UNIVERSITAT DE BARCELONA

FUNDAMENTAL PRINCIPLES OF DATA SCIENCE MASTER'S THESIS

Simulation of physical systems with variants of the Interaction Network

Author:
Alfons CÓRDOBA

Supervisor:
Albert DÍAZ & Oriol PUJOL

A thesis submitted in partial fulfillment of the requirements for the degree of MSc in Fundamental Principles of Data Science

in the

Facultat de Matemàtiques i Informàtica

January 18, 2021

UNIVERSITAT DE BARCELONA

Abstract

Facultat de Matemàtiques i Informàtica

MSc

Simulation of physical systems with variants of the Interaction Network

by Alfons CÓRDOBA

The aim of this thesis is to primarily learn to predict trajectories of physical systems by using Machine Learning. We have used the Interaction Network as a base model and introduced tweaks to its structure in the framework of Graph Networks in order to deal with systems without interaction, with force-based interactions and systems governed by the Vicsek model. We have been able to replicate very well systems without interaction and systems governed by a Vicsek model of infinite reach. However the results with force-based systems are mediocre because they need more trainable parameters and training data. The results for the Vicsek model with a finite radius of reach are the worst but we have learned the necessity and the methodology of introducing attention to the base model to deal with this class of problem.

Acknowledgements

I'd like to thank primarily to my family: my mother and my sister.

Secondly, I'd like to thank my supervisors Albert Díaz and Oriol Pujol.

Chapter 1

INTRODUCTION

The aim of this thesis is to study the behavior of system of agents or particles that interact with each other using Machine Learning (ML) techniques from data obtained by simulation. The article *Towards Automated Statistical Physics : Data-driven Modeling of Complex Systems with Deep Learning* (Ha and Jeong, 2020) , on Data Driven Learning of Complex systems has served as a motivation and as a starting point for the research of this project. We will attempt to use both simple physical systems and complex systems as subjects of study. The code is all at https://github.com/alcome1614/ModifiedInteractionNetworks.

In the pursuit of knowledge, humanity has always tried to break down the object of study in smaller and simpler parts. For instance, if one is trying to understand the behaviour of a group of animals, say a honeybee colony, one should first try to distinguish between the physical place and the individuals. Then one should aim to come up with a hierarchy and/or classification of said individuals by observing their distinct behavior, the roles and the interactions with each other. This way one shall find there is a queen and there are workers and drones. Even when there was no evidence of the existence of this smaller constituents, the Greeks (Leucippus and Democritus) in the 5th centure BCE theorized that matter was made of tiny, elemental particles: atoms. This single claim is very powerful and has lead us to a deep understanding of the mechanisms of nature. Not only that, if the human civilisation were to be restarted with all the knowledge erased but one single statement that should arguably be the one that would help them the most. For this reason the base approach developed will revolve around studying the particles by themselves by means of tracking variables that describe their state in the context of the collective.

1.1 Complex Systems

And this approach is similar to what the field of Complex Systems includes in its methodology. It studies systems with different components that interact with each other non-trivially and that can be represented with a graph. This field does not treat or care about the nature of the components and this makes it interdisciplinary. Once the system is mapped into a graph, the field or nature of the components may become unnecessary.

The field of Complex Systems is a relatively new one, although under other names it is something that has already been carried on by humanity for centuries in different disciplines. It was officially born in the 1980s and its history and development are to some extent linked to that of neural networks that were born more or less at the same time.

1.2 Data Driven Learning

The before-mentioned theory of atoms or atomism will also help one introduce another key concept: science has mostly been hypothesis driven. By observing and studying their natural environment: the night sky is composed of infinitely many stars, the sand in the beach is made of an inconceivable number of grains, water that is composed by an uncountable quantity of tiny drops, a town is composed of houses and people, etc. the atomists reached their conclusion. Even though they did not have the means to prove it at that time, it laid the groundwork philosophically speaking for more accurate and elaborated theories centuries later and the consequent experimental evidence of their existence. In the field of sciences, knowledge has been acquired following a concrete methodology that has been canonical since the 18th century: the scientific method. A hypothesis or theory must be developed and evaluated with data descriptive of reality, using statistics to show that the hypothesis is good enough to explain the events of the experiment. It was the development of this methodology and in technology that helped the study the structure of the atom and the discovery of particles like the electron or the neutron in the 20th century.

The key here is that the hypothesis even though is based on the observed data, it is a biased model where the person formulating it has to elaborate it and base it according to their previous knowledge and intuition. A case can be made for the role of luck in intuition in the great scheme of things. But what cannot be denied is the limitations of this methodology despite its more than notable success.

In the kind of problem that occupies us we will distance a bit ourselves from the classical Statistical Physics approach that "*derives observables* -like pressure or temperature- *from the microscopic description of the particles*"[1].

The technological improvement has made the obtention, storage and the processing of huge amounts of data, and specifically experimental data, significantly easier than it was. This has motivated the development of Data Driven Models. These are models that instead of relying purely on hypothesizing the relationship between the variables of study, they let the model find this relations by fitting the observed data. The drawback of this methodology is, as (Montáns et al., 2019) mention, the lack of interpretability of the parameters learned: in layman's terms, most of the times Data Driven Learning is a black box. It can provide accurate results but it does not give an intuitive explanation of the relationship between the variables.

1.3 Deep Learning

Deep Learning (DL) is a subset of Machine Learning (ML) that tries to emulate the learning process of the human brain. It perfectly exemplifies data driven learning. Not only that, but one cannot tell the story of ML without that of neural networks (NNs). Neural networks are relevant to us because they are one of the main tools that are going to be used for learning the systems that will be presented later on.

DL and NNs have been trending in the last years but it has not always been like that. Their origin is the study of the human brain as previously stated and trying to emulate its functioning by Warren McCulloch and Walter Pitts in 1943. Later, in 1957, Frank Rosenblatt built the algorithm for the perceptron. However, in 1969, Marvin Minsky and Seymour Papert wrote a book in which they argued the prohibitive time needed to make Rosenblatt's perceptron work, and them being an algorithm

[1] This appreciation has been extracted from the article:(Ha and Jeong, 2020)

that could not deal with the exclusive-or operation on top of that. The interest and excitement in NNs plummeted as a consequence and gave place to the first winter of neural networks.

This field's interest did not quite come back until more than a decade later when David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams came up with ideas that would later become what today is known as backpropagation that is one of the key parts of NNs algorithm and that has made them computationally more efficient; unfortunately not at that time: computers were still not capable of dealing with NNs in reasonable times.

It was not until the 21st century that the hype about NNs came back with an innovative paper: *A fast learning algorithm for deep belief nets* (Hinton et al., 2016). Since then, the computational power has grown exponentially yearly following Moore's law making them a very useful and practical tool to solve all kinds of problems of ML. Variants based on NNs have proliferated as well like Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs). On top of that other new types have been produced like Graph Networks (GNs) that learn structured data from graphs that has multiple variants like Graph Convolutional Networks (GCNs) or Graph Attention Networks (GATs). Some of these variants will be introduced later on.

1.4 Graph Neural Networks

It is precisely this last type of neural networks, Graph Neural Networks (Scarselli et al., 2009), the ones that we will introduce in order to tackle this problem. Graph Neural Networks were thought to face problems that can be described as graphs: composed of elements that have some sort of relationship between them (not all of them must interact with each other).

Since we are dealing with physical systems and we want to start from the most simplistic approach we will use Interaction Networks (Battaglia et al., 2016) that are one of the first attempts of learning and simulating physical systems (n-body, systems with strings, etc). By understanding their structure in the context of Graph Neural Networks and more concretely using the notation of Graph Networks (Battaglia et al., 2018), we will be present a series of tweaks and modifications to the original Interaction Network.

The objective of this project is to code an Interaction Network and use it to learn to predict the trajectories of differents systems. Firstly, a system without interactions; secondly, a system with force-based interactions and lastly, a system governed by the Vicsek model. Additionally, if the results with the Vicsek model are satisfactory we would like to learn and predict abstract quantities like the radius that determines the neighbourhood of agents in this system. We are not aiming to produce really accurate results but to get an intuition of what works or can work because we work with very limited resources computationally and in terms of time.

Chapter 2

Theoretical Background

In this thesis one will produce the data that later will be used to train, validate and test the different models presented. The data generated corresponds to systems of agents or particles that interact with each other according to different rules. Therefore we will first introduce the theory that governs the systems that we will study and afterwards we will introduce and review the Machine Learning models that we will use to learn from this data.

2.1 Physics

2.1.1 Kinematics

The base of classical physics is the motion of objects. But before that, it is important to set a system of coordinates to be able to describe its position $\mathbf{x}$ in space. A system of coordinates needs an origin and a base of the n-dimensional space ($\mathbb{R}^n$). A base is a set of linear independent vectors that can generate any vector of the space by means of a linear combination.

In the case of $\mathbb{R}^2$ the conventional base is $\{(1,0);(0,1)\}$.

With a spatial system of coordinates we can already describe the state of an immobile object or a collection of them, as long as they do not interact. But to describe motion we need to add another dimension to the n spatial ones: time t.

Motion implies a change of the position. If we take a picture of the system at an instant of time there will be a change in the position from one frame to another. The introduction of the concept of time allows one to define a new variable: velocity . Velocity measures the rate at which position changes.

$$\mathbf{v} = \frac{d\mathbf{x}}{dt} \tag{2.1}$$

If we take equation (2.1) and we isolate the position we can obtain its dependency on the velocity $\mathbf{v}$.

$$\boldsymbol{\Delta}\mathbf{x} = \mathbf{x} - \mathbf{x_0} = \int_{\mathbf{x_0}}^{\mathbf{x}} d\mathbf{x} = \int_{t_0}^{t} \mathbf{v} dt \tag{2.2}$$

$$\mathbf{x} = \mathbf{x_0} + \int_{t_0}^{t} \mathbf{v} dt \tag{2.3}$$

If an object changes position at a constant rate, which means the velocity is constant, its motion is totally determined by

$$\mathbf{x}(t) = \mathbf{x_0} + \boldsymbol{\Delta}\mathbf{x} = \mathbf{x_0} + \mathbf{v}(t - t_0) = \mathbf{x_0} + \mathbf{v}\Delta t \tag{2.4}$$

Similarly, we define acceleration $\mathbf{a}$ as the rate of change of velocity:

$$\mathbf{a} = \frac{d\mathbf{v}}{dt} \tag{2.5}$$

And we can isolate the velocity $\mathbf{v}$ to obtain the analogue of eq. (2.3):

$$\Delta\mathbf{v} = \mathbf{v} - \mathbf{v_0} = \int_{\mathbf{v_0}}^{\mathbf{v}} d\mathbf{v} = \int_{t_0}^{t} \mathbf{a}dt \tag{2.6}$$

$$\mathbf{v} = \mathbf{v_0} + \int_{t_0}^{t} \mathbf{a}dt \tag{2.7}$$

In the case of an object with a constant acceleration we obtain:

$$\mathbf{v}(t) = \mathbf{v_0} + \Delta\mathbf{v} = \mathbf{v_0} + \mathbf{a}(t - t_0) = \mathbf{v_0} + \mathbf{a}\Delta t \tag{2.8}$$

Now if we take eq. (2.5) and combine it with eq. (2.1), we obtain the relationship of the acceleration $\mathbf{a}$ and the position $\mathbf{x}$:

$$\mathbf{a} = \frac{d^2\mathbf{x}}{dt^2} \tag{2.9}$$

We can proceed take the result obtained in eq. (2.3) and the one from eq. (2.7) to obtain the position $\mathbf{x}$ as :

$$\mathbf{x} = \mathbf{x_0} + \int_{t_0}^{t} \left(\mathbf{v_0} + \int_{t_0}^{t} \int_{t_0}^{t} \mathbf{a}dt \right) = \mathbf{x_0} + \mathbf{v_0}(t - t_0) + \int_{t_0}^{t} \int_{t_0}^{t} \mathbf{a}dtdt \tag{2.10}$$

If the acceleration $\mathbf{a}$ is constant the double integral can easily be solved:

$$\mathbf{x} = \mathbf{x_0} + \mathbf{v_0}(t - t_0) + \frac{1}{2}\mathbf{a}(t - t_0)^2 \tag{2.11}$$

2.1.2 Numerical integration of the differentials equations of motion

The basics of kinematics have been introduced but in this project we won't be solving the differential equations of motion analytically but numerically using a computer.

First Order Differential Equations

For the sake of completeness we will introduce first the Runge-Kutta (RK) family of methods to solve ordinary differential equations (ODEs).

Let us consider a first order differential equation:

$$\mathbf{x}' = f(t, \mathbf{x}) \tag{2.12}$$

where we are using the notation:

$$\mathbf{x}' = \frac{d\mathbf{x}}{dt} \tag{2.13}$$

The result for a general m-order Runge-Kutta method is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \sum_{i=1}^{m} b_i k_i \tag{2.14}$$

where $\Delta t = t - t_n$ is the increment of time. And k_i is defined as:

$$k_i = f(t_n + \Delta t c_i, x_n + \Delta t \sum_{j=1}^{m} a_{ij} k_j) \tag{2.15}$$

with a_{ij}, b_i, c_i are coefficients that depend on the quadrature rule. The most known and simple member of the family of RK methods is the Euler method. It is a first order RK method and therefore the matrix a_{ij} is a first order matrix and therefore a number and therefore its value is irrelevant, and the value of $c_1 = 0$ and $b_1 = 1$.

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t f(t_n, x_n) = \mathbf{x}_n + \Delta t \frac{d\mathbf{x}}{dt}(t_n, x_n) \tag{2.16}$$

which is the same as:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \mathbf{v_n} \tag{2.17}$$

As we can see it has the same form of eq. (2.4). The main difference is that in this case the velocity is not constant and we are not accounting for the fact that the velocity may vary within this time frame.

It is also important to notice that initial conditions are needed to do the calculation. Therefore $t_0, \mathbf{x_0}$ need to be provided.

