

UNIVERSITÀ DI BOLOGNA



School of Engineering
Master Degree in Automation Engineering

Distributed Autonomous Systems
Course Project #1

Professors:
Giuseppe Notarstefano
Ivano Notarnicola

Students:
Julie Dahl Hjelle
Helene Bolkan

Academic year 2021/2022

Abstract

This report focuses on practical problems in related to Distributed Autonomous Systems. We study distributed classification via neural networks and formation control. The distributed classification is related to handwritten letters from the MNIST database. The neural network is trained with the Distributed Gradient Tracking. The second part is related to cyber-physical systems, where you turn smart devices into Cooperative Intelligent Systems. We can have smart cities, smart grid and automated factories. In the formation control section we study distributed maneuver control of multi-agent formations in two-dimensions. The goal is to control the translation and scale of the formation while maintaining the desired formation pattern. The approach is bearing-based, where the target formation is defined by inter-neighbor bearings.

Contents

Introduction	5
1 Distributed Classification via Neural Networks	7
1.1 Architecture of the task	7
1.2 MNIST database and manipulation of training set	7
1.3 Distributed Gradient Tracking	8
1.4 Results	10
1.4.1 Scenario 1: 15 iters, 25 images, 5 and 10 agents	10
1.4.2 Scenario 2: 15 iters, 100 images, 5 and 10 agents . . .	14
1.4.3 Scenario 3: 10 agents, 25 images, 15 and 30 iterations	14
1.4.4 Scenario 4: 10 agents, 15 iterations, 25 and 500 images	20
2 Formation Control	23
2.1 Architecture of the task	23
2.2 Discretization of model	23
2.3 ROS2 - Robot Operating Systems 2	24
2.4 Results	27
2.4.1 Formation patterns with different number of robots .	27
2.4.2 Formation patterns with letters of a chosen word . . .	30
Conclusions	35
2.5 Distributed Classification	35
2.6 Formation Control	36
Bibliography	38

Introduction

The first part will elaborate the Distributed classification via Neural networks, and the second part will describe Formation control.

In task number one the goal was to retrieve a set of hand written digits with corresponding label from the MNIST database, and train a neural network to recognise these digits. In order to do so, we had to split the retrieved data into different training sets for each agent, and run the Distributed Gradient Tracking algorithm on each agent. To make the training process a bit easier for the agents, the neural network only had to learn to recognise one of nine digits. After running through the entire training set for each agent, we checked the accuracy of the results by running the same algorithm on a training set, where each node tried to correctly label the same digits. In this way we could easily compute the accuracy of the agents. We also compute the total cost of the neural network, to see how expensive it is for the agents to train themselves to recognise the digits.

For the second part we used the ROS2 (Robot Operating Systems) software to do the formation control. We make a network of N agents where each agent has a position and velocity. The robotic agents will control the translation and scale of a desired formation while maintaining the desired formation pattern. We have leaders and followers, where the followers will move with a specified following control law. The formation patterns include a square, a hexagon and an octagon. In addition we developed formation patterns so each agent will draw one letter at a time. All the letters will show the word ciao. These formations will be visualized with Rviz.

Motivations

As time goes by, more and more distributed autonomous systems are taking part of our human life. These autonomous systems can give us huge benefits in life, businesses and production. Repetitive tasks before done by humans, can now be set in process by systems of technology. We have seen these past years, that Unmanned Aerial Vehicle, also commonly known as drones, has been incorporated into many sectors. These drones can also be autonomous and able to take intelligent decisions without input from a pi-

lot and learning to adapt to the environment. There are many applications of distributed systems; for example distributed optimization. Distributed optimization can include distributed machine learning, distributed decision-making in cooperative robotics, and distributed optimal control in energy systems and cooperative robotics. In this project we use our applied knowledge of the theory from the course to solve practical problems. We will focus on consensus and formation control.

Contributions

Thanks to the tutor Lorenzo Pichierri for help with the project. Thanks to Professor Notarnicola for a theory meeting. The chapters will explain how agents could reach consensus by communicating with its neighbors.

Chapter 1

Distributed Classification via Neural Networks

The first task concerns a distributed classification problem, where N agents should agree on a common classifier, after training on different parts of a training set. Each agent should train their own neural network, and communicate with their neighbours to agree on the weights on the different neurons in order to come up with the optimal solution. The agents are using the Gradient Tracking algorithm. The digits to recognise was retrieved from the MNIST database.

1.1 Architecture of the task

The entire task is programmed using the Python language, and everything is implemented in one single file. We have made this choice in order to keep everything in one place, and have full control of the behavior of the code. The code do not use many built-in libraries from Python, we mainly program everything ourselves. The three libraries that we have included in the system is: mnist from keras.datasets, to help us with retrieving the data from MNIST, numpy to help with some mathematical functions - particularly related to arrays, and lastly matplotlib to help us with the plotting of the computed cost after running the algorithm.

1.2 MNIST database and manipulation of training set

The MNIST, which stands for Modified National Institute of Standard and Technology, database is a large collection of handwritten digits, from 0-9, and their corresponding labels. It is commonly used for training image processing systems, to teach them to recognise different digits. We used

the Keras data sets to help us with retrieving the digits from the MNIST database. The set retrieved consists of 60.000 digits in the form of matrices of 28x28 pixels, and corresponding labels. In addition we retrieve a test set of 10.000 digits and labels. In order to make it possible for us to use the matrices for the training algorithm, we had to reshape the matrices, to make it easier for the agents we turned the different matrices into arrays of 784 integers length. Thereafter we divided the value of each pixel by 255 (the maximum value of the RGB color codes), in order to get values between 0 and 1 for each pixel. [2]

After getting all the matrices in the correct order, we had to change the labels. As mentioned the neural networks is only supposed to learn to recognise one digit. This was implemented by setting the label of one digit, in our case we chose the digit 4, to be equal to 'one', while the label of all the other digits was set to 'zero'. This makes this optimization problem into a binary problem, as there is now only two labels to categorize the digits as: [0, 1]. In the description of the project it said to divide the labels into '1' and '-1', but we found this to give a low accuracy of the agents, and thus we made the decision to change to '1' and '0' instead. After this we randomly shuffled the two arrays, while keeping in mind that the matrices had to keep their corresponding label. Lastly we split the training set into N parts, one for each agent in the network. After all this data manipulation was complete, we were finally ready to use the MNIST data in our Gradient Tracking Algorithm.

1.3 Distributed Gradient Tracking

The algorithm that we were supposed to use for the training of the agents is the Gradient Tracking algorithm, which is based of the distributed gradient method. This gradient algorithm tries to solve an optimization problem, where we are looking for the lowest cost of computation for the network as a whole. A gradient method is an algorithm with an update rule that uses the gradient of the cost function in combination with the current state and the step size to determine the next state. Here the states are values that bring us closer to the optimal solution, denoted as $f(x^*)$. Where f is the cost function and x^* is the optimal solution.

The distributed gradient algorithm solves a distributed optimization problem. This means that the different agents in the network are only aware of a part of the optimization problem, and none of them are able to compute the overall solution alone. In order to find the optimal solution each agents computes their own prediction, and then updates their computation based on

the received computations from their neighbours. This process repeats until they all agree on a common solution. The distributed gradient algorithm is as follows:

$$v_i^{k+1} = \sum_j a_{ij} x_j^k \quad (1.1)$$

$$x_i^{k+1} = v_i^{k+1} - \alpha^k \nabla f_i(v_i^{k+1}) \quad (1.2)$$

Where v_i^{k+1} is the summation of the received calculations for this round k, from all of the agents neighbours j in N_i , with respect to their weight, given by a_{ij} . Thereafter we compute the x_i^{k+1} , which is given by the computed v_i in correspondence to the gradient of the cost function of the current value for the optimal solution, given by v_i .

This algorithm is the basis for the gradient tracking, which we have used for this task. The gradient tracking is an improvement of the distributed gradient method. The idea is that we use a local descent direction at each agent, that aims to reconstruct the correct gradient. Each agent keeps a local average of all the received signals from their neighbours, and tries to track or estimate this average. This can mathematically be written as:

$$x_{t+1} = Ax_t - \alpha y_t \quad (1.3)$$

$$y_{t+1} = Ay_t + \nabla f(x_{t+1}) - \nabla f(x_t) \quad (1.4)$$

Where $A \in \mathbb{R}^{N \times N}$ is the matrix of the weights a_{ij} . Both x, y and ∇f are given as a vector from 1,...,N for all agents in the network. If you also want to take into account the non-linear change of coordinates in the system, the equations can be changed to:

$$x_{t+1} = Ax_t + x_t - \alpha \nabla f(x_t) \quad (1.5)$$

$$z_{t+1} = Az_t - \alpha(A - I)\nabla f(x_t) \quad (1.6)$$

Where I is the identity matrix. This is a causal dynamic system, and it is this system that we have implemented in our code to execute the gradient tracking over our nodes.

