UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Distributed Autonomous System

Distributed Learning and Formation Control

Professors:
Notarstefano Giuseppe
Notarnicola Ivano

Tutor:
Pichierri Lorenzo

Students:
Simonazzi Daniele
Musolesi Nicoló
Sartoni Marco

Academic year 2022/2023

Abstract

In the first chapter, we focus on introducing two distributed gradient descent techniques: the first one being simpler, and the second one more complex but capable of "gradient tracking". Afterwards, we apply what has been shown to the specific case of training a neural network, thereby achieving distributed training. In the last chapter, we shift gears and address the issue of formation control by implementing obstacle and collision avoidance.

Contents

Task 1

Chapter 1

Introduction to distributed optimization

Distributed optimization refers to the process of optimizing a problem in a decentralized manner, where multiple agents collaborate to find an optimal solution. In traditional optimization, a central entity gathers all the relevant information and computes the optimal solution. However, in scenarios where the optimization problem is large-scale or computationally intensive, distributed optimization becomes useful.

Among the advantages of distributed optimization, the most relevant are:

- Scalability: distributed optimization allows us to handle large-scale problems that can be very demanding on a single agent.
- Efficiency: distributed optimization can exploit parallel computation to speed up the optimization process.
- Robustness: distributed optimization is more resilient to failures. If one or more nodes in the system fail, the optimization process can continue with the remaining nodes.

1.1 Distributed optimization of a quadratic function

The distributed gradient descent algorithm we are asked to implement works for cost functions $J(x) : \mathbb{R}^n \to \mathbb{R}$ with the following properties:

- convexity
- bounded gradient: $\exists C_1 \in \mathbb{R},\ C_1 > 0 : \|\nabla J(x)\| \leq C_1 \quad \forall x \in \mathbb{R}^n$
- there exists at least one optimal solution

Thus, in order to test our implementation of the algorithm, we chose a simple case: optimize a quadratic function $J(x)$ (which satisfies all the previous properties).

$$J(x) = \sum_{i=1}^{n} J_i(x) = \sum_{i=1}^{n} \frac{1}{2} \cdot (x^\top \cdot Q_i \cdot x) + (r_i^\top \cdot x) \tag{1.1}$$

The vector $x \in \mathbb{R}^n$ represents the optimization variable and, in our code, $n = 5$. Matrices $Q_i \in \mathbb{R}^{5\times5}$ are positive definite so that the solution to the optimization problem exists (since the search space is the whole $\mathbb{R}^5$) and is unique ($Q = \sum_{i=1}^{n} Q_i$ is invertible). Vectors $r_i \in \mathbb{R}^5$ instead can be chosen as we want, no conditions on them.

The idea here is to create N agents, assign to each one of them a "piece" of the whole $J(x)$, namely $J_i(x)$, and make them collaborate to compute the optimum for $J(x)$, that we'll call x^{opt}. The difficulty lies in the fact that the generic i_{th} agent only knows[1] $J_i(x)$ and its own current estimation of x^{opt}, x_i^{opt}. At the end of the algorithm, all the agents should agree on the estimation of x^{opt} (we say that they reach consensus).

1.1.1 Theoretical background

The agents can collaborate only if they can communicate. The communication "structure" is dictated by a graph and we chose to use a binomial one. A binomial graph is a type of graph generated using a probabilistic model called the binomial distribution. This distribution is characterized by two parameters: the number of vertices N and the probability of an edge between any two vertices, p (we set $p = 0.5$). To create the binomial graph we used the function `binomial_graph()`, provided by the Python library Networkx, which contains utilities for graph creation and manipulation.

NetworkX provides another highly useful function, `to_numpy_array()`, which takes a graph $\mathcal{G}$ as input and returns the corresponding binary adjacency matrix A as output. A is a square matrix which is used to represent the presence or absence of edges in a graph:

- $A[i, j] = 1$ means there is an edge between node i and j
- $A[i, j] = 0$ means there is no edge between node i and j

In consensus algorithms it is customary to use a "modified" version of A, which is called "weighted adjacency matrix W". In essence, the weighted adjacency matrix W extends the binary adjacency matrix A by assigning a

[1] Pay attention to the fact even though $agent_i$ only knows $J_i(x)$, it estimates x^{opt} at each iteration of the algorithm, which is the optimum for $J(x)$

numerical weight to each edge in the graph. This allows for a more detailed representation of the relationships between vertices, as the weights provide additional information beyond just the presence or absence of an edge (for example, the strength of the connection between two vertices).

This theoretical digression was necessary to introduce the main "actors" of consensus alghorithms, namely $\mathcal{G}$ and W. For the distributed gradient descent to converge to the optimum, we need $\mathcal{G}$ and W to satisfy a few properties:

- the graph $\mathcal{G}$ must be undirected ($W[i,j] = W[j,i]$), namely if edge $i \to j$ exists then also edge $j \to i$ exists
- the graph $\mathcal{G}$ must be strongly connected, namely there exists a directed path from any node to any other node
- the weighted adjacency matrix W must be doubly stochastic, namely the summation along the rows and the columns must yelds 1 (from definition of double stochasticity, W is also non-negative)

The condition on W can be easily met by using the Metropolis-Hastings algorithm to compute the weights. Is equally simple to create an undirected binomial graph since, by default, `binomial_graph()` provides and undirected graph. We have to pay more attention to last request: $\mathcal{G}$ strongly connected. Indeed, even when W is doubly stochastic, a binomial graph is not necessarily a strongly-connected one. For this reason, we added a connectivity-test on $\mathcal{G}$. The test is based on the following theorem:

Thm. 4.3 (Francesco Bullo, Lessons on Network Systems, 2019) [pag. 48]: *"Let G be a weighted digraph with N≥2 nodes and with weighted adjacency matrix W, then W is irreducible if and only if G is strongly connected"*

In case $\mathcal{G}$ has a self-loop at each node, and our $\mathcal{G}$ has, the theorem simplifies and we have that:

"Let G be a weighted digraph with N≥2 nodes, a self loop at each node and with weighted adjacency matrix W, then W is primitive if and only if G is strongly connected"

Thus, we check if W is primitive, namely $W^{N-1} > 0$ (all entries are positive), and if not we just build a new graph and repeat the test until W is primitive[2].

[2]We actually carry out the test on A and not on W. However, there is no contradiction in doing so, as A is a particular choice of W and the connectivity test doesn't require the matrix to be double stochastic.

Once we have done that, there is one last property which needs to be satisfied in order to ensure convergence: decreasing stepsize. To be rigorous, the stepsize α (α^k is the stepsize at iteration k) must meet the following conditions:

- $\alpha^k \geq 0$
- $\sum_{k=0}^{\infty} \alpha^k = \infty$
- $\sum_{k=0}^{\infty} (\alpha^k)^2 < \infty$

To update α in such a way that all the previous properties are met, we use a simple formula:

$$\alpha^k = \frac{\alpha^0}{(1+k)} \tag{1.2}$$

...where α^0 is a user defined initialization value.

1.1.2 Algorithm

For the generic i_{th} agent, the distributed algorithm reads:

Initialization:
$x_i^0 =$ `np.random.rand(5, N)`
$\alpha^0 = 1e-3$

For k from 0 to MAXITERS:
$\alpha^k = \frac{\alpha^0}{(1+k)}$

$v_i^{k+1} = \sum_{j=1}^{N} W[i,j] \cdot x_j^k$

$x_i^{k+1} = v_i^{k+1} - \alpha^k \cdot \nabla J_i(v_i^{k+1})$

1.1.3 Observations

Putting aside all the mathematics involved, let's discuss the intuition here. Each agent computes a weighted mean of its neighbours' states, which then becomes its current estimation of the optimum, v_i^{k+1}. Then, it updates such estimation with a gradient descent performed on its own $J_i(\cdot)$, which yelds x_i^{k+1}.

The algorithm works correctly and all the agents converge to the same estimation of x^{opt}, as we'll see. Therefore, we can say that the distributed optimization problem is solved. However, can we say that this is the best solution

possible? Absolutely not. There is still room for improvement in the computation of the descent direction, which in this case is $d_i^k = -\nabla J_i(v_i^{k+1})$. This "local" descent direction neither coincides nor tends to the descent direction of the complete cost function, $-\nabla J(v_i^{k+1})$. The vector $d_i^k = -\nabla J_i(v_i^{k+1})$ is a direction which serves its purpose in the algorithm, but nothing more. We would like $d_i^k \to -\nabla J(v_i^{k+1})$ as $k \to \infty$, $\forall i$ (thus, for each agent). This topic will be expanded in the next chapter, let's now see the results of this "rougher" implementation of the distributed gradient descent algorithm.

1.1.4 Results with binomial graph

Being $J(x)$ a sum of quadratic functions, we can actually compute J^{opt} and x^{opt} by means of very simple formulas:

$$x^{opt} = -(\sum_{i=1}^{N} Q_i)^{-1} \cdot \sum_{i=1}^{N} r_i \tag{1.3}$$

$$J^{opt} = \sum_{i=1}^{N} \frac{1}{2} \cdot (x^{opt^\top} \cdot Q_i \cdot x^{opt}) + (r_i^\top \cdot x^{opt}) \tag{1.4}$$

In Figure 1.1 we appreciate what a binomial graph looks like.

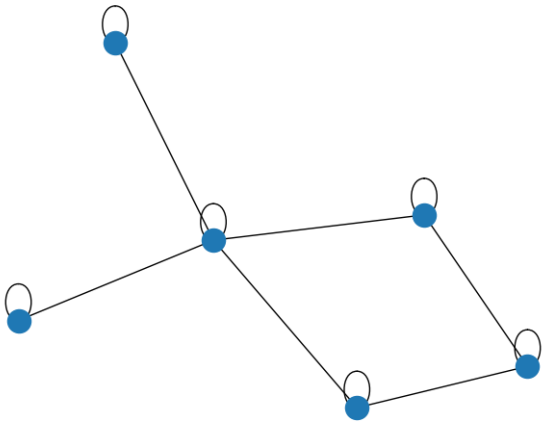


Figure 1.1: Example of binomial graph with self-loops

In Figure 1.2 we plot the difference between the cost at iteration k, $J^k = \sum_{i=1}^{N} J_i(x_i^k)$, and J^{opt}. After 2 million iterations, the difference becomes of order 10^{-3}, indicating that the algorithm works as intended.

To visualize the consensus we adopted the following methodology: at each iteration k, for each agent i, compute $\sum_{j=1}^{N} \left| \|x_i^k\| - \|x_j^k\| \right|$. This sum quantifies how distant is the estimation of an agent from the estimations of all the other agents. The behaviour of $\sum_{j=1}^{N} \left| \|x_i^k\| - \|x_j^k\| \right|$, $\forall i$, is plotted in Figure 1.3. After 2 million iterations we can say that consensus is reached since the sums become of order 10^{-5}.