In order to estimate the uncertainty of this method we need to understand first of all that at each step the Euler method is approximating the function with a first order Taylor polynomial. The Taylor expansion of a the function $x = f(t)$ which will be one-dimensional around the point t_0 is

$$x = \sum_{i=0}^{\infty} \frac{1}{i!} f^{(i)}(t_0)(t - t_0)^i = f(t_0) + f'(t_0)(t - t_0) + \mathcal{O}((t - t_0)^2) \tag{2.18}$$

We show it only for a one-dimensional function but a vectorial function would have the same expression for each of its components independently. We can see that if we cut the expansion at the second term then we would be missing the target by a quantity of the order of $(t - t_0)^2$, therefore the square of the step Δt chosen. In order to obtain the global error we need to take into account that the number of steps is inversely proportional to the step length Δt while the uncertainty of each step is proportional to $\Delta t)^2$ and thus the global error is proportional of Δt.

Second Order Differential Equations

Solving second degree differential equations numerically it is a bit more tricky. It is important for physics since most differentials equations are of second order.

Let us consider a second order ordinary differential equation:

$$\mathbf{x}'' = f(t, \mathbf{x}, \mathbf{x}') \tag{2.19}$$

First step, is to define a new variable $\mathbf{v}$ such that:

$$\mathbf{v} = \mathbf{x}' \tag{2.20}$$

Using this change of variable in eq. (2.19), it becomes a first order ODE:

$$\mathbf{v}' = f(t, \mathbf{x}, \mathbf{v}) \tag{2.21}$$

while eq. (2.20) is the other first order ODE to solve numerically. Both of them can be solved using the methods presented in the previous section 2.1.2. For the sake of completeness let us show how to proceed in the case we used the Euler method.

On the one hand, to compute the first step of eq. (2.21) we need the initial conditions $t_0, \mathbf{x_0}, \mathbf{v_0}$. The equation needed to calculate the next step is:

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \Delta t f(t_n, \mathbf{x}_n, \mathbf{v}_n) \tag{2.22}$$

On the other hand, to compute the first step of eq. (2.20) we need the initial condition $\mathbf{x_0}, \mathbf{v_0}$. The equation to calculate the next step is identical to eq. (2.17).

2.1.3 Dynamics

Dynamics studies the interactions between bodies that rule their motion. We have used different interactions types between the agents that compose the different problem systems to test different model architectures. There are two kinds of interactions that are treated: elemental interactions linked to elemental forces and more complex interactions that can not be trivially explained with the elemental forces of nature.

The basic concept and the origin of dynamics are forces. One cannot talk about forces without mentioning the figure of Sir Isaac Newton. He set the foundation for the field of Physics in various publications. One of his most notable publications is *Philosophiæ naturalis principia mathematica* where he establishes what a force is and its characteristics. It contains the infamous Newton's laws which are:

Newton's First Law

An object that has a null net force acting on it has a constant velocity motion which also includes the case in which it is immobile, in which case it remains immobile. Therefore, a null force acting on a body implies its acceleration is null.

Newton's Second Law

The net force an object receives causes a change in the rate of change of momentum of this object.

$$\sum_{i=1}^{N} \mathbf{F}_i = \frac{d\mathbf{p}}{dt} \tag{2.23}$$

where the momentum of an object is defined as:

$$\mathbf{p} = m\mathbf{v} \tag{2.24}$$

or more generally:

$$\mathbf{p} = \int_M \mathbf{v} dm \tag{2.25}$$

with M being the total mass.
In the case of an object with a constant mass eq. (2.23) can be written like below.

$$\sum_{i=1}^{N} \mathbf{F}_i = m\frac{d\mathbf{v}}{dt} = m\mathbf{a} \tag{2.26}$$

The conclusion is that if an object has an acceleration (which is the same as a non-constant velocity) then that object has a non-zero net force acting upon it.

Newton's Third Law

Two objects interacting exert a pair of forces that are equal in magnitude but of different sign to each other such that:

$$\mathbf{F}_{ij} + \mathbf{F}_{ji} = 0 \tag{2.27}$$

Where $\mathbf{F}_{ij}$ is the force exerted on object j by object i and vice versa.

Newton was just considering two-body interactions. For the sake of completeness let us consider a 4-body interaction. In such case Newton's third law expression would be as follows:

$$\mathbf{F}_{i,jkl} + \mathbf{F}_{j,ikl} + \mathbf{F}_{k,ijl} + \mathbf{F}_{l,ijk} = 0 \tag{2.28}$$

2.1.4 Central Forces

Once we have introduced the basics of the forces, we can introduce the kind of forces that will be used later on. Central forces are forces that have the direction of the vector joining the interacting objects $\mathbf{r} = \mathbf{r}_j - \mathbf{r}_i$ and their magnitude only depends on the norm of the $\mathbf{r}$ vector ($r = |\mathbf{r}|$), or in other words, the distance between the two objects.

$$\mathbf{F}_{\text{central}} = F(r)\mathbf{r} \tag{2.29}$$

Due to that, it is easy to prove that such a force follows:

$$\nabla \times \mathbf{F} = \nabla \times F(r)\mathbf{r} = 0 \tag{2.30}$$

Let us introduce Stoke's theorem which states:

$$\oint_C \mathbf{F} \cdot d\mathbf{r} = \iint_S \nabla \times \mathbf{F}\, dS \tag{2.31}$$

With S being a closed surface and C its contour.

And the fact that work is the energy transferred to a body by the aplication of a force $\mathbf{F}$ along a path Γ that starts at point A and ends at point B. Its formal definition is:

$$W_{A\to B} = \int_\Gamma \mathbf{F} \cdot d\mathbf{r} \tag{2.32}$$

We can now apply Stoke's theorem (eq. (2.31)) to the definition of work (eq. (2.32)) because it is essentially a line integral. But the line integral must be along a closed path, therefore Γ starts and ends at point A. Now applying eq. (2.30), the result is zero. which means that the work done by a central force along any closed path is 0. The net transfer of energy is null. Physically this means that the force is conservative because the mechanical energy is conserved and does not depend on the path or time. Two different paths that start at A and end at B are associated to the same variation of energy, no matter how different they are.

2.1.5 Vicsek Model

The Vicsek model (Vicsek et al., 1995) tries to describe systems very different in nature to the ones studied by Mechanics for example. It studies the collective behavior

of agents of active matter. Active matter is constituted by agents that consume energy and that present complex behavior that is studied by the field of Complex Systems. The Vicsek model is one of the simplest (if not the simplest) models of agents that can reach a collective behavior phase.

Given a system of N agents, the equation to calculate the position $\mathbf{r}_i$ of the i-th agent at the $(n+1)$-th time-step is:

$$\mathbf{r}_i^{n+1} = \mathbf{r}_i^n + v\Delta t \mathbf{s}_i^n \tag{2.33}$$

where v is the speed that is constant, Δt the increment of time considered and $\mathbf{s}_i$ is the orientation vector of the i-th agent. The orientation vector is simply a unit vector. In a 2-dimensional space it has the following form:

$$\mathbf{s}_i^n = \begin{pmatrix} \cos\theta_i^n(t) \\ \sin\theta_i^n(t) \end{pmatrix} \tag{2.34}$$

where θ_i^n is the angle of the orientation vector that is calculated at each time-step following:

$$\theta_i^{n+1}(t) = Arg\left(\sum_j A_{ij}^n s_j^n\right) + \eta\xi_i^n \tag{2.35}$$

Here ξ_i^n is a noise variable weighted by the parameter of the model η and A_{ij}^n is the adjacency matrix that is dynamic because it takes values according to neighborhoods that can change since the agents are moving in space.

$$A_{ij}^n = \begin{cases} 1 & \text{if } r_{ij} < R \\ 0 & \text{if } r_{ij} > R \end{cases} \tag{2.36}$$

where r_{ij} is the distance between the i-th and j-th agent, R is a positive parameter of the model that regulates the neighbourhood radius. In a more complex version of the base model, another condition for vicinity (a 1 in the adjacent matrix) is to be inside of the field of view that depends on an angle as parameter. It is worth noting that i and j can take the same value and as a consequence it means that an agent has into account its own direction with the same weight as its neighbours directions. This is the formalism used in (Ginelli, 2016) but in the original formalism (Vicsek et al., 1995) it is more clear that the interaction is an averaging of the angle that dictates the direction of the velocities. So Eq. (2.35) is originally shown with an averaging operator.

2.1.6 Neural Networks

We have already introduced the history of neural networks but let us dive into the specifics of what is a neural network.

Neural networks main components are neurons and weights. A neuron is a node that receives one or multiple inputs and outputs a single value. The weights are the parameters that multiply the inputs of a neuron and are generally learnable parameters. We can summarize this mathematically as follows:

$$y = \phi\left(\sum_{j=1}^{N} \omega_j x_j\right) \tag{2.37}$$

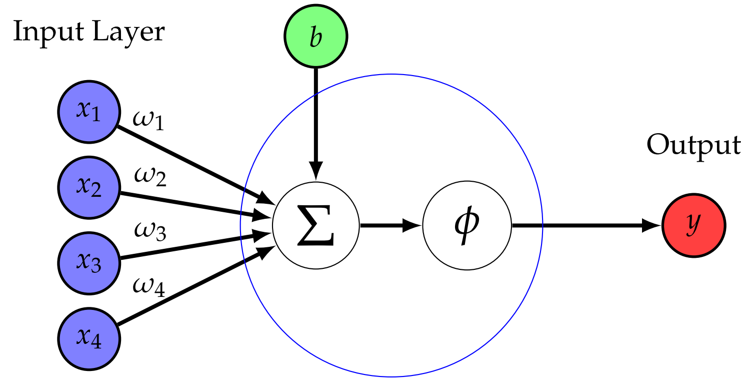

FIGURE 2.1: Single artifical neuron. The input layer contains the input vector $\mathbf{x}$ and is composed by each of its components x_i for $i \in \{1,2,3,4\}$. The weights of each of the inputs w_i are drawn in the edges that all go to the neuron and more specifically to an aggregator that in this case it is a summation of all the products $w_i x_i$ and additionally the bias of the neuron b. The result of this sum is then the input of the activation function ϕ that outputs the final output of the neuron y.

where y corresponds to the output of neuron, x_j and ω_j correspond to the neuron input and its corresponding associated weight, N is the number of inputs of the neuron. Additionally ϕ is the activation function of the neuron. It introduces a non-linearity to the function, it would be linear otherwise. Fig. 2.1 shows a scheme of a single neuron with multiple inputs. Some of the most common activation functions are:

$$\begin{array}{cr} \text{Sigmoid:} & \phi(z) = \frac{1}{1+e^{-z}} \\ \text{Hyperbolic tangent:} & \phi(z) = \tanh z \\ \text{Rectifier Linear Unit (ReLU):} & \phi(z) = \begin{cases} x & \text{if} \quad z > 0 \\ 0 & \text{if} \quad z < 0 \end{cases} \end{array} \tag{2.38}$$

Single-layer perceptron

If we have several neurons instead of just one eq. (2.38) becomes:

$$y_i = \phi_i(\sum_{j=1}^{N} \omega_{ij} x_j) \tag{2.39}$$

where y_i corresponds to the output of the i-th neuron, x_j and ω_{ij} correspond to the j-th input and its weight for the i-th neuron respectively. This system is formed by a determined number of neurons that all receive the same inputs but they do not have necessarily the same weights associated to them. These neurons are independent from each other and each of them produces an output by themselves. Under such conditions we can claim that these neurons form a layer and since there is only one layer this neural network is called a single-layer perceptron. Fig. 2.2 displays a single-layer perceptron. The single-layer perceptron is the simplest form of feed forward neural network which is a type of neural network where the connections between neurons do not form cycles.

To last main element necessary is a loss or error function that will be minimized by performing a gradient descent of sorts on the trainable weights of the neural network. The gradient is computed using backpropagation. The loss must be a function

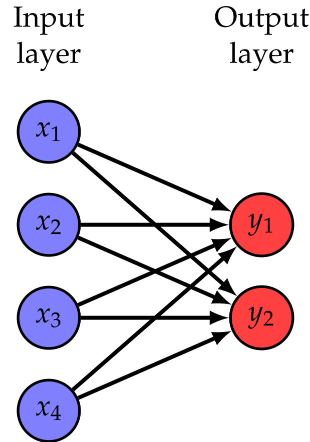

FIGURE 2.2: Single-layer perceptron. It has 2 neurons that make the Output layer. The input layer of size 4 corresponds to the input vector dimensions. The weights are the incoming arrows to the neurons, the bias and the possible activation function are implicit in each of the neurons.

of the weights somehow because they are the variables with respect to which the gradient is going to be performed. Normally the loss is a function of the output of the system and often it has a so-called Tikhonov regularization term that is a direct function of the weights.

Regularization is a group of adjustments that one can do to the model or the training process to avoid overfitting of the data and improve generalization.

Multi-layer Perceptron (MLP)

A multi-layer perceptron is composed by more than one layer of neurons. The output of the n-th layer of neurons serves as the input for the $(n+1)$-th layer of neurons. In turn, this means that there is at least one hidden layer. A hidden layer is one layer the output of which is not the final output. In the single-layer perceptron case we had one layer that received the input and produced the output that was simultaneously the output of the model. The smallest would have an additional layer which, for example, could receive the output of the other one and produce the final result and thus, the first one would become a hidden layer. Fig. 2.3 shows an example of MLP with 4 inputs, two hidden layers of size 8 and an output layer of size 2.