We needed some methods that could help us with the underlying logic, i.e. finding the gradient of the cost function, for the update of the optimal value at the agents. Firstly we implemented a Sigmoid-function and also its derivative. The Sigmoid was used for the forward propagation part of the method, while the derivative was used for the backwards propagation. The forward propagation was used to generate the state of the next layer, x_{t+1} by applying the inference dynamics to the current state. The backwards propagation was used to find the gradient that we needed in the update of the weights and local estimate, in this process we used the adjoint dynamics.

1.4 Results

The results of this neural network training with the usage of the gradient tracking method can be measured in several ways. Firstly we can calculate the cost of the solution, that gives an indication to whether we have found the optimal solution or not. In addition we can calculate the accuracy that the trained agents receive on the test set of the database, after executing the gradient tracking algorithm on the training set. Both of these methods have been implemented in this project, and we will here show you the results of the estimate of cost as well as the accuracy of agents.

The results of the distributed classification will be visualized in the form of a plot of the evolution of the cost function with respect to the number of iterations of the algorithm, as well as a summary of the accuracy of the agent, computed and printed to the terminal in the project.

The accuracy of the different agents will depend on the different parameters that are possible to tune in the project. These include the number of agents, the number of iterations of the algorithm, the number of images per agent, and lastly the step size or learning rate of the agents. The cost function will mainly depend on the number of images computed for each agent, but this also depend a bit on the number of agents in the network.

In order to observe how the different factors influence the results of our classification, we have tried changing both the number of agents, the number of iterations and the number of images per agent. For agents we have tried with both 5 and 10 agents, for iterations we have tried with both 15 and 30, and for number of images we have tried with 25, 50, 100 and even 500. We have not had the time to tried all possible combinations of these parameters, but we have tried to try enough variations to obtain a balanced picture of how the different parameters influence the result.

We will divide the result part of this task into several parts, dependent on the different parameter configurations:

- 15 iterations, 25 images per agent for both 5 and 10 agents
- 15 iterations, 100 images per agent for both 5 and 10 agents
- 10 agents, 25 images with 15 and 30 iterations
- 10 agents, 15 iterations and 25 and 500 images per agent.

1.4.1 Scenario 1: 15 iters, 25 images, 5 and 10 agents

The first scenario that we will be discussing in this project is having 15 iterations of the algorithm per agent and giving each agent 25 images to

train on. For this scenario we wish to see how much the accuracy and cost is influenced by the amount of agents we have in the network. For this task we have tried with 5 and 10 agents.

We can firstly observe the evolution of the cost function for each agent in the two cases. We see that both of the cases have a convergence of the cost function, which means that the agents come to an agreement that leads to the optimal solution. We can see from Figure 1.1 and Figure 1.2 that for the situation with only 5 agents the initial calculation of the cost was more equal across the agents than it was for 10 agents, i.e. there was a higher variation in the initial cost for 10 agents. This also means that it seems like the agents are not reaching as much of an consensus as they do for the 10 agent scenario, but this is not necessarily correct. One square along the y-axis represents 0.025 change for Figure 1.1, while one square is equal to 0.05 for Figure 1.2. Therefore the agents are about equally converged in the two scenarios.

On the other hand we can see that the scenario with 10 agents seem to reach the consensus earlier than for 5 agents. We observe that already after 6 iterations they have agreed on the optimal solution, while it seems that in the 5 agent case they have barely reached consensus on the 15 iterations. Perhaps it would be beneficial for the 5 agent case to increase the number of iterations, to reach more of a distinct consensus. One reason as to why the agents reach a consensus earlier when there are 10 agents could be that each agent has a higher number of neighbours. This means that each agent receives more input per iteration, which also makes it possible to adjust the estimate for the next state more for each iteration. This can lead to the agents reaching consensus earlier than when there are only 5 agents.

When it comes to the accuracy of the agents, these can be seen in Figure 1.3 and Figure 1.4. We have computed the accuracy of the agents by counting the amount of label '1' and '0' that occur in the test sequence, and then counting the amount of digits that the agent label correctly, within a threshold of 0.5 - so the label is accepted as '0' for $[0, 0.49]$ and accepted as '1' for $[0.5, 1]$. We can see that for both of these versions of scenario 1 the agents are very good at recognising the '1' labels, and not very good at recognising '0'. This can be because the digits that are labelled as '0' are everything except for digit 4. This means that $[0, 1, 2, 3, 5, 6, 7, 8, 9]$ are all passed as '0' to the algorithm, and this means that the agents need some time to figure out how to recognise a '0'. On the other hand there is only 4 that has the label '1', and this is therefore easier for the agents to learn the appearance of.

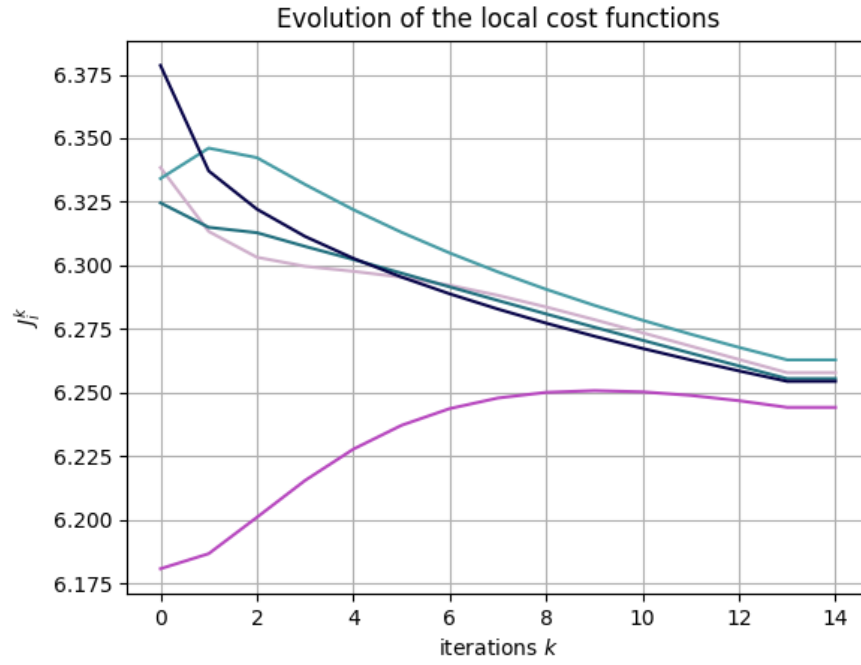


Figure 1.1: The cost function for 5 agents with 25 images and 15 iterations

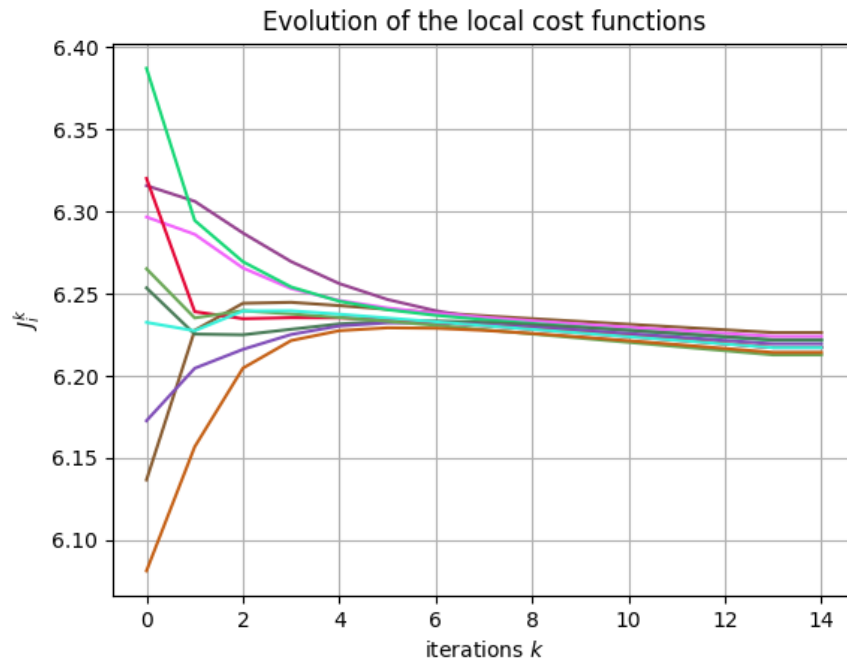


Figure 1.2: The cost function for 10 agents with 25 images and 15 iterations

```

11 11
11 13
AGENT ACCURACY
Number of test samples per agent: 100
---- Correct '1' ---- Correct '0' ----- TOTAL -----
Agent [1]: 0.786 - 0.151 - 0.240
Agent [2]: 0.857 - 0.093 - 0.200
Agent [3]: 0.786 - 0.105 - 0.200
Agent [4]: 0.786 - 0.128 - 0.220
Agent [5]: 0.786 - 0.151 - 0.240

```

Figure 1.3: The computed accuracy for 5 agents with 25 images.

```

11 26
11 28
11 28
11 29
AGENT ACCURACY
Number of test samples per agent: 100
---- Correct '1' ---- Correct '0' ----- TOTAL -----
Agent [1]: 0.786 - 0.221 - 0.300
Agent [2]: 0.786 - 0.279 - 0.350
Agent [3]: 0.786 - 0.279 - 0.350
Agent [4]: 0.786 - 0.291 - 0.360
Agent [5]: 0.786 - 0.291 - 0.360
Agent [6]: 0.786 - 0.302 - 0.370
Agent [7]: 0.786 - 0.302 - 0.370
Agent [8]: 0.786 - 0.326 - 0.390
Agent [9]: 0.786 - 0.326 - 0.390
Agent [10]: 0.786 - 0.337 - 0.400

```

Figure 1.4: The computed accuracy for 10 agents with 25 images.