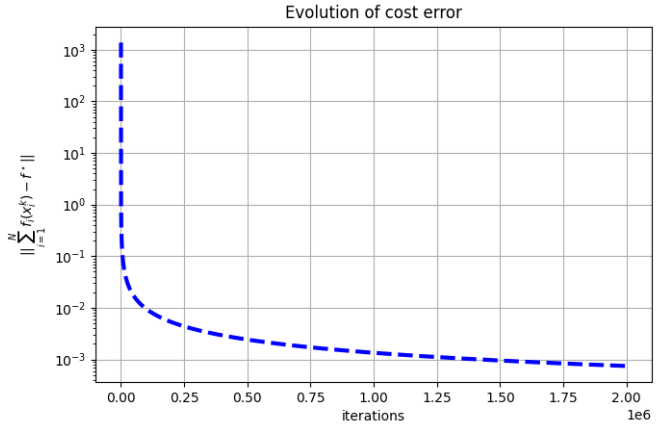


Figure 1.2: Evolution of the cost error. The error is computed with respect to the minimum cost J^{opt}

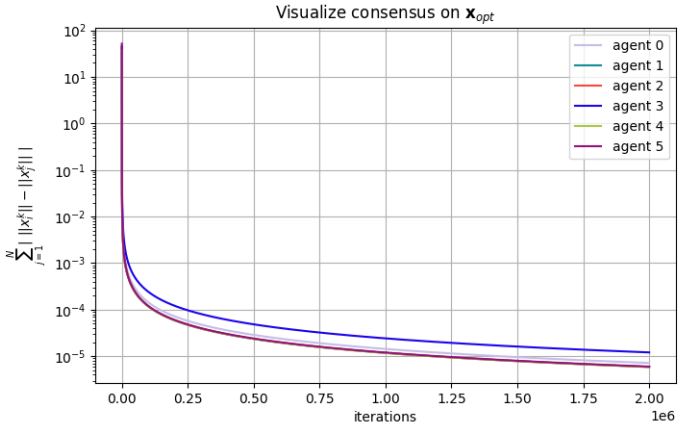


Figure 1.3: All the agents agree on x^{opt}

1.1.5 Results with cycle graph

In Figure 1.6 we can appreciate how the consensus is slightly weaker than the binomial case. As we will see, in the star-graph case the consensus is even weaker. A possible interpretation is that a randomic communication pattern allows for the most efficient spreading of information.

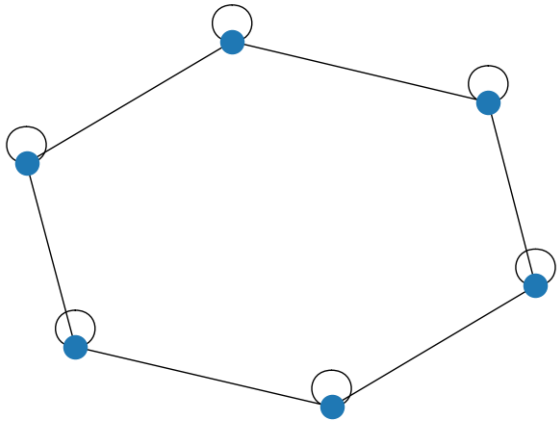


Figure 1.4: Example of cycle graph with self-loops

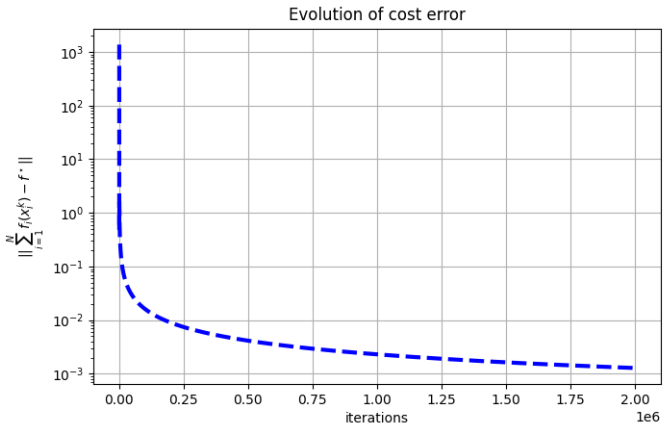


Figure 1.5: Evolution of the cost error. The error is computed with respect to the minimum cost J^{opt}

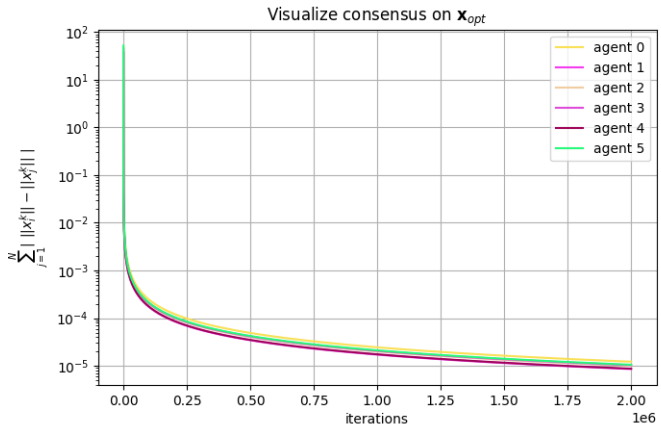


Figure 1.6: All the agents agree on x^{opt}

1.1.6 Results with star graph

The only thing worth noticing is that, as we can appreciate in Figure 1.9, this case is the one with the "weakest" consensus. A possible cause may be found in the fact that, except for the central agent which communicates with every node in the graph, all the other agents have just one neighbour: the central agent itself. Thus, the central agent acts as a bottleneck for the spreading of information, which is less efficient.

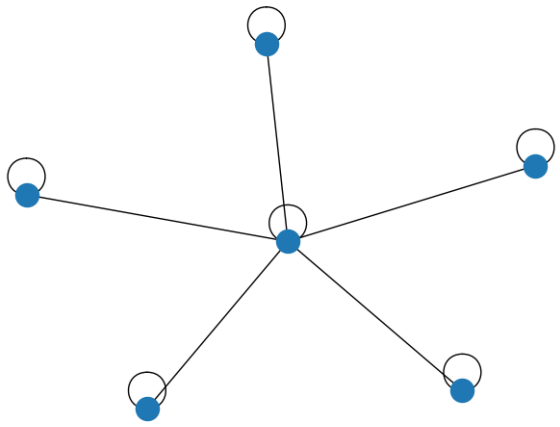


Figure 1.7: Example of star graph with self-loops

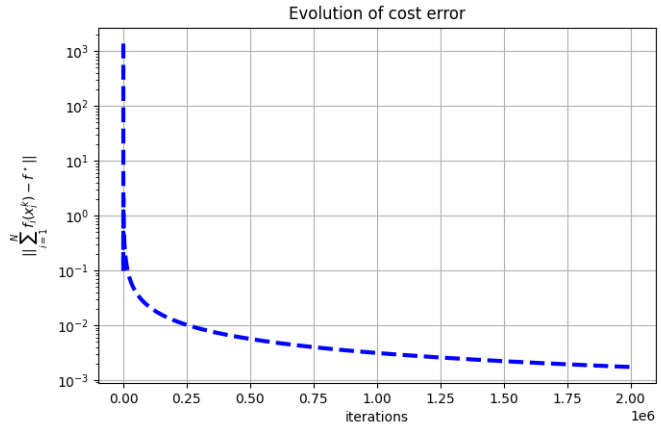


Figure 1.8: Evolution of the cost error. The error is computed with respect to the minimum cost J^{opt}

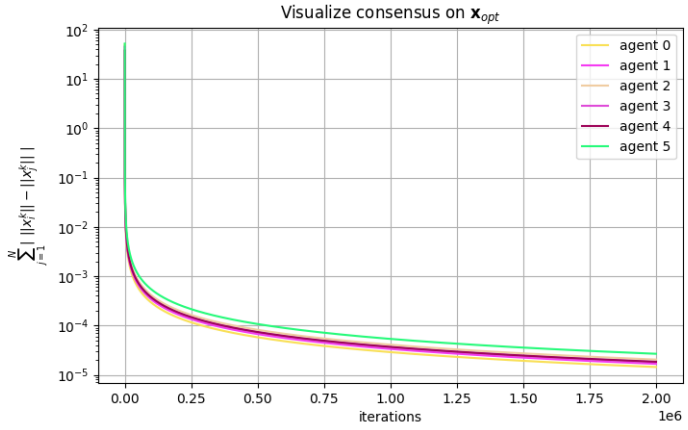


Figure 1.9: All the agents agree on x^{opt}

1.2 Distributed optimization of a quadratic function with gradient tracking

As stated in the previous chapter, we want to improve the distributed gradient descent algorithm by adding an additional feature: "gradient tracking". In order to do so, it is necessary to dive again into the theory.

1.2.1 Theoretical background

The signal $-\nabla J(x)$ changes over time, and we would like the agents to both track it and reach a consensus on its estimation. In the course, we have specifically discussed a subset of consensus algorithms known as averaging algorithms. These algorithms enable the agents to reach a consensus on the mean of their initial states. Here, the averaging algorithms are of no help since we deal with a completely different problem:

- the signal we want to track is not the mean of the initial states
- the signal we want to track changes over time

Let's consider the following framework, which will be kept generic and later adapted to our specific problem:

- at each iteration k of the algorithm, each agent measures a local time-varying signal $r_i^k : \mathbb{N} \to \mathbb{R}^n, \forall i$
- at each iteration k of the algorithm, each agent want to estimate the average signal $\bar{r}^k = \frac{1}{N} \cdot \sum_{i=1}^{N} r_i^k$

A distributed algorithm based on dynamic average consensus reads:

Initialization:
$s_i^0 = r_i^0$

For k from 0 to MAXITERS:
$s_i^{k+1} = \sum_{j=1}^{N} W[i,j] \cdot s_j^k + (r_i^{k+1} - r_i^k)$

In the pseudocode, s_i^k represents the estimation of $\bar{r}^k$ performed by the i_th agent at iteration k. It is possible to show that:

$$\begin{aligned} &\text{if } \|r_i^{k+1} - r_i^k\| \leq C_1 \text{ for some } C_1 > 0, \forall i, \forall k \\ &\text{then } \lim_{k \to \infty} \|s_i^k - \bar{r}^k\| \leq C_2 \text{ for some } C_2, \forall i \end{aligned}$$

Notice that a perfect tracking is not achieved, there is a residual error we can't get rid of. However, with the additional condition $\|r_i^{k+1} - r_i^k\| \to 0$, the steady state error becomes zero and a perfect tracking is achieved.

We can clearly use the previous result to track with zero steady-state error the signal $\bar{r}^k = \frac{1}{N} \sum_{i=1}^{N} \nabla J_i(x_i^k)$. Indeed, as the optimization process gets closer to the optimum, we have that $\|\nabla J_i(x_i^{k+1}) - \nabla J_i(x_i^k)\| \to 0, \forall i$ and the theoretical result cited above guarantees a tracking with no residual error.

To conlude, let's point out some aspects which may have passed unnoticed.

$\|\nabla J_i(x_i^{k+1}) - \nabla J_i(x_i^k)\| \to 0$ only if $x_i^k \to x_{opt}$, where x_{opt} is the optimum for $J(x)$. Thus, the convergence of $\nabla J_i(x_i^k)$ to a constant vector is dependent on the convergence of x_i^k to x_{opt}

In the limit of $k \to \infty$, supposing that $x_i^k \to x_{opt}$, $\nabla J_i(x_i^k)$ does not actually converge to the zero vector. Indeed, the optimum for $J(x)$ is not the optimum for $J_i(x)$. $\nabla J_i(x_i^k)$ converges to a constant vector, whose entries depend on the optimum for $J(x)$: $\nabla J_i(x_i^k) \to \nabla J_i(x_{opt})$

In the limit of $k \to \infty$, supposing that $x_i^k \to x_{opt}$, and as a consequence $\nabla J_i(x_i^k) \to \nabla J_i(x_{opt})$, each agent measures $\frac{1}{N}\sum_{i=1}^{N} \nabla J_i(x_{opt}) = \frac{1}{N}\nabla J(x_{opt})$. Thus, each agent measures a vector which has the same direction as the gradient of $J(x)$, yet a different modulus. Is this a problem? No, in gradient descent algorithms we place more importance on the direction of descent rather than the modulus of the gradient. The modulus can be adjusted using the stepsize.

