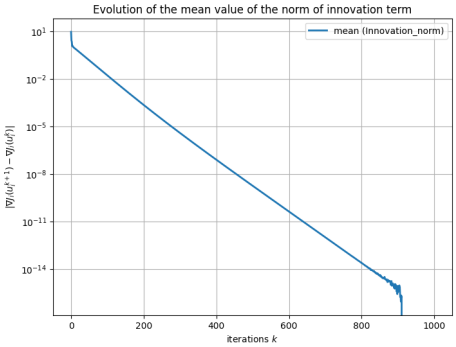
(c) Evolution of the local estimate Distributed Gradient Tracking



(d) Evolution of the local estimate Distributed Gradient



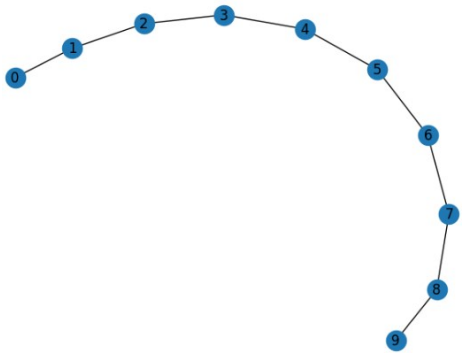
(e) Cost Error evolution



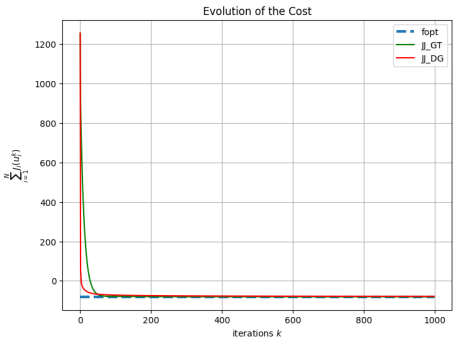
(f) Norm of the innovation term

Figure 1.8: Results using a Binomial graph.

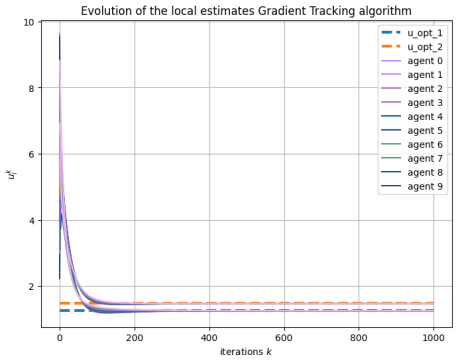
1.2.4.2 Path graph results



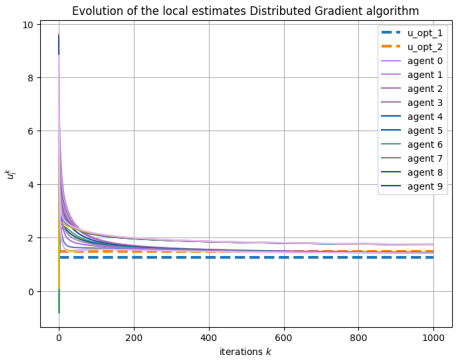
(a) Path Graph



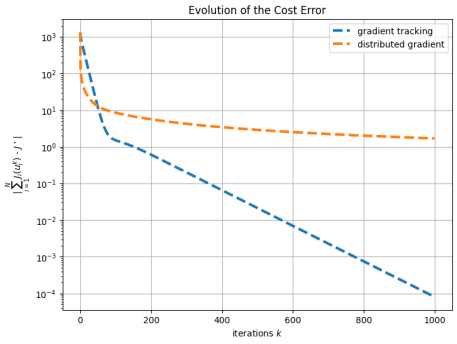
(b) Cost evolution



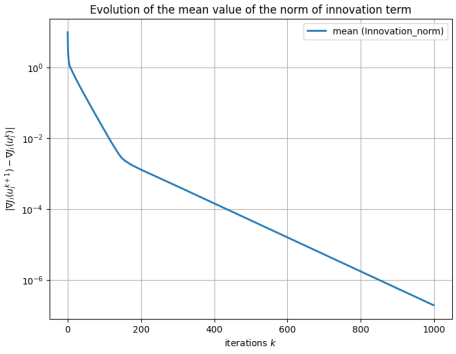
(c) Evolution of the local estimate Distributed Gradient Tracking



(d) Evolution of the local estimate Distributed Gradient



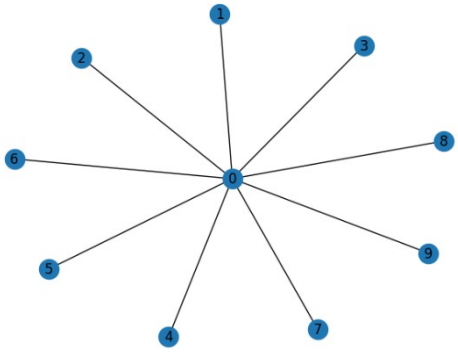
(e) Cost Error evolution



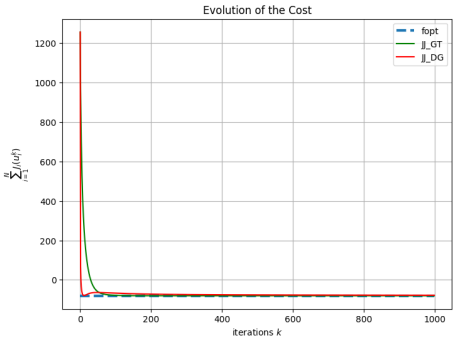
(f) Norm of the innovation term

Figure 1.9: Results using a Path graph.

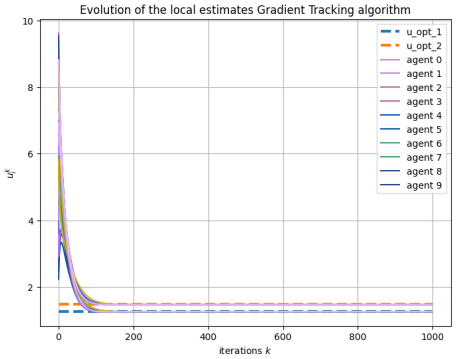
1.2.4.3 Star graph results



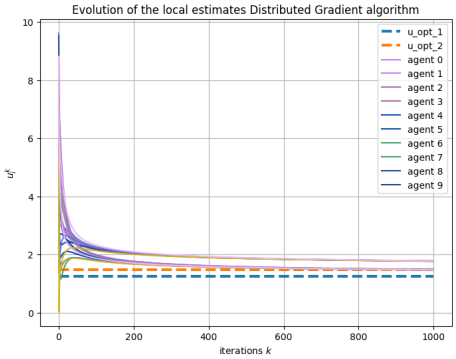
(a) Star Graph



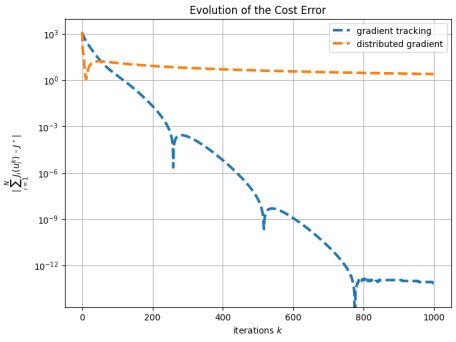
(b) Cost evolution



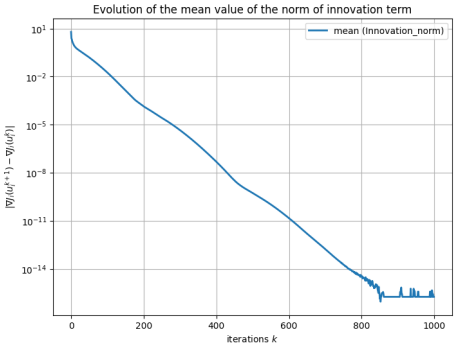
(c) Evolution of the local estimate Distributed Gradient Tracking



(d) Evolution of the local estimate Distributed Gradient



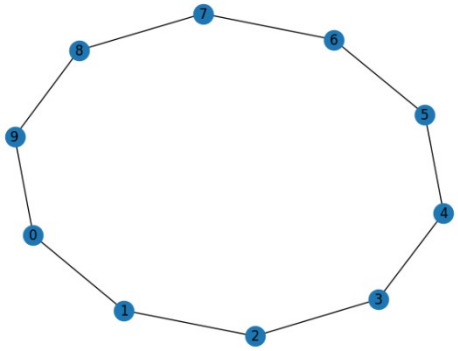
(e) Cost Error evolution



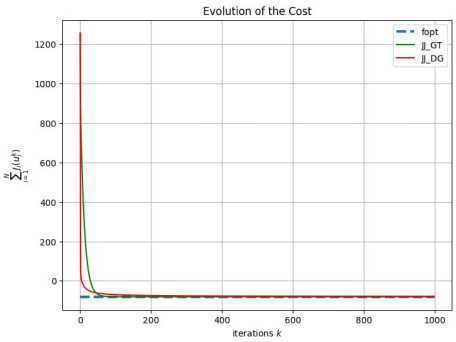
(f) Norm of the innovation term

Figure 1.10: Results using a Star graph.

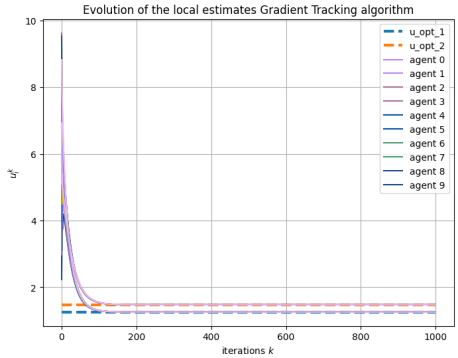
1.2.4.4 Cycle graph results



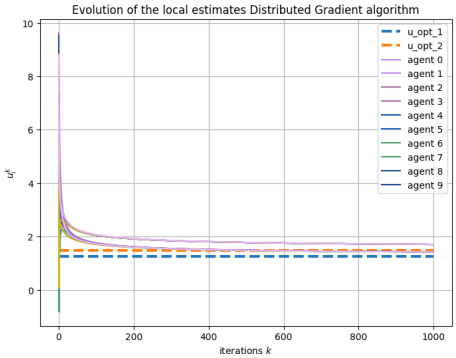
(a) Cycle Graph



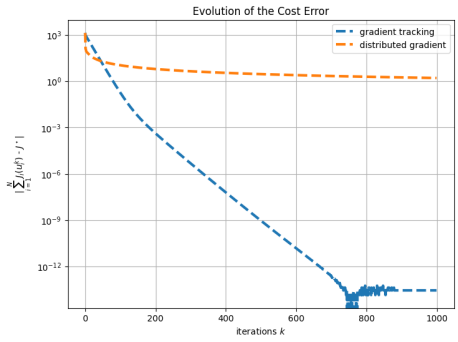
(b) Cost evolution



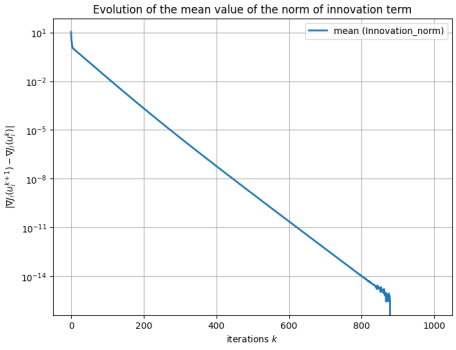
(c) Evolution of the local estimate Distributed Gradient Tracking



(d) Evolution of the local estimate Distributed Gradient



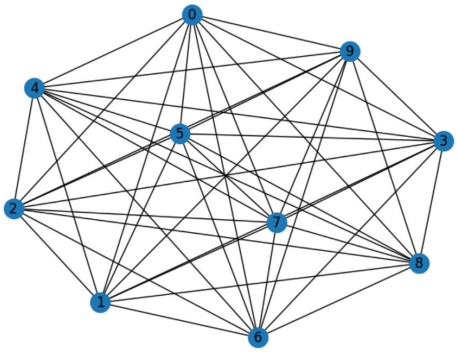
(e) Cost Error evolution



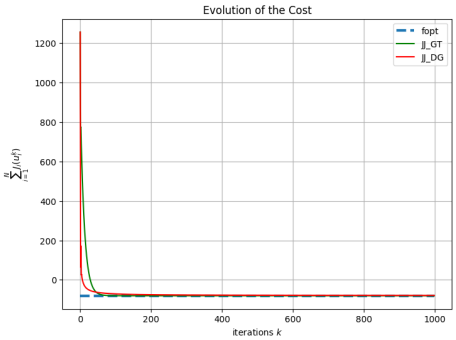
(f) Norm of the innovation term

Figure 1.11: Results using a Cycle graph.

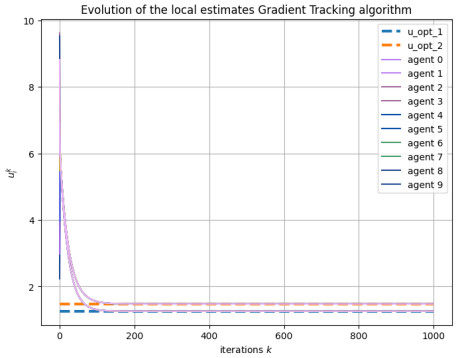
1.2.4.5 Complete graph results



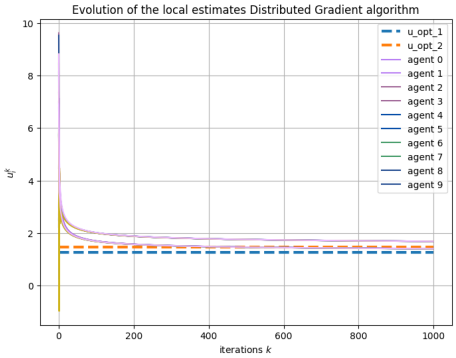
(a) Complete Graph



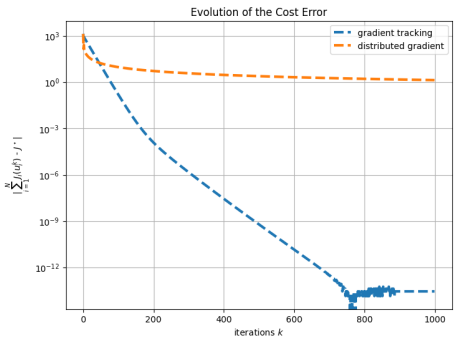
(b) Cost evolution



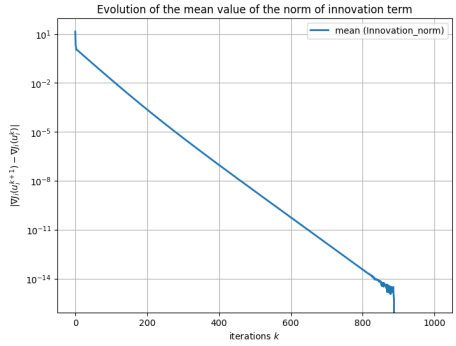
(c) Evolution of the local estimate Distributed Gradient Tracking



(d) Evolution of the local estimate Distributed Gradient



(e) Cost Error evolution



(f) Norm of the innovation term

Figure 1.12: Results using a Complete graph.

1.3 Centralized training

Centralized training refers to a training approach where all the training data is gathered and processed in a single, central location. This is often contrasted with decentralized training, that is implemented in the next chapter, where data is distributed across multiple agents or nodes. Having all the data in one place makes things straightforward.

1.3.1 Importing MNIST Dataset

To obtain our starting set of data we exploit the Keras Python Lybrary which provides the MNIST dataset. In particular, the MNIST dataset, often used in machine learning, is a collection of 70.000 of greyscale images of hand-written digits from 0 to 9, represented in a matrix form as $[0, 255]^{28x28}$. In order to evaluate the performance of the final model we split the entire dataset in training and test set. As just said MNIST dataset is made of 70,000 handwritten digits. So, we splitted MNIST $\{D^i, y^i\}_{i=1}^{70,000}$ in a *Training set* $\{D^j, y^j\}_{j=1}^{60,000}$ and *Test set* $\{D^k, y^k\}_{k=1}^{10,000}$. In figure 1.13 it is possible to see an example of the that we obtain.

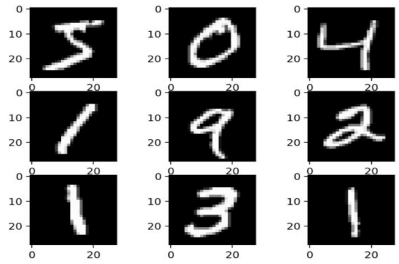


Figure 1.13: Example of grayscale digits from MNIST dataset.

Then, in figure 1.14 it is possible to see the distribution of the original MNIST dataset. To plot this figure we have defined a python function t_SNE_plot which is well explained in the subparagraph 1.3.4.

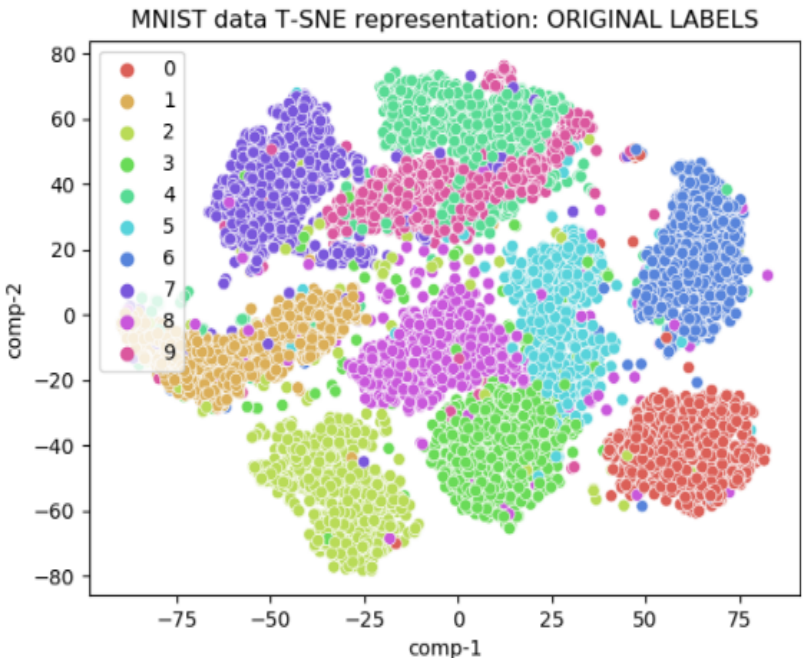


Figure 1.14: MNIST data T-SNE representation: ORIGINAL LABELS.

1.3.2 Dataset manipulation

Once we have obtained the data we normalize and reshape them. So, since each image has a set of pixel values in a range [0,255], we normalize this range between [0,1] in order to avoid the saturation of the activation functions of our neurons. Now we examine our normalized data plotting two images with the relative histograms in order to see the density of the white and black pixels that define our images. In particular we have chosen the digit 1 and the digit 5.