1.4.2 Scenario 2: 15 iters, 100 images, 5 and 10 agents

The next scenario that we wanted to explore in our project is having 15 iterations per agent of the algorithm, and having 100 images per agent. We wanted to observe how this looked for both 5 and 10 agents, and also see if we see some different results than we got for 25 images per agent.

We can start with looking at the plotting of the cost function here as well. The version with 5 agents is shown in Figure 1.5 while the version with 10 agents is shown in Figure 1.6. Here we can see a different behaviour for the version with 5 agents than for the one with 10 agents. This behaviour is also different from the one we saw for the Scenario 1 with 25 images. For the 5 agents we can observe that the initial cost for all of the agents is larger than the reached consensus at the last iteration. This shows that all of the agents initially computed the cost to be larger than what they agreed on in the end. For the other cost evolutions we have seen thus far the agents agree on a consensus that is somewhat in the middle of the different initial computations. We can observe here, as in the last scenario, that the variation is smaller for the 5 agent scenario than for the 10 agent scenario, but not by much. We can also observe that there is a higher variation from the highest to the lowest initial computation than what is was with only 25 images.

When it comes to the computed accuracy for the agents these values were obtained in the same way as described in the last scenario. The accuracy for the 5 agent scenario can be seen in Figure 1.7, while for 10 agents we can look to Figure 1.8. We can observe here that the computed accuracy is better for 10 agents than for 5 agents. In the scenario with 10 agents the network no longer manages to capture a single digit with the label '1', while it manages 100% of the '0' labels. Therefore the accuracy in total is better for this scenario than for the one with 5 agents, as there are a higher percentage of digits with the label '0' in the training set. We can see that for the scenario with 5 agents the algorithm is able to correctly identify the majority of the '0' labels, and also some of the '1' labels. Therefore you can argue that this version of the algorithm actually performs better than the one with 10 agents, even though the overall accuracy is a bit lower for the 5 agent version.

1.4.3 Scenario 3: 10 agents, 25 images, 15 and 30 iterations

The next scenario we wanted to look at in this project is having 25 images for each of our 10 agents, and then seeing how the amount of iterations will influence the performance. In this case we have decided to look at 15 and 30 iterations.

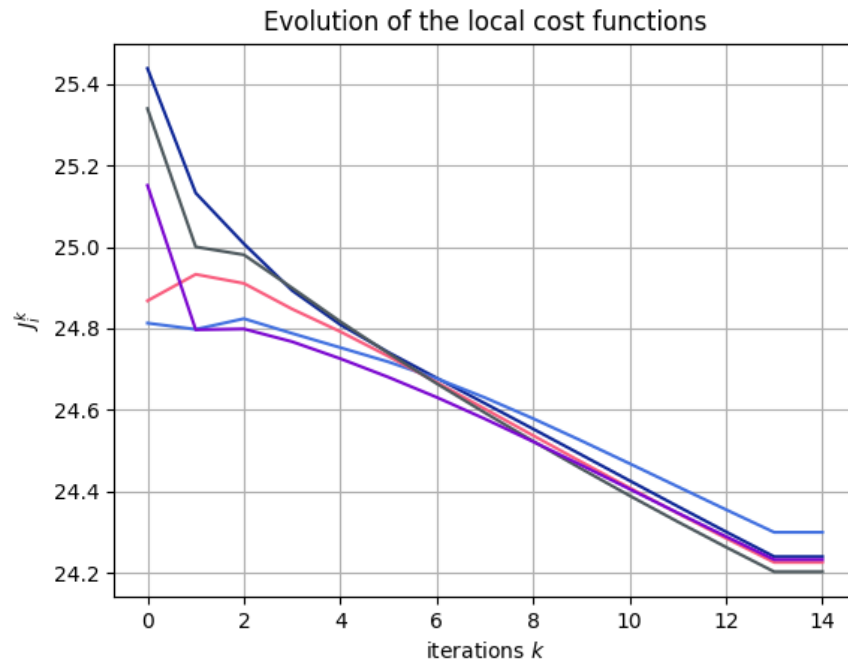


Figure 1.5: The evolution of the cost for 5 agents with 100 images and 15 iterations

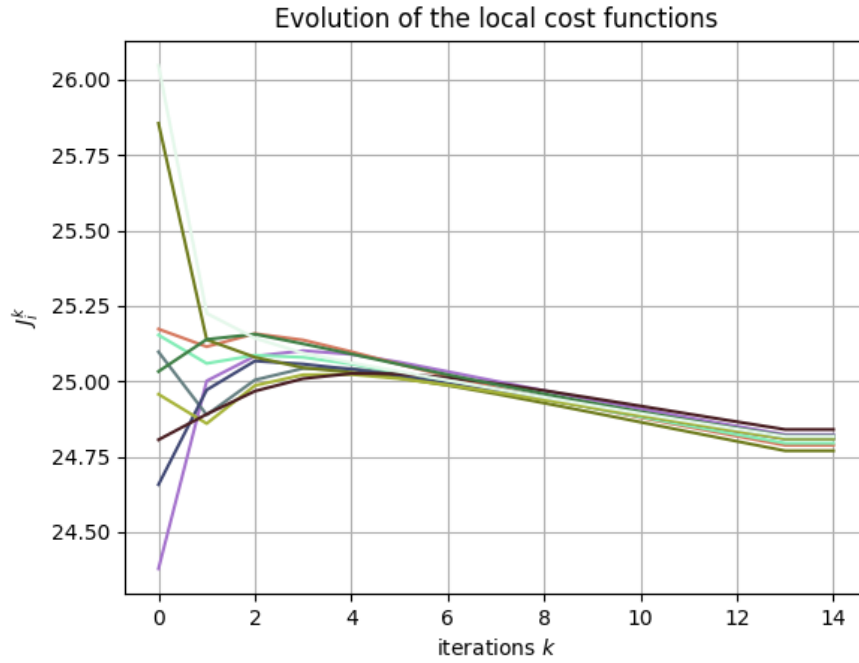


Figure 1.6: The evolution of the cost for 10 agents with 100 images and 15 iterations.

```

1 82
1 83
AGENT ACCURACY
Number of test samples per agent: 100
---- Correct '1' ---- Correct '0' ----- TOTAL -----
Agent [1]: 0.071 - 0.942 - 0.820
Agent [2]: 0.071 - 0.919 - 0.800
Agent [3]: 0.071 - 0.942 - 0.820
Agent [4]: 0.071 - 0.953 - 0.830
Agent [5]: 0.071 - 0.965 - 0.840

```

Figure 1.7: The computed accuracy for 5 agents with 100 images.


```

0 86
0 86
AGENT ACCURACY
Number of test samples per agent: 100
---- Correct '1' ---- Correct '0' ----- TOTAL -----
Agent [1]: 0.000 - 1.000 - 0.860
Agent [2]: 0.000 - 1.000 - 0.860
Agent [3]: 0.000 - 1.000 - 0.860
Agent [4]: 0.000 - 1.000 - 0.860
Agent [5]: 0.000 - 1.000 - 0.860
Agent [6]: 0.000 - 1.000 - 0.860
Agent [7]: 0.000 - 1.000 - 0.860
Agent [8]: 0.000 - 1.000 - 0.860
Agent [9]: 0.000 - 1.000 - 0.860
Agent [10]: 0.000 - 1.000 - 0.860

```

Figure 1.8: The computed accuracy for 10 agents with 100 images.