However, since this technique is called "gradient tracking", it was worth highlighting the fact that what each agent tracks is not actually the real gradient of $J(x)$.

From what said before we can deduce that the whole "gradient tracking" technique is hinged on the fact that we are able to steer x_i^k towards x_{opt}. It can actually be shown that if each agent runs an s-dynamics and uses s_i^k as descent direction, then x_i^k converges to x_{opt}.

We can now appreciate the big picture:

- if each agent runs an s-dynamics and uses s_i^k as descent direction, then x_i^k converges to x_{opt}, $\forall i$
- thanks to convergence of x_i^k $\forall i$, we can claim that $\|\nabla J_i(x_i^{k+1}) - \nabla J_i(x_i^k)\| \to 0$ $\forall i$, thus we can claim the $s_i^k \to \bar{r}^k$ $\forall i$
- again, thanks to convergence of x_i^k $\forall i$, we have that $\bar{r}^k \to \frac{1}{N}\nabla J(x_{opt})$
- chaining the previous results, we deduce that $s_i^k \to \frac{1}{N}\nabla J(x_{opt})$, $\forall i$

In the limit of $k \to \infty$, the descent direction we use in the gradient descent algorithm estimates the gradient direction: we have realized the "gradient tracking" technique.

1.2.2 Algorithm

At each iteration of the distributed gradient descent algorithm, each agent should perform two estimations:

- First, a new estimation of x_{opt}
- Second, a new estimation of the descent direction

Initialization:
$x_i^0 = \text{np.random.rand(5, N)}$
$s_i^0 = \nabla J_i(x_i^0)$

$\alpha^0 = 1e-3$

For k from 0 to MAXITERS:
$\alpha^k = \frac{\alpha^0}{(1+k)}$
$x_i^{k+1} = \sum_{j=1}^{N} W[i,j] \cdot x_j^k - \alpha^k \cdot s_i^k$

$s_i^{k+1} = \sum_{j=1}^{N} W[i,j] \cdot s_j^k + (\nabla J_i(x_i^{k+1}) - \nabla J_i(x_i^k))$

We usually refer to the term "$\nabla J_i(x_i^{k+1}) - \nabla J_i(x_i^k)$" as the "innovation term".

1.2.3 Observations

In order to avoid excessively burdening the theoretical exposition, we have decided to overlook certain assumptions that are necessary for the convergence of the algorithm. The framework is the same as in the previous chapter, thus many of the assumptions will be repeated. However, for the sake of completeness, we will report all of them:

- $\mathcal{G}$ must be undirected and strongly connected
- the weighted adjacency matrix W must be doubly stochastic
- $J(x)$ must be strongly convex with coefficient $\mu > 0$ (this condition implies a unique optimal solution)
- $J(x)$ must have Lipschitz continuous gradient with constant $L > 0$
- $J(x)$ must have bounded gradient:
  $\exists C_1 \in \mathbb{R},\ C_1 > 0 : \|\nabla J(x)\| \leq C_1 \quad \forall x \in \mathbb{R}^n$
- $J(x)$ must admit at least one optimal solution

- decreasing stepsize

By...

- adopting a binomial/star/cycle graph, as the ones used in the previous chapter (every node has a self-loop)
- computing W via the Metropolis-Hastings algorithm
- adopting a quadratic cost function with shape $J(x) = \sum_{i=1}^{N} J_i(x) = \sum_{i=1}^{N} \frac{1}{2} \cdot (x^\top \cdot Q_i \cdot x) + (r_i^\top \cdot x)$
- adopting an α-schedule as: $\alpha^k = \frac{\alpha^0}{(1+k)}$

...we satisfy all the previous requirements and we ensure that the algorithm works as intended.

1.2.4 Results with binomial graph

In Figure 1.12 we can appreciate how each agent estimates a gradient whose module goes to zero. The tracking is fast and precise.

Last but not least, from Figure 1.11, let's notice how the algorithm with gradient tracking achieves a stronger consensus with respect to the algorithm presented in the previous chapter.

Results with star graph and cycle graph are omitted are excluded as they do not contribute significantly to the discussion (consensus is weaker, as before).

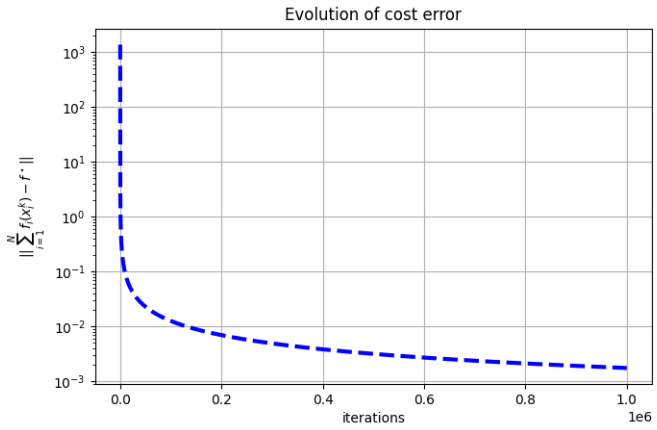


Figure 1.10: Evolution of the cost error. The error is computed with respect to the minimum cost J^{opt}

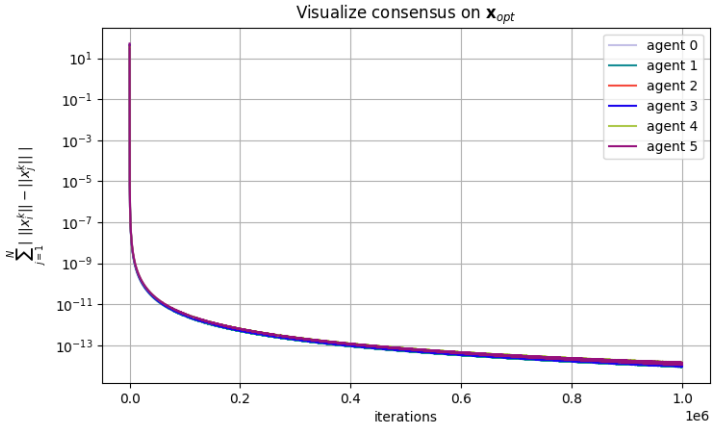


Figure 1.11: All the agents agree on x^{opt}

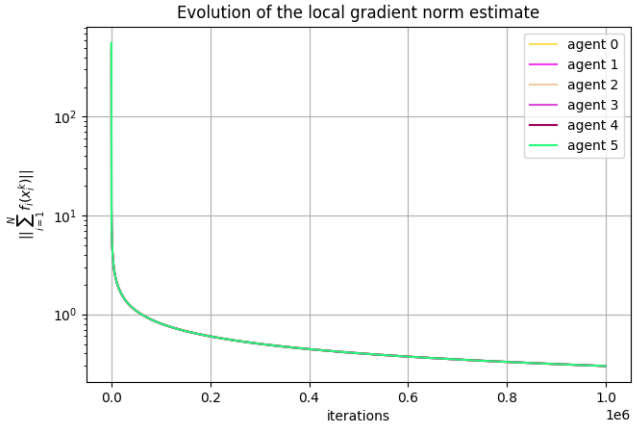


Figure 1.12: Estimation of gradient norm

Chapter 2

Centralized training of a Neural Network

In this chapter, our main focus is on implementing by hand a simple neural network for image classification. Specifically, we address a binary classification problem. The objective of the neural network is to determine whether the input image belongs or does not belong to a specific class (which we select as the class of interest). To train and test, we have been provided with a popular dataset of grayscale hand-written digits, called "MNIST-digits". The images are organized into 10 classes as the digits range from 0 to 9.

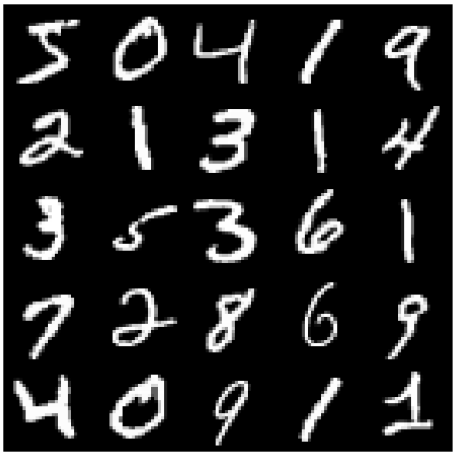


Figure 2.1: Example of training images

Each image is represented as a 28×28 matrix of grayscale values ranging from 0 to 255.

2.1 Choice of the architecture: MLP vs CNN

When dealing with images, the best choice is always to use a CNN (Convolutional Neural Network). This family of neural networks has been precisely developped to extract features from images, and has shown to work really well in a variety of different applications. However, CNNs are way more complex to implement by hand than a traditional MLP (Multi Layer Perceptron). Moreover, considering the small size of the images (28×28), the simplicity of the images (grayscale with limited information), and the straightforward nature of the classification problem (binary classification), an MLP is fully capable of addressing the task. Just remember to flatten the images before feeding them as input.

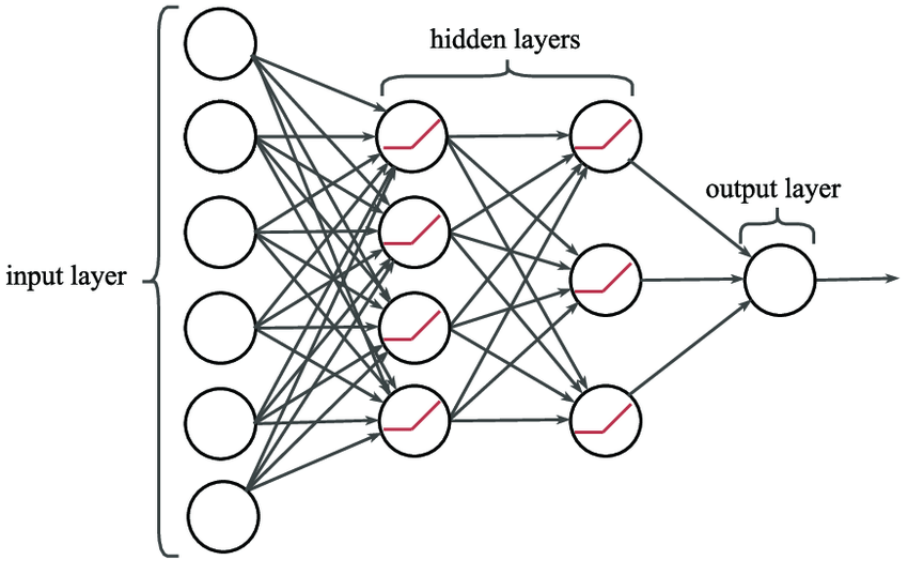


Figure 2.2: Example of MLP

The first MLP we tested had the following architecture: input layer of size 784, sigmoid activation, hidden layer of size 128, sigmoid activation, output layer of size 1. After several tests, we moved to a simpler architecture that demonstrated nearly equivalent performance but with improved speed: input layer of size 784, softmax activation, output layer of size 2.

2.2 Choice of the activation function for the output layer: sigmoid vs softmax

The output of the neural network is a discrete probability distribution over two possible outcomes: "the input image belongs to the class of interest", "the input image does not belong to the class of interest".

In case we use the sigmoid activation, the neural network must have an output layer of size 1. The value of the output neuron represents the probability p that the input image belongs to class of interest. Since probabilities add up to 1, the probability that the image does not belong to the class of interest can be simply computed as $1 - p$. It would be an error to use the sigmoid activation in combination with an output layer of size 2, as there is no guarantee that the two probabilities would add up to 1.