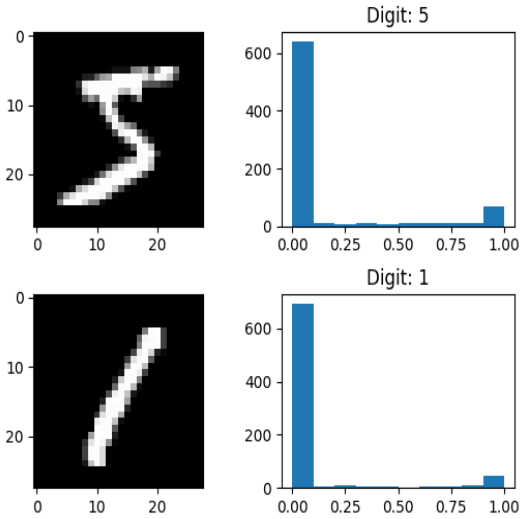


Figure 1.15: Example of normalized data.

As a result we can notice that we have an higher density of points that are close to zero, these represents the black pixels of our images. Instead, the rest of the grey scale points that are close to 1, so to white color, represent the digit. Looking better the two histograms we can notice a little difference in the distribution of the white pixels, in particular we can understand that for digit 1 less white pixels are necessary.

Then, since each image is represented by a matrix size $[28 \times 28]$ we reshape it in a vector of dimension $[784 \times 1]$ in order to provide the correct dimension of the inputs of our neural network. In the following figure it is possible to see an example of a vector $\mathbb{R}^{748\times1}$ obtained from a matrix $\mathbb{R}^{28\times28}$

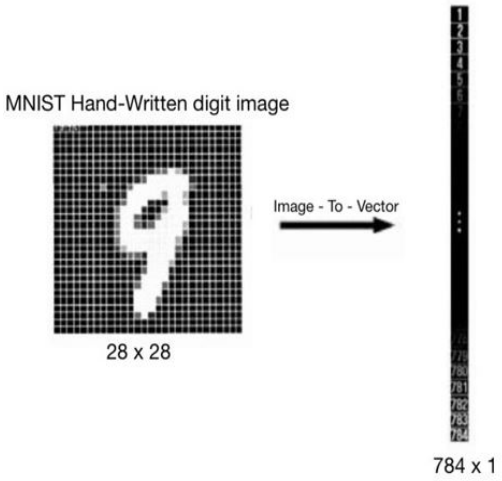


Figure 1.16: Obtained vector from a matrix.

As last step, we chose a desired digit that our neural network should classify. For this purpose we chose the digit 4 and then we modify the labels of our Data as follow:

$$y^i = \begin{cases} 1 & \text{if digit} = 4 \\ 0 & \text{other cases} \end{cases}$$

In the figure 1.17 it is possible to see an example.

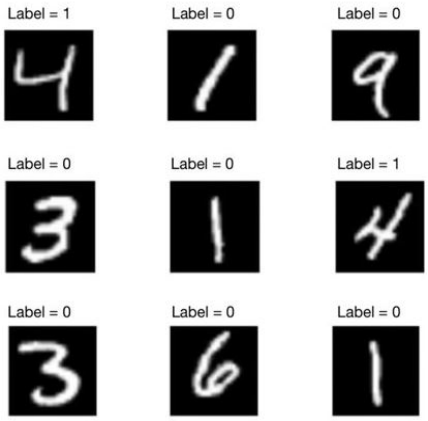


Figure 1.17: Chosen labels example.

The figure 1.18 shows the distribution of the data with the associated label 1 and 0.

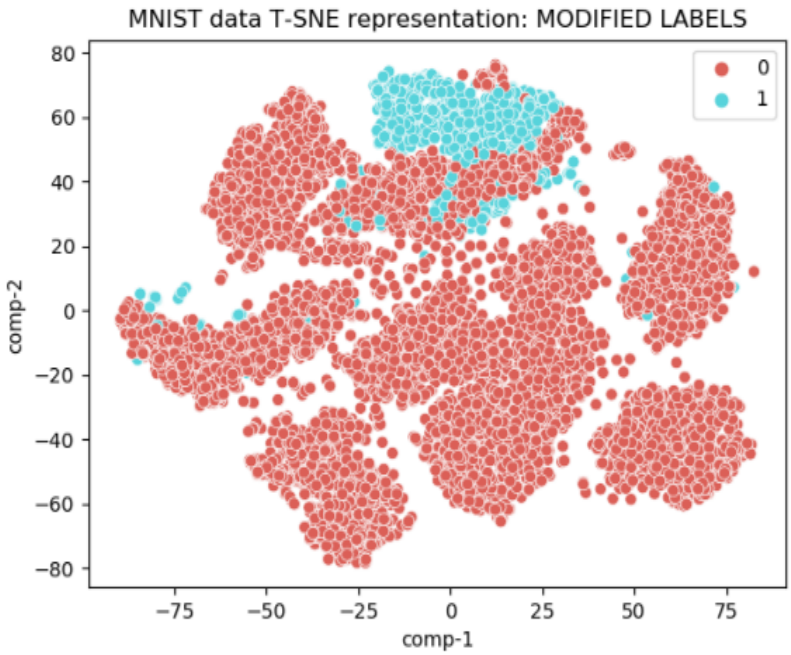


Figure 1.18: MNIST data T-SNE representation: MODIFIED LABELS.

1.3.3 Balancing the Dataset

Due to the previous manipulation our training set is not balanced, because the class related to the label 0 is a *Majority class* while the one related to the label 1 is a *Minority class*. There are different tecniques to convert an imbalanced dataset into balanced dataset, for example Undersampling or Oversampling technique. To accomplish this we exploit the Undersampling technique, figure 1.19, which remove examples from the training dataset that belong to the *Majority class* in order to better balance the class distribution. In this way we are able to obtain a better training of the machine learning model.

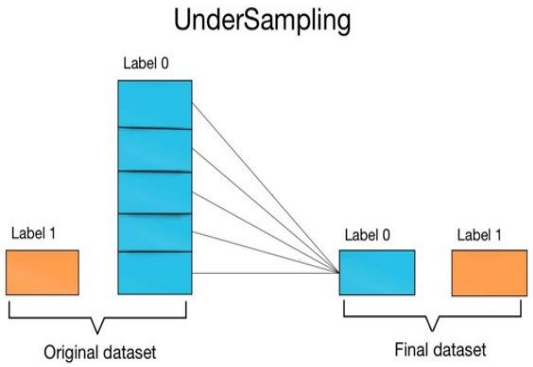


Figure 1.19: Undersampling Technique.

The distribution of the balanced data is shown in figure 1.20.

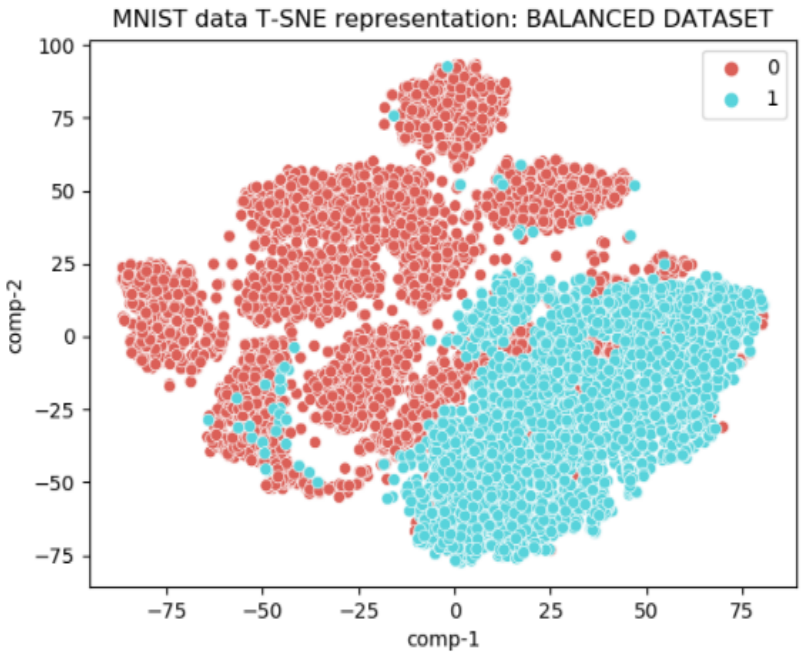


Figure 1.20: MNIST data T-SNE representation: BALANCED DATASET.

1.3.4 Functions

1.3.4.1 t_SNE Function Plot()

The python function *t_SNE_plot* exploit the the t-distributed stochastic neighbor embedding (t-SNE) tecnique. This approach is used in machine learnig for the dimensionality reduction and it is useful for the visualization of high dimension dataset. The function takes as input the Dataset and a

number that allow to select the different configurations of data that we want to represent. In particular by selecting the number 1 we plot the original Mnist dataset t-SNE representation. Then, by selecting number 2 we plot the original Mnist data with the modified labels. Finally, by selecting number 3 we plot the balanced data t-SNE representation. In the table 1.2 it is possible to see *t_SNE_plot()* function's schematization:

t_SNE_plot()			
Input:	X_train	y_train	1
0utput:	$None$		

Table 1.2: Python function *t_SNE_plot()*

1.3.4.2 Histogram Function Plot

The function *histogram_plot()* allow us to plot two selected images with the relative histograms. In this way we can see and compare the the density of the white and black pixels that define the digits. The function is schematized in tab 1.3:

histogram_plot()		
Input:	X_hist	y_train
0utput:	$None$	

Table 1.3: Python function *histogram_plot()*

1.3.4.3 Wrong Digits Classification Function Plot

The function *wrong_classified_digits_plot()* allows to see images that are incorrectly classified. In particular, in figure 1.21 we can see an example of 10 images where we can notice how the NN predicts labels which have different value with respect to the true labels.

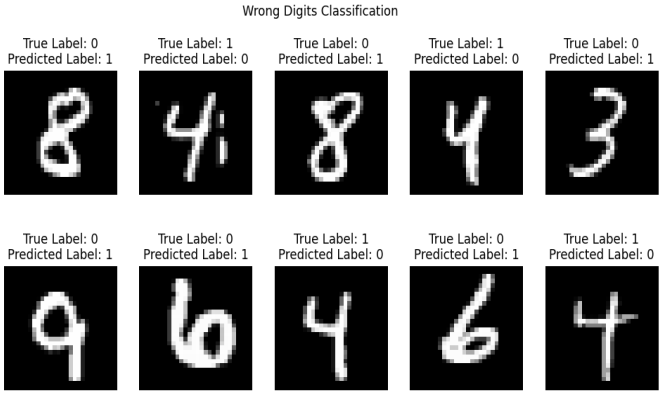


Figure 1.21: Misclassification digits example

The schematization of *wrong_classified_digits_plot()* function can be seen in table 1.4:

wrong_classified_digits_plot()				
Input:	*X_test_blc*	*y_test_blc*	missclass_indexes	prediction
0utput:	*None*			

Table 1.4: Python function *wrong_classified_digits_plot()*

1.3.4.4 Confusion Matrix Function Plot

With this function we are able to generate and to show the confusion matrix, one of the metrics used in our project to evaluate the performances of our neural network. The matrix will be shown later in the results and considerations section. The *confusion_matrix_plot()* function is schematized in 1.5.

confusion_matrix_plot()		
Input:	*y_test_blc*	prediction
0utput:	*None*	

Table 1.5: Python function *confusion_matrix_plot()*

1.3.4.5 Sigmoid Function

With the *sigmoid_fn()* function we implement the Sigmoid activation function 1.15. The function is schematized in table 1.6. In figure 1.22 is possible to see its shape. The activation function is a mathematical function that

determines the output of a neuron in the neural network. More precisely activation functions are used in NN to compute the weighted sum of inputs and biases used to decide whether a neuron can be activated or not. The purpose of activation function is to introduce non-linearity into the output of a neuron allowing the neural network to learn complex patterns. Since we are dealing with a binary classification problem we implement the Sigmoid function which allows to map input values in a range of [0,1]. In this way we are able to assign a probability to each input of the NN belonging to one of the two classes. It will be used in the *inference_dynamic()* function.

$$\sigma(\xi) = \frac{1}{1+e^{-\xi}} = \frac{e^{\xi}}{e^{\xi}+1} \tag{1.15}$$

sigmoid_fn()	
Input:	x_i
0utput:	$1/(1+\exp(-xi))$

Table 1.6: Python function *sigmoid_fn()*

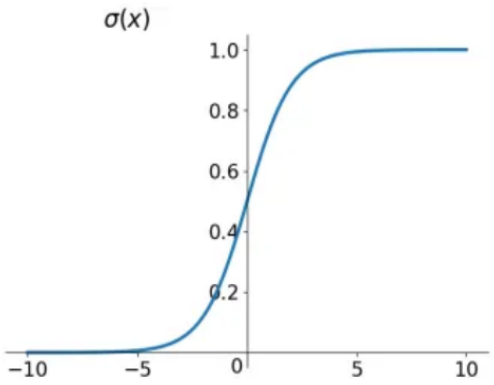


Figure 1.22: Sigmoid function

1.3.4.6 Sigmoid Derivative Function

With the *sigmoid_fn_derivative()* function we implement the derivative of the Sigmoid activation function 1.16. The function is schematized in table 1.7. In figure 1.23 is possible to see its shape. It will be used in the *adjoint_dynamics()* function.

$$\dot{\sigma}(\xi) = \frac{1}{1+e^{-\xi}}\left(1 - \frac{1}{1+e^{-\xi}}\right) \tag{1.16}$$

sigmoid_fn_derivative()	
Input:	x_i
0utput:	$sigmoid\_fn(xi) * (1 - sigmoid\_fn(xi))$

Table 1.7: Python function *sigmoid_fn_derivative()*

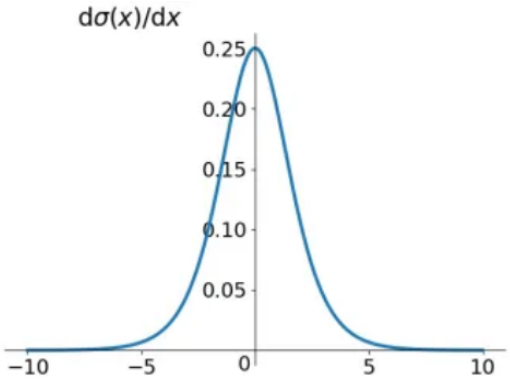


Figure 1.23: Sigmoid function derivative

1.3.4.7 Inference Dynamics Function

With this function we implement the inference dynamic 1.17 where $u_{h,0}$ is the bias and $\sigma : \mathbb{R} \to \mathbb{R}$ the activation function. We compute the scalar product between the current state and the current input and apply the activation function. In other words we find the update of each neuron $h = 1, .., d_n$ of layer $t + 1$.

The function is schematized in the table 1.8. It take as input:

- xt: Current state $\in \mathbb{R}^{d_c}$
- ut: Current input (optimal control scenario)/ current weights (machine learning scenario) $\in \mathbb{R}^{d_n} \times \mathbb{R}^{d_c+1}$

and as output:

- xtp: Next state $\in \mathbb{R}^{d_n}$

Where d_c is the number of neuron in the current layer and d_n is the number of neuron in the next layer. $d_c + 1$ because we consider the bias.

$$x_{h,t+1} = \sigma((x_t)^T u_{h,t} + u_{h,0}) \tag{1.17}$$

The aggregate formulation of the inference dynamic can be written as :

$$\begin{bmatrix} x_{1,t+1} \\ \vdots \\ x_{d_n,t+1} \end{bmatrix} = \begin{bmatrix} \sigma(x_t^T u_{1,t}) \\ \vdots \\ \sigma(x_t^T u_{d_c,t}) \end{bmatrix} \quad \Rightarrow \quad x_{t+1} = f(x_t, u_t),$$

inference_dynamics()		
Input:	x_t	u_t
0utput:	x_{tp}	

Table 1.8: Python function *inference_dynamics()*

1.3.4.8 Forward Pass Function

With this function we implement the Forward propagation. It refers to the calculation process and storage of intermediate variables, so values output from the layers based on the input data. More in details, the input X_0 provides the initial information which propagates to the hidden layers producing the output $\tilde{y}$ which is our prediction. The function allows us to explore all neurons from the first layer to the last. For this reason, in this function we iterate the inference dynamics for each layer (except the last one) present in the network. In particular, for each layer step, using a recurrent approach, we update the trajectory of state xx based on an input sequence. The *FORWARD_pass()* function is schematized in table 1.9. It takes as input:

- uu Input trajectory: $u[0], u[1], ..., u[T-1] \in \mathbb{R}^{T-1} \times \mathbb{R}^{d_n} \times \mathbb{R}^{d_c+1}$
- x_0 Initial condition $\in \mathbb{R}^{d_c}$
- T number of NN layers

and as output:

- xx State trajectory: $x[0], x[1], ..., x[T] \in \mathbb{R}^T \times \mathbb{R}^{d_c}$

FORWARD_pass()			
Input:	uu	x_0	T
Output:	xx		

Table 1.9: Python function *FORWARD_pass()*

1.3.4.9 Adjoint Dynamics Function

With this function we want to apply the *Adjoint method*, for this reason we compute the derivative of the vector field $\mathbf{f} : \mathbb{R}^{d_c} \times \mathbb{R}^{(d_n \times d_{c+1})} \to \mathbb{R}^{d_n}$:

$$f({x_t}^k, {u_t}^k) = \begin{bmatrix} f_1({x_t}^k, {u_t}^k) \\ \vdots \\ f_{d_n}(x_t^k, {u_t}^k) \end{bmatrix} = \begin{bmatrix} \sigma({x_t^k}^T u_{1,t}^k) \\ \vdots \\ \sigma({x_t^k}^T u_{d_c,t}^k) \end{bmatrix} \tag{1.18}$$

The function is schematized in table 1.10. As reported, the function take as input the current state $x_t \in \mathbb{R}^{d_c}$, the current input $u_t \in \mathbb{R}^{d_n} \times \mathbb{R}^{d_c+1}$ and the current costate that has the same dimension as x_t for all t = 1,....,T. The function return the next costate and the loss of the gradient with respect u_t. By considering the derivative with respect the state, we have:

$$\begin{aligned} &\nabla_1 f(x_t^k, u_t^k) = [\nabla_1 f_1(x_t^k, u_t^k) \dots \nabla_1 f_{d_n}(x_t^k, u_t^k)] = \\ &= [\hat{u}_{1,t}^k \sigma'({x_t^k}^T u_{1,t}^k) \dots \hat{u}_{d_c,t}^k \sigma'({x_t^k}^T u_{d_c,t}^k)] \in \mathbb{R}^{d_c \times d_c} \end{aligned} \tag{1.19}$$

where $\hat{u}_{h,t}^k$ denotes the weight $u_{h,t}^k$ ignoring its first entry (i.e., the bias). We define $A_t^k = \nabla_1 f(x_t^k, u_t^k)^T$.
By considering the derivative with respect to the state we have:

$$\nabla_2 f(x_t^k, u_t^k) = [\nabla_2 f_1(x_t^k, u_t^k) \dots \nabla_2 f_{d_n}(x_t^k, u_t^k)] =$$

$$= \begin{bmatrix} \begin{bmatrix} 1 \\ x_t^k \end{bmatrix} \sigma'({x_t^k}^T u_{1,t}^k) & \dots & 0_{d_{c+1}} \\ 0_{d_{c+1}} & \ddots & 0_{d_{c+1}} \\ \vdots & \dots & \vdots \\ 0_{d_{c+1}} & \dots & \begin{bmatrix} 1 \\ x_t^k \end{bmatrix} \sigma'({x_t^k}^T u_{d_c,t}^k) \end{bmatrix} \in \mathbb{R}^{((d_c+1) \times d_n) \times d_n}$$

A rectangular matrix. We define $B_t^k = \nabla_2 f(x_t^k, u_t^k)^T$.

adjoint_dynamics()			
Input:	*ltp*	x_t	u_t
0utput:	*llambda_t*	*delta_ut*	

Table 1.10: Python function *adjoint_dynamics()*

In this way we are able to define the *Adjoint Dynamics*:

$$\begin{gathered}\lambda_t^i = A_t^{i,k^T}\lambda_{t+1}^i, \quad \lambda_T^i = \nabla J(x_T^{i,k}; j^i) \\ \Delta u_t^{i,k} = B_t^{i,k^T}\lambda_{t+1}^i\end{gathered} \tag{1.20}$$

that we will simulate in backward way for $t = T-1, ..., 0$, where k is the epoch index and i is the input data index. For this reason we will call the function *adjoint_dynamics()* in the next function *BACKWARD_pass()*.