Other than that, the non-linearity introduced is essential. Let us take Fig. 2.3 as an example. Let us consider that ϕ is the identity for all neurons. We can write the calculations this way:

$$\begin{aligned} \mathbf{y}_1 &= \phi(\mathbf{W_1 x}) \\ \mathbf{y}_2 &= \phi(\mathbf{W_2 y_1}) \\ \mathbf{y}_f &= \phi(\mathbf{W_3 y_2}) \end{aligned} \tag{2.40}$$

If we apply the selected ϕ we can now simplify eq. (2.40) as following:

$$\mathbf{y_f} = \mathbf{W_3 W_2 W_1 x} = \mathbf{W x} \tag{2.41}$$

Rendering most of the parameters redundant. And this model would not be able to reproduce non-linear functions. In fact, it would not longer be a multi-layer

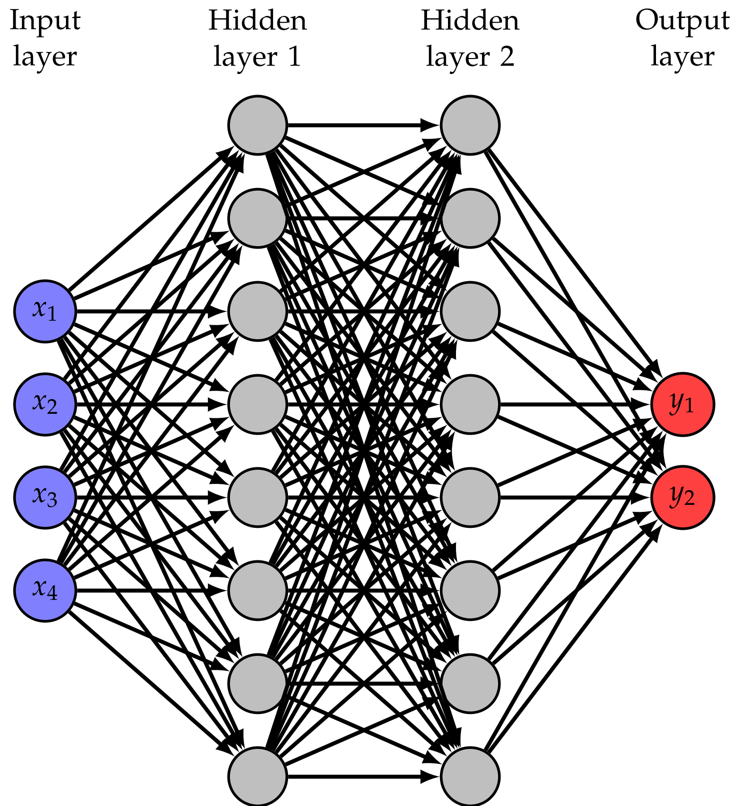

FIGURE 2.3: Multi-layer perceptron. It has 2 neurons that make the Output layer. The input layer of size 4 corresponds to the input vector dimensions. The weights are the incoming arrows to the neurons, the bias and the possible activation function are implicit in each of the neurons.

perceptron but a single-layer perceptron since the parameters from the hidden layers collapse as long with the output layer parameters.

Chapter 3

Related Work

3.1 Graph Neural Networks

Graph Neural Networks (GNNs) (Scarselli et al., 2009) are a family of models that capture the graph structure of a system. They were developed to unify the tackling of node-based problems and graph-based problems. Graph-based problems are those that look for a function that takes a graph as an input and returns a real vector of values. Some graph-based problems look to classify a graph for example or obtain a property or set of properties of the graph. On the other hand, node-based applications aim to obtain a target function that takes as arguments a graph and a node and returns again a vector of values describing that node in the context of the graph. For instance, a node-based problem would be a node classification task. GNNs unify these problems.

GNNs try to preserve as much of the topological information of the graph before processing the data. Some supervised neural networks that already did this are Recursive Neural Networks (RNNs) or Markov chains. The issue with these two pre-existing models is that they are only suitable for very specific type of graphs. For example, RNNs work with cyclic directed graphs. GNNs do not have this sort of constraint and can practically be used for any sort of graph.

In a GNN, nodes try to encode elements and the edges encode their relationship or interaction. We will use the notation used in (Scarselli et al., 2009). Let us consider a graph $\mathcal{G}$ that is a pair $(\mathbf{N}, \mathbf{E})$ where $\mathbf{N}$ is the set of nodes and $\mathbf{E}$ is the set of edges. Let us first define some components: the neighbourhood of a node n as $\mathrm{ne}[n]$, the incoming edges to a node n as $\mathrm{co}[n]$ [1], the label or labels that describe the node n as vector $\mathbf{l}_n$ such that $l_n \in \mathbb{R}^N$ and the label or labels that describe the interaction/relationship between nodes n_i, n_j as $l_{(n_i,n_j)}$ such that $\mathbf{l}_{(n_i,n_j)} \in \mathbb{R}^E$.

The main idea of this model is the representation of the state of a node n, $\mathbf{x_n} \in \mathbb{R}^s$, by using the information of the node itself and its surroundings (be it neighbouring nodes or incoming edges). This state is then used to obtain the target for the node/graph $\mathbf{o}$.

Let $\mathbf{f}$ be the local transition function that in the most generic case it depends on the node, but for most of the cases it is node invariant. And let $\mathbf{g}$ be the local output function. $\mathbf{x}_n$ and $\mathbf{o}_n$ are mathematically defined as follows:

$$\begin{aligned} \mathbf{x}_n &= \mathbf{f}_\omega(\mathbf{l}_n, \mathbf{l}_{\mathrm{co}[n]}, \mathbf{x}_{\mathrm{ne}[n]}, \mathbf{l}_{\mathrm{ne}[n]}) \\ \mathbf{o}_n &= \mathbf{g}_\omega(\mathbf{l}_n, \mathbf{x}_n) \end{aligned} \tag{3.1}$$

[1] The original article treats the case of an undirected graph and $\mathrm{co}[n]$ contains all the edges as a consequence.

where $\mathbf{l}_{\text{co}[n]}$ are the labels of the incoming edges of the node n, $\mathbf{x}_{\text{ne}[n]}$ are the states of the neighbouring nodes of node n and $\mathbf{l}_{\text{ne}[n]}$ are the labels of the neighbouring nodes of node n. Both, $\mathbf{f}$ and $\mathbf{g}$ are functions with parameters ω that can be learned.

Let us assume a supervised learning framework $\mathcal{L}$ that contains various graphs (independent from one another, disconnected) that belong to the set $\mathcal{G}$ such that:

$$\mathcal{L} = \{(\mathbf{G}_i, n_{i,j}, \mathbf{t}_{i,j}), \mathbf{G}_i = (\mathbf{N}_i, \mathbf{E}_i) \in \mathcal{G}; n_{i,j} \in \mathbf{N}_i; \mathbf{t}_{i,j} \in \mathbb{R}^m, 1 \leq i \leq p, 1 \leq j \leq q_i\} \tag{3.2}$$

where $n_{i,j}$ corresponds to the j-th node of $\mathbf{N}_i$ and $\mathbf{t}_{i,j}$ is the target for this node.

Banach's fixed point theorem proves the uniqueness of the solution of eq. (3.1) and it implements the Jacobi iterative method for solving nonlinear equations. Therefore the solution can be reached using the following equations iteratively:

$$\begin{aligned} \mathbf{x}_n(t+1) &= \mathbf{f}_\omega(\mathbf{l}_n, \mathbf{l}_{\text{co}[n]}, \mathbf{x}_{\text{ne}[n]}(t), \mathbf{l}_{\text{ne}[n]}) \\ \mathbf{o}_n(t) &= \mathbf{g}_\omega(\mathbf{l}_n, \mathbf{x}_n(t)) \end{aligned} \tag{3.3}$$

If we set a Mean Squared Error (MSE) loss of the learning framework (3.2) to measure the distance between the outputs $\mathbf{o}_{i,j}$ of the different nodes and the targets $\mathbf{t}_{i,j}$.

$$MSE_\omega = \sum_{i=1}^{p} \sum_{j=1}^{q_i} (\mathbf{t}_{i,j} - \mathbf{o}_{i,j})^2 \tag{3.4}$$

The outputs depend on the parameters ω that are learned according to the data used.

3.2 Graph Networks

There are a lot of types of graph neural networks as explained in (Zhou et al., 2019) that have improved the performance of the original graph neural networks by becoming specific for the graph type, by using different changes in the propagation step (convolution, gate mechanism, attention mechanism and skip connection). On top of that there are different general frameworks that try to integrate some of this methods. The most notable ones are the message passing neural network (MPNN) that unifies several graph neural network and graph convolutional network approaches, and the non-local neural network (NLNN) that unifies self-attention methods.

In this context, Graph Networks (GNs) (Battaglia et al., 2018) is a framework that unifies MPNN and NLNN and can reproduce almost any known graph neural network model. In this generalization a graph G is defined as $G = (\mathbf{u}, V, E)$ that is a tuple of three class of elements: the global attributes of the graph $\mathbf{u}$, the set of vertices or nodes $V = \{\mathbf{v}_i\}_{i=1:N^v}$ (where N^v is the number of nodes of G) and the the set of edges $E = \{(\mathbf{e}_k, r_k, s_k)\}_{k=1:N^e}$ (where N^e is the number of edges of G). Edges are a tuple of three elements: the edge features $\mathbf{e}_k$, the index of the receiver node r_k and the index of the sender node s_k.

The main unit of computation in this framework is the GN block (Fig. 3.1). Its main objective in its most general form is to update the values of the attributes of the three elements that compose the graph $\mathbf{u}_i, \mathbf{v}_j, \mathbf{e}_k$. It starts by updating the edge attributes, then the node attributes and lastly the global attributes.

First of all, the edges are updated using an edge update function ϕ^e that depends only on the attributes of the edge, its two nodes' attributes and the global attributes.

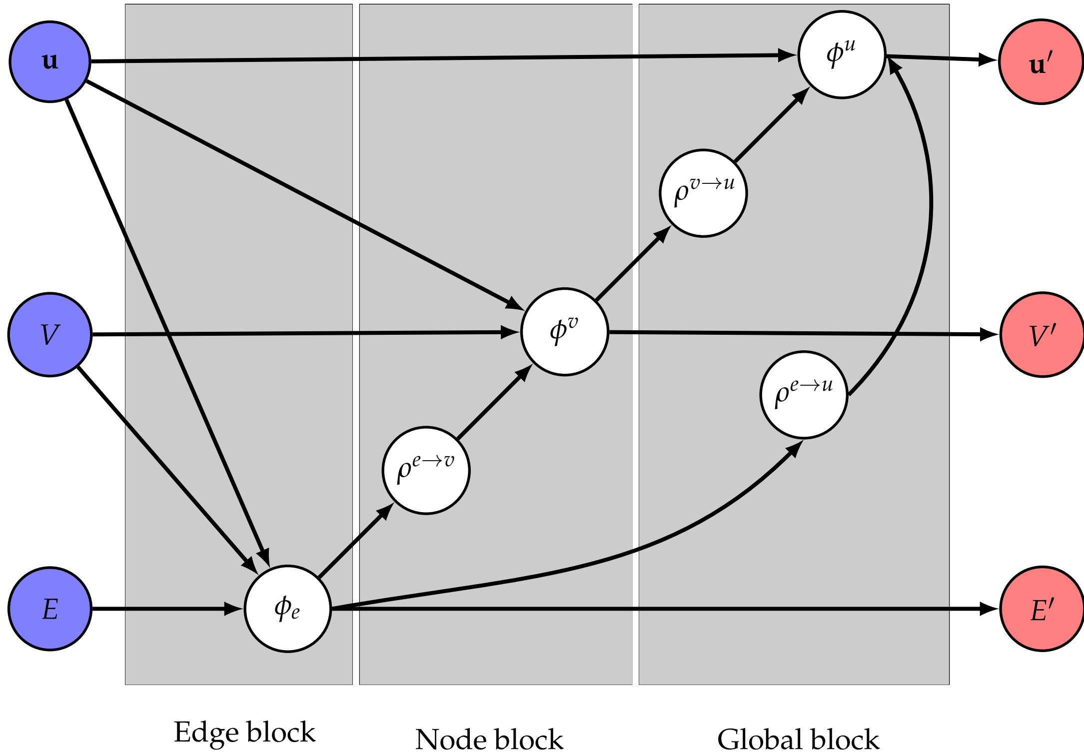

FIGURE 3.1: GN block scheme. Schematic representation of the Graph Network block with its different components. On the left there are the initial values of the components of the graph (global attributes **u**, node states V and edge states E). On the right there are the updated states of this components. And in the box in the middle there are the functions (white nodes) that make the update of states possible. The incoming edges of a function node are its inputs and the outgoing edges are its output. If a component of the graph (blue node) is directly connected to the same graph component but updated (red node) it means the value of the component has not been updated.

$$\mathbf{e}'_k = \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}) \tag{3.5}$$

Once all the edges are updated and in order to update the nodes, all the edges for which node $\mathbf{v}_i$ is the receiver are aggregated by means of an aggregation function $\rho^{e\to v}$.

$$\bar{\mathbf{e}}'_i = \rho^{e\to v}(E'_i) \tag{3.6}$$

where $E'_i\{(\mathbf{e}_k, i, s_k)\}_{k=1:N^e} \in E$ that is the subset of E that contains all the edges that have the node i as a receiver node. Now the nodes are updated using a node update function ϕ^v that depends on the recently calculated aggregated edge, the old attributes of the node itself and the global attributes.

$$\mathbf{v}'_i = \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}) \tag{3.7}$$

In order to update the global attributes, the aggregation of nodes and edges separetely it is necessary and so the introduction of two global attribute aggregators is deemed necessary.

$$\bar{\mathbf{e}}' = \rho^{e \to u}(E') \tag{3.8}$$

$$\bar{\mathbf{v}}' = \rho^{v \to u}(V) \tag{3.9}$$

Now with the new aggregated variables $\bar{\mathbf{v}}', \bar{\mathbf{e}}'$ along with the old global attributes as inputs, we define a new function ϕ^u that outputs the modified global attributes.

$$\mathbf{u}' = \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}) \tag{3.10}$$

The GN block allows for a lot of flexibility. In the original paper some examples are provided of how to obtain different graph neural networks by choosing certain functions. And by reducing for example the number of input variables. All in all, this framework can deal with graph-focused, node-focused and edge-focused applications.

3.3 Interaction Networks

Interaction Networks (INs) (Battaglia et al., 2016) is a model inspired by graph neural networks specifically designed to deal with complex systems in a simulation-like way. It separates the learning of the interactions and the object dynamics but at the same time they are linked. Even though it was conceived before GNs, we are going to use the notation and the GN formulation to describe its structure.

It is worth noting that there are two kind of Interaction Networks: node-focused and graph-focused. The first type when applied to complex systems deduce the future states of the agents or particles, while the second type extract information from the system like the energy.

The computation scheme for the node-based case is displayed in Fig. 3.2. The equations corresponding to it are the following. First of all the edge update function ϕ^e:

$$\mathbf{e}'_k = \mathrm{NN}_e([\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}]) \tag{3.11}$$

Where NN_e is a neural network (single-layer perceptron or multi-layer perceptron) that does not depend on the global attributes $\mathbf{u}$ and takes as input a concatenation of the other vectors (represented with square brackets).