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

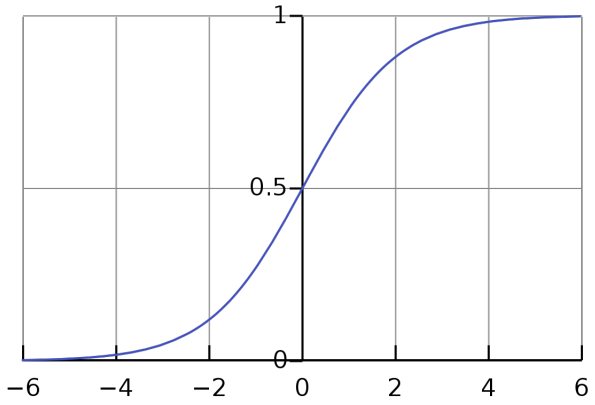


Figure 2.3: sigmoid function

In case we use the softmax activation, the neural network must have an output layer of size 2. Let's visualize the function:

$$\text{softmax}(x) = \left[\frac{e^{x_1}}{\sum_{i=1}^{n} e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^{n} e^{x_i}}, \ldots, \frac{e^{x_n}}{\sum_{i=1}^{n} e^{x_i}} \right] \tag{2.2}$$

Given as input a vector of length$N$, the softmax activation return a discrete probability distribution of length N. In our binary classification problem, we feed a vector of length 2 and we get as output a discrete probability distribution of length 2. Even if the introduction of the softmax does not

sensibly improve the solution of the binary classification problem, it allows us to cast such problem into a multi-class classification problem with almost no effort: just change the size of the output layer from 2 to 10.

2.3 The loss: Categorical Cross Entropy

Categorical Cross Entropy is a commonly used loss function in deep learning, particularly for multi-class classification problems. It measures the dissimilarity (or "distance") between the predicted probability distribution over the classes ($\hat{\mathbf{y}}$, the one outputted by the neural network), and the true probability distribution over the classes ($\mathbf{y}$, the one provided by the MNIST-digits dataset).

$$\text{CCE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{N} y_i \log(\hat{y}_i) \tag{2.3}$$

The true probability distribution $\mathbf{y}$ is a vector of length N, where N represents the number of classes. All its entries are set to 0 except for one entry, which is set to 1. This non-zero entry corresponds to the class to which the image belongs.

Categorical Cross Entropy boils down to a simpler formula, Binary Cross Entropy, in case of a binary classification problem.

$$\text{BCE}(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \tag{2.4}$$

When training a neural network, the loss function we use in the gradient descent technique is a summation of CCEs computed over a batch of images. Let's use "$\phi(\mathbf{I}_i, \mathbf{u})$" to identify the output of the neural network, where $\mathbf{I}_i$ is the input image and $\mathbf{u}$ is the collection of weights and biases. Let's also use "b_{size}" to identify the batch size ($b_{size} = 128$, for example). The loss function J has the following shape:

$$J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{b_{size}}; \mathbf{u}) = \sum_{j=1}^{b_{size}} \text{CCE}(\mathbf{y}_j, \phi(\mathbf{I}_j; \mathbf{u})) \tag{2.5}$$

At the end of each batch, we compute the gradient of J with respect to $\mathbf{u}$ and we use it to perform gradient descent:

$$\mathbf{u}^{k+1} = \mathbf{u}^k - \alpha^k \cdot \nabla_u J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{b_{size}}; \mathbf{u}^k) \tag{2.6}$$

At the end of each epoch we shuffle the whole dataset and we re-build the batches, in this way we avoid patterns while training.

Let's point out that J, being the sum of logarithms, is convex and thus admits a unique optimal solution.

2.3.1 Observations

In theory, the cost function we should minimize is a summation of CCEs performed over the whole set of images used for training (12800 in our case).

$$J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{12800}; \mathbf{u}) = \sum_{j=1}^{12800} \mathrm{CCE}(\mathbf{y}_j, \phi(\mathbf{I}_j; \mathbf{u})) \tag{2.7}$$

However, computing the gradient and utilizing the gradient descent technique would introduce a significant computational cost. Additionally, the gradient obtained would have an exceptionally large magnitude, therfore introducing issues in the gradient descent technique (requiring the selection of an extremely small step size).

Due to these factors, it is customary to use the so called "mini-batch gradient descent" technique. The intuition is that if we update weights and biases based on the cumulative error computed over a batch of images, the new weights and biases would make the neural network perform better on all the subsequent batches. This idea holds merit, as a batch of proper dimension can be supposed to "well represent" the overall data distribution. Some take this intuition to the extreme and perform gradient descent after each image. While this approach may sometimes provide good results, it is usually not recommended as it may cause the gradient to "zig-zag around". Therefore, selecting an appropriate batch size is crucial for the success of the gradient descent process. As a general rule of thumb, batches with a size of 2^n where $5 \leq n \leq 8$ tend to work well.

Last but not least, let's point out that even if J depends on the images we use to compute it, such images are not considered "variables". In the training of a neural network, we refer with "variables" to all the learnable parameters, thus to all the parameters on which we can perform gradient descent. Obviously, the images are not learnable parameters. Given that, the fact that every batch is a different collection of images does not affect the minimization process, which involves only weights and biases of the network.

2.4 Train 'n Test

As anticipated, the first neural network we implemented had a single hidden layer, sigmoid activation everywhere and single output neuron. With this set-up, the training process consumed a significant amount of time, leading us to realize that adding another layer was not feasible. Despite the long wait, the training resulted in a success, with the neural network achieving

an accuracy[1] of nearly 97%. Encouraged by this positive outcome, we decided to explore a simpler architecture by removing the hidden layer. As a result, the training process improved significantly in terms of speed while maintaining the same level of accuracy (Figure 2.4).

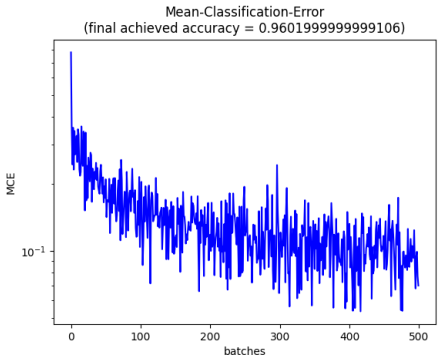


Figure 2.4: Behaviour of the MCE during the training process. MCE is the average BCE over a batch

Before transitioning to the multi-class classification problem, we experimented with the binary problem using softmax activation. The purpose was to gain a better understanding of the softmax function and its functionality. We maintained the same architecture with zero hidden layers, but increased the output dimension from 1 to 2 (as previously suggested). The training process achieved again an accuracy around 97% (Figure 2.5).

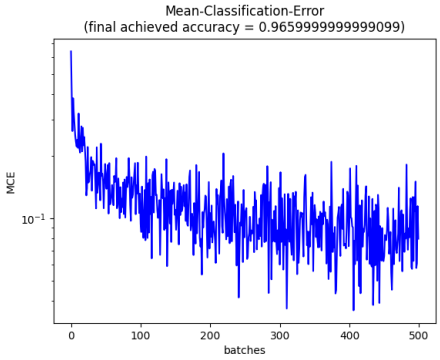


Figure 2.5: Behaviour of the MCE during the training process. MCE is the average BCE over a batch

[1]The accuracy has been computed as percentage of correctly classified images in the test set

Next, we adjusted the output dimension to 10 in order to address the multi-class classification problem. The training process took slightly longer and achieved a lower accuracy of approximately 90% (Figure 2.6). This outcome was expected as the multi-class classification problem is more complex. Although we could have attempted to improve the accuracy by adding a hidden layer, we decided that it wasn't worth it, as a 90% accuracy is already sufficient to affirm the effectiveness of the gradient descent algorithm.

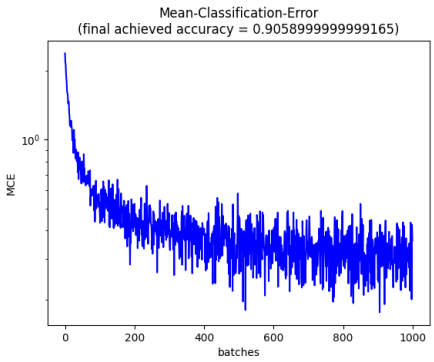


Figure 2.6: Behaviour of the MCE during the training process. MCE is the average BCE over a batch

2.5 Implementation of the Centralized Learning

Without loss of generality, we will focus on:

- the binary classification problem
- an architecture with T layers ($T-2$ hidden layers, in our case $T-2=0$) and softmax activation at last layer.

There are different ways to interpret the training of a neural network. However, considering our background in Optimal Controls, it was natural for us to frame it as a constrained minimization problem. Consider the following framework:

- Each layer t has d_t neurons.
- The activation function at layer t is $\sigma_t(\cdot)$. Since we won't use any hidden layer, $\sigma_t(\cdot)$ is kept generic, except for $\sigma_{T-1}(\cdot)$, which is the softmax

- All the weights and biases associated to layer t are stored in the matrix $u_t \in \mathbb{R}^{(d_{t+1})\times(d_t+1)}$
- $\mathbf{u}$ is the collection of the $\{u_t\}_{t=1,2,..T-1}$: $\mathbf{u} = [u_1, u_2, ..., u_{T-1}]$
- To update all neurons, in a single layer, in a single shot, we run the "inference dynamics": $x_{t+1} = \sigma_t(x_t, u_t) = \sigma_t(u_t[:, 1:] \cdot x_t + u_t[:, 0])$. x_t is a column vector.
- $x_T = \sigma_{T-1}(x_{T-1}, u_{T-1}) = \phi(\mathbf{I}, \mathbf{u})$ is the output of the neural network, it is the predicted probability distribution given the input image I.

Given all the previous notation, the minimization process over a batch reads:

$$\begin{aligned} &\text{Minimize over } \mathbf{u}: && J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_a; \mathbf{u}) = \sum_{j=1}^{a} \text{CCE}(\mathbf{y}_j, \phi(\mathbf{I}_j; \mathbf{u})) \\ &\text{subj. to:} && x_T = \sigma_{T-1}(x_{T-1}, u_{T-1}) = \phi(\mathbf{I}; \mathbf{u}), \\ & && x_{t+1} = \sigma_t(x_t, u_t), \quad i = 1, 2, \dots, T-2 \end{aligned}$$

The parallelism with what we have seen in Optimal Controls is evident. However, it is important to note that there is a minor detail to consider: for each image $\mathbf{I}$ in the batch, the classification error CCE$(\cdot)$ depends solely on the output of the neural network, namely x_T: CCE$(\mathbf{y}, x_T)$. It should be emphasized that the true probability distribution, $\mathbf{y}$, is not a collection of learnable parameters, but rather a given value.