1.3.4.10 Backward function

We define this function in order to implement backward propagation, that is the main part of the NN training. It is referred as the process of fine-tuning the weights (de facto learning), based on the error rate (i.e. loss function) obtained in the previous epoch (i.e. iteration) using a Gradient descent algorithm. The function is schematized in 1.11. With this function we obtain the costate trajectory and the costate output (i.e the loss gradient) starting from the terminal condition $\lambda_T^i = \nabla J(x_T^{i,k}; y^i)$

BACKWARD_pass()				
Input:	xx	uu	λ_T	T
0utput:	$llambda$	$Delta\_u$		

Table 1.11: Python function *BACKWARD_pass()*

1.3.4.11 Binary Cross Entropy (BCE) Function

With this function we implement the BCE loss function or logistic loss 1.21 and its gradient 1.22. The function is schematized in the table 1.12. It is a Function that compares the target and predicted output values. It is suitable in case of binary classification.

$$J = (y \log(\Phi(\mathbf{u}, D)) + (1-y)\log(1-\Phi(\mathbf{u}, D)) \tag{1.21}$$

$$\frac{\partial J}{\partial \Phi(\mathbf{u}, D)} = \frac{y}{\Phi(\mathbf{u}, D)} - \frac{1-y}{1-\Phi(\mathbf{u}, D)} \tag{1.22}$$

Where $\Phi(\mathbf{u}, D)$ represent the predicted label and the y the actual (True) label. $\Phi()$ is called *shooting map*.

BCE()		
Input:	$Pred$	y
0utput:	$-loss$	$Grad\_loss$

Table 1.12: Python function *BCE()*

1.3.5 Neural Network Implementation

The problem of training a neural network is an optimal control problem, in the multi-sample case can be expressed as follows:

Given multiple data-label pairs $((D^1, y^1), ..., (D^I, y^I))$, we aim at solving:

$$\begin{aligned} \min_{x^1,...,x^1,\mathbf{u}} \quad & \sum_{i=1}^{I} J(x_T^i; y^i) = \sum_{i=1}^{I} J(\phi(\mathbf{u}, x_0^i); y^i) \\ \text{s.t.} \quad & x_{t+i}^i = f(x_t^i, u_t) \quad x_0^i = D^i \\ & \forall t = 0, ..., T-1 \\ & \forall i = 1, ..., I \end{aligned} \tag{1.23}$$

Where $J(x_T^i; y^i)$ is the loss function at last layer of the NN, x_T^i is the output of the activation function at last layer, u_t are the weights relative to layer t, I is the number of input data-label pairs, T $\in \mathbf{N}$ is the number of layers. In particular we want to find $\mathbf{u}$ such that the predicted output $\Phi(\mathbf{u}; D^i)$ is as close as possible to y^i (true label). Since the number of training data is very large we use the **Stochastic Mini-Batch Gradient Method**. It is a generalization of stochastic gradient descent that split the training dataset into small batches that are used to calculate the model error and update the model coefficient. The batch size is equal to 32.

In our code we provide the possibility to implement the neural network with different layers and number of neurons. The default structure is:

$$d = [784, 64, 1] \tag{1.24}$$

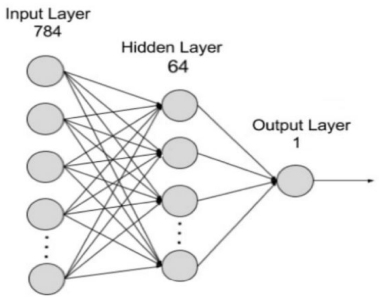


Figure 1.24: Default structure neural network

As we can see the first layer (input layer) $t = 0$ is set equal to the input data:

$$x_0 = \mathcal{D} = \left([\mathcal{D}]_1, ..., [\mathcal{D}]_{d_c}\right) \tag{1.25}$$

while the last layer (output layer) is composed by only one neuron since we have to detect only one digit. We have chose only one hidden layer in order to reduce the computation time.

Now we start to describe the learning algorithm. As first step we initialize and define all the variables and data structures used in our procedure.

We use as data structure for the input trajectory (weights in machine learning scenario) and state trajectory the python list. In particular:

- the input trajectory: $\mathbf{uu} \in \mathbb{R}^{T-1} \times \mathbb{R}^{d_n} \times \mathbb{R}^{d_c+1}$
- the state trajectory: $\mathbf{xx} \in \mathbb{R}^{T} \times \mathbb{R}^{d_c}$

Where d_c is the current number of neurons in the current layer, d_n is the number of neurons in the next layer, and T is the number of layers. $d_c + 1$ is because we consider the bias. For the initialization of the input trajectory **uu** we assign random values, in particular, since we have chosen as activation function the Sigmoid we exploit the Xavier/Glorot initialization.

For each layer t of the network the update of each neuron $h = 1, ...d_n$ of the next layer t+1 is performed by exploiting the *inference_dynamics()* function 1.3.4.7. Then, since we need to store the intermediate variables (neurons updates of each layer) between the layers we call the *FORWARD_pass()* function 1.3.4.8. For this reason, as mentioned in the last indicated subsection, we iterate the *inference_dynamics()* function for each layer present in our neural network. In this way we fill the python list **xx** (state trajectory) with all the signals between each layer. The last element of that list will be the predicted label. Subsequently, we implement the so-called backward propagation using the *BACKWARD_pass()* function 1.3.4.10 in order to simulate the adjoint dynamics (defined with the function *adjoint_dynamics()*) in backward way for $t = T - 1, ..., 0$. In other words we do what is shown in algorithm 3:

Algorithm 3 Mini-Batch gradient method steps

for K = 0,1,2... **do**:
  for batch_count in N_batches **do**:
    for batch_el in range batch_size **do**:
      i=batch_count × batch_size+batch_el
      Backward simulation of the NN:
      for layer t = T-1,...,0 **do**:

$$\lambda_t^i = A_t^{i,k^T}\lambda_{t+1}^i, \quad \lambda_T^i = \nabla J(x_T^{i,k}; j^i) \qquad \Delta u_t^{i,k} = B_t^{i,k^T}\lambda_{t+1}^i \tag{1.26}$$

with $A_t^{i,k} = \nabla_1 f(x_t^k, u_t^k)^T$ and $B_t^{i,k} = \nabla_2 f(x_t^k, u_t^k)^T$.

      end for
      Descent step on the control input:
      for layer t = 0,...,T-1 **do**:

$$u_t^{k+1} = u_t^k - \alpha^k \frac{1}{batch\_size} \sum_{i=0}^{batch\_size-1} \Delta u_t^{i,k} \tag{1.27}$$

      end for
      Forward Simulation of the NN dynamics:
      for t = 0,...,T-1 **do**:

$$x_{t+1}^{i,k+1} = f(x_t^{i,k+1}, u_t^{k+1}) \qquad x_0^{i,k+1} = D^i \tag{1.28}$$

      end for
    end for
  end for
end for

We tuning the weights by exploiting the following loss function:

$$J = \sum_{i=1}^{I} (y^i \log(\Phi(\mathbf{u}, D^i)) + (1 - y^i) \log(1 - \Phi(\mathbf{u}, D^i)) \tag{1.29}$$

that is implemented with the function *BCE()* 1.3.4.11.

1.3.6 Performance metrics

After training the NN using the train dataset, we test its performance on the test dataset. The test dataset is a set of data that has not been used for neural network training, but is used to evaluate performance of the neural network on data not seen during training. This allows us to evaluate the ability of the neural network to generalize and make accurate predictions

on new data. As said in the 1.3.2 the digit that our NN has to detect is 4. So, the NN return 1 as output when the selected digit is detected and 0 otherwise. More in detail the model is evaluated using additional metrics. In general, the metrics to be calculated are highly dependent on the nature of the model output. Since we are dealing with a classification, we can use *Confusion matrix*, *Accuracy*, *Misclassification Rate*, *Precision*, *Sensitivity* and *Specificity*.

- **Confusion Matrix**: A Confusion matrix is an $N \times N$ matrix used for evaluating the performance of a classification model, where N is the number of target classes. Since we deal with a binary classifier $N = 2$. The matrix compares the actual target values with those predicted by the machine learning model.

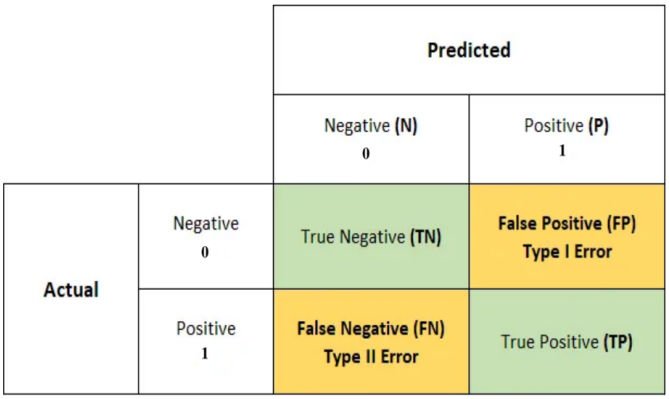


Figure 1.25: Confusion matrix for Binary Classification.

A confusion matrix is a tabular summary of the number of correct and incorrect predictions made by a classifier. It is used to measure the performance of a classification model. Confusion matrix is widely used because it give a better idea on how the model performs and which kind of error it makes. As we can see in figure 1.25 we have :

  - True positive **TP**: when the actual (True) value is Positive and predicted is also Positive. So, the number of test results predicted as 1 when they are actually 1.
  - False positive **FP**: When the actual (True) is negative but prediction is Positive. So, the number of test results predicted as 1 when they are actually 0. Also known as the **Type 1 error**.
  - True negative **TN**: when the actual (True) value is Negative and prediction is also Negative. So the number of test results predicted as 0 when they are actually 0.

  - False negative **FN**: When the actual (True) is Positive but the prediction is Negative. So the number of test results predicted as 0 when they are actually 1. Also known as the **Type 2 error**.

  In general at a good model correspond higher value of TP and TN on the main diagonal of the confusion matrix and lower values of FN and FP outsize the main diagonal.

- **Accuracy**: Measures how often the classifier makes the correct prediction. It is the ratio between the number of correct predictions and the total number of predictions:

$$\frac{TP + TN}{TP + TN + FP + FN} \times 100 \tag{1.30}$$

- **Misclassification Rate**: Is a metric that tell us the percentage of observations that were incorrectly predicted by a classification model. (Accuracy = 1 - misclassification)

$$\frac{FP + FN}{TN + FP + FN + TF} \times 100 \tag{1.31}$$

- **Precision**: It is a measure of correctness that is achieved in true prediction. In simple words, it tells us how many predictions are actually positive out of all the total positive predicted. *Precision should be high(ideally 1).*

$$\frac{TP}{TP + FP} \times 100 \tag{1.32}$$

- **Sensitivity**: The "True positive rate". The percentage of positive outcomes the model is able to detect.

$$\frac{TP}{TP + FN} \times 100 \tag{1.33}$$

- **Specificity**: The "True negative rate". The percentage of negative outcomes the model is able to detect.

$$\frac{TN}{TN + FP} \times 100 \tag{1.34}$$

Furthermore, to evaluate the performance we used some useful graphs such as the **loss trend**, the **plot of the Gradient norm** of the cost function both along the epochs

1.3.7 Results and Considerations

To test our algorithm we fix the training data size and the test data size to speed up the training and the test process. In particular we analyze the performances of our NN with the following parameters:

- select_digit = 4
- MAX_EPHOCS = 50
- batch_size = 32
- train_size = 1000
- test_size = 500
- $d = [784, 64, 1]$
- step_size = 0.1

We use the metrics explained in section 1.3.6 to have a better understanding of the performances of our neural network. A global view of the NN classification work is given by the confusion matrix in figure 1.26.

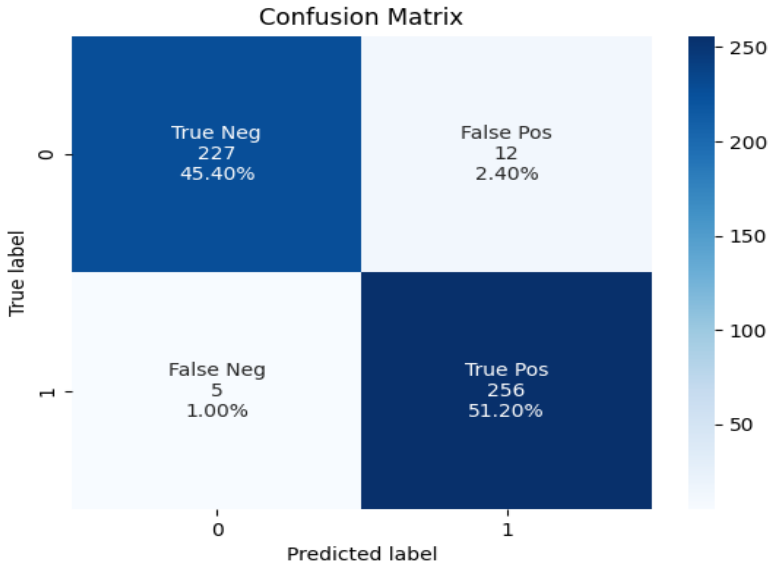


Figure 1.26: Confusion Matrix

We can immediately notice how we have higher values on the main diagonal; this is synonymous with good efficiency of the network since it is able to correctly detect the true negative labels that correspond to numbers

different than 4 (associated label 0) and the true positive labels that are instead all numbers equal to 4 (associated label 1). The metrics of evaluation are report in the figure 1.27.

Evaluation of the network:

Metric	Value(%)
Accuracy	96.6
Misclassification Rate	3.4
Precision	95.52
Sensitivity	98.08
Specificity	94.98

Number of Goodclass: 483 Number of Missclass: 17

Figure 1.27: Metrics

Furthermore, we evaluate the loss function for each epoch in order to understand if the neural network is able to properly classify our input data. The loss function's trend is crucial because provide a measure of performance of a neural network during the training. This is shown in figure:

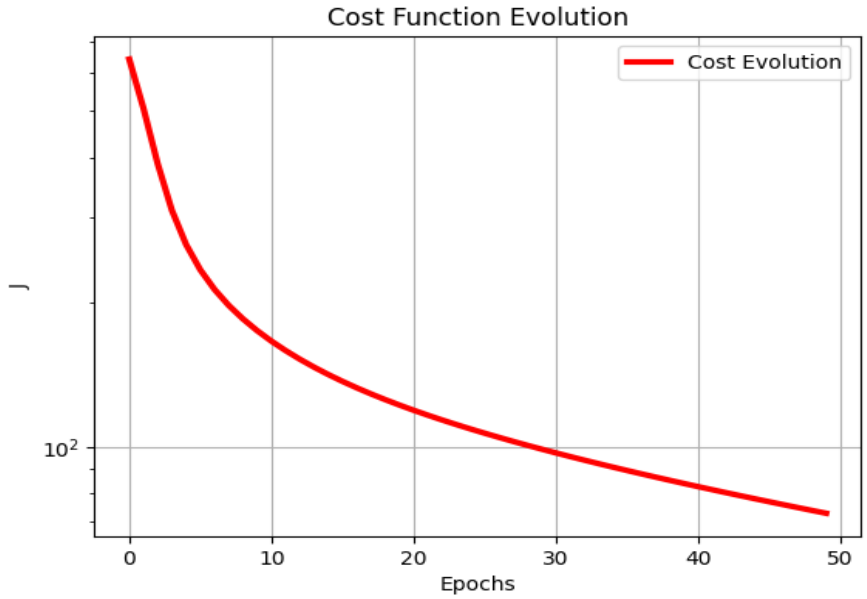


Figure 1.28: Cost Function Evolution

As we can see the loss function decrease during the train along the epochs. This means that the network is improving its predictions and it is getting closer to the desired result.

Another plot evaluated is the Norm of the Cost Function Gradient Evolution shown in figure 1.29.

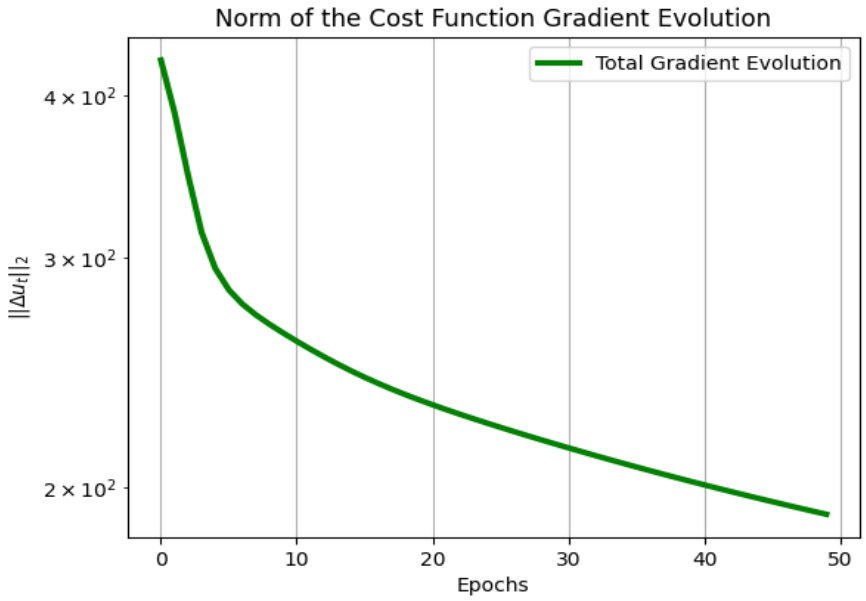


Figure 1.29: Norm of the Cost Function Gradient Evolution

This plot is important to better understand how the optimization process is progressing and how much the network weights change in response to training.

1.4 Distributed training

The last point of task 1 requires to distribute the training of the previous NN over multiple agents. We want to simulate a scenario where multiple agents aim to solve a shared classification task having the knowledge of their local data only. The data manipulation is the same of the previous point. The only difference is that in this case the train data are split across the agents. In this case the optimization is performed in a different way. We have to use a distributed optimization algorithm in order to compute the weights update.

1.4.1 Functions

To implement the distributed training we exploit the functions used for Task 1.2, which are explained in the section 1.3.4 In particular used functions are:

- *sigmoid_fn()* 1.3.4.5

- *sigmoid_fn_derivative()* 1.3.4.6
- *inference_dynamics()* 1.3.4.7
- *Forward_pass()* 1.3.4.8
- *adjoint_dynamics()* 1.3.4.9
- *BACKWARD_pass()* 1.3.4.10
- BCE() 1.3.4.11

Furthermore, for the readability of the code we create three other functions:

- *graph_generation()*: in order to get the Graph
- *Adj_matrix()*: in order to get the Adjiacency matrix
- *weighted_matrix()*: in order to get the Weighted Adjiacency Matrix

The code within each function is the same as that used in Task 1.1.

