UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Distributed Autonomous Systems

Final Project Report

Professors:
Giuseppe Notarstefano
Ivano Notarnicola

Students:
Rossella Attanasi
Filippo Guarda
Cristian Urbinati

Academic year 2021/2022

Abstract

These projects are aimed to acquire an understanding of distributed systems control trough an hands-on approach.
The first task concerns the training of a distributed neural network model on a classification task over the MNIST dataset of handwritten digits. The model has to confidently classify a chosen digit against the others.
The second task involves the formation control of multiple agents modeled by double-integrator dynamics in ROS2. The agents have to control the translation and scale of a desired formation while maintaining the desired formation pattern. The leaders move with constant velocity (possibly zero), while the followers dispose themselves to keep the formation.

Contents

Chapter 1

Distributed Classification via Neural Network

The objective of this task is to build a set of N agents that cooperatively determine a binary classifier capable of discriminating one selected digit over the set of 70.000, 28×28 grayscale images of hand-written digits from the **MNIST** dataset.

1.1 Resolution strategy

We start loading the dataset train and test splits from keras.datasets [2], a module that provides a few datasets already vectorized in Numpy format. Each sample represents a set of pairs $(\mathcal{D}^i, y^i)$ where $\mathcal{D}^i \in [0, 255]^{28\times 28}$ is the grayscale image of the digit represented as a matrix and y^i its associated label in $\{0, \cdots, 9\}$, with $i \in [1, n\_samples]$. As suggested, we re-scale images values in the interval $[0, 1]$ dividing each image by 255.0 and then we reshape it to a columnar vector with shape $(1, 784)$. We also change label values to fit the binary classification scenario assigning 1 at the chosen class and 0 to the others. This is because we intended to use the Softmax activation function that squeezes output values in the range $[0, 1]$.
Next step was to equip each agent i with a set of $m_i \in \mathbb{N}$ images and labels in order to perform the training. We manage to perform this task in two ways: with and without sampling.
Without sampling the training images are randomly picked from the shuffled dataset with a probability distribution of $\frac{1}{10}$ to select the chosen class (determined by the balanced distribution of images per digits in the dataset).
With sampling instead we force half of the samples to belong to the class we want to classify, this allows to have a distribution of $\frac{1}{2}$ of positive class per agent and contribute to speed up the convergence.
In this context, each agent sees only a subset of images and share its weights with neighbors. Edges between agents have been modelled generating a strongly connected and aperiodic binomial graph. Also a mixing matrix, satisfying row and column stochasticity properties, is used to determine the relevance of the weights of the self agent and the one of its neighbors in the consensus pipeline.

At this point, we manage to run the Distributed Gradient Tracking algorithm (causal form) to train the neural network using different dataset sizes. Each Neural Network is fed with one image at a time then, since we are interested only in a binary prediction but we have 784 neuron at each layer (including the last one), we choose the last neuron as representative of the probability score in predicting the chosen class (one is as good as the other in this case except for the bias). We found out the optimal number of layers around 3 and 4 since with less layers the network suffers in expressivity capacity, while with more the network requires more time to train and since it becomes deep it may suffer of vanishing gradients. In order to calculate the cost we implemented and tested both Mean Squared Error and Binary Cross Entropy loss functions but, even if BCE fits better the task, we have not been able to obtain convergence using it at the beginning. After some researches we find out that Cross Entropy losses applied next to an activation function like Sigmoid as in our case, are subjected to numerical instability, and for that reason libraries like Tensorflow implement a slightly modified version [1]. Think for example the case in which the output of the final layer is a value $z << 0$ (especially at the beginning of training when a positive example might be confidently classified as a negative one), in this case the Sigmoid function will return a value $\sigma(z) \approx 0$ and so the first logarithm inside the BCE may cause overflow. The trick here is to modify the Sigmoid function using its two equivalent expressions to deal with numeric overflow and directly integrate it inside the loss function as below:

$$
\begin{aligned}
\sigma(z) &= \begin{cases} \frac{1}{1+e^{-z}} & z \geq 0 \\ \frac{e^{z}}{1+e^{z}} & z < 0 \end{cases} \\
L_{BCE}(z, y) &= -y \log(\sigma(z)) - (1-y)\log(1-\sigma(z)) \\
&= \begin{cases} y \log(1+e^{-z}) - (1-y)\left(\log(e^{-z}) - \log(1+e^{-z})\right) & z \geq 0 \\ -y\left(z - \log(1+e^{z})\right) - (1-y)\left(\log(1) - \log(1+e^{z})\right) & z < 0 \end{cases} \\
&= \begin{cases} y \log(1+e^{-z}) + (1-y)\left(z + \log(1+e^{-z})\right) & z \geq 0 \\ -y\left(z - \log(1+e^{z})\right) + (1-y)\left(\log(1+e^{z})\right) & z < 0 \end{cases} \\
&= \begin{cases} z - yz + \log(1+e^{-z}) & z \geq 0 \\ -yz + \log(1+e^{z}) & z < 0 \end{cases}
\end{aligned}
$$

The choice between BCE and MSE lies in their respective specialization for classification and estimation. Binary cross entropy, being based on logistic regression, is better suited on an error that follows a binomial distribution while MSE is better suited for errors following a normal distribution. During the learning process, for each iteration, both the costs and weight updates obtained from backward pass are summed over the entire set of samples for each agent and then the system states' distributed update is performed.