Then the edge aggregation done by $\rho^{e \to v}$ is a summation:

$$\bar{\mathbf{e}}'_i = \sum_{j=1}^{|E'_i|} \bar{\mathbf{e}}_j \tag{3.12}$$

where $|E'_i|$ is the cardinality of the set of incoming edges of node i. The node update is done as follows:

$$\mathbf{v}'_i = \mathrm{NN}_v([\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}]) \tag{3.13}$$

Where NN_v is a neural network that takes as input a concatenation of the vectors inside of the square brackets.

The computation scheme for the graph-focused case is displayed in Fig. 3.3. It is identical to the node-focused case but it incorporates the update of the global attributes. But first it computes the aggregation of node attributes:

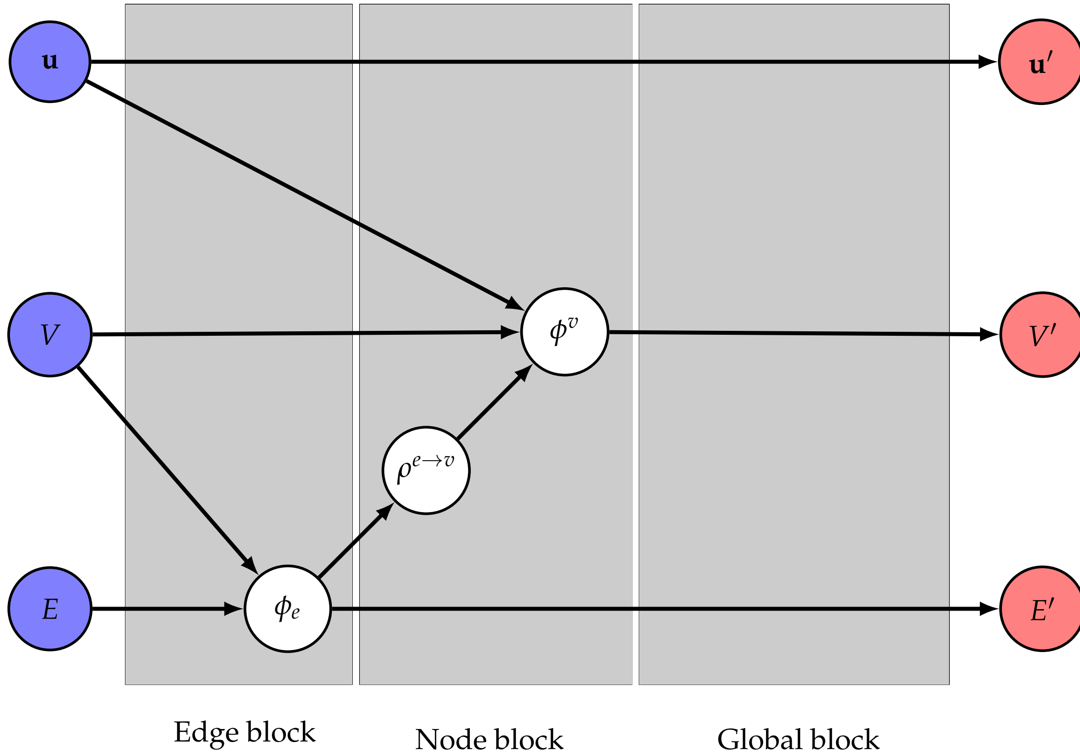

FIGURE 3.2: Node-focused Interaction network scheme. Schematic representation of the Interaction Network node-focused variant block with its different components. On the left there are the initial values of the components of the graph (global attributes **u**, node states V and edge states E). On the right there are the updated states of this components. And in the box in the middle there are the functions (white nodes) that make the update of states possible. The incoming edges of a function node are its inputs and the outgoing edges are its output. If a component of the graph (blue node) is directly connected to the same graph component but updated (red node) it means the value of the component has not been updated.

$$\bar{\mathbf{v}}' = \sum_{i=1}^{|\mathbf{V}|} \mathbf{v}_i \tag{3.14}$$

where $|\mathbf{V}|$ is the cardinality of the node set and therefore the total number of nodes of G. Again in this case the aggregator $\rho^{v\to u}$ is a summation.

Let us now introduce the Interaction Network notation because it will allow us to understand better the implementation later. In this notation the vertices are called objects and denoted by $O = \{o_j\}_{j=1,...,N_O}$ such that total number is N_O. The edges are called relations and denoted by $R = \{< i, j, r_k >_k\}_{k=1,...,N_R}$ where each relation is a triplet of two objects and a relation vector with $i \neq j$ and $i, j \in \{1, ..., N_O\}$ and the total number of relations is N_R. The external effects that could be thought as global effects (attributes) that affect all vertices are denoted by X. As a consequence a graph is $G = (O, R, X)$[2] that is analogous to $G = (V, E, \mathbf{u})$. One can define the node-foscused (or object-focused) Interaction Network model as:

[2]Even tho in (Battaglia et al., 2016) they only define a graph as a tuple $G = (O, R)$

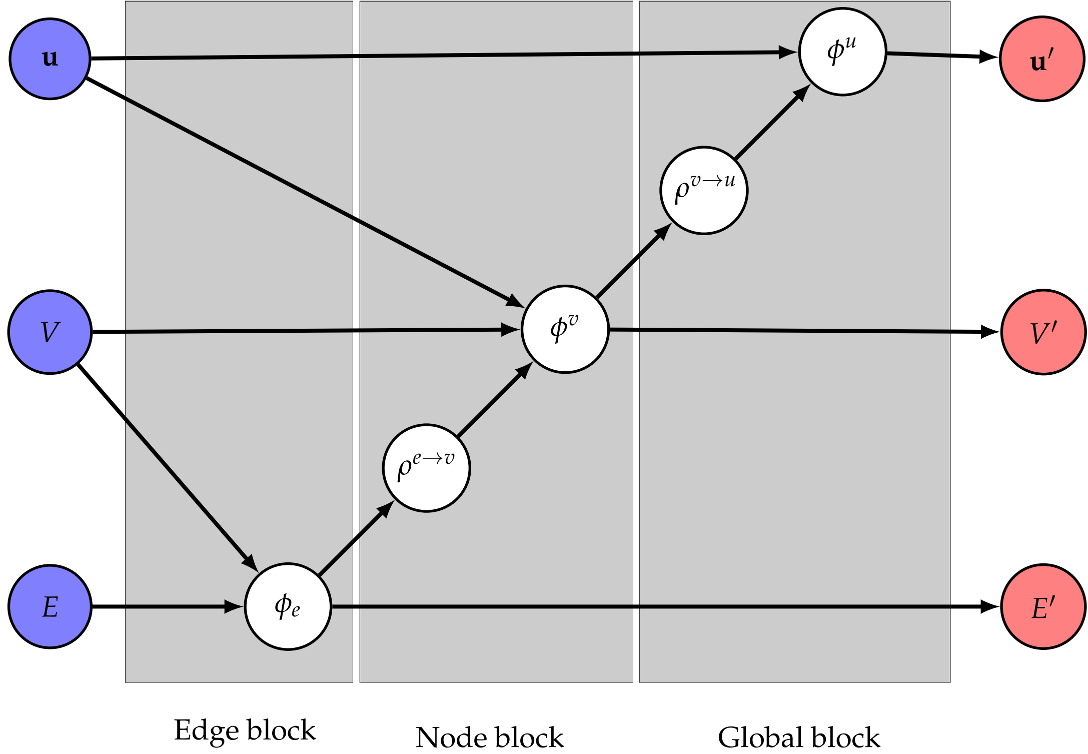

FIGURE 3.3: Graph-focused Interaction network scheme. Schematic representation of the Interaction Network graph-focused variant block with its different components. On the left there are the initial values of the components of the graph (global attributes **u**, node states V and edge states E). On the right there are the updated states of this components. And in the box in the middle there are the functions (white nodes) that make the update of state possible. The incoming edges of a function node are its inputs and the outgoing edges are its output. If a component of the graph (blue node) is directly connected to the same graph component but updated (red node) it means the value of the component has not been updated. different components.

$$\text{IN}(G) = \phi_O(a(G, X, \phi_R(m(G)))) \tag{3.15}$$

$$\begin{array}{lllllll} m(G) & = B & = \{b_k\}_{k=1,\ldots,N_R} & a(G,X,E) & = C & = \{c_j\}_{j=1,\ldots,N_O} \\ f_r(b_k) & = e_k & & f_O(c_j) & = p_j & \\ \phi_R(B) & = E & = \{e_k\}_{k=1,\ldots,N_R} & \phi_O(C) & = P & = \{p_j\}_{j=1,\ldots,N_O} \end{array} \tag{3.16}$$

where m is the marshalling function responsible of rearranging the elements of the graph (objects and relations) into interaction terms such that each interaction term $b_k =< o_i, o_j, r_k >\in B$ contains information on the receiver and the sender objects, and the relation itself. The relational model ϕ_R predicts the effects $\{e_k\}_{k=1,\ldots,N_R}$ of the interaction between each pair and so it is the same as the edge update function shown in Eq. (3.5). Then the aggregation function a, combines the predicted effects on an object and it merges it with the global effects (attributes) (X) and the object attributes O to produce C. Using the former (C) as input the object model ϕ_O

(analogous to the node update function shown in Eq. (3.7)) predicts the new object attributes or some variable related to them (P) with which to obtain O_{t+1}.

And the graph-focused IN is defined as :

$$\text{IN}(G) = \phi_A(g(\phi_O(a(G, X, \phi_R(m(G)))))) \tag{3.17}$$

$$\begin{array}{llllllllll} m(G) & = B & = \{b_k\}_{k=1,\dots,N_R} & a(G,X,E) & = C & = \{c_j\}_{j=1,\dots,N_O} & g(P) & = \bar{P} \\ f_r(b_k) & = e_k & & f_O(c_j) & = p_j & & & \\ \phi_R(B) & = E & = \{e_k\}_{k=1,\dots,N_R} & \phi_O(C) & = P & = \{p_j\}_{j=1,\dots,N_O} & \phi_A(\bar{P}) & = q \end{array} \tag{3.18}$$

It has the same components as the node-based but it incorporates two new functions. Firstly, the aggregator function g that merges all object predicted attributes $\{p_j\}_{j=1,\dots,N_O}$ into $(\bar{P})$. Secondly, an abstraction model ϕ_A (analogous to Eq. (3.10)) that produces a single output q.

3.3.1 Implementation

In order to implement this it is important to define some matrices to see how exactly is the data going to be handled. Firstly, the node attribute information is stored in the matrix O that has dimensions $D_S \times N_O$ where D_S is the dimension of the attributes (their number of components) and N_O is the number of objects. The information about the relations between nodes is stored in two binary matrices R_s, R_s which correspond to the source nodes and sink nodes respectively. Both have dimension $N_O \times N_R$ where N_R is the number of relations or edges. For every edge they have a 1 in the position of the node that is its source or sink depending on the matrix. Therefore every edge and object are indexed. Additionally, R_a that is the matrix with the edge or relation attributes has dimensions $D_R \times N_R$, where D_R is the dimension of said attributes. And lastly, X is a $D_X \times N_O$ matrix with the external effect that each particle experiments.

The marshalling function has the aim to prepare all the attributes regarding a relation therefore it takes as inputs

$$m(G) = m(O, R_r, R_s, R_a) = [OR_r; OR_s; R_a] = B \tag{3.19}$$

where "[ ]" is the concatenation operation and ";" means horizontal concatenation. As a result B is a $(2D_O + D_R) \times N_R$ dimension matrix. B serves as input to the relational MLP with input size $(2D_O + D_R)$ and output size D_E that produces the matrix E with dimension $D_E \times N_R$ that learns new attributes for each relation.

The aggregator function is responsible of collecting all the information to update a node.

$$a(G, X, E) = a(O, R_r, X, E) = [O; X; \bar{E}] = C \tag{3.20}$$

where $\bar{E} = ER_r^T$. C has a dimensions $(D_S + DX + D_E) \times N_O$.

The C matrix serves as input for the object MLP that has an input size $D_S + DX + D_E$ and an output size D_P and as a consequence the resulting matrix P has dimensions $D_P \times N_O$. That is all for the node focused version.

The graph focused version though has an additonal aggregtor function.

$$g(P) = \sum_k p_k = \bar{P} \tag{3.21}$$

where p_k is the k-th column of P. The result $\bar{P}$ is a D_P-variable vector that is the input of the abstract MLP that was an input size D_P and output size D_A. As a result, the output of the MLP, q, has D_A dimensions.

Chapter 4

CREATING TRAJECTORIES

The main objective is to learn and infer the motion of collective of particles or agents by using Data Driven Learning techniques to do so. On top of that the data will not be extracted from the real world but obtained by applying physical rules on ideal physical systems.

4.1 `Scene` class

In order to deal with trajectories we have created the `Scene` class. It creates, handles and stores trajectories. We define a trajectory as a list of successive states where the temporal distance between successive states is a time step Δt. The collection of temporal states are stored as the ground truth for the models that will later use it to learn the rules of this systems. We will use a time step $\Delta t = 1$ in order to simplify the equations to calculate the trajectories. However the class `Scene` supports the use of an arbitrary value for the time step Δt.

Each state contains the necessary information to describe the kinematics of the system at that instant of time: position $\mathbf{r}$, velocity $\mathbf{v}$ and acceleration $\mathbf{a}$.

We want to make a series of experiments under controlled conditions and for that reason we will define a 2×2 box where the agents or particles will move. We have chosen a 2-dimensional system in order to be able to produce visually helpful representations of the system and for the sake of simplicity. It is not a hard box and therefore they may leave it but it means that we will try and create trajectories that happen in their entirety inside of the box boundaries.

The initialisation of the data is done in different ways. Accelerations are initialised to 0. On the other hand, positions and velocities are initialized using a initialisation function that is an input when initialising the scene. We have implemented two. The first initialisation function `random_init` generates 2D vectors with random directions and with random norm with an upper bound the value of which can be set by the user. The other initialisation function `polygon_init` generates vectors corresponding to a regular polygon of as many sides as bodies the system has that is centered in the origin of coordinates.

The class can generate trajectories from any rule or set of equations as long as they are contained within an `interaction` function that takes the positions and velocities of the particles/agents as arguments and returns their updated accelerations which are used to deduce the new velocities and then the new positions. The ones we have implemented are :

Zero interaction

This is the lack of interaction between the particles or agents. If there is no interaction the accelerations returned are 0 using Newton's second law (2.1.3). And without

acceleration the trajectories are straight lines. Eq. (2.17) to update the position becomes:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v_n} \tag{4.1}$$

where we have used $\Delta t = 1$.