Thus, the equivalent Optimal Controls problem would read:

$$\begin{aligned} &\text{Minimize over } \mathbf{u}: && l_T(x_T) = l_T(\phi_T(\mathbf{u})) \\ &\text{subj. to:} && x_{t+1} = \sigma_t(x_t, u_t), \quad i = 1, 2, \dots, T-1 \end{aligned}$$

We can recognize that there are no stage costs, just the terminal cost. This simplifies the solution process we have seen for this kind of problems. The solution process to this Optimal Controls problem reads:

For all k = 0,1,2 ... do:

Initialization:
$\lambda_T = \nabla_{x_T} l(x_T^k)$

For t from T-1 to 0:
$\lambda_t = \nabla_{x_t} \sigma_t(x_t^k, u_t^k) \cdot \lambda_{t+1}$

$\nabla_{u_t} J(\mathbf{u}) = \nabla_{u_t} \sigma_t(x_t^k, u_t^k) \cdot \lambda_{t+1}$

#Gradient descent on u#
For t from 0 to T-1:
$u_t^{k+1} = u_t^k - \alpha^k \cdot \nabla_{u_t} J(\mathbf{u}^k)$

#Forward simulation of the dynamics#
Initialization:
$x_0^{k+1} = x_{init}$

For t from 0 to T-1:
$x_{t+1}^{k+1} = \sigma_t(x_t^{k+1}, u_t^{k+1})$

When training a neural network, an "iteration" is a complete pass over a batch of images and is marked at the end by a gradient descent on **u**. The previous pseudocode, adapted to the neural network case, becomes[2]:

[2]we are at the generic iteration k of the algorithm, we suppress the apices "^{k}" to simplify the notation

For each image $\mathbf{I_i}$ in a batch do:

#Inference dynamics to compute $x_{T,i}$#
Initialization:
$\nabla_u J(\mathbf{u}) = 0$
$\mathbf{x}_{0,i} = \mathbf{I_i}$.flatten()

For t from 0 to T-1:
$x_{t+1,i} = \sigma_t(x_{t,i}, u_t)$

#Backward pass to compute $\nabla_u J_i(\mathbf{y}_i, \phi(\mathbf{I_i}; \mathbf{u})) = \nabla_u CCE(\mathbf{y}_i, \phi(\mathbf{I_i}, \mathbf{u}))$#
Initialization:
$\lambda_T = \nabla_{x_T} CCE(\mathbf{y_i}, x_{T,i})$

For t from T-1 to 0:
$\lambda_t = \nabla_{x_t} \sigma_t(x_{t,i}, u_t) \cdot \lambda_{t+1}$
$\nabla_{u_t} CCE(\mathbf{y}_i, \phi(\mathbf{I_i}; \mathbf{u})) = \nabla_{u_t} \sigma_t(x_{t,i}, u_t) \cdot \lambda_{t+1}$

#Update the complete gradient#
For t from T-1 to 0:
$\nabla_{u_t} J(\mathbf{u}) = \nabla_{u_t} J(\mathbf{u}) + \nabla_{u_t} CCE(\mathbf{y}_i, \phi(\mathbf{I_i}; \mathbf{u}))$

At the end of the batch, perform gradient descent on $\mathbf{u}$. To do that, $\nabla_{u_t} J(\mathbf{u})$ should be re-shaped into a matrix. Suppose it has been done, we have:

For t from 0 to T-1:
$u_t^{k+1} = u_t^k - \alpha \cdot \nabla_{u_t} J(\mathbf{u}^k)$

Repeat the process for all the batches in an epoch. Then shuffle the dataset, re-build the batches and start a new epoch.

Chapter 3

Distributed training of a Neural Network

In this chapter, we employ all the concepts and information previously discussed to realize the distributed learning of a neural network with gradient tracking.

3.1 First attempt: 100 agents

As shown in Chapter 2, the neural network's loss function computed over the whole MNIST-digits dataset looks like this:

$$J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{12800}; \mathbf{u}) = \sum_{j=1}^{12800} \text{CCE}(\mathbf{y}_j, \phi(\mathbf{I}_j; \mathbf{u}))$$

We also said that, tipically, we replace it with its mini-batch version:

$$J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{b_{size}}; \mathbf{u}) = \sum_{j=1}^{b_{size}} \text{CCE}(\mathbf{y}_j, \phi(\mathbf{I}_j; \mathbf{u}))$$

Thus, the first idea that comes to mind is to create as many agents as many batches there are. For the centralized training of the neural network we used 12800 images splitted into 100 batches of size 128. Moving-on to distributed learning we didn't change that structure, therefore our first attempt at solving the distributed problem involved 100 agents.

If there are 100 agents, each one of them is given its own "immutable" set of 128 images, "immutable" in the sense that each agent always handles the same 128 images across the epochs. With this choice, the complete loss function J is split into 100 loss functions, $\{J_i(\mathbf{u}_i)\}_{i=0,1,2,\dots,100}$:

i_{th} agent $\rightarrow J_i(\mathbf{u}_i) = J(\mathbf{I}_1^i, \mathbf{I}_2^i, \dots \mathbf{I}_{128}^i; \mathbf{u}_i) = \sum_{j=1}^{128} \text{CCE}(\mathbf{y}_j^i, \phi(\mathbf{I}_j^i; \mathbf{u}_i))$

Where $[\mathbf{I}_j^i, \mathbf{y}_j^i]$ is a generic pair `[image, ground truth]` which belongs to the i_{th} agent, and $\mathbf{u}_i$ is the collection of weights and biases which belongs to the i_{th} agent.

Let's recap the structure of the problem:

- we want to find u_{opt} for $J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{12800}; \mathbf{u}) = \sum_{i=1}^{100} J(\mathbf{I}_1^i, \mathbf{I}_2^i, \dots \mathbf{I}_{128}^i; \mathbf{u}) = \sum_{i=1}^{100} J_i(\mathbf{u})$.
- at the same time, we want each agent to track the gradient of the complete cost function, thus we want $s_i \rightarrow \frac{1}{100} \nabla_u J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{12800}; \mathbf{u}_{opt})$
- to do so, we create 100 agents, each agent with its own set of 128 images, $\{\mathbf{I}_j^i\}_{j=1,2,\dots 128}$, its own set of weights and biases, $\mathbf{u}_i$, and its own cost function $J_i(\mathbf{u}_i)$
- the communication among agents is dictated by a weighted graph $\mathcal{G}$, with weighted adjacency matrix W

Recall from the first chapter that, in order to have convergence of s_i and $\mathbf{u}_i$ $\forall i$, we need some assumption to be satisfied:

- $\mathcal{G}$ must be undirected and strongly connected
- the weighted adjacency matrix W must be doubly stochastic
- the complete cost function must be strongly convex with coefficient $\mu > 0$ (this condition implies a unique optimal solution)
- the complete cost function must have bounded gradient: $\exists C_1 > 0$ such that $\|\nabla_u J(\mathbf{u})\| \leq C_1$ at any $\mathbf{u}$
- the complete cost function must have Lipschitz continuous gradient with constant $L > 0$
- the complete cost function must admit at least one optimal solution
- decreasing stepsize

By...

- adopting a binomial graph with a self loop at every node
- computing W via the Metropolis-Hastings algorithm
- adopting $J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{12800}; \mathbf{u})$ as complete cost function
- adopting an α-schedule as: $\alpha^k = \frac{\alpha^0}{(1+k)}$

...we satisy all the previous constraints.

Unfortunately, even if the theoretical results ensures convergence of the algorithm independently from the number of agents, we didn't experience convergence in practice. After several trials, we realized that the high number of agents was causing the algorithm to converge so slowly that we couldn't appreciate it. As a consequece, we decided to change the number of agents: from 100 to 5.

3.2 Second attempt: 5 agents

Such a significant change in the number of agents resulted in modifications to the problem's structure. Instead of an immutable set of images, each agent is now provided with a set of 15 batches, each containing 128 images. At the end of each batch, every agent performs gradient descent according to the explanation in Chapter 2. To prevent pattern learning during training, at the end of each epoch the dataset of each agent is shuffled and the batches reconstructed.

The introduction of batches poses a slight challenge, so let's discuss it briefly. The crucial point is that at the end of each batch, each agent had to reset the innovation term in its s-dynamics. This was unnecessary when there were no batches, as the $J_i(\cdot)$ function was always the same. However, now, each agent has a $J_i(\cdot)$ function that changes from batch to batch, as each batch differs from the previous one.

The incorporation of batches proved to be successful and the results can be observed in the next section.

3.3 Results with batchsize of 128 images

As shown in from Figure 3.1, after a first rapid descent in the cost, the training process gets stuck and there is no further improvement. This behaviour can also be observed in Figure 3.2, where we see that all the agents converge to the same estimation of $\|\nabla J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{12800}; \mathbf{u}_i^k)\|$, and such estimation reaches a plateau after approximately 1000 iterations. However, in Figure 3.3, we can appreciate the fact that consensus has been reached.

Let's start with the fact that consensus on $\mathbf{u}_{opt}$ is achieved. This is guaranteed by the fact that we are performing gradient descent as shown in Section 1.2.2. However, such $\mathbf{u}_{opt}$ is not associated to $\nabla J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{12800}; \mathbf{u}_{opt}) = 0$. Is this a problem for gradient tracking? No, what matters for gradient tracking is that $\|\nabla J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{12800}; \mathbf{u}_i^{k+1}) - \nabla J(\mathbf{I}_1, \mathbf{I}_2, \dots \mathbf{I}_{12800}; \mathbf{u}_i^k)\| \to 0, \forall i$, and this happens as a consequence of the consensus on $\mathbf{u}_{opt}$.

Eventually, it should be noted that the use of mini-batches affects the convergence to the optimum.

Results with different batchsizes and in the multi-class case have been omitted as they do not add any useful insight.

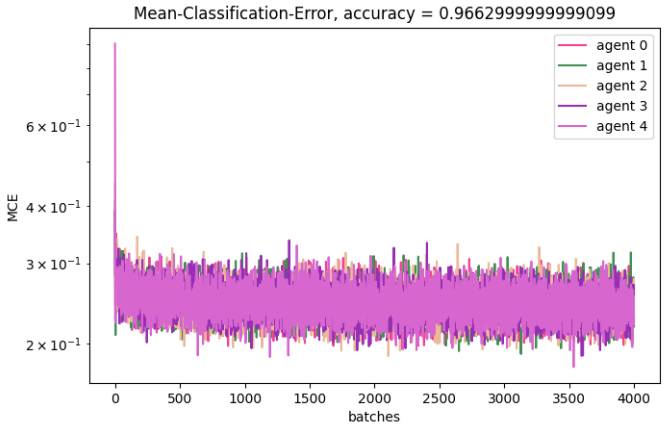


Figure 3.1: Behaviour of the MCE for each one of the 5 agents during the training process. MCE is the average BCE over a batch.

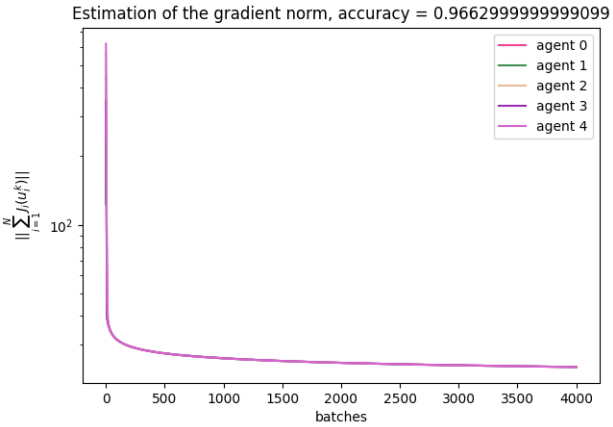


Figure 3.2: Estimation of $\|\nabla J(\mathbf{I}_1, \mathbf{I}_2, \ldots \mathbf{I}_{12800}; \mathbf{u}_i^k)\|$ for each one of the 5 agents during the training process

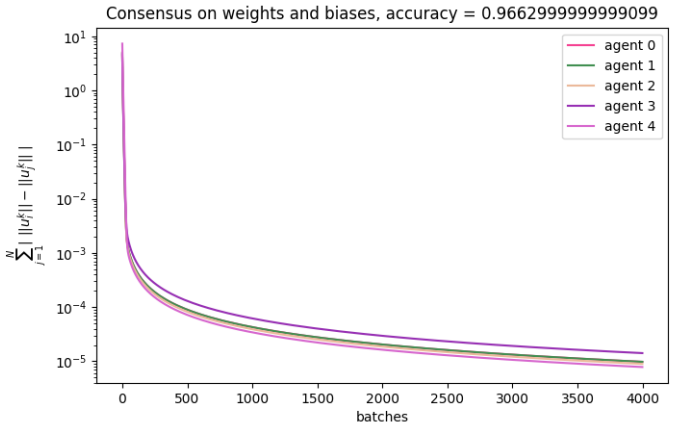


Figure 3.3: All the agents agree on u^{opt}

Task 2

Chapter 4

Formation control

Formation control in a distributed setup refers to the coordination and control of multiple autonomous agents or robots to achieve a desired formation or spatial arrangement. It involves designing control algorithms and communication protocols that enable the agents to work together to maintain a specific geometric pattern or formation while adapting to changes in the environment.