1.4.1.1 Split Data Agent Function

With this function we randomly split the data and the labels among the agents. The function is schematized in table 1.13. This function takes as input :

- NN: number of agents
- X_train: all train data available
- y_train: train labels available
- N_train_data_agent: number of train data for each agent
- show: to show the array dimension

and as output:

- X_train_agent: train data for each agent
- y_train_agent: train labels for each agent

split_data_agent()			
Input:	NN	X_train	y_train
	N_train_data_agent	show	
0utput:	X_train_agent	y_train_agent	

Table 1.13: Python function *split_data_agents()*

Furthermore, in this function we check that all dimensions are set correctly and shuffle the train data a second time before dividing it among the agents. In general, shuffling data before training improves model performance. With this function we show also the occurrences of each label for each agent.

1.4.2 Distributed Algorithm

In this scenario each agent has its own neural network (local model) to train with some local dataset (we split our training dataset among the agents), during the update phase the agents will communicate (based on some communications graph) in order to update the weights by reaching a consensual solution. So our optimization problem becomes the following:

$$\min_{\mathbf{u}} \sum_{i=1}^{\mathcal{N}} J_i(\mathbf{u}) = \min_{\mathbf{u}} \quad \sum_{i=1}^{\mathcal{N}} \sum_{r=1}^{m_i} J(\Phi(\mathbf{u}; D^r); y^r) \tag{1.35}$$

where :

- $\mathcal{N}$ is the number of agents
- $J_i((u))$ is the loss function (BCE)
- Each agent i has local private data (D^r, y^r), with $r = 1, ..., m_i$

In the project we use the minibatch, using minibatches helps reduce the frequency of communications needed. Instead of updating the model after each individual observation, updates are made after processing a batch of data. It also allows you to speed up the training process, contributes to greater stability during training and to having a more stable convergence. Taking batches into account we can rewrite the optimization problem in the following way:

$$\min_{\mathbf{u}} \sum_{i=1}^{\mathcal{N}} J_i(\mathbf{u}) = \min_{\mathbf{u}} \quad \sum_{i=1}^{\mathcal{N}} \sum_{n=1}^{NB_i} \sum_{h \in I^n} J(\Phi(\mathbf{u}; D^h); y^h) \tag{1.36}$$

where:

- NB_i is the number of batches agent i (given by the $\frac{m_i}{batch\_size}$)
- I^n is the minibatch
- D^h is the h-th sample inside the minibatch

The optimization problem can be solved via *Distributed Gradient Tracking*, in which, the weights of the network are updates as follows

$$\mathbf{u}_i^{k+1} = \sum_{j \in \mathcal{N}_i} a_{ij} \mathbf{u}_j^k - \alpha \mathbf{s}_i^k \tag{1.37}$$

$$\mathbf{s}_i^{k+1} = \sum_{j \in \mathcal{N}_i} a_{ij} \mathbf{s}_j^k + (\nabla J_i(\mathbf{u}_i^{k+1}) - \nabla J_i(\mathbf{u}_i^k)) \tag{1.38}$$

1.4.3 Results and Considerations

To evaluate the quality of the distributed model we carried out several tests, in order to find the best combination between the hyperparameters and the NN shape. Experimentation and monitoring performance are key components of finding the optimal hyperparameter configuration for distributed training. These hyperparameters can influence model behavior and training effectiveness. After several tests we noticed the following things:

- The topology of the network, i.e. how agents are connected and communicate with each other, can influence the accuracy of a distributed training model. In particular, the topology can influence the communication time between agents, which in turn can influence the convergence rate and accuracy of the model. If the graph is less connected during distributed training, the computation speed may be slower. A more connected graph allows for more efficient communication between nodes, which can lead to faster training speed in a distributed environment.

- In addition, the accuracy of a distributed training model is affected by the batch size. The batch size determines the number of data that will be propagated through the NN before the weights are updated. Is convenient to reduce the batch size as more agents are added to the network to preserve the model accuracy. A larger batch size can lead to a more accurate gradient estimate, but it can also lead to slower convergence and worse generalization performance. On the other hand, a smaller batch size can lead to faster convergence and better generalization performance, but it can also lead to a less accurate gradient estimate.

- The choice of learning rate is often coupled with the batch size. Smaller batch sizes might require a smaller learning rate to prevent divergence, while larger batch sizes might benefit from a larger learning rate.

  Small Batch Sizes: When using a small batch size, individual updates to the model are based on a small subset of the data. This can lead to more frequent but noisy updates. In such cases, a smaller learning rate might be beneficial to ensure that the model doesn't oscillate too much and converges steadily.

  Large Batch Sizes: With larger batch sizes, updates are based on a more comprehensive view of the data, potentially leading to smoother convergence. Larger batch sizes might tolerate a higher learning rate, as the updates are more stable and less affected by individual data points.

- Reducing the batch size during the neural networks training, it is advisable to increase the number of epochs to compensate for delivering the amount of data processed at each iteration. This is because the batch size affects the stability of updating model weights during optimization. In general, it is important to balance the batch size, number of epochs, and data splitting among agents to achieve optimal results in a distributed training context. Reducing the step size, the computation time can increase. The step size, or learning rate, is a hyperparameter that controls the size of the steps that the model takes during optimization. If the step size is smaller, the model will take smaller steps towards convergence, which may require more iterations or epochs to reach the same accuracy. A smaller step can make the optimization more precise, but at the cost of requiring more computational time.

Below are some plots that represent the results of the distributed training. We test our algorithm considering the following neural network configuration:

$$d = [784, 8, 1] \tag{1.39}$$

with the following hyperparameters:

- graph_type = Cycle (Binomial ,Path ,Star ,Cycle, Complete)
- NN = 10

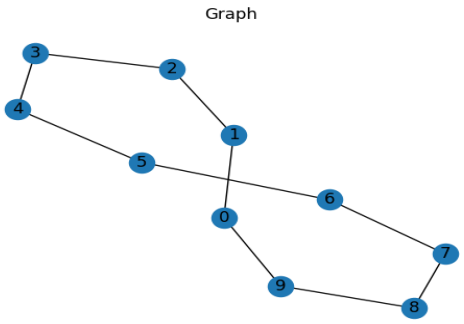


Figure 1.30: Cycle graph during simulation

- AGENT_train_data = 512
- MAX_EPOCHS = 200
- batch_size = 128
- step_size = 5e-3
- initialization: we initialize the weights with random values from a normal distribution. With np.random.randn we generate random numbers and then divide them by 10 for scaling.

Compared to the previous point (Centralized training) we use a hidden layer with fewer neurons. In a distributed context, each agent can have a simpler network structure than in the centralized model. For debugging and anomaly detection purposes we plot the accuracy of each agent on its train set during the distributed training, along the epochs. This plot can provide valuable insights into the performance and convergence of the individual agents in our distributed system.

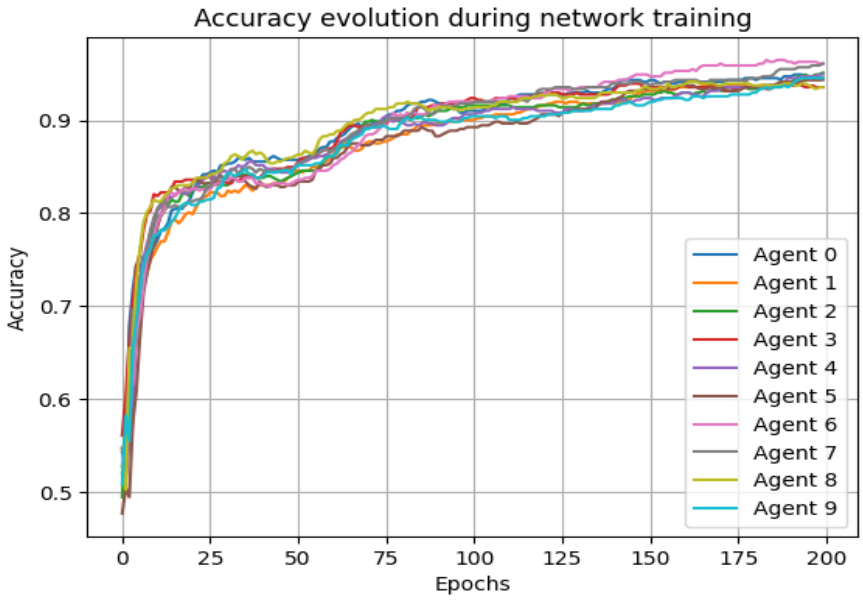


Figure 1.31: Accuracy along the Epochs

As we can see from the figure the accuracy of each agent increase along the epochs. This phenomenon is indicative of the fact that each model in the distributed network is learning from the data and improving its ability to make correct predictions. Other useful graphs to evaluate the correct behavior of our algorithm are the loss trend plot, the norm of the gradient of the loss function and the consensus error.

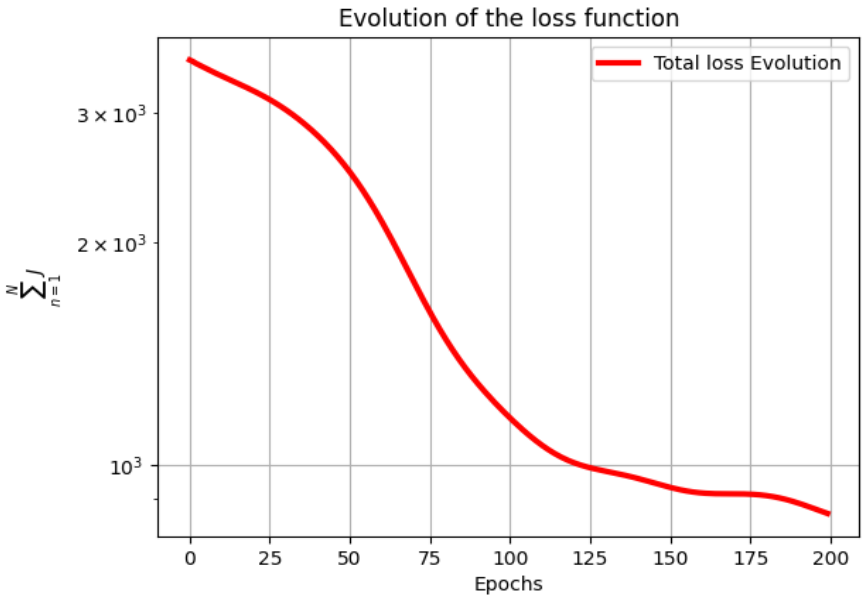


Figure 1.32: Loss along the Epochs

Plotting the loss during distributed training is crucial for several reasons: The loss plot helps to monitor the convergence of the model. A decreasing loss over epochs indicates that the model is improving and converging towards a solution. The loss plot serves as a diagnostic tool. By analyzing changes in the loss over time, we can identify when the problem occurred and investigate the cause. Loss plots provide insights into the impact of hyperparameters on the training process. Adjusting learning rates, batch sizes, or other hyperparameters based on the loss plot can improve the overall performance of the distributed training. In distributed training, where multiple agents or nodes are involved, comparing the loss plots of different agents helps identify variations in learning progress. If one agent consistently lags behind or diverges, it signals potential issues that need attention.

The norm of the gradient of the cost function is an important metric, especially in the context of distributed systems and optimization algorithms. The norm of the gradient is often used as a convergence criterion. If the norm approaches zero, it indicates that the optimization algorithm is converging. Monitoring this value helps ensure that the distributed training process is progressing effectively toward a solution.

Figure 1.33: Norm of the gradient

In optimization algorithms like gradient descent, the step size is often adjusted based on the norm of the gradient. If the norm is large, a smaller step size may be chosen to prevent overshooting the minimum. In a distributed system, coordinating the step size across multiple nodes can be crucial for efficient convergence.

The consensus error plot in distributed training is a visual representation of how well the models across different nodes or agents agree on the learned parameters. The consensus error measures the difference or discrepancy in the model parameters among the distributed nodes.

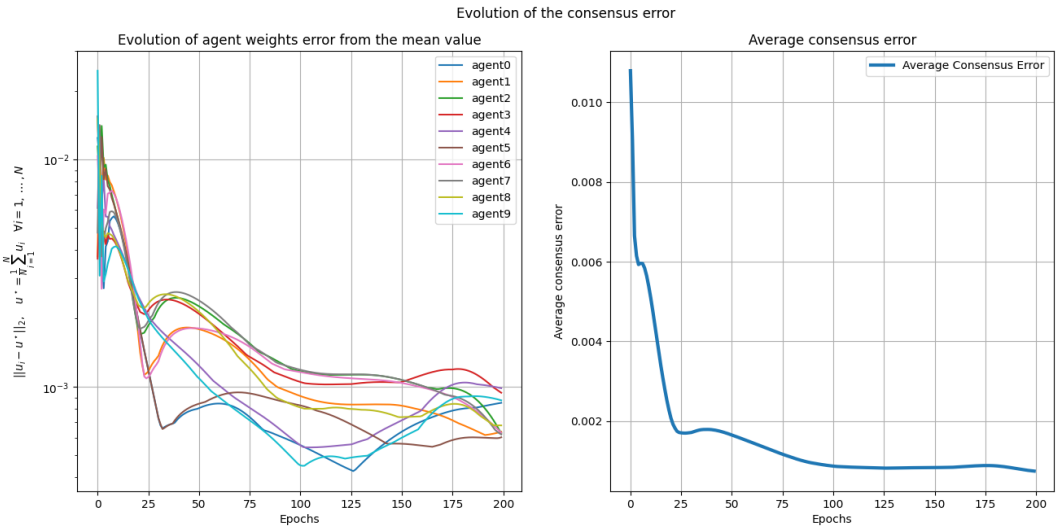


Figure 1.34: Consensus Error.

In distributed training, it's crucial to ensure that models on different nodes are synchronized and converge to a similar solution. The consensus error plot helps us to monitor the degree of agreement among the models.

A consistent decrease in the consensus error over epochs indicates that the models on different nodes are converging and becoming more consistent. This is a positive sign that the distributed training is working effectively.

Also in this case as in the Task 1.2, we use the metrics explained in section 1.3.6, to evaluate the performance of each agent on test set.

As first step we find the error for each agent after the whole traing of the network. The following are the results:

Agent erros

Agent	Error
0	0.020967
1	0.0227932
2	0.0260119
3	0.0272809
4	0.0243022
5	0.0185831
6	0.0212947
7	0.0262797
8	0.0261492
9	0.022696

Figure 1.35: Agents errors.

Then we compare the accuracy on each agent's training data to the accuracy on the total test data:

- High accuracy on training data: this may indicate that the model has learned well from the training data, but is not a sure indicator of generalization ability because a model may fit too much to training data (overfitting) and therefore perform poorly on new data.
- High accuracy on test data: indicates that the model is able to generalize well on new data, which it did not see during training. This is the desired result because the main goal of a model is to make accurate predictions on new data.

As we can see from the figure 1.36 we obtain high accuracies in both cases, this means that the distributed training was successful and that each agent, reaching consensus, is able to provide correct predictions on new data.

Agent Accuracies

Agent	Accuracy on train set per Agent	Accuracy on test set
0	94.53 %	95.18 %
1	94.53 %	95.18 %
2	95.12 %	95.18 %
3	93.55 %	95.18 %
4	94.92 %	95.18 %
5	94.34 %	95.18 %
6	96.09 %	95.18 %
7	96.09 %	95.18 %
8	93.55 %	95.18 %
9	94.53 %	95.18 %

Figure 1.36: Accuracy of each agent on its own train data set VS Accuracy on common test data for all agents.

We report results only for agent 0 because all agents reach consensus, so everyone's performance is the same. We consider the performance of agent 0 on the test data, that are common for each agent.

From the figure 1.37 we can see a very good accuracy of 95.18%, and a low misclassification rate of 4.82%. The accuracy of 96.31% indicates that the majority of positive predictions are correct, while the sensitivity of 94.13% suggests a good ability to detect true positives. Furthermore, a specificity of 96.27% indicates good control of true negatives. These metrics are robust and indicate that the model generalizes well to unseen data. The high accuracy, sensitivity and specificity suggest that the agent is capable of handling different cases accurately.

Metrics Table agent 0

Metric	Value(%)
Accuracy	95.18
Misclassification Rate	4.82
Precision	96.31
Sensitivity	94.13
Specificity	96.27

Number of Goodclass: 1402 Number of Missclass: 71

Figure 1.37: Metrics Agent 0.

We evaluate also the confusion matrix reported in figure 1.38. Also in that case we observe the good performace of the agent 0, since the values on the main diagonal are higher with respect those out.

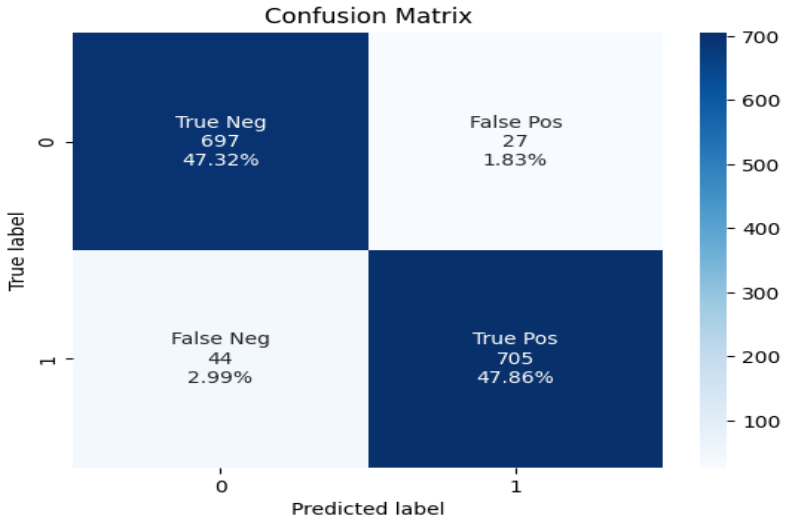
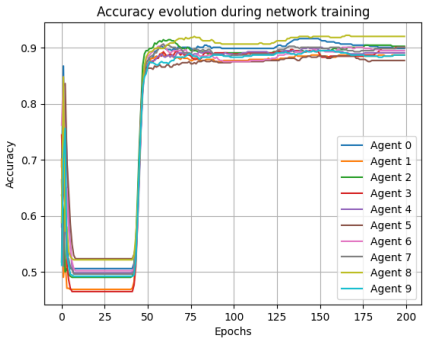


Figure 1.38: Confusion Matrix Agent 0.