Central Force

These are a family of forces that are exerted in the direction of the $\mathbf{r}$ vector joining the interacting particles. All the details have already been presented in 2.1.4. We have implemented a subfamily of the family of central forces that are characterised with this equation:

$$\mathbf{F}_{ij} = \alpha \frac{\hat{\mathbf{r}}_{ij}}{r_{ij}^{\beta}} \tag{4.2}$$

where $\mathbf{F}_{ij}$ is the force exerted by the j-th particle on the i-th particle, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the vector, $\hat{\mathbf{r}}_{ij}$ is the unitary vector with the same direction, $r_{ij} = |\mathbf{r}_{ij}|$ is the norm of $\mathbf{r}_{ij}$ and α and β are parameters that dictate the effective range of interaction. For example the gravitational force would have $\alpha = -Gm_1m_2$ and $\beta = 2$ while Coulomb's law would $\alpha = \frac{q_1q_2}{4\pi\epsilon_0}$ and $\beta = 2$. In that sense our force doesn't have a "charge" or we could consider that all particles have the same.

By Newton's third law (2.1.3) the force experienced by the j-th particle exerted by the i-th particle $\mathbf{F}_{ji}$ is just the same as $\mathbf{F}_{ij}$ but with different sign (antiparallel) which speeds up the computations.

Using again Eq. (2.26) and applying for the case of a multiple-particle system, it becomes:

$$\mathbf{F}_i = \sum_{j \neq i} \mathbf{F}_{ij} = \mathbf{a}_i \tag{4.3}$$

Which reads as the net force experienced by the i-th particle is the result of the sum of the forces that the rest of the particles exert on it and it results in a acceleration. For the sake of simplicity, the function implemented only will create and handle particles of mass $m = 1$ unit.

Once $\mathbf{a}$ is obtained from the interaction function the class integrates the equations of motion. In this case it is a second order ODE and using Euler method the updating functions are:

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_n \tag{4.4}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}_{n+1} \tag{4.5}$$

Vicsek model

In the case of the Vicsek model(2.1.5), all agents are treated equally following the simplest version. Eq. (2.33) becomes:

$$\mathbf{r}_i^{n+1} = \mathbf{r}_i^n + v\mathbf{s}_i^n \tag{4.6}$$

when using $\Delta t = 1$. The rest of the equations are the same. The velocity update could be expressed as:

$$\mathbf{v}_{n+1} = v\mathbf{s}_n \tag{4.7}$$

Since the function must return the acceleration instead of the new velocity state we can express the acceleration as:

$$\mathbf{a}_n = \mathbf{v}_{n+1} - \mathbf{v}_n \tag{4.8}$$

where normally the left-hand side would be divided by the time interval Δt but it will be 1 for the reasons showed previously.

4.2 `MultiScene` Class

The `MultiScene` class creates and stores a list of `Scene` objects so as to create more complete datasets given that our intention is not to learn one single trajectory but to generalize.

Similarly, it can make predictions on this scenes and compute the mean squared error (MSE) by combining the MSE for each scene with the appropriate weight.

Chapter 5

LEARNING TRAJECTORIES

The objective is to learn to predict the trajectories of systems of particles/agents. For instance, take a flock of birds as an example of a system. We can record it and with some software or by hand, label each bird and follow their motion. To do so, at each frame of the video we need to identify the position of the bird using system of coordinates.

Let's move to theoretical particles or agents. Position alone at each frame would be enough in normal conditions to develop a strategy to process this data such that it can be learned by some Machine Learning model. In our case we have the advantage of producing the data which means we can have all the data we need, in terms of features but also in terms of entries or observations. Nevertheless, it introduces at least a degree of complexity to the process. Therefore we will include the velocity as a feature of these agents.

The reason to include velocity is because it makes things easier to get good results but also a physical argument: classical mechanics differential equations of motion are second order ODEs in general. In order to solve a second order ODE one needs initial conditions on position and its first time derivative (velocity). However, velocity and even acceleration could be approximated by using the finite difference method like the Euler method. This way the velocity at time step $\mathbf{v_n}$ could be calculated using the position at two successive time steps as follows:

$$\mathbf{v}_n = \frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{\Delta t} \tag{5.1}$$

where Δt is the time interval between frames. Using the same approach we can get the equation for the acceleration $\mathbf{a}_n$:

$$\mathbf{a}_n = \frac{\mathbf{v}_{n+1} - \mathbf{v}_n}{\Delta t} = \frac{\mathbf{x}_{n+2} - 2\mathbf{x}_{n+1} + \mathbf{x}_n}{\Delta t} \tag{5.2}$$

5.1 Model

We have already set the information that the model will use. Now we need to discuss the architecture of the model. That is, we need to discuss how that information will be processed and used to obtain the target which we also have to set.

The target of the model are the variables that it will output as a prediction. We are dealing with trajectories and the essence at each time step are the positions of the particles/agents. Yet outputting velocity or acceleration is not a bad idea either due to them being its derivatives and by integrating them in a suitable way we could obtain the positions. As a result we will use the mean squared error (MSE) to track the error of prediction of our model. It is important to take into account that the value of the error for a model that outputs velocities is not comparable to the value

of the error of a model that outputs accelerations or directly the position. Since, as we stated, the ultimate goal is to learn to predict trajectories and we will use the position to compare the results, we have chosen the MSE on the positions predicted (directly or through one of its derivatives) as the main metric.

The `Scene` class will receive the output of the model and integrate the equations of motion if needed to completely calculate the new state (positions and velocities) or states since it is capable of predicting whole trajectories from an initial state. This model can predict complete trajectories just by using as input the its own output over and over for the amount of time steps desired. Some articles[1] call this a multistep rollout prediction.

This approach has the limitation that its prediction is limited by the time step of the training which has to be constant. Therefore one could not strictly predict what happens in a time inferior to the specific time interval. Yet by using interpolation it could be approximated for example. In fact one can generalize and claim that the model cannot predict what happens in an increment of time that is not a multiple of the time step given that one can predict multiple successive steps. Furthermore, the accuracy in a multistep rollout prediction is also limited, it is lower bounded specifically, such that in general the best accuracy is obtained for predictions of a single step and the accuracy decreases with the amount of steps.

We will use the python library TensorFlow [2] to implement all our models.

5.1.1 Naive model

We can already formulate a naive model based on the considerations we made. One of the most simplest one can think of is a neural network that takes as input all the positions and velocities of the agents/particles in a single one-dimensional vector at one instant and returns the updated positions and velocities in a single-one dimensional vector.

We have implemented a neural network that will do the task of naively learning and inferring on the desired systems and we have called that class `Naive`. It will be used as a baseline to compare the other models that will be tested. One can tune the size of the hidden layers and the quantity of these so that it can even be a single-layer perceptron. We have chosen that the neural network outputs will be the updated states of the particle/agent the attributes of which occupy the first positions in the input vector. Therefore the network has input size $N_O \times 4$ and output size 2. Notice how the network depends on the number of objects and as a consequence is not flexible.

It is important to remark here that this approach is already atomist in the sense that it differentiates between agents/particles and considers them the fundamental part that constitutes the system. Therefore we can say it has some bias and it is not completely naive. The model must infer on each of the elements of the system with the same weights. A completely naive neural network model would output in a single vector the states of the agents/particles.

5.1.2 Interaction Network

One of the objectives of this thesis is to implement an Interaction Network, train it and test it. The reason for this is that it is a form of graph neural network (GNN)

[1] (Battaglia et al., 2016)

[2] https://www.tensorflow.org/

developed specially for the treatment of physical systems with interaction. This approach associates the elements of the system with the elements of a directed graph. In this sense we will constantly switch between the Graph Network notation and the Interaction Network one.

Each agent/particle is a node of the graph. The edges are directed such that the source node is the one that exerts an effect and the sink node is the receiver of that effect. That effect could be a force for example. And due to its nature and concretely, to Newton's third law, there should be an antiparallel edge.

We have not just copied the IN but we have imposed some new conditions on our own model. We have programmed a family of models that use the Interaction Network as a starting point but that include slight changes and tweaks that are all inside of the `InteractionNetwork` class. Firstly, our model does not depend on absolute positions which means neither the relational update function nor the object update function use as input absolute positions but relative if any.

Moreover, the presence of the velocity in the input of the object update function that is a perceptron is optional. To understand the inclusion of this option we need to recall that the model can be set to output velocities or accelerations. And in the latter case, the case of acceleration, it makes sense from a physical point of view that the function that calculates the acceleration from the interactions (forces) with others does not depend on the velocity of said object nor its position. However if one is inferring the updated velocity it is logic that it must depend on the previous velocity state while still being independent of the position.

Furthermore, when dealing with n-body systems in the original IN paper (Battaglia et al., 2016) they only use connections between different particles and so the number of edges is $N_R = \binom{N_O}{2}$ while we introduce the possibility of considering loops (edges that have the same node as source and sink) making this way the number of edges $N_R = N_O^2$. Other articles like (Sanchez-Gonzalez et al., 2020) use a "connectivity radius" as a parameter to determine the edges which allow them to ignore non-local interactions.

As it is explained in (Battaglia et al., 2018) the edge aggregation function (using Graph Networks terminology) $\rho^{e\to v}$ does not have to be a sum, it can also be a mean, the minimum or maximum. In this context our model also has the option to use the mean additionally to the sum that the original IN already implements.

Lastly, we do not consider external effects neither we provide edge information and therefore the matrices X, R_a are null.

As it is an IN in essence, it uses neural networks as edge and node update functions. Similarly to the naive model one can tune the size of the hidden layers and the quantity of these sot that it can even be a single-layer perceptron. It also lets D_E, the dimension of learned edge attributes, be chosen by the user when creating the model.

Implementation

The list of changes enumerated with respect to the original IN is reflected in their implementation. Also since we are applying the model to the n-body problem we can already describe with more precision some of the variables and matrices that are going to be used.

The object matrix O contains the positions and the velocities. Let us consider the matrices R, V that contain the positions and velocities respectively and both have dimensions $N_O \times 2$ since we are working in a 2D space. This way:

$$O = [R^T; V^T] \tag{5.3}$$

where O has dimensions $4 \times N_O$.

The matrices R_s, R_r have dimensions $N_O \times N_R$ where N_R can be either $N_R = \binom{N_O}{2}$ or $N_R = N_O^2$ depending on the inclusion of loops.

The marshalling function is slightly different from the original Eq. (3.19). Since we want to make the system independent of absolute positions we introduce a tweak:

$$m(G) = m(O, R_r, R_s) = OR_r - OR_s = B \tag{5.4}$$

where B has dimensions $4 \times N_R$. B is then the input of the relational neural network with input size 4 and output size D_E with D_E being chosen by the user. As a result when applying the function to B we obtain the matrix E with dimensions $D_E \times N_R$.

The aggregator function includes the calculation of $\bar{E}$ that can be $\bar{E} = ER_r^T$ if the edge aggregation function is a summation or $\bar{E} = \frac{1}{N_O} ER_r^T$ if it is a mean. We must also take into account that we are skipping absolute positions of the objects but in this case they don not come in pairs and cannot be subtracted one from the other, which signifies getting rid of the position information. When it comes to the velocity it is optional to include it as we have explained before. This way the aggregator function can be expressed:

$$a(G, X, E) = a(V, R_r, E) = \begin{cases} [V^T; \bar{E}] & \text{if } V \text{ is included} \\ \bar{E} & \text{if } V \text{ is not included} \end{cases} = C \tag{5.5}$$

where C has dimensions $(D_E + 2) \times N_O$ if V is included or else $D_E \times N_O$. C is the input of the object neural network with input size $D_E + 2$ and output size 2 since we are predicting the 2D velocity or acceleration for an object. The result of of applying this function to C is the matrix P with dimensions $2 \times N_O$ that it becomes one of V or A by transposing it.

5.2 Experiments

We will study the performance of different models of the family of Interaction Networks when learning trajectories that correspond to different kind of interactions. As a general rule, we will work with a low number of particles to make training easier.

5.2.1 No Interaction

The first round of experiments to carry target the learning of systems with constant-velocity motions as a consequence of the lack of interaction.

Data

The systems to be studied will have 3 bodies. Furthermore, we will try to generate data that occupies the whole scene 2×2 box. For this reason we will create scenes with different parameters as shown in table 5.1. The main idea is to create 2-step trajectories to train inside of the box and that at least 10 steps from the validation

Nº scenes	N_O	Mode r	r	Mode v	v	time steps
200	3	Random	0.2	Random	0.4	2
200	3	Random	0.4	Random	0.3	2
200	3	Random	0.6	Random	0.2	2
200	3	Random	0.8	Random	0.1	2
200	3	Random	0.9	Random	0.2	2
100	3	Random	0.1	Random	0.09	100
100	3	Random	0.3	Random	0.05	100
100	3	Random	0.4	Random	0.04	100
100	3	Random	0.5	Random	0.05	100
100	3	Random	0.7	Random	0.03	100

TABLE 5.1: Scene parameters for the creation of the training and validation datasets of the zero interaction experiments. They are separated by one horizontal line, the training is on top and the validation on the bottom of the table. The N_O paramter is the number of bodies of the system. The parameters "r" and "v" are the maximum norm that the position and velocity vectors can achieve in their initialisation and their modes correspond to the ones introduced in section 4.1

trajectories are inside the box. The validation has 100 steps because we can also see how the error evolves with different number of successive predictions.

Models

We are going to use the simplest versions of the models for this experiment with no interaction.

That is two naive models without hidden layers: one that predicts velocities (naive 1) and another that predicts accelerations (naive 2). Therefore they are single-layer perceptrons with input size 12 and output size 2.

Similarly for the two IN models that do not have hidden layers in the two neural networks that each of them contains. The difference between both is that IN 1 outputs velocities while IN 2 outputs accelerations. Other than that both rely on creating adjacency matrices almost complete (without loops) and symmetrical. Both of them have relational and object functions with the same architecture (without hidden layers). By default both learn effects in 2 dimensions ($D_E = 2$), use regular addition between edges as the edge aggregating function $\rho^{e\rightarrow v}$ and use velocities in the input of the object function. These parameters can be visualized in table 5.2.