Firstly we can observe how the evolution of the cost changes with the amount of iterations of the algorithm that we execute for each agent. We can see the scenario with 15 iterations in the Figure 1.9, while we have the 30 iterations version in Figure 1.10. We see, as expected, that for these two versions the variation is about the same for the initial cost calculation of all the agents. This is expected as it is the same amount of agents and images, and this initial calculation is not affected by the amount of iterations of the algorithm. We observe that the agents reach consensus after about 7 iterations for both of the two scenarios. What is interesting about these two plots, is that we can observe that after reaching the consensus, there is not much of an improvement when it comes to the agreement as the iterations move on. We can observe that all the agents do not meet in exactly the same value, but rather obtain a value with deviation of about ± 0.01 . This shows that there is some error margin in our algorithm, and this does not decrease with the amount of iterations that we run.

When it comes to the accuracy of the agents, we would expect that the accuracy is the same for both of these versions or better for the scenario of 30 iterations. This is because the accuracy calculation is not directly affected by the amount of iterations for the algorithm, but the agents should be better trained to recognise the digits since the algorithm has had more iterations. We can see the computed accuracy for 15 iterations in Figure 1.11, and the accuracy for 30 iterations in Figure 1.12. We see here that the accuracy is higher for 30 iterations, based on the fact that the agents are now better at correctly identifying the '0' labels, and thus receive a perfect score on this. There is a much higher variation between the different agents and their accuracy for 15 iterations. We can also compare this 15

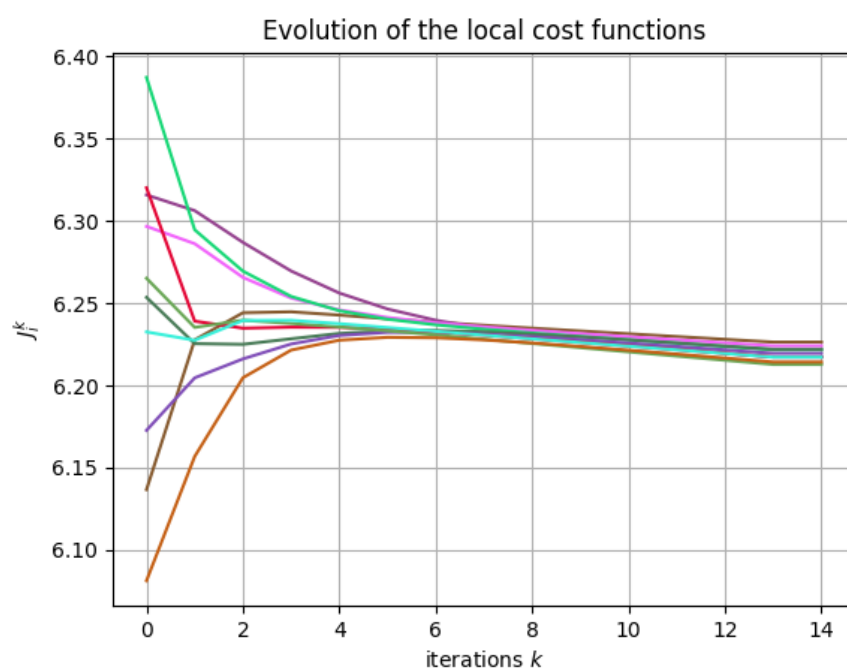


Figure 1.9: The evolution of the cost for 10 agents with 25 images and 15 iterations

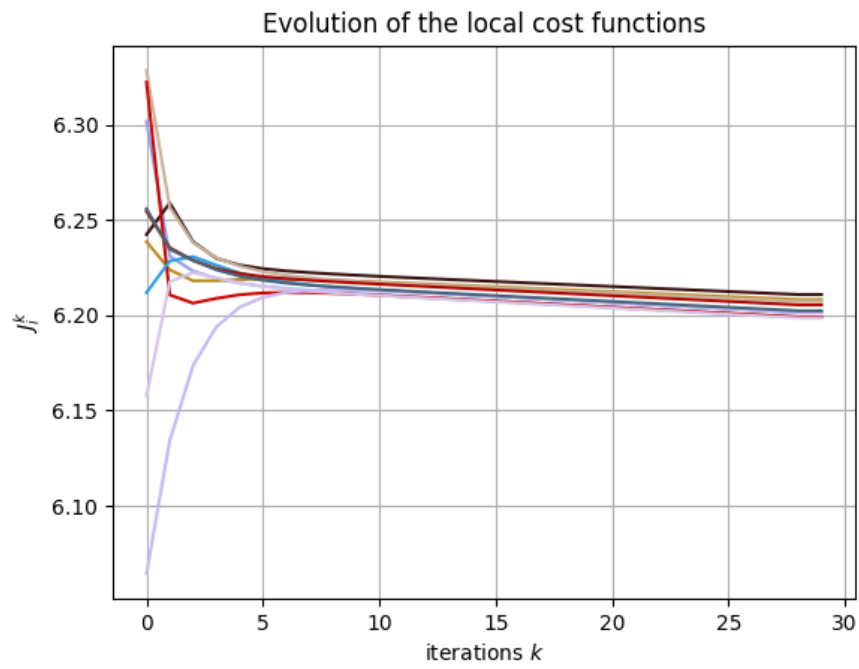
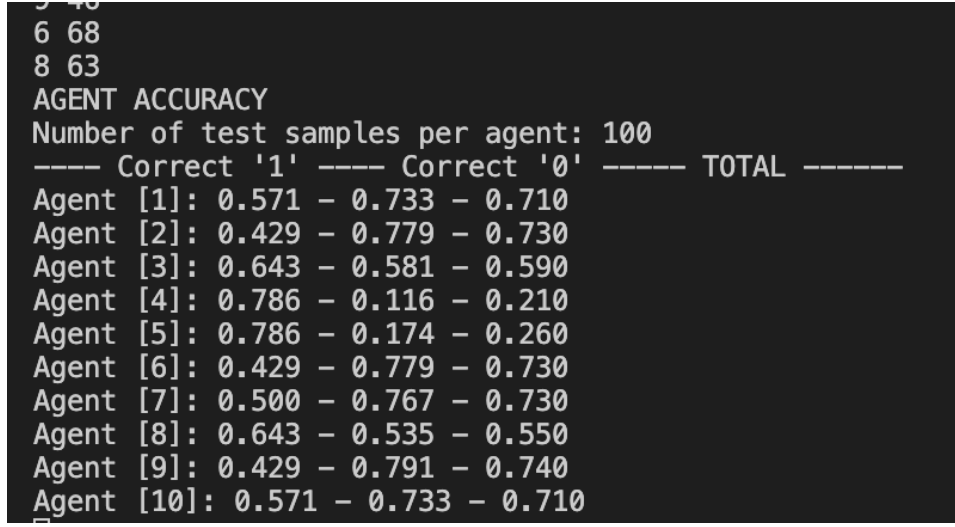


Figure 1.10: The evolution of the cost for 10 agents with 25 images and 30 iterations.



```

5 48
6 68
8 63
AGENT ACCURACY
Number of test samples per agent: 100
---- Correct '1' ---- Correct '0' ----- TOTAL -----
Agent [1]: 0.571 - 0.733 - 0.710
Agent [2]: 0.429 - 0.779 - 0.730
Agent [3]: 0.643 - 0.581 - 0.590
Agent [4]: 0.786 - 0.116 - 0.210
Agent [5]: 0.786 - 0.174 - 0.260
Agent [6]: 0.429 - 0.779 - 0.730
Agent [7]: 0.500 - 0.767 - 0.730
Agent [8]: 0.643 - 0.535 - 0.550
Agent [9]: 0.429 - 0.791 - 0.740
Agent [10]: 0.571 - 0.733 - 0.710

```

Figure 1.11: The computed accuracy for 10 agents with 15 iterations.

iterations with the one from Figure 1.4, which is also for 10 agents with 25 images. Here we see clearly how much the randomization of the splitting of the training set influences the result. In Figure 1.11 the agents are much better at recognising the label '1' than they were in the previous scenario, which can stem from which samples from the training set that they have used in the algorithm.

1.4.4 Scenario 4: 10 agents, 15 iterations, 25 and 500 images

The last scenario that we wish to explore in this project is the situation of having 10 agents and running the algorithm for 15 iterations, and then changing the amount of pictures for each agent. To really see the effect that the amount of pictures have on the accuracy and cost, we here want to compare giving the agents 25 images and 500 images for their training.