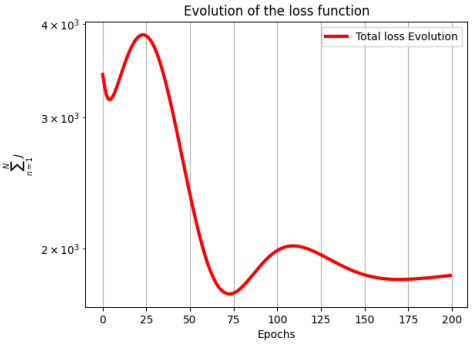
1.4.3.1 Different initializations

During the project development we understand that the weights plays a core role in the training of neural networks. So, in our code we provide the possibility to change the initializations of the NN of each agent. To better understand this aspect we carried out some simulations using a different inizialization by exploiting Xavier/Glorot one. What we noticed is that to maintain a good performance and accuracy it is necessary to decrease the

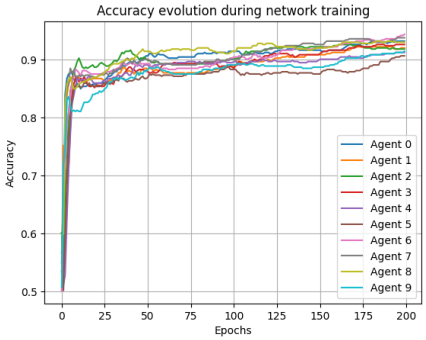
batch size. Therefore in the figure 1.39 we can see different results compared to the previous initialization technique.



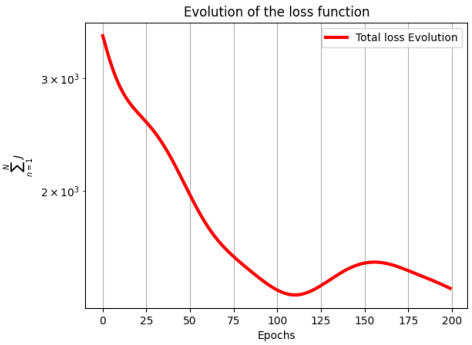
(a) Accuracy during the training with Xavier/Glorot inizialization



(b) Evolution of the loss



(c) Accuracy with Xavier/Glorot inizialization using 64 as batch size



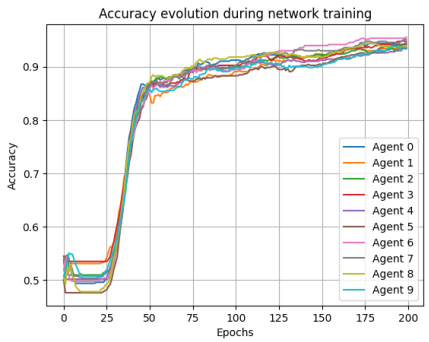
(d) Evolution of the loss

Figure 1.39: Results using a different inizialization.

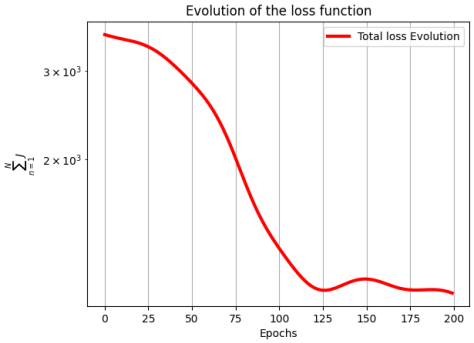
-

1.4.3.2 Batch size from 128 to 64

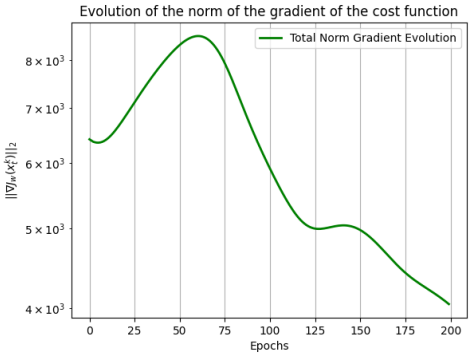
In this simulation we use the starting initialization, we only change the batch size from 128 to 64, to understand how the batch size affects distributed training.



(a) Accuracy during the training



(b) Evolution of the loss



(c) Norm of the gradient

Figure 1.40: Results using a different batch size from 128 to 64.

Comparing the figure 1.32 with the figure 1.40 (b) we can observe that a smaller batch can lead to less stable and less consistent model updates. Thus the convergence speed is slower. In 1.40 (b) from epoch 125 the descent of the function becomes slower. This can lead to a slower convergence.

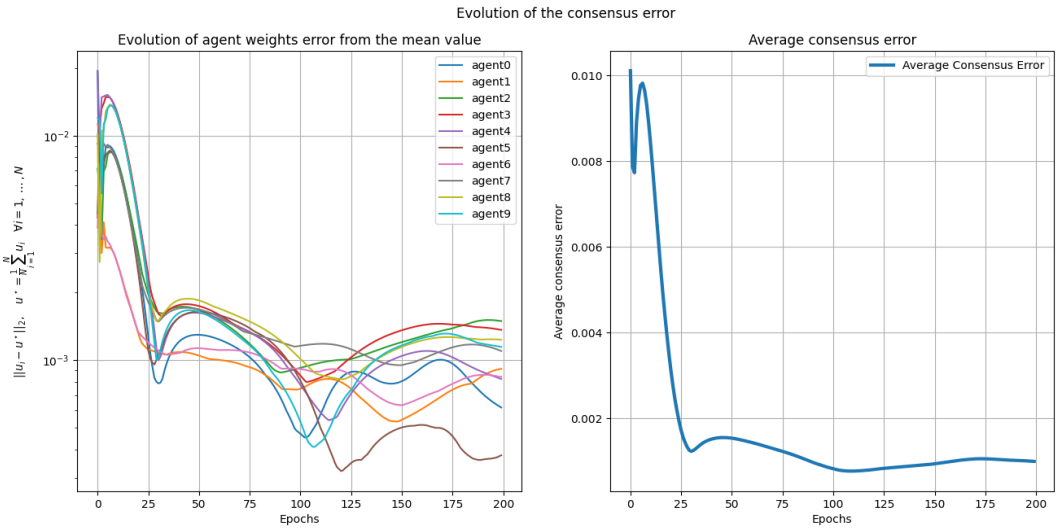
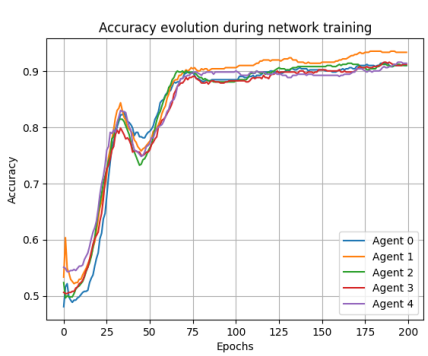


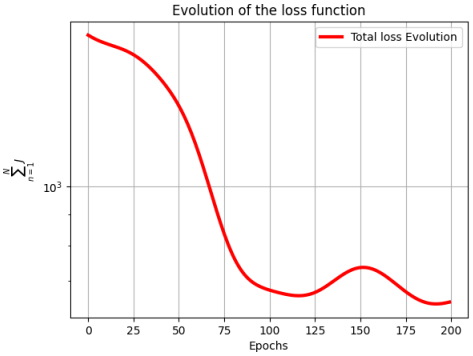
Figure 1.41: Consensus error using a different batch size from 128 to 64.

1.4.3.3 Agents reduction from 10 to 5

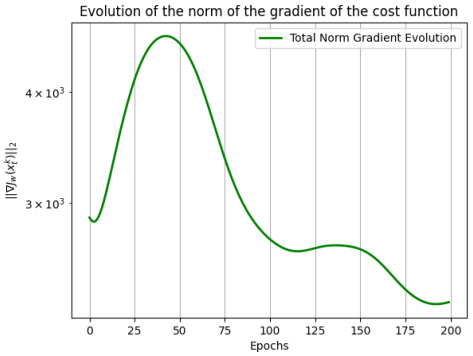
Decreasing the number of agents in a distributed training could have several effects on the training process. In particular, since there are less train data available there could be various problems such as less stability, a slowdown in the network's learning speed, overfitting and a high risk to converge into local minima. For this reason, to see how the number of agents influences the distributed training we made a simulation by reducing the number of agents from 10 to 5 without changing the other hyperparameters. We can see the results in figure 1.42.



(a) Accuracy during the training



(b) Evolution of the loss

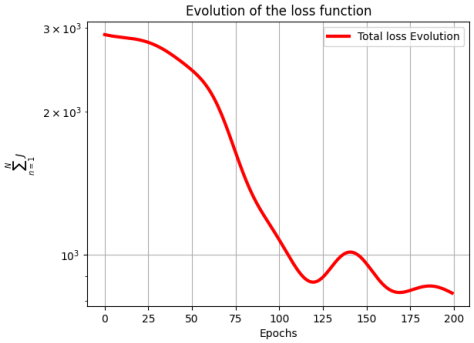


(c) Norm of the gradient

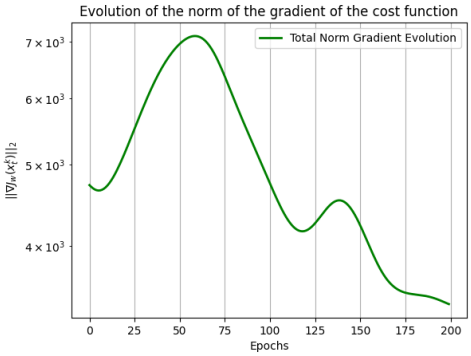
Figure 1.42: Results reducing the number of agents from 10 to 5.

Comparing figure 1.31 with figure 1.42 (a) we can notice a less stable accuracy behaviour with the presence of local minima. Then by analyzing the loss function in figure 1.42 (b) we can observe a slower descent with some oscillations with respect the one in figure 1.32. Then, we made several simulations by changing the hyperparameters in order to get a better solution with 5 agents. In particular, by increasing the number of train data for each agent and decreasing the batch size we were able to obtain greater accuracy. We can see the results in figure 1.43. A thing that we can notice is that comparing the loss function in figure 1.42 with the one figure 1.43 (b) we can observe an higher variability, this could be due to the increase in data and decrease in agents.

(a) Accuracy during the training



(b) Evolution of the loss

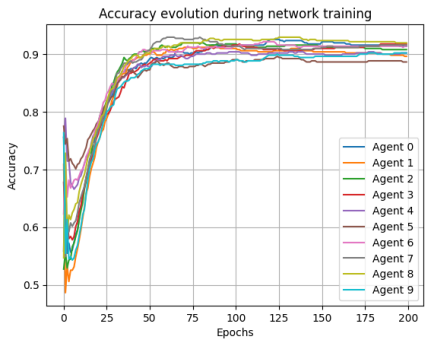


(c) Norm of the gradient

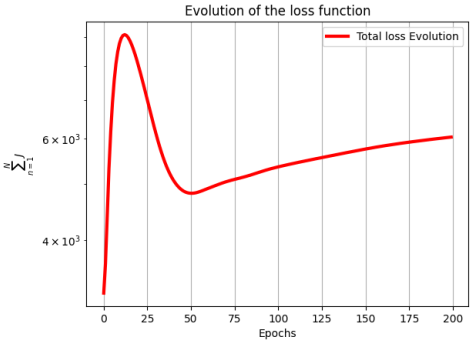
Figure 1.43: Results reducing the number of agents from 10 to 5 increasing agents' data (from 512 to 832) and reducing batch size (from 128 to 64).

1.4.3.4 Without Hidden Layer

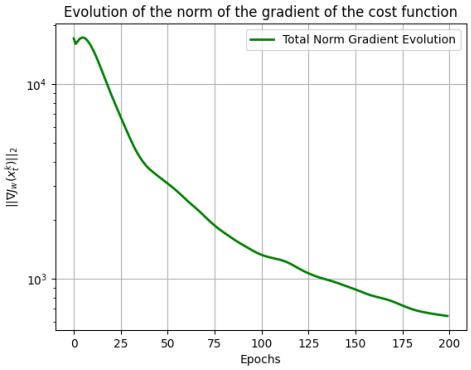
A single-layer perceptron is one of the simplest neural network architecture. It consists of a single input layer directly connected with a single output layer. This kind of NN is suitable for binary classification tasks, but it present some limitations; with this structure we are not able to handle complex problems that require non linear decision making. In other word it can only work with linear decision boundaries. To see what happen with this kind of architecture we made first a test by eliminating the hidden layer without changing the other hyperparameters. The results can be seen in figure 1.44.



(a) Accuracy during the training



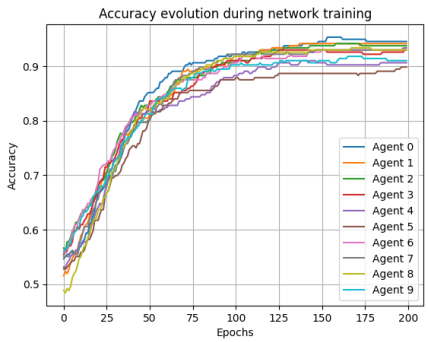
(b) Evolution of the loss



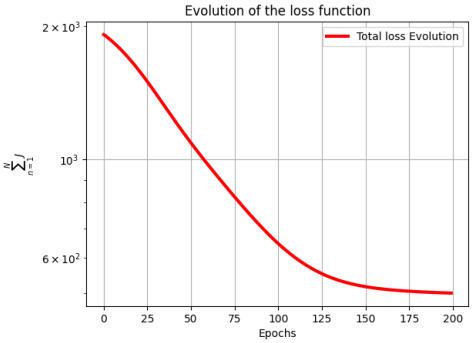
(c) Norm of the gradient

Figure 1.44: Results neural network without hidden layer.

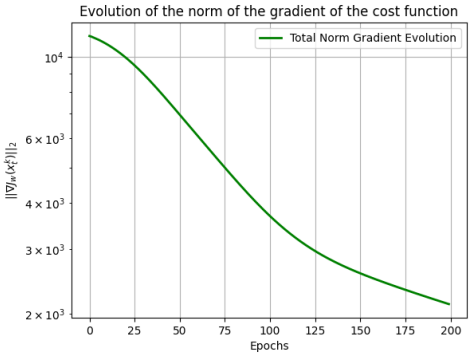
We can see from the figure 1.44 how the graphs get worse by removing the hidden layer. So, to get better solution we do some simulations changing some hyperparameters. We noticed that to obtain an acceptable solution with an high accuracy we need to reduce the number of train data for each agent, the step-size and the batch size. The results are shown in figure 1.45



(a) Accuracy during the training



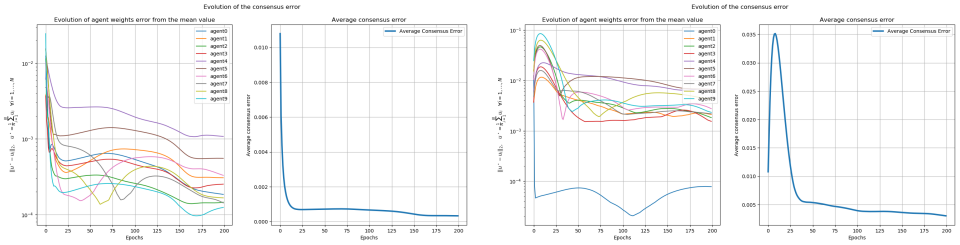
(b) Evolution of the loss



(c) Norm of the gradient

Figure 1.45: Results neural network without hidden layer reducing step size (from 5e-3 to 5e-4), batch size (from 128 to 16) and agents' data (from 512 to 256).

In conclusion, several simulations were carried out also modifying the network connection, i.e. the graph type. In our code we provide the possibility to choose between the following graphs: Cycle, Path, Binomial, Complete, Star. We observed that the change in the graph type influences the trend of the consensus error as shown in figure 1.46. The consensus error represents the discrepancy between the model weights at the various nodes.



(a) Consensus error when *Binomial* graph is used

(b) Consensus error when *Star* graph is used

Figure 1.46: Consensus error when different graphs are used.

As can be seen from the figures, the consensus error shows more pronounced oscillations in less connected graph (Star).

Chapter 2

Task 2 - Formation Control

Formation control (or flocking) is a framework related to distributed optimization, in which robots of a team coordinate their motion to maintain a desired shape. A distinctive feature of distributed optimization is that each robot knows only a part of the optimization problem and can interact only with neighboring robots. Thereby, in order to solve the problem, robots have to exchange some kind of information with neighboring robots and perform local computations based on local data [2]. We will see that from a mathematical perspective, this communication can be modeled by means of a weighted undirected graph. There are several ways to handle a formation control: formation control based on relative position, formation control based on distance, formation control based on bearing. In this project we exploit the **Distance-Based Formation control**.

2.1 Preliminaries

In this section, we introduce the main theoretical ingredients to model optimization-based multi-robot frameworks as the a formation control.

2.1.1 Graph Representations

Let's consider a set $\mathbb{I} \triangleq \{1, \ldots, N\}$ of robots/agents. In our multi-agent system, each agent can communicate with some other agents that are defined as its neighbors. Graphs are used to represent the networked systems. Communication among the agents can be modeled by means of a *weighted undirected grah* $\mathcal{G} = (\mathbb{I}, \mathcal{E}, \mathcal{W})$, where $\mathcal{W}$ is the *Weighted Adjacency Matrix* associated to graph and $\mathcal{E} \subset \mathbb{I} \times \mathbb{I}$ represents the set of edges of $\mathcal{G}$. $\mathcal{W}$ is a non-negative matrix such that the entry (i, j) of $\mathcal{W}$, denoted a_{ij}, satisfied: $a_{ij} > 0$ if $(i, j) \in \mathcal{E}$ and $a_{ij} = 0$ otherwise. A graph is said to be *undirected* if for each edge $(i, j) \in \mathcal{E}$ it stands $(j, i) \in \mathcal{E}$. Otherwise, the graph is said to be *directed* and is also called digraph. A Graph $\mathcal{G}$ models inter-robot com-

munications in the sense that there is an edge $(i, j) \in \mathcal{E}$ if and only if robot i can send information to robot j. We will denote the set of in-neighbors of agent i by $\mathcal{N}_i^{in}$, that is, the set of j such that there exists an edge $(j, i) \in \mathcal{E}$. The set of out-neighbors of agent i by $\mathcal{N}_i^{out}$ that is, the set of j such that there exists an edge $(i, j) \in \mathcal{E}$.

2.2 System and Problem Statement

The objective of distance-based formation control is to steer the agents from some initial positions to converge to a desired geometric pattern defined by inter-neighbor distances $\{d_{ij}\}_{(i,j)\in\mathcal{E}}$. Where $d_{ij} \in \mathbb{R}$ the assigned distances between two agents and $\mathcal{E}$ is the edge set.