1.2 Simulation results

We run different experiments both with Mean Squared Error and Binary Cross Entropy losses. Figures 1.1 and 1.2 show respectively the trend of the cost error total for each agent with MSE, while figures 1.4 and 1.5 do the same for BCE loss. Furthermore we plot the evolution of the gradient of an agent for each layer by summing gradients of all neuron, as we can see in Fig. 1.3 for the MSE and in Fig. 1.6 for the BCE (In blue the first layer, in orange the second). We can that the MSE doesn't fit very well the binary classification scenario as the generated gradient goes up and down in the last layer.

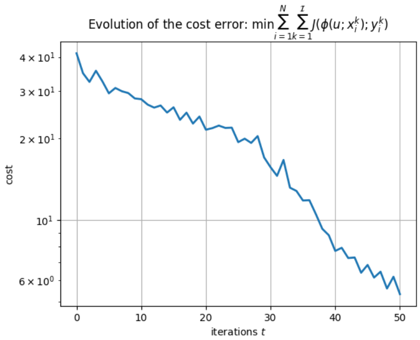
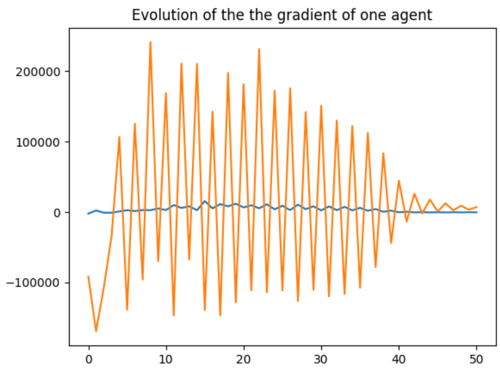


Figure 1.1 – Total cost error

Figure 1.2 – Cost error per agent

Figure 1.3 – Evolution of the gradient of one agent for each layer



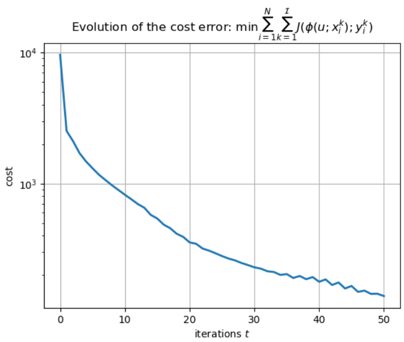


Figure 1.4 – Total cost error

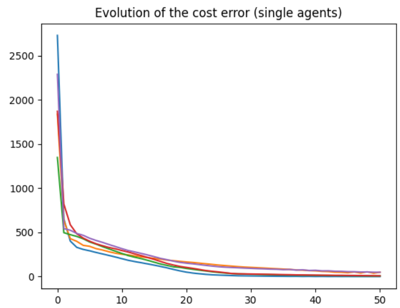
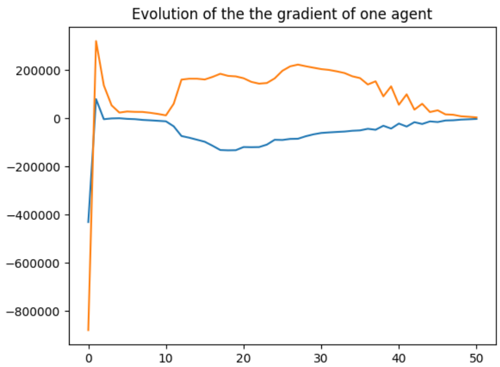