We first want to look at the cost functions for this scenario. We have seen the cost function for 10 agents with 25 images and 15 iterations per agent earlier, so this is nothing new. We here want to observe how much the cost function is altered by adding 475 more pictures per agent. The Figure for 25 images and 10 agents is given in Figure 1.11, while the cost function of the version with 500 images is given in Figure 1.13. We can observe that for 500 images, the variation is much larger for the initial calculations of the cost than it is with only 25 images. This makes sense as it would be harder to estimate the cost when it is that many pictures to run through. We also see that here as well the initial estimates are all larger than the computed optimal cost in the end, which can also be because it is harder to estimate

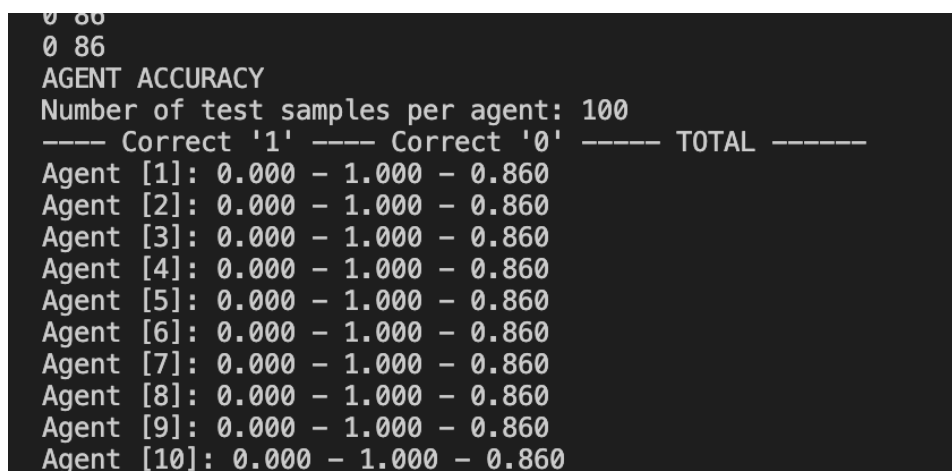


Figure 1.12: The computed accuracy for 10 agents with 30 iterations.

the correct cost. This is also reflected in Figure 1.5 seen earlier, where the larger amount of images for 5 agents result in the starting computations being higher than the consensus value.

We can also observe that the consensus has a higher deviation for 500 images than for 25 images. Another interesting observation is that for both of the plots, it seems as though the agents are closer to each other in the value of the cost at 8 iterations than they are at 15 iterations. It seems as though in the calibration of the consensus the agents actually have a more similar cost value than they do after agreeing.

Lastly we can look at the computed accuracy for the different scenarios. We can observe the accuracy with 25 images in Figure 1.11, while the accuracy with 500 images is shown in Figure 1.14. As expected the accuracy is better for the 500 images variation, as this is able to correctly recognise all of the '0' labels. We can see a clear correlation with the amount of images the agents use in their training and the amount of label '0' they are able to recognise.

This brings us to the end of the results obtained from this task of our project. We have seen some clear correlations between the different parameters and also observed some unexpected behavior. The conclusions and comments on our observations will be discussed further in the Conclusions chapter.

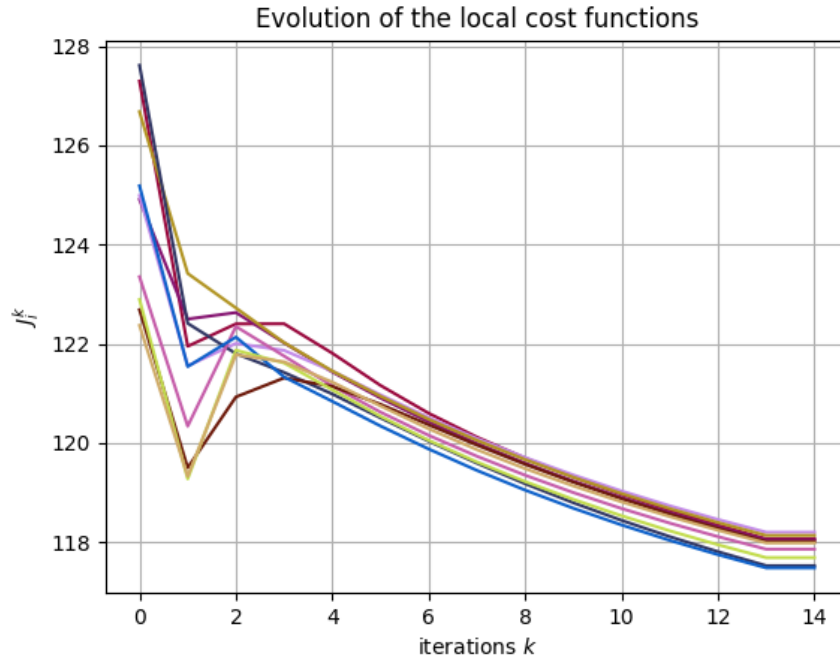


Figure 1.13: The evolution of the cost for 5 agents with 500 images and 15 iterations.

```

0 86
0 86
AGENT ACCURACY
Number of test samples per agent: 100
---- Correct '1' ---- Correct '0' ----- TOTAL -----
Agent [1]: 0.000 - 1.000 - 0.860
Agent [2]: 0.000 - 1.000 - 0.860
Agent [3]: 0.000 - 1.000 - 0.860
Agent [4]: 0.000 - 1.000 - 0.860
Agent [5]: 0.000 - 1.000 - 0.860
Agent [6]: 0.000 - 1.000 - 0.860
Agent [7]: 0.000 - 1.000 - 0.860
Agent [8]: 0.000 - 1.000 - 0.860
Agent [9]: 0.000 - 1.000 - 0.860
Agent [10]: 0.000 - 1.000 - 0.860

```

Figure 1.14: The computed accuracy for 10 agents with 500 images.

Chapter 2

Formation Control

Formation control is a coordinated control for a fleet of robots to follow a predefined trajectory while maintaining a desired spatial pattern [1]. In this task we have a network with N robotic agents where the state of the agents include position and velocity. The agents will reach a formation specified in the formation patterns while using the formation law specified in [3]. The robots are divided into leaders and followers, where the first N_l agents will be the leaders and the remaining N_f agents will be the followers.

2.1 Architecture of the task

This task is implemented in ROS2 and written in Python. We create our own packet called formation control. We have a launch file with values that get sent to our iterative agent_i. The agent_i.py contains the logic for formation update and the sending and receiving of messages in ROS2. We also have the visualizer.py file which contains the required code for showing simulations in rviz. The folder called resource contain the config for rviz. The package.xml contains all the extra packet dependencies we need for the project, namely rclpy, std_msgs, ros2launch, geometry_msgs and visualization_msgs. The setup.py will setup the packet for launch and rviz, and we also specify the entry points in this file which will be the agents and visualizers.

We build the project with "colcon build --symlink-install" in the terminal. Then we launch the ros2 packet with "ros2 launch formation_control formation.launch.py", where formation_control is the name of our packet and formation is the name of our launch file.

2.2 Discretization of model

The x_i state of the agent consists of P_i and V_i , where P_i consists of P_{xi} and P_{yi} , and V_i consists of V_{xi} and V_{yi} . In the continuous-time model the

acceleration input $u_i(t)$ is defined as the derivative of the velocity which again is the derivative of the position. Each followers is modeled as a double-integrator like this illustration:

$$\dot{p}_i(t) = v_i(t), \dot{v}_i(t) = u_i(t) \quad (2.1)$$

Here "i" is part of the index set of the followers. The leaders are stationary and will have a constant velocity which is set to be zero. The bearing-based control law for follower i is defined as:

$$u_i = \sum_{j \in N_i} P g_{ij} \star [k_p(p_i(t) - p_j(t)) + k_v(v_i(t) - v_j(t))] \quad (2.2)$$

Where k_p and k_v is positive constant gain, that we set to be 1 and $P g_{ij} \star$ is:

$$P g_{ij} \star = I_d - g_{ij} \star (g_{ij} \star)^T \quad (2.3)$$

$P g_{ij} \star$ is a constant orthogonal projection matrix associated to the desired bearing unit vector of agent j relative to agent i. The desired bearing unit vector $g_{ij} \star$ is related to the position of the agents in the desired formation, and given by:

$$P g_{ij} \star = \frac{p_j(t) - p_i}{\|p_i(t) - p_j\|} \quad (2.4)$$

The discretization of the continuous time model described in [3] where the agent state x_i consists of position and velocity in two-dimensions, is done by applying the Forward Euler Method:

$$\begin{aligned} p_{xi}(t+1) &= p_{xi}(t) + \alpha v_{xi}(t) \\ p_{yi}(t+1) &= p_{yi}(t) + \alpha v_{yi}(t) \\ v_{xi}(t+1) &= p_{xi}(t) + \alpha u_{xi}(t) \\ v_{yi}(t+1) &= p_{yi}(t) + \alpha u_{yi}(t) \end{aligned} \quad (2.5)$$

The α symbol is the discretization step size, which we set to communication time divided by 10. The step size should be really small. We modified the function `formation_update()` where we give the agent a state with both the position and the velocity, in the two-dimensional space, with the discrete time model we computed. The computed `formation_update()` is illustrated in Figure 2.1.