We consider a set $\mathbb{I} \triangleq \{1, \ldots, N\}$ of robots/agents communicating according to a fixed, undirected graph $\mathcal{G} = (\mathcal{I}, \mathcal{E})$. We denote the position (state) of robot $i \in \{1, \ldots, N\}$ at time $t \geq 0$ with $x_i(t) \in \mathbb{R}^3$, and with $p_i^k \in \mathbb{R}^3$ its discretized version at time $k \in \mathbb{N}$. Hence, we represent with $x(t) = [x_1(t), \ldots, x_N(t)]^T \in \mathbb{R}^{3N}$ and $p^k \in \mathbb{R}^{3N}$ respectively the stack vector of the continuous and the discrete positions. The agents/robots run the following Laplacian dynamics:

$$\dot{x}_i(t) = -\sum_{j\in\mathcal{N}_i} a_{ij}(x_i(t) - x_j(t)) \quad \forall i \in 1, \ldots, N \tag{2.1}$$

being $\mathcal{N}_i$ the set of neighbours of the i-th agent (more compact notation). In this project, we deal with formation control of multi-Agent system based on Potential Function. We can rewrite it as the sum of derivatives of local potential function (Energy) V_{ij}:

$$\dot{x}_i(t) = -\sum_{j\in\mathcal{N}_i} \frac{\partial V_{ij}(x(t))}{\partial x_i} \quad \forall i \in 1, \ldots, N \tag{2.2}$$

A natural potential function for agents i and agent j in $\mathbb{R}^3$ frequently used in distance-based formation control is:

$$V_{ij} = \frac{1}{4}(\|x_i(t) - x_j(t)\|^2 - {d_{ij}}^2)^2 \quad \forall i \in 1, \ldots, N \tag{2.3}$$

where $x_i(t)$ and $x_j(t)$ is the actual distance of agent i and agent j, while d_{ij} is the desired distance between them. As consequence, for each robot i we have:

$$\dot{x}_i(t) = f_i(x(t)) = -\sum_{j\in\mathcal{N}_i} (\|x_i(t) - x_j(t)\|^2 - {d_{ij}}^2)^2 (x_i(t) - x_j(t)) \tag{2.4}$$

Remarks: The above nonlinear formation control dynamics may have also other equilibria besides the one in which agents are at the assigned distances

(it can be possible to show that it is locally asymptotically stable). In particular, $x_i(t) = x_j(t)\ \forall i, j$ is also an equilibrium.

$$\dot{x}_i(t) = -\sum_{j \in \mathcal{N}_i} \left(\|x_i(t) - x_j(t)\|^2 - {d_{ij}}^2 \right)^2 \underbrace{(x_i(t) - x_j(t))}_{if\ x_i(t)=x_j(t) \Rightarrow \dot{x}_i(t)=0} \tag{2.5}$$

That's not what we want. If we start from a condition in which the agents are too close to each other it could happen that our distributed control law, instead of obtaining the desired information, could cause all the agents to collapse on each other. To avoid the problem we can consider the *Barrier potential function* $V_{ij}(x)$ such that $\lim_{\|x_i(t)-x_j(t)\|\to 0} V_{ij}(x(t)) = +\infty$. Similarly, Barrier potential function can be used to avoid obstacle.

An important step is the discretization that allows to obtain the discrete time dynamics. The simplest procedure to approximate a continuous dynamic system with a discrete one is the application of *Forward Euler method*, that is defined as follows:

$$\dot{x}(t) \approx \frac{p^{k+1} - p^k}{\Delta}$$

From the previous we obtain the discretized continuous time dynamics by $\Delta \in \mathbb{R}$ sampling time.

$$\boxed{p^{k+1} = p^k + \Delta \dot{x}(t) = p^k + \Delta f_i(p^k)}$$

In explicitly way :

$$\begin{bmatrix} p_1^{k+1} \\ p_2^{k+1} \\ p_3^{k+1} \\ \vdots \\ p_N^{k+1} \end{bmatrix} = \begin{bmatrix} p_1^{k} \\ p_2^{k} \\ p_3^{k} \\ \vdots \\ p_N^{k} \end{bmatrix} + \Delta \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \dot{x}_3(t) \\ \vdots \\ \dot{x}_4(t) \end{bmatrix}$$

In other words we get the evolution in time of the discretized system:

$$p^{k+1} = f(x^k, u^k)$$

The solutions of the obtained system approximate the solutions of the continuous system. The difference between the two solutions, i.e. the discretization error, is a rapidly increasing function with the step size Δ. For this reason we have to select a small step size, in particular, we have chosen $\Delta = 5e^{-3}$.

CODE NOTATION: for readability reason in our code we don't use p but x.

2.2.1 Potential function

As we said previously, we consider a formation control of multi-Agent system based on potential function. The potential function V_{ij} is a nonnegative function of the distance $x_{ij} = ||x_i - x_j||$ between agents i and j , such that:

1. $V_{ij}(x_{ij}) \to \infty \quad as \quad x_{ij} \to \infty$
2. V_{ij} attains its unique minimum when agents i and j are located at a desired distance d_{ij}.

The potential function V_{ij} between the agent i and agent j is shown in figure 2.1, while the repelling and attractive forces ∇V_{ij} are expalined below:

- as $x_{ij} < d_{ij}$, $\nabla V_{ij} < 0$ means agent i will repel agent j;
- as $x_{ij} > d_{ij}$, $\nabla V_{ij} > 0$ means agent i will attract agent j;
- as $x_{ij} = ||x_i - x_j|| = d_{ij}$, $\nabla V_{ij} = 0$ means the forces between agent i and the agent j can be balanced. There are no attraction and repelling between them and V_{ij} is minimum at the same time. So we have a global minimum when $||x_i - x_j|| = d_{ij}$.

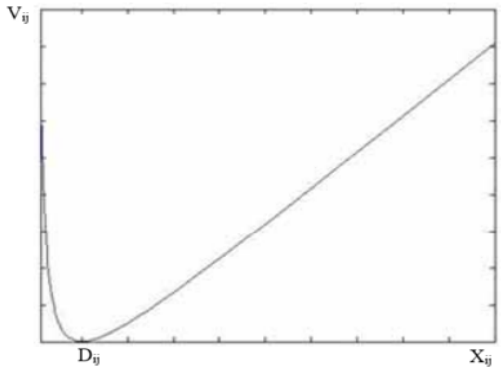


Figure 2.1: Potential function [3]

2.3 Formation Patterns

The reference formation is specified in terms of desired distances between any pair of agents and can be treated as a given rigid body. To model the desired formation of the agents we use $\mathcal{G} = (\mathbb{I}, \mathcal{E}, \mathcal{W}, \mathcal{D})$. Where $\mathbb{I}, \mathcal{E}, \mathcal{W}$ are the same described in the previous section, $\mathcal{D}$ is the *desired distance matrix*, shows the desired distance between neighbor agents:

$$\{D_{ij}\}_{(i,j)\in\mathcal{E}} = \begin{cases} d_{ij} & j \in \mathcal{N}_i^{in} \\ 0 & otherwise \end{cases} \tag{2.6}$$

where $\{D_{ij}\}_{(i,j)\in\mathcal{E}}$ is an element of the desired distance matrix and d_{ij} is the desired distance value chosen between agent i and agent j [3]. As required in the assignment we define different formation patterns with a different number of agents. In order to obtain minimally rigid formations we exploit the following relation:

- for 2D formations:

$$2N - 3 \tag{2.7}$$

- for 3D formations

$$Nn - \frac{n(n+1)}{2} \tag{2.8}$$

2.3.1 Pyramid

As first example of formation we chose the pyramid with square base 2.2. The formation setting are reported in table 2.1.

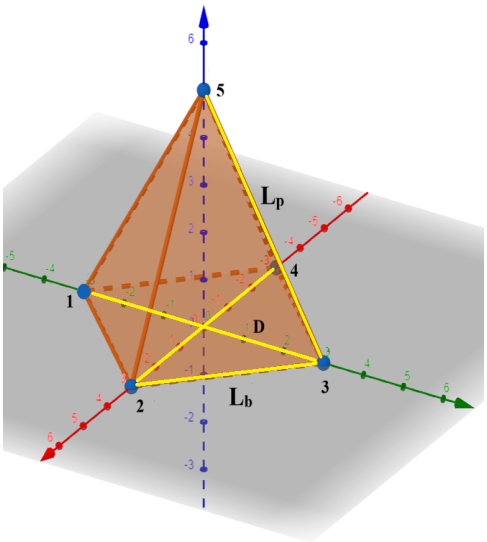


Figure 2.2: Pyramid formation

Piramid with square base		
Agents	NN	5
Base edge	L_b	2
Diagonal base	D	$L_b\sqrt{2}$
Height	H	3
Lateral edge	L_p	$\sqrt{H^2 + {L_b}^2/2}$

Table 2.1: Piramid with square base formation setting

$$\mathcal{D} = \begin{bmatrix} 0 & L_b & D & 0 & L_p \\ L_b & 0 & L_b & D & L_p \\ D & L_b & 0 & L_b & L_p \\ 0 & D & L_b & 0 & L_p \\ L_p & L_p & L_p & L_p & 0 \end{bmatrix} \in \mathbb{R}^{NN \times NN} \tag{2.9}$$

2.3.2 Cube

As second example of formation we chose the cube 2.3. In this case we have a greater number of agents with respect the previous case. The geometry of the formation is described in table 2.2.

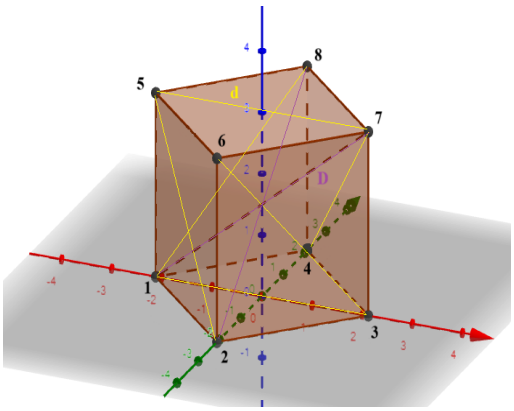


Figure 2.3: Cubic formation

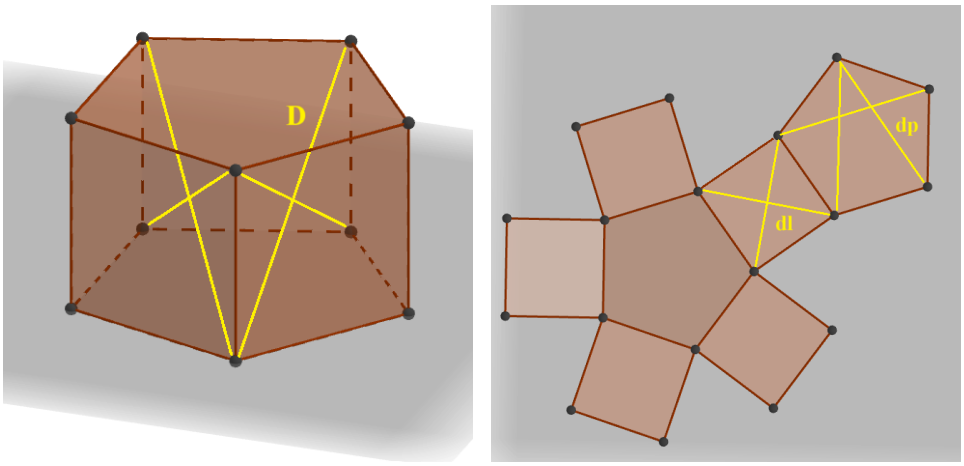
Cube		
Agents	NN	8
Base edge	L_b	3
Diagonal base	d	$L_b\sqrt{2}$
Diagonal cube	D	$L_b\sqrt{3}$

Table 2.2: Cubic formation setting

$$\mathcal{D} = \begin{bmatrix} 0 & L_b & d & L_b & D & 0 & L_b & d \\ L_b & 0 & L_b & 0 & d & D & 0 & L_b \\ d & L_b & 0 & L_b & L_b & d & D & 0 \\ L_b & 0 & L_b & 0 & 0 & L_b & d & D \\ D & d & L_b & 0 & 0 & L_b & d & L_b \\ 0 & D & d & L_b & L_b & 0 & L_b & 0 \\ L_b & 0 & D & d & d & L_b & 0 & L_b \\ d & L_b & 0 & D & L_b & 0 & L_b & 0 \end{bmatrix} \in \mathbb{R}^{NN \times NN} \tag{2.10}$$

2.3.3 Pentagonal prism

As last example of formation we chose the pentagonal prism. The geometry of the formation is described in 2.4. In particular we consider a special case where each edge has the same length. This formation requires a greater number of agent to 10.



(a) Pentagonal prism formation (b) imposed diagonals

Figure 2.4: Pentagonal prism formation.

Pentagonal prism		
Agents	NN	10
Edge	L	3
Height	$H = L$	3
Diagonal pentagon	d_p	$L(\sqrt{(5)}+1)/2$
Diagonal prism	D	$\sqrt{L^2 + {d_p}^2}$
Diagonal lateral square face	d_l	$L\sqrt{2}$

Table 2.3: Pentagonal prism formation setting

$$\mathcal{D} = \begin{bmatrix} 0 & L & d_p & d_p & L & L & d_l & D & D & d_l \\ L & 0 & L & 0 & d_p & d_l & L & d_l & D & D \\ d_p & L & 0 & L & 0 & D & d_l & L & d_l & D \\ d_p & 0 & L & 0 & L & D & D & d_l & L & d_l \\ L & d_p & 0 & L & 0 & d_l & D & D & d_l & L \\ L & d_l & D & D & d_l & 0 & L & d_p & d_p & L \\ d_l & L & d_l & D & D & L & 0 & L & 0 & d_p \\ D & d_l & L & d_l & D & d_p & L & 0 & L & 0 \\ D & D & d_l & L & d_l & d_p & 0 & L & 0 & L \\ d_l & D & D & d_l & L & L & d_p & 0 & L & 0 \end{bmatrix} \in \mathbb{R}^{NN \times NN} \tag{2.11}$$

In the code we also provide the possibility to choose other planar shapes as square, pentagon and letter A.

2.3.4 Results

This section shows the results obtained from the simulations performed in **Rviz** by launching the *formation.launch.py* file. Next, to better analyze the results, we consider the *plot_csv_FORMATION.py* file in which the data is represented using the Matplotlib library. In this file we show the trajectories of all agents while reaching the desired formation. To better understand how to run the files it is advisable to look at the README.txt file provided in the project folder. Let's start by considering the 2D formations first then the 3D formations.

2.3.4.1 Square Formation

The simulation results in Rviz are shown in the figure 2.5.

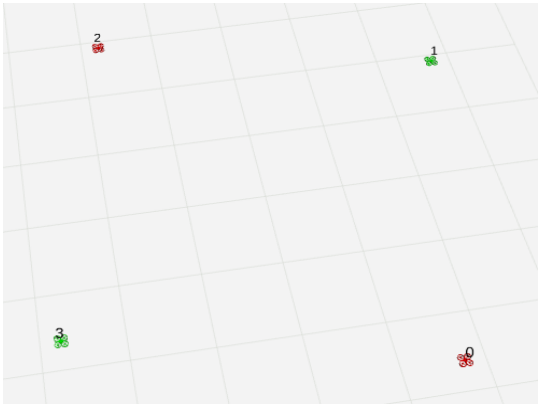


Figure 2.5: Rviz square formation

To better visualize the movement of each agent in figure 2.6 we show the trajectories while reaching the formation.