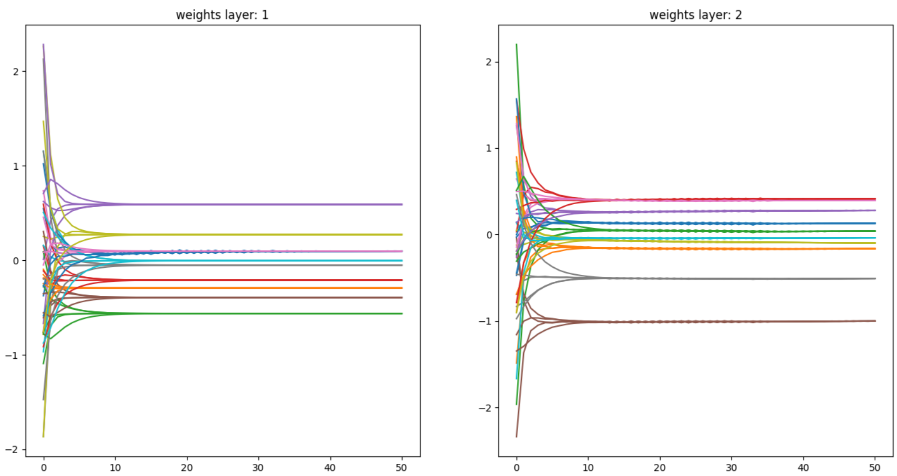
Figure 1.5 – Cost error per agent

Figure 1.6 – Evolution of the gradient of one agent for each layer



We noticed that errors generated by BCE are much higher due to the logarithm but the general trend of learning curve appears more stable and with a faster convergence.
Furthermore we plot the convergence of first 10 weights of each Neural Network layer (except for the output one) for a simulation with a 3-layers model as shown in Fig. 1.7. Same color means the same weight index for the set of agents.

Figure 1.7 – Weights convergence



At the end we test the the first agent model on a sample of test set images evaluating the prediction 1 if the output score is higher than 0.5, 0 otherwise. The highest accuracy obtained is 92.55% using the BCE loss and has been calculated using the function provided in **sklearn.metrics** library taking in consideration the distribution of samples in the test set as well.

Chapter 2

Formation Control

The objective of this task is to model a double-integrator dynamics system to control the translation and scale of a network of N robotic agents, moving on the Euclidean plane ($d = 2$), while maintaining a desired formation pattern.
Agents are partitioned into two groups, namely leaders and followers, the leaders have their own independent dynamics and condition the followers dynamics by influencing the state error.

2.1 Resolution strategy

We manage to describe the desired formation positions and the adjacency matrix in a single configuration files, one for each experiment. In the code `n_leaders` is the number of the leaders, chosen equal to 2 in any simulation shown in this report while the remaining ones as followers. For each robot $i = 1, \cdots, N$ we define $\mathbf{p}_i$, $\mathbf{v}_i \in \mathbb{R}^d$ as the position and velocity of agent i. So we are able to collect the positions and velocities of all the agents in a column vector: $\mathbf{p}$, $\mathbf{v} \in \mathbb{R}^{dN}$, as a result we define the state space as follows:

$$\mathbf{x} = \begin{bmatrix} \mathbf{p} \\ \mathbf{v} \end{bmatrix}.$$

Leaders initial velocities can be zero or constant while followers initial velocities are randomly initialized. Leaders initial positions are assigned to the desired position in case of zero-velocities or randomly initialized in the other case. Followers positions are always initialized randomly.
Next we move to the design of the control law for followers acceleration described in [3]:

$$u_i(t) = -\sum_{j \in \mathcal{N}_i} P^*_{g_{ij}} \left[k_p \left(\mathbf{p}_i(t) - \mathbf{p}_j(t) \right) + k_v \left(\mathbf{v}_i(t) - \mathbf{v}_j(t) \right) \right]; \tag{2.1}$$

where $\mathbf{p}_i, \mathbf{v}_i \in \mathbb{R}^d$ are respectively the position and the velocity in the plane of agent i; k_p and k_v are positive constant gains and $P^*_{g_{ij}} \in \mathbb{R}^{d \times d}$ an orthogonal projection matrix associated to the desired bearing unit vector of agent j relative to agent i defined as follows:

$$P^*_{\mathbf{g}_{ij}} := I_d - \mathbf{g}^*_{ij} \mathbf{g}^{*\top}_{ij}.$$

The desired bearing unit vectors $\mathbf{g}_{ij}^*$ have been calculated through a dedicated function starting from the position of the agents in the desired formation as:

$$\mathbf{g}_{ij}^* := \frac{\mathbf{p}_j^* - \mathbf{p}_i^*}{||\mathbf{p}_j^* - \mathbf{p}_i^*||}.$$

We also calculated and stored each bearing unit vector in the matrix `GG` to check that it is anti-symmetric. Starting from them we menage to calculate matrix $P_{\mathbf{g}_{ij}}^*$ and then we check its correctness through the determinant of the Bearing Laplacian Matrix defined as:

$$[\mathcal{B}(G(\mathbf{p}^*))]_{ij} = \begin{cases} \mathbf{0}_{d\times d} & i \neq j, (i,j) \notin \mathcal{E} \\ -P_{\mathbf{g}_{ij}^*} & i \neq j, (i,j) \in \mathcal{E} \\ \sum_{k\in\mathcal{N}_i} P_{\mathbf{g}_{ij}^*} & i = j, i \in \mathcal{V} \end{cases},$$

such that,

$$\mathcal{B} = \begin{bmatrix} \mathcal{B}_{ll} & \mathcal{B}_{lf} \\ \mathcal{B}_{fl} & \mathcal{B}_{ff} \end{bmatrix}.$$

In fact, in order to guarantee the Uniqueness of the Target Formation the Bearing Laplacian Matrix should be unique and so $\det(\mathcal{B}_{ff}) \neq 0$, as demonstrated in [3, Theorem 1].