2.3 ROS2 - Robot Operating Systems 2

To perform this formation control we use ROS2. To use it we had to install VMware with Ubuntu version 20.04 and then install ROS2 on that Ubuntu


```

def formation_update(dt, x_i, neigh, data, kp, kv, P_, agent_id, type):
    # dt = discretization step
    # x_i = state pf agent i
    # neigh = list of neighbors
    # data = state of neighbors
    # kv = coefficient for formation control law
    # kv = coefficient for formation control law
    # I_D = Identity matrix

    I_D = np.identity(2) #Two because dimension is 2. 2x2 nodes which is 4.
    xdot_i = np.zeros(4) #X_i shape is 4.
    P = np.zeros((2,2))
    u_ij = np.zeros(2) #Vector that contains the acceleration
    g_ij_vec = np.zeros(2)
    diff = np.zeros(2)

    p_i = np.array(x_i[0:2]) #Position
    v_i = np.array(x_i[2:4]) #Velocity
    print(neigh)

    for j in neigh:
        print(data[j])
        x_j = np.array(data[j].pop(0)[1:]) #Pop the first element, and take the rest. The first is just the time.
        print(x_j)
        p_j = np.array(x_j[0:2]) #to np array
        v_j = np.array(x_j[2:4]) #to np array

        g_ij_vec[0] = P_[j,0] - P_[agent_id,0]
        g_ij_vec[1] = P_[j,1] - P_[agent_id,1]
        g_ij_norm = np.linalg.norm(g_ij_vec)
        g_ij = np.array([g_ij_vec] / g_ij_norm

        P = I_D - g_ij.T@g_ij #Left associate
        diff = [kp*(p_i[0] - p_j[0]) + kv*(v_i[0]-v_j[0]),
                kp*(p_i[1] - p_j[1]) + kv*(v_i[1]-v_j[1])]
        u_ij += -np.matmul(P,diff) #Format [x_p,x_p,y,v_x,v_y]. U is acceleration

    #For leaders, this u_ij must be zero
    if type == 'leader': #The first two are leaders
        u_ij[0] = 0
        u_ij[1] = 0

    #Forward Euler. The control law will be applied to the agent if it is a follower.
    xdot_i[0] = p_i[0] + dt*v_i[0]
    xdot_i[1] = p_i[1] + dt*v_i[1]
    xdot_i[2] = v_i[0] + dt*u_ij[0]
    xdot_i[3] = v_i[1] + dt*u_ij[1]

    return xdot_i

```

Figure 2.1: Formation update function

machine. For the virtual simulations we used the software Rviz. We created a packet that deals with the described formation control problem. Then we generated a set of simulations with different numbers of robots, and made the robots draw, one at a time, letters of a chosen word. The word the robots will draw is chosen to be "ciao". This is achieved by the code in the launch file and in agent_i file. The launch file will pass the following parameters to the agent_i:

- agent_id
- max iterations
- communication time
- x initialization values
- neighbours
- kp
- kv
- desired positions
- agent type

The max iterations value is set according to the formation patterns. The more nodes, the higher the iterations is needed to reach consensus. It will be in the range 3000 to 60000. Usually the agents will converge before 60000, but it is to be sure of convergence. The x initialization values are random values generated by np.random, so each follower will be distributed randomly in space at the start of the algorithm.

The adjacency matrix is a complete graph, where all the nodes are connected and there will be no self-loops. It is important that each follower communicates with at least one leader. The positive gain values, kp and kv, are set to be 1. They will typically be 0.5 or 1, something small.

The desired position are set according to the formation pattern. It is defined as an array where the rows represent the agents and the columns are the desired positions. We do not define any desired velocity since the point is to let the control laws decide the velocities so that all the agents reach consensus.

The leaders will start with their initial position as their desired position. The agent type is either a leader or a follower. If the agent is a leader, the formation update will consider $u(t) = 0$ so the formation update will not

happen and the leaders stays stationary in space. For a follower the formation update will happen so u will be updated accordingly.

The agent.i file will have the formation_update() and the Class Agent(Node) with the following functions:

- listener_callback
- timer_callback

The formation_update() function has the logic for the control laws and the discrete-time system we made. The timer callback function will be in charge of the formation update of x_i and deals with the iterations. It will be updated according to the communication time variable set in the launch file. We tested with different communication time, ranging from 0.1 to 0.005. With four agents, 0.005 as communication time works good.

The listener callback will add the received data to a dictionary which has a list of received messages from each neighbour j (a queue). When we create a subscription, we need to define a callback specified for each of the neighbors. The agent will create a publisher, a subscriber and a timer.

The "Consensus ROS 2 package" from Virtuale was the starting point for our package, but we also used the Formation package, and the Rviz package from Virtuale.

2.4 Results

The results include a section for formation patterns for different number of robots, and a section where robots draw letters of the chosen word "CIAO". The formation pattern simulations are handled with Rviz.

2.4.1 Formation patterns with different number of robots

The set of formation patters with different numbers of robots we developed included:

- A square with 4 agents, 2 leaders
- A hexagon with 6 agents, 2 leaders
- A octagon with 8 agents, 3 leaders

The square, the hexagon and the octagon were successfully formed by our agents. The unformed square and the formed square when the agents reach consensus is illustrated in Figure 2.4. We can see that the two leaders are

positioned in the left corners of the square. The terminals for the four agents is showed in Figure 2.2, and here we used 6000 iterations. The positions of the leaders are not changing, which are agent_0 and agent_1. Agent_2 is trying to reach position [1,1] and agent_3 is trying to reach position [0,1]. The formation is achieved so the velocities is around 0. Figure 2.3 shows the different positions with respect to iterations. The horizontal lines are the leaders x and y positions. Agent zero have both x and y position at 0, while agent 1 has x at 1 and y at 0. In the end all agents converge to having x and y positions either 1 or 0, which is correct.

agent_2		agent_3	
Iter = 59981	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59981	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59982	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59982	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59983	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59983	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59984	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59984	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59985	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59985	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59986	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59986	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59987	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59987	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59988	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59988	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59989	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59989	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59990	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59990	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59991	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59991	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59992	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59992	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59993	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59993	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59994	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59994	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59995	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59995	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59996	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59996	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59997	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59997	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59998	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59998	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 59999	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 59999	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 60000	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 60000	Value = [0.0001, 0.9998, -0.0001, 0.0002]
Iter = 60001	Value = [0.9999, 0.9998, 0.0001, 0.0002]	Iter = 60001	Value = [0.0001, 0.9998, -0.0001, 0.0002]
MAXITERS reached		MAXITERS reached	
█		█	
agent_1		agent_0	
Iter = 59981	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59981	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59982	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59982	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59983	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59983	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59984	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59984	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59985	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59985	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59986	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59986	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59987	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59987	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59988	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59988	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59989	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59989	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59990	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59990	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59991	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59991	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59992	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59992	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59993	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59993	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59994	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59994	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59995	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59995	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59996	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59996	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59997	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59997	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59998	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59998	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 59999	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 59999	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 60000	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 60000	Value = [0.0, 0.0, 0.0, 0.0]
Iter = 60001	Value = [1.0, 0.0, 0.0, 0.0]	Iter = 60001	Value = [0.0, 0.0, 0.0, 0.0]
MAXITERS reached		MAXITERS reached	
█		█	

Figure 2.2: Convergence for square formation

The hexagon is visualized in Figure 2.5, with an image before iterations are done and an image with the hexagon after iterations are done. The two leaders are positioned farthest to the left in the pictures.

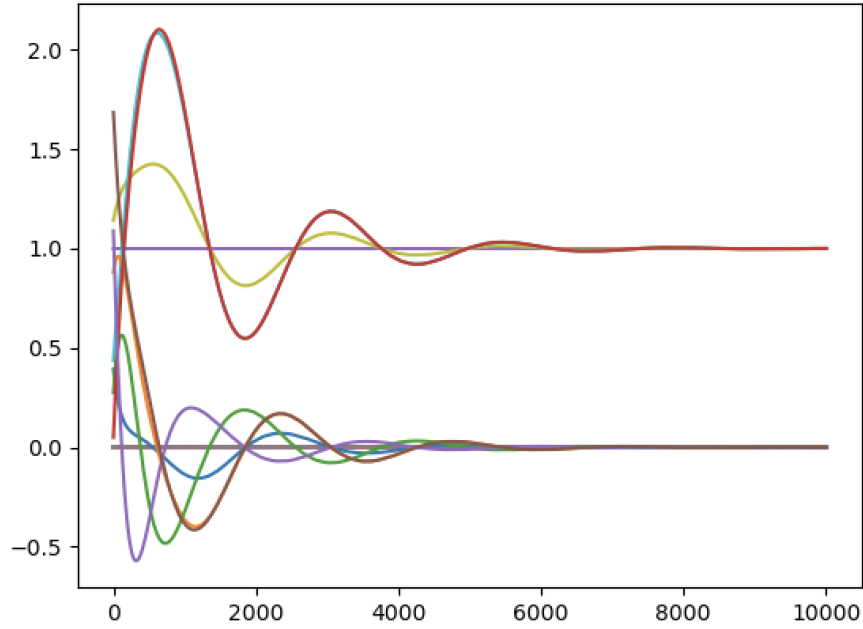


Figure 2.3: Positions of agents forming a square with respect to iterations.