4.1 Required theoretical knowledge

4.1.1 Preliminaries on Laplacian dynamics

Consider a network of dynamical systems with dynamics

$$\dot{\mathbf{x}}_i(t) = u_i(t) \qquad i \in I = \{1, ..., N\}$$

with states $\mathbf{x}_i(t) \in \mathbb{R}$ and inputs $u_i(t) \in \mathbb{R}$, communicating (or interacting) according to a digraph $\mathcal{G} = (I, E)$, where $I = \{1, ..., N\}$ is the set of nodes and $E \subset I \times I$ is a set of ordered pair of nodes called edges.

Consider a distributed 'proportional' feedback control:

$$u_i(t) = -\sum_{j \in N_i^{IN}} a_{ij}(\mathbf{x}_i(t) - \mathbf{x}_j(t)) \qquad \forall i \in \{1, ..., N\}$$

with $N_i^{IN} = \{j \in \{1, ..., N\} \,|(j, i) \in E\}$ being the set of in-neighbors of i and a_{ij} being the non-negative weight, for which $a_{ij} > 0$ if $j \in N_i^{IN}$;
or the out-neighbor version

$$u_i(t) = -\sum_{j \in N_i^{OUT}} a_{ij}(\mathbf{x}_i(t) - \mathbf{x}_j(t)) \qquad \forall i \in \{1, ..., N\}$$

with $N_i^{OUT} = \{j \in \{1, ..., N\} \,|(i, j) \in E\}$ being the set of out-neighbors of i and a_{ij} being the non-negative weight, $a_{ij} > 0$ if $j \in N_i^{OUT}$

Considering the out-neighbors version, let us define weights $a_{ij} = 0$ for $(i,j) \notin E$. We can rewrite the distributed control system as

$$\dot{\mathbf{x}}_i(t) = -\sum_{j=1}^{N} a_{ij}(\mathbf{x}_i(t) - \mathbf{x}_j(t)) \qquad \forall i \in \{1, ..., N\}$$

Defining $\mathbf{x} := [\mathbf{x}_1 \dots \mathbf{x}_N]^T$, it can be shown that it can be rewritten as the following Linear Time Invariant (LTI) continuous-time system

$$\dot{\mathbf{x}}_i(t) = -L\mathbf{x}(t)$$

where $L = (D^{OUT} - A)$ is the weighted Laplacian associated to the digraph $\mathcal{G}$ with weighted adjacency matrix A and D^{OUT} being the weighted out-degree matrix.

Theorem 7.4 LNS
Let L be a (weighted) Laplacian matrix with associated strongly connected (weighted) digraph $\mathcal{G}$.
Consider the Laplacian dynamics $\dot{\mathbf{x}}_i(t) = -L\mathbf{x}(t), t \geq 0$, then

- $$\lim_{t\to\infty} \mathbf{x}(t) = \left(\frac{\omega^T \mathbf{x}(0)}{\omega^T \mathbf{1}}\right)\mathbf{1}$$

  with $\mathbf{1} = [1 \dots \ 1]^T$ and $\omega^T L = 0$, i.e. ω is a left eigenvector for the eigenvalue $\lambda = 0$. This means that consensus is reached.

- if additionally $\mathcal{G}$ is weight-balanced (i.e. $deg_i^{IN} = deg_i^{OUT}$ for all $i \in \{1, ..., N\}$ with deg_i^{IN} and deg_i^{OUT} being the weighted in/out degrees of agent i)

  $$\lim_{t\to\infty} \mathbf{x}(t) = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i(t)\mathbf{1}$$

  i.e. they reach average consensus, which means that the consensus value is the average of the initial states.

These results can be extended to $\mathbf{x}_i(t) \in \mathbb{R}^d$ with $d > 1$ by using the Kronecker product

$$\tilde{L} = L \otimes I \qquad I \in \mathbb{R}^{d\times d}$$

with $\tilde{L}$ being the extended Laplacian.

4.1.2 Introduction of Formation Control

Consider a platoon of N masses such that each mass i is connected with mass $i-1$ and $i+1$ through a spring with elastic constants respectively $a_{i,i-1} > 0$ and $a_{i,i+1} > 0$. Let $\mathbf{x}_i(t) \in \mathbb{R}$ be the position of mass i. As abuse

of notation we will omit the time (t).
The elastic force at mass i, $F_{e,i}(x)$ is given by:

$$F_{e,i}(\mathbf{x}) = -a_{i,i-1}(\mathbf{x}_i - \mathbf{x}_{i-1}) - a_{i,i+1}(\mathbf{x}_i - \mathbf{x}_{i+1})$$

We can generalize it to masses connected according to a graph $\mathcal{G}$ as:

$$F_{e,i}(\mathbf{x}) = -\sum_{j \in N_i} a_{i,j}(\mathbf{x}_i - \mathbf{x}_j)$$

If we denote $F_e = \begin{bmatrix} F_{e,1}(\mathbf{x}) \\ \vdots \\ F_{e,N}(\mathbf{x}) \end{bmatrix}$ the elastic force of all masses, then

$$F_e = -L\mathbf{x}$$

with L being the weighted Laplacian of the associated weighted graph.

Consider the elastic energy

$$V_i(\mathbf{x}) = \sum_{j \in N_i} V_{ij}(\mathbf{x})$$

with $V_{ij}(\mathbf{x}) = \frac{1}{2}a_{i,j}(\mathbf{x}_i - \mathbf{x}_j)^2$

The elastic force is the negative gradient of the energy, i.e.,

$$F_{e,i}(\mathbf{x}) = -\sum_{j \in N_i} \frac{\partial V_{ij}(\mathbf{x})}{\partial \mathbf{x}_i}$$

This idea can be extended to general potential functions and applied to distributed control systems.
Notice that spring-mass systems have a second order dynamics, thus we should write $\ddot{\mathbf{x}} = -L\mathbf{x}$. We will, instead, apply the potential function idea to first order dynamics.

4.1.3 Formation control based on potential functions

Consider a network of N agents communicating/interacting according to a fixed, undirected graph $\mathcal{G} = (\{1, ..., N\}, E)$. Let $\mathbf{x}_i(t) \in \mathbb{R}^d$ be the state of agent i, with $d \geq 1$ being the state dimension.

Let agents run a Laplacian dynamics:

$$\dot{\mathbf{x}}_i(t) = -\sum_{j \in N_i} a_{ij}(\mathbf{x}_i(t) - \mathbf{x}_j(t)) \qquad \forall i \in \{1, ..., N\}$$

with compact form $\dot{\mathbf{x}}_i(t) = -L\mathbf{x}(t)$ and N_i being the set of neighbors of agent i.
Recall that if agents run a Laplacian dynamics, then they reach consensus and, since the graph is undirected, the consensus value is the average of the initial states.

We can rewrite it as

$$\dot{\mathbf{x}}_i(x) = -\sum_{j \in N_i} \frac{\partial V_{ij}(\mathbf{x})}{\partial \mathbf{x}_i} \qquad \forall i \in \{1, ..., N\}$$

with $V_{ij}(\mathbf{x}) = \frac{1}{2} a_{ij} \|\mathbf{x}_i - \mathbf{x}_j\|^2$

We can define a desired formation by assigning a set of distances d_{ij} that agents in the network need to satisfy. The defined distances must be consistent.
The main idea is to define local potential functions $V_{ij}(\mathbf{x})$ that have a minimum when $\|\mathbf{x}_i - \mathbf{x}_j\| = d_{ij}$.

In order to reach a formation with assigned distances d_{ij}, let us define

$$V_{ij}(\mathbf{x}) = \frac{1}{4}\left(\|\mathbf{x}_i - \mathbf{x}_j\|^2 - d_{ij}^2\right)^2$$

which has a global minimum for $\|\mathbf{x}_i - \mathbf{x}_j\| = d_{ij}$ as we wanted.

Thus, the dynamics of each agent i is given by

$$\dot{\mathbf{x}}_i(t) = -\sum_{j \in N_i} \frac{1}{4}\big(\|\mathbf{x}_i - \mathbf{x}_j\|^2 - d_{ij}^2\big)(\mathbf{x}_i(t) - \mathbf{x}_j(t)) \qquad \forall i \in \{1, ..., N\}$$

The above formation control dynamics may have also other equilibria besides the one in which agents are at the assigned distances. In particular, $\mathbf{x}_1 = \mathbf{x}_2 = \cdots = \mathbf{x}_N$ is also an equilibrium. This particular condition is called *rendez-vous* and it can be avoided by means of suitable barrier functions, as explained in the following parts of the report.

4.2 Task 2.1 - Problem Set-up

Consider a network of N robots, with N being dependent on the pattern we want to achieve.
Let $I = \{1, ..., N\}$ be the set of robots.
Let robots communicate according to a fixed, undirected graph $\mathcal{G} = (I, E)$, with the set of edges E being defined differently for each pattern, as we will see below.

We denote the position of robot $i \in I$ at time $t \geq 0$ with $\mathbf{x}_i(t) \in \mathbb{R}^3$, and with $\mathbf{x}_i^k \in \mathbb{R}^3$ its discretized version at time $k \in \mathbb{N}$. Hence, we represent with $\mathbf{x}(t) \in \mathbb{R}^{3N}$ and $\mathbf{x}^k \in \mathbb{R}^{3N}$ respectively the stack vector of the continuous and the discrete positions.
Since $\mathbf{x}_i(t) \in \mathbb{R}^3$ we are representing a 3-dimensional position in the space $\mathbf{x}_i(t) = [x_i(t), y_i(t), z_i(t)]$ with $x_i(t), y_i(t), z_i(t) \in \mathbb{R}$.

In order to perform the formation control of agents, complying with theory, we have implemented the following derivative of the potential function:

$$\frac{\partial V_{form,ij}(\mathbf{x})}{\partial \mathbf{x}_i} = k_{form}\big(\|\mathbf{x}_i(t)-\mathbf{x}_j(t)\|^2-d_{ij}^2\big)(\mathbf{x}_i(t)-\mathbf{x}_j(t)) = F_{form,ij} \qquad \forall i \in I, y \in N_i$$

with k_{form} being the formation control gain, which it will be tuned differently for any pattern we will achieve (as any other gain we will introduce).