Results

The error of each of the models is shown in the table 5.3 and graphically at Fig. 5.1. The prediction on 3 of the scenes for 2 of the models compared to the ground truth are showed in Fig. 5.2.

5.2.2 Central force

The second round of experiments pretends to study systems with an interaction. More specifically, we will study systems in which the particles interact by means of a central force as the one in eq. (4.2). The central force of the system will not be inversely proportional in order to avoid the interaction using a large range and

Model	Params.	Output	Architerture	Loops	D_E	ϕ_R	ϕ_O	aggregator	V feature
naive 1	26	V	[12, 2]	-	-	-	-	-	-
naive 2	26	A	[12, 2]	-	-	-	-	-	-
IN 1	20	V	-	False	2	[4, 2]	[4, 2]	sum	True
IN 2	20	A	-	False	2	[4, 2]	[4, 2]	sum	True
IN 12	188	A	-	False	2	[4, 8, 8, 2]	[4, 8, 2]	mean	True

TABLE 5.2: Model hyperparameters used to learn systems without interaction. The parameters column presents the number of trainable parameters of a model. The layer sizes of the naive modules are under the architecture column while each of the relation function and object function architectures are under ϕ_R, ϕ_O columns respectively. The loops column indicates if the model contemplates edges with the same node as source and sink. D_E refers to the dimensionality of the edge attributes that are learned. The aggregator is the function that aggregates the edge attributes. The last column indicates if the velocity of an object is used as a feature or input in the object model.

Model	MSE 1	MSE 5	MSE 10	MSE 20
naive 1	1.04e-12	2.00e-10	2.21e-09	2.22e-08
naive 2	1.07e-12	2.07e-10	2.29e-09	2.32e-08
IN 1	5.18e-09	1.10e-06	1.38e-05	1.79e-04
IN 2	3.47e-13	7.90e-11	1.16e-09	2.33e-08
IN 12	1.73e-07	4.02e-05	5.64e-04	9.04e-03

TABLE 5.3: Mean squared error for different length multistep rollout predictions using different models that try to learn systems without interaction.

singularities if two agents are too close. Therefore they will have the parameter $\beta = 0$.

Data

The systems to be studied will again have 3 bodies. We want that the trajectories happen in the box and that they have several steps relatively small to have more precision and for this reason we need that the interaction force is not really big neither the velocities; at the same time we want to see the bending of the trajectories due to the effect and therefore we have chosen the parameter $\alpha = 1\text{e-}4$. Notice how it is positive and therefore it determines a repelling force. For this reason we are interested in the particles positions being initiated close to the center. In this case we have chosen that both kind of scenes will have the same number of time steps:100. We will be training the system with 500 scenes and using 100 scenes for validation. The parameters for the creation of this datasets are shown in table 5.4.

Models

We will use a variety of models that have some variation with respect to a base one. Their features are summarised in table 5.5. The base one (IN 4) is an Interaction Network that outputs accelerations, does not allow loops in the adjacency matrix, has a relational function composed of two 8-neuron hidden layers and an object function with one 8-neuron hidden layer. It uses regular addition as edge aggregating function and includes the velocities in the input of the object function.

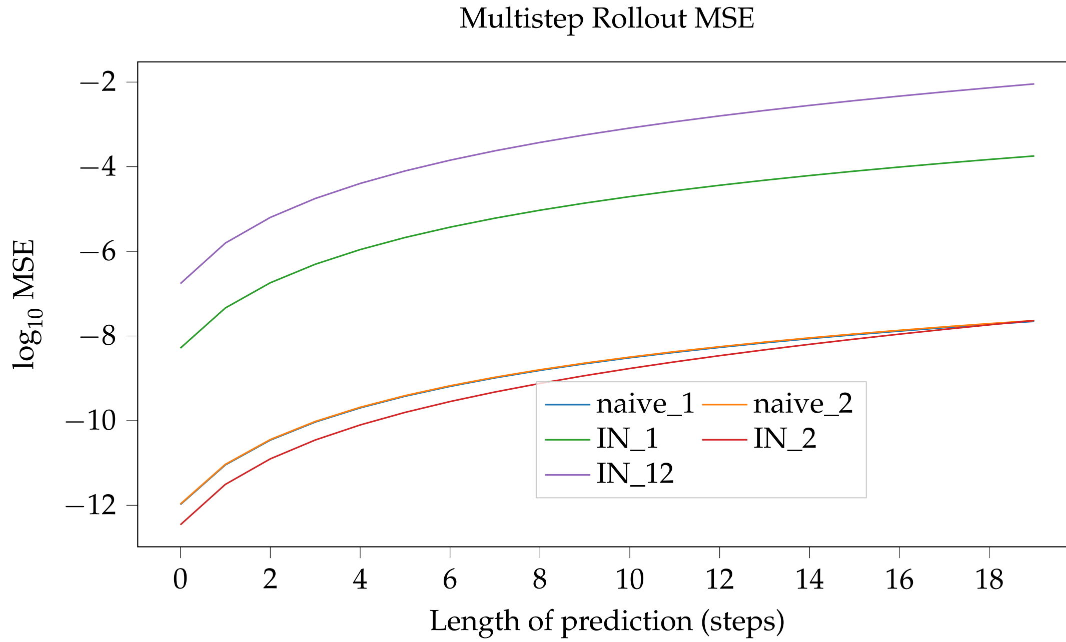

FIGURE 5.1: Logarithm in base 10 of the mean squared error (MSE) as function of the number of steps of the trajectory predicted for different models that try to learn a system without interaction. The curve corresponding to the model 'naive 1' and the curve corresponding to the model 'naive 2' overlap.

1. The model IN 3 has the same structure but outputs velocity instead of acceleration.

2. The model IN 5 does not include the velocities in the input of the object function.

3. The model IN 7 uses larger hidden layers in the relational function (two 16-neuron hidden layers). It does not include the velocities in the input of the object function.

4. The model IN 8 implements instead a larger hidden layer in the object function (16-neuron layer instead of a 8-neuron layer). It does not include the velocities in the input of the object function.

Nº scenes	N_O	Mode r	r	Mode v	v	time steps
500	3	Random	7.5e-1	Random	8e-3	100
100	3	Random	7.5e-1	Random	8e-3	100

TABLE 5.4: Scene parameters for the creation of the training and validation datasets of the central force interaction. They are separated by one horizontal line, the training is on top and the validation on the bottom of the table. The N_O parameter is the number of bodies of the system. The parameters "r" and "v" are the maximum norm that the position and velocity vectors can achieve in their initialisation and their modes correspond to the ones introduced in section 4.1

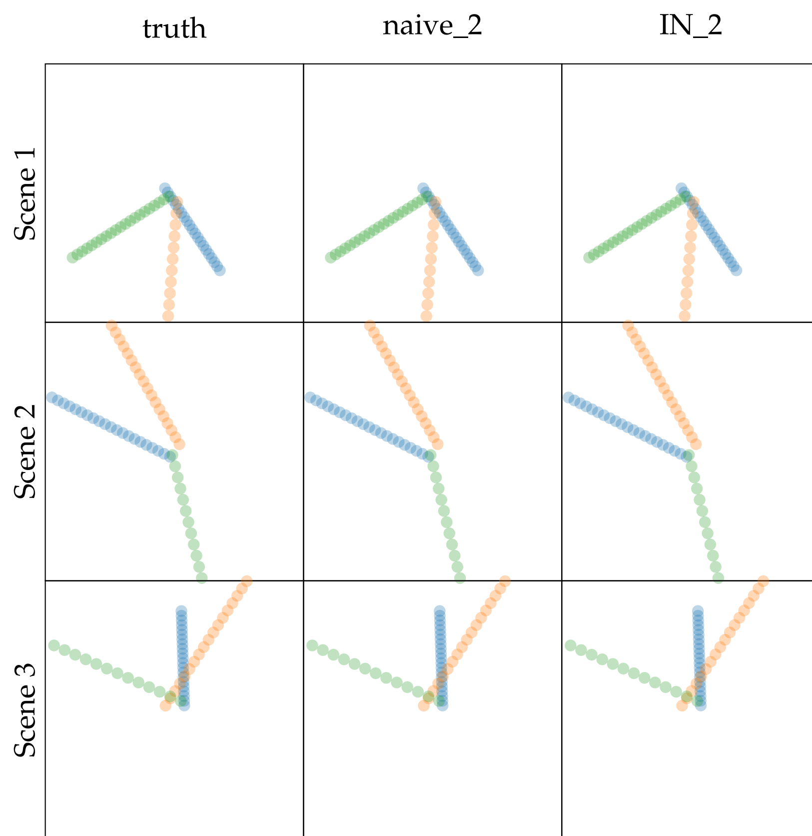

FIGURE 5.2: No interaction systems (3 scenes) prediction using two models against the ground truth.

Model	Params.	Output	Architerture	Loops	D_E	ϕ_R	ϕ_O	aggregator	V feature
naive 3	182	A	[12, 12, 2]	-	-	-	-	-	-
IN 3	188	V	-	False	2	[4, 8, 8, 2]	[4, 8, 2]	sum	True
IN 4	188	A	-	False	2	[4, 8, 8, 2]	[4, 8, 2]	sum	True
IN 5	172	A	-	False	2	[4, 8, 8, 2]	[2, 8, 2]	sum	False
IN 7	428	A	-	False	2	[4, 16, 16, 2]	[2, 8, 2]	sum	False
IN 8	212	A	-	False	2	[4, 8, 8, 2]	[2, 16, 2]	sum	False
IN 10	206	A	-	False	4	[4, 8, 8, 4]	[4, 8, 2]	sum	False
IN 13	155	A	-	False	1	[4, 8, 8, 1]	[1, 8, 2]	sum	False

TABLE 5.5: Model hyperparameters used to learn systems with a central force interaction. The parameters column presents the number of trainable parameters of a model. The layer sizes of the naive modules are under the architecture column while each of the relation function and object function architectures are under ϕ_R, ϕ_O columns respectively. The loops column indicates if the model contemplates edges with the same node as source and sink. D_E refers to the dimensionality of the edge attributes that are learned. The aggregator is the function that aggregates the edge attributes. The last column indicates if the velocity of an object is used as a feature or input in the object model.

Model	MSE 1	MSE 5	MSE 10	MSE 20
naive 3	3.60e-05	3.71e-03	1.81e-01	2.25e+04
IN 3	9.91e-06	2.50e-04	1.03e-03	4.65e-03
IN 4	1.40e-08	3.13e-06	4.19e-05	6.05e-04
IN 5	8.84e-09	1.99e-06	2.66e-05	3.85e-04
IN 7	1.44e-08	3.24e-06	4.35e-05	6.31e-04
IN 8	1.06e-08	2.37e-06	3.18e-05	4.60e-04
IN 10	2.26e-08	5.07e-06	6.80e-05	9.87e-04
IN 13	1.48e-08	3.33e-06	4.48e-05	6.50e-04

TABLE 5.6: Mean squared error for different length multistep rollout predictions using different models that try to learn systems with a central force.

5. The model IN 10 uses a higher edge attribute (effect) dimensionality ($D_E = 4$). It does not include the velocities in the input of the object function.
6. The model IN 13 uses a lower edge attribute (effect) dimensionality ($D_E = 1$). It does not include the velocities in the input of the object function.

Additionally we will train and use another naive model (naive 3) with a 12-neuron hidden layer so that it has roughly the same parameters as the base model (IN 4).

Results

The error of each of the models is shown in the table 5.6 and graphically at Fig. 5.3. The prediction on 3 of the scenes for 2 of the models compared to the ground truth are showed in Fig. 5.4.

5.2.3 Vicsek Model

The last kind of trajectories that we will study are the ones caused by an interaction modelled by the Vicsek model. As it is been explained before this is a complex

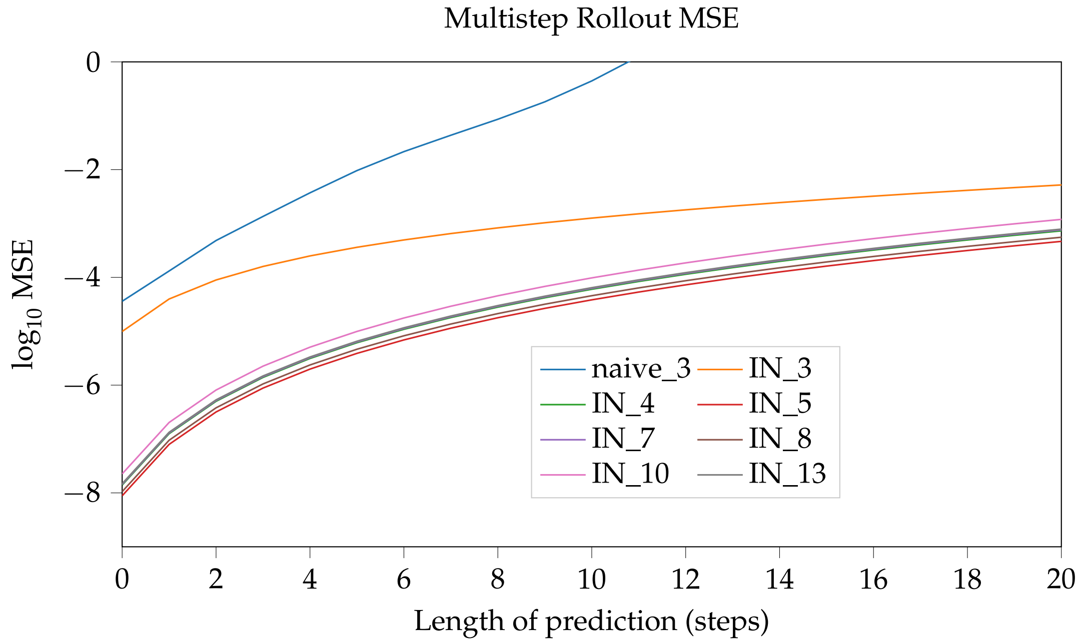

FIGURE 5.3: Logarithm in base 10 of the mean squared error (MSE) as function of the number of steps of the trajectory predicted for different models that try to learn a system with a central force interaction.

interaction and therefore we do not expect the best of results in this section. We will try a variant of the Vicsek model with effective infinite radius. This means a system where all agents see each other. And additionally we will try a Vicsek model with a finite radius.