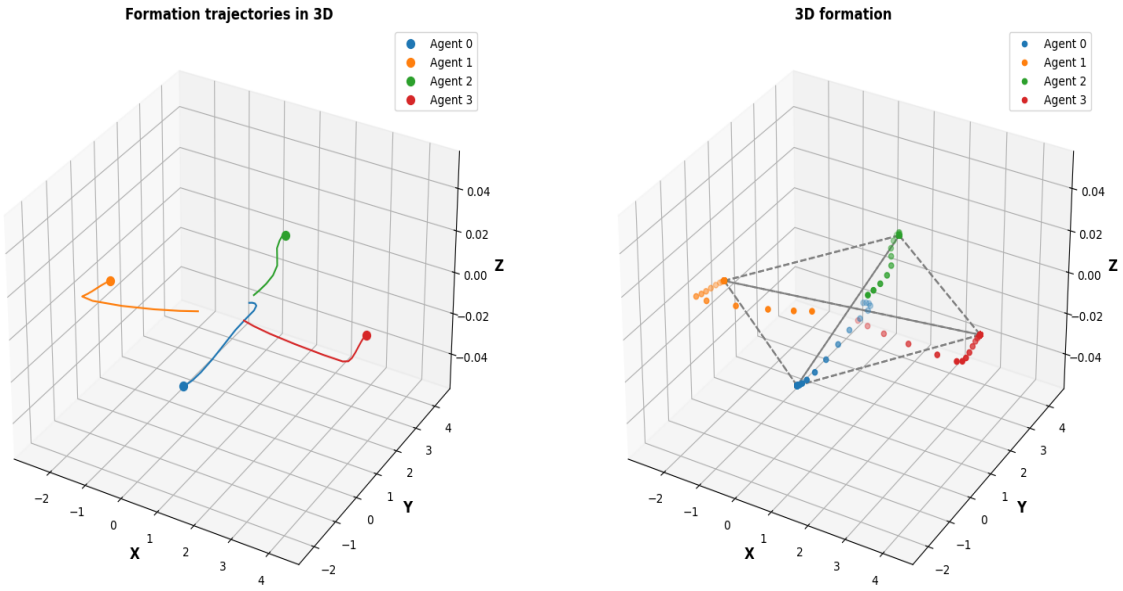


Figure 2.6: Square formation trajectories

2.3.4.2 Pentagonal Formation

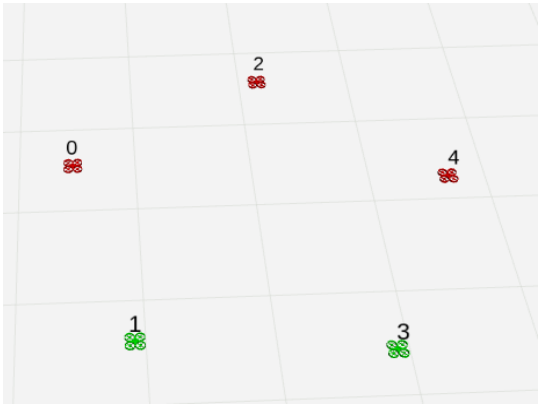


Figure 2.7: Rviz pentagonal formation

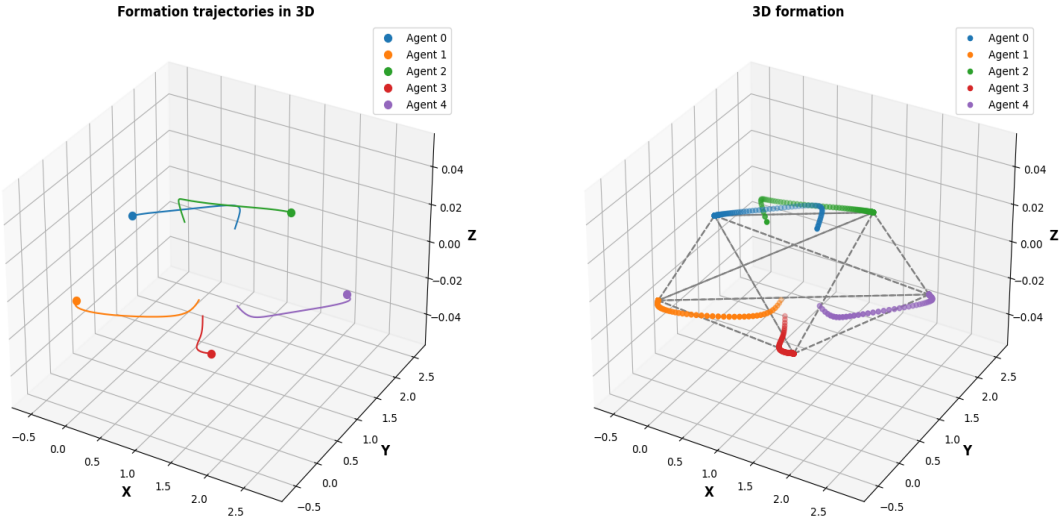


Figure 2.8: Pentagonal formation trajectories

2.3.4.3 A Formation

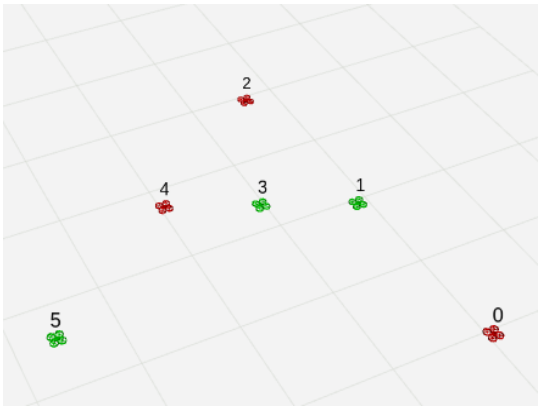


Figure 2.9: Rviz A formation

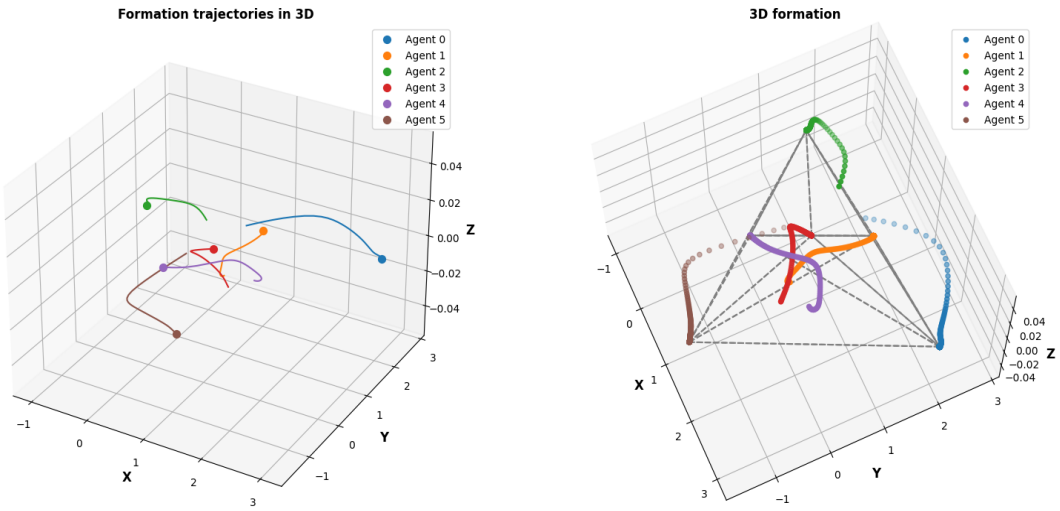


Figure 2.10: A formation trajectories

2.3.4.4 Cube Formation

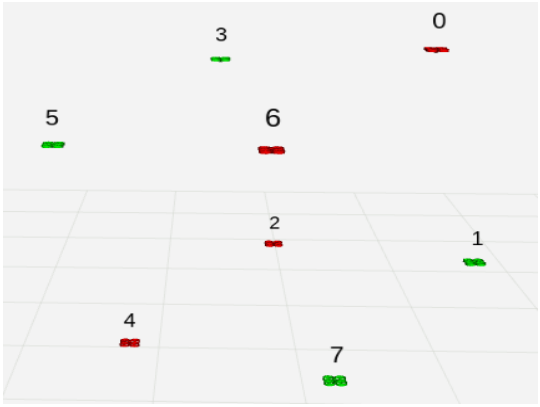


Figure 2.11: Rviz cubic formation

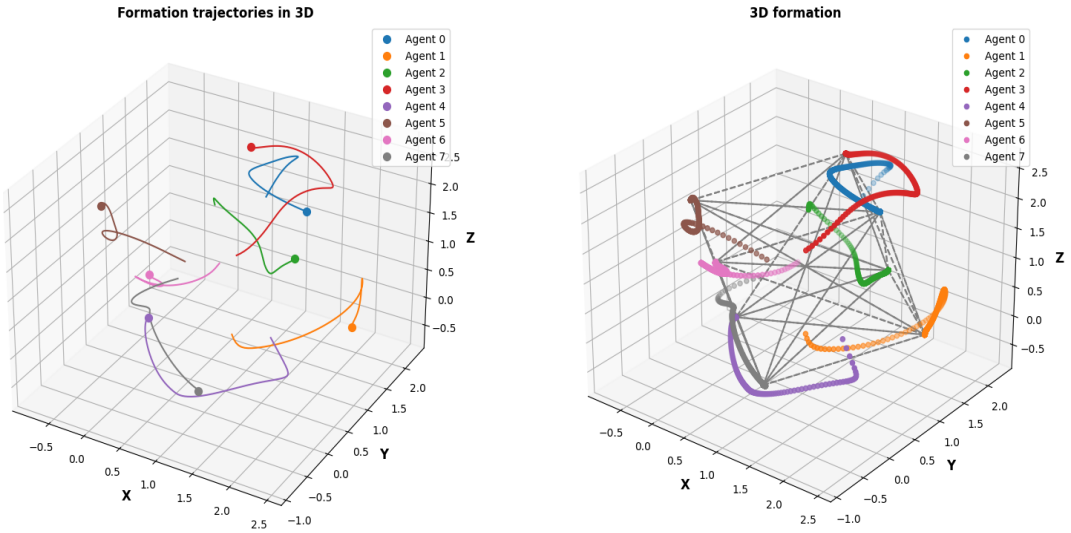


Figure 2.12: Cubic formation trajectories

2.3.4.5 Pentagonal Prism Formation

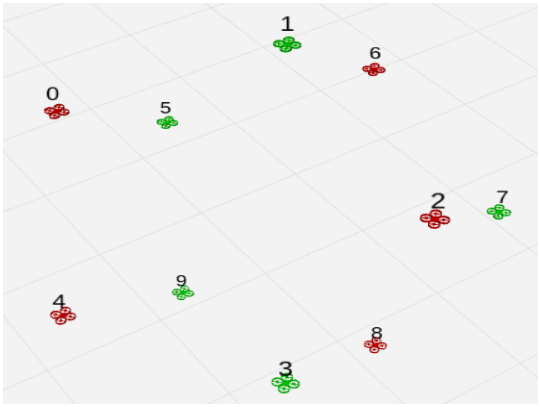


Figure 2.13: Rviz pentagonal prism formation

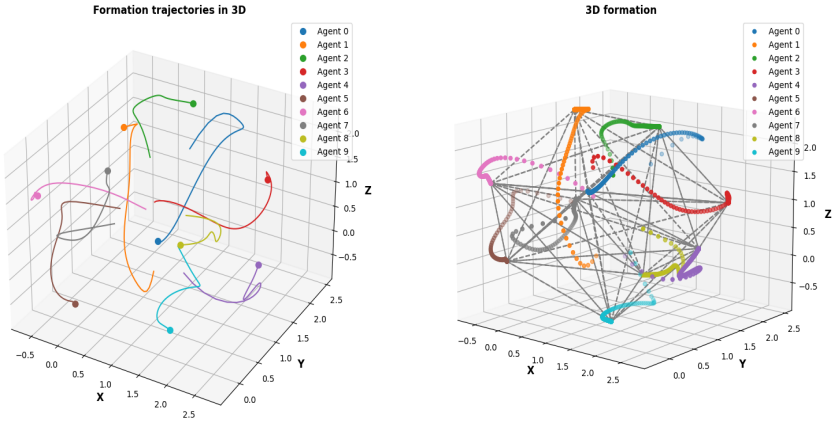


Figure 2.14: Pentagonal prism formation trajectories

2.3.4.6 Pyramid Formation

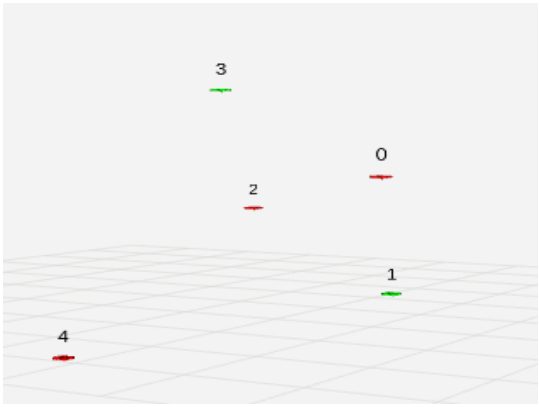


Figure 2.15: Rviz pyramid formation

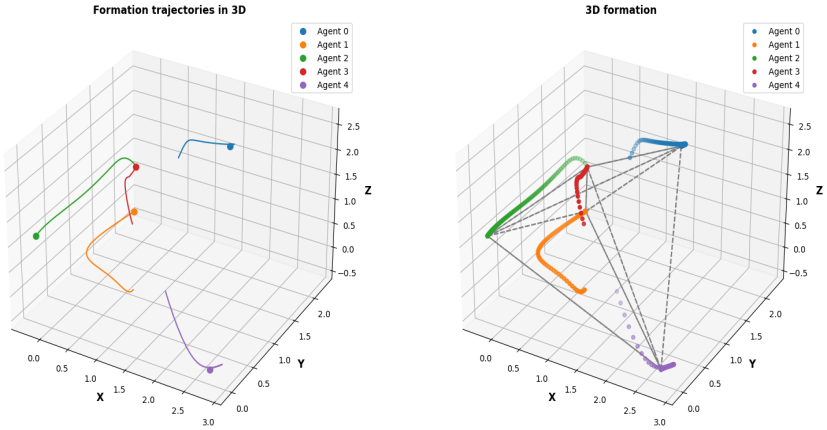
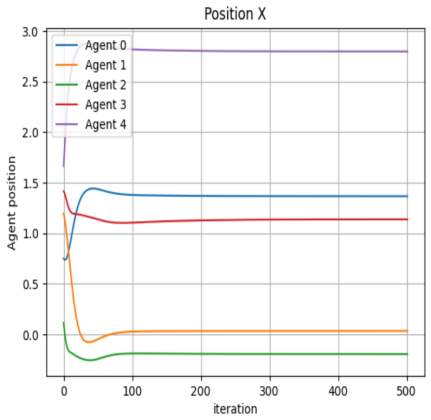
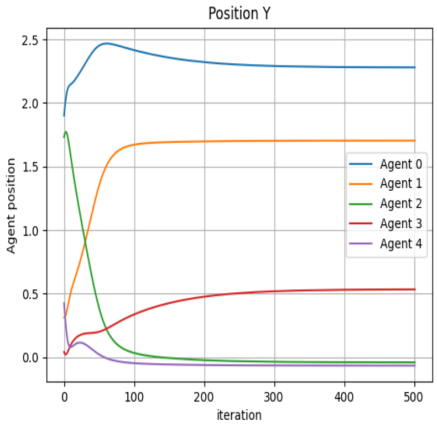


Figure 2.16: Pyramid formation trajectories

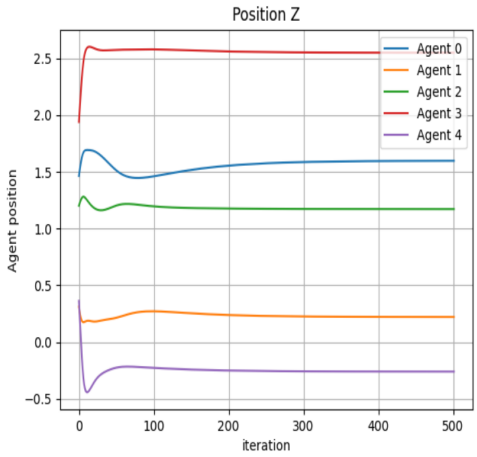
To understand whether the agents actually reach the desired formation we provide some graphs regarding the positions of the agents and the potential function (force) trends during the iterations of the algorithm. The graphs are related to the *plot_csv_FORMATION.py* file. In particular, in our code there is the possibility of drawing these graphs for each formation, below are just some of them. For example considering the pyramid formation we have the following results:



(a) x coordinate



(b) y coordinate



(c) z coordinate evolution

Figure 2.17: Agents coordinates evolution of Pyramid formation.

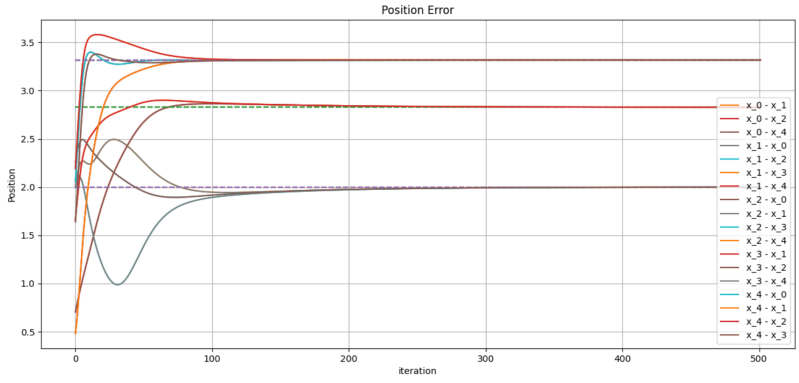
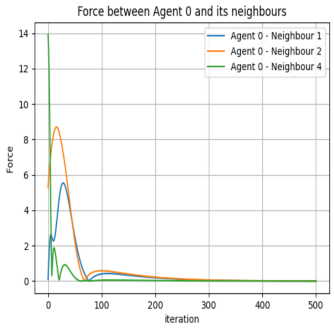
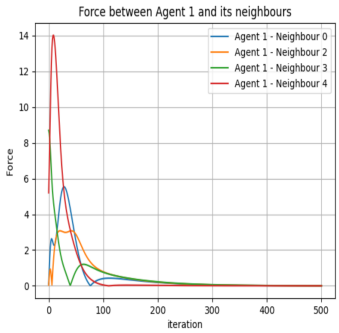


Figure 2.18: Agents position errors Pyramid formation

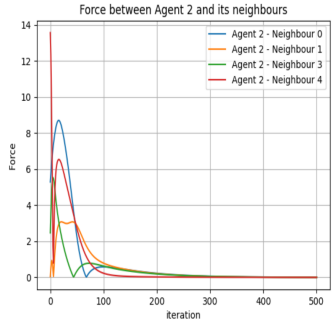
When we implement the update of the dynamic formation (in the formation_update function) we consider the derivative of the potential function 2.3 that is a force. For each agent this force depends on the distances of its neighbors.



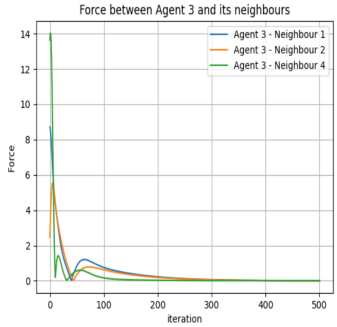
(a) Forces between agent 0 and its neighbours



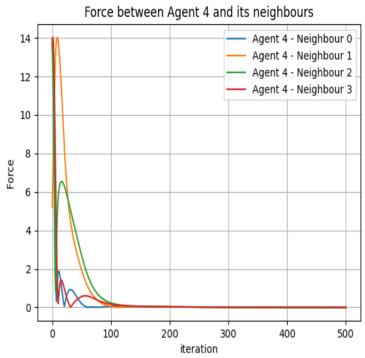
(b) Forces between agent 1 and its neighbours



(c) Forces between agent 2 and its neighbours



(d) Forces between agent 3 and its neighbours



(e) Forces between agent 4 and its neighbours

Figure 2.19: Forces between agents while achieving desired formation.

We can observe from the images that the forces between agent i and its neighbors decrease when reaching the desired distances (desired formation).

2.4 Collision Avoidance

Doing several simulations we noticed drone collisions while reaching the desired formation. This is a behavior to avoid because in real life it could lead to damage to the drones in the formation. To solve the problem we implement a modified version of the Formation Control in order to include a collision avoidance barrier functions. In this way we are able to add a repulsive potential to the total field. The repulsive potential acts as a barrier that pushes the drones away when they get too close to each other, with a force that increases as their relative distances decrease. The proposed barrier function V_{ij}^{barr} between the i-th agent and its j-th neighbour is the following:

$$V_{ij}{}^{barr}(x) = -\log(||x_i(t) - x_j(t)||^2) \tag{2.12}$$

As you can see the *disadvantage of this approach is that collisions between two non-neighboring agents cannot be prevented*, because they cannot know each other. In the dynamics of each agent i we consider the derivative of the potential 2.12, so a force. The contribution of this force contributes to the agent's dynamics only if the following condition is satisfied.

$$||x_i{}^k - x_j{}^k|| \leq 1.5 \tag{2.13}$$

So in that case we add the term:

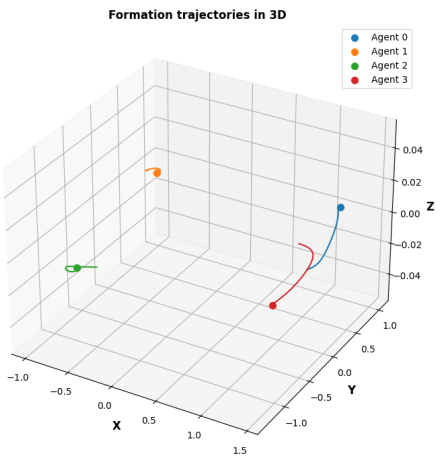
$$-\frac{2(x_i{}^k - x_j{}^k)}{||x_i{}^k - x_j{}^k||^2} \tag{2.14}$$

The dynamics of each agent $i \in 1, \ldots, N$ becomes:

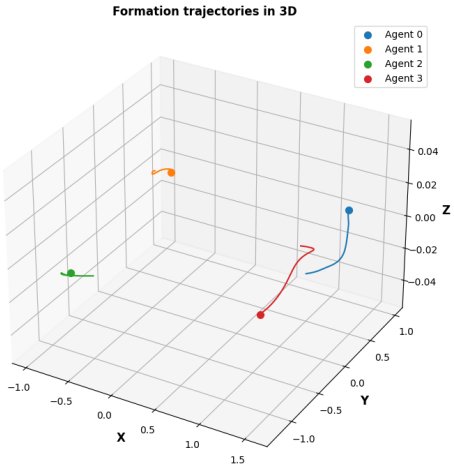
$$x_i{}^{k+1} = x_i^k - dt \sum_{j \in \mathcal{N}_i} \left((||x_i{}^k - x_j{}^k||^2 - d_{ij}{}^2)(x_i{}^k - x_j{}^k) - \frac{2(x_i{}^k - x_j{}^k)}{||x_i{}^k - x_j{}^k||^2} \right) \tag{2.15}$$

2.4.1 Results

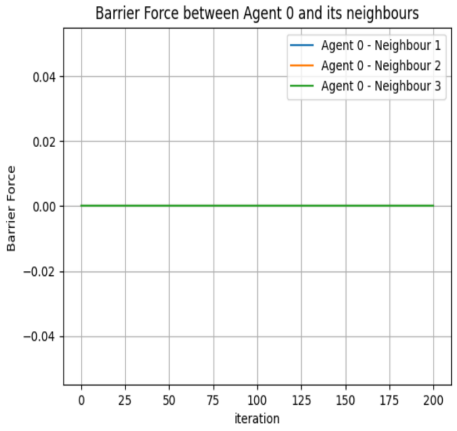
To show the effectiveness of the barrier we report below some graphs obtained by running the *plot.csv_COLLISION.py* file. To force the collision, we set the barrier_proof flag to True in the *collision.launch.py* file, in order to change the initialization, then we disable the barrier within the update_formation function. Below we consider the square one as the desired formation by creating a collision condition between agent 0 and agent 3. In the figure 2.20 we collect the trend of the barrier force



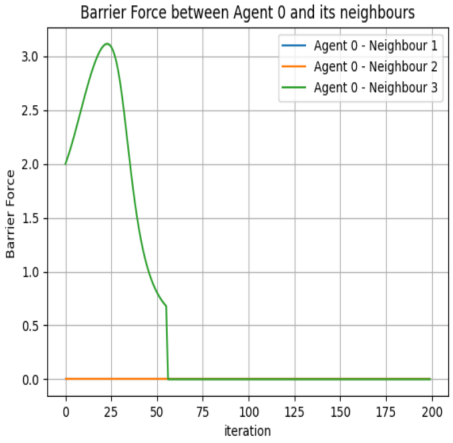
(a) Colliding trajectories



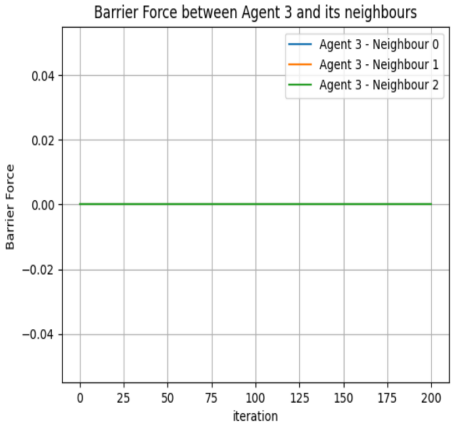
(b) Non-colliding trajectories



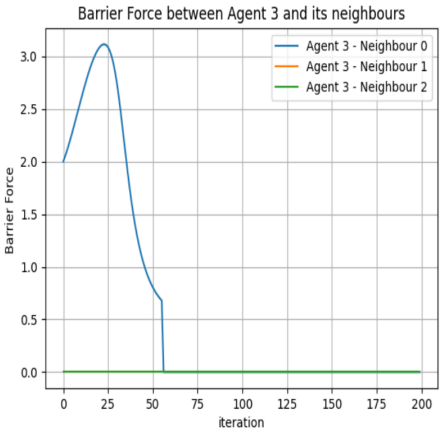
(c) No barrier force



(d) With barrier force



(e) No barrier force



(f) With barrier force

Figure 2.20: Collision proof square formation

As can be seen when the collision is avoided we have a repulsive force between agents 0 and 3.