The distances d_{ij} are defined differently for each pattern. If $d_{ij} > 0$ it means that agents i and j, with $i, j \in I$, are constrained to maintain a distance of d_{ij}. While, if $d_{ij} = 0$, it means that no distance constraint is imposed between agents i and j. Clearly, d_{ii} will be always equal to 0. Furthermore, distances will be defined to be consistent and d_{ij} will be equal to $d_{ji} \quad \forall (i, j) \in I$
Distances are collected in a matrix

$$D_{i,j} = \begin{bmatrix} d_{1,1} & d_{1,2} & \dots & d_{1,N} \\ d_{2,1} & d_{2,2} & \dots & d_{2,N} \\ \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ d_{N,1} & d_{N,2} & \dots & d_{N,N} \end{bmatrix}$$

which will be used to compute the adjacency matrix A by considering only the positive elements of $D_{i,j}$. This will define the communication network between the agents.
Clearly, if $d_{ij} = 0$, there will be no communication between agents i and j. Furthermore, since $d_{ij} = d_{ji} \forall (i, j) \in I$ the matrix A will be symmetric and the graph $\mathcal{G}$ undirected.

Remark: Let's recall that rigidity theory gives us the conditions on the number of distances to assign. In particular, a planar regular polygon is called minimally rigid if at least $2N - 3$ distances are defined.
The number of fixed distances will influence the speed of convergence ergo the number of iteration to reach the desired configuration.

In order to achieve a planar pattern we have defined a new elastic potential function

$$V_{plan,i} = \frac{1}{2}(z_i)^2 \qquad \forall i \in I$$

This function introduces a virtual spring, whose rest position is when $z = 0$, thus it has a minimum when the pattern is aligned with plane $z = 0$. Its derivative

$$\frac{\partial V_{plan,i}(\mathbf{x})}{\partial \mathbf{x}_i} = k_{plan} \begin{bmatrix} 0 \\ 0 \\ z_i \end{bmatrix} = F_{plan,i}$$

with k_{pos} being the position gain, will be added to the one previously defined.

Thus, the dynamics of each agent i is given by

$$\dot{\mathbf{x}}_i(x) = f_i(\mathbf{x}(t)) = -\sum_{j \in N_i} F_{form,ij} - F_{plan,i} \qquad \forall i \in I$$

We have discretized the dynamics by means of Euler method

$$\mathbf{x}_i^{k+1} = \mathbf{x}_i^k + \Delta \cdot f_i(\mathbf{x}^k)$$

where Δ is the sampling period. This discretization has been also applied to the next tasks.

4.3 Task 2.2 - Collision Avoidance

In order to implement collision avoidance, we have implemented the following barrier function

$$V_{coll,ij}(\mathbf{x}) = -\log(\|\mathbf{x}_i - \mathbf{x}_j\|^2)$$

The idea is to get a function which has the behaviour

$$\lim_{\|\mathbf{x}_i - \mathbf{x}_j\| \to 0} V_{coll,ij}(\|\mathbf{x}_i - \mathbf{x}_j\|) = +\infty$$

which means that if two agents get closer the potential value increases.
The implemented function has the following graph:

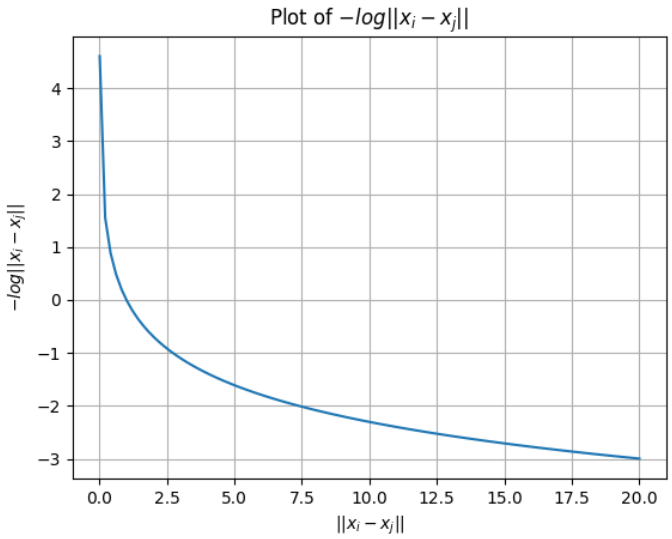


Figure 4.1: Plot of collision avoidance barrier function

Following the previously explained approach, we have added to the derivative of the potential function the following derivative of the barrier function

$$\frac{\partial V_{coll,ij}(\mathbf{x})}{\partial \mathbf{x}_i} = k_{coll} \frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|^2} = F_{coll,ij} \qquad \forall i \in I, j \in \{I \setminus i\}$$

with k_{coll} being the position gain.

In order to make the collision avoidance working, each agent must know the position of any other agents. This means that we have implemented, only for this task of collision avoidance, a "full" communication. It is a simplification of a real world scenario in which the collision avoidance mechanism will be activated only when two agents distance is under a certain threshold by means of a RADAR or LIDAR system.

Thus being said, the dynamics of each agent i is now given by:

$$\dot{\mathbf{x}}_i(t) = -\sum_{j \in N_i} F_{form,ij} - F_{plan,i} + \sum_{j \in \{I \setminus i\}} F_{coll,ij} \qquad \forall i \in I$$

In all the simulations we have run, our collision avoidance system has proven to work correctly, without achieving any robot collision.

4.4 Task 2.3 - Moving Formation and Leader control

In order to implement this task we have first to identify if the chosen formation has been reached. To recognise that we have evaluated the norm of the dynamics $f_i(\mathbf{x}_i(t))\forall i \in I$ and checked if these values were below a certain threshold, which we have empirically defined.
Once this condition has been verified for every robots we can proceed by selecting one of the agents as a leader. Then we will modify its dynamics in such a way it will be able to reach a given target position. To do so, we have implemented the following proportional controller

$$u_l(t) = k_{speed}(\mathbf{x}_{target} - \mathbf{x}(t))$$

thus, the dynamic of the leader l becomes

$$\dot{\mathbf{x}}_l(t) = u_l(t) - F_{plan,l} + \sum_{j \in \{I \setminus l\}} F_{coll,lj}$$

Since the dynamics of the other robots remains unchanged, they will try to follow the leader movement in order to keep the formation.

By doing so, a problem at the end of the motion arises: once the leader's control action fades away since the goal has been reached, it will still move. This is due to the fact that followers keep moving towards him due to their formation attractive potential, while the leader will try to stay as far as possible to them in order to decrease the repulsive contribution of the potential (collision avoidance).
In order to tackle the problem we have modified the collision avoidance mechanism by inserting a threshold below which the repulsive action activates.

We've tested this task in several simulations, as we'll show below. In all of them it worked well.

4.5 Task 2.4 - Obstacle Avoidance

In order to implement an obstacle avoidance system, we have again exploited the concept of barrier functions. Since we have designed obstacles as volumetric objects much larger than the robots, we needed a steepest barrier function that could guarantee that the robots wouldn't enter the volume of the obstacles. Thus, we have chosen an hyperbolic barrier function

$$V_{obst,ij}(\mathbf{x}) = \frac{1}{\|\mathbf{x}_i - \mathbf{x}_{obst,j}\|^4}$$

with $\mathbf{x}_{obst,j} \in \mathbb{R}^3$ being the obstacle position. The obstacle avoidance barrier function is plotted together with the collision avoidance barrier function in the following graph:

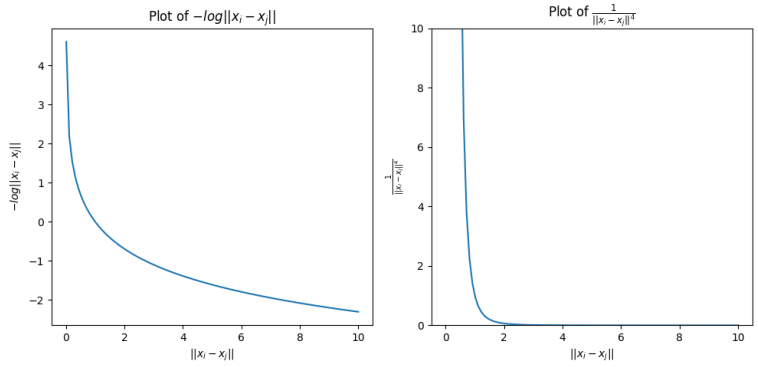


Figure 4.2: Plot of collision avoidance and obstacle avoidance barrier function

As we can see, the hyperbolic barrier function is more aggressive than the logaritmic counterpart.

Following the previously explained approach, we have added to the derivative of the potential function the following derivative of the barrier function

$$\frac{\partial V_{obst,ij}(\mathbf{x})}{\partial \mathbf{x}_i} = k_{obst} \frac{\mathbf{x}_i - \mathbf{x}_{obst,j}}{\|\mathbf{x}_i - \mathbf{x}_{obst,j}\|^5} = F_{obst,ij} \qquad \forall i \in I, j \in O$$

with k_{obst} being the position gain and O the set of obstacles.

The obstacle positions have been selected in order to obstruct the goal of the previous task.

Thus being said, the dynamics of each agent i excluding the leader is now given by:

$$\dot{\mathbf{x}}_i(t) = -\sum_{j \in N_i} F_{form,ij} - F_{plan,i} + \sum_{j \in \{I \setminus i\}} F_{coll,ij} + \sum_{j \in O} F_{obst,ij} \qquad \forall i \in \{I \setminus l\}$$

while, the dynamic of the leader l is given by

$$\dot{\mathbf{x}}_l(t) = u_l(t) - F_{plan,l} + \sum_{j \in \{I \setminus l\}} F_{coll,lj} + \sum_{j \in O} F_{obst,ij}$$

4.6 ROS 2 implementation

In order to implement such Formation Control algorithm we used ROS 2, which stands for Robot Operating System 2. It's an open-source, dis-

tributed and modular framework designed to support real-time and resource-constrained systems.

4.6.1 Package structure

We then created a workspace and a package called *formation_control*, within which we defined the launch files, the scripts to be launched and the configurations file. Let's take a closer look at each of these elements.

4.6.2 Launch files

In ROS2, a launch file is a configuration file that defines how to start a set of ROS nodes and configure their parameters. A Python launch file is written using the launch API in Python, instead of the traditional XML format. It typically starts with importing the necessary modules, such as os and launch, and then creates a launch description object to define the nodes and their parameters.
In our case, we decided to create as many launch files as the number of formations we'd like to perform:

- *pentagon.launch.py*
- *hexagon.launch.py*
- *cube.launch.py*
- *letterD.launch.py*
- *letterA.launch.py*
- *letterS.launch.py*

Within each of these files, we defined the distance matrix $D_{i,j}$, as well as many other useful parameters, such as the list of the agent's neighbours or the target position to be reached at the end of the movement.
Finally, a node running *generic_agent.py* was launched for each of the robots and the parameters just defined were given as input. Let's note that for visualisation purposes additional nodes running **rviz2**, *generic_obstacle.py* and *visualizer.py* were also launched.