With all functions set, we start writing a discrete-time version of the model first; at each timestep we calculate the acceleration vector $\mathbf{u}$, which modifies speeds and positions in the following iterations with the control law function in (2.1). Then the state update is calculated by means of the system:

$$\mathrm{d}\mathbf{x} = \begin{bmatrix} \mathrm{d}\mathbf{p} \\ \mathrm{d}\mathbf{v} \end{bmatrix} = A\mathbf{x} + B\mathbf{u} = \begin{bmatrix} 0 & I \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ \mathbf{v} \end{bmatrix} + \begin{bmatrix} 0 \\ I \end{bmatrix} \begin{bmatrix} \mathbf{u}_l \\ \mathbf{u}_f \end{bmatrix},$$

and finally we update the state $\mathbf{x} = \mathbf{x} + \mathrm{d}\mathbf{x} * \mathrm{d}t$.

Finally we move to implement the same logic in ROS2. The only differences in that case is that each agent stores its own position and velocity, and communicates them only to its neighbors. Moreover the control law is calculated by each agent independently given the positions and velocities read from its neighbors topics.

2.2 Simulation results

In this section we present some simulation results.

First we show a simulation with 4 agents, 2 leaders and 2 followers, which aim is to reach a square formation. We manage to store robots positions at each time step and distance error from the desired formation in order to get some plots. For example we can see in Fig.2.1 trajectories of agents reaching the formation where leaders have constant velocity and in Fig.2.2 the evolution of the distance error that asymptotically converges to zero.

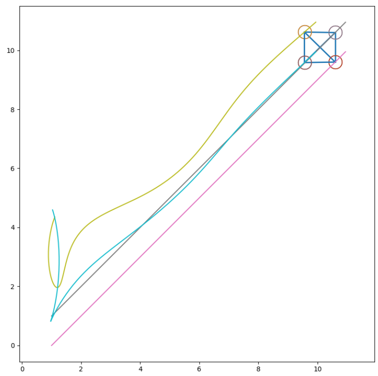


Figure 2.1 – Square formation

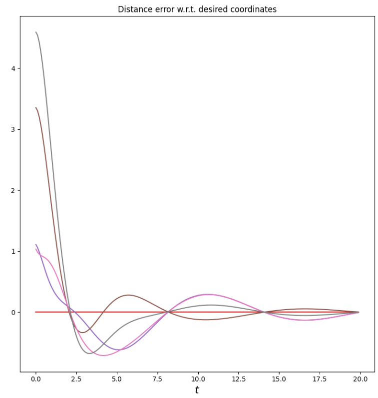


Figure 2.2 – Distance Error

We also handle some simulation in ROS2 varying the formation pattern and the number of agents so that robots draw letters of the word `DAS` and we display the resulting agent configuration with RViz as can be seen in Fig.2.3.

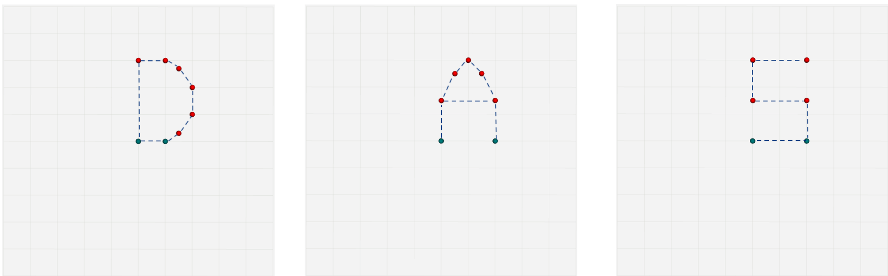


Figure 2.3 – Formations displaying the word DAS

Conclusions

The objective of this work was to develop an understanding of distributed systems and their control. This has been achieved by successfully completing two projects that display the effectiveness of control systems both for distributed neural networks and multi-agent robotics.
We achieved high accuracy in the the digit classification task and convergence speed in the formation control.

Bibliography

[1] Rafay Khan. How do tensorflow and keras implement binary classification and the binary cross-entropy function?, 2019.

[2] Ronald T. Kneusel. *Practical Deep Learning: A Python-Based Introduction.* No Starch Press, 2021.

[3] Shiyu Zhao and Daniel Zelazo. Translational and scaling formation maneuver control via a bearing-based approach, 2015.