2.5 Moving Formation and Leader control

As requested in the project assignment we carry out a moving formation. To do this, we define an agent belonging to the desired formation as a leader and implement a proportional control law to guide the formation towards the chosen target position. In this case the leader and the followers have a different dynamics. If agent i is a follower it has the following dynamics:

$$x_i{}^{k+1} = x_i^k - dt \sum_{j \in \mathcal{N}_i} \left((||x_i{}^k - x_j{}^k||^2 - d_{ij}{}^2)(x_i{}^k - x_j{}^k) - \frac{2(x_i{}^k - x_j{}^k)}{||x_i{}^k - x_j{}^k||^2} \right) \tag{2.16}$$

Instead if the agent i a leader the dynamic is:

$$x_i{}^{k+1} = x_i^k - dt \left(\sum_{j \in \mathcal{N}_i} \left((||x_i{}^k - x_j{}^k||^2 - d_{ij}{}^2)(x_i{}^k - x_j{}^k) - \frac{2(x_i{}^k - x_j{}^k)}{||x_i{}^k - x_j{}^k||^2} \right) + K_p(x_i{}^k - x^{target}) \right) \tag{2.17}$$

Where we indicate with $x^{target} \in \mathbb{R}^3$ the target position and with K_p the proportional gain. Modifying the value of the constant K_p we change the speed with which the leader agent reaches the target position. In particular we choose $K_p = 20$. Also in this case the repulsive force for collision avoidance acts only the condition 2.13 holds. We want to highlight that with the previous implementation the agents aim to reach the desired formation and then move in a coordinated manner following the leader, in this way the formation will move towards the target point following the leader's movements.

2.5.1 Different target positions

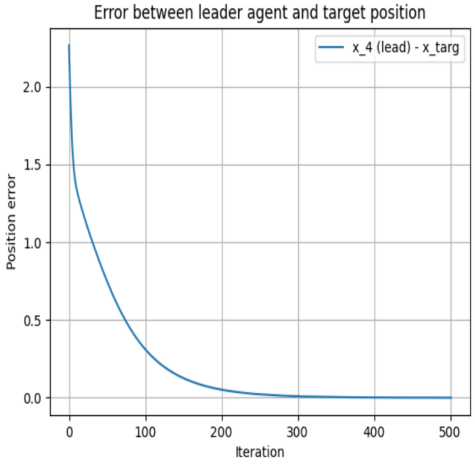
In our code we provide the possibility to run simulations by choosing different types of target points. In particular, the target point can be a fixed point or a moving point along trajectories defined in 3D space. If we choose a 2D formation the target point will be in the 2D plane otherwise by choosing a 3D formation the target point will be in the 3D space.

The possible target position are:

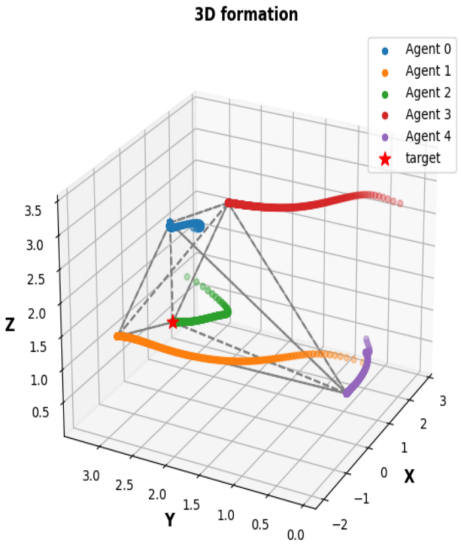
- Fixed Point
- Point moving along a circle
- Point moving along an ellipse
- Point moving along a line

2.5.2 Results

To show the correct functioning of the implemented equations, in addition to providing videos of the simulations, we provide some plots to better analyze the movement of the chosen leader and therefore of the formation. The graphs are generated via the *plot_csv_LEADER_FOLLOWER.py* file. In particular, for reasons of space, here we only report the results considering a pyramid formation and as target points a fixed point and a moving point on a circumference and on a line.



(a) Leader agent position error

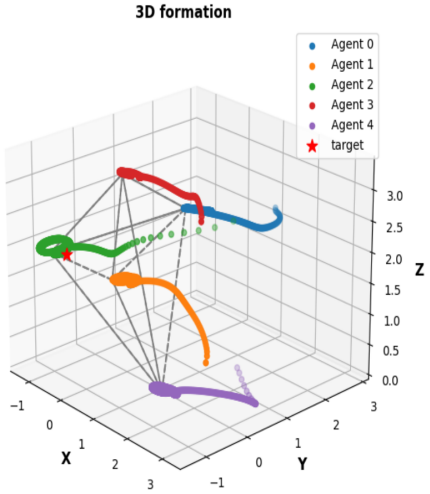


(b) Formation that reaches/follows the target position

Figure 2.21: Leader-followers results with fixed target

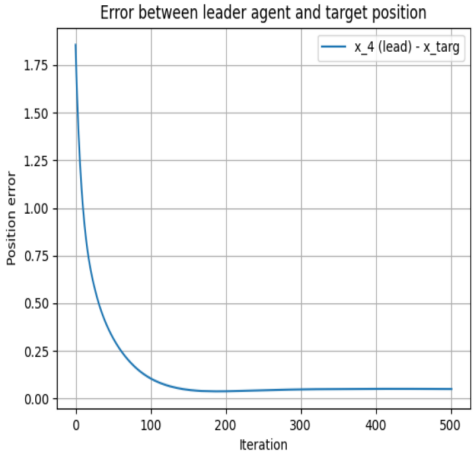
In this case 2.21, since the target is fixed, the leader reaches the target perfectly. At the end the error is zero.

(a) Leader agent position error

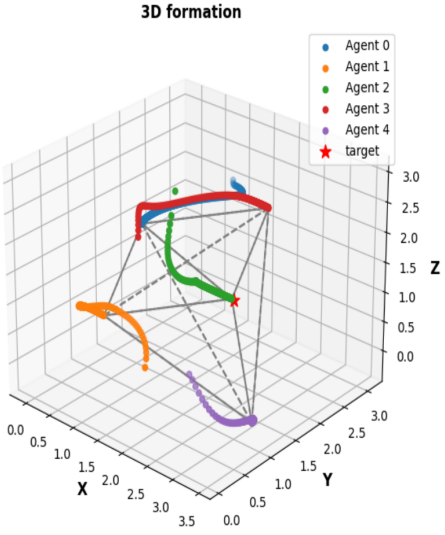


(b) Formation that reaches/follows the target position

Figure 2.22: Leader-followers results with moving target (along a circumference)



(a) Leader agent position error



(b) Formation that reaches/follows the target position

Figure 2.23: Leader-followers results with moving target (along a line)

In 2.22 and 2.23 the error is small but not exactly zero. This is because the target moves at a greater speed than the leader. In particular in 2.23 the error becomes smaller because we reduce the speed of the target.

2.6 (Optional) Obstacle Avoidance

With this point we want to add a way to avoid fixed obstacles. In particular we want that our formation reach a target point avoiding a possible obstacle. As in the previous point, we consider only one leader. To do that we consider a repulsive potential as 2.12. The structure is the same, but in this case in the formulation we consider the obstacle position x^{obst}:

$$V_{i,obst} = -log(||x_i^k - x^{obst}||^2) \tag{2.18}$$

To take into account the new potential we modify the equations 2.16 and 2.17 by adding the barrier force that allows us to avoid the obstacle. Therefore if the agent i is a follower:

$$x_i{}^{k+1} = x_i^k - dt\left[\sum_{j\in\mathcal{N}_i}\left((||x_i{}^k - x_j{}^k||^2 - d_{ij}{}^2)(x_i{}^k - x_j{}^k) - \frac{2(x_i{}^k - x_j{}^k)}{||x_i{}^k - x_j{}^k||^2}\right) - \frac{2(x_i{}^k - x^{obst})}{||x_i{}^k - x^{obst}||^2}\right] \tag{2.19}$$

if it is a leader:

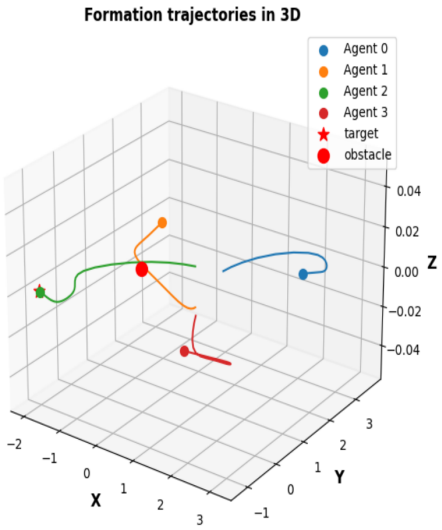
$$x_i{}^{k+1} = x_i^k - dt\left[\sum_{j\in\mathcal{N}_i}\left((||x_i{}^k - x_j{}^k||^2 - d_{ij}{}^2)(x_i{}^k - x_j{}^k) - \frac{2(x_i{}^k - x_j{}^k)}{||x_i{}^k - x_j{}^k||^2}\right) + K_p(x_i{}^k - x^{target}) - \frac{2(x_i{}^k - x^{obst})}{||x_i{}^k - x^{obst}||^2}\right] \tag{2.20}$$

Also for this case the contribution of barrier collision force and barrier obstacle force contributes to the agent's dynamics only if 2.13 and the following condition are satisfied:

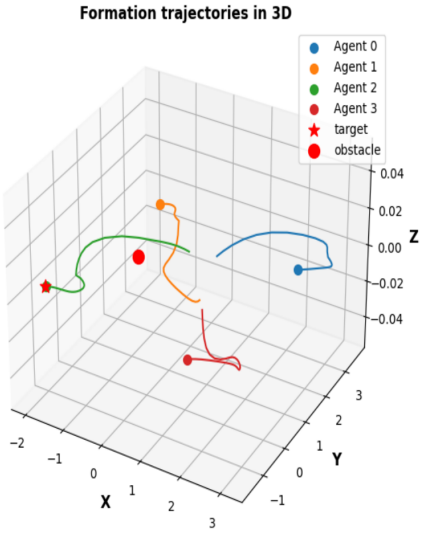
$$||x_i{}^k - x^{obst}|| \leq 1.5 \tag{2.21}$$

2.6.1 Results

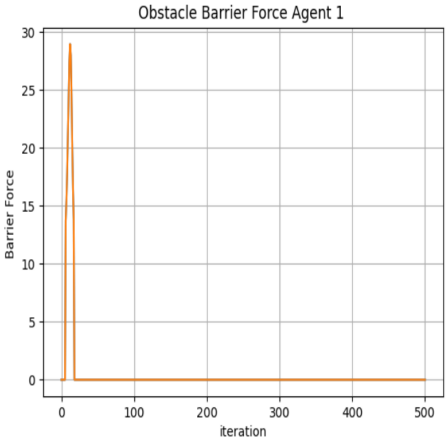
To show the correct functioning of the implemented equations we provide some videos to highlight the avoidance of the obstacle. The plots are generated via the *plot_csv_OBSTACLE_AVOIDANCE.py* file. We start with the square formation. As a first step we consider the case without obstacle barrier force. In this case a collision occurs with the obstacle as shown in the figure 2.24(a). Then we consider the case with the addition of the barrier force of the obstacle. In this case we have no collisions, as shown in 2.24(b), thanks to the added repulsion forces 2.24(c)(d).



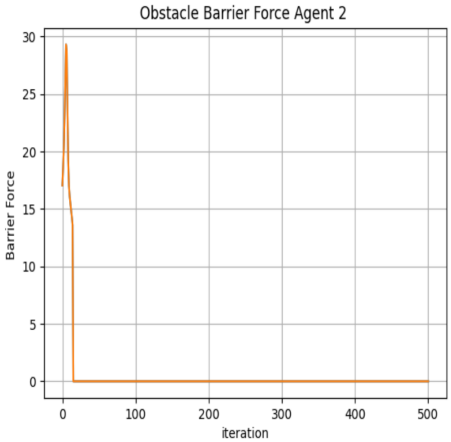
(a) collision with the obstacle



(b) No collision with the obstacle



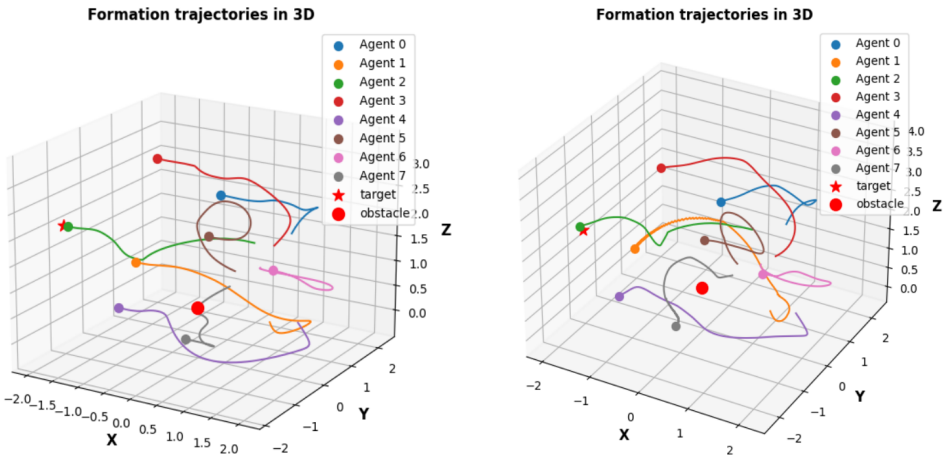
(c) Obstacle barrier force agent 1



(d) Obstacle barrier force agent 2

Figure 2.24: Obstacle proof square formation

The same things are done considering the cubic formation. In particular we focus on the movement of agent 7. As show in the 2.25 without the obstacle barrier force there is a collision with the obstacle.



(a) collision with the obstacle (b) No collision with the obstacle

Figure 2.25: Obstacle proof Cubic formation

Conclusions

The realization of the first task made it possible to understand how to implement distributed classification via neural networks. As mentioned previously, the task was achieved through three main steps. In the first stage, the gradient tracking algorithm was implemented to solve a consensus optimization problem of a quadratic cost function. In particular, a comparison was made with the distributed gradient method. By analyzing the results of several simulations with different weighted graphs, it became evident that the Gradient Tracking (GT) algorithm is more efficient and converges faster than the Distributed Gradient method. In particular, DG method requires a diminishing step size to converge. A multi-sample centralized neural network was subsequently developed to detect a selected digit from a set of images, providing practical insights into machine learning. Key learnings include preparing the dataset for neural network training and understanding neural network operations. Despite the availability of Python frameworks such as PyTorch, which provide pre-built structures for easy neural network implementation, this project involved building a neural network from scratch. This approach provided comprehensive informations on the functioning of the network, from weight initialization to the backpropagation algorithm, e offered a deeper understanding of the optimization of network parameters and their influence on network development. As a final step, a distributed neural network training was implemented by randomly splitting the train data into N subsets, one for each agent. This method demonstrated how parallel processing enhances learning efficiency and accuracy, as each agent processes a portion of data for a better understanding of the dataset. Furthermore, this approach allowed us to see the impact that different graph structures have on training process efficiency. To assess the algorithms' performance, key metrics like the confusion matrix, accuracy, sensitivity, and others were employed and analyzed. This offered a detailed evaluation of the algorithms' effectiveness and a comprehensive insight into the results, highlighting strengths and areas for improvement.

Regarding the second task, a discrete-time Formation Control algorithm was implemented, enabling various agents to form predefined shapes in both 3D and 2D spaces. Following this, a modified version of the Formation Control algorithm was developed by incorporating a collision avoidance barrier

function, ensuring the agents avoid collisions during the formation. As the final step, a leader-follower control law was established. By defining one agent as the leader, the goal was to steer the entire formation towards a specific target position. Additionally, to enhance this setup, another barrier function was integrated, facilitating the agents' formation in reaching a target point avoiding potential obstacle. Thanks to the use of RViz it was possible to visualize and understand in more detail the behavior of the agents in the execution of their tasks, allowing control strategies to be refined and optimised. However, thanks to the creation of CSV files and the use of matplotlib libray, it was possible to create detailed graphs that allowed to precisely visualize and analyze the position errors of the agents during the formations and the forces at play between them, offering a clear and detailed representation of the results obtained. Overall, this task allowed to learn in depth the management of multi-agent systems and the acquisition of significant skills in the use of ROS 2.

Bibliography

[1] Giuseppe Notarstefano, Ivano Notarnicola, and Andrea Camisa. 2019.

[2] Andrea Testa, Guido Carnevale, and Giuseppe Notarstefano. A tutorial on distributed optimization for cooperative robotics: from setups and algorithms to toolboxes and research directions, 09 2023.

[3] Li Wang, Zengqiang Chen, Zhongxin Liu, and Qiang Wang. Formation control of multi-agent system based on potential function. In *2008 Asia Simulation Conference - 7th International Conference on System Simulation and Scientific Computing*, pages 101–105, 2008.