4.6.3 Generic Agent

Let's give a quick overview of the code which will run for each robot.
Each node initializes itself by subscribing to two kind of topics:

- topic_agent_i $\quad \forall i \in \{I \setminus i\}$
- topic_obstacle_j $\quad \forall j \in O$

and by creating a publisher:

- called topic_agent_i

Topics are a useful communication channel provided by ROS 2 used to exchange data between different nodes.
Here is a quick example of a structure of nodes communicating via topics on ROS 2

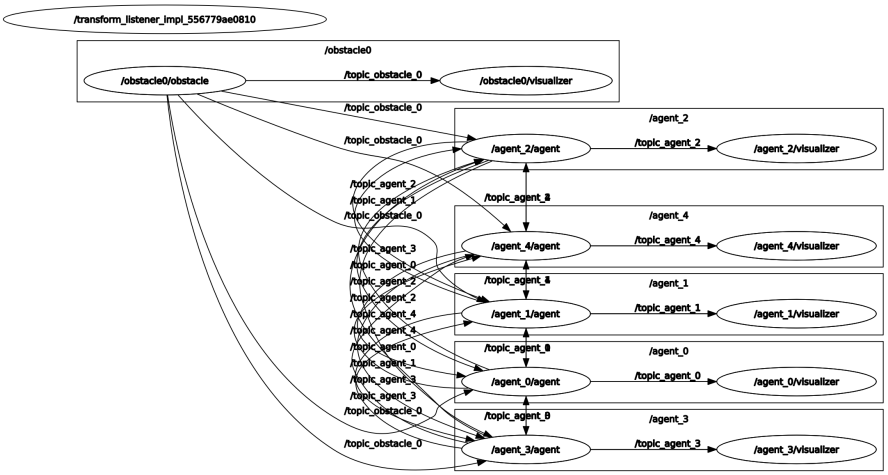


Figure 4.3: nodes-topics structure provided by rqt_graph

In order to be able to guarantee the synchronisation among all the nodes we had to check, thanks to the agent's attribute *self.kk*, whether the data analyzed belonged to the same iteration or not.

Once the synchronisation is achieved we can proceed to compute the update of each agent's position. This was done by simply implementing the discretized dynamics shown earlier, obtained with an Euler step of 10^{-3}.
We decided to add a flag to check that the agents are in position and that the formation has been obtained. As soon as this flag becomes True, the formation starts moving towards the target.

4.7 Results

We have run a set of simulation imposing different patterns, that have shown the effectiveness of our algorithm, as we will see below.

For all the simulations we have recorded videos, that you can find in the shared Onedrive folder.

4.7.1 Letters D,A,S

In order to show the effectiveness of the first two tasks (formation control and collision avoidance) we have chosen as patterns the letters D, A and S. We have imposed them to be planar. For these patterns we have chosen to use $N = 6$ robots.
In order to get them we have defined the following distances matrices:

- Letter D:

$$D_{i,j} = \begin{bmatrix} 0 & L & 2L & 0 & 0 & L \\ L & 0 & L & L & H & L \\ 2L & L & 0 & L & 0 & 0 \\ 0 & L & L & 0 & \frac{L}{2} & L \\ 0 & H & 0 & \frac{L}{2} & 0 & \frac{L}{2} \\ L & L & 0 & L & \frac{L}{2} & 0 \end{bmatrix}$$

  with distances L and $H = \frac{\sqrt{3}}{2}L$ being specifically selected.

- Letter A:

$$D_{i,j} = \begin{bmatrix} 0 & L & L & H & 2L & 2L \\ L & 0 & L & \frac{L}{2} & L & 0 \\ L & L & 0 & \frac{L}{2} & 0 & L \\ H & \frac{L}{2} & \frac{L}{2} & 0 & 0 & D \\ 2L & L & 0 & 0 & 0 & 2L \\ 2L & 0 & L & D & 2L & 0 \end{bmatrix}$$

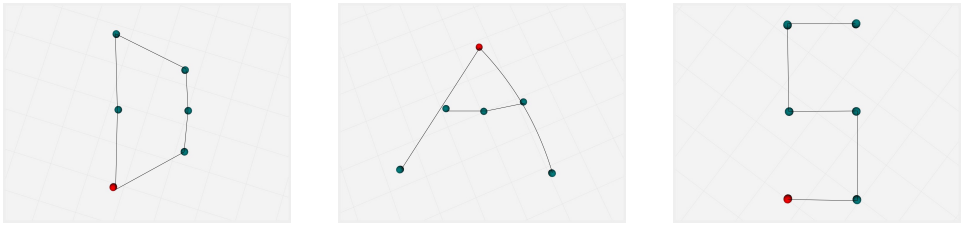
  with distances L, $H = \frac{\sqrt{3}}{2}L$ and $D = \frac{\sqrt{7}}{2}L$ being specifically selected.

- Letter S:

$$D_{i,j} = \begin{bmatrix} 0 & H & D & L & 2L & 0 \\ H & 0 & L & D & 0 & 0 \\ D & L & 0 & H & D & L \\ L & D & H & 0 & L & D \\ 2L & 0 & D & L & 0 & H \\ 0 & 0 & L & D & H & 0 \end{bmatrix}$$

  with distances L, $H = \frac{3}{4}L$ and $D = \frac{5}{4}L$ being specifically selected.

These matrices have been designed with the minimum number of fixed distances possible in order to appreciate the stability and robustness of our distributed algorithm.
The results of the simulations are



(a) Letter D (b) Letter A (c) Letter S

Figure 4.4: The three letters in a row

We have manually added links between the nodes to enhance visualization.

Letter D pattern is reached after approximately 1400 iterations (which are equivalent to 1.4 seconds).
Letter A pattern is reached after approximately 1000 iterations (which are equivalent to 1.0 seconds).
Letter S pattern is reached after approximately 1500 iterations (which are equivalent to 1.5 seconds).

4.7.2 Pentagon

After the letters, we moved on to the polygon patterns, for which we have chosen a leader who will start to move when the pattern is achieved in order to reach a randomly chosen target. The first polygon we have selected was a pentagon, with $N = 5$ robots. We have imposed it to be planar.
For the pentagon, we have defined the following distances matrix

$$D_{i,j} = \begin{bmatrix} 0 & L & D & D & L \\ L & 0 & L & D & D \\ D & L & 0 & L & 0 \\ D & D & L & 0 & L \\ L & D & 0 & L & 0 \end{bmatrix}$$

with distances L and $D = \frac{\sqrt{5}+1}{2}L$ being specifically selected.

The results of the simulation after approximately 2700 iterations (which are equivalent to 2.7 seconds) is

Figure 4.5: Pentagon

After reaching the pattern, the leader correctly switches its dynamics and starts moving towards the target. The other robots follow the leader correctly, however the formation is temporarily broken. Eventually, the target is reached by the leader and the formation is reconstructed.
It is worth to note that, by decreasing the gain of the proportional controller, we can reach the target while keeping the formation intact.

4.7.3 Hexagon

The second polygon we have imposed was a hexagon, with $N = 6$ robots. We have imposed it to be planar. Again, we have implemented the moving leader.
For the hexagon, we have defined the following distances matrix

$$D_{i,j} = \begin{bmatrix} 0 & L & 0 & D & H & L \\ L & 0 & L & 0 & D & 0 \\ 0 & L & 0 & L & 0 & D \\ D & 0 & L & 0 & L & 0 \\ H & D & 0 & L & 0 & L \\ L & 0 & D & 0 & L & 0 \end{bmatrix}$$

with distances L, $D = 2L$ and $H = \sqrt{3}L$ being specifically selected.

The results of the simulation after approximately 900 iterations (which are equivalent to 0.9 seconds) is

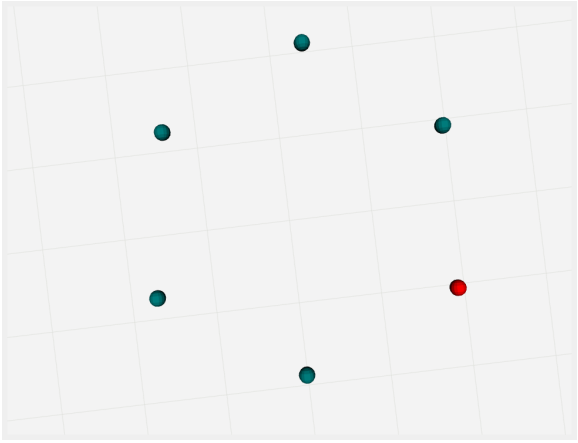


Figure 4.6: Hexagon

Again, the moving formation and leader control task worked correctly.

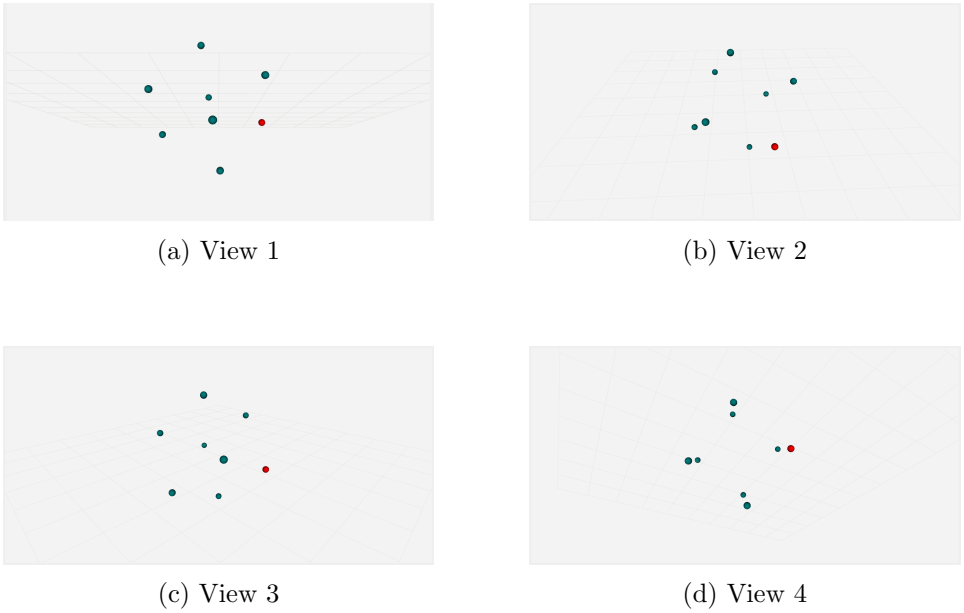
4.7.4 Cube

After polygons we have moved to a 3D figure: a cube, with N=8 robots. In this case we have implemented the obstacle avoidance task.
For the cube, we have defined the following distance matrix

$$D_{i,j} = \begin{bmatrix} 0 & L & D_m & 0 & 0 & D_m & D_M & D_m \\ L & 0 & 0 & D_m & D_m & 0 & D_m & D_M \\ D_m & 0 & 0 & 0 & D_M & D_m & 0 & D_m \\ 0 & D_m & 0 & 0 & D_m & D_M & D_m & 0 \\ 0 & D_m & D_M & D_m & 0 & L & D_m & 0 \\ D_m & 0 & D_m & D_M & L & 0 & L & D_m \\ D_M & D_m & 0 & D_m & D_m & L & 0 & 0 \\ D_m & D_M & D_m & 0 & 0 & D_m & 0 & 0 \end{bmatrix}$$

with distances L, $D_M = \sqrt{3}L$ and $D_m = \sqrt{2}L$ being specifically selected.

We now provide several views of the result of the simulation after approximately 1700 iterations (which are equivalent to 1.7 seconds)



(a) View 1 (b) View 2

(c) View 3 (d) View 4

Figure 4.7: Cube

After reaching the pattern, the leader correctly switches its dynamics and starts moving towards the target. The other robots follow the leader correctly. All the robots are able to avoid correctly the obstacle during the movement. Eventually, the target is reached by the leader and the formation is reconstructed.

Chapter 5

Conclusions

To conclude our work, we would like to highlight some crucial points observed during the development of the project.

In the first task, the "gradient tracking" introduction greatly improved the speed of the convergence to consensus. We expected a similar behaviour, yet not to such a considerable degree.

In the second task, the main surprise was the significant impact of the sparsity of the distance matrix on the convergence to the desired formation. Eventually, it was intriguing to observe the agents' response to the various potentials we tested, particularly the repulsive ones introduced by obstacles.