Data Experiment 1

We will use 5-body systems in order to observe different synchronisations that would not be possible with less than 4 bodies. The first datasets have a radius of interaction $radius = 1e10$ which is 10 orders of magnitude greater than the box and therefore it can be considered infinite. The training has 500 scenes each of them has 2 prediction time steps. This because the in the first time step the system reaches synchronicity due to its nature and the second step the system continues with the synchronicity but just updates the position. Beyond this point everything will be just regular constant-velocity motion. The validation has 100 scenes each of them with 10 time steps.

The second experiment has a finite radius or interaction range: $radius = 2.5e\text{-}1$. The training set uses 500 scenes with 100 time steps each while the validation set has 100 scenes of 100 time steps each.

The parameters used to create the scenes are listed in table 5.7.

Models Experiment 1

For the first experiment with an infinite radius even though it is Vicsek model, we will use a simple Interaction Network (IN 14) without hidden layers in the object function nor the relational function. It outputs velocities, uses loops, uses the mean

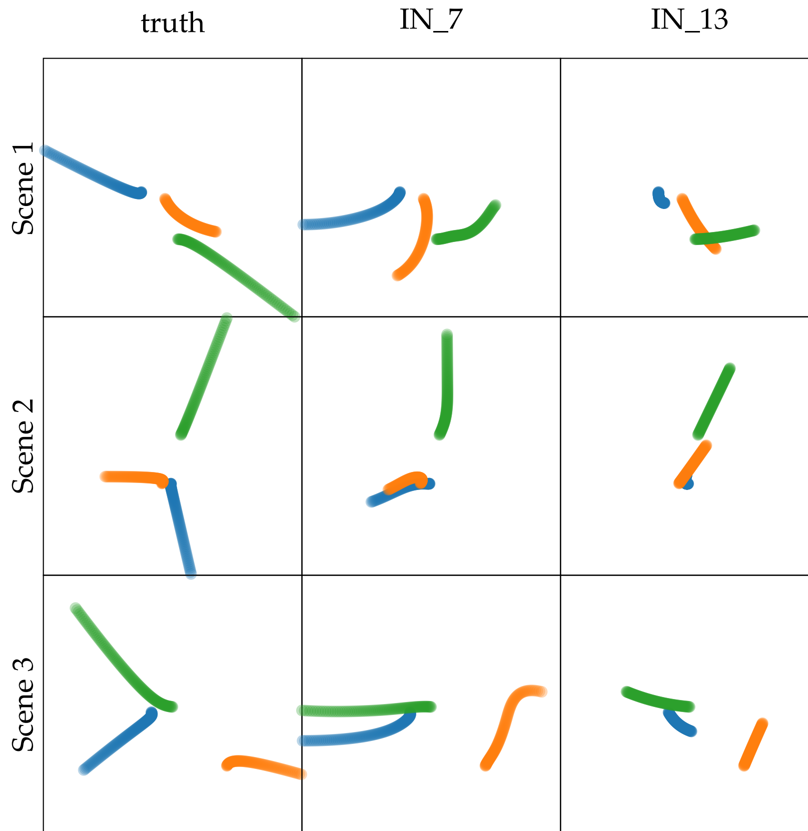

FIGURE 5.4: Central force systems (3 scenes) prediction using two models against the ground truth.

Experiment	N° scenes	N_O	Mode r	r	Mode v	v	time steps
1	500	3	Random	6e-1	Random	5e-1	2
2	500	3	Random	6e-1	Random	1e-2	100
1	100	3	Random	6e-1	Random	5e-1	10
2	500	3	Random	6e-1	Random	1e-2	100

TABLE 5.7: Scene parameters for the creation of the training and validation datasets of the Vicsek model. Experiment 1 corresponds to system with an "infinit" radius while experiment 2 corresponds to systems with radius with $radius = 2.5e - 1$. They are separated by one horizontal line, the training is on top and the validation on the bottom of the table. The N_O parameter is the number of bodies of the system. The parameters "r" and "v" are the maximum norm that the position and velocity vectors can achieve in their initialisation and their modes correspond to the ones introduced in section 4.1

Model	Params.	Output	Architerture	Loops	D_E	ϕ_R	ϕ_O	aggregator	V feature
naive 4	462	A	[20, 20, 2]	-	-	-	-	-	-
IN 6	188	V	-	False	2	[4, 8, 8, 2]	[4, 8, 2]	sum	True
IN 9	172	V	-	True	2	[4, 8, 8, 2]	[2, 8, 2]	mean	False
IN 11	188	A	-	False	2	[4, 8, 8, 2]	[4, 8, 2]	sum	True
IN 12	188	A	-	False	2	[4, 8, 8, 2]	[4, 8, 2]	mean	True
IN 14	16	V	-	True	2	[4, 2]	[2, 2]	mean	False

TABLE 5.8: Model hyperparameters used to learn systems that follows a Vicsek model. The parameters column presents the number of trainable parameters of a model. The layer sizes of the naive modules are under the architecture column while each of the relation function and object function architectures are under ϕ_R, ϕ_O columns respectively. The loops column indicates if the model contemplates edges with the same node as source and sink. D_E refers to the dimensionality of the edge attributes that are learned. The aggregator is the function that aggregates the edge attributes. The last column indicates if the velocity of an object is used as a feature or input in the object model.

Model	MSE 1	MSE 5	MSE 10	MSE 20
IN 14	1.19e-11	2.68e-09	3.61e-08	5.26e-07

TABLE 5.9: Mean squared error for different length multistep rollout predictions using different models that try to learn systems that follow the Vicsek model with $radius = 1e10$ (first experiment).

as aggregating function and does not use the velocity as a feature for the object function.

The parameteres of these models are summarized in table 5.8.

Results Experiment 1

For the first experiment: The error of each of the models is shown in the table 5.9 and graphically at Fig. 5.5. The prediction on 3 of the scenes for 2 of the models compared to the ground truth are showed in Fig. 5.6.

Data Experiment 2

We will use 5-body systems in order to observe different synchronisations that would not be possible with less than 4 bodies.The second experiment has a finite radius or interaction range: $radius = 2.5e - 1$. The training set uses 500 scenes with 100 time steps each while the validation set has 100 scenes of 100 time steps each.

The parameters used to create the scenes are listed in table 5.7.

Models Experiment 2

For the second experiment, which is a way more complex system, we will try a variety of models:

1. The model IN 6 with the same hyperparameters as the base model of the experiments with systems governed by central (IN 4) but it outputs velocities instead.

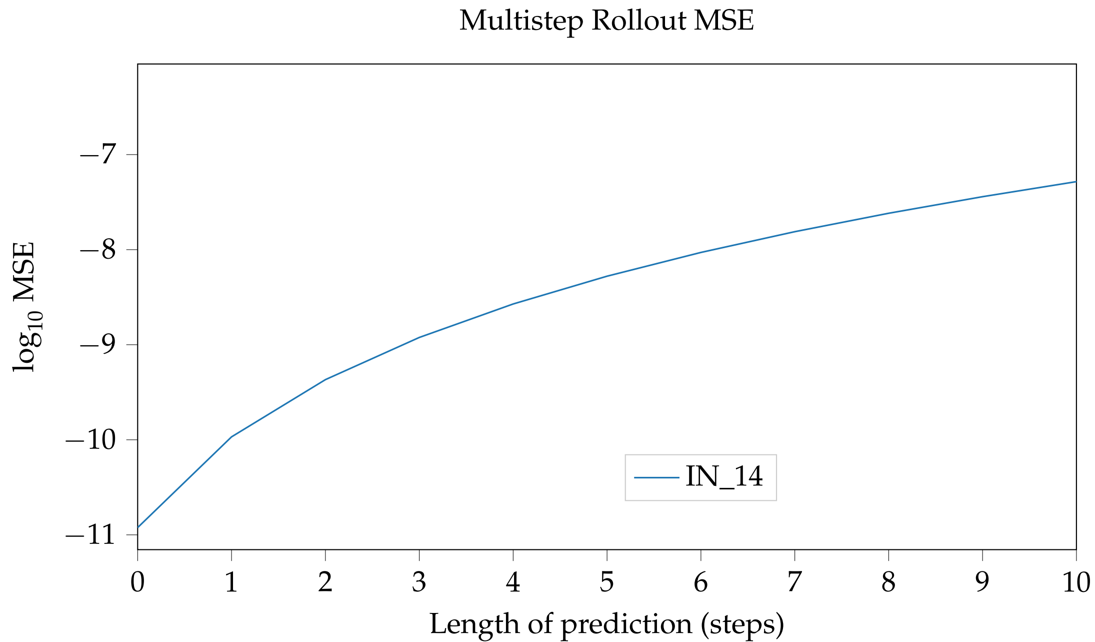

FIGURE 5.5: Logarithm in base 10 of the mean squared error (MSE) as function of the number of steps of the trajectory predicted for different models that try to learn a system governed by the Vicsek model with $radius = 1e10$ (experiment 1).

2. The model IN 9 averages the edge attributes instead of adding them and it does not include the velocity in the inputs of the object function. Additionally it considers loops as edges and it outputs velocities.

3. The model IN 11 has the same hyperparameters as the base model in the central force experiments (IN 4).

4. The model IN 12 is like the model IN 11 but it uses the mean as aggregating function.

5. The model IN 14 has been trained for experiment 1.

For this second experiment we have also trained a naive model (naive 4) with a 20-neuron hidden layer to match the the input size that is 20 due to the systems having 5 agents.

The parameteres of these models are summarized in table 5.8.

Results Experiment 2

For the second experiment: The error of each of the models is shown in the table 5.10 and graphically at Fig. 5.7. The prediction on 3 of the scenes for 2 of the models compared to the ground truth are showed in Fig. 5.8.

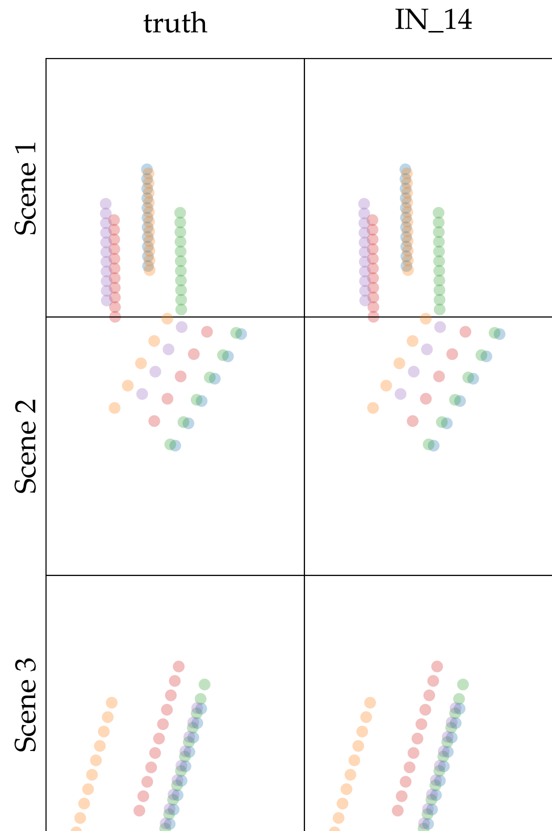

FIGURE 5.6: Vicsek model systems (3 scenes) prediction using two models against the ground truth in experiment 1.

Model	MSE 1	MSE 5	MSE 10	MSE 20
naive 4	7.99e-05	1.23e-02	2.22e-01	2.93e+03
IN 6	2.60e-05	6.22e-04	2.32e-03	8.46e-03
IN 9	2.75e-05	6.28e-04	2.33e-03	8.47e-03
IN 11	2.22e-05	5.52e-04	2.24e-03	8.99e-03
IN 12	2.23e-05	5.59e-04	2.29e-03	9.22e-03
IN 14	4.56e-02	4.67e-02	4.94e-02	5.87e-02

TABLE 5.10: Mean squared error for different length multistep roll-out predictions using different models that try to learn systems that follow the Vicsek model with $radius = 2.5e - 1$ (second experiment).

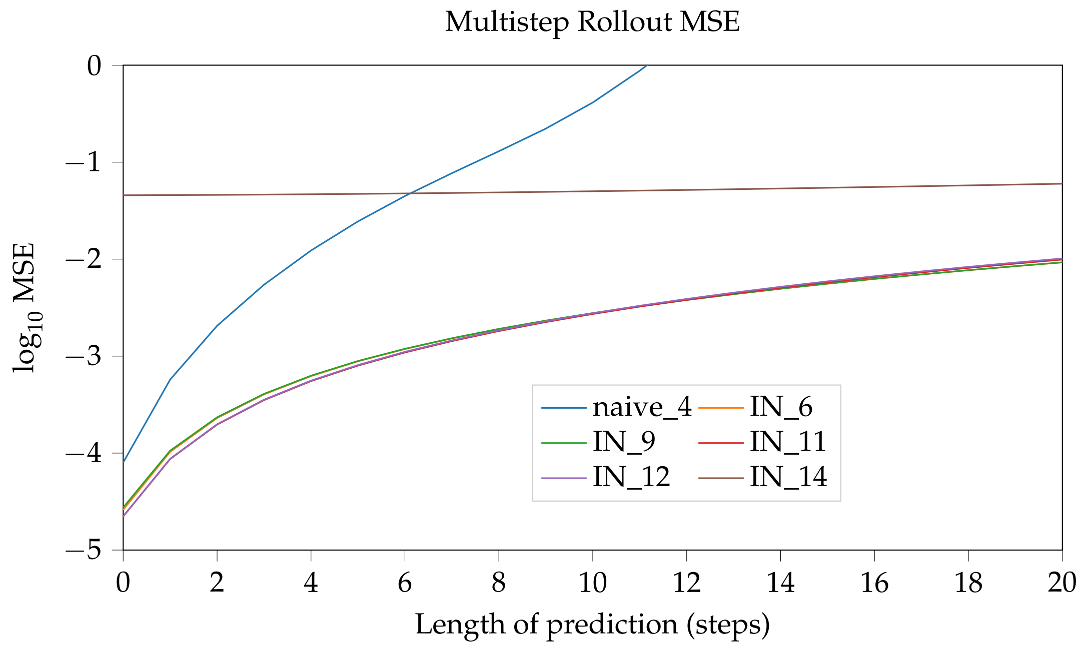

FIGURE 5.7: Logarithm in base 10 of the mean squared error (MSE) as function of the number of steps of the trajectory predicted for different models that try to learn a system governed by the Vicsek model with *radius* $= 2.5e - 1$ (experiment 2).