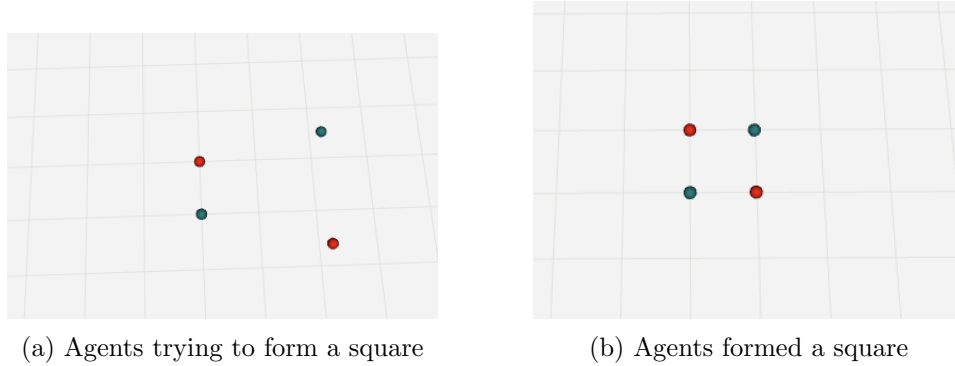


Figure 2.4: A square formation

The virtual machine we used was able to run with six agents one day, but the next day the same code would not work for more than four agents and the same communication time. The problem was that the agents did not receive all the packets and therefore never got synced and did not enter the formation update. All neighbours have not been received for every agent. All messages at time $t-1$ has not arrived at each agent and they never do. We tried modifying the communication time and erasing unneeded files on both the virtual machine and the local machine where the virtual machine was

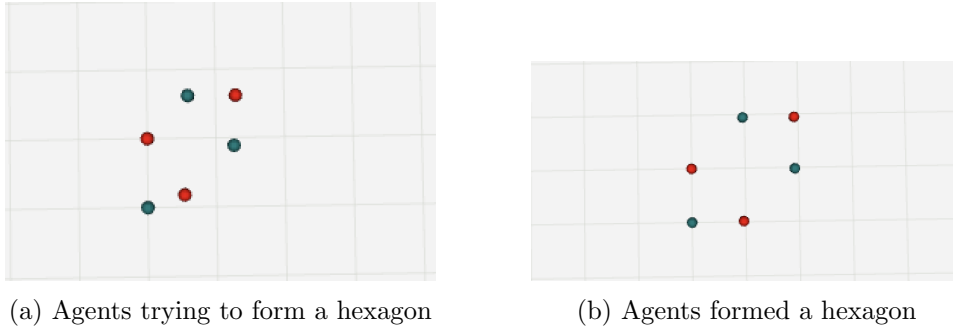


Figure 2.5: A hexagon formation

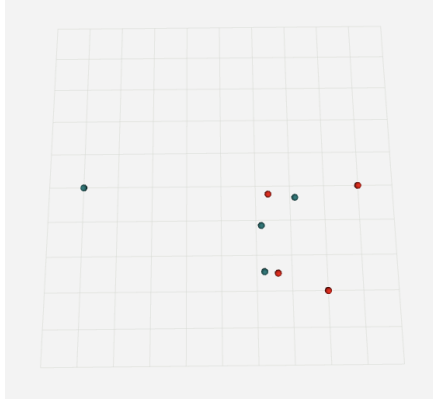
running. The same code we were running were tested on another computer and there it worked fine for six agents. The error could therefore be related to the virtual machine. We made it work by setting the communication time up to 2. This will work, it will only take a lot longer time per iteration as compared to 0.005 as communication time for four nodes. The step size is coherent of the communication time, so when the communication time gets bigger, the step size will get bigger as well.

To make the octagon we still need to choose a high communication time in order for the virtual machine to handle all the agents. Figure 2.6 shows agents forming an octagon in rviz. We can see that with higher communication time, the agents will reach consensus in less iterations, as illustrated in figure 2.7. The horizontal lines are the positions of the leaders that are stationary. For the octagon there were three leaders, with position $[3,3]$, $[0, -4]$ and $[0, 4]$.

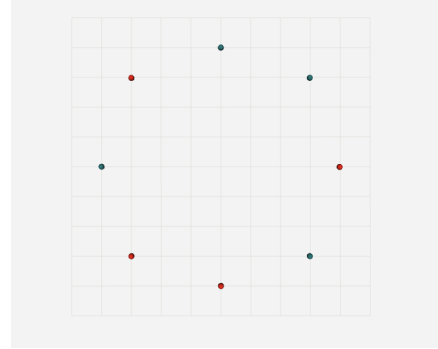
2.4.2 Formation patterns with letters of a chosen word

The second formation patterns include letters of a chosen word. Our agents were successfully able to draw letters, one at a time, forming a word. We set the iterations to between 10 000 to 30 000, depending on number of agents, to be sure the agents reach consensus.

There are two developed formation patterns, one with four agents and one with six. We developed a formation pattern so the robots draw the word "CIAO", illustrated with four agents in Figure 2.8 and with six agents in Figure 2.9. For the letter I, the picture with four nodes were used in both because the convergence did not work and three agents in the same line was the best result. This part consists of four formations, since there are four



(a) Agents formed an octagon formation



(b) Agents formed an octagon formation

Figure 2.6: An octagon formation

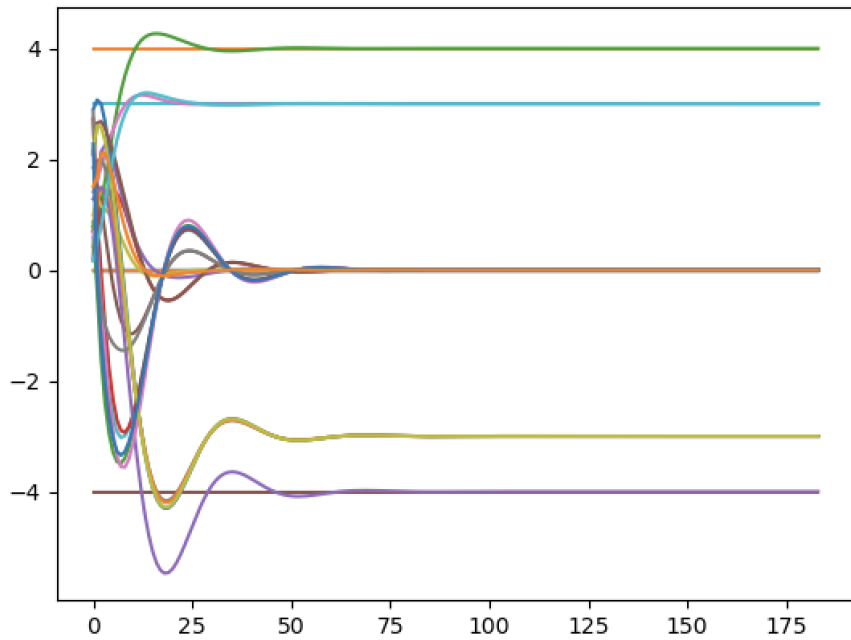
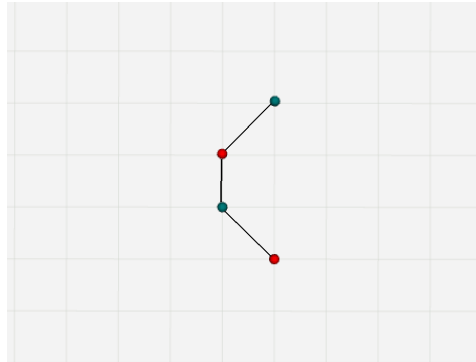


Figure 2.7: Positions of agents forming an octagon with respect to iterations.

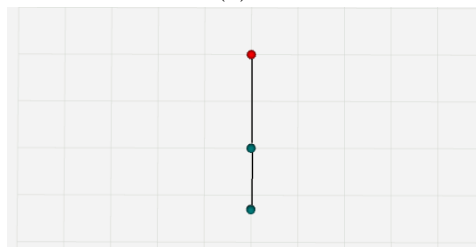
letters in ciao. To make these letters, we take a look at the two-dimensional coordinate system and mark the positions where it would be natural to position agents in order to get a letter.

For the letter I, all the agents did not reach consensus and some converged towards infinity when using both four and six agents. We did several tests and all the agents did not reach consensus for the letter I. This was the only case where one of our agents did not act as expected.

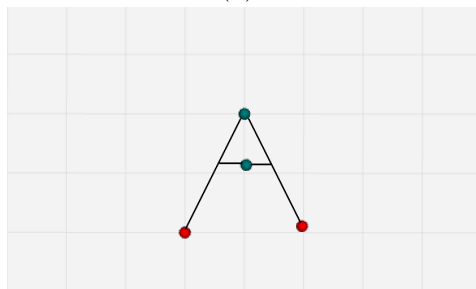
We developed a specific formation pattern for each letter. In the launch file we decide what formation the agents are going to draw. We decide a letter, one after one.



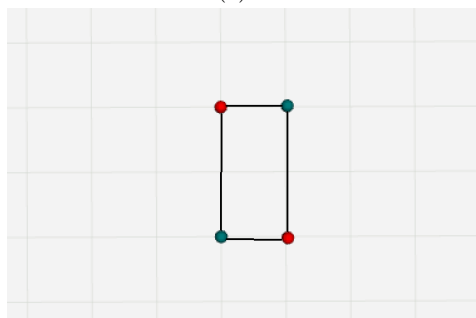
(a) C



(b) I

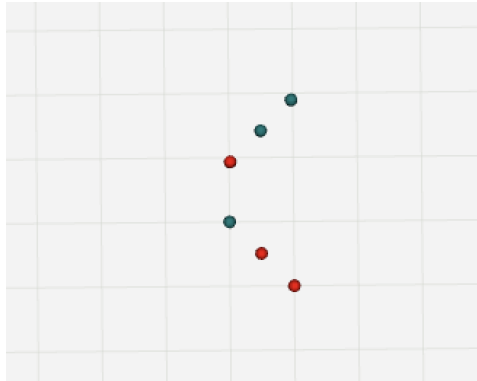


(c) A

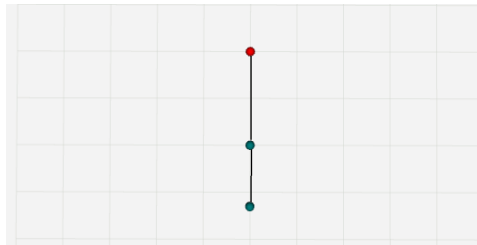


(d) O

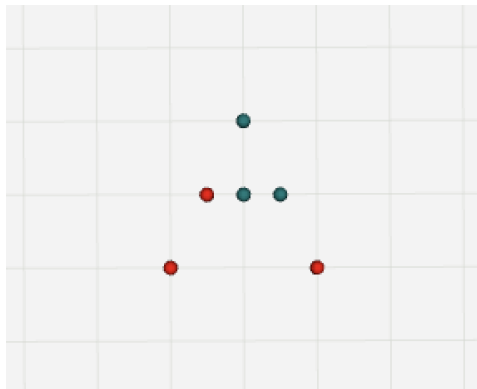
Figure 2.8: CIAO four agents



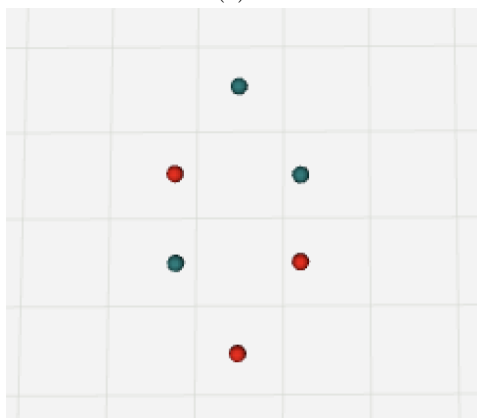
(a) C



(b) I



(c) A



(d) O

Figure 2.9: CIAO six agents

Conclusions

2.5 Distributed Classification

For the Distributed Classification task with the training of a neural network we looked at the cost function and accuracy of the different agents in the network. We had some different variations of combinations, in order to see how the different parameters influenced the obtained results. We therefore changed the number of agents in the network, the number of images given to each agent for their training, as well as the number of iterations of the algorithm. Through the different scenarios presented in the results part of the discussion of this task we made some clear observations. These will be summarized here.

First for evolution of the cost function for the different agents. We observed that the variation of the initial computation of cost was larger when the amount of pictures was larger. In addition we observed that if the amount of pictures for each agent, in correlation to the amount of agents in the network, became very large, the initial calculations were higher than the agreed consensus after all the iterations. In the other cases the initial calculations were both higher and lower than the optimal value, which was calculated to somewhat of an average. In general the agents seem to reach consensus after about 8 iterations, regardless of the amount of pictures given to each agent. The only exception is for the graphs with 5 agents, where they seem to need a bit more than 15 iterations to reach a proper common cost value. We also see that the computed cost value for each agent is not entirely similar at the end of the iterations, there is usually an deviation of about ± 0.01 .

Then we have the accuracy of the agents, when it comes to recognising the labels '0' and '1'. Here we can see very clearly that the amount of images and iterations will influence the agents in their ability to recognise label '0'. It is only when the amount of iterations and images are quite low that the agents will recognise the label '1' at all. A reason for this can be that the only digit that is labeled with '1' is digit 4. In contrast all the other digits, [0, 1, 2, 3, 5, 6, 7, 8, 9] is labeled with '0'. This means that as the amount of images and/or iterations improve, the agents can get a wider and

wider perception of what accounts to label '0'. After a while even digit 4 can be included in the label '0' category, because the agents have such a wide understanding of what the label '0' includes. There are also a lot more of label '0' in the training set than label '4', statistically speaking only about 1/10 of the 60.000 digits are 4. This means that when we split the training set randomly and only give a part of the training set to each agent (both by splitting and by choosing the amount of images from their subset that they should use), the chances are quite big that the agent will have very few label '4' to practice on. Therefore it is logical that the amount of label '4' recognised by the agents decrease with the amount of pictures or run troughs of the pictures that the agents perform.

Briefly summarized we can see that the gradient tracking algorithm is a good algorithm for reaching consensus for the agents. We also observe that for the classification problem at hand the accuracy was higher for an increased amount of images per agent, but this also led to fewer digits with the label '4' being recognized by the agents. We have seen that the cost is quite low for each agent, and that every agent in the network will have the same cost of calculations when the agents have reached consensus.

2.6 Formation Control

For the task 2 of our project, we can conclude that the implemented control law was effective at achieving formation control among the agents in the network. We managed to get the formation law to work for different numbers of agents in the network. The nodes in the network are split into leaders and followers, where the followers are the ones following the control law. We developed the case where leaders are stationary, and the followers will communicate with the leaders and neighbours and end up in their desired position. This means that the leaders were put in their desired position from the start, and did not move during run time. The next step for this project would be to see how the agents would behave if the leaders also were initialized at a different position than the desired one, and then moved with constant velocity to their desired position.

For the formation patterns with letters we can observe that the agents are able to reach consensus and achieve the desired pattern within the set amount of iterations each time. This means that the control law is robust, as it caters to different types of formations without fail. We also see that these letters are possible to make with different amounts of agents, as long as the desired positions for the agents are defined in advance. The only letter that we had trouble with for both the version with four agents and the version with six agents is the letter I. This could be because the de-

sired formation is a straight line, and it might be harder for the agents to communicate properly with their neighbours in this scenario. The reason for this could be that the agents share coordinates in a large way, and thus have trouble with understanding where they are positioned in relation to each other. Nevertheless this is surprising behavior, as the other formations are achieved without any difficulty.

We can also observe from Figure 2.2 and Figure 2.7 that the desired position of the agents is reached before we have 100 iterations in the case of the octagon, and before 6000 iterations for the square. The reason for this is because the discretization step is bigger for the octagon. The discretization step is dependent on the communication time, which is bigger for the formations with more than four nodes in order to make the communication work. If we would use the same step size as for the network with four agents, it will take a really long time to reach consensus. We are using a full matrix as the adjacency matrix, so the agents have a high degree of communication as the number of agents increase, and thus they could reach consensus quicker.

In conclusion we can say that the implemented consensus algorithm is quick and effective. The desired formation is reached in the vast majority of the cases tried in this project. There is little change in performance with changes in the number of agents or the pattern the agents should follow, which shows that this is a good all-round formation update law.

Bibliography

- [1] IGI Global. What is formation control.
- [2] Analytics Vidhya. Mnist dataset prediction using keras!
- [3] S. Zhao and D. Zelazo. Translational and scaling formation maneuver control via a bearing-based approach.