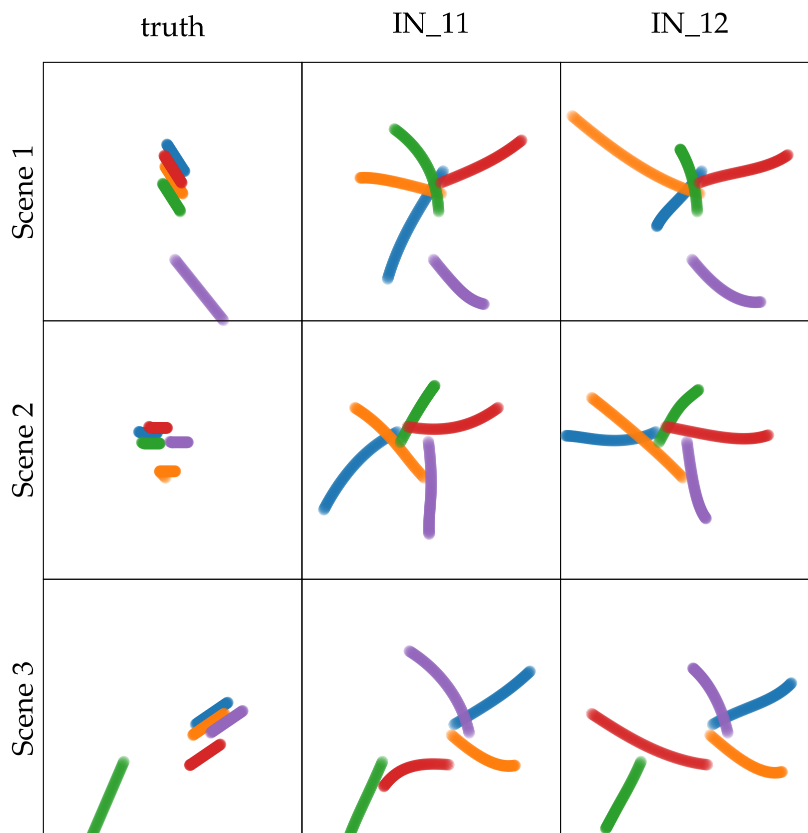

FIGURE 5.8: Vicsek model systems (3 scenes) prediction using two models against the ground truth in experiment 1

Chapter 6

Discussion of the experimental results

6.1 No interaction

The results involving systems without interaction are the most satisfactory ones. All the models tried specifically tailored for this got good results numerically and the eye test confirms it. The best one-step prediction MSE (as it can be seen in table 5.3) corresponding to the model IN 2 reaches the order of 1e-13 while the naive models are one order above and the model IN 1 is 3 orders of magnitude greater than IN 2.

The evolution of the mean squared error has the same shape for all of them as we can see in Fig. 5.1. That is the error grows exponentially with the same coefficient for all of them for a number of steps greater than 10 because the curve in the logarithmic graph apparently gets a constant derivative.

It is worth noting that both naive models manage similar results. On the other hand, It is interesting to notice how the Interaction Network that outputs accelerations (IN 2) instead of velocities is significantly better than the one outputting velocities (IN 1) given a similar amount of training epochs used. With more training in the velocity outputting models probably we would obtain similar results to the former.

6.2 Central force

As we stated previously to learn this problem we have used different models that are slight variations of a base one (IN 4). This base one has reached a one-step MSE of the order of 1e-8 (as it can be see in table 5.6) which is far from the best MSE obtained for the systems without interaction. We notice that the best result is yielded by model IN 5 which is the same as IN 4 but it does not include the velocities in the input of the object function. Using a smaller input implies a smaller number of input neurons in the object neural network and as a consequence fewer trainable parameters are needed. This means a smaller space where to find the solution and therefore potentially less minimums to visit and since we have trained them with a similar number of epochs, it makes sense that the model with less parameters but containing the relevant information has reached a better solution.

The model IN 7 implements two 16-neuron hidden layers in the relational function as opposed to the two 8-neuron hidden layers that the base model IN 4 implements. The results are very close but marginally worse probably because of the argument used before. More neurons implies more trainable parameters and a bigger search space and generally needs a larger number of epochs in training to reach the solution.

However, this phenomenon is not observed with model IN 8 which uses a 16-neuron hidden layer in the object relation. But since it is only a layer there is such a difference in the number of trainable parameters. IN 8 has a slightly worse MSE but of the same order of magnitude.

The models IN 10 and IN 13 explore the use of a different dimension for the learned edge attributes that are the output of the edge attribute update function. They implement respectively $D_E = 4$ and $D_E = 1$ while the base model IN 4 has $D_E = 2$. All of them three have close results and of the same magnitude and the first two do not manage to improve the results obtained with IN 4. It is worth noting that IN 13 has a really close value to IN 4 which is surprising because one would expect the edge attributes that are the information shared between particles to need two dimensions since the interaction is a force in a 2D space. All this with the addition of the results graphically (Fig. 5.3) leads to believe that the models fail to capture the nature of the interaction.

All these commented results are really close between each other and more or less of the same order of magnitude. Nevertheless, the model IN 3 with a one-step MSE of the order 1e-6 which is 3 orders of magnitude worse than the best. This probably has to do with the fact that IN 3 is the only one outputting velocities and as we know the equations that must be solved to obtain the equation of motion of this system are second order and therefore they operate at the acceleration level.

6.3 Vicsek Model

The results of the first experiment are incredibly good reaching a one-step MSE of the order 1e-11 as shown in table 5.9 with the model IN 14. The graphical results shown in Fig. 5.5 confirm it.

The results of the second experiment are fully different. Here the radius parameter of the Vicsek model is finite. We tried to reuse the model IN 14 but the approach with a one-step MSE of the order 1e-2 (as shown in table 5.10) is far from acceptable.

The rest of the models used have all a one-step MSE of the order of 1e-5 which is not good enough. They all yield similar results despite being quite different when it comes to the hyperparameters.

The model IN 6 uses the same hyperparameters as IN 3 that was already used for the central force experiments (outputting velocities). In this case we expected it to be better because the Vicsek model equations operate at the level of the velocities.

After this model, we have tried the model (IN 9) that we thought was the most suitable for this system. This model is first tried that uses the mean as the edge aggregating function. Not only that but it considers loops and so an agent sees itself as a neighbour. This is precisely what the Vicsek model does. It averages the velocities of an agent and its neighbours with the same weight. So in this sense we expected that the message sent between nodes would be the velocities and so the mean would suit nicely to emulate the theory. Unfortunately for our disappointment, the results were really close to the model IN 6 tried just before this one.

The models IN 11 and IN 12 are an attempt to find a better solution by tuning the hyperparameters without success. We have chosen them to show their performance graphically and as it can be seen in Fig. 5.7. We can see the failure in trying to capture the system. It seems as if the effects are not strong enough even though there is a slight bending of the trajectories. For this reason we have decided to test one of them (IN 12) trained with this system to predict a system without interaction. The numerical results are in table 5.3. The one-step prediction MSE is of order of

magnitude 1e-7 which is not as good as the 1e-13, 1e-12 obtained with the dedicated models for that task but it is still 2 orders of magnitude smaller than it is for the task it has been trained for which is very telling.

To confirm the lack of meaningful learning the model naive 4 reaches a one-step prediction MSE fo the same order of magnitude. However, it is important to notice that the MSE for longer step predictions grows significantly faster in this case compared to the IN models which probably has to do with the fact that those models already capture the graph structure (although not the correct one) of the system.

Chapter 7

Conclusion and future directions

The objective of this thesis project was to learn trajectories of physical systems of agents or particles that interact in some way by using Interaction Networks with proposed modifications. We have designed three kinds of scenes which are in a sense boxes of a reality or compartmentalized parts of a reality where we have full control of the rules (interactions) governing the system. We have managed astounding results in systems without interaction but while a positive thing this was not our main goal.

The second type of system studied is a system governed by central forces that each particle exerts on each other. To tackle this problem we modified the Interaction Network primarily so that it outputted accelerations instead of velocities. This approach is not a novelty since recent articles like (Sanchez-Gonzalez et al., 2020) use it. Similarly to them our models do not use absolute positions as inputs and we even conceive the possibility of eliminating the velocity as an input in the object function.

The results obtained when trying to learn these systems are not as accurate as the previous ones but they are not disheartening. If we compare our training approach to that of the original Interaction Networks (Battaglia et al., 2016) we see that they used infinitely more graphs in their training dataset: 2000 scenes of 1000 time steps each which is equivalent to 2 million individual graphs. To that we have to add the fact that they are using a far larger number of trainable parameters: their relational function has 4 150-neuron hidden layers and their object function has one 100-neuron hidden layer. On top of that, they used a 150 dimensional representation for the edges (D_E). Thus if we had the time we could reach similar results or even better since the prediction of accelerations seems to have gained relevance in the field for a reason.

Lastly, the other model we wanted to learn from was the Vicsek model. It was our primary goal because it is something that has not been studied with an approach based on Interaction Networks yet. Additionally we initially wanted to infer parameters from the Vicsek model that generated the data like the radius or the angle of vision because Interaction Networks have the option of including a global function that updates the global attributes of the graph and allow one to obtain abstract quantities as these (we hypothesize) or physical magnitudes like the energy or momentum for instance.

Unfortunately the results were bad for a general Vicsek model with a finite radius. While it is true that for a effectively infinite radius of neighbourhood we could very accurately predict the future states, for finite radius the results were awful. We designed a Interaction Network that implements an edge aggregating update function that computes the mean of the edge attributes for a node instead of just their sum as in the original version and it takes into account loops (what we call self edges) in the adjacency matrix. We expected this variant to yield the best results but

we were not successful. One of the problems of our approach is that we are considering all the possible edges which means that when computing the mean of the edge attributes for each agent we are effectively dividing all nodes by the same constant and so this is practically the same as a regular sum. As we said in the discussion of the results, the message (the edges) that one node sent to others and itself should be the velocities and that is why loops makes sense. Since the averaging is wrong and we believe it can not be learned effectively by our architecture then it would be impossible to learn this kind of system even if we trained the model with 2 million graphs.

The success of this results was a necessary condition to even start to think about a graph-focused Interaction Network capable of deducing the radius of any Vicsek model system. As a consequence we have decided to abort that task.

7.1 Future directions

However, even if the overall results are not very accurate and of straight use and even if the tweaks and modifications have not succeeded we should not discard them.

We have presented Interaction Networks in the notation and formalism of Graph Networks (Battaglia et al., 2018) and that has already been useful to propose modifications like using the mean as an aggregating function. This is not a bad idea it is actually really useful, but it has been used wrongly for the reasons already stated. In order to use it correctly we need that in general the nodes can have a different number of neighbours. And the key to obtain that is having a finite radius of interaction in the model. But how can we choose one when in general it will not coincide with the one of the physical model (in this case the Vicsek model)? That is a question that we have tried to answer practically without success by letting the radius of connectivity be a trainable variable of the model as any of the weights in the neural networks are. This approach would yield the new state of the agents and the radius at the same time. This would be a really smart idea if it were not for the fact that the loss, and more concretely its gradient, does not depend on it; and for this reason it has not been included in the experiments.

Is there any hope though ? There is. The solution is to have a model that assigns a weight to each pair of agents (even to themselves). This assignation of weights can be set or more generally can be learned by means of the implementation of a neural network. Therefore given two agents (given their positions and velocities) we can set a function that outputs a single positive quantity. This would mean that that adjacency matrix and consequently the matrices R_S, R_r would no longer be binary and in general they would be dynamic since they would change according to the evolution of the node attributes.

The answer had been right under our noses all this time. Vecause in the article (Ha and Jeong, 2020) that we used as inspiration they approached this kind of system with Graph Attention Networks. We discarded it because we deemed it too complex at the time. The next step is thus the implementation of a node-focused Attention Interaction Network in order to be able to predict the trajectories of agents governed by a Vicsek model. After that we should aim to implement a graph-focused variant that will be trained to predict the radius of connectivity given a Vicsek-based system.

Bibliography

Battaglia, Peter W. et al. (2016). "Interaction Networks for Learning about Objects, Relations and Physics". In: arXiv: 1612.00222 [cs.AI].

Battaglia, Peter W. et al. (2018). "Relational inductive biases, deep learning, and graph networks". In: arXiv: 1806.01261 [cs.LG].

Ginelli, Francesco (2016). "The Physics of the Vicsek model". In: *The European Physical Journal Special Topics* 225.11-12, 2099–2117. ISSN: 1951-6401. DOI: 10.1140/epjst/e2016-60066-8. URL: http://dx.doi.org/10.1140/epjst/e2016-60066-8.

Ha, Seungwoong and H. Jeong (2020). "Towards Automated Statistical Physics : Data-driven Modeling of Complex Systems with Deep Learning". In: *ArXiv* abs/2001.02539.

Hinton, Geoffrey E et al. (2016). "A fast learning algorithm for deep belief nets". In: *Comptes Rendus Mécanique* 347.18. Neural computation, pp. 845 –855. ISSN: 0899-7667. DOI: https://doi.org/10.1162/neco.2006.18.7.1527. URL: https://pubmed.ncbi.nlm.nih.gov/16764513/.

Montáns, Francisco J. et al. (2019). "Data-driven modeling and learning in science and engineering". In: *Comptes Rendus Mécanique* 347.11. Data-Based Engineering Science and Technology, pp. 845 –855. ISSN: 1631-0721. DOI: https://doi.org/10.1016/j.crme.2019.11.009. URL: http://www.sciencedirect.com/science/article/pii/S1631072119301809.

Sanchez-Gonzalez, Alvaro et al. (2020). "Learning to Simulate Complex Physics with Graph Networks". In: arXiv: 2002.09405 [cs.LG].

Scarselli, F. et al. (2009). "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20.1, pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

Vicsek, Tamás et al. (1995). "Novel Type of Phase Transition in a System of Self-Driven Particles". In: *Physical Review Letters* 75.6, 1226–1229. ISSN: 1079-7114. DOI: 10.1103/physrevlett.75.1226. URL: http://dx.doi.org/10.1103/PhysRevLett.75.1226.

Zhou, Jie et al. (2019). "Graph Neural Networks: A Review of Methods and Applications". In: arXiv: 1812.08434 [cs